



The Urbi Software Development Kit

Version 2.1

Gostai

July 8, 2010
(Revision 2.1)



This documentation is updated regularly on the Gostai Web site both as a [PDF document](#), and as a [set of HTML pages](#). You may also want to look at the documentation of the latest versions.

Urbi SDK 2.1 This version.

- <http://www.gostai.com/downloads/urbi-sdk/2.1/doc>
- <http://www.gostai.com/downloads/urbi-sdk/2.1/doc/urbi-sdk.pdf>
- <http://www.gostai.com/downloads/urbi-sdk/2.1/doc/urbi-sdk.html>

Urbi SDK 2.1.x The latest version of the 2.1 family of versions.

- <http://www.gostai.com/downloads/urbi-sdk/2.1.x/doc>
- <http://www.gostai.com/downloads/urbi-sdk/2.1.x/doc/urbi-sdk.pdf>
- <http://www.gostai.com/downloads/urbi-sdk/2.1.x/doc/urbi-sdk.html>

Urbi SDK 2.x The latest version.

- <http://www.gostai.com/downloads/urbi-sdk/2.x/doc>
- <http://www.gostai.com/downloads/urbi-sdk/2.x/doc/urbi-sdk.pdf>
- <http://www.gostai.com/downloads/urbi-sdk/2.x/doc/urbi-sdk.html>

Chapter 1

Introduction

Urbi SDK is a fully-featured environment to orchestrate complex organizations of components. It relies on a middleware architecture that coordinates components named UObjects. It also features urbiscript, a scripting language that can be used to write orchestration programs.

1.1 Urbi and UObjects

Urbi makes the orchestration of independent, concurrent, components easier. It was first designed for robotics: it provides all the needed features to coordinate the execution of various components (actuators, sensors, software devices that provide features such as text-to-speech, face recognition and so forth). Languages such as C++ are well suited to program the local, low-level, handling of these hardware or software devices; indeed one needs efficiency, small memory footprint, and access to low-level hardware details. Yet, when it comes to component orchestration and coordination, in a word, when it comes to addressing concurrency, it can be tedious to use such languages.

Middleware infrastructures make possible to use remote components as if they were local, to allow concurrent execution, to make synchronous or asynchronous requests and so forth. The *UObject* C++ architecture provides exactly this: a common API that allows conforming components to be used seamlessly in highly concurrent settings. Components need not be designed with UObjects in mind, rather, UObjects are typically “shells” around “regular” components.

Components with an UObject interface are naturally supported by the urbiscript programming language. This provides a tremendous help: one can interact with these components (making queries, changing them, observing their state, monitoring various kinds of events and so forth), which provides a huge speed-up during development.

Finally, note that, although made with robots in mind, the UObject architecture is well suited to tame any heavily concurrent environment, such as video games or complex systems in general.

1.2 Urbi and urbiscript

urbiscript is a programming language primarily designed for robotics. It's a dynamic, prototype-based, object-oriented scripting language. It supports and emphasizes parallel and event-based programming, which are very popular paradigms in robotics, by providing core primitives and language constructs.

Its main features are:

- syntactically close to C++. If you know C or C++, you can easily write urbiscript programs.
- fully integrated with C++. You can bind C++ classes in urbiscript seamlessly. urbiscript is also integrated with many other languages such as Java, MatLab or Python.
- object-oriented. It supports encapsulation, inheritance and inclusion polymorphism. Dynamic dispatching is available through monomethods — just as C++, C# or Java.
- concurrent. It provides you with natural constructs to run and control high numbers of interacting concurrent tasks.
- event-based. Triggering events and reacting to them is absolutely straightforward.
- functional programming. Inspired by languages such as LISP or Caml, urbiscript features first class functions and pattern matching.
- client/server. The interpreter accepts multiple connections from different sources (human users, robots, other servers ...) and enables them to interact.
- distributed. You can run objects in different processes, potentially remote computers across the network.

1.3 Genesis

Urbi what first designed and implemented by Jean-Christophe Baillie, together with Matthieu Nottale. Because its users wildly acclaimed it, Jean-Christophe founded Gostai, a France-based Company that develops software for robotics with a strong emphasis on personal robotics.

Authors Urbi SDK 1 was further developed by Akim Demaille, Guillaume Deslandes, Quentin Hocquet, and Benoît Sigoure.

The Urbi SDK 2 project was started and developed by Akim Demaille, Quentin Hocquet, Matthieu Nottale, and Benoît Sigoure. Samuel Tardieu provided an immense help during the year 2008, in particular for the concurrency and event support.

The maintenance is currently carried out by Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Jean-Christophe Baillie is still deeply involved in the development of urbiscript, he regularly submits ideas, and occasionally even code!

Contributors A number of people contributed significantly to Urbi, including Romain Bezut, Thomas Moulard, Nicolas Pierron.

1.4 Outline

This multi-part document provides a complete guide to Urbi. See [Chapter 23](#) for the various notations that are used in the document.

Part I — Urbi and UObjects User Manual

This part covers the Urbi architecture: its core components (client/server architecture), how its middleware works, how to include extensions as UObjects (C++ components) and so forth.

No knowledge of the urbiscript language is needed. As a matter of fact, Urbi can be used as a standalone middleware architecture to orchestrate the execution of existing components.

Yet urbiscript is a feature that “comes for free”: it is easy using it to experiment, prototype, and even program fully-featured applications that orchestrate native components. The interested reader should read either the urbiscript user manual ([Part II](#)), or the reference manual ([Chapter 19](#)).

Chapter 3 — The UObject API

This section shows the various steps of writing an Urbi C++ component using the UObject API.

Chapter 4 — Use Cases

Interfacing a servomotor device as an example on how to use the UObject architecture as a middleware.

Part II — urbiscript User Manual

This part, also known as the “urbiscript tutorial”, teaches the reader how to program in urbiscript. It goes from the basis to concurrent and event-based programming. No specific knowledge is expected. There is no need for a C++ compiler, as UObject will not be covered here (see [Part I](#)). The reference manual contains a terse and complete definition of the Urbi environment ([Part IV](#)).

Chapter 5 — First Steps

First contacts with urbiscript.

Chapter 6 — Basic Objects, Value Model

A quick introduction to objects and values.

Chapter 7 — Flow Control Constructs

Basic control flow: `if`, `for` and the like.

Chapter 8 — Advanced Functions and Scoping

Details about functions, scoped, and lexical closures.

Chapter 9 — Objective Programming, urbiscript Object Model

A more in-depth introduction to object-oriented programming in urbiscript.

Chapter 10 — Functional Programming

Functions are first-class citizens.

Chapter 11 — Parallelism, Concurrent Flow Control

The urbiscript operators for concurrency, tags.

Chapter 12 — Event-based Programming

Support for event-driven concurrency in urbiscript.

Chapter 13 — Urbi for ROS Users

How to use ROS from Urbi, and vice-versa.

Part III — Guidelines and Cook Books

This part contains guides to some specific aspects of Urbi SDK.

Chapter 14 — Installation

Complete instructions on how to install Urbi SDK.

Chapter 15 — Frequently Asked Questions

Some answers to common questions.

Chapter 16 — Migration from urbiscript 1 to urbiscript 2

This chapter is intended to people who want to migrate programs in urbiscript 1 to urbiscript 2.

Chapter 17 — Building Urbi SDK

Building Urbi SDK from the sources. How to install it, how to check it and so forth.

Part IV — Urbi SDK Reference Manual

This part defines the specifications of the urbiscript language version 2.0. It defines the expected behavior from the urbiscript interpreter, the standard library, and the SDK. It can be used to check whether some code is valid, or browse urbiscript or C++ API for a desired feature. Random reading can also provide you with advanced knowledge or subtleties about some urbiscript aspects.

This part is not an urbiscript tutorial; it is not structured in a progressive manner and is too detailed. Think of it as a dictionary: one does not learn a foreign language by reading a dictionary. The urbiscript Tutorial ([Part II](#)), or the live urbiscript tutorial built in the interpreter are good introductions to urbiscript.

This part does not aim at giving advanced programming techniques. Its only goal is to define the language and its libraries.

Chapter 18 — Programs

Presentation and usage of the different tools available with the Urbi framework related to urbiscript, such as the Urbi server, the command line client, `umake`, ...

Chapter 19 — urbiscript Language Reference Manual

Core constructs of the language and their behavior.

Chapter 20 — urbiscript Standard Library

Listing of all classes and methods provided in the standard library.

Chapter 21 — Communication with ROS

Urbi provides a set of tools to communicate with ROS (Robot Operating System). For more information about ROS, see <http://www.ros.org>. Urbi, acting as a ROS node, is able to interact with the ROS world.

Chapter 22 — Gostai Standard Robotics API

Also known as “The Urbi Naming Standard”: naming conventions in for standard hardware/software devices and components implemented as UObject and the corresponding slots/events to access them.

Part V — Tables and Indexes

This part contains material about the document itself.

Chapter 23 — Notations

Conventions used in the type-setting of this document.

Chapter 25 — Licenses

Licenses of components used in Urbi SDK.

Chapter 24 — Release Notes

Release notes of Urbi SDK.

Chapter 26 — Glossary

Definition of the terms used in this document.

Contents

1	Introduction	3
1.1	Urbi and UObjects	3
1.2	Urbi and urbiscript	4
1.3	Genesis	4
1.4	Outline	5
	Contents	9
2	Getting Started	27
I	Urbi and UObjects User Manual	31
3	The UObject API	35
3.1	Compiling UObjects	35
3.1.1	Compiling by hand	35
3.1.2	The <code>umake-*</code> family of tools	36
3.1.3	Using the Visual C++ Wizard	36
3.2	Creating a class, binding variables and functions	38
3.3	Creating new instances	39
3.4	Binding functions	39
3.4.1	Simple binding	39
3.4.2	Asynchronous binding	40
3.5	Notification of a variable change or access	40
3.6	Data-flow based programming: exchanging UVars	41
3.7	Timers	41
3.8	The special case of sensor/effector variables	42
3.9	Using Urbi variables	42
3.10	Emitting events	42
3.11	UObject and Threads	43
3.12	Using binary types	43
3.12.1	UVar conversion and memory management	43
3.12.2	0-copy mode	43

3.13	Using hubs to group objects	43
3.14	Sending Urbi code	44
4	Use Cases	45
4.1	Writing a Servomotor Device	45
4.1.1	Caching	48
4.1.2	Using Timers	48
4.2	Using Hubs to Group Objects	49
4.2.1	Alternate Implementation	51
4.3	Writing a Camera Device	51
4.3.1	Optimization in Plugin Mode	54
4.4	Writing a Speaker or Microphone Device	54
4.5	Writing a Softdevice: Ball Detection	54
II	urbiscript User Manual	57
5	First Steps	61
5.1	Comments	61
5.2	Literal values	61
5.3	Function calls	62
5.4	Variables	63
5.5	Scopes	63
5.6	Method calls	63
5.7	Function definition	64
5.8	Conclusion	65
6	Basic Objects, Value Model	67
6.1	Objects in urbiscript	67
6.2	Methods	70
6.3	Everything is an object	72
6.4	The urbiscript values model	72
6.5	Conclusion	74
7	Flow Control Constructs	77
7.1	if	77
7.2	while	77
7.3	for	78
7.4	switch	78
7.5	do	79
8	Advanced Functions and Scoping	81
8.1	Scopes as expressions	81
8.2	Advanced scoping	81

<i>CONTENTS</i>	11
8.3 Local functions	82
8.4 Lexical closures	82
9 Objective Programming, urbiscript Object Model	85
9.1 Prototype-based programing in urbiscript	85
9.2 Prototypes and slot lookup	86
9.3 Copy on write	88
9.4 Defining pseudo-classes	89
9.5 Constructors	90
9.6 Operators	91
10 Functional Programming	93
10.1 First class functions	93
10.2 Lambda functions	94
10.3 Lazy arguments	94
11 Parallelism, Concurrent Flow Control	97
11.1 Parallelism operators	97
11.2 Detach	99
11.3 Tags for parallel control flows	99
11.4 Advanced example with parallelism and tags	101
12 Event-based Programming	103
12.1 Event related constructs	103
12.2 Events	105
13 Urbi for ROS Users	107
13.1 Communication on topics	107
13.1.1 Starting a process from Urbi	107
13.1.2 Listening to Topics	108
13.1.3 Advertising on Topics	109
13.1.3.1 Simple Talker	109
13.1.3.2 Turtle Simulation	109
13.2 Using Services	110
III Guidelines and Cook Books	113
14 Installation	117
14.1 Download	117
14.2 Install & Check	117
14.2.1 GNU/Linux and Mac OS X	118
14.2.2 Windows	118

15 Frequently Asked Questions	119
15.1 Build Issues	119
15.1.1 Complaints about ‘+=’	119
15.1.2 error: ‘janonymousꞀ’ is used uninitialized in this function	119
15.1.3 AM.LANGINFO_CODESET	119
15.1.4 ‘make check’ fails	120
15.2 Troubleshooting	120
15.2.1 Error 1723: ”A DLL required for this install to complete could not be run.”	120
15.2.2 When executing a program, the message “The system cannot execute the specified program.” is raised.	120
15.2.3 When executing a program, the message “This application has failed to start” is raised.	120
15.2.4 The server dies with “stack exhaustion”	120
15.2.5 ‘myuobject: file not found’. What can I do?	121
15.3 urbiscript	124
15.3.1 Objects lifetime	124
15.3.1.1 How do I create a new Object derivative?	124
15.3.1.2 How do I destroy an Object?	124
15.3.2 Slots and variables	124
15.3.2.1 Is the lobby a scope?	124
15.3.2.2 How do I add a new slot in an object?	128
15.3.2.3 How do I modify a slot of my object?	129
15.3.2.4 How do I create or modify a local variable?	129
15.3.2.5 How do I make a constructor?	130
15.3.2.6 How can I manipulate the list of prototypes of my objects?	130
15.3.2.7 How can I know the slots available for a given object?	130
15.3.2.8 How do I create a new function?	130
15.3.3 Tags	131
15.3.3.1 How do I create a tag?	131
15.3.3.2 How do I stop a tag?	131
15.3.3.3 Can tagged statements return a value?	131
15.3.4 Events	131
15.3.4.1 How do I create an event?	131
15.3.4.2 How do I emit an event?	132
15.3.4.3 How do I catch an event?	132
15.3.5 Standard Library	132
15.3.5.1 How can I iterate over a list?	132
15.4 UObjects	132
15.4.1 Is the UObject API Thread-Safe?	132
15.4.1.1 Plugin mode	132
15.4.1.2 Remote mode	133
15.5 Miscellaneous	134
15.5.1 What has changed since the latest release?	134

15.5.2	How can I contribute to the code?	134
15.5.3	How do I report a bug?	135
16	Migration from urbiscript 1 to urbiscript 2	137
16.1	\$(Foo)	137
16.2	delete Foo	137
16.3	emit Foo	138
16.4	eval(Foo)	139
16.5	foreach	139
16.6	group	139
16.7	loopn	139
16.8	new Foo	139
16.9	self	140
16.10	stop Foo	140
16.11	# line	140
16.12	tag+end	140
17	Building Urbi SDK	141
17.1	Requirements	141
17.1.1	Bootstrap	141
17.1.2	Build	143
17.2	Check out	144
17.3	Bootstrap	144
17.4	Configure	144
17.4.1	configuration options	145
17.4.2	Windows: Cygwin	145
17.4.3	building For Windows	145
17.4.4	Building for Windows using Cygwin	146
17.5	Compile	146
17.6	Install	146
17.7	Relocatable	146
17.8	Run	147
17.9	Check	147
17.9.1	Lazy test suites	148
17.9.2	Partial test suite runs	148
IV	Urbi SDK Reference Manual	151
18	Programs	155
18.1	Environment Variables	155
18.1.1	Search Path Variables	155
18.1.2	Environment Variables	156
18.2	Special Files	156

18.3	urbi — Running an Urbi Server	157
18.3.1	Options	157
18.4	urbi-image — Querying Images from a Server	158
18.4.1	Options	159
18.5	urbi-launch — Running a UObject	160
18.5.1	Invoking urbi-launch	160
18.5.2	Examples	161
18.6	urbi-send — Sending urbiscript Commands to a Server	161
18.7	umake — Compiling UObject Components	162
18.7.1	Invoking umake	162
18.7.2	umake Wrappers	163
19	urbiscript Language Reference Manual	165
19.1	Syntax	165
19.1.1	Characters, encoding	165
19.1.2	Comments	165
19.1.3	Synclines	166
19.1.4	Identifiers	166
19.1.5	Keywords	167
19.1.6	Literals	167
19.1.6.1	Angles	167
19.1.6.2	Dictionaries	167
19.1.6.3	Durations	169
19.1.6.4	Floats	169
19.1.6.5	Lists	170
19.1.6.6	Strings	171
19.1.6.7	Tuples	172
19.1.7	Statement Separators	173
19.1.7.1	‘;’	173
19.1.7.2	‘,’	173
19.1.7.3	‘ ’	174
19.1.7.4	‘&’	175
19.1.8	Operators	175
19.1.8.1	Arithmetic operators	175
19.1.8.2	Assignment operators	176
19.1.8.3	Bitwise operators	176
19.1.8.4	Logical operators	176
19.1.8.5	Comparison operators	177
19.1.8.6	Container operators	178
19.1.8.7	Object operators	179
19.1.8.8	All operators summary	179
19.2	Scopes and local variables	179
19.2.1	Scopes	179

19.2.2	Local variables	181
19.3	Functions	182
19.3.1	Function Definition	182
19.3.2	Arguments	183
19.3.3	Return value	183
19.3.4	Call messages	184
19.3.5	Strictness	184
19.3.6	Lexical closures	185
19.4	Objects	186
19.4.1	Slots	186
19.4.1.1	Manipulation	186
19.4.1.2	Syntactic Sugar	187
19.4.2	Prototypes	187
19.4.2.1	Manipulation	187
19.4.2.2	Inheritance	187
19.4.2.3	Copy on write	188
19.4.3	Sending messages	189
19.5	Structural Pattern Matching	190
19.5.1	Basic Pattern Matching	190
19.5.2	Variable	191
19.5.3	Guard	191
19.6	Imperative flow control	192
19.6.1	break	192
19.6.2	continue	193
19.6.3	do	193
19.6.4	if	193
19.6.5	for	194
19.6.5.1	C-like for	194
19.6.5.2	Range-for	195
19.6.5.3	for <i>n</i> -times	196
19.6.6	if	196
19.6.7	loop	196
19.6.8	switch	197
19.6.9	while	198
19.6.9.1	while;	198
19.6.9.2	while—	199
19.7	Exceptions	200
19.7.1	Throwing exceptions	200
19.7.2	Catching exceptions	200
19.7.3	Inspecting exceptions	201
19.8	Assertions	201
19.8.1	Asserting an Expression	201
19.8.2	Assertion Blocks	202

19.9	Parallel and event-based flow control	203
19.9.1	at	203
19.9.1.1	at on Events	203
19.9.1.2	at on Boolean Expressions	203
19.9.1.3	Scoping at at	204
19.9.2	every	204
19.9.2.1	every	204
19.9.2.2	every,	206
19.9.3	for	206
19.9.3.1	C-for,	207
19.9.3.2	range-for& (:)	207
19.9.3.3	for& (n)	208
19.9.4	loop,	208
19.9.5	waituntil	209
19.9.5.1	waituntil on Events	209
19.9.5.2	waituntil on Boolean Expressions	210
19.9.6	whenever	210
19.9.6.1	whenever on Events	211
19.9.6.2	whenever on Boolean Expressions	212
19.9.7	While	213
19.9.7.1	while,	213
19.10	Trajectories	214
19.11	Garbage collection and limitations	215
20	urbiscript Standard Library	217
20.1	Barrier	217
20.1.1	Prototypes	217
20.1.2	Construction	217
20.1.3	Slots	217
20.2	Binary	218
20.2.1	Prototypes	218
20.2.2	Construction	218
20.2.3	Slots	219
20.3	Boolean	220
20.3.1	Prototypes	220
20.3.2	Construction	220
20.3.3	Truth Values	220
20.3.4	Slots	221
20.4	CallMessage	221
20.4.1	Examples	221
20.4.1.1	Evaluating an argument several times	221
20.4.1.2	Strict Functions	222
20.4.2	Slots	222

20.5	Channel	225
20.5.1	Prototypes	225
20.5.2	Construction	225
20.5.3	Slots	225
20.6	Code	227
20.6.1	Prototypes	227
20.6.2	Construction	227
20.6.3	Slots	228
20.7	Comparable	229
20.7.1	Slots	230
20.8	Container	230
20.8.1	Prototypes	230
20.8.2	Slots	230
20.9	Control	231
20.9.1	Prototypes	231
20.9.2	Slots	231
20.10	Date	233
20.10.1	Prototypes	233
20.10.2	Construction	233
20.10.3	Slots	234
20.11	Dictionary	235
20.11.1	Example	235
20.11.2	Prototypes	235
20.11.3	Construction	236
20.11.4	Slots	236
20.12	Directory	239
20.12.1	Prototypes	240
20.12.2	Construction	240
20.12.3	Slots	240
20.13	Duration	241
20.13.1	Prototypes	241
20.13.2	Construction	241
20.13.3	Slots	242
20.14	Event	242
20.14.1	Prototypes	242
20.14.2	Examples	242
20.14.3	Construction	242
20.14.4	Slots	242
20.15	Exception	243
20.15.1	Prototypes	244
20.15.2	Construction	244
20.15.3	Slots	244
20.15.4	Specific Exceptions	245

20.16Executable	248
20.16.1 Prototypes	248
20.16.2 Construction	248
20.16.3 Slots	248
20.17File	248
20.17.1 Prototypes	248
20.17.2 Construction	248
20.17.3 Slots	249
20.18Finalizable	250
20.18.1 Example	250
20.18.2 Prototypes	250
20.18.3 Construction	251
20.18.4 Slots	252
20.19Float	252
20.19.1 Prototypes	252
20.19.2 Construction	252
20.19.3 Slots	253
20.20Float.limits	261
20.20.1 Slots	261
20.21FormatInfo	262
20.21.1 Prototypes	262
20.21.2 Construction	262
20.21.3 Slots	263
20.22Formatter	265
20.22.1 Prototypes	265
20.22.2 Construction	265
20.22.3 Slots	265
20.23Global	266
20.23.1 Prototypes	266
20.23.2 Slots	266
20.24Group	273
20.24.1 Example	273
20.24.2 Prototypes	274
20.24.3 Construction	274
20.24.4 Slots	274
20.25InputStream	275
20.25.1 Prototypes	275
20.25.2 Construction	275
20.25.3 Slots	275
20.26IoService	276
20.26.1 Example	277
20.26.2 Prototypes	277
20.26.3 Construction	277

20.26.4 Slots	277
20.27 Job	278
20.27.1 Prototypes	278
20.27.2 Construction	278
20.27.3 Slots	278
20.28 Kernel1	280
20.28.1 Prototypes	280
20.28.2 Construction	280
20.28.3 Slots	280
20.29 Lazy	282
20.29.1 Examples	282
20.29.1.1 Evaluating once	282
20.29.1.2 Evaluating several times	283
20.29.2 Caching	283
20.29.3 Prototypes	284
20.29.4 Construction	284
20.29.5 Slots	284
20.30 List	285
20.30.1 Prototypes	285
20.30.2 Construction	285
20.30.3 Slots	286
20.31 Loadable	295
20.31.1 Prototypes	295
20.31.2 Example	296
20.31.3 Construction	296
20.31.4 Slots	296
20.32 Lobby	297
20.32.1 Prototypes	297
20.32.2 Construction	297
20.32.3 Examples	298
20.32.4 Slots	298
20.33 Location	302
20.33.1 Prototypes	302
20.33.2 Construction	302
20.33.3 Slots	302
20.34 Math	304
20.34.1 Prototypes	304
20.34.2 Construction	304
20.34.3 Slots	304
20.35 Mutex	306
20.35.1 Prototypes	306
20.35.2 Construction	306
20.35.3 Slots	307

20.36	nil	307
20.36.1	Prototypes	307
20.36.2	Construction	307
20.36.3	Slots	307
20.37	Object	307
20.37.1	Prototypes	308
20.37.2	Construction	308
20.37.3	Slots	308
20.38	Orderable	318
20.39	OutputStream	318
20.39.1	Prototypes	318
20.39.2	Construction	319
20.39.3	Slots	319
20.40	Pair	319
20.40.1	Prototype	319
20.40.2	Construction	320
20.40.3	Slots	320
20.41	Path	320
20.41.1	Prototypes	320
20.41.2	Construction	320
20.41.3	Slots	321
20.42	Pattern	323
20.42.1	Prototypes	323
20.42.2	Construction	323
20.42.3	Slots	323
20.43	Position	325
20.43.1	Prototypes	325
20.43.2	Construction	325
20.43.3	Slots	325
20.44	Primitive	327
20.44.1	Prototypes	327
20.44.2	Construction	327
20.44.3	Slots	327
20.45	Process	327
20.45.1	Prototypes	327
20.45.2	Example	328
20.45.3	Construction	328
20.45.4	Slots	329
20.46	Profiling	331
20.46.1	Prototypes	331
20.46.2	Construction	331
20.47	PseudoLazy	332
20.48	PubSub	332

20.48.1 Prototypes	332
20.48.2 Construction	332
20.48.3 Slots	332
20.49 PubSub.Subscriber	333
20.49.1 Prototypes	333
20.49.2 Construction	333
20.49.3 Slots	333
20.50 RangeIterable	334
20.50.1 Prototypes	334
20.50.2 Slots	334
20.51 Regexp	335
20.51.1 Prototypes	335
20.51.2 Construction	335
20.51.3 Slots	335
20.52 StackFrame	337
20.52.1 Construction	337
20.52.2 Slots	338
20.53 Semaphore	338
20.53.1 Prototypes	338
20.53.2 Construction	338
20.53.3 Slots	338
20.54 Server	340
20.54.1 Prototypes	340
20.54.2 Construction	340
20.54.3 Slots	340
20.55 Singleton	341
20.55.1 Prototypes	341
20.55.2 Construction	341
20.55.3 Slots	342
20.56 Socket	342
20.56.1 Example	342
20.56.2 Prototypes	344
20.56.3 Construction	344
20.56.4 Slots	344
20.57 String	345
20.57.1 Prototypes	345
20.57.2 Construction	346
20.57.3 Slots	346
20.58 System	350
20.58.1 Prototypes	351
20.58.2 Slots	351
20.59 System.PackageInfo	358
20.60 System.Platform	358

20.61	Tag	359
20.61.1	Examples	359
20.61.1.1	Stop	359
20.61.1.2	Block/unblock	361
20.61.1.3	Freeze/unfreeze	362
20.61.1.4	Scope tags	362
20.61.1.5	Enter/leave events	363
20.61.1.6	Begin/end	365
20.61.2	Construction	365
20.61.3	Slots	365
20.61.4	Hierarchical tags	366
20.62	Timeout	367
20.62.1	Prototypes	367
20.62.2	Construction	367
20.62.3	Examples	368
20.62.4	Slots	368
20.63	TrajectoryGenerator	368
20.63.1	Prototypes	369
20.63.2	Examples	369
20.63.2.1	Accel	369
20.63.2.2	Cos	369
20.63.2.3	Sin	369
20.63.2.4	Smooth	370
20.63.2.5	Speed	370
20.63.2.6	Time	371
20.63.2.7	Trajectories and Tags	371
20.63.3	Construction	372
20.63.4	Slots	373
20.64	Triplet	373
20.64.1	Prototype	373
20.64.2	Construction	373
20.64.3	Slots	374
20.65	Tuple	374
20.65.1	Prototype	374
20.65.2	Construction	374
20.65.3	Slots	375
20.66	UObject	376
20.66.1	Prototypes	377
20.66.2	Slots	377
20.67	UValue	377
20.68	UVar	377
20.68.1	Construction	377
20.68.2	Prototypes	377

20.68.3 Slots	377
20.69 void	378
20.69.1 Prototypes	378
20.69.2 Construction	378
20.69.3 Slots	378
21 Communication with ROS	381
21.1 Ros	381
21.1.1 Construction	381
21.1.2 Slots	382
21.2 Ros.Topic	383
21.2.1 Construction	383
21.2.2 Slots	383
21.2.2.1 Common	383
21.2.2.2 Subscription	383
21.2.2.3 Advertising	384
21.2.3 Example	385
21.3 Ros.Service	386
21.3.1 Construction	386
21.3.2 Slots	386
22 Gostai Standard Robotics API	389
22.1 The Structure Tree	389
22.2 Frame of Reference	391
22.3 Component naming	392
22.4 Localization	392
22.5 Interface	394
22.5.1 Identity	394
22.5.2 Network	395
22.5.3 Motor	395
22.5.4 LinearMotor (subclass of Motor)	395
22.5.5 LinearSpeedMotor (subclass of Motor)	395
22.5.6 RotationalMotor (subclass of Motor)	396
22.5.7 RotationalSpeedMotor (subclass of Motor)	396
22.5.8 Sensor	396
22.5.9 DistanceSensor (subclass of Sensor)	396
22.5.10 TouchSensor (subclass of Sensor)	396
22.5.11 AccelerationSensor (subclass of Sensor)	397
22.5.12 GyroSensor (subclass of Sensor)	397
22.5.13 TemperatureSensor (subclass of Sensor)	397
22.5.14 Laser (subclass of Sensor)	397
22.5.15 Mobile	398
22.5.16 Tracker	398
22.5.17 VideoIn	398

22.5.18	AudioOut	399
22.5.19	AudioIn	400
22.5.20	BlobDetector	400
22.5.21	TextToSpeech	401
22.5.22	SpeechRecognizer	401
22.5.23	Led	402
22.5.24	RGBLed (subclass of Led)	402
22.5.25	Battery	402
22.6	Standard Components	402
22.6.1	Yaw/Pitch/Roll orientation	403
22.6.2	Standard Component List	403
22.7	Compact notation	407
22.8	Support classes	408
22.8.1	Interface	409
22.8.2	Component	409
22.8.3	Localizer	409
V	Tables and Indexes	411
23	Notations	415
23.1	Words	415
23.2	Frames	415
23.2.1	Shell Sessions	415
23.2.2	urbiscript Sessions	416
23.2.3	urbiscript Assertions	416
23.2.4	C++ Code	416
24	Release Notes	419
24.1	Urbi SDK 2.1	419
24.1.1	Fixes	419
24.1.2	New Features	419
24.1.3	Optimization	420
24.1.4	Documentation	420
24.2	Urbi SDK 2.0.3	420
24.2.1	New Features	420
24.2.2	Fixes	421
24.2.3	Documentation	421
24.3	Urbi SDK 2.0.2	422
24.3.1	urbiscript	422
24.3.2	Fixes	422
24.3.3	Documentation	422
24.4	Urbi SDK 2.0.1	422
24.4.1	urbiscript	422

24.4.2	Documentation	422
24.4.2.1	Fixes	422
24.5	Urbi SDK 2.0	423
24.5.1	urbiscript	423
24.5.1.1	Changes	423
24.5.1.2	New features	423
24.5.2	UObjects	424
24.5.3	Documentation	424
24.6	Urbi SDK 2.0 RC 4	424
24.6.1	urbiscript	425
24.6.1.1	Changes	425
24.6.1.2	New objects	425
24.6.1.3	New features	425
24.6.2	UObjects	425
24.7	Urbi SDK 2.0 RC 3	425
24.7.1	urbiscript	425
24.7.1.1	Fixes	425
24.7.1.2	Changes	425
24.7.2	Documentation	426
24.8	Urbi SDK 2.0 RC 2	426
24.8.1	Optimization	426
24.8.2	urbiscript	426
24.8.2.1	New constructs	426
24.8.2.2	New objects	426
24.8.2.3	New features	427
24.8.2.4	Fixes	428
24.8.2.5	Deprecations	428
24.8.2.6	Changes	428
24.8.3	UObjects	429
24.8.4	Documentation	429
24.8.5	Various	431
24.9	Urbi SDK 2.0 RC 1	431
24.9.1	Auxiliary programs	431
24.9.2	urbiscript	431
24.9.2.1	Syntax of events	431
24.9.2.2	Changes	431
24.9.2.3	Fixes	432
24.9.3	URBI Remote SDK	432
24.9.4	Documentation	432
24.10	Urbi SDK 2.0 beta 4	432
24.10.1	Documentation	432
24.10.2	urbiscript	432
24.10.2.1	Bug fixes	432

24.10.2.2 Changes	433
24.10.3 Programs	433
24.10.3.1 Environment variables	433
24.10.3.2 Scripting	433
24.10.3.3 urbi-console	433
24.10.3.4 Auxiliary programs	434
24.11 Urbi SDK 2.0 beta 3	434
24.11.1 Documentation	434
24.11.2 urbiscript	434
24.11.2.1 Fixes	434
24.11.2.2 Changes	434
24.11.3 UObjects	437
24.11.4 Auxiliary programs	437
24.12 Urbi SDK 2.0 beta 2	438
24.12.1 urbiscript	438
24.12.2 Standard library	438
24.12.3 UObjects	439
24.12.4 Run-time	439
24.12.5 Bug fixes	440
24.12.6 Auxiliary programs	440
25 Licenses	441
25.1 BSD License	441
25.2 Expat License	442
25.3 Independent JPEG Group's Software License	442
25.4 Libcoroutine License	443
25.5 OpenSSL License	444
25.6 Urbi Open Source Contributor Agreement	447
26 Glossary	449
27 Index	452

Chapter 2

Getting Started

urbiscript comes with a set of tools, two of which being of particular importance:

urbi launches an Urbi server. There are several means to interact with it, which we will see later.

urbi-launch runs Urbi components, the UObjects, and connects them to an Urbi server.

Please, first make sure that these tools are properly installed. If you encounter problems, please see the frequently asked questions ([Chapter 15](#)), and the detailed installation instructions ([Chapter 14](#)).

```
# Make sure urbi is properly installed.
$ urbi --version
Urbi Kernel version preview/2.0/beta3-425 rev. 000913e
Copyright (C) 2005-2010 Gostai SAS.

URBI SDK Remote version preview/1.6/beta1-666 rev. 92ec3b4
Copyright (C) 2004-2010 Gostai SAS.

Libport version preview/1.0/beta1-1048 rev. f1c5170
Copyright (C) 2005-2010 Gostai SAS.
```

There are several means to interact with a server spawned by ***urbi***, see [Section 18.3](#) for details. First of all, you may use the options ‘-e’/‘--expression *code*’ and ‘-f’/‘--file *file*’ to send some *code* or the contents of some *file* to the newly run server. The option ‘-q’/‘--quiet’ discards the banner.

You may combine any number of these options, but beware that being event-driven, the server does not “know” when a program ends. Therefore, batch programs should end by calling ***shutdown***. Using a Unix shell:

```
# A classical program.
$ urbi -q -e 'echo("Hello, World!");' -e 'shutdown;'
[00000004] *** Hello, World!
```

Listing 2.1: A batch session under Unix.

If you are running Windows, then, since the quotation rules differ, run:

```
# A classical program.
$ urbi -q -e "echo("Hello, World!");" -e "shutdown;"
[00000004] *** Hello, World!
```

Listing 2.2: A batch session under Windows.

To run an interactive session, use option ‘-i’/‘--interactive’. Like most interactive interpreters, Urbi will evaluate the given commands and print out the results.

```
$ urbi -i
[00000987] *** *****
[00000990] *** Urbi SDK version 2.0.3 rev. d6a568d
[00001111] *** Copyright (C) 2005-2010 Gostai S.A.S.
[00001111] ***
[00001111] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00001112] *** be used under certain conditions. Type 'license;',
[00001112] *** 'authors;', or 'copyright;' for more information.
[00001112] ***
[00001112] *** Check our community site: http://www.urbiforge.org.
[00001112] *** *****
1+2;
[00001200] 3
shutdown;
```

Listing 2.3: An interactive session under Unix.

The output from the server is prefixed by a number surrounded by square brackets: this is the date (in milliseconds since the server was launched) at which that line was sent by the server. This is useful at occasions, since Urbi is meant to run many parallel commands. But since these timestamps are irrelevant in most examples, they will often be filled with zeroes through this documentation.

Under Unix, the program `rlwrap` provides additional services (history of commands, advanced command line edition etc.); run ‘`rlwrap urbi -i`’.

In either case the server can also be made available for network-based interactions using option ‘--port *port*’. Note that while `shutdown` asks the server to quit, `quit` only quits one interactive session. In the following example (under Unix) the server is still available for other, possibly concurrent, sessions.

```
$ urbi --port 54000 &
[1] 77024
$ telnet localhost 54000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[00000987] *** *****
[00000990] *** Urbi SDK version 2.0.3 rev. d6a568d
[00001111] *** Copyright (C) 2005-2010 Gostai S.A.S.
[00001111] ***
[00001111] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00001112] *** be used under certain conditions. Type 'license;',
```

```
[00001112] *** 'authors;', or 'copyright;' for more information.
[00001112] ***
[00001112] *** Check our community site: http://www.urbiforge.org.
[00001112] *** *****
12345679*8;
[00018032] 98765432
quit;
```

Listing 2.4: An interactive session under Unix.

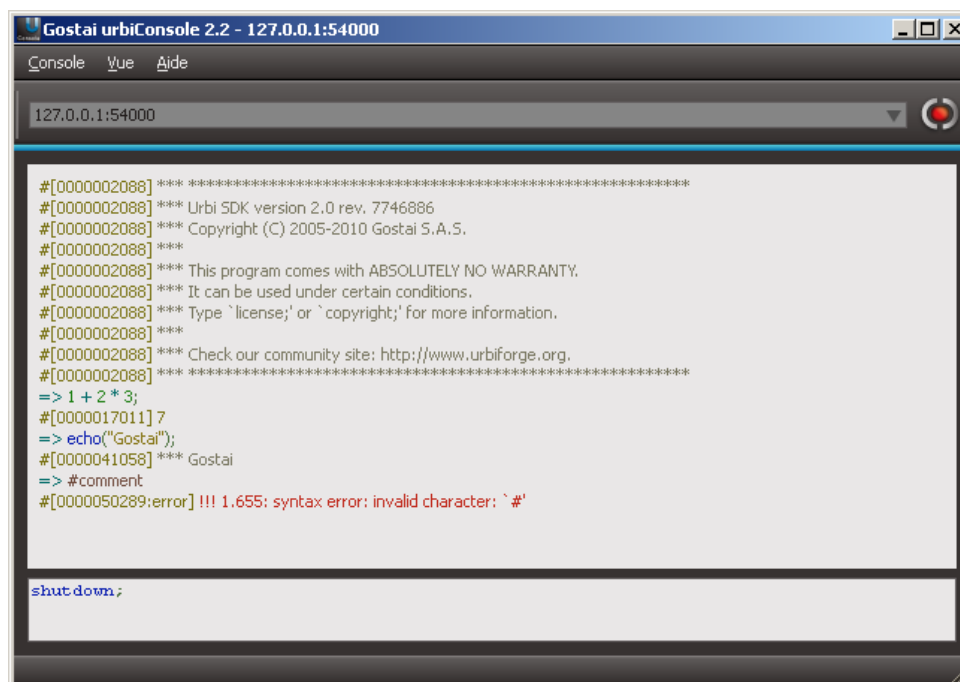
Under Windows, instead of using `telnet`, you may use `urbiConsole` (part of the package), which provides a Graphical User Interface to a network-connection to an Urbi server. To launch the server, run:

```
C:\...\> start urbi --port 54000
```

Listing 2.5: Starting an interactive session under Windows.

and to launch the client, click on `urbiConsole` which is installed by the installer.

Then, the interaction proceeds in the `urbiConsole` windows. Specify the host name and port to use ('127.0.0.1:54000') in the text field in the top of the window and click on the right to start the connection.



The program `urbi-send` (see [Section 18.6](#)) provides a nice interface to send batches of instructions (and/or files) to a running server.

```
$ urbi-send -P 54000 -e "1+2*3;" -Q
[00018032] 7
# Have the server shutdown;
$ urbi-send -P 54000 -e "shutdown;"
```

You can now send commands to your Urbi server. If at any point you get lost, or want a fresh start, you can simply close and reopen your connection to the server to get a clean environment. In some cases, particularly if you made global changes in the environment, it is simpler to start anew: shut your current server down, and spawn a new one.

You are now ready to proceed to the urbiscript tutorial: see [Part II](#).
Enjoy!

Part I

Urbi and UObjects User Manual

About This Part

This part covers the Urbi architecture: its core components (client/server architecture), how its middleware works, how to include extensions as UObjects (C++ components) and so forth.

No knowledge of the urbiscript language is needed. As a matter of fact, Urbi can be used as a standalone middleware architecture to orchestrate the execution of existing components.

Yet urbiscript is a feature that “comes for free”: it is easy using it to experiment, prototype, and even program fully-featured applications that orchestrate native components. The interested reader should read either the urbiscript user manual ([Part II](#)), or the reference manual ([Chapter 19](#)).

Chapter 3 — The UObject API

This section shows the various steps of writing an Urbi C++ component using the UObject API.

Chapter 4 — Use Cases

Interfacing a servomotor device as an example on how to use the UObject architecture as a middleware.

Chapter 3

The UObject API

The UObject API can be used to add new objects written in C++ to the urbiscript language, and to interact from C++ with the objects that are already defined. We cover the use cases of controlling a physical device (servomotor, speaker, camera...), and interfacing higher-level components (voice recognition, object detection...) with Urbi.

The C++ API defines the UObject class. To each instance of a C++ class deriving from UObject will correspond an urbiscript object sharing some of its methods and attributes. The API provides methods to declare which elements of your object are to be shared. To share a variable with Urbi, you have to give it the type UVar. This type is a container that provides conversion and assignment operators for all types known to Urbi: `double`, `std::string` and `char*`, and the binary-holding structures `UBinary`, `USound` and `UImage`. This type can also read from and write to the liburbi UValue class. The API provides methods to set up callbacks functions that will be notified when a variable is modified or read from Urbi code. Instance methods of any prototype can be rendered accessible from urbiscript, providing all the parameters types and the return type can be converted to/from UValue.

3.1 Compiling UObjects

UObjects can be compiled easily directly with any regular compiler. Nevertheless, Urbi SDK provides two tools to compile UObject seamlessly.

In the following sections, we will try to compile a shared library named ‘`factory.so`’ (or ‘`factory.dll`’ on Windows platforms) from a set of four files (‘`factory.hh`’, ‘`factory.cc`’, ‘`ufactory.hh`’, ‘`ufactory.cc`’). These files are stored in a ‘`factory.uob`’ directory; its name bares no importance, yet the ‘`*.uob`’ extension makes clear that it is a UObject.

In what follows, *urbi-root* denotes the top-level directory of your Urbi SDK package, see [Section 14.2](#).

3.1.1 Compiling by hand

On Unix platforms, compiling by hand into a shared library is straightforward:

```
$ g++ -I urbi-root/include \
```

```
-fPIC -shared \  
factory.uob/*.cc -o factory.so  
$ file factory.so  
factory.so: ELF 32-bit LSB shared object, Intel 80386, \  
version 1 (SYSV), dynamically linked, not stripped
```

On Mac OS X the flags ‘-Wl,-undefined,dynamic_lookup’ are needed:

```
$ g++ -I urbi-root/include \  
-shared -Wl,-undefined,dynamic_lookup \  
factory.uob/*.cc -o factory.so  
$ file factory.so  
factory.so: Mach-O 64-bit dynamically linked shared library x86_64
```

3.1.2 The umake-* family of tools

umake can be used to compile UObjects. See [Section 18.7](#) for its documentation.

You can give it a list of files to compile:

```
$ umake -q --shared-library factory.uob/*.cc -o factory.so  
umake: running to build library.
```

or directories in which C++ sources are looked for:

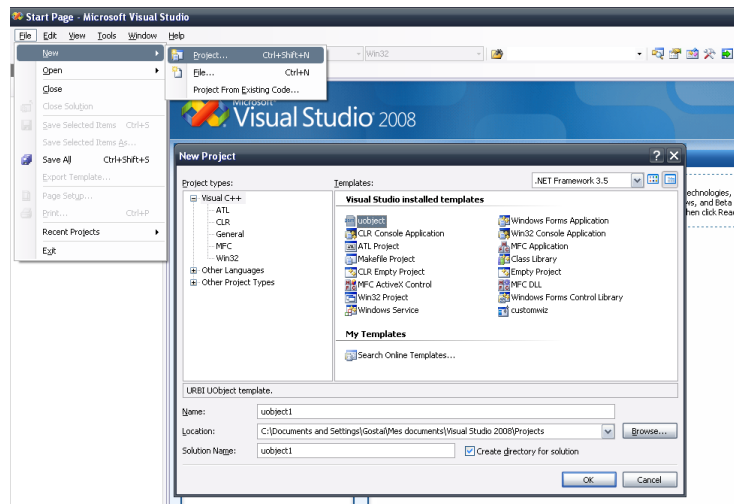
```
$ umake -q --shared-library factory.uob -o factory.so  
umake: running to build library.
```

or finally, if you give no argument at all, the sources in the current directory:

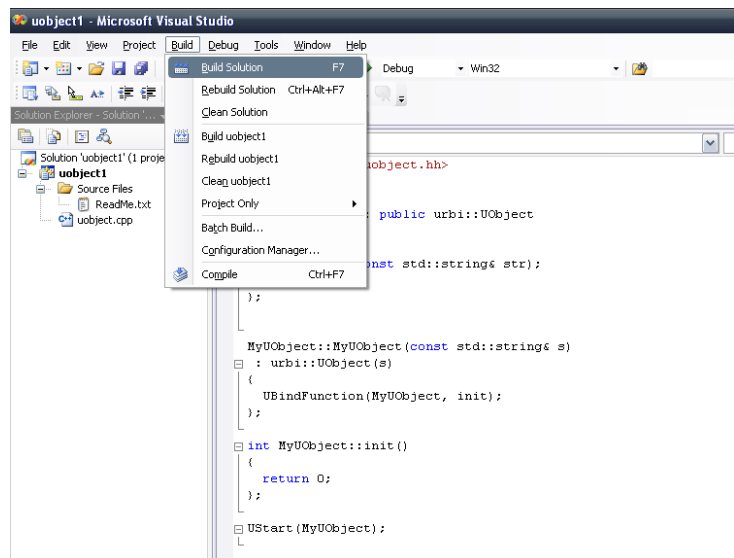
```
$ cd factory.uob  
$ umake -q --shared-library -o factory.so  
umake: running to build library.
```

3.1.3 Using the Visual C++ Wizard

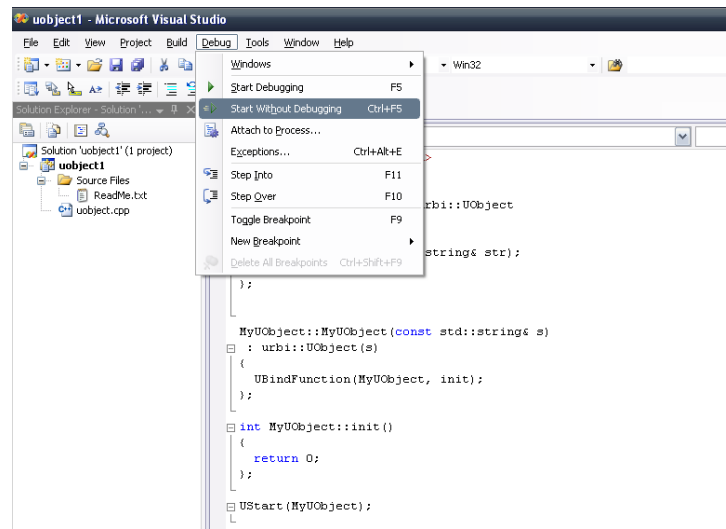
If you installed Urbi SDK using its installer, and if you had Visual C++ installed, then the UObject wizard was installed. Use it to create your UObject code:



Then, compile your UObject.



And run it.



3.2 Creating a class, binding variables and functions

Let's illustrate those concepts by defining a simple object: **adder**. This object has one variable **v**, and a method **add** that returns the sum of this variable and its argument.

- First the required include:

```
#include <urbi/uobject.hh>
```

- Then we declare our **adder** class:

```
class adder : public urbi::UObject // Must inherit from UObject.
{
public:
    // The class must have a single constructor taking a string.
    adder (const std::string& s);

    // Our variable.
    urbi::UVar v;

    // Our method.
    double add (double);
};
```

- The implementation of the constructor and our **add** method:

```
// the constructor defines what is available from Urbi
adder::adder (const std::string& s)
: UObject (s) // required
{
    // Bind the variable.
    UBindVar (adder, v);
```

```

    // Bind the function.
    UBindFunction (adder, add);
}

double
adder::add (double rhs)
{
    return v + rhs;
}

```

- And register this class:

```

// Register the class to the Urbi kernel.
UStart (adder);

```

To summarize:

- Declare your object class as inheriting from `urbi::UObject`.
- Declare a single constructor taking a string, and pass this string to the constructor of `urbi::UObject`.
- Declare the variables you want to share with Urbi with the type `urbi::UVar`.
- In the constructor, use the macros `UBindVar(class-name, variable-name)` for each `UVar` you want as an instance variable, and `UBindFunction(class-name, function-name)` for each function you want to bind.
- Call the macro `UStart` for each object.

3.3 Creating new instances

When you start an Urbi server, an object of each class registered with `UStart` is created with the same name as the class. New instances can be created from Urbi using the `new` method. For each instance created in Urbi, a corresponding instance of the C++ object is created. You can get the arguments passed to the constructor by defining and binding a method named `init` with the appropriate number of arguments.

3.4 Binding functions

3.4.1 Simple binding

You can register any member function of your `UObject` using the macro

```
UBindFunction(class-name, function-name).
```

Once done, the function can be called from urbiscript.

The following types for arguments and return value are supported:

- Basic integer and floating types (int, double, float...).
- `const std::string&` or `const char*`.
- `urbi::UValue` or any of its subtypes (`UBinary`, `UList`...).
- `std::list` or `std::vector` of the above types.

3.4.2 Asynchronous binding

Functions bound using `UBindFunction` are called synchronously, and thus block everything until they return.

If you wish to bind a function that requires a non-negligible amount of time to execute, you can have it execute in a separate thread by calling

```
UBindThreadedFunction(class-name, function-name, lockMode).
```

The function code will be executed in a separate thread without breaking the urbiscript execution semantics.

The `lockMode` argument can be used to prevent parallel execution of multiple bound functions if your code is not thread-safe. It can be any of `LOCK_NONE`, `LOCK_FUNCTION`, `LOCK_INSTANCE`, `LOCK_CLASS` or `LOCK_MODULE`. When set to `LOCK_NONE`, no locking is performed. Otherwise, it limits parallel executions to:

- One instance of the bound function for `LOCK_FUNCTION`.
- One bound function for each object instance for `LOCK_INSTANCE`.
- One bound function for the class for `LOCK_CLASS`.
- One bound function for the whole module (shared object) for `LOCK_MODULE`.

There is however a restriction: you cannot mix multiple locking modes: for instance a function bound with `LOCK_FUNCTION` mode will not prevent another function bound with `LOCK_INSTANCE` from executing in parallel.

You can perform your own locking using semaphores if your code needs a more complex locking model.

You can limit the maximum number of threads that can run in parallel by using the `setThreadLimit` function.

3.5 Notification of a variable change or access

You can register a function that will be called each time a variable is modified or accessed (for embedded components only) by calling `UNotifyChange` and `UNotifyAccess`, passing either an `UVar` or a variable name as first argument, and a member function of your `UObject` as second argument. This function can take zero or one argument: a `UVar` reference pointing to the `UVar` being accessed or modified. The `notifyChange` callback function is called after the variable

value is changed, whereas the `notifyAccess` callback is called before the variable is accessed, giving you the possibility to update its value.

Notify functions can be unregistered by calling the `unnotify` function of the `UVar` class.

3.6 Data-flow based programming: exchanging UVars

The `UNotifyChange` and `UNotifyAccess` features can be used to link multiple `UObject`s together, and perform data-flow based programming: the `UNotifyChange` can be called to monitor `UVars` from other `UObject`s. Those `UVars` can be transmitted through bound function calls.

One possible pattern is to have each data-processing `UObject` take its input from monitored `UVars`, given in its constructor, and output the result of its processing in other `UVars`. Consider the following example of an object-tracker:

```
class ObjectTracker: public urbi::UObject
{
    ObjectTracker(const std::string& n)
        : urbi::UObject(n)
    {
        // Bind our constructor.
        UBindFunction(ObjectTracker, init);
    }
    // Take our data source in our constructor.
    void init(UVar& image)
    {
        UNotifyChange(image, &ObjectTracker::onImage);
        // Bind our output variable.
        UBindVar(ObjectTracker, val);
    }
    void onImage(UVar& src)
    {
        UBinary b = src;
        // Processing here.
        val = processing_result;
    }
    UVar val;
};
UStart(ObjectTracker);
```

The following urbiscript code would be used to initialize an `ObjectTracker` given a camera:

```
var tracker = ObjectTracker.new(camera.getSlot("val"));
```

An other component could then take the tracker output as its input.

Using this model, chains of processing elements can be created. Each time the `UObject` at the start of the chain updates, all the `notifyChange` will be called synchronously in cascade to update the state of the intermediate components.

3.7 Timers

The API provides two methods to have a function called periodically:

- `void urbi::UObject::USetUpdate(ufloat period)`
Set up a timer that calls the virtual method `UObject::update()` with the specified period (in milliseconds). Disable updates if *period* is -1.
- `urbi::TimerHandle urbi::UObject::USetTimer<T>(ufloat period, void (T::*fun)())`
Invoke an `UObject` member function *fun* every *period* milliseconds. *fun* is a regular member-function pointer, for instance `MyUObject::my_function`. The function returns a `TimerHandle` that can be passed to the `UObject::removeTimer(h)` function to disable the timer.

3.8 The special case of sensor/effector variables

In Urbi, a variable can have a different meaning depending on whether you are reading or writing it: you can use the same variable to represent the target value of an effector and the current value measured by an associated sensor. This special mode is activated by the `UObject` defining the variable by calling `UOwned` after calling `UBindVar`. This call has the following effects:

- When Urbi code or code in other modules read the variable, they read the current value.
- When Urbi code or code in other modules write the variable, they set the target value.
- When the module that called `UOwned` reads the variable, it reads the target value. When it writes the variable, it writes the current value.

3.9 Using Urbi variables

You can read or write any Urbi variable by creating an `UVar` passing the variable name to the constructor. Change the value by writing any compatible type to the `UVar`, and access the value by casting the `UVar` to any compatible type.

Some care must be taken in remote mode: changes on the variable coming from Urbi code or an other module can take time to propagate to the `UVar`. To have more control on the bandwidth used, you can disable the automatic update by calling `unnotify()`. Then you can get the value on demand by calling `UNotifyOnRequest (variable, function)`. Then to get an updated value, call the `requestValue` method on the `UVar`, and your callback function will be called as soon as the value is available.

You can read and write all the Urbi properties of an `UVar` by reading and writing the appropriate `UProp` object in the `UVar`.

3.10 Emitting events

The `UEvent` class can be used to create and emit urbiscript events. Instances are created and initialized exactly as `UVar`: either by using the `UBindEvent` macro, or by calling one of its constructors or the `init` function.

Once initialized, the `emit` function will trigger the emission of the associated urbiscript event. It can be called with any number of arguments, of any compatible type.

3.11 UObject and Threads

The `UObject` API is thread-safe in both plugin and remote mode: All API calls including operations on `UVar` can be performed from any thread.

3.12 Using binary types

Urbi can store binary objects of any type in a generic container, and provides specific structures for sound and images. The generic containers is called `UBinary` and is defined in the ‘`urbi/ubinary.hh`’ header. It contains an enum field type giving the type of the binary (`UNKNOWN`, `SOUND` or `IMAGE`), and an union of a `USound` and `UIImage` struct containing a pointer to the data, the size of the data and type-specific meta-information.

3.12.1 UVar conversion and memory management

The `UBinary` manages its memory: when destroyed (or going out-of-scope), it frees all its allocated data. The `USound` and `UIImage` do not.

Reading an `UBinary` from a `UVar`, and writing a `UBinary`, `USound` or `UIImage` to an `UVar` performs a deep-copy of the data (by default, see below).

Reading a `USound` or `UIImage` from an `UVar` performs a shallow copy. Modifying the data is not allowed in that case.

3.12.2 0-copy mode

In plugin mode, you can setup any `UVar` in 0-copy mode by calling `setBypass(true)`. In this mode, binary data written to the `UVar` is not copied, but a reference is kept. As a consequence, the data is only available from within registered `notifyChange` callbacks. Those callbacks can use `UVar::val()` or cast the `UVar` to a `UBinary &` to retrieve the reference. Attempts to read the `UVar` from outside `notifyChange` will result in `nil` being returned.

3.13 Using hubs to group objects

Sometimes, you need to perform actions for a group of `UObjects`, for instance devices that need to be updated together. The API provides the `UObjectHub` class for this purpose. To create a hub, simply declare a subclass of `UObjectHub`, and register it by calling once the macro `UStartHub (class-name)`. A single instance of this class will then be created upon server start-up. `UObject` instances can then register to this hub by calling `URegister (hub-class-name)`. Timers can be attached to `UObjectHub` the same way as to `UObject` (see [Section 3.7](#)). The kernel will call the `update()` method of all `UObject` before calling the `update()` method of the hub. A hub instance can be retrieved by calling `getUObjectHub (string classname)`. The hub also holds the list of registered `UObject` in its `members` attribute.

3.14 Sending Urbi code

If you need to send Urbi code to the server, the `URBI()` macro is available, as well as the `send()` function. You can either pass it a string, or directly Urbi code inside a double pair of parentheses:

```
send ("myTag:1+1;");  
  
URBI (( at (myEvent?(var x)) { myTag:echo x; }; ));
```

You can also use the `call` method to make an urbiscript function call:

```
// C++ equivalent of urbiscript 'System.someFunc(12, "foo");'  
call("System", "someFunc", 12, "foo");
```

Chapter 4

Use Cases

4.1 Writing a Servomotor Device

Let's write a `UObject` for a servomotor device whose underlying API is:

- `bool initialize (int id)`
Initialize the servomotor with given ID.
- `double getPosition (int id)`
Read servomotor of given id position.
- `void setPosition (int id, double pos)`
Send a command to servomotor.
- `void setPID (int id, int p, int i, int d)`
Set P, I, and D arguments.

First our header. Our servo device provides an attribute named `val`, the standard Urbi name, and two ways to set PID gain: a method, and three variables.

```
class servo : public urbi::UObject // must inherit UObject
{
public:
    // the class must have a single constructor taking a string
    servo(const std::string&);

    // Urbi constructor
    void init(int id);

    // main attribute
    urbi::UVar val;

    // position variables:
    // P gain
    urbi::UVar P;
    // I gain
```

```

urbi::UVar I;
// D gain
urbi::UVar D;

// callback for val change
void valueChanged(UVar& v);
//callback for val access
void valueAccessed(UVar& v);
// callback for PID change
void pidChanged(UVar& v);

// method to change all values
void setPID(int p, int i, int d);

// motor ID
int id_;
};

```

The constructor only registers `init`, so that our default instance `servo` does nothing, and can only be used to create new instances.

```

servo::servo (const std::string& s)
: urbi::UObject (s)
{
    // register init
    UBindFunction (servo, init);
}

```

The `init` function, called in a new instance each time a new Urbi instance is created, registers the four variables (`val`, `P`, `I` and `D`), and sets up callback functions.

```

// Urbi constructor.
void
servo::init (int id)
{
    id_ = id;

    if (!initialize (id))
        return 1;

    UBindVar (servo, val);

    // val is both a sensor and an actuator.
    Uowned (val);

    // Set blend mode to mix.
    val.blend = urbi::UMIX;

    // Register variables.
    UBindVar (servo, P);
    UBindVar (servo, I);
    UBindVar (servo, D);

    // Register functions.
}

```

```

UBindFunction (servo, setPID);

// Register callbacks on functions.
UNotifyChange (val, &servo::valueChanged);
UNotifyAccess (val, &servo::valueAccessed);
UNotifyChange (P, &servo::pidChanged);
UNotifyChange (I, &servo::pidChanged);
UNotifyChange (D, &servo::pidChanged);
}

```

Then we define our callback methods. `servo::valueChanged` will be called each time the `val` variable is modified, just after the value is changed: we use this method to send our servo commands. `servo::valueAccessed` is called just before the value is going to be read. In this function we request the current value from the servo, and set `val` accordingly.

```

// Called each time val is written to.
void
servo::valueChanged (urbi::UVar& v)
{
    // v is a reference to our class member val: you can use both
    // indifferently.
    setPosition (id, (double)val);
}

// Called each time val is read.
void
servo::valueAccessed (urbi::UVar& v)
{
    // v is a reference to val.
    val = getPosition (id);
}

```

`servo::pidChanged` is called each time one of the PID variables is written to. The function `servo::setPID` can be called directly from Urbi.

```

void
servo::pidChanged (urbi::UVar& v)
{
    setPID(id, (int)P, (int)I, (int)D);
}

void
servo::setPID (int p, int i, int d)
{
    setPID (id, p, i, d);
    P = p;
    I = i;
    D = d;
}

// Register servo class to the Urbi kernel.
UStart (servo);

```

That's it, compile this module, and you can use it within urbiscript:

```

// Create a new instance.  Calls init (1).
headPan = new servo (1);

// Calls setPID ().
headPan.setPID (8,2,1);

// Calls valueChanged ().
headPan.val = 13;

// Calls valueAccessed ().
headPan.val * 12;

// Periodically calls valueChanged ().
headPan.val = 0 sin:1s ampli:20,

// Periodically calls valueAccessed ().
at (headPan.val < 0)
    echo ("left");

```

The sample code above has one problem: `valueAccessed` and `valueChanged` are called each time the value is read or written from Urbi, which can happen quite often. This is a problem if sending the actual command (`setPosition` in our example) takes time to execute. There are two solutions to this issue.

4.1.1 Caching

One solution is to remember the last time the value was read/written, and not apply the new command before a fixed time. Note that the kernel is doing this automatically for `UOwned`'d variables that are in a blend mode different than `normal`. So the easiest solution to the above problem is likely to set the variable to the `mix` blending mode. The unavoidable drawback is that commands are not applied immediately, but only after a small delay.

4.1.2 Using Timers

Instead of updating/fetching the value on demand, you can chose to do it periodically based on a timer. A small difference between the two API methods comes in handy for this case: the `update()` virtual method called periodically after being set up by `USetUpdate(interval)` is called just after one pass of Urbi code execution, whereas the timers set up by `USetTimer` are called just before one pass of Urbi code execution. So the ideal solution is to read your sensors in the second callback, and write to your actuators in the first. Our previous example (omitting PID handling for clarity) can be rewritten. The header becomes:

```

// Inherit from UObject.
class servo : public urbi::UObject
{
public:
    // The class must have a single constructor taking a string.
    servo (const std::string&)

```



```
// Urbi constructor.
void init (int id);

// Called periodically.
virtual int update ();
// Called periodically.
void getVal ();

// Our position variable.
urbi::UVar val;

// Motor ID.
int id_;
};
```

Constructor is unchanged, `init` becomes:

```
// Urbi constructor.
void
servo::init (int id)
{
    id_ = id;

    if (!initialize (id))
        return 0;

    UBindVar (servo,val);
    // Val is both a sensor and an actuator.
    UOwned(val);

    // Will call update () periodically.
    USetUpdate(1);
    // Idem for getVal ().
    USetTimer (1, &servo::getVal);
}
```

`valueChanged` becomes `update` and `valueAccessed` becomes `getVal`. Instead of being called on demand, they are now called periodically. The period of the call cannot be lower than the value returned by `Object.getPeriod`; so you can set it to 0 to mean “as fast as is useful”.

4.2 Using Hubs to Group Objects

Now, suppose that, for our previous example, we can speed things up by sending all the servo-motor commands at the same time, using the following method that takes two arrays of ids and positions.

```
void setPositions(int count, int* ids, double* positions);
```

A hub is the perfect way to handle this task. The `UObject` header stays the same. We add a hub declaration:

```
class servohub : public urbi::UObjectHub
```

```

{
public:
    // The class must have a single constructor taking a string.
    servohub (const std::string&);

    // Called periodically.
    virtual int update ();

    // Called by servo.
    void addValue (int id, double val);

    int* ids;
    double* vals;
    int size;
    int count;
};

```

`servo::update` becomes a call to the `addValue` method of the hub:

```

int
servo::update()
{
    ((servohub*)getUObjectHub ("servohub"))->addValue (id, (double)val);
};

```

The following line can be added to the `servo init` method, although it has no use in our specific example:

```

URegister(servohub);

```

Finally, the implementation of our hub methods is:

```

servohub::servohub (const std::string& s)
: UObjectHub (s)
, ids (0)
, vals (0)
, size (0)
, count (0)
{
    // setup our timer
    USetUpdate (1);
}

int
servohub::update ()
{
    // Called periodically.
    setPositions (count, ids, vals);

    // Reset position counter.
    count = 0;

    return 0;
}

```

```

void
servohub::addValue (int id, double val)
{
    if (count + 1 < size)
    {
        // Allocate more memory.
        ids = (int*) realloc (ids, (count + 1) * sizeof (int));
        vals = (double*) realloc (vals, (count + 1) * sizeof (double));
        size = count + 1;
    }
    ids[count] = id;
    vals[count++] = val;
}

UStartHub (servohub);

```

Periodically, the `update` method is called on each servo instance, which adds commands to the hub arrays, then the `update` method of the hub is called, actually sending the command and resetting the array.

4.2.1 Alternate Implementation

Alternatively, to demonstrate the use of the members `hub` variable, we can entirely remove the `update` method in the servo class (and the `USetUpdate()` call in `init`), and rewrite the hub `update` method the following way:

```

int servohub::update()
{
    //called periodically
    for (UObjectList::iterator i = members.begin ();
         i != members.end ();
         ++i)
        addValue (((servo*)*i)->id, (double)((servo*)*i)->val);
    setPositions(count, ids, vals);
    // reset position counter
    count = 0;

    return 0;
}

```

4.3 Writing a Camera Device

A camera device is an `UObject` whose `val` field is a binary object. The Urbi kernel itself doesn't make any difference between all the possible binary formats and data type, but the API provides image-specific structures for convenience. You must be careful about memory management. The `UBinary` structure handles its own memory: copies are deep, and the destructor frees the associated buffer. The `UImage` and `USound` structures do not.

Let's suppose we have an underlying camera API with the following functions:

- `bool initialize (int id)`

Initialize the camera with given ID.

- `int getWidth (int id)`

Return image width.

- `int getHeight (int id)`

Return image height.

- `char* getImage (int id)`

Get image buffer of format RGB24. The buffer returned is always the same and doesn't have to be freed.

Our device code can be written as follows:

```
// Inherit from UObject.
class Camera : public urbi::UObject
{
public:
    // The class must have a single constructor taking a string.
    Camera(const std::string&);

    // Urbi constructor. Throw in case of error.
    void init (int id);

    // Our image variable and dimensions.
    urbi::UVar val;
    urbi::UVar width;
    urbi::UVar height;

    // Called on access.
    void getVal (UVar&);

    // Called periodically.
    virtual int update ();

    // Frame counter for caching.
    int frame;
    // Frame number of last access.
    int accessFrame;
    // Camera id.
    int id_;
    // Storage for last captured image.
    UBinary bin;
};
```

The constructor only registers `init`:

```
Camera::Camera (const std::string& s)
: urbi::UObject (s)
, frame (0)
{
    UBindFunction (Camera, init);
}
```

The `init` function binds the variable, a function called on access, and sets a timer up on update. It also initializes the `UBinary` structure.

```
void
Camera::init (int id)
{
    //urbi constructor
    id_ = id;
    frame = 0;
    accessFrame = 0;

    if (!initialize (id))
        throw std::runtime_error("Failed to initialize camera");

    UBindVar (Camera, val);
    UBindVar (Camera, width);
    UBindVar (Camera, height);
    width = getWidth (id);
    height = getHeight (id);

    UNotifyAccess (val, &Camera::getVal);

    bin.type = BINARY_IMAGE;
    bin.image.width = width;
    bin.image.height = height;
    bin.image.imageFormat = IMAGE_RGB;
    bin.image.size = width * height * 3;

    // Call update () periodically.
    USetUpdate (1);
}
```

The `update` function simply updates the frame counter:

```
int
Camera::update ()
{
    ++frame;
    return 0;
}
```

The `getVal` updates the camera value, only if it hasn't already been called this frame, which provides a simple caching mechanism to avoid performing the potentially long operation of acquiring an image too often.

```
void
```

```

Camera::getVal(urbi::UVar&)
{
    if (frame == accessFrame)
        return;

    bin.image.data = getImage (id);
    // Assign image to bin.
    val = bin;
}

UStart(Camera);

```

The image data is copied inside the kernel when proceeding this way.

Be careful, suppose that we had created the `UBinary` structure inside the `getVal` method, our buffer would have been freed at the end of the function. To avoid this, set it to 0 after assigning the `UBinary` to the `UVar`.

4.3.1 Optimization in Plugin Mode

In plugin mode, it is possible to access the buffer used by the kernel by casting the `UVar` to a `UImage`. You can modify the content of the kernel buffer but no other argument.

4.4 Writing a Speaker or Microphone Device

Sound handling works similarly to image manipulation, the `USound` structure is provided for this purpose. The recommended way to implement a microphone is to fill the `UObject` val variable with the sound data corresponding to one kernel period. If you do so, the Urbi code `loop tag:micro.val`, will produce the expected result.

4.5 Writing a Softdevice: Ball Detection

Algorithms that require intense computation can be written in C++ but still be usable within Urbi: they acquire their data using `UVar` referencing other modules' variables, and output their results to other `UVar`. Let's consider the case of a ball detector device that takes an image as input, and outputs the coordinates of a ball if one is found.

The header is defined like:

```

class BallTracker : public urbi::UObject
{
public:
    BallTracker (const std::string&);
    void init (const std::string& varname);

    // Is the ball visible?
    urbi::UVar visible;

    // Ball coordinates.
    urbi::UVar x;

```

```
urbi::UVar y;  
};
```

The constructor only registers `init`:

```
// The constructor registers init only.  
BallTracker::BallTracker (const::string& s)  
: urbi::UObject (s)  
{  
    UBindFunction (BallTracker, init);  
}
```

The `init` function binds the variables and a callback on update of the image variable passed as a argument.

```
void  
BallTracker::init (const std::string& cameraval)  
{  
    UBindVar (BallTracker, visible);  
    UBindVar (BallTracker, x);  
    UBindVar (BallTracker, y);  
    UNotifyChange (cameraval, &BallTracker::newImage);  
  
    visible = 0;  
}
```

The `newImage` function runs the detection algorithm on the image in its argument, and updates the variables.

```
void  
BallTracker::newImage (urbi::UVar& v)  
{  
    // Cast to UImage.  
    urbi::UImage i = v;  
    int px,py;  
    bool found = detectBall (i.data, i.width, i.height, &px, &py);  
  
    if (found)  
    {  
        visible = 1;  
        x = px / i.width;  
        y = py / i.height;  
    }  
    else  
        visible = 0;  
}
```


Part II

urbiscript User Manual

About This Part

This part, also known as the “urbiscript tutorial”, teaches the reader how to program in urbiscript. It goes from the basis to concurrent and event-based programming. No specific knowledge is expected. There is no need for a C++ compiler, as `UObject` will not be covered here (see [Part I](#)). The reference manual contains a terse and complete definition of the Urbi environment ([Part IV](#)).

Chapter 5 — First Steps

First contacts with urbiscript.

Chapter 6 — Basic Objects, Value Model

A quick introduction to objects and values.

Chapter 7 — Flow Control Constructs

Basic control flow: `if`, `for` and the like.

Chapter 8 — Advanced Functions and Scoping

Details about functions, scoped, and lexical closures.

Chapter 9 — Objective Programming, urbiscript Object Model

A more in-depth introduction to object-oriented programming in urbiscript.

Chapter 10 — Functional Programming

Functions are first-class citizens.

Chapter 11 — Parallelism, Concurrent Flow Control

The urbiscript operators for concurrency, tags.

Chapter 12 — Event-based Programming

Support for event-driven concurrency in urbiscript.

Chapter 13 — Urbi for ROS Users

How to use ROS from Urbi, and vice-versa.

Chapter 5

First Steps

This section introduces the most basic notions to write urbiscript code. Some aspects are presented only minimally. The goal of this section is to bootstrap yourself with the urbiscript language, to be able to study more in-depth examples afterward.

5.1 Comments

Commenting your code is crucial, so let's start by learning how to do this in urbiscript. Comments are ignored by the interpreter, and can be left as documentation, reminder, ... urbiscript supports C and C++ style comments:

- C style comments start with `/*` and end with `*/`.
- C++ style comments start with `//` and last until the end of the line.

```
1; // This is a C++ style comment.  
[00000000] 1  
2 + /* This is a C-style comment. */ 2;  
[00000000] 4  
/* Contrary to C/C++, this type of comment /* does nest */. */  
3;  
[00000000] 3
```

5.2 Literal values

As already seen, we can evaluate literal integers. urbiscript supports several other literals, such as:

floats floating point numbers.

strings character strings.

lists ordered collection of values.

dictionary unordered collection of associations.

nil neutral value. Think of it as the value that fits anywhere.

void absence of value. Think of it as the value that fits nowhere.

These literal values can be obtained with the syntax presented below.

```
42; // Integer literal.
[00000000] 42
3.14; // Floating point number literal.
[00000000] 3.14
"string"; // Character string literal.
[00000000] "string"
[1, 2, "a", "b"]; // List literal.
[00000000] [1, 2, "a", "b"]
["a" => 1, "b" => 2]; // Dictionary literal.
[00000000] ["a" => 1, "b" => 2]
nil;
void;
```

This listing highlights some point:

- Lists and Dictionaries in urbiscript are heterogeneous. That is, one list can hold values of different types.
- The printing of nil and void is empty.

5.3 Function calls

You can call functions with the classical, mathematical notation.

```
cos(0); // Compute cosine
[00000000] 1
max(1, 3); // Get the maximum of the arguments.
[00000000] 3
max(1, 3, 4, 2);
[00000000] 4
```

Again, the result of the evaluation are printed out. You can see here that function in urbiscript can be variadic, that is, take different number of arguments, such as the `max` function. Let's now try the `echo` function, that prints out its argument.

```
echo("Hello world!");
[00000000] *** Hello world!
```

The server prints out `Hello world!`, as expected. Note that this output is still prepended with the time stamp. Since `echo` returns void, no evaluation result is printed.

5.4 Variables

Variables can be introduced with the `var` keyword, given a name and an initial value. They can be assigned new values with the `=` operator.

```
var x = 42;
[00000000] 42
echo(x);
[00000000] *** 42
x = 51;
[00000000] 51
x;
[00000000] 51
```

Note that, just as in C++, assignments return the (right-hand side) value, so you can write code like “`x = y = 0`”. The rule for valid identifiers is also the same as in C++: they may contain alphanumeric characters and underscores, but they may not start with a digit.

You may omit the initialization value, in which case it defaults to `void`.

```
var y;
y;
// Remember, the interpreter remains silent
// because void is printed out as nothing.
// You can convince yourself that y is actually
// void with the following methods.
y.asString;
[00000000] "void"
y.isVoid;
[00000000] true
```

5.5 Scopes

Scopes are introduced with curly brackets (`{}`). They can contain any number of statements. Variables declared in a scope only exist within this scope.

```
{
  var x = "test";
  echo(x);
};
[00000000] *** test
// x is no longer defined here
```

Note that the interpreter waits for the whole scope to be inputted to evaluate it. Also note the mandatory terminating semicolon after the closing curly bracket.

5.6 Method calls

Methods are called on objects with the dot (`.`) notation as in C++. Method calls can be chained. Methods with no arguments don’t require the parentheses.

```
0.cos();
[00000000] 1
"a-b-c".split("-");
[00000000] ["a", "b", "c"]
// Empty parentheses are optional
"foo".length();
[00000000] 3
"foo".length;
[00000000] 3
// Method call can be chained
"".length.cos;
[00000000] 1
```

In `obj.method`, we say that `obj` is the *target*, and that we are sending him the *method message*.

5.7 Function definition

You know how to call routines, let's learn how to write some. Functions can be declared thanks to the `function` keyword, followed by the comma separated, parentheses surrounded list of formal arguments, and the body between curly brackets.

```
// Define myFunction
function myFunction()
{
    echo("Hello world");
    echo("from my function!");
};
[00000000] function () {
    echo("Hello world");
    echo("from my function!");
}

// Invoke it
myFunction();
[00000000] *** Hello world
[00000000] *** from my function!
```

Note the strange output after you defined the function. `urbiscript` seems to be printing the function you just typed in again. This is because a function definition evaluates to the freshly created function.

Functions are first class citizen: they are values, just as `0` or `"foobar"`. The evaluation of a function definition yields the new function, and as always, the interpreter prints out the evaluation result, thus showing you the function again:

```
// Work in a scope.
{
    // Define f
    function f()
    {
```



```

    echo("f")
  };
  // This does not invoke f, it returns its value.
  f;
};
[00000000] function () {
  echo("f")
}
{
  // Define f
  function f()
  {
    echo("Hello World");
  };
  // This actually calls f
  f();
};
[00000000] *** Hello World

```

Here you can see that `f` is actually a simple value. You can just evaluate it to see its value, that is, its body. By adding the parentheses, you can actually call the function. This is a difference with methods calling, where empty parentheses are optional: method are always evaluated, you cannot retrieve their functional value — of course, you can with a different construct, but that's not the point here.

Since this output is often irrelevant, most of the time it is hidden in this documentation using the `| {};` trick (or even `|;`): when evaluating `code | {};`, the server first evaluates `code`, then evaluates `{}` and return its value, `void`, which prints to nothing.

```

function sum(a, b, c)
{
  return a + b + c;
} | {};
sum(20, 2, 20);
[00000000] 42

```

The `return` keyword enables to return a value from the function. If no `return` statement is executed, the evaluation of the last expression is returned.

```

function succ(i) { i + 1 } | {};
succ(50);
[00000000] 51

```

5.8 Conclusion

You're now up and running with basic urbiscript code, and we can dive in details into advanced urbiscript code.

Chapter 6

Basic Objects, Value Model

In this section, we focus on urbiscript values as objects, and study urbiscript by-reference values model. We won't study classes and actual objective programming yet, these points will be presented in [Chapter 9](#).

6.1 Objects in urbiscript

An object in urbiscript is a rather simple concept: a list of slots. A *slot* is a value associated to a name. So an *object* is a list of slot names, each of which indexes a value — just like a dictionary.

```
// Create a fresh object with two slots.
class Foo
{
  var a = 42;
  var b = "foo";
};
[00000000] Foo
```

The `localSlotNames` method lists the names of the slots of an object (`Object (??sec:std-Object)`).

```
// Inspect it.
Foo.localSlotNames;
[00000000] ["a", "asFoo", "b", "type"]
```

You can get an object's slot value by using the dot (`.`) operator on this object, followed by the name of the slot.

```
// We now know the name of its slots. Let's see their value.
Foo.a;
[00000000] 42
Foo.b;
[00000000] "foo"
```

It's as simple as this. The `inspect` method provides a convenient short-hand to discover an object (`Object (??sec:std-Object)`).

```

Foo.inspect;
[00000009] *** Inspecting Foo
[00000010] *** ** Prototypes:
[00000011] ***   Object
[00000012] *** ** Local Slots:
[00000014] ***   a : Float
[00000015] ***   asFoo : Code
[00000016] ***   b : String
[00000013] ***   type : String

```

Let's now try to build such an object. First, we want a fresh object to work on. In urbiscript, `Object` is the parent type of every object (in fact, since urbiscript is prototype-based, `Object` is the uppermost prototype of every object, but we'll talk about prototypes later). An instance of object, is an empty, neutral object, so let's start by instantiating one with the `clone` method of `Object`.

```

// Create the o variable as a fresh object.
var o = Object.clone;
[00000000] Object_0x00000000
// Check its content
o.inspect;
[00006725] *** Inspecting Object_0x00000000
[00006725] *** ** Prototypes:
[00006726] ***   Object
[00006726] *** ** Local Slots:

```

As you can see, we obtain an empty fresh object. Note that it still inherits from `Object` features that all objects share, such as the `localSlotNames` method.

Also note how `o` is printed out: `Object_`, followed by an hexadecimal number. Since this object is empty, its printing is quite generic: its type (`Object`), and its unique identifier (every urbiscript object has one). Since these identifiers are often irrelevant and might differ between two executions, they are often filled with zeroes in this document.

We're now getting back to our empty object. We want to give it two slots, `a` and `b`, with values `42` and `"foo"` respectively. We can do this with the `setSlot` method, which takes the slot name and its value.

```

o.setSlot("a", 42);
[00000000] 42
o.inspect;
[00009837] *** Inspecting Object_0x00000000
[00009837] *** ** Prototypes:
[00009837] ***   Object
[00009838] *** ** Local Slots:
[00009838] ***   a : Float

```

Here we successfully created our first slot, `a`. A good shorthand for setting slot is using the `var` keyword.

```

// This is equivalent to o.setSlot("b", "foo").
var o.b = "foo";
[00000000] "foo"

```

```
o.inspect;
[00072678] *** Inspecting Object_0x00000000
[00072678] *** ** Prototypes:
[00072679] ***   Object
[00072679] *** ** Local Slots:
[00072679] ***   a : Float
[00072680] ***   b : String
```

The latter form with `var` is preferred, but you need to know the name of the slot at the time of writing the code. With the former one, you can compute the slot name at execution time. Likewise, you can read a slot with a run-time determined name with the `getSlot` method, which takes the slot name as argument. The following listing illustrates the use of `getSlot` and `setSlot` to read and write slots whose names are unknown at code-writing time.

```
function set(object, name, value)
{
  // We have to use setSlot here, since we don't
  // know the actual name of the slot.
  return object.setSlot("x_" + name, value);
} |;

function get(object, name)
{
  // We have to use getSlot here, since we don't
  // know the actual name of the slot.
  return object.getSlot("x_" + name);
} |;

var x = Object.clone;
[00000000] Object_0x00000000
set(x, "foo", 0);
[00000000] 0
set(x, "bar", 1);
[00000000] 1
x.localSlotNames;
[00000000] ["x_bar", "x_foo"]
get(x, "foo");
[00000000] 0
get(x, "bar");
[00000000] 1
```

Right, now we can create fresh objects, create slots in them and read them afterward, even if their name is dynamically computed, with `getSlot` and `setSlot`. Now, you might wonder if there's a method to update the value of the slot. Guess what, there's one, and it's named...`updateSlot` (originality award). Getting back to our `o` object, let's try to update one of its slots.

```
o.a;
[00000000] 42
o.updateSlot("a", 51);
[00000000] 51
o.a;
[00000000] 51
```

Again, there's a shorthand for `updateSlot`: operator `=`.

```
o.b;
[00000000] "foo"
// Equivalent to o.updateSlot("b", "bar")
o.b = "bar";
[00000000] "bar"
o.b;
[00000000] "bar"
```

Likewise, prefer the '=' notation whenever possible, but you'll need `updateSlot` to update a slot whose name you don't know at code-writing time.

Note that defining the same slot twice, be it with `setSlot` or `var`, is an error. The slot must be defined once with `setSlot`, and subsequent writes must be done with `updateSlot`.

```
var o.c = 0;
[00000000] 0
// Can't redefine a slot like this
var o.c = 1;
[00000000:error] !!! slot redefinition: c
// Okay.
o.c = 1;
[00000000] 1
```

Finally, use `removeSlot` to delete a slot from an object.

```
o.localSlotNames;
[00000000] ["a", "b", "c"]
o.removeSlot("c");
[00000000] Object_0x00000000
o.localSlotNames;
[00000000] ["a", "b"]
```

Here we are, now you can inspect and modify objects at will. Don't hesitate to explore urbiscript objects you'll encounter through this documentation like this. Last point: reading, updating or removing a slot which does not exist is, of course, an error.

```
o.d;
[00000000:error] !!! lookup failed: d
o.d = 0;
[00000000:error] !!! lookup failed: d
```

6.2 Methods

Methods in urbiscript are simply object slots containing functions. We made a little simplification earlier by saying that `obj.slot` is equivalent to `obj.getSlot("slot")`: if the fetched value is executable code such as a function, the dot form evaluates it, as illustrated below. Inside a method, `this` gives access to the target — as in C++. It can be omitted if there is no ambiguity with local variables.

```
var o = Object.clone;
[00000000] Object_0x0
```

```
// This syntax stores the function in the 'f' slot of 'o'.
```

```
function o.f ()
{
  echo("This is f with target " + this);
  return 42;
} |;
// The slot value is the function.
o.getSlot("f");
[00000001] function () {
  echo("This is f with target " + (this));
  return 42;
}
// Huho, the function is invoked!
o.f;
[00000000] *** This is f with target Object_0x0
[00000000] 42
// The parentheses are in fact optional.
o.f();
[00000000] *** This is f with target Object_0x0
[00000000] 42
```

This was designed this way so as one can replace an attribute, such as an integer, with a function that computes the value. This enables to replace an attribute with a method without changing the object interface, since the parentheses are optional.

This implies that `getSlot` can be a better tool for object inspection to avoid invoking slots, as shown below.

```
// The 'empty' method of strings returns whether the string is empty.
"foo".empty;
[00000000] false
"".empty;
[00000000] true
// Using getSlot, we can fetch the function without calling it.
"".getSlot("asList");
[00000000] function () {
  split("")
}
```

The `asList` function simply bounces the task to `split`. Let's try `getSlot`'ing another method:

```
"foo".size;
[00000000] 3
"foo".getSlot("size");
[00000000] Primitive_0x0
```

The `size` method of `String` (`sec:std-String`) is another type of object: a `Primitive` (`sec:std-Primitive`). These objects are executable, like functions, but they are actually opaque primitives implemented in C++.

6.3 Everything is an object

If you're wondering what is an object and what is not, the answer is simple: every single bit of value you manipulate in urbiscript is an object, including primitive types, types themselves, functions, ...

```
var x = 0;
[00000000] 0
x.localSlotNames;
[00000000] []
var x.slot = 1;
[00000000] 1
x.localSlotNames;
[00000000] ["slot"]
x.slot;
[00000000] 1
x;
[00000000] 0
```

As you can see, integers are objects just like any other value.

6.4 The urbiscript values model

We are now going to focus on the urbiscript value model, that is how values are stored and passed around. The whole point is to understand when variables point to the same object. For this, we introduce `uid`, a method that returns the target's unique identifier — the same one that was printed when we evaluated `Object.clone`. Since uids might vary from an execution to another, their values in this documentation are dummy, yet not null to be able to differentiate them.

```
var o = Object.clone;
[00000000] Object_0x100000
o.uid;
[00000000] "0x100000"
42.uid;
[00000000] "0x200000"
42.uid;
[00000000] "0x300000"
```

Our objects have different uids, reflecting the fact that they are different objects. Note that entering the same integer twice (42 here) yields different objects. Let's introduce new operators before diving in this concept. First the equality operator: `==`. This operator is the exact same as C or C++'s one, it simply returns whether its two operands are *semantically* equal. The second operator is `===`, which is the *physical* equality operator. It returns whether its two operands are the same object, which is equivalent to having the same uid. This can seem a bit confusing; let's have an example.

```
var a = 42;
[00000000] 42
var b = 42;
```



```
[00000000] 42
a == b;
[00000000] true
a === b;
[00000000] false
```

Here, the `==` operator reports that `a` and `b` are equal — indeed, they both evaluate to 42. Yet, the `===` operator shows that they are not the same object: they are two different instances of integer objects, both equal 42.

Thanks to this operator, we can point out the fact that slots and local variables in urbiscript have a reference semantic. That is, when you defining a local variable or a slot, you're not copying any value (as you would be in C or C++), you're only making it refer to an already existing value (as you would in Ruby or Java).

```
var a = 42;
[00000000] 42
var b = 42;
[00000000] 42
var c = a; // c refers to the same object as a.
[00000000] 42
// a, b and c are equal: they have the same value.
a == b && a == c;
[00000000] true
// Yet only a and c are actually the same object.
a === b;
[00000000] false
a === c;
[00000000] true
```

So here we see that `a` and `c` point to the same integer, while `b` points to a second one. This is a non-trivial fact: any modification on `a` will affect `c` as well, as shown below.

```
a.localSlotNames;
[00000000] []
b.localSlotNames;
[00000000] []
c.localSlotNames;
[00000000] []
var a.flag; // Create a slot in a.
a.localSlotNames;
[00000000] ["flag"]
b.localSlotNames;
[00000000] []
c.localSlotNames;
[00000000] ["flag"]
```

Updating slots or local variables does not update the referenced value. It simply redirects the variable to the new given value.

```
var a = 42;
[00000000] 42
var b = a;
[00000000] 42
```

```
// b and a point to the same integer.
a === b;
[00000000] true
// Updating b won't change the referred value, 42,
// it makes it reference a fresh integer with value 51.
b = 51;
[00000000] 51
// Thus, a is left unchanged:
a;
[00000000] 42
```

Understanding the two latter examples is really important, to be aware of what your variable are referring to.

Finally, function and method arguments are also passed by reference: they can be modified by the function.

```
function test(arg)
{
  var arg.flag; // add a slot in arg
  echo(arg.uid); // print its uid
} |;
var x = Object.clone;
[00000000] Object_0x1
x.uid;
[00000000] "0x1"
test(x);
[00000000] *** 0x1
x.localSlotNames;
[00000000] ["flag"]
```

Beware however that arguments are passed by reference, and the behavior might not be what you may expected.

```
function test(arg)
{
  // Updates the local variable arg to refer 1.
  // Does not affect the referred value, nor the actual external argument.
  arg = 1;
} |;
var x = 0;
[00000000] 0
test(x);
[00000000] 1
// x wasn't modified
x;
[00000000] 0
```

6.5 Conclusion

You should now understand the reference semantic of local variables, slots and arguments. It's very important to keep them in mind, otherwise you will end up modifying variables you didn't

want, or change a copy of reference, failing to update the desired one.

Chapter 7

Flow Control Constructs

In this section, we'll introduce some flow control structures that will prove handy later. Most of them are inspired by C/C++.

7.1 if

The `if` construct is the same as C/C++'s one. The `if` keyword is followed by a condition between parentheses and an expression, and optionally the `else` keyword and another expression. If the condition evaluates to true, the first expression is evaluated. Otherwise, the second expression is evaluated if present.

```
if (true)
  echo("ok");
[00000000] *** ok
if (false)
  echo("ko")
else
  echo("ok");
[00000000] *** ok
```

The `if` construct is an expression: it has a value.

```
echo({ if (false) "a" else "b" });
[00000000] *** b
```

7.2 while

The `while` construct is, again, the same as in C/C++. The `while` keyword is followed by a condition between parentheses and an expression. If the condition evaluation is false, the execution jumps after the while block; otherwise, the expression is evaluated and control jumps before the while block.

```
var x = 2;
[00000000] 2
```

```
while (x < 40)
{
    x += 10;
    echo(x);
};
[00000000] *** 12
[00000000] *** 22
[00000000] *** 32
[00000000] *** 42
```

7.3 for

The `for` keyword supports different constructs, as in languages such as Java, C#, or even the forthcoming C++ revision.

The first construct is hardly more than syntactic sugar for a `while` loop.

```
for (var x = 2; x < 40; x += 10)
    echo(x);
[00000000] *** 2
[00000000] *** 12
[00000000] *** 22
[00000000] *** 32
```

The second construct allows to iterate over members of a collection, such as a list. The `for` keyword, followed by `var`, an identifier, a colon (or `in`), an expression and a scope, executes the scope for every element in the collection resulting of the evaluation of the expression, with the variable named with the identifier referring to the list members.

```
for (var e : [1, 2, 3]) { echo(e) };
[00000000] *** 1
[00000000] *** 2
[00000000] *** 3
```

7.4 switch

The syntax of the `switch` construct is similar to C/C++'s one, except it works on any kind of object, not only integral ones. Comparison is done by semantic equality (operator `==`). Execution will jump out of the `switch`-block after a case has been executed (no need to `break`). Also, contrary to C++, the whole construct has a value: that of the matching `case`.

```
switch ("bar")
{
    case "foo": 0;
    case "bar": 1;
    case "baz": 2;
    case "qux": 3;
};
[00000000] 1
```

7.5 do

A `do` scope is a shorthand to perform several actions on an object.

```
var o1 = Object.clone;
[00000000] Object_0x100000
var o1.one = 1;
[00000000] 1
var o1.two = 2;
[00000000] 2
echo(o1.uid);
[00000000] *** 0x100000
```

The same result can be obtained with a short `do` scope, that redirect method calls to their target, as in the listing below. This is similar to the Pascal “`with`” construct. The value of the `do`-block is the target itself.

```
var o2 = Object.clone;
[00000000] Object_0x100000
// All the message in this scope are destined to o.
do (o2)
{
  var one = 1; // var is a shortcut for the setSlot
  var two = 2; // message, so it applies on obj too.
  echo(uid);
};
[00000000] *** 0x100000
[00000000] Object_0x100000
```


Chapter 8

Advanced Functions and Scoping

This section presents advanced uses of functions and scoping, as well as their combo: lexical closures, which prove to be a very powerful tool.

8.1 Scopes as expressions

Contrary to other languages from the C family, scopes are expressions: they can be used where values are expected, just as `1 + 1` or `"foo"`. They evaluate to the value of their last expression, or `void` if they are empty. The following listing illustrates the use of scopes as expressions. Note that the last semicolon inside a scope is optional.

```
// Scopes evaluate to the last expression they contain.
{ 1; 2; 3};
[00000000] 3
// They are expressions.
echo({1; 2; 3});
[00000000] *** 3
```

8.2 Advanced scoping

Scopes can be nested. Variables can be redefined in sub-scopes. In this case, the inner variables hide the outer ones, as illustrated below.

```
var x = 0; // Define the outer x.
[00000000] 0
{
  var x = 1; // Define an inner x.
  x = 2;     // These refer to
  echo(x);   // the inner x
};
[00000000] *** 2
x;           // This is the outer x again.
[00000000] 0
{
```

```

x = 3;      // This is still the outer x.
echo(x);
};
[000000000] *** 3
x;
[000000000] 3

```

8.3 Local functions

Functions can be defined anywhere local variables can — that is, about anywhere. These functions’ visibility are limited to the scope they’re defined in, like variables. This enables for instance to write local helper functions like “max2” in the example below.

```

function max3(a, b, c) // Max of three values
{
  function max2(a, b)
  {
    if (a > b)
      return a
    else
      return b
  };
  max2(a, max2(b, c));
} | {};

```

8.4 Lexical closures

A *closure* is the capture by a function of a variable external to this function. urbiscript supports lexical closure: functions can refer to outer local variables, as long as they are visible (in scope) from where the function is defined.

```

function printSalaries(rate)
{
  var charges = 100;
  function computeSalary(hours)
  {
    // Here rate and charges are captured
    // from the environment by closure
    rate * hours - charges
  };

  echo("Alice's salary is " + computeSalary(35));
  echo("Bob's salary is " + computeSalary(30));
} | {};
printSalaries(15);
[000000000] *** Alice's salary is 425
[000000000] *** Bob's salary is 350

```

Closures can also write to captured variables, as shown below.

```
var a = 0;
[00000000] 0
var b = 0;
[00000000] 0
function add(n)
{
    // x and y are updated by closure
    a += n;
    b += n;
    void
} | {};
add(25);
add(25);
add(1);
a;
[00000000] 51
b;
[00000000] 51
```

Closure can be really powerful tools in some situation, and they are even more useful when combined with functional programming, as described in [Chapter 10](#).

Chapter 9

Objective Programming, urbiscript Object Model

This section presents object programing in urbiscript: the prototype-based object model of urbiscript, and how to define and use classes.

9.1 Prototype-based programing in urbiscript

You're probably already familiar with class-based object programing, since this is the C++ model. Classes and objects are very different entities. Types are static entities that do not exist at run-time, while objects are dynamic entities that do not exist at compile time.

Prototype-based object programing is different: the difference between classes and objects, between types and values, is blurred. Instead, you have an object, that is already an instance, and that you might clone to obtain a new one that you can modify afterward. Prototype-based programming was introduced by the Self language, and is used in several popular script languages such as Io or JavaScript.

Class-based programming can be considered with an industrial metaphor: classes are molds, from which objects are generated. Prototype-based programming is more biological: a prototype object is cloned into another object which can be modified during its lifetime.

Consider pairs for instance (see [Pair \(??sec:std-Pair\)](#)). Pairs hold two values, **first** and **second**, like an `std::pair` in C++. Since urbiscript is prototype-based, there is no pair class. Instead, **Pair** is really a pair (object).

```
Pair;  
[00000000] (nil, nil)
```

We can see here that **Pair** is a pair whose two values are equal to **nil** — which is a reasonable default value. To get a pair of our own, we simply clone **Pair**. We can then use it as a regular pair.

```
var p = Pair.clone;  
[00000000] (nil, nil)  
p.first = "101010";
```

```
[00000000] "101010"
p.second = true;
[00000000] true
p;
[00000000] ("101010", true)
Pair;
[00000000] (nil, nil)
```

Since `Pair` is a regular pair object, you can modify and use it at will. Yet this is not a good idea, since you will alter your base prototype, which alters any derivative, future and even past.

```
var before = Pair.clone;
[00000000] (nil, nil)
Pair.first = false;
[00000000] false
var after = Pair.clone;
[00000000] (false, nil)
before;
[00000000] (false, nil)
// before and after share the same first: that of Pair.
assert(Pair.first === before.first);
assert(Pair.first === after.first);
```

9.2 Prototypes and slot lookup

In prototype-based language, *is-a* relations (being an instance of some type) and inheritance relations (extending another type) are simplified in a single relation: prototyping. You can inspect an object prototypes with the `protos` method.

```
var p = Pair.clone;
[00000000] (nil, nil)
p.protos;
[00000000] [(nil, nil)]
```

As expected, our fresh pair has one prototype, `(nil, nil)`, which is how `Pair` displays itself. We can check this as presented below.

```
// List.head returns the first element.
p.protos.head;
[00000000] (nil, nil)
// Check that the prototype is really Pair.
p.protos.head === Pair;
[00000000] true
```

Prototypes are the base of the slot lookup mechanism. Slot lookup is the action of finding an object slot when the dot notation is used. So far, when we typed `obj.slot`, `slot` was always a slot of `obj`. Yet, this call can be valid even if `obj` has no `slot` slot, because slots are also looked up in prototypes. For instance, `p`, our clone of `Pair`, has no `first` or `second` slots. Yet, `p.first` and `p.second` work, because these slots are present in `Pair`, which is `p`'s prototype. This is illustrated below.

```

var p = Pair.clone;
[00000000] (nil, nil)
// p has no slots of its own.
p.localSlotNames;
[00000000] []
// Yet this works.
p.first;
// This is because p has Pair for prototype, and Pair has a 'first' slot.
p.protos.head === Pair;
[00000000] true
"first" in Pair.localSlotNames && "second" in Pair.localSlotNames;
[00000000] true

```

As shown here, the `clone` method simply creates an empty object, with its target as prototype. The new object has the exact same behavior as the cloned one thanks to slot lookup.

Let's experience slot lookup by ourselves. In `urbiscript`, you can add and remove prototypes from an object thanks to `addProto` and `removeProto`.

```

// We create a fresh object.
var c = Object.clone;
[00000000] Object_0x1
// As expected, it has no 'slot' slot.
c.slot;
[00000000:error] !!! lookup failed: slot
var p = Object.clone;
[00000000] Object_0x2
var p.slot = 0;
[00000000] 0
c.addProto(p);
[00000000] Object_0x1
// Now, 'slot' is found in c, because it is inherited from p.
c.slot;
[00000000] 0
c.removeProto(p);
[00000000] Object_0x1
// Back to our good old lookup error.
c.slot;
[00000000:error] !!! lookup failed: slot

```

The slot lookup algorithm in `urbiscript` is a depth-first traversal of the object prototypes tree. Formally, when the s slot is requested from x :

- If x itself has the slot, the requested value is found.
- Otherwise, the same lookup algorithm is applied on all prototypes, most recent first.

Thus, slots from the last prototype added take precedence over other prototype's slots.

```

var proto1 = Object.clone;
[00000000] Object_0x10000000
var proto2 = Object.clone;
[00000000] Object_0x20000000

```

```

var o = Object.clone;
[00000000] Object_0x30000000
o.addProto(proto1);
[00000000] Object_0x30000000
o.addProto(proto2);
[00000000] Object_0x30000000
// We give o an x slot through proto1.
var proto1.x = 0;
[00000000] 0
o.x;
[00000000] 0
// proto2 is visited first during lookup.
// Thus its "x" slot takes precedence over proto1's.
var proto2.x = 1;
[00000000] 1
o.x;
[00000000] 1
// Of course, o's own slots have the highest precedence.
var o.x = 2;
[00000000] 2
o.x;
[00000000] 2

```

You can check where in the prototype hierarchy a slot is found with the `locateSlot` method. This is a very handy tool when inspecting an object.

```

var p = Pair.clone;
[00000000] (nil, nil)
// Check that the 'first' slot is found in Pair
p.locateSlot("first") === Pair;
[00000000] true
// Where does locateSlot itself come from? Object itself!
p.locateSlot("locateSlot");
[00000000] Object

```

The prototype model is rather simple: creating a fresh object simply consists in cloning a model object, a prototype, that was provided to you. Moreover, you can add behavior to an object at any time with a simple `addProto`: you can make any object a fully functional `Pair` with a simple `myObj.addProto(Pair)`.

9.3 Copy on write

One point might be bothering you though: what if you want to update a slot value in a clone of your prototype?

Say we implement a simple prototype, with an `x` slot equal to 0, and clone it twice. We have three objects with an `x` slot, yet only one actual 0 integer. Will modifying `x` in one of the clone change the prototype's `x`, thus altering the prototype and the other clone as well?

The answer is, of course, no, as illustrated below.

```

var proto = Object.clone;
[00000000] Object_0x1

```



```

var proto.x = 0;
[00000000] 0
var o1 = proto.clone;
[00000000] Object_0x2
var o2 = proto.clone;
[00000000] Object_0x3
// Are we modifying proto's x slot here?
o1.x = 1;
[00000000] 1
// Obviously not
o2.x;
[00000000] 0
proto.x;
[00000000] 0
o1.x;
[00000000] 1

```

This work thanks to copy-on-write: slots are first duplicated to the local object when they're updated, as we can check below.

```

// This is the continuation of previous example.

// As expected, o2 finds "x" in proto
o2.locateSlot("x") === proto;
[00000000] true
// Yet o1 doesn't anymore
o1.locateSlot("x") === proto;
[00000000] false
// Because the slot was duplicated locally
o1.locateSlot("x") === o1;
[00000000] true

```

This is why, when we cloned `Pair` earlier, and modified the “first” slot of our fresh `Pair`, we didn't alter `Pair` one all its other clones.

9.4 Defining pseudo-classes

Now that we know the internals of urbiscript's object model, we can start defining our own classes.

But wait, we just said there are no classes in prototype-based object-oriented languages! That is true: there are no classes in the sense of C++: compile-time entities that are not objects. Instead, prototype-based languages rely on the existence of a canonical object (the *prototype*) from which (pseudo) *instances* are derived. Yet, since the syntactic inspiration for urbiscript comes from languages such as Java, C++ and so forth, it is nevertheless the `class` keyword that is used to define the pseudo-classes, i.e., prototypes.

As an example, we define our own `Pair` class. We just have to create a pair, with its `first` and `second` slots. For this we use the `do` scope described in [Section 7.5](#). The listing below defines a new `Pair` class. The `asString` function is simply used to customize pairs printing — don't give it too much attention for now.

```

var MyPair = Object.clone;
[00000000] Object_0x1
do (MyPair)
{
    var first = nil;
    var second = nil;
    function asString ()
    {
        "MyPair: " + first + ", " + second
    };
} | {};
// We just defined a pair
MyPair;
[00000000] MyPair: nil, nil
// Let's try it out
var p = MyPair.clone;
[00000000] MyPair: nil, nil
p.first = 0;
[00000000] 0
p;
[00000000] MyPair: 0, nil
MyPair;
[00000000] MyPair: nil, nil

```

That's it, we defined a pair that can be cloned at will! urbiscript provides a shorthand to define classes as we did above: the `class` keyword.

```

class MyPair
{
    var first = nil;
    var second = nil;
    function asString() { "(" + first + ", " + second + ")"; };
};
[00000000] (nil, nil)

```

The `class` keyword simply creates `MyPair` with `MyPairObject.clone`—, and provides you with a `do (MyPair)` scope. It actually also pre-defines a few slots, but this is not the point here.

9.5 Constructors

As we've seen, we can use the `clone` method on any object to obtain an identical object. Yet, some classes provide more elaborate constructors, accessible by calling `new` instead of `clone`, potentially passing arguments.

```

var p = Pair.new("foo", false);
[00000000] ("foo", false)

```

While `clone` guarantees you obtain an empty fresh object inheriting from the prototype, `new` behavior is left to the discretion of the cloned prototype — although its behavior is the same as `clone` by default.

To define such constructors, prototypes only need to provide an `init` method, that will be called with the arguments given to `new`. For instance, we can improve our previous `Pair` class with a constructor.

```
class MyPair
{
  var first = nil;
  var second = nil;
  function init(f, s) { first = f;  second = s;  };
  function asString() { "(" + first + ", " + second + ")"; };
};
[00000000] (nil, nil)
MyPair.new(0, 1);
[00000000] (0, 1)
```

9.6 Operators

In urbiscript, operators such as `+`, `&&` and others, are regular function that benefit from a bit of syntactic sugar. To be more precise, `a+b` is exactly the same as `a.'+'(b)`. The rules to resolve slot names apply too, i.e., the `'+'` slot is looked for in `a`, then in its prototypes.

The following example provides arithmetic between pairs.

```
class ArithPair
{
  var first = nil;
  var second = nil;
  function init(f, s) { first = f;  second = s;  };
  function asString() { "(" + first + ", " + second + ")"; };
  function '+'(rhs) { new(first + rhs.first, second + rhs.second); };
  function '-'(rhs) { new(first - rhs.first, second - rhs.second); };
  function '*'(rhs) { new(first * rhs.first, second * rhs.second); };
  function '/'(rhs) { new(first / rhs.first, second / rhs.second); };
};
[00000000] (nil, nil)
ArithPair.new(1, 10) + ArithPair.new(2, 20) * ArithPair.new(3, 30);
[00000000] (7, 610)
```


Chapter 10

Functional Programming

urbiscript support functional programming through first class functions and lambda expressions.

10.1 First class functions

urbiscript has first class functions, i.e., functions are regular values, just like integers or strings. They can be stored in variables, passed as arguments to other functions, and so forth. For instance, you don't need to write `function object.f(){/* ... */}` to insert a function in an object, you can simply use `setSlot`.

```
var o = Object.clone | {};
// Here we can use f as any regular value
o.setSlot("m1", function () { echo("Hello") }) | {};
// This is strictly equivalent
var o.m2 = function () { echo("Hello") } | {};
o.m1;
[00000000] *** Hello
o.m2;
[00000000] *** Hello
```

This enables to write powerful pieces of code, like functions that take function as argument. For instance, consider the `all` function: given a list and a function, it applies the function to each element of the list, and returns whether all calls returned true. This enables to check very simply if all elements in a list verify a predicate.

```
function all(list, predicate)
{
  for (var elt : list)
    if (!predicate(elt))
      return false;
  return true;
} | {};
// Check if all elements in a list are positive.
function positive(x) { x >= 0 } | {};
all([1, 2, 3], getSlot("positive"));
[00000000] true
```

```
all([1, 2, -3], getSlot("positive"));
[00000000] false
```

It turns out that `all` already exists: instead of `all(list, predicate)`, use `list.all(predicate)`, see [List \(??sec:std-List\)](#).

10.2 Lambda functions

Another nice feature is the ability to write lambda functions, which are anonymous functions. You can create a functional value as an expression, without naming it, with the syntax shown below.

```
// Create an anonymous function
function (x) {x + 1} | {};
// This enable to easily pass function
// to our "all" function:
[1, 2, 3].all(function (x) { x > 0});
[00000000] true
```

In fact, the `function` construct we saw earlier is only a shorthand for a variable assignment.

```
// This ...
function obj.f (/*...*/) {/*...*/};
// ... is actually a shorthand for this
var obj.f = function (/*...*/) {/* ... */};
```

10.3 Lazy arguments

Most popular programming languages use strict arguments evaluation: arguments are evaluated before functions are called. Other languages use lazy evaluation: argument are evaluated by the function only when needed. In `urbiscript`, evaluation is strict by default, but you can ask a function not to evaluate its arguments, and do it by hand. This works by not specifying formal arguments. The function is provided with a `call` object that enables you to evaluate arguments.

```
// Note the lack of formal arguments specification
function first
{
  // Evaluate only the first argument.
  call.evalArgAt(0);
} | {};
first(echo("first"), echo("second"));
[00000000] *** first
function reverse
{
  call.evalArgAt(1);
  call.evalArgAt(0);
} | {};
reverse(echo("first"), echo("second"));
```

```
[00000000] *** second  
[00000000] *** first
```

A good example are logic operators. Although C++ is a strict language, it uses a few logic operators. For instance, the logical and (`&&`) does not evaluate its right operand if the left operand is false (the result will be false anyway).

urbiscript logic operator mimic this behavior. The listing below shows how one can implement such a behavior.

```
function myAnd  
{  
  if (call.evalArgAt(0))  
    call.evalArgAt(1)  
  else  
    false;  
};  
  
function f()  
{  
  echo("f executed");  
  return true;  
};  
  
myAnd(false, f());  
[00000000] false  
  
myAnd(true, f());  
[00000000] *** f executed  
[00000000] true
```


Chapter 11

Parallelism, Concurrent Flow Control

Parallelism is a major feature of urbiscript. So far, all we've seen already existed in other languages — although we tried to pick, mix and adapt features and paradigms to create a nice scripting language. Parallelism is one of the corner stones of its paradigm, and what makes it so well suited to high-level scripting of interactive agents, in fields such as robotics or AI.

11.1 Parallelism operators

For now, we've separated our different commands with a semicolon (;). There are actually four statement separators in urbiscript:

- “;”: Serialization operator. Wait for the left operand to finish before continuing.
- “&”: Parallelism n-ary operator. All its operands are started simultaneously, and executed in parallel. The & block itself finishes when all the operands have finished. & has higher precedence than other separators.
- “,”: Background operator. Its left operand is started, and then it proceeds immediately to its right operand. This operator is bound to scopes: when used inside a scope, the scope itself finishes only when all the statements backgrounded with ‘,’ have finished.

The example below demonstrates the use of & to launch two functions in parallel.

```
function test(name)
{
  echo(name + ": 1");
  echo(name + ": 2");
  echo(name + ": 3");
} |;
// Serialized executions
test("left") ; test ("middle"); test ("right");
[00000000] *** left: 1
```

```

[00000000] *** left: 2
[00000000] *** left: 3
[00000000] *** middle: 1
[00000000] *** middle: 2
[00000000] *** middle: 3
[00000000] *** right: 1
[00000000] *** right: 2
[00000000] *** right: 3
// Parallel execution
test("left") & test("middle") & test ("right");
[00000000] *** left: 1
[00000000] *** middle: 1
[00000000] *** right: 1
[00000000] *** left: 2
[00000000] *** middle: 2
[00000000] *** right: 2
[00000000] *** left: 3
[00000000] *** middle: 3
[00000000] *** right: 3

```

In this test, we see that the `&` runs its operands simultaneously.
 The difference between “`&`” and “`,`” is rather subtle:

- In the top level, no operand of a job will start “`&`” until all are known. So if you send a line ending with “`&`”, the system will wait for the right operand (in fact, it will wait for a “`,`” or a “`;`”) before firing its left operand. A statement ending with “`,`” will be fired immediately.
- Execution is blocked after a “`&`” group until all its children have finished.

```

function test(name)
{
  echo(name + ": 1");
  echo(name + ": 2");
  echo(name + ": 3");
} | {};
// Run test and echo("right") in parallel,
// and wait until both are done before continuing
test("left") & echo("right"); echo("done");
[00000000] *** left: 1
[00000000] *** right
[00000000] *** left: 2
[00000000] *** left: 3
[00000000] *** done
// Run test in background, then both echos without waiting.
test("left") , echo("right"); echo("done");
[00000000] *** left: 1
[00000000] *** right
[00000000] *** left: 2
[00000000] *** done
[00000000] *** left: 3

```

That's about all there is to say about these operators. Although they're rather simple, they are really powerful and enables you to include parallelism anywhere at no syntactical cost.

11.2 Detach

The `Control.detach` function backgrounds the execution of its argument. Its behavior is the same as the comma `(,)` operator, except that the execution is completely detached, and not waited for at the end of the scope.

```
function test()
{
  // Wait for one second, and echo "foo".
  detach({sleep(1s); echo("foo")});
}

test();
echo("Not blocked");
[00000000] Job<shell_4>
[00000000] *** Not blocked
sleep(2s);
echo("End of sleep");
[00001000] *** foo
[00002000] *** End of sleep
```

11.3 Tags for parallel control flows

A **Tag** (`??sec:std-Tag`) is a multipurpose code execution control and instrumentation feature. Any chunk of code can be tagged, by preceding it with a tag and a colon `(:)`. Tag can be created with `Tag.new(name)`. Naming tags is optional, yet it's a good idea since it will be used for many features. The example below illustrates how to tag chunks of code.

```
// Create a new tag
var mytag = Tag.new("name");
[00000000] Tag<name>
// Tag the evaluation of 42
mytag: 42;
[00000000] 42
// Tag the evaluation of a block.
mytag: { "foo"; 51 };
[00000000] 51
// Tag a function call.
mytag: echo("tagged");
[00000000] *** tagged
```

You can use tags that were not declared previously, they will be created implicitly (see below). However, this is not recommended since tags will be created in a global scope, the `Tag` object. This feature can be used when inputting test code in the top level to avoid bothering to declare each tag, yet it is considered poor practice in regular code.

```
// Since mytag is not declared, this will first do:
// var Tag.mytag = Tag.new("mytag");
mytag : 42;
[00000000] 42
```

So you can tag code, yet what's the use? One of the primary purpose of tags is to be able to control the execution of code running in parallel. Tags have a few control methods (see [Tag](#) (`??sec:std-Tag`)):

freeze Suspend execution of all tagged code.

unfreeze Resume execution of previously frozen code.

stop Stop the execution of the tagged code. The flows of execution that where stopped jump immediately at the end of the tagged block.

block Block the execution of the tagged code, that is:

- Stop it.
- When an execution flow encounters the tagged block, it simply skips it.

You can think of **block** like a permanent **stop**.

unblock Stop blocking the tagged code.

The three following examples illustrate these features.

```
// Launch in background (using the comma) code that prints "ping"
// every second. Tag it to keep control over it.
mytag:
  every (1s)
    echo("ping"),
sleep(2.5s);
[00000000] *** ping
[00001000] *** ping
[00002000] *** ping
// Suspend execution
mytag.freeze;
// No printing anymore
sleep(1s);
// Resume execution
mytag.unfreeze;
sleep(1s);
[00007000] *** ping
```

```
// Now, we print out a message when we get out of the tag.
{
  mytag:
    every (1s)
      echo("ping");
  // Execution flow jumps here if mytag is stopped.
```

```

    echo("Background job stopped")|
},
sleep(2.5s);
[00000000] *** ping
[00001000] *** ping
[00002000] *** ping
// Stop the tag
mytag.stop;
[00002500] *** Background job stopped
// Our background job finished.
// Unfreezing the tag has no effect.
mytag.unfreeze;

// Now, print out a message when we get out of the tag.
loop
{
    echo("ping"); sleep(1s);
    mytag: { echo("pong"); sleep(1s); };
},
sleep(3.5s);
[00000000] *** ping
[00001000] *** pong
[00002000] *** ping
[00003000] *** pong

// Block printing of pong.
mytag.block;
sleep(3s);

// The second half of the while isn't executed anymore.
[00004000] *** ping
[00005000] *** ping
[00006000] *** ping

// Reactivate pong
mytag.unblock;
sleep(3.5s);
[00008000] *** pong
[00009000] *** ping
[00010000] *** pong
[00011000] *** ping

```

11.4 Advanced example with parallelism and tags

In this section, we implement a more advanced example with parallelism.

The listing below presents how to implement a `timeOut` function, that takes code to execute and a timeout as arguments. It executes the code, and returns its value. However, if the code execution takes longer than the given timeout, it aborts it, prints `"Timeout!"` and returns `void`. In this example, we use:

- Lazy evaluation, since we want to delay the execution of the given code, to keep control on it.
- Concurrency operators, to launch a timeout job in background.

```
// timeout (Code, Duration).
function timeout
{
  // In background, launch a timeout job that waits
  // for the given duration before aborting the function.
  // call.evalArgAt(1) is the second argument, the duration.
  {
    sleep(call.evalArgAt(1));
    echo("Timeout!");
    return;
  },
  // Run the Code and return its value.
  return call.evalArgAt(0);
} |;
timeout({sleep(1s); echo("On time"); 42}, 2s);
[00000000] *** On time
[00000000] 42
timeout({sleep(2s); echo("On time"); 42}, 1s);
[00000000] *** Timeout!
```

Chapter 12

Event-based Programming

When dealing with highly interactive agent programming, sequential programming is inconvenient. We want to react to external, random events, not execute code linearly with a predefined flow. `urbiscript` has a strong support for event-based programming.

12.1 Event related constructs

The first construct we will study is the `at` keyword. Given a condition, and an expression, `at` will evaluate the expression every time the condition becomes true. That is, when a rising edge occurs on the condition.

```
var x = 0;
[00000000] 0
at (x > 5)
  echo("ping");
x = 5;
[00000000] 5
// This triggers the event
x = 6;
[00000000] 6
[00000000] *** ping
// Does not trigger, since the condition is already true.
x = 7;
[00000000] 7
// The condition becomes false here.
x = 3;
[00000000] 3

x = 10;
[00000000] 10
[00000000] *** ping
```

An `onleave` block can be appended to execute an expression when the expression *becomes* false — that is, on falling edges.

```
var x = false;
```

```

[00000000] false
at (x)
  echo("x")
onleave
  echo("!x");
x = true;
[00000000] true
[00000000] *** x
x = false;
[00000000] false
[00000000] *** !x

```

See [Section 19.9.1](#) for more details on `at` statements.

The `whenever` construct is similar to `at`, except the expression evaluation is systematically restarted when it finishes as long as the condition stands true.

```

var x = false;
[00000000] false
whenever (x)
{
  echo("ping");
  sleep(1s);
};
x = true;
[00000000] true
sleep(3s);
// Whenever keeps triggering
[00000000] *** ping
[00001000] *** ping
[00002000] *** ping
x = false;
[00002000] false
// Whenever stops triggering

```

Just like `at` has `onleave`, `whenever` has `else`: the given expression is evaluated as long as the condition is false.

```

var x = false;
[00002000] false
whenever (x)
{
  echo("ping");
  sleep(1s);
}
else
{
  echo("pong");
  sleep(1s);
};
sleep (3s);
[00000000] *** pong
[00001000] *** pong
[00002000] *** pong
x = true;

```



```

[00003000] true
sleep (3s);
[00003000] *** ping
[00004000] *** ping
[00005000] *** ping
x = false;
[00006000] false
sleep (2s);
[00006000] *** pong
[00007000] *** pong

```

12.2 Events

urbiscript enables you to define events, that can be caught with the `at` and `whenever` constructs we saw earlier. You can create events by cloning the `Event` prototype. They can then be emitted with the `!` keyword.

```

var myEvent = Event.new;
[00000000] Event_0x0
at (myEvent?)
  echo("ping");
myEvent!;
[00000000] *** ping
// events work well with parallelism
myEvent! & myEvent!;
[00000000] *** ping
[00000000] *** ping

```

Both `at` and `whenever` have the same behavior on punctual events. However, if you emit an event for a given duration, `whenever` will keep triggering for this duration, contrary to `at`.

```

var myEvent = Event.new;
[00000000] Event_0x0
whenever (myEvent?)
{
  echo("ping (whenever)")|
  sleep(200ms)
};
at (myEvent?)
{
  echo("ping (at)")|
  sleep(200ms)
};
// Emit myEvent for .3 second.
myEvent! ~ 300ms;
[00000000] *** ping (whenever)
[00000100] *** ping (whenever)
[00000000] *** ping (at)

```


Chapter 13

Urbi for ROS Users

This chapter extends the ROS official tutorials (<http://www.ros.org/wiki/ROS/Tutorials>). Make sure to complete this tutorial before reading this document.

13.1 Communication on topics

First we will take back examples from topics, make sure talker and listener in ‘beginner_tutorial’ package are compiled. You can recompile it with the following command:

```
$ rosmake beginner_tutorial
```

13.1.1 Starting a process from Urbi

To communicate with ROS components, you need to launch them. You can do it without Urbi, or ask Urbi to start them for you. To launch new processes through Urbi, we will use the class `Process` ([??sec:std-Process](#)).

Let’s say we want to start `roscore`, and the talker of the beginner tutorial. Open an Urbi shell by typing the command ‘`rlwrap urbi -i`’. Here `rlwrap` makes ‘`urbi -i`’ acts like a shell prompt, with features like line editing, history, ...

```
var core = Process.new("roscore", []);
[000000001] Process roscore
var talker = Process.new("roscrun", ["beginner_tutorial", "talker"]);
[000000002] Process roscrun
core.run;
talker.run;
```

At this point, the processes are launched. The first argument of `Process.new` is the name of the command to launch, the second is a list of arguments.

Then you can check the status of the processes, get their stdout/stderr buffers, kill them in `urbiscript` (see [Process](#) ([??sec:std-Process](#))).

13.1.2 Listening to Topics

First you need to make sure that roscore is running, and the Ros module is loaded correctly:

```
Global.hasSlot("Ros");
[00016931] true
```

Then we can get the list of nodes launched through:

```
Ros.nodes;
```

This returns a [Dictionary \(??sec:std-Dictionary\)](#), with the name of the node as key, and a dictionary with topics subscribed, topics advertised, topics advertised as value.

We can check that our talker is registered, and on which channel it advertises:

```
// Get the structure (|; makes the instruction quiet).
var nodes = Ros.nodes|;

// List of nodes (keys).
nodes.keys;
[00000002] ["/rosout", "/urbi_1273060422295250703", "/talker"]

// Details of the node "talker".
nodes["talker"]["publish"];
[00000003] ["/rosout", "/chatter"]
```

Here we see that this node advertises `/rosout` and `/chatter`. Let's subscribe to `/chatter`:

```
// Initialize the subscription object.
var chatter = Ros.Topic.new("/chatter")|;
// Subscribe.
chatter.subscribe;
// This is the way we are called on new message.
chatTag: at(chatter.onMessage?(var e)) {
  // This will be executed on each message.
  echo(e)
};
```

In this code, `e` is a dictionary that follows the structure of the ROS message. Here is an example of what this code produces:

```
[00000004] *** ["data" => "Hello there! This is message [4]"]
[00000005] *** ["data" => "Hello there! This is message [5]"]
[00000006] *** ["data" => "Hello there! This is message [6]"]
```

We can also get a template for the message structure on this channel with:

```
chatter.structure;
[00000007] ["data" => ""]
```

To stop temporarily the echo, we take advantages of tags, by doing `chatTag.freeze`. Same thing goes with `unfreeze`. Of course you could also call `chatter.unsubscribe`, which unsubscribes you completely from this channel.

13.1.3 Advertising on Topics

To advertise a topic, this is roughly the same procedure.

13.1.3.1 Simple Talker

Here is a quick example:

```
// Initialize our object.
var talker = Ros.Topic.new("/chatter")|;
// Advertise (providing the ROS Type of this topic).
talker.advertise("/std_msgs/String");

// Get a template of our structure.
var msg = talker.structure.new;
msg["data"] = "Hello ROS world"|;
talker << msg;
```

We have just sent our first message to ROS, here if you launch the chatter, you will be able to get the message we have just sent.

The << operator is a convenient alias for `Ros.Topic.publish`.

13.1.3.2 Turtle Simulation

Now we are going to move the turtle with Urbi. First let's launch the turtle node:

```
var turtle = Process.new("roslaunch", ["turtlesim", "turtlesim_node"]|);
turtle.run;
```

With the help of `Ros.topics`, we can see that this turtle subscribes to a topic `'/turtle1/command_velocity'`. Let's advertise on it:

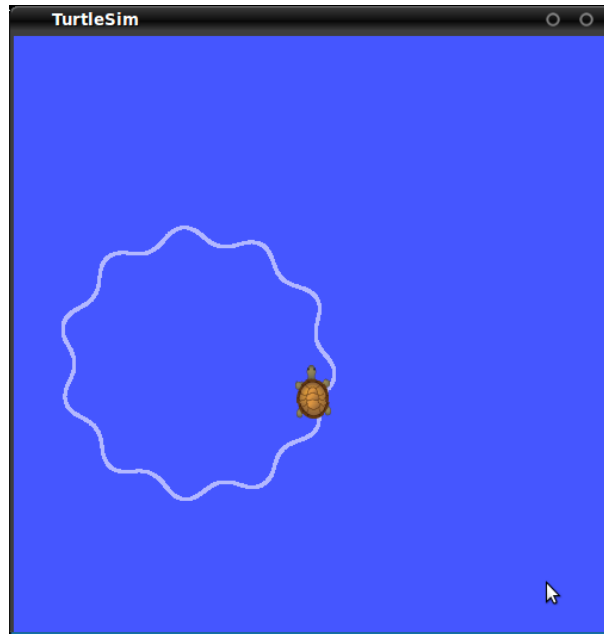
```
var velocity = Ros.Topic.new("/turtle1/command_velocity")|;
velocity.advertise("turtlesim/Velocity");
velocity.structure;
[00000001] ["linear" => 0, "angular" => 0]
```

Now we want to have it moving in circle with a small sinusoid wave. Here is the code:

```
// Get our template structure
var m = velocity.structure.new |;
m["linear"] = 0.8 |;
var angular = 0 |;
// Every time angular is changed, we send a message.
angTag: {
  at(angular->changed?) {
    m["angular"] = angular;
    velocity << m
  };
  // Generates a trajectory updated each System.period (default 20ms).
  angular = 0.3 sin: 2s ampli: 2
}; // Put this in background since the instruction above is blocking.
sleep(20);
angTag.freeze;
```

We won't cover this code in details, but the general principle is that `angular` is updated every 20ms with the values of a sinusoid wave trajectory with 0.3 as average value, 2 seconds for the period and 2 for the amplitude. See [TrajectoryGenerator](#) ([??sec:std-TrajectoryGenerator](#)) for more information.

Every time `angular` is changed, a new message is sent on the Topic `'/turtle1/command-velocity'`, thus updating the position of the turtle. After 20 seconds the tag is frozen, pausing the trajectory generation and the `at`.



13.2 Using Services

Services work the same way topics do, with minor differences.

Let's take back the turtle simulation example ([Section 13.1.3.2](#)). Then we can list the services available, and filter out loggers:

```
var r = Regexp.new("(get|set)_logger") |;
var services = Ros.services.keys |;
services.filter(closure (var item) { !r.match(item) });
[000000001] ["/clear", "/kill", "/turtle1/teleport_absolute", "/turtle1/teleport_relative", "/turtle1/set_pen", ...]
```

The use the `closure` construct allows us to keep the current “scope”, and thus, to keep our variable `r`.

Now we see that there is a service called `'/spawn'`. Here is the code to initialize its use:

```
var spawn = Ros.Service.new("/spawn", false) |;
waituntil(spawn.initialized);
```

The `new` function takes the service name as first argument, and whether or not the connection should be kept alive as second argument.

Since the creation of this object checks the service name, you should wait until `initialized` is `true` to use this service. You can also see the structure of the request through `spawn.reqStruct`, and the structure of the response by doing `spawn.resStruct`.

Now let's spawn a turtle called Jenny, at position (4, 4).

```
var req = spawn.reqStruct.new |;  
req["x"] = 4 |  
req["y"] = 4 |  
req["name"] = "Jenny" |;  
spawn.request(req);  
[00000001] ["name" => "Jenny"]
```


Part III

Guidelines and Cook Books

About This Part

This part contains guides to some specific aspects of Urbi SDK.

Chapter 14 — Installation

Complete instructions on how to install Urbi SDK.

Chapter 15 — Frequently Asked Questions

Some answers to common questions.

Chapter 16 — Migration from urbiscript 1 to urbiscript 2

This chapter is intended to people who want to migrate programs in urbiscript 1 to urbiscript 2.

Chapter 17 — Building Urbi SDK

Building Urbi SDK from the sources. How to install it, how to check it and so forth.

Chapter 14

Installation

14.1 Download

Various pre-compiled packages are provided. They are named

`'urbi-sdk-version-arch-os-compiler.ext'`

where

version specifies the exact revision of Urbi that you are using. It can be simple, 2.0, or more complex, 2.0-beta3-137-g28f8880. In that case,

2.0 is the version of the urbiscript Kernel,

beta3 designates the third pre-release,

137 is the number of changes since beta3 (not counting changes in sub-packages),

g28f8880 is a version control identifier, used internally to track the exact version that is being tested by our users.

arch describes the architecture, the CPU: **ARM**, **ppc**, or **x86**.

os is the operating system: **linux** for GNU/Linux, **osx** for Mac OS X, or **windows** for Microsoft Windows.

compiler is the tool chain used to compile the programs: **gcc4** for the GNU Compiler Collection 4.x, **vcxx2005** for Microsoft Visual C++ 2005, **vcxx2008** for Microsoft Visual C++ 2008.

ext is the package format extension: **tar.gz** for Unix style tar balls (uncompressed with **tar** **zxf** *tarfile*), and **zip** for Windows style zip files.

14.2 Install & Check

The package is *relocatable*, i.e., it does not need to be put at a specific location, nor does it need special environment variables to be set. It is not necessary to be a super-user to install it.

The *root* of the package, denoted by *urbi-root* hereafter, is the absolute name of the directory which contains the package.

After the install, the quickest way to test your installation is to run the various programs.

14.2.1 GNU/Linux and Mac OS X

Decompress the package where you want to install it:

```
$ rm -rf urbi-root
$ cd /tmp
$ tar xzf path-to/urbi-sdk-2.0-linux-x86-gcc4.tar.gz
$ mv urbi-sdk-2.0-linux-x86-gcc4 urbi-root
```

This directory, *urbi-root*, should contain ‘bin’, ‘FAQ.txt’ and so forth. Do not move things around inside this directory. In order to have an easy access to the Urbi programs, set up your PATH:

```
$ export PATH="urbi-root/bin:$PATH"
```

```
# Check that urbi is properly set up.
$ urbi --version

# Check that urbi-launch is properly installed.
$ urbi-launch --version
# Check that urbi-launch can find its dependencies.
$ urbi-launch -- --version

# Check that Urbi can compute.
$ urbi -e '1+2*3; shutdown;'
[00000175] 7
```

14.2.2 Windows

Decompress the zip file wherever you want or execute the installer.

Execute the script ‘urbi.bat’, located at the root of the uncompressed package. It should open a terminal with an interactive Urbi session.

Cygwin Issues

Inputs and outputs of windows native application are buffered under Cygwin. Thus, either running the interactive mode of Urbi or watching the output of the server under Cygwin is not recommended.

Chapter 15

Frequently Asked Questions

15.1 Build Issues

15.1.1 Complaints about ‘+=’

At random places we use ‘+=’ in `/bin/sh` scripts, which `Ash` (aka, `dash` and `sash`) does not support. Please, use `bash` or `zsh` instead of `Ash` as `/bin/sh`.

15.1.2 error: ‘anonymous;’ is used uninitialized in this function

If you encounter this error:

```
cc1plus: warnings being treated as errors
parser/ugrammar.hh: In member function \
    ‘void yy::parser::yypush_(const char*, int, yy::parser::symbol_type&)’:
parser/ugrammar.hh:1240: error: ‘<anonymous>’ is used uninitialized \
    in this function
parser/ugrammar.cc:1305: note: ‘<anonymous>’ was declared here
parser/ugrammar.hh: In member function \
    ‘void yy::parser::yypush_(const char*, yy::parser::stack_symbol_type&)’:
parser/ugrammar.hh:1240: error: ‘<anonymous>’ is used uninitialized \
    in this function
parser/ugrammar.cc:1475: note: ‘<anonymous>’ was declared here
```

then you found a problem that we don’t know how to resolved currently. Downgrade from `GCC-4.4` to `GCC-4.3`.

15.1.3 AM_LANGINFO_CODESET

If at bootstrap you have something like:

```
configure:12176: error: possibly undefined macro: AM_LANGINFO_CODESET
If this token and others are legitimate, please use m4_pattern_allow.
See the Autoconf documentation.
```

```
configure:12246: error: possibly undefined macro: gl_GLIBC21
```

it probably means your Automake installation is incomplete. See the Automake item in [Section 17.1](#).

15.1.4 ‘make check’ fails

Be sure to read [Section 17.9](#). In particular, run ‘make check’ several times (see [Section 17.9](#) to know why). If the failures remain, please submit the ‘test-suite.log’ file(s) (see [Section 15.5.3](#)).

15.2 Troubleshooting

15.2.1 Error 1723: "A DLL required for this install to complete could not be run."

This error is raised when you try to install a program like `vcredist-x86.exe`. This program use the “Windows Installer” which is probably outdated on your system.

To fix this problem, update the “Windows Installer” and re-start the installation of `vcredist` which should no longer fail.

15.2.2 When executing a program, the message “The system cannot execute the specified program.” is raised.

This library is necessary to start running any application. Run ‘`vcredist-x86.exe`’ to install the missing libraries.

If you have used the Urbi SDK installer, it is ‘`vcredist-x86.exe`’ in your install directory. Otherwise download it from the Microsoft web site. Be sure to get the one corresponding to the right Visual C++ version.

15.2.3 When executing a program, the message “This application has failed to start” is raised.

Same answer as [Section 15.2.2](#).

15.2.4 The server dies with “stack exhaustion”

Your program might be deeply recursive, or use large temporary objects. Use ‘`--stack-size`’ to augment the stack size, see [Section 18.3](#).

Note that one stack is allocated per “light thread”. This can explain why programs that heavily rely on concurrency might succeed where sequential programs can fail. For instance the following program is very likely to quickly exhaust the (single) stack.

```
function consume (var num)
{
  if (num)
```



```

    consume(num - 1) | consume(num - 1)
  }|;
consume (512);

```

But if you use `&` instead of `|`, then each recursive call to `consume` will be spawn with a fresh stack, and therefore none will run out of stack space:

```

function consume (var num)
{
  if (num)
    consume(num - 1) & consume(num - 1)
}|;
consume (512);

```

However your machine will run out of resources: this heavily concurrent program aims at creating no less than 2^{513} threads, about 2.68×10^{156} (a 156-digit long number, by far larger than the number of atoms in the observable universe, estimated to 10^{80}).

15.2.5 'myuobject: file not found'. What can I do?

If `urbi-launch` (or `urbi`) fails to load a UObject (a shared library or DLL) although the file exists, then the most probable cause is an undefined symbol in your shared library.

Getting a better diagnostic First, set the `GD_LEVEL` environment variable (see [Section 18.1.2](#)) to some high level, say `DUMP`, to log messages from `urbi-launch`. You might notice that your library is not exactly where you thought `urbi-launch` was looking at.

GNU/Linux A `ld` quirk prevents us from displaying a more accurate error message. You can use a tool named `ltrace` to obtain the exact error message. `Ltrace` is a standard package on most Linux distributions. Run it with `'ltrace -C -s 1024 urbi-launch ...'`, and look for lines containing `'dlerror'` in the output. One will contain the exact message that occurred while trying to load your shared library.

It is also useful to use `ldd` to check that the dependencies of your object are correct. See the documentation of `ldd` on your machine (`'man ldd'`). The following run is successful: every request (left-hand side of `=>`) is satisfied (by the file shown on the right-hand side).

```

$ all.so
  linux-gate.so.1 => (0xb7fe8000)
  libstdc++.so.6 => \
    /usr/lib/gcc/i686-pc-linux-gnu/4.4.1/libstdc++.so.6 (0xb7eba000)
  libm.so.6 => /lib/libm.so.6 (0xb7e94000)
  libc.so.6 => /lib/libc.so.6 (0xb7d51000)
  libgcc_s.so.1 => \
    /usr/lib/gcc/i686-pc-linux-gnu/4.4.1/libgcc_s.so.1 (0xb7d35000)
  /lib/ld-linux.so.2 (0xb7fe9000)

```

The following run shows a broken dependency.

```
# A simple C++ program.
$ echo 'int main() {}' >foo.cc

# Compile it, and depend on the libport shared library.
$ g++ foo.cc -Lurbi-root/gostai/lib -lport -o foo

# Run it.
$ ./foo
./foo: error while loading shared libraries: \
libport.so: cannot open shared object file: No such file or directory

# See that ldd is unhappy.
$ ldd foo
    linux-gate.so.1 => (0xb7fa4000)
    libport.so => not found
    libstdc++.so.6 => \
        /usr/lib/gcc/i686-pc-linux-gnu/4.4.1/libstdc++.so.6 (0xb7eae000)
    libm.so.6 => /lib/libm.so.6 (0xb7e88000)
    libgcc_s.so.1 => \
        /usr/lib/gcc/i686-pc-linux-gnu/4.4.1/libgcc_s.so.1 (0xb7e6c000)
    libc.so.6 => /lib/libc.so.6 (0xb7d29000)
    /lib/ld-linux.so.2 (0xb7fa5000)
```

Notice the ‘not found’ message. The shared object could not be loaded because it is not found in the *runtime path*, which is the list of directories where the system looks for shared objects to be loaded when running a program.

You may extend your `LD_LIBRARY_PATH` to include the missing directory.

```
$ export LD_LIBRARY_PATH=urbi-root/gostai/lib:$LD_LIBRARY_PATH
# Run it.
$ ./foo
```

Mac OS X Set the `DYLD_PRINT_LIBRARIES` environment variable to 1 to make the shared library loader report the libraries it loads on the standard error stream.

Use `otool` to check whether a shared object “finds” all its dependencies.

```
$ otool -L all.so
all.so:
    /usr/lib/libstdc++.6.dylib \
        (compatibility version 7.0.0, current version 7.4.0)
    /usr/lib/libgcc_s.1.dylib \
        (compatibility version 1.0.0, current version 1.0.0)
    /usr/lib/libSystem.B.dylib \
        (compatibility version 1.0.0, current version 111.1.4)
```

The following run shows a broken dependency.

```
# A simple C++ program.
$ echo 'int main() {}' >foo.cc

# Compile it, and depend on the libport shared library.
```

```

$ g++ foo.cc -Lurbi-root/gostai/lib -lport -o foo

# Run it.
$ ./foo
dyld: Library not loaded: @loader_path/libport.dylib
  Referenced from: /private/tmp/./foo
  Reason: image not found

# See that otool is unhappy.
$ otool -L ./foo
./foo:
    @loader_path/libport.dylib \
      (compatibility version 0.0.0, current version 0.0.0)
    /usr/lib/libstdc++.6.dylib \
      (compatibility version 7.0.0, current version 7.4.0)
    /usr/lib/libgcc_s.1.dylib \
      (compatibility version 1.0.0, current version 1.0.0)
    /usr/lib/libSystem.B.dylib \
      (compatibility version 1.0.0, current version 111.1.5)

```

The fact that the ‘libport.dylib’ was not found shows by the unresolved relative runtime-path: ‘@loader_path’ still shows. Use DYLD_LIBRARY_PATH to specify additional directories where the system should look for runtime dependencies.

```

$ DYLD_PRINT_LIBRARIES=1 \
  DYLD_LIBRARY_PATH=urbi-root/lib:$DYLD_LIBRARY_PATH \
  ./foo
dyld: loaded: /private/tmp/./foo
dyld: loaded: urbi-root/lib/libport.dylib
dyld: loaded: /usr/lib/libstdc++.6.dylib
dyld: loaded: /usr/lib/libgcc_s.1.dylib
dyld: loaded: /usr/lib/libSystem.B.dylib
dyld: loaded: urbi-root/lib/libboost_filesystem-mt.dylib
dyld: loaded: urbi-root/lib/libboost_signals-mt.dylib
dyld: loaded: urbi-root/lib/libboost_system-mt.dylib
dyld: loaded: urbi-root/lib/libboost_thread-mt.dylib
dyld: loaded: /usr/lib/system/libmathCommon.A.dylib
$

```

Windows In plain native Windows, use DependencyWalker (see <http://dependencywalker.com>) to check that a given DLL finds all its dependencies.

Alternatively, under Cygwin, you can use the `cygcheck.exe` program to check dependencies.

If you need to add a DLL that was missing, put it somewhere in the PATH, or in the current working directory.

15.3 urbiscript

15.3.1 Objects lifetime

15.3.1.1 How do I create a new Object derivative?

Urbi is based on prototypes. To create a new Object derivative (which will inherit all the Object methods), you can do:

```
var myObject = Object.new;  
[00000001] Object_0x76543210
```

15.3.1.2 How do I destroy an Object?

There is no `delete` in Urbi, for a number of reasons (see [Section 16.2](#)). Objects are deleted when they are no longer used/referenced to.

In practice, users who want to “delete an object” actually want to remove a slot — see [Section 6.1](#). Users who want to clear an object can empty it — see [Section 16.2](#).

Note that `myObject = nil` does not explicitly destroy the object bound to the name `myObject`, yet it may do so provided that `myObject` was the last and only reference to this object.

15.3.2 Slots and variables

15.3.2.1 Is the lobby a scope?

One frequently asked question is what visibility do variables have in urbiscript, especially when they are declared at the top-level interactive loop. In this section, we will see the mechanisms behind slots, local variables and scoping to fully explain this behavior and determine how to proceed to give the right visibility to variables.

For instance, this code might seem confusing at first:

```
var mind = 42;  
[00000001] 42  
function get()  
{  
  echo(mind);  
}|;  
get();  
[00000000] *** 42  
function Object.get()  
{  
  echo(mind)  
}|;  
// Where is my mind?  
Object.get;  
[00000000:error] !!! lookup failed: mind
```

Local variables, slots and targets The first point is to understand the difference between local variables and slots. Slots are simply object fields: a name in an object referring to another object, like members in C++. They can be defined with the `setSlot` method, or with the `var` keyword.

```
// Add an 'x' slot in Object, with value 51.
Object.setSlot("x", 51);
[00000000] 51
// This is an equivalent version, for the 'y' slot.
var Object.y = 51;
[00000000] 51

// We can access these slots with the dot operator.
Object.x + Object.y;
[00000000] 102
```

On the other hand, local variables are not stored in an object, but in the execution stack: their lifetime spans from their declaration point to the end of the current scope. They are declared with the `'var'` keyword.

```
function foo()
{
    // Declare an 'x' local variable, with value 51.
    var x = 51;
    // 'x' isn't stored in any object. It's simply
    // available until the end of the scope.
    echo(x);
};
```

You probably noticed that in the last two code snippets, we used the `var` keyword to declare both a slot in `Object` and a local variable. The rule is simple: `var` declares a slot if an owning object is specified with the dot notation, as in `var owner.slot`, and a local variable if only an unqualified name is given, as in `var name`.

```
{
    // Store a 'kyle' slot in Object.
    var Object.kyle = 42;
    // Declare a local variable, limited to this scope.
    var kenny = 42;
}; // End of scope.
[00000000] 42

// Kyle survived.
echo(Object.kyle);
[00000000] *** 42

// Oh my God, they killed Kenny.
echo(kenny);
[00000000:error] !!! lookup failed: kenny
```

There is however an exception to this rule: `do` and `class` scopes are designed to define a target where to store slots. Thus, in `do` and `class` scopes, even unqualified `var` uses declare slots in the target.

```
// Classical scope.
{
  var arm = 64; // Local to the scope.
};
[00000000] 64

// Do scope, with target Object
do (Object)
{
  var chocolate = 64; // Stored as a slot in Object.
};
[00000000] Object

// No arm...
echo(arm);
[00000000:error] !!! lookup failed: arm
// ... but still chocolate!
echo(chocolate);
[00000000] *** 64
```

Last tricky rule you must keep in mind: the top level of your connection — your interactive session — is a `do` (lobby) scope. That is, when you type `var x` directly in your connection, it stores an `x` slot in the lobby object. So, what is this *lobby*? It's precisely the object designed to store your top-level variables. Every Urbi server has a unique `Lobby` (`??sec:std-Lobby`) (note the capital), and every connection has its `lobby` that inherits the `Lobby`. Thus, variables stored in `Lobby` are accessible from any connection, while variables stored in a connection's `lobby` are local to this connection.

To fully understand how lobbies and the top-level work, we must understand how calls — message passing — work in urbiscript. In urbiscript, every call has a target. For instance, in `Object.x`, `Object` is the target of the `x` call. If no target is specified, as in `x` alone, the target defaults to `this`, yielding `this.x`. Knowing this rules, plus the fact that at the top-level `this` is `lobby`, we can understand better what happens when defining and accessing variables at the top-level:

```
// Since we are at the top-level, this stores x in the lobby.
// It is equivalent to 'var lobby.x'.
var x = "hello";
[00000000] "hello"

// This is an unqualified call, and is thus
// equivalent to 'this.x'.
// That is, 'lobby.x' would be equivalent.
x;
[00000000] "hello"
```

Solving the tricky example We now know all the scoping rules required to explain the behavior of the first code snippet. First, let's determine why the first access to `mind` works:

```
// This is equivalent to 'var lobby.myMind = 42'.
```

```

var myMind = 42;
[00000001] 42
// This is equivalent to 'function lobby.getMine...'
function getMine()
{
  // This is equivalent to 'echo(this.myMind)'
  echo(myMind);
};
// This is equivalent to 'this.getMine()', i.e. 'lobby.getMine()'.
getMine();
[00000000] *** 42

```

Step by step:

- We create a `myMind` slot in `lobby`, with value 42.
- We create a `getMine` function in `lobby`.
- We call the `lobby`'s `getMine` method.
- We access `this.myMind` from within the method. Since the method was called with `lobby` as `targetMine`, `this` is `lobby`, and `lobby.x` resolves to the previously defined 42.

We can also explain why the second test fails:

```

// Create the 'hisMind' slot in the lobby.
var hisMind = 42;
[00000000] 42
// Define a 'getHis' method in 'Object'.
function Object.getHis()
{
  // Equivalent to echo(this.hisMind).
  echo(hisMind)
};
// Call Object's getHis method.
Object.getHis;
[00000000:error] !!! lookup failed: hisMind

```

Step by step:

- We create a `hisMind` slot in `lobby`, with value 42, like before.
- We create a `getHis` function in `Object`.
- We call `Object`'s `getHis` method.

In the method, `this` is `Object`. Thus `hisMind`, which is `this.hisMind`, fails because `Object` has no such slot.

The key to understanding this behavior is that any unqualified call — unless it refers to a local variable — is destined to `this`. Thus, variables stored in the `lobby` are only accessible from the top-level, or from functions that are targeted on the `lobby`.

So, where to store global variables? From these rules, we can deduce a simple statement: since unqualified slots are searched in `this`, for a slot to be global, it must always be accessible through `this`. One way to achieve this is to store the slot in `Object`, the ancestor of any object:

```
var Object.global = 1664;
[00000000] 1664

function any_object()
{
  // This is equivalent to echo(this.global)
  echo(global);
} |;
```

In the previous example, typing `global` will look for the `global` slot in `this`. Since `this` necessarily inherits `Object`, it will necessarily be found.

This solution would work; however, storing all global variables in `Object` wouldn't be very clean. `Object` is rather designed to hold methods shared by all objects. Instead, a `Global` object exists. This object is a prototype of `Object`, so all his slots are accessible from `Object`, and thus from anywhere. So, creating a genuine global variable is as simple as storing it in `Global`:

```
var Global.g = "I'm global!";
[00000000] "I'm global!"
```

Note that you might want to reproduce the `Global` system and create your own object to store your related variables in a more tidy fashion. This is for instance what is done for mathematical constants:

```
// Store all constants here
class Constants
{
  var Pi = 3.14;
  var Euler = 2.17;
  var One = 1;
  // ...
} |;
// Make them global by making them accessible from Global.
Global.addProto(Constants);
[00000000] Global

// Test it.
Global.Pi;
[00000000] 3.14
Pi;
[00000000] 3.14
function Object.testPi() { echo(Pi) } |;
42.testPi;
[00000000] *** 3.14
```

15.3.2.2 How do I add a new slot in an object?

To add a slot to an object `O`, you have to use the `var` keyword, which is syntactic sugar for the `setSlot` method:


```

var O2 = Object.new |
// Syntax...
var O2.mySlot1 = 42;
[00000001] 42

// and semantics.
O2.setSlot("mySlot2", 23);
[00000001] 23

```

Note that in a method, `this` designates the current object. It is needed to distinguish the name of a slot in the current object, versus a local variable name:

```

{
  // Create a new slot in the current object.
  var this.bar = 42;

  // Create a local variable, which will not be known anymore
  // after we exit the current scope.
  var qux = 23;
}|
qux;
[00000001:error] !!! lookup failed: qux
bar;
[00000001] 42

```

15.3.2.3 How do I modify a slot of my object?

Use the `=` operator, which is syntactic sugar for the `updateSlot` method.

```

class O
{
  var mySlot = 42;
}|
// Sugarful.
O.mySlot = 51;
[00000001] 51

// Sugar-free.
O.updateSlot("mySlot", 23);
[00000001] 23

```

15.3.2.4 How do I create or modify a local variable?

Use `var` and `=`.

```

// In two steps: definition, and initial assignment.
var myLocalVariable;
myLocalVariable = "foo";
[00000001] "foo"
// In a single step: definition with an initial value.
var myOtherLocalVariable = "bar";

```

```
[00000001] "bar"
```

15.3.2.5 How do I make a constructor?

You can define a method called `init` which will be called automatically by `new`. For example:

```
class myObject
{
  function init(x, y)
  {
    var this.x = x;
    var this.y = y;
  };
};
myInstance = myObject.new(10, 20);
```

15.3.2.6 How can I manipulate the list of prototypes of my objects?

The `protos` method returns a list (which can be manipulated) containing the list of your object prototype.

```
var myObject = Object.new;
myObject.protos;
[00000001] [Object]
```

15.3.2.7 How can I know the slots available for a given object?

The `localSlotNames` and `allSlotNames` methods return respectively the local slot names and the local+inherited slot names.

15.3.2.8 How do I create a new function?

Functions are first class objects. That means that you can add them as any other slot in an object:

```
var myObject = Object.new;
var myObject.myFunction = function (x, y)
{ echo ("myFunction called with " + x + " and " + y) };
```

You can also use the following notation to add a function to your object:

```
var myObject = Object.new;
function myObject.myFunction (x, y) { /* ... */ };
```

or even group definitions within a `do` scope, which will automatically define new slots instead of local variables and functions:

```
var myObject = Object.new;  
do (myObject)  
{  
  function myFunction (x, y) { /* ... */ };  
};
```

or group those two statements by using a convenient `class` scope:

```
class myObject  
{  
  function myFunction (x, y) { /* ... */ };  
};
```

15.3.3 Tags

See [Section 11.3](#), in the urbiscript User Manual, for an introduction about Tags. Then for a definition of the Tag objects (construction, use, slots, etc.), see [Tag \(??sec:std-Tag\)](#).

15.3.3.1 How do I create a tag?

See [Section 20.61.2](#).

15.3.3.2 How do I stop a tag?

Use the `stop` method (see [Section 20.61.1.1](#)).

```
myTag.stop;
```

15.3.3.3 Can tagged statements return a value?

Yes, by giving it as a parameter to `stop`. See [Section 20.61.1.1](#).

15.3.4 Events

See [Chapter 12](#), in the urbiscript User Manual, for an introduction about event-based programming. Then for a definition of the Event objects (construction, use, slots, etc.), see [Event \(??sec:std-Event\)](#).

15.3.4.1 How do I create an event?

Events are objects, and must be created as any object by using `new` to create derivatives of the Event object.

```
var ev = Event.new;
```

See [Section 20.14.3](#).

15.3.4.2 How do I emit an event?

Use the `!` operator.

```
ev!(1, "foo");
```

15.3.4.3 How do I catch an event?

Use the `at(event?args)` construct (see [Section 19.9.1](#)).

```
at(ev?(1, var msg))  
  echo ("Received event with 1 and message " + msg);
```

The `?` marker indicates that we are looking for an event instead of a Boolean condition. The construct `var msg` indicates that the `msg` variable will be bound (as a local variable) in the body part of the `at` construct, with whatever value is present in the event that triggered the `at`.

15.3.5 Standard Library

15.3.5.1 How can I iterate over a list?

Use the `for` construct ([Section 19.6.5.2](#)), or the `each` method ([List \(??sec:std-List\)](#)):

```
for (var i: [10, 11, 12]) echo (i);  
[00000001] *** 10  
[00000002] *** 11  
[00000003] *** 12
```

15.4 UObjects

15.4.1 Is the UObject API Thread-Safe?

We are receiving a lot of questions on thread-safety issues in UObject code. So here comes a quick explanation on how things work in plugin and remote mode, with a focus on those questions.

15.4.1.1 Plugin mode

In *plugin mode*, all the UObject callbacks (timer, bound functions, `notifyChange` and `notifyAccess` targets) are called synchronously in the same thread that executes urbiscript code. All reads and writes to Urbi variables, through *UVar*, are done synchronously. Access to the UObject API (reading/writing UVars, using `call()`...) is possible from other threads, though those operations are currently using one serialization lock with the main thread: each UObject API call from an other thread will wait until the main thread is ready to process it.

15.4.1.2 Remote mode

Execution model In *remote mode*, a single thread is also used to handle all UObject callbacks, for all the UObjects in the same executable. It means that two bound functions registered from the same executable will never execute in parallel. Consider this sample C++ function:

```
int MyObject::test(int delay)
{
    static const int callNumber = 0;
    int call = ++callNumber;
    std::cerr << "in " << call << ": " << time() << std::endl;
    sleep(delay);
    std::cerr << "out " << call << ": " << time() << std::endl;
    return 0;
}
```

If this function is bound in a remote uobject, the following code:

```
MyObject.test(1), MyObject.test(1)
```

will produce the following output (assuming the first call to `time` returns 1000).

```
in 1: 1000
out 1: 1001
in 2: 1001
out 2: 1002
```

However, the execution of the Urbi kernel is not “stuck” while the remote function executes, as the following code demonstrates:

```
var t = Tag.new;
test(1) | t.stop,
t.every(300ms)
  cerr << "running";
```

The corresponding output is (mixing the kernel and the remote outputs):

```
[0] running
in 1: 1000
[300] running
[600] running
[900] running
out 1: 1001
```

As you can see, Urbi semantics is respected (the execution flow is stuck until the return value from the function is returned), but the kernel is not stuck: other pieces of code are still running.

Thread-safety The `liburbi` and the UObject API in remote mode are thread safe. All operations can be performed in any thread. As always, care must be taken for all non-atomic operations. For example, the following function is not thread safe:

```
void
writeToVar(UClient* cl, std::string varName, std::string value)
{
    (*cl) << varName << " = " << value << ";";
}
```

Two simultaneous calls to this function from different threads can result in the two messages being mixed. The following implementation of the same function is thread-safe however:

```
void
writeToVar(UClient* cl, std::string varName, std::string value)
{
    std::stringstream s;
    s << varName << " = " << value << ";";
    (*cl) << s.str();
}
```

since a single call to UClient's `operator <<` is thread-safe.

15.5 Miscellaneous

15.5.1 What has changed since the latest release?

See [Chapter 24](#).

15.5.2 How can I contribute to the code?

You are encouraged to submit patches to kernel@lists.gostai.com, where they will be reviewed by the Urbi team. If they fit the project and satisfy the quality requirements, they will be accepted. As of today there is no public repository for Urbi SDK (there will be, eventually), patches should be made against the latest source tarballs (see <http://gostai.com/downloads/urbi-sdk/2.x/>).

Even though Urbi SDK is free software (GNU Affero General Public License 3+, see the 'LICENSE.txt' file), licensing patches under GNU AGPL3+ does not suffice to support our dual licensed products. This situation is common, see for instance the case of Oracle VM Virtual Box, http://www.virtualbox.org/wiki/Contributor_information.

There are different means to ensure that your contributions to Urbi SDK can be accepted. None require that you “give away your copyright”. What is needed, is the right to use contributions, which can be achieved in two ways:

- Sign the Urbi Open Source Contributor Agreement, see [Section 25.6](#). This may take some time initially, but it will cover all your future contributions.
- Submit your contribution under the Expat license (also known as the “MIT license”, see [Section 25.2](#)), or under the modified BSD license (see [Section 25.1](#)).

15.5.3 How do I report a bug?

Bug reports should be sent to kernel-bugs@lists.gostai.com, it will be addressed as fast as possible. Please, be sure to read the FAQ (possibly updated on our web site), and to have checked that no more recent release fixed your issue.

Each bug report should contain a self-contained example, which can be tested by our team. Using self-contained code, i.e., code that does not depend on other code, helps ensuring that we will be able to duplicate the problem and analyze it promptly. It will also help us integrating the code snippet into our non-regression test suite so that the bug does not reappear in the future.

If your report identifies a bug in the Urbi kernel or its dependencies, we will prepare a fix to be integrated in a later release. If the bug takes some time to fix, we may provide you with a workaround so that your developments are not delayed.

In your bug report, make sure that you indicate the Urbi version you are using (use ‘`urbi --version`’ to check it out) and whether this bug is blocking you or not. Also, please keep kernel-bugs@lists.gostai.com in copy of all your correspondence, and do not reply individually to a member of our team as this may slow down the handling of the report.

If your bug report is about a failing ‘`make check`’, first be sure to read [Section 17.9](#).

Chapter 16

Migration from urbiscript 1 to urbiscript 2

This chapter is intended to people who want to migrate programs in urbiscript 1 to urbiscript 2. Backward compatibility is *mostly* ensured, but some urbiscript 1 constructs were removed because they prevented the introduction of cleaner constructs in urbiscript 2. When possible, urbiscript 2 supports the remaining urbiscript 1 constructs. The [Kernel1 \(??sec:std-Kernel1\)](#) object contains functions that support some urbiscript 1 features.

16.1 \$(Foo)

This construct was designed to build identifiers at run-time. This used to be a common idiom to work around some limitations of urbiscript 1 which are typically *no longer needed in urbiscript 2*. For instance, genuine local variables are simpler and safer to use than identifiers forged by hand to be unique. In order to associate information to a string, use a [Dictionary \(??sec:std-Dictionary\)](#).

If you really need to forge identifiers at run-time, use `setSlot`, `updateSlot`, and `getSlot`, which all work with strings, and possibly `asString`, which converts arbitrary expressions into strings. The following table lists common patterns.

Deprecated	Updated
<code>var \$(Foo) = ...;</code> <code>\$(Foo) = ...;</code> <code>\$(Foo)</code>	<code>setSlot(Foo.asString, ...);</code> <code>updateSlot(Foo.asString, ...);</code> <code>getSlot(Foo.asString);</code>

16.2 delete Foo

In order to maintain an analogy with the C++ language, urbiscript used to support `delete Foo`, but this was removed for a number of reasons:

- urbiscript 2 features genuine local variables for which `delete` makes no sense.

- in C++ `delete` really targets the object: destroy yourself, then the system will reclaim the memory. In urbiscript one cannot destroy an object and reclaim the memory, it is the task of the system to notice objects that are no longer used, and to reclaim the memory. This is called *garbage collection*. Therefore in urbiscript `delete` is actually bounced to `removeSlot` sent to the owner of the object.
- `delete` is an unsafe feature that makes only sense in pointer-based languages such as C and C++. It enables nice bugs such as:

```
var this.a := A.new;
// ...
delete this.a;
// ...
cout << this.a;
```

For these reasons, and others, `delete Foo` was removed. To remove the *name* `Foo`, run `removeSlot("Foo")` (Section 6.1) — the garbage collector will reclaim memory if there are no other use of `Foo`. To remove the contents of `Foo`, you remove all its slots one by one:

```
class Foo
{
  var a = 12;
  var b = 23;
} | {};
function Object.removeAllSlots()
{
  for (var s: localSlotNames)
    removeSlot(s);
} | {};
Foo.removeAllSlots;
Foo.localSlotNames;
[00000000] []
```

16.3 emit Foo

The keyword `emit` is deprecated in favor of `!`.

Deprecated	Updated
<code>emit e;</code>	<code>e!;</code>
<code>emit e(a);</code>	<code>e!(a);</code>
<code>emit e ~ 1s;</code>	<code>e! ~ 1s;</code>
<code>emit e(a) ~ 1s;</code>	<code>e!(a) ~ 1s;</code>

The `?` construct is changed for symmetry.

Deprecated	Updated
<code>at (?e)</code>	<code>at (e?)</code>
<code>at (?e(var a))</code>	<code>at (e?(var a))</code>
<code>at (?e(var a) if a == 2)</code>	<code>at (e?(var a) if a == 2)</code>

16.4 eval(Foo)

`eval` is still supported, but its use is discouraged: one can often easily do without. For instance, `eval` was often used to manipulate forged identifiers; see [Section 16.1](#) for means of getting rid of them.

16.5 foreach

The same feature with a slightly different syntax is now provided by `for`. See [Section 19.6.5.2](#).

16.6 group

Where support for groups was a built-in feature in urbiscript 1, it is now provided by the standard library, see [Group \(??sec:std-Group\)](#). Instead of

```
group myGroup {a, b, c}
```

write

```
var myGroup = Group.new(a,b,c)
```

16.7 loopn

The same feature and syntax is now provided by `for`. See [Section 19.6.5.3](#).

16.8 new Foo

See [Section 9.5](#) for details on `new`. The construct `new Foo` is no longer supported because it is (too) ambiguous: what does `new a(1,2).b(3,4)` mean? Is `a(1,2).b` the object to clone and `(3,4)` are the arguments of the constructor? Or is it the result of `a(1,2).b(3,4)` that must be cloned?

In temporary versions, urbiscript 2 used to support this `new` construct, but too many users got it wrong, and we decided to keep only the simpler, safer, and more consistent method-call-style construct: `Foo.new`. Every single possible interpretation of `new a(1,2).b(3,4)` is reported below, unambiguously.

- `a(1,2).b(3,4).new`
- `a(1,2).b.new(3,4)`
- `a(1,2).new.b(3,4)`
- `a.new(1,2).b(3,4)`
- `new.a(1,2).b(3,4)`

16.9 self

For consistency with the C++ syntax, urbiscript now uses `this`.

16.10 stop Foo

Use `Foo.stop` instead, see [Tag \(??sec:std-Tag\)](#).

16.11 # line

Use `//#line` instead, see [Section 19.1.3](#).

16.12 tag+end

To detect the end of a statement, instead of

```
mytag+end: { echo ("foo") },
```

use the `leave?` method of the [Tag \(??sec:std-Tag\)](#) object:

```
{
  var mytag = Tag.new("mytag");
  at (mytag.leave?)
    Channel.new("mytag") << "code has finished";
  mytag: { echo ("foo") },
};
[00000002] *** foo
[00000003:mytag] "code has finished"
```

Chapter 17

Building Urbi SDK

This section is meant for people who want to *build* the Urbi SDK. If you just want to install a pre-built Urbi SDK, see [Chapter 14](#).

17.1 Requirements

This section lists all the dependencies of this package. Some of them are required to *bootstrap* the package, i.e., to build it from the repository. Others are required to *build* the package, i.e., to compile it from a tarball, or after a bootstrap.

17.1.1 Bootstrap

To bootstrap this package from its repository, and then to compile it, the following tools are needed.

Autoconf 2.64 or later

```
package: autoconf
```

Automake 1.11.1 or later Note that if you have to install Automake by hand (as opposed to “with your distribution’s system”), you have to tell its `aclocal` that it should also look at the files from the system’s `aclocal`. If `‘/usr/local/bin/aclocal’` is the one you just installed, and `‘/usr/bin/aclocal’` is the system’s one, then run something like this:

```
$ dirlist=$(/usr/local/bin/aclocal --print-ac-dir)/dirlist
$ sudo mkdir -p $(dirname $dirlist)
$ sudo /usr/bin/aclocal --print-ac-dir >>$dirlist
```

```
package: automake
```

Cvs This surprising requirement comes from the system Bison uses to fetch the current version of the message translations.

```
package: cvs
```

Git 1.6 or later Beware that older versions behave poorly with submodules.

```
package: git-core
```

Gettext 1.17 Required to bootstrap Bison. In particular it provides autopoint.

```
package: gettext
```

GNU sha1sum We need the GNU version of sha1sum.

```
package: coreutils
```

Help2man Needed by Bison.

```
package: help2man
```

Python You need ‘xml/sax’, which seems to be part only of Python 2.6. Using `python.select` can be useful to state that you want to use Python 2.6 by default (‘`sudo python.select python26`’).

```
deb: python2.6
MacPorts: python26
MacPorts: python_select
```

Texinfo Needed to compile Bison.

```
package: texinfo
```

yaml for Python The AST is generated from a description written in YAML. Our (Python) tools need to read these files to generate the AST classes. See <http://pyyaml.org/wiki/PyYAML>. The installation procedure on Cygwin is:

```
$ cd /tmp
$ wget http://pyyaml.org/download/pyyaml/PyYAML-3.09.zip
$ unzip PyYAML-3.09.zip
$ cd pyYAML-3.09
$ python setup.py install
```

```
MacPorts: py26-yaml
deb: python-yaml
```

At least on OS X you also need to specify the PYTHONPATH:

```
export PYTHONPATH="\
/opt/local/Library/Frameworks/Python.framework/Versions/2.6\
/lib/python2.6/site-packages:$PYTHONPATH"
```

17.1.2 Build

bc Needed by the test suite.

```
package: bc
```

Boost 1.38 or later Don't try to build it yourself, ask your distribution's management to install it.

```
deb: libboost-dev
MacPorts: boost
windows: http://www.boostpro.com/download
```

Ccache Not a requirement, but it's better to have it.

```
package: ccache
```

Flex 2.5.35 or later Beware that 2.5.33 has bugs that prevents this package from building correctly.

```
package: flex
```

G++ 4.0 or later GCC 4.2 or later is a better option. Beware that we have a problem with GCC-4.4 which rejects some Bison generated code. See [Section 15.1.2](#).

```
deb: g++-4.2
MacPorts: gcc42
```

Gnuplot Required to generate charts of trajectories for the documentation.

```
package: gnuplot
```

GraphViz Used to generate some of the figures in the documentation. There is no GraphViz package for Cygwin, so download the MSI file from GraphViz' site, and install it. Then change your path to go into its bin directory.

```
wget http://www.graphviz.org/pub/graphviz/stable/windows/graphviz-2.26.msi
PATH=/cygdrive/c/Program\ Files/Graphviz2.26/bin:$PATH
```

```
package: graphviz
```

ImageMagick Used to convert some of the figures in the documentation.

```
MacPorts: ImageMagick
deb: imagemagick
```

PDFLaTeX TeX Live is the most common T_EX distribution nowadays.

```
deb: texlive-base texlive-latex-extra
MacPorts: texlive texlive-latex-extra texlive-htmlxml
```

py-docutils This is not a requirement, but it's better to have it. Used by the test suite. Unfortunately the name of the package varies between distributions. It provides `rst2html`.

```
MacPorts: py26-docutils
```

socat Needed by the test suite to send messages to an Urbi server.

```
package: socat
```

TeX4HT Used to generate the HTML documentation.

```
deb: tex4ht
MacPorts: texlive_texmf-full
```

Transfig Needed to convert some figures for documentation (using `fig2dev`).

```
package: transfig
```

Valgrind Not needed, but if present, used by the test suite.

```
package: valgrind
```

xsltproc This is not a requirement, but it's better to have it.

17.2 Check out

Get the open source version tarball from <http://www.urbiforge.com/index.php/Main/Downloads> and uncompress it. With this version, the bootstrap step can be skipped.

17.3 Bootstrap

Run

```
$ ./bootstrap
```

17.4 Configure

You should compile in another directory than the one containing the sources.

```
$ mkdir _build
$ cd _build
$ ../configure OPTIONS...
```


17.4.1 configuration options

See `../configure --help` for detailed information. Unless you want to do funky stuff, you probably need no option.

To use ccache, pass `CC='ccache gcc' CXX='ccache g++'` as arguments to configure:

```
$ ../configure CC='ccache gcc' CXX='ccache g++' ...
```

17.4.2 Windows: Cygwin

The builds for windows use our wrappers. These wrappers use a database to store the dependencies (actually, to speed up their computation). We use Perl, and its DBI module. So be sort to have sqlite, and DBI.

```
$ perl -MCPAN -e 'install Bundle::DBI'
```

It might fail. The most important part is

```
$ perl -MCPAN -e 'install DBD::SQLite'
```

It might suffice, I don't know...

17.4.3 building For Windows

We support two builds: using Wine on top of Unix, and using Cygwin on top of Windows.

Both builds use our wrappers around MSVC's cl.exe and link.exe. It is still unclear whether it was a good or a bad idea, but the wrappers use the same names. Yet libtool will also need to use the genuine link.exe. So to set up Libtool properly, you will need to pass the following as argument to configure:

```
$ AR=lib.exe \
  CC='ccache cl.exe' \
  CC_FOR_BUILD=gcc \
  CXX='ccache cl.exe' \
  LD=link.exe \
  DUMPBIN='/cygdrive/c/vcxx8/VC/bin/link.exe -dump -symbols' \
  RANLIB=: \
  host_alias=mingw32 \
  --host=mingw32
```

where:

- `'lib.exe'`, `'cl.exe'`, `'link.exe'` are the wrappers
- `'/cygdrive/c/vcxx8/VC/bin/link.exe'` is the genuine MSVC tool.

Since we are cross-compiling, we also need to specify `CC_FOR_BUILD` so that `config.guess` can properly guess the type of the build machine.

17.4.4 Building for Windows using Cygwin

We use our `cl.exe` wrappers, which is something that Libtool cannot know. So we need to tell it that we are on Windows with Cygwin, and pretend we use GCC, so we pretend we run mingw.

The following options have been used with success to compile Urbi SDK with Visual C++ 2005. Adjust to your own case (in particular the location of Boost).

```
$ ./configure \
-C \
--prefix=/usr/local/gostai \
--enable-compilation-mode=debug \
--enable-shared \
--disable-static \
--enable-dependency-tracking \
--with-boost=/cygdrive/c/gt-win32-2/d/boost_1_39 \
AR=lib.exe \
CC='ccache cl.exe' \
CC_FOR_BUILD=gcc \
CXX='ccache cl.exe' \
LD=link.exe \
DUMPBIN='/cygdrive/c/vcxx8/VC/bin/link.exe -dump -symbols' \
RANLIB=: \
host_alias=mingw32 \
--host=mingw32
```

17.5 Compile

Should be straightforward.

```
$ make -j3
```

Using `distcc` and `ccache` is recommended.

17.6 Install

Running `'make install'` works as expected. It is a good idea to check that your installation works properly: run `'make installcheck'` (see [Section 17.9](#)).

17.7 Relocatable

If you intend to make a relocatable tree of Urbi SDK (i.e., a self-contained tree that can be moved around), then run `'make relocatable'` after `'make install'`.

This step *requires* that you use a `DESTDIR` (see the Automake documentation for more information). Basically, the sequence of commands should look like:

```
$ DESTDIR=/tmp/install
$ rm -rf $DESTDIR
$ make install DESTDIR=$DESTDIR
```

```
$ make relocatable DESTDIR=$DESTDIR
$ make installcheck DESTDIR=$DESTDIR
```

You may now move this tree around and expect the executable to work properly.

17.8 Run

There are some special variables in addition to the environment variables documented in the manual.

URBI_CONSOLE_MODE Skip lines in input that look like shell output. A way to accept *.chk as input.

URBI_DESUGAR Display the desugared ASTs instead of the original one.

URBI_IGNORE_URBI_U Ignore failures (such as differences between kernel revision and ‘urbi.u’ revision) during the initialization.

URBI_LAUNCH The path to `urbi-launch` that `urbi.exe` will exec.

URBI_NO_ICE_CATCHER Don’t try to catch SEGVs.

URBI_PARSER Enable Bison parser traces.

URBI_REPORT Display stats about execution rounds performed by the kernel.

URBI_ROOT_LIBname The location of the libraries to load, without the extension. The **LIBname** are: **LIBJPEG4URBI**, **LIBPLUGIN** (libuobject plugin), **LIBPORT**, **LIBREMOTE** (libuobject remote), **LIBSCHED**, **LIBSERIALIZE**, **LIBURBI**.

URBI_SCANNER Enable Flex scanner traces.

URBI_TOPLEVEL Force the display the result of the top-level evaluation into the lobby.

17.9 Check

There are several test suites that will be run if you run ‘`make check`’ (‘-j4’ works on most machines).

To rerun only the tests that failed, use ‘`make recheck`’. Some tests have explicit dependencies, and they are not rerun if nothing was changed (unless they had failed the previous time). It is therefore expected that after one (long) run of ‘`make check`’, the second one will be “instantaneous”: the previous log files are reused.

Note that running ‘`make`’ in the top-level is needed when you change something deep in the package. If you forget this ‘`make`’ run, the timestamps on which the test suite depends will not be updated, and therefore the results of the previous state of the package will be used, instead of your fresh changes.

Some tests are extremely “touchy”. Because the test suite exercises Urbi under extreme conditions, some tests may fail not because of a problem in Urbi, but because of non-determinism in the test itself. In this case, another run of ‘make check’ will give an opportunity for the test to pass (remind that the tests that passed will not be run again). Also, using ‘make check -j16’ is a sure means to have the Urbi scheduler behave insufficiently well for the test to pass. **Do not send bug reports for such failures..** Before reporting bugs, make sure that the failures remain after a few ‘make check -j1’ invocations.

17.9.1 Lazy test suites

The test suites are declared as “lazy”, i.e., unless its dependencies changed, a successful test will be run only once — failing tests do not “cache” their results. Because spelling out the dependencies is painful, we rely on a few timestamps:

‘libraries.stamp’ Updated when a library is updated (libport, libuobject, etc.).

‘executables.stamp’ Updated when an executable is updated (urbi-launch, etc.). Depends on libraries.stamp.

‘urbi.stamp’ When Urbi sources (‘share/urbi/*.u’) are updated.

‘all.stamp’ Updated when any of the three aforementioned timestamps is.

These timestamps are updated **only** when make is run in the top-level. Therefore, the following sequence does not work as expected:

```
$ make check -C tests      # All passes.
$ emacs share/urbi/foo.u
$ make check -C tests      # All passes again.
```

because the timestamps were not given a chance to notice that some Urbi source changed, so Make did not notice the tests really needed to be rerun and **the tests were not run**.

You may either just update the timestamps:

```
$ make check -C tests      # All passes.
$ emacs share/urbi/foo.u
$ make stamps              # Update the timestamps.
$ make check -C tests      # All passes again.
```

or completely disable the test suite laziness:

```
$ make check -C tests LAZY_TEST_SUITE=
```

17.9.2 Partial test suite runs

You can run each test suite individually by hand as follows:

‘sdk-remote/libport/test-suite.log’ Tests libport.

```
$ make check -C sdk-remote/libport
```

‘sdk-remote/src/tests/test-suite.log’ Checks liburbi, and some of the executables we ship. Requires the kernel to be compiled in order to be able to test some of the uobjects.

```
$ make check -C sdk-remote/src/tests
```

‘tests/test-suite.log’ Tests the kernel and uobjects.

```
$ make check -C tests
```

Partial runs can be invoked:

```
$ make check -C tests TESTS='2.x/echo.chk'
```

wildcards are supported:

```
$ make check -C tests TESTS='2.x/*'
```

To check remote uobjects tests:

```
$ make check -C tests TESTS='uob/remote/*'
```

‘doc/tests/test-suite.log’ The snippets of code displayed in the documentation are transformed into test files.

```
$ make check -C doc
```


Part IV

Urbi SDK Reference Manual

About This Part

This part defines the specifications of the urbiscript language version 2.0. It defines the expected behavior from the urbiscript interpreter, the standard library, and the SDK. It can be used to check whether some code is valid, or browse urbiscript or C++ API for a desired feature. Random reading can also provide you with advanced knowledge or subtleties about some urbiscript aspects.

This part is not an urbiscript tutorial; it is not structured in a progressive manner and is too detailed. Think of it as a dictionary: one does not learn a foreign language by reading a dictionary. The urbiscript Tutorial ([Part II](#)), or the live urbiscript tutorial built in the interpreter are good introductions to urbiscript.

This part does not aim at giving advanced programming techniques. Its only goal is to define the language and its libraries.

Chapter 18 — Programs

Presentation and usage of the different tools available with the Urbi framework related to urbiscript, such as the Urbi server, the command line client, `umake`, ...

Chapter 19 — urbiscript Language Reference Manual

Core constructs of the language and their behavior.

Chapter 20 — urbiscript Standard Library

Listing of all classes and methods provided in the standard library.

Chapter 21 — Communication with ROS

Urbi provides a set of tools to communicate with ROS (Robot Operating System). For more information about ROS, see <http://www.ros.org>. Urbi, acting as a ROS node, is able to interact with the ROS world.

Chapter 22 — Gostai Standard Robotics API

Also known as “The Urbi Naming Standard”: naming conventions in for standard hardware/software devices and components implemented as UObject and the corresponding slots/events to access them.

Chapter 18

Programs

18.1 Environment Variables

There is a number of environment variables that alter the behavior of the Urbi tools.

18.1.1 Search Path Variables

Some variables define *search-paths*, i.e., colon-separated lists of directories in which library files (urbiscript programs, UObjects and so forth) are looked for.

The tools have predefined values for these variables which are tailored for your installation — so that Urbi tools can be run without any special adjustment. In order to provide the user with a means to override or extend these built-in values, the path variables support a special syntax: a lone colon specifies where the standard search path must be inserted. See the following examples about `URBI_PATH`.

```
# Completely override the system path. First look for files in
# /home/jessie/urbi, then in /usr/local/urbi.
export URBI_PATH=/home/jessie/urbi:/usr/local/urbi

# Prepend the previous path to the default path. This is dangerous as
# it may result in some standard files being hidden.
export URBI_PATH=/home/jessie/urbi:/usr/local/urbi:

# First look in Jessie's directory, then the default location, and
# finally in /usr/local/urbi.
export URBI_PATH=/home/jessie/urbi::/usr/local/urbi

# Extend the default path, i.e., files that are not found in the
# default path will be looked for in Jessie's place, and then in
# /usr/local/urbi
export URBI_PATH=:/home/jessie/urbi:/usr/local/urbi
```

Windows Issues

On Windows too directories are separated by colons, but backslashes are used instead of forward-slashes. For instance

```
URBI_PATH=C:\cygwin\home\jessie\urbi:C:\cygwin\usr\local\urbi
```

18.1.2 Environment Variables

GD LEVEL Set the verbosity level of traces. This environment variable is meant for the developers of Urbi SDK, yet it is very useful when tracking problems such as a UObject that fails to load properly. Valid values are, in increasing verbosity order:

1. NONE, no log messages at all.
2. LOG, the default value.
3. TRACE
4. DEBUG
5. DUMP, maximum verbosity.

URBI_PATH The search-path for urbiscript source files (i.e., ‘*.u’ files).

URBI_ROOT The Urbi SDK is relocatable: its components know the relative location of each other. Yet they need to “guess” the Urbi root, i.e., the path to the directory that contains the files. This variable also to override that guess. Do not use it unless you know exactly what you are doing.

URBI.UOBJECT_PATH The search-path for UObjects files. This is used by `urbi-launch`, by `System.loadModule` and `System.loadLibrary`.

18.2 Special Files

‘CLIENT.INI’ This is the obsolete name for ‘global.u’.

‘global.u’ If found in the `URBI_PATH` (see [Section 18.1](#)), this file is loaded by Urbi server upon start-up. It is the appropriate place to install features you mean to provide to all the users of the server. It is will be loaded via a special system connection, with its own private lobby. Therefore, purely local definitions will not be reachable from users; global modifications should be made in globally visible objects, say `Global` ([??sec:std-Global](#)).

‘local.u’ If found in the `URBI_PATH` (see [Section 18.1](#)), this file is loaded by every connection established with an Urbi server. This is the appropriate place for enhancements local to a lobby.

‘URBI.INI’ This is the obsolete name for ‘global.u’.

18.3 urbi — Running an Urbi Server

The `urbi` program launches an Urbi server, for either batch, interactive, or network-based executions. It is subsumed by, but simpler to use than, `urbi-launch` ([Section 18.5](#)).

18.3.1 Options

General Options

‘-h’, ‘--help’

Display the help message and exit successfully.

‘--version’

Display version information and exit successfully.

Tuning

‘-d’, ‘--debug=*level*’

Set the verbosity level of traces. See the `GD_LEVEL` documentation ([Section 18.1.2](#)).

‘-F’, ‘--fast’

Ignore system time, go as fast as possible. Do not use this option unless you know exactly what you are doing.

The ‘--fast’ flag makes the kernel run the program in “simulated time”, as fast as possible. A `sleep` in fast mode will not actually wait (from the wall-clock point of view), but the kernel will internally increase its simulated time.

For instance, the following session behaves equally in fast and non-fast mode:

```
{ sleep(2s); echo("after") } & { sleep(1s); echo("before") };
[000000463] *** before
[000001463] *** after
```

However, in non fast mode the execution will take two seconds (wall clock time), while it be instantaneous in fast mode. This option was designed for testing purpose; *it does not preserve the program semantics*.

‘-s’, ‘--stack-size=*size*’

Set the coroutine *stack size*. The unit of *size* is KB; it defaults to 128.

This option should not be needed unless you have “stack exhausted” messages from `urbi` in which case you should try ‘--stack-size=512’ or more.

Alternatively you can define the environment variable `URBI_STACK_SIZE`. The option ‘--stack-size’ has precedence over the `URBI_STACK_SIZE`.

‘-q’, ‘--quiet’

Do not send the welcome banner to incoming clients.

Networking

‘-H’, ‘--host=*address*’

Set the *address* on which network connections are listened to. Typical values of *address* include:

localhost only local connections are allowed (no other computer can reach this server).

127.0.0.1 same as localhost.

0.0.0.0 any IP v4 connection is allowed, including from remote computers.

Defaults to 0.0.0.0.

‘-P’, ‘--port=*port*’

Set the port to listen incoming connections to. If *port* is -1, no networking. If *port* is 0, then the system will chose any available port (see ‘--port-file’). Defaults to -1.

‘-w’, ‘--port-file=*file*’

When the system is up and running, and when it is ready for network connections, create the file named *file* which contains the number of the port the server listens to.

Execution

‘-e’, ‘--expression=*exp*’

Send the urbiscript expression *exp*. No separator is added, you have to pass yours.

‘-f’, ‘--file=*file*’

Send the contents of the file *file*. No separator is added, you have to pass yours.

‘-i’, ‘--interactive’

Start an interactive session.

The options ‘-e’, ‘-f’ accumulate, and are run in the same [Lobby \(??sec:std-Lobby\)](#) as ‘-i’ if used. In other words, the following session is valid:

```
# Create a file "two.u".
$ echo "var two = 2;" >two.u
$ urbi -q -e 'var one = 1;' -f two.u -i
[00000000] 1
[00000000] 2
one + two;
[00000000] 3
```

18.4 urbi-image — Querying Images from a Server

```
urbi-image option...
```

Connect to an Urbi server, and fetch images from it, for instance from its camera.

18.4.1 Options

General Options

`-h`, `--help`
Display the help message and exit successfully.

`--version`
Display version information and exit successfully.

Networking

`-H`, `--host=host`
Address to connect to.

`-P`, `--port=port`
Port to connect to.

`--port-file=file`
Connect to the port contained in the file *file*.

Tuning

`-p`, `--period=period`
Specify the period, in millisecond, at which images are queried.

`-F`, `--format=format`
Select format of the image (`'rgb'`, `'ycrcb'`, `'jpeg'`, `'ppm'`).

`-r`, `--reconstruct`
Use reconstruct mode (for aibo).

`-j`, `--jpeg=factor`
JPEG compression factor (from 0 to 100, defaults to 70).

`-d`, `--device=device`
Query image on *device.val* (default: `camera`).

`-o`, `--output=file`
Query and save one image to *file*.

`-R`, `--resolution=resolution`
Select resolution of the image (0=biggest).

`-s`, `--scale=factor`
Rescale image with given *factor* (display only).

18.5 urbi-launch — Running a UObject

The `urbi-launch` program launches an Urbi system. It is more general than `urbi` ([Section 18.3](#)): everything `urbi` can do, `urbi-launch` can do it too.

18.5.1 Invoking urbi-launch

`urbi-launch` launches UObjects, either in plugged-in mode, or in remote mode. Since UObjects can also accept options, the command line features two parts, separated by ‘--’:

```
urbi-launch [urbi-launch-option...] module... [-- module-option...]
```

The *modules* are looked for in the `URBI_UOBJECT_PATH`. The *module* extension (‘.so’, or ‘.dll’) does not need to be specified.

Urbi-launch options

‘-h’, ‘--help’

Display the help message and exit successfully.

‘--version’

Display version information and exit successfully.

‘-c’, ‘--customize=*file*’

Start the Urbi server in *file*. This option is mostly for developers.

‘-d’, ‘--debug=*level*’

Set the verbosity level of traces. See the `GD_LEVEL` documentation ([Section 18.1.2](#)).

Mode selection

‘-p’, ‘--plugin’

Attach the *module* onto a currently running Urbi server (identified by *host* and *port*). This is equivalent to running `loadModule("module")` on the corresponding server.

‘-r’, ‘--remote’

Run the *modules* as separated processes, connected to a running Urbi server (identified by *host* and *port*) via network connection.

‘-s’, ‘--start’

Start an Urbi server with plugged-in *modules*. In this case, the *module-option* are exactly the options supported by `urbi`.

Networking `urbi-launch` supports the same networking options (‘--host’, ‘--port’, ‘--port-file’) as `urbi`, see [Section 18.3](#).

18.5.2 Examples

To launch a fresh server in an interactive session with the UFactory UObject compiled as the file ‘factory.so’ (or ‘factory.dll’ plugged in, run:

```
urbi-launch --start ufactory -- --interactive
```

To start an Urbi server accepting connections on the local port 54000 from any remote host, with UFactory plugged in, run:

```
urbi-launch --start --host 0.0.0.0 --port 54000 ufactory
```

18.6 urbi-send — Sending urbiscript Commands to a Server

```
urbi-send option...
```

Connect to an Urbi server, and send commands or file contents to it. Stay connected, until server disconnection, or user interruption (such as C-c under a Unix terminal).

‘-e’, ‘--expression=*script*’

Send *script* to the server.

‘-f’, ‘--file=*file*’

Send the contents of *file* to the server.

‘-h’, ‘--help’

Display the help message and exit successfully.

‘-H’, ‘--host=*host*’

Address to connect to.

‘-P’, ‘--port=*port*’

Port to connect to.

‘--port-file=*file*’

Connect to the port contained in the file *file*.

‘-Q’, ‘--quit’

Disconnect from the server immediately after having sent all the commands. This is equivalent to ‘-e ’quit;’’. This is inappropriate if code running in background is expected to deliver its result asynchronously: the connection will be closed before the result was sent.

Without this option, **urbi-send** prompts the user to hit C-c to end the connection.

‘--version’

Display version information and exit successfully.

18.7 umake — Compiling UObject Components

The `umake` programs builds loadable modules, UObjects, to be later run using `urbi-launch` (Section 18.5). Using it is not mandatory: users familiar with their compilation tools will probably prefer using them directly. Yet `umake` makes things more uniform and simpler, at the cost of less control.

18.7.1 Invoking umake

Usage:

```
umake option... file...
```

Compile the *file*. The *files* can be of different kinds:

- objects files (`*.o`, `*.obj` and so forth) and linked into the result.
- libraries (`*.a`) and linked into the result.
- source files (`*.cc`, `*.cpp`, `*.c`, `*.C`) are compiled.
- header files (`*.h`, `*.hh`, `*.hxx`, `*.hpp`) are *not* compiled, but used as dependencies: if a header file is changed, the next `umake` run will actually recompile.
- directories are recursively traversed, and files of the above types are gathered as if they were given on the command line.

General options

`-D`, `--debug`

Turn on shell debugging (`set -x`) to track `umake` problems.

`-h`, `--help`

Display the help message and exit successfully.

`-q`, `--quiet`

Produce no output except errors.

`-V`, `--version`

Display version information and exit successfully.

`-v`, `--verbose`

Report on what is done.

Compilation options

- `--deep-clean`
Remove all building directories and exit.
- `-c`, `--clean`
Clean building directory before compilation.
- `-j`, `--jobs=jobs`
Specify the numbers of compilation commands to run simultaneously.
- `-l`, `--library`
Produce a library, don't link to a particular core.
- `-s`, `--shared`
Produce a shared library loadable by any core.
- `-o`, `--output=file`
Set the output file name.
- `-C`, `--core=core`
Set the build type.
- `-H`, `--host=host`
Set the destination host.
- `-m`, `--disable-automain`
Do not add the main function.

Developer options

- `-p`, `--prefix=dir`
Set library files location.
- `-P`, `--param-mk=file`
Set '`param.mk`' location.
- `-k`, `--kernel=dir`
Set the kernel location.

18.7.2 umake Wrappers

As a convenience for common **umake** usages, some wrappers are provided:

umake-deepclean — Cleaning

Clean the temporary files made by running **umake** with the same arguments. Same as '`umake --deep-clean`'.

umake-shared — Compiling Shared UObjects

Build a shared object to be later run using **urbi-launch** ([Section 18.5](#)). Same as '`umake --shared-library`'.

Chapter 19

urbiscript Language Reference Manual

19.1 Syntax

19.1.1 Characters, encoding

Currently urbiscript makes no assumptions about the encoding used in the programs, but the streams are handled as 8-bit characters.

While you are allowed to use whatever character you want in the string literals (especially using the binary escapes, [Section 19.1.6.6](#)), only plain ASCII characters are allowed in the program body. Invalid characters are reported, possibly escaped if they are not “printable”. If you enter UTF-8 characters, since they possibly span over several 8-bit characters, a single (UTF-8) character may be reported as several invalid (8-bit) characters.

```
#Été;  
[00048238:error] !!! syntax error: invalid character: '#'  
[00048239:error] !!! syntax error: invalid character: '\xc3'  
[00048239:error] !!! syntax error: invalid character: '\x89'  
[00048239:error] !!! syntax error: invalid character: '\xc3'  
[00048239:error] !!! syntax error: invalid character: '\xa9'
```

19.1.2 Comments

Comments are used to document the code, they are ignored by the urbiscript interpreter. Both C++ comment types are supported.

- A `//` introduces a comment that lasts until the end of the line.
- A `/*` introduces a comment that lasts until `*/` is encountered. Comments nest, contrary to C/C++: if two `/*` are encountered, the comment will end after two `*/`, not one.

```

1; // This is a one line comment.
[00000001] 1

2; /* an inner comment */ 3;
[00000002] 2
[00000003] 3

4; /* nested /* comments */ 5; */ 6;
[00000004] 4
[00000005] 6

7
  /*
    /*
      Multi-line.
    */
  */
;
[00000006] 7

```

19.1.3 Synclines

While the interaction with an urbiscript kernel is usually performed via a network connection, programmers are used to work with files which have names, line numbers and so forth. This is most important in error messages. Since even loading a file actually means sending its content as if it were typed in the network session, in order to provide the user with meaningful locations in error messages, urbiscript features *synclines*, a means to change the “current location”, similarly to `#line` in C-like languages. This is achieved using special `//#` comments.

The following special comments are recognized only as a whole line. If some component does not match exactly the expected syntax, or if there are trailing items, the whole line is treated as a comment.

- `//#line line "file"`
Specify that the *next* line is from the file named *file*, and which line number is *line*. The current location (i.e., current file and line) is lost.
- `//#push line "file"`
Save the current location, and then behave as if `//#line` was used.
- `//#pop`
Restore the current location. `//#push` and `//#pop` must match.

19.1.4 Identifiers

Identifiers in urbiscript are composed of one or more alphanumeric or underscore (`_`) characters, not starting by a digit, i.e., identifiers match the `[a-zA-Z_][a-zA-Z0-9_]*` regular expression. Additionally, identifiers must not match any of the urbiscript reserved words¹ documented in

¹ The only exception to this rule is `new`, which can be used as the method identifier in a method call.

[Section 19.1.5](#). Identifiers can also be written between simple quotes (`'`), in which case they may contain any character.

```
var x;
var foobar51;
var this.a_name_with_underscores;
// Invalid.
// var 3x;
// obj.3x();

// Invalid because "if" is a keyword.
// var if;
// obj.if();
// However, keywords can be escaped with simple quotes.
var 'if';
var this.'else';

// Identifiers can be escaped with simple quotes
var '%x';
var '1 2 3';
var this.'[]';
```

19.1.5 Keywords

Keywords are reserved words that cannot be used as identifiers, for instance. They are listed in [Table 19.1](#).

19.1.6 Literals

19.1.6.1 Angles

Angles are floats (see [Section 19.1.6.4](#)) followed by an angle unit. They are simply equivalent to the same float, expressed in radians. For instance, `180deg` (180 degrees) is equal to `pi`. Available units and their equivalent are presented in [Table 19.2](#).

```
pi == 180deg;
pi == 200grad;
```

19.1.6.2 Dictionaries

Literal *dictionaries* are represented with a comma-separated, potentially empty list of arbitrary associations enclosed in square brackets (`[]`), as shown in the listing below. Empty dictionaries are represented with an association arrow between the brackets to avoid confusion with empty lists. See [Dictionary \(??sec:std-Dictionary\)](#) for more details.

Each association is composed of a key, which is represented by a string, an arrow (`=>`) and an expression.

```
[ => ]; // The empty dictionary
[00000000] [ => ]
```

Keyword	Remark	Keyword	Remark
<code>and</code>	Synonym for <code>&&</code>	<code>long</code>	Reserved
<code>and_eq</code>	Synonym for <code>&=</code>	<code>loop</code>	<code>loop&</code> and <code>loop </code> flavors
<code>asm</code>	Reserved	<code>loopn</code>	Deprecated, use <code>for</code>
<code>at</code>		<code>mutable</code>	Reserved
<code>auto</code>	Reserved	<code>namespace</code>	Reserved
<code>bitand</code>	Synonym for <code>&</code>	<code>new</code>	
<code>bitor</code>	Synonym for <code> </code>	<code>not</code>	Synonym for <code>!</code>
<code>bool</code>	Reserved	<code>not_eq</code>	Synonym for <code>!=</code>
<code>break</code>			
<code>call</code>		<code>onleave</code>	
<code>case</code>		<code>or</code>	Synonym for <code> </code>
<code>catch</code>		<code>or_eq</code>	Synonym for <code> =</code>
<code>char</code>	Reserved	<code>private</code>	Ignored
<code>class</code>		<code>protected</code>	Ignored
<code>closure</code>		<code>public</code>	Ignored
<code>compl</code>	Synonym for <code>~</code>	<code>register</code>	Reserved
<code>const</code>		<code>reinterpret_cast</code>	Reserved
<code>const_cast</code>	Reserved	<code>return</code>	
<code>continue</code>		<code>short</code>	Reserved
<code>default</code>		<code>signed</code>	Reserved
<code>delete</code>	Reserved	<code>sizeof</code>	Reserved
<code>do</code>		<code>static</code>	Deprecated
<code>double</code>	Reserved	<code>static_cast</code>	Reserved
<code>dynamic_cast</code>	Reserved	<code>stopif</code>	
<code>else</code>		<code>struct</code>	Reserved
<code>emit</code>	Deprecated	<code>switch</code>	
<code>enum</code>	Reserved	<code>template</code>	Reserved
		<code>this</code>	
<code>every</code>		<code>throw</code>	
<code>explicit</code>	Reserved	<code>timeout</code>	
<code>export</code>	Reserved	<code>try</code>	
<code>extern</code>	Reserved	<code>typedef</code>	Reserved
<code>external</code>		<code>typeid</code>	Reserved
<code>float</code>	Reserved	<code>typename</code>	Reserved
<code>for</code>	<code>for&</code> and <code>for </code> flavors	<code>union</code>	Reserved
<code>foreach</code>	Deprecated, use <code>for</code>	<code>unsigned</code>	Reserved
<code>freezeif</code>		<code>using</code>	Reserved
<code>friend</code>	Reserved	<code>var</code>	
		<code>virtual</code>	Reserved
<code>function</code>		<code>volatile</code>	Reserved
<code>goto</code>	Reserved	<code>waituntil</code>	
<code>if</code>		<code>wchar_t</code>	Reserved
<code>in</code>		<code>whenever</code>	
<code>inline</code>	Reserved	<code>while</code>	<code>while&</code> and <code>while </code> flavors
<code>int</code>	Reserved	<code>xor</code>	Synonym for <code>^</code>
<code>internal</code>	Deprecated	<code>xor_eq</code>	Synonym for <code>^=</code>

Table 19.1: Keywords

unit	abbreviation	equivalence for n
radian	rad	n
degree	deg	$n/180 * \pi$
grad	grad	$n/200 * \pi$

Table 19.2: Angle units

```
["a" => 1, "b" => 2, "c" => 3];
[00000000] ["a" => 1, "b" => 2, "c" => 3]
```

19.1.6.3 Durations

Durations are floats (see [Section 19.1.6.4](#)) followed by a time unit. They are simply equivalent to the same float, expressed in seconds. For instance, `1s 1ms`, which stands for “one second and one millisecond”, is strictly equivalent to `1.0001`. Available units and their equivalent are presented in [Table 19.3](#).

```
1d == 24h;
0.5d == 12h;
1h == 60min;
1min == 60s;
1s == 1000ms;

1s == 1;
1s 2s 3s == 6;
1s 1ms == 1.001;
1ms 1s == 1.001;
```

19.1.6.4 Floats

urbiscript supports the *scientific notation* for floating-point literals. See [Float \(??sec:std-Float\)](#) for more details. Examples include:

```
1 == 1;
1 == 1.0;
1.2 == 1.2000;
1.234e6 == 1234000;
```

unit	abbreviation	equivalence for n
millisecond	ms	$n/1000$
second	s	n
minute	min	$n \times 60$
hour	h	$n \times 60 \times 60$
day	d	$n \times 60 \times 60 \times 24$

Table 19.3: Duration units

```
1e+11 == 1E+11;
```

Numbers are displayed rounded by the top level, but internally, as seen above, they keep their accurate value.

```
0.000001;
[00000011] 1e-06

0.0000001;
[00000012] 1e-07

0.00000000001;
[00000013] 1e-11

1e+3;
[00000014] 1000

1E-5;
[00000014] 1e-05
```

In order to make numbers with units (`'1min'`) and calling a method on a number (`'1.min'`), numbers that include a period must have a fractional part. In other words, `'1.'`, if not followed by digits, is always read as `'1 .'`:

```
1.;
[00004701:error] !!! syntax error: unexpected ;
```

Hexadecimal notation is supported for integers: `0x` followed by one or more hexadecimal digits, whose case is irrelevant.

```
0x2a == 42;
0x2A == 42;
0xabcdef == 11259375;
0xABCDEF == 11259375
```

Numbers with unknown suffixes are invalid tokens:

```
123foo;
[00005658:error] !!! syntax error: invalid token: '123foo'
12.3foo;
[00018827:error] !!! syntax error: invalid token: '12.3foo'
0xabcdef;
[00060432] 11259375
0xabcdefg;
[00061848:error] !!! syntax error: invalid token: '0xabcdefg'
```

19.1.6.5 Lists

Literal *lists* are represented with a comma-separated, potentially empty list of arbitrary expressions enclosed in square brackets (`[]`), as shown in the listing below. See [List \(??sec:std-List\)](#) for more details.

```

[]; // The empty list
[00000000] []
[1, 2, 3];
[00000000] [1, 2, 3]

```

19.1.6.6 Strings

String literals are enclosed in double quotes (") and can contain arbitrary characters, which stand for themselves, with the exception of the escape character, backslash (\), see below. The escapes sequences are defined in [Table 19.4](#).

\\	backslash
\"	double-quote
\a	bell ring
\b	backspace
\f	form feed
\n	line feed
\r	carriage return
\t	tabulation
\v	vertical tabulation
\[0-7]{1,3}	eight-bit character corresponding to a one-, two- or three-digit octal number. For instance, \0, \000 and 177. The matching is greedy: as many digits as possible are taken: \0, \000 are both resolved in the null character.
\x[0-9a-fA-F]{2}	eight-bit character corresponding to a two-digit hexadecimal number. For instance, 0xFF.
\B(<i>length</i>)(<i>content</i>)	binary blob. A <i>length</i> -long sequence of verbatim <i>content</i> . <i>length</i> is expressed in decimal. <i>content</i> is not interpreted in any way. The parentheses are part of the syntax, they are mandatory. For instance \B(2)(\B)

Table 19.4: String escapes

```

// Special characters.
"\\" == "\\";
"\" == "\";

// ASCII characters.
"\a" == "\007"; "\a" == "\x07";
"\b" == "\010"; "\b" == "\x08";
"\f" == "\014"; "\f" == "\x0c";
"\n" == "\012"; "\n" == "\x0a";
"\r" == "\015"; "\r" == "\x0d";
"\t" == "\011"; "\t" == "\x09";
"\v" == "\013"; "\v" == "\x0b";

```

```
// Octal escapes.
"\0" == "\00"; "\0" == "\000";
"\0000" == "\0""0";
"\062\063" == "23";

// Hexadecimal escapes.
"\x00" == "\0";
"\x32\x33" == "23";

// Binary blob escape.
"\B(3)("\")" == "\"\\\"";
```

Consecutive string literals are glued together into a single string. This is useful to split large strings into chunks that fit usual programming widths.

```
"foo" "bar" "baz" == "foobarbaz";
```

The interpreter prints the strings escaped; for instance, line feed will be printed out as `\n` when a string result is dumped and so forth. An actual line feed will of course be output if a string content is printed with `echo` for instance.

```
"";
[00000000] ""
"foo";
[00000000] "foo"
"a\nb"; // urbiscript escapes string when dumping them
[00000000] "a\nb"
echo("a\nb"); // We can see there is an actual line feed
[00000000] *** a
b
echo("a\\nb");
[00000000] *** a\nb
```

See [String \(??sec:std-String\)](#) for more details.

19.1.6.7 Tuples

Literal *tuples* are represented with a comma-separated, potentially empty list of arbitrary elements enclosed in parenthesis (`()`), as shown in the listing below. One extra comma can be added after the last element. To avoid confusion between a 1 member `Tuple` and a parenthesized expression, the extra comma must be added. See [Tuple \(??sec:std-Tuple\)](#) for more details.

```
();
[00000000] ()
(1,);
[00000000] (1,)
(1, 2);
[00000000] (1, 2)
(1, 2, 3, 4,);
[00000000] (1, 2, 3, 4)
```

19.1.7 Statement Separators

Sequential languages such as C++ support a single way to compose two statements: the sequential composition, “denoted” by ‘;’. To support concurrency and more fined tuned sequentiality, urbiscript features four different statement-separators (or connectors):

‘;’ sequentiality

‘|’ tight sequentiality

‘,’ background concurrency

‘&’ fair-start concurrency

19.1.7.1 ‘;’

The ‘;’-connector waits for the first statement to finish before starting the second statement. When used in the top-level interactive session, both results are displayed.

```
1; 2; 3;
[00000000] 1
[00000000] 2
[00000000] 3
```

19.1.7.2 ‘,’

The ‘,’-connector sends the first statement in background for concurrent execution, and starts the second statement when possible. When used in interactive sessions, the value of backgrounded statements are *not* printed — the time of their arrival being unpredictable, such results would clutter the output randomly. Use [Channels \(??sec:std-Channel\)](#) or [Events \(??sec:std-Event\)](#) to return results asynchronously.

```
{
  for (3)
  {
    sleep(1s);
    echo("ping");
  },
  sleep(0.5s);
  for (3)
  {
    sleep(1s);
    echo("pong");
  },
};
[00000316] *** ping
[00000316] *** pong
[00000316] *** ping
[00000316] *** pong
[00000316] *** ping
[00000316] *** pong
```

Both ‘;’ and ‘,’ have equal precedence. They are scoped too: the execution follow “waits” for the end of the jobs back-grounded with ‘,’ before proceeding. Compare the two following executions.

```
{
  sleep(100ms) | echo("1"),
  sleep(400ms) | echo("2"),
  echo("done");
};
[00000316] *** done
[00000316] *** 1
[00000316] *** 2
```

```
{
  sleep(100ms) | echo("1"),
  sleep(400ms) | echo("2"),
};
echo("done");
[00000316] *** 1
[00000316] *** 2
[00000316] *** done
```

19.1.7.3 ‘|’

When using the ‘;’ connector, the scheduler is allowed to run other commands between the first and the second statement. The ‘|’ does not yield between both statements. It is therefore more efficient, and, in a way, provides some atomicity for concurrent tasks.

```
{
  { echo("11") ; sleep(100ms) ; echo("12") },
  { echo("21") ; sleep(400ms) ; echo("22") },
};
[00000316] *** 11
[00000316] *** 21
[00000316] *** 12
[00000316] *** 22
```

```
{
  { echo("11") | echo("12") },
  { echo("21") | echo("22") },
};
[00000316] *** 11
[00000316] *** 12
[00000316] *** 21
[00000316] *** 22
```

In an interactive session, both statements must be “known” before launching the sequence. The value of the composed statement is the value of the second statement.

19.1.7.4 ‘&’

The ‘&’ is very similar to the ‘,’ connector, but for its precedence. Urbi expects to process the whole statement before launching the connected statements. This is especially handy in interactive sessions, as a means to fire a set of tasks concurrently.

19.1.8 Operators

urbiscript supports many *operators*, most of which are inspired from C++. Their syntax is presented here, and they are sorted and described with their original semantics — that is, + is an arithmetic operator that sums two numeric values. However, as in C++, these operators might be use for any other purpose — that is, + can also be used as the concatenation operator on lists and strings. Their semantics is thus not limited to what is presented here.

Tables in this section sort operators top-down, by precedence order. Group of rows (not separated by horizontal lines) describe operators that have the same precedence. Many operators are syntactic sugar that bounce on a method. In this case, the equivalent desugared expression is shown in the “Equivalence” column. This can help you determine what method to override to define an operator for an object (see [Section 9.6](#)).

This section defines the syntax, precedence and associativity of the operators. Their semantics is described in [Chapter 20](#) in the documentation of the classes that provide them.

19.1.8.1 Arithmetic operators

urbiscript supports classic *arithmetic operators*, with the classic semantics on numeric values. See [Table 19.5](#) and the listing below.

Oper.	Syntax	Assoc.	Semantics	Equivalence
+	+a	-	Identity	a.'+'()
-	-a	-	Opposite	a.'-'()
**	a ** b	Right	Exponentiation	a.'**'(b)
*	a * b	Left	Multiplication	a.'*(b)
/	a / b	Left	Division	a.'/(b)
%	a % b	Left	Modulo	a.'%(b)
+	a + b	Left	Sum	a.'+(b)
-	a - b	Left	Difference	a.'-(b)

Table 19.5: Arithmetic operators

```

1 + 1 == 2;
1 - 2 == -1;
2 * 3 == 6;
10 / 2 == 5;
2 ** 10 == 1024;
-(1 + 2) == -3;
1 + 2 * 3 == 7;

```

```
(1 + 2) * 3 == 9;
-2 ** 2 == -4;
- - - - 1 == 1;
```

19.1.8.2 Assignment operators

Assignment in urbiscript can be performed with the `=` operator. Shorthands such as `+=` exist; they are not redefinable since they are equivalent to a regular assignment combined with another operator. See [Table 19.6](#) and the listing below.

Oper.	Syntax	Assoc.	Semantics	Equivalence
<code>=</code>	<code>a = b</code>	Right	Assignment	<code>updateSlot("a", b)</code> ²
<code>+=</code>	<code>a += b</code>	Right	In place assignment	<code>a = a + b</code>
<code>-=</code>	<code>a -= b</code>	Right	In place assignment	<code>a = a - b</code>
<code>*=</code>	<code>a *= b</code>	Right	In place assignment	<code>a = a * b</code>
<code>/=</code>	<code>a /= b</code>	Right	In place assignment	<code>a = a / b</code>
<code>%=</code>	<code>a %= b</code>	Right	In place assignment	<code>a = a % b</code>
<code>^=</code>	<code>a ^= b</code>	Right	In place assignment	<code>a = a ^ b</code>

Table 19.6: Assignment operators

```
var y = 0;
[00000000] 0
y = 10;
[00000000] 10
y += 10;
[00000000] 20
y /= 5;
[00000000] 4
y++;
[00000000] 4
y;
[00000000] 5
```

19.1.8.3 Bitwise operators

urbiscript features *bitwise operators*. They are also used for other purpose than bit-related operations. See [Table 19.7](#) and the listing below.

```
4 << 2 == 16;
4 >> 2 == 1;
```

19.1.8.4 Logical operators

urbiscript supports the usual *Boolean operators*. See the table and the listing below. The operators `&&` and `||` are short-circuiting: their right-hand side is evaluated only if needed.

²For object fields only. Assignment to local variables cannot be redefined.

Oper.	Syntax	Assoc.	Semantics	Equivalence
<<	a << b	Left	Left bit shift	a. '<<'(b)
>>	a >> b	Left	Right bit shift	a. '>>'(b)
^	a ^ b	Left	Bitwise exclusive or	a. '^'(b)

Table 19.7: Bitwise operators

Oper.	Syntax	Assoc.	Semantics	Equivalence
!	!a	Left	Logical negation	a. '!'(b)
&&	a && b	Left	Logical and	a. '&&'(b)
	a b	Left	Logical or	a. ' (b)

Table 19.8: Boolean operators

```

true && true;
true || false;
!true == false;
true || (1 / 0);
(false && (1 / 0)) == false;

```

19.1.8.5 Comparison operators

urbiscript supports classical *comparison operators*. See [Table 19.9](#) and the listing below.

Oper.	Syntax	Assoc.	Semantics	Equivalence
==	a == b	None	Equality	a. '==(b)
!=	a != b	None	Inequality	a. '!=(b)
===	a === b	None	Physical equality	a. '===(b)
!==	a !== b	None	Physical inequality	a. '!=='(b)
~=	a ~= b	None	Relative approximate equality	a. '~=(b)
=~=	a =~= b	None	Absolute approximate equality	a. '=~=(b)
<	a < b	None	Less than	a. '<'(b)
<=	a <= b	None	Less than or equal to	a. '<='(b)
>	a > b	None	Greater than	a. '>'(b)
>=	a >= b	None	Greater than or equal to	a. '>='(b)

Table 19.9: Comparison operators

```

assert
{
  ! (0 < 0);
  0 <= 0;
  0 == 0;
  0 !== 0;
};

```

```

var z = 0;
[00000000] 0
assert
{
  z == z;
  ! (z != z);
};

```

19.1.8.6 Container operators

These operators work on containers and their members. See [Table 19.10](#).

Oper.	Syntax	Assoc.	Semantics	Equivalence
<code>in</code>	<code>a in b</code>	-	Membership	<code>b.has(a)</code>
<code>not in</code>	<code>a not in b</code>	-	Non-membership	<code>b.hasNot(a)</code>
<code>[]</code>	<code>a[args]</code>	-	Subscript	<code>a.'[]'(args)</code>
<code>[] =</code>	<code>a[args] = v</code>	-	Subscript assignment	<code>a.'[]='(args, v)</code>

Table 19.10: Container operators

The *in* and *not in* operators test the membership of an element in a container. They bounce to the container's `has` and `hasNot` methods (see [Container \(??sec:std-Container\)](#)). They are non-associative.

```

1   in [0, 1, 2];
3 not in [0, 1, 2];

"one"   in   ["zero" => 0, "one" => 1, "two" => 2];
"three" not in ["zero" => 0, "one" => 1, "two" => 2];

```

The following operators use an index. Note that the *subscript* (square bracket) operator is *variadic*: it takes any number of arguments that will be passed to the `'[]'` method of the targeted object.

```

// On lists.
var l = [1, 2, 3, 4, 5];
[00000000] [1, 2, 3, 4, 5]
assert
{
  l[0] == 1;
  l[-1] == 5;
  (l[0] = 10) == 10;
  l == [10, 2, 3, 4, 5];
};

// On strings.
var s = "abcdef";
[00000005] "abcdef"
assert
{

```

```
s[0] == "a";
s[1,3] == "bc";
(s[1,3] = "foo") == "foo";
s == "afoodef";
};
```

19.1.8.7 Object operators

These core operators provide access to slots and their properties. See [Table 19.11](#).

Oper.	Syntax	Assoc.	Semantics	Equivalence
.	a.b	-	Message sending	Not redefinable
.	a.b(args)	-	Message sending	Not redefinable
->	a->b	-	Property access	getProperty("a", "b")
->	a->b = v	-	Property assignment	setProperty("a", "b", v)

Table 19.11: Object operators

19.1.8.8 All operators summary

[Table 19.12](#) is a summary of all operators, to highlight the overall precedences. Operators are sorted by decreasing precedence. Groups of rows represent operators with the same precedence.

19.2 Scopes and local variables

19.2.1 Scopes

Scopes are sequences of statements, enclosed in curly brackets (`{}`). Statements are separated with the four statements separators (see [Section 19.1.7](#)). A trailing `;` or `,` is ignored. A trailing `&` or `|` behaves as if `& {}` or `| {}` was used. This particular case is heavily used by urbiscript programmers to discard the value of an expression:

```
// Return value is 1. Displayed.
1;
[00000000] 1
// Return value is that of {}, i.e., void. Nothing displayed.
1 | {};
// Same as "1 | {}", a valueless expression.
1|;
```

Scopes are themselves expressions, and can thus be used in composite expressions, nested, and so forth.

```
// Scopes evaluate to their last expression
{
  1;
```

Oper.	Syntax	Assoc.	Semantics	Equivalence
.	a.b	-	Message sending	Not redefinable
.	a.b(args)	-	Message sending	Not redefinable
->	a->b	-	Property access	getProperty("a", "b")
->	a->b = v	-	Property assignment	setProperty("a", "b", v)
[]	a[args]	-	Subscript	a.'[]' (args)
[] =	a[args] = v	-	Subscript assignment	a.'[]=' (args, v)
+	+a	-	Identity	a.'+' ()
-	-a	-	Opposite	a.'-' ()
**	a ** b	Right	Exponentiation	a.'**' (b)
*	a * b	Left	Multiplication	a.'*' (b)
/	a / b	Left	Division	a.'/' (b)
%	a % b	Left	Modulo	a.'%' (b)
+	a + b	Left	Sum	a.'+' (b)
-	a - b	Left	Difference	a.'-' (b)
<<	a << b	Left	Left bit shift	a.'<<' (b)
>>	a >> b	Left	Right bit shift	a.'>>' (b)
==	a == b	None	Equality	a.'==' (b)
!=	a != b	None	Inequality	a.'!=' (b)
===	a === b	None	Physical equality	a.'===' (b)
!==	a !== b	None	Physical inequality	a.'!==' (b)
=~	a =~ b	None	Absolute approximate equality	a.'=~' (b)
~	a ~ b	None	Relative approximate equality	a.'~' (b)
<	a < b	None	Less than	a.'<' (b)
<=	a <= b	None	Less than or equal to	a.'<=' (b)
>	a > b	None	Greater than	a.'>' (b)
>=	a >= b	None	Greater than or equal to	a.'>=' (b)
^	a ^ b	Left	Bitwise exclusive or	a.'^' (b)
!	!a	Left	Logical negation	a.'!' ()
in	a in b	-	Membership	b.has(a)
not in	a not in b	-	Non-membership	b.hasNot(a)
&&	a && b	Left	Logical and	a.'&&' (b)
	a b	Left	Logical or	a.' ' (b)
=	a = b	Right	Assignment	updateSlot("a", b)
+=	a += b	Right	In place assignment	a = a + b
-=	a -= b	Right	In place assignment	a = a - b
*=	a *= b	Right	In place assignment	a = a * b
/=	a /= b	Right	In place assignment	a = a / b
%=	a %= b	Right	In place assignment	a = a % b
^=	a ^= b	Right	In place assignment	a = a ^ b
++	a++	-	Incrementation	(a = a + 1) - 1
--	a--	-	Incrementation	(a = a - 1) + 1

Table 19.12: Operators summary

```

2;
3; // This last separator is optional.
};
[00000000] 3
// Scopes can be used as expressions
{1; 2; 3} + 1;
[00000000] 4

```

19.2.2 Local variables

Local variables are introduced with the `var` keyword, followed by an identifier (see [Section 19.1.4](#)) and an optional initialization value assignment. If the initial value is omitted, it defaults to `void` (`??sec:std-void`). Variable declarations evaluate to the initialization value. They can later be referred to by their name. Their value can be changed with the assignment operator; such an assignment expression returns the new value. The use of local variables is illustrated below.

```

// This declare variable x with value 42, and evaluates to 42.
var t = 42;
[00000000] 42
// x equals 42
t;
[00000000] 42
// We can assign it a new value
t = 51;
[00000000] 51
t;
[00000000] 51
// Initialization defaults to void
var u;
u.isVoid;
[00000000] true

```

The lifespan of local variables is the same as their enclosing scope. They are thus only accessible from their scope and its sub-scopes³. Two variables with the same name cannot be defined in the same scope. A variable with the same name can be defined in an inner scope, in which case references refer to the innermost variable, as shown below.

```

{
  var x = "x";
  var y = "outer y";
  {
    var y = "inner y";
    var z = "z";
    // We can access variables of parent scopes.
    echo(x);
    // This refers to the inner y.
    echo(y);
    echo(z);
  }
};

```

³Local variables can actually escape their scope with lexical closures, see [Section 19.3.6](#).

```

// This refers to the outer y.
echo(y);
// This would be invalid: z does not exist anymore.
// echo(z);
// This would be invalid: x is already declared in this scope.
// var x;
};
[00000000] *** x
[00000000] *** inner y
[00000000] *** z
[00000000] *** outer y

```

19.3 Functions

19.3.1 Function Definition

Functions in urbiscript are first class citizens: a function is a value, like floats and strings, and can be handled as such. This is different from most C-like languages. One can create a functional value thanks to the `function` keyword, followed by the list of formal arguments and a compound statement representing the body of the function. Formal arguments are a possibly-empty comma-separated list of identifiers. Non-empty lists of formal arguments may optionally end with a trailing comma. The listing below illustrates this.

```

function () { echo(0) };
[00000000] function () {
    echo(0)
}

function (arg1, arg2) { echo(0) };
[00000000] function (var arg1, var arg2) {
    echo(0)
}

function (
    arg1, // Ignored argument.
    arg2, // Also ignored.
)
{
    echo(0)
};
[00000000] function (var arg1, var arg2) {
    echo(0)
}

```

Usually functions are bound to an identifier to be invoked later. The listing below shows a short-hand to define a named function.

```

// Functions are often stored in variables to call them later.
var f1 = function () {
    echo("hello")
}

```

```
f1();
[00000000] *** hello

// This form is strictly equivalent, yet simpler.
function f2()
{
  echo("hello")
}|
f2();
[00000000] *** hello
```

19.3.2 Arguments

The list of formal arguments defines the number of argument the function requires. They are accessible by their name from within the body. If the list of formal arguments is omitted, the number of effective arguments is not checked, and arguments themselves are not evaluated. Arguments can then be manipulated with the call message, explained below.

```
var f = function(a, b) {
  echo(b + a);
}|
f(1, 0);
[00000000] *** 1
// Calling a function with the wrong number of argument is an error.
f(0);
[00000000:error] !!! f: expected 2 arguments, given 1
f(0, 1, 2);
[00000000:error] !!! f: expected 2 arguments, given 3
```

Non-empty lists of effective arguments may end with an optional comma.

```
f(
  "bar",
  "foo",
);
[00000000] *** foobar
```

19.3.3 Return value

The *return value* of the function is the evaluation of its body — that is, since the body is a scope, the last evaluated expression in the scope. Values can be returned manually with the `return` keyword followed by the value, in which case the evaluation of the function is stopped. If `return` is used with no value, the function returns `void`.

```
function g1(a, b)
{
  echo(a);
  echo(b);
  a // Return value is a
}|
```

```

g1(1, 2);
[00000000] *** 1
[00000000] *** 2
[00000000] 1

function g2(a, b)
{
    echo(a);
    return a; // Stop execution at this point and return a
    echo(b); // This is not executed
}|
g2(1, 2);
[00000000] *** 1
[00000000] 1

function g3()
{
    return; // Stop execution at this point and return void
    echo(0); // This is not executed
}|
g3(); // Returns void, so nothing is printed.

```

19.3.4 Call messages

Functions can access meta-information about how they were called, through a `CallMessage` object. The *call message* associated with a function can be accessed with the `call` keyword. It contains several information such as not-yet evaluated arguments, the name of the function, the target ...

19.3.5 Strictness

urbiscript features two different function calls: *strict* function calls, effective arguments are evaluated before invoking the function, and *lazy* function calls, arguments are passed as-is to the function. As a matter of fact, the difference is rather that there are strict functions and lazy functions.

Functions defined with a (possibly empty) list of formal arguments in parentheses are strict: the effective arguments are first evaluated, and then their value is given to the called function.

```

function first1(a, b) {
    echo(a); echo(b)
}|
first1({echo("Arg1"); 1},
      {echo("Arg2"); 2});
[00000000] *** Arg1
[00000000] *** Arg2
[00000000] *** 1
[00000000] *** 2

```


A function declared with no formal argument list is lazy. Use its call message to manipulate its *unevaluated* arguments. The listing below gives an example. More information about this can be found in the [CallMessage \(??sec:std-CallMessage\)](#) class documentation.

```
function first2
{
  echo(call.evalArgAt(0));
  echo(call.evalArgAt(1));
}|
first2({echo("Arg1"); 1},
      {echo("Arg2"); 2});
[00000000] *** Arg1
[00000000] *** 1
[00000000] *** Arg2
[00000000] *** 2
```

A lazy function may implement a strict interface by evaluating its arguments and storing them as local variables, see below. This is less efficient than defining a strict function.

```
function first3
{
  var a = call.evalArgAt(0);
  var b = call.evalArgAt(1);
  echo(a); echo(b);
}|
first3({echo("Arg1"); 1},
      {echo("Arg2"); 2});
[00000000] *** Arg1
[00000000] *** Arg2
[00000000] *** 1
[00000000] *** 2
```

19.3.6 Lexical closures

Lexical closures are an additional scoping rule, with which a function can refer to a local variable located outside the function — but still in the current context. urbiscript supports read/write lexical closures, meaning that the variable is shared between the function and the outer environment, as shown below.

```
var n = 0|
function cl()
{
  // x refers to a variable outside the function
  n++;
  echo(n);
}|
cl();
[00000000] *** 1
n;
[00000000] 1
n++;
[00000000] 1
```

```
cl();
[00000000] *** 3
```

The following listing illustrate that local variables can even escape their declaration scope by lexical closure.

```
function wrapper()
{
  // Normally, x is local to 'wrapper', and is limited to this scope.
  var x = 0;
  at (x > 1)
    echo("ping");
  // Here we make it escape the scope by returning a closure on it.
  return function() { x++ };
} |

var w = wrapper() |
w();
[00000000] 0
w();
[00000000] 1
[00000000] *** ping
```

19.4 Objects

Any value in urbiscript is an object. Objects contain:

- A list of prototypes, which are also objects.
- A list of slots, which to a name associate an object.

19.4.1 Slots

19.4.1.1 Manipulation

Objects can contain any number of *slots*, every slot has a name and a value. Slots are often called “fields”, “attributes” or “members” in other object-oriented languages.

The `createSlot` function adds a slot to an object with the void ([Section 20.69](#)) value. The `updateSlot` function changes the value of a slot; `getSlot` reads it. The `setSlot` method creates a slot with a given value. Finally, the `localSlotNames` method returns the list of the object slot’s name. The listing below shows how to manipulate slots. More documentation about these methods can be found in [Section 20.37](#).

```
var o = Object.new |
o.localSlotNames;
[00000000] []
o.createSlot("test");
o.localSlotNames;
[00000000] ["test"]
o.getSlot("test").asString;
```

```
[00000000] "void"
o.updateSlot("test", 42);
[00000000] 42
o.getSlot("test");
[00000000] 42
```

19.4.1.2 Syntactic Sugar

There is some syntactic sugar for slot methods:

- `var o.name` is equivalent to `o.createSlot("name")`.
- `var o.name = value` is equivalent to `o.setSlot("name", value)`.
- `o.name = value` is equivalent to `o.updateSlot("name", value)`.

19.4.2 Prototypes

19.4.2.1 Manipulation

urbiscript is a prototype-based language, unlike most classical object oriented language, which are class-based. In prototype-based languages, objects have no type, only *prototypes*, from which they inherit behavior.

urbiscript objects can have several prototypes. The list of prototypes is given by the `protos` method; they can be added or removed with `addProto` and `removeProto`. See [Section 20.37](#) for more documentation.

```
var ob = Object.new|
ob.protos;
[00000000] [Object]
ob.addProto(Pair);
[00000000] (nil, nil)
ob.protos;
[00000000] [(nil, nil), Object]
ob.removeProto(Object);
[00000000] (nil, nil)
ob.protos;
[00000000] [(nil, nil)]
```

19.4.2.2 Inheritance

Objects inherit their prototypes' slots: `getSlot` will also look in an object prototypes' slots. `getSlot` performs a depth-first traversal of the prototypes hierarchy to find slots. That is, when looking for a slot in an object:

- `getSlot` checks first if the object itself has the requested slot. If so, it returns its value.

- Otherwise, it applies the same research on every prototype, in the order of the prototype list (since `addProto` inserts in the front of the prototype list, the last prototype added has priority). This search is recursive: `getSlot` will also look in the first prototype's prototype, etc before looking in the second prototype. If the slot is found in a prototype, it is returned.
- Finally, if no prototype had the slot, an error is raised.

This listing shows how slots are inherited.

```
var a = Object.new|
var b = Object.new|
var c = Object.new|
a.setSlot("x", "slot in a")|
b.setSlot("x", "slot in b")|
// c has no "x" slot
c.getSlot("x");
[00000000:error] !!! lookup failed: x
// c can inherit the "x" slot from a.
c.addProto(a)|
c.getSlot("x");
[00000000] "slot in a"
// b is prepended to the prototype list, and has thus priority
c.addProto(b)|
c.getSlot("x");
[00000000] "slot in b"
// a local slot in c has priority over prototypes
c.setSlot("x", "slot in c")|
c.getSlot("x");
[00000000] "slot in c"
```

19.4.2.3 Copy on write

The `updateSlot` method has a particular behavior with respect to prototypes. Although performing an `updateSlot` on a non existent slot is an error, it is valid if the slot is inherited from a prototype. In this case, the slot is however not updated in the prototype, but rather created in the object itself, with the new value. This process is called *copy on write*; thanks to it, prototypes are not altered when the slot is overridden in a child object.

```
var p = Object.new|
var p.slot = 0|
var d = Object.new|
d.addProto(p)|
d.slot;
[00000000] 0
d.slot = 1;
[00000000] 1
// p's slot was not altered
p.slot;
[00000000] 0
// It was copied in d
```

```
d.slot;
[00000000] 1
```

19.4.3 Sending messages

A *message* in urbiscript consists in a message name and arguments. One can send a message to an object with the dot (.) operator, followed by the message name (which can be any valid identifier) and the arguments, as shown below. When there are no arguments, the parentheses can be omitted. As you might see, sending messages is very similar to calling methods in classical languages.

```
// Send the message msg to object obj, with arguments arg1 and arg2.
obj.msg(arg1, arg2);
// Send the message msg to object obj, with no arguments.
obj.msg();
// This is strictly equivalent.
obj.msg;
```

When a message *msg* is sent to object *obj*:

- The *msg* slot of *obj* is retrieved (i.e., *obj.getSlot("msg")*). If the slot is not found, the classic lookup error is raised.
- If the object is a *routine* (either a primitive, written in C++ for instance, or a function implemented in urbiscript), it is invoked with the message arguments, and the returned value is the result. As a consequence, the number of arguments in the message sending must match the one required by the routine.
- Otherwise (the object is not a routine), this object is the result of the message sending. There must be no argument.

Such message sending is illustrated below.

```
var obj = Object.new|
var obj.a = 42|
var obj.b = function (x) { x + 1 }|
obj.a;
[00000000] 42
obj.a();
[00000000] 42
obj.a(50);
[00000000:error] !!! a: expected 0 argument, given 1
obj.b;
[00000000:error] !!! b: expected 1 argument, given 0
obj.b();
[00000000:error] !!! b: expected 1 argument, given 0
obj.b(50);
[00000000] 51
```

19.5 Structural Pattern Matching

Structural *pattern matching* is useful to deconstruct tuples, lists and dictionaries with a small and readable syntax.

These patterns can be used in the following clauses:

- The left hand side of an assignment.
- `case`
- `catch`
- `at`
- `waituntil`
- `whenever`

The following examples illustrate the possibilities of *structural pattern matching* inside `case` clauses:

```
switch ( ("foo", [1, 2]) )
{
  // The pattern does not match the values of the list.
  case ("foo", [2, 1]):
    echo("fail");

  // The pattern does not match the tuple.
  case ["foo", [1, 2]]:
    echo("fail");

  // The pattern matches and binds the variable "l"
  // but the condition is not verified.
  case ("foo", var l) if l.size == 0:
    echo("fail");

  // The pattern matches.
  case ("foo", [var a, var b]):
    echo("foo(%s, %s)" % [a, b]);
};

[00000000] *** foo(1, 2)
```

19.5.1 Basic Pattern Matching

Matching is used in many locations and allows to match literal values (e.g., `List` (`??sec:std-List`), `Tuple` (`??sec:std-Tuple`), `Dictionary` (`??sec:std-Dictionary`), `Float` (`??sec:std-Float`), `String` (`??sec:std-String`)). In the following expressions each pattern (on the left hand side) matches the value (on the right hand side).

```
(1, "foo") = (1, "foo");
[00000000] (1, "foo")
[1, "foo"] = [1, "foo"];
[00000000] [1, "foo"]
["b" => "foo", "a" => 1] = ["a" => 1, "b" => "foo"];
[00000000] ["a" => 1, "b" => "foo"]
```

A `Exception.MatchFailure` exception is thrown when a pattern does not match.

```
try
{
    (1, 2) = (3, 4)
}
catch (var e if e.isA(Exception.MatchFailure))
{
    e.message
};
[00000000] "pattern did not match"
```

19.5.2 Variable

Patterns can contain variable declarations, to match any value and to bind it to a new variable.

```
{
    (var a, var b) = (1, 2);
    echo("a = %d, b = %d" % [a, b]);
};
[00000000] *** a = 1, b = 2
{
    [var a, var b] = [1, 2];
    echo("a = %d, b = %d" % [a, b]);
};
[00000000] *** a = 1, b = 2
{
    ["b" => var b, "a" => var a] = ["a" => 1, "b" => 2, "c" => 3];
    echo("a = %d, b = %d" % [a, b]);
};
[00000000] *** a = 1, b = 2
```

19.5.3 Guard

Patterns used inside a `switch`, a `catch` or an event catching construct accept guards.

Guard are used by appending a `if` after a pattern or after a matched event.

The following example is inspired from the [TrajectoryGenerator](#) ([??sec:std-TrajectoryGenerator](#)) where a [Dictionary](#) ([??sec:std-Dictionary](#)) is used to set the trajectory type.

```
switch (["speed" => 2, "time" => 6s])
{
    case ["speed" => var s] if s > 3:
        echo("Too fast");
```

```

case ["speed" => var s, "time" => var t] if s * t > 10:
    echo("Too far");
};
[00000000] *** Too far

```

The same guard are available for `catch` statement.

```

try
{
    throw ("message", 0)
}
catch (var e if e.isA(Exception))
{
    echo(e.message)
}
catch ((var msg, var value) if value.isA(Float))
{
    echo("%s: %d" % [msg, value])
};
[00000000] *** message: 0

```

Events catchers can have guards on the pattern arguments. You can add these inside `at`, `whenever` and `waituntil` statements.

```

{
    var e = Event.new;
    at (e?(var msg, var value) if value % 2 == 0)
        echo("%s: %d" % [msg, value]);

    // Does not trigger the "at" because the guard is not verified.
    e!("message", 1);

    // Trigger the "at".
    e!("message", 2);
};
[00000000] *** message: 2

```

19.6 Imperative flow control

19.6.1 break

When encountered within a `for` or a `while` loop, `break` makes the execution jump after the loop.

```

var i = 5|
for (; true; echo(i))
{
    if (i > 8)
        break;
    i++;
};
[00000000] *** 6

```



```
[00000000] *** 7
[00000000] *** 8
[00000000] *** 9
```

19.6.2 continue

When encountered within a `for` or a `while` loop, `continue` short-circuits the rest of the loop-body, and runs the next iteration (if there remains one).

```
for (var i = 0; i < 8; i++)
{
  if (i % 2 != 0)
    continue;
  echo(i);
};
[00000000] *** 0
[00000000] *** 2
[00000000] *** 4
[00000000] *** 6
```

19.6.3 do

The `do` construct changes the target (`this`) when evaluating an expression. It is a convenient means to avoid repeating the same target several times.

```
do (target)
{
  body
};
```

It evaluates `body`, with `this` being `target`, as shown below. The whole construct evaluates to the value of `body`.

```
do (1024)
{
  assert(this == 1024);
  assert(sqrt == 32);
  setSlot("y", 23);
}.y;
[00000000] 23
```

19.6.4 if

As in most programming languages, conditionals are expressed with `if`.

```
if (condition) then-clause
if (condition) then-clause else else-clause
```

First `condition` is evaluated; if it evaluates to a value which is true ([Section 20.3.3](#)), evaluate `then-clause`, otherwise, if applicable, evaluate `else-clause`.

```

if (true) assert(true) else assert(false);
if (false) assert(false) else assert(true);
if (true) assert(true);

```

Beware that *there must not be a terminator after the `then-clause`*:

```

if (true)
  assert(true);
else
  assert(false);
[00000002:error] !!! syntax error: unexpected else

```

Contrary to C/C++, it has value: it also implements the *condition ? then-clause : else-clause* construct. Unfortunately, due to syntactic constraints inherited from C, it is a *statement*: it cannot be used directly as an expression. But as everywhere else in urbiscript, to use a statement where an expression is expected, use braces:

```

assert(1 + if (true) 3 else 4 == 4);
[00000003:error] !!! syntax error: unexpected if
assert(1 + { if (true) 3 else 4 } == 4);

```

The *condition* can be any statement list. Variables which it declares are visible in both the *then-clause* and the *else-clause*, but do not escape the `if` construct.

```

assert({if (false) 10 else 20} == 20);
assert({if (true) 10 else 20} == 10);

assert({if (true) 10      } == 10);

assert({if (var x = 10) x + 2 else x - 2} == 12);
assert({if (var x = 0)  x + 2 else x - 2} == -2);

if (var xx = 123) xx | xx;
[00000005:error] !!! lookup failed: xx

```

19.6.5 for

`for` comes in several flavors.

19.6.5.1 C-like for

urbiscript support the classical C-like `for` construct.

```

for (initialization; condition; increment)
  body

```

It has the exact same behavior as C's `for`:

1. The *initialization* is evaluated.
2. *condition* is evaluated. If the result is false, executions jump after `for`.

3. *body* is evaluated. If `continue` is encountered, execution jumps to point 4. If `break` is encountered, execution jumps after the `for`.
4. The *increment* is evaluated.
5. Execution jumps to point 2.
6. The loop evaluates to `void`.

19.6.5.2 Range-for

urbiscript supports iteration over a collection with another form of the `for` loop.

```
for (var name : collection)
  body;
```

It evaluates *body* for each element in *collection*. The loop evaluates to `void`. Inside *body*, the current element is accessible via the *name* local variable. The listing below illustrates this.

```
for (var x : [0, 1, 2, 3, 4])
  echo(x.sqr);
[00000000] *** 0
[00000000] *** 1
[00000000] *** 4
[00000000] *** 9
[00000000] *** 16
```

This form of `for` simply sends the `each` message to *collection* with one argument: the function that takes the current element and performs *action* over it. Thus, you can make any object acceptable in a `for` by defining an adequate `each` method.

```
var Hobbits = Object.new|
function Hobbits.each (action)
{
  action("Frodo");
  action("Merry");
  action("Pippin");
  action("Sam");
}|
for (var name in Hobbits)
  echo("%s is a hobbit." % [name]);
[00000000] *** Frodo is a hobbit.
[00000000] *** Merry is a hobbit.
[00000000] *** Pippin is a hobbit.
[00000000] *** Sam is a hobbit.
// This for statement is equivalent to:
Hobbits.each(function (name) { echo("%s is a hobbit." % [name]) });
[00000000] *** Frodo is a hobbit.
[00000000] *** Merry is a hobbit.
[00000000] *** Pippin is a hobbit.
[00000000] *** Sam is a hobbit.
```

19.6.5.3 for n -times

urbiscript provides some support for simple replication of computations: it allow to repeat a loop body n -times. With the exception that the loop index is not available within the body, `for (n)` is equivalent to `for (var i: n)`. It supports the same flavors: `for;`, `for|`, and `for&`. The loop evaluates to `void`.

```
{ var res = []; for (3) { res << 1; res << 2 } ; res }
  == [1, 2, 1, 2, 1, 2];

{ var res = []; for|(3) { res << 1; res << 2 } ; res }
  == [1, 2, 1, 2, 1, 2];

{ var res = []; for&(3) { res << 1; res << 2 } ; res }
  == [1, 1, 1, 2, 2, 2];
```

Note that since these `for` loops are merely anonymous foreach-style loops, the argument needs not being an integer, any iterable value can be used.

```
3 == { var r = 0; for ([1, 2, 3]) r += 1; r};
3 == { var r = 0; for ("123")    r += 1; r};
```

19.6.6 if

urbiscript supports the usual `if` constructs.

```
if (condition)
  action;

if (condition)
  action
else
  otherwise;
```

If the *condition* evaluation is true, *action* is evaluated. Otherwise, in the latter version, *otherwise* is executed. Contrary to C/C++, there *must not* be a semicolon after the *action*; it would end the `if/else` construct prematurely.

19.6.7 loop

Endless loops can be created with `loop`, which is equivalent to `while (true)`. The loop evaluates to `void`. Both sequential flavors, `loop;` and `loop|`, are supported. The default flavor is `loop;`.

```
{
  var n = 10|;
  var res = []|;
  loop;
  {
    n--;
    res << n;
    if (n == 0)
```

```

        break
    };
    res
}
==
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0];

```

```

{
    var n = 10|;
    var res = []|;
    loop|
    {
        n--;
        res << n;
        if (n == 0)
            break
    };
    res
}
==
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0];

```

19.6.8 switch

The `switch` statement in urbiscript is similar to C's one.

```

switch (value)
{
    case value_one:
        action_one;
    case value_two:
        action_two;
    //case ...:
    // ...
    default:
        default_action;
};

```

It might contain an arbitrary number of cases, and optionally a default case. The *value* is evaluated first, and then the result is compared sequentially with the evaluation of all cases values, with the `==` operator, until one comparison is true. If such a match is found, the corresponding action is executed, and execution jumps after the `switch`. Otherwise, the default case — if any — is executed, and execution jumps after the switch. The switch itself evaluates to case that was evaluated, or to void if no match was found and there's no default case. The listing below illustrates `switch` usage.

Unlike C, there are no `break` to end `case` clauses: execution will never span over several cases. Since the comparisons are performed with the generic `==` operator, `switch` can be performed on any comparable data type.

```

function sw(v)

```

```

{
  switch (v)
  {
    case "":
      echo("Empty string");
    case "foo":
      "bar";
    default:
      v[0];
  }
};
sw("");
[00000000] *** Empty string
sw("foo");
[00000000] "bar"
sw("foobar");
[00000000] "f"

```

19.6.9 while

The `while` loop is similar to C's one.

```

while (condition)
  body;

```

If *condition* evaluation, is true, *body* is evaluated and execution jumps before the `while`, otherwise execution jumps after the `while`.

```

var j = 3;
while (0 < j)
{
  echo(j);
  j--;
};
[00000000] *** 3
[00000000] *** 2
[00000000] *** 1

```

The default flavor for `while` is `while;`.

19.6.9.1 while;

The semantics of

```

while; (condition)
  body;

```

is the same as

```

condition | body ; condition | body ; ...

```

as long as *cond* evaluates to true, or until `break` is invoked. If `continue` is evaluated, the rest of the body is skipped, and the next iteration is started.

```

{
  var i = 4|
  while (true)
  {
    i -= 1;
    echo ("in: " + i);
    if (i == 1)
      break
    else if (i == 2)
      continue;
    echo ("out: " + i);
  };
};
[00000000] *** in: 3
[00000000] *** out: 3
[00000000] *** in: 2
[00000000] *** in: 1

```

19.6.9.2 while—

The semantics of

```

while| (condition)
  body;

```

is the same as

```

condition | body | condition | body | ...

```

The execution is can be controlled by `break` and `continue`.

```

{
  var i = 4|
  while| (true)
  {
    i -= 1;
    echo ("in: " + i);
    if (i == 1)
      break
    else if (i == 2)
      continue;
    echo ("out: " + i);
  };
};
[00000000] *** in: 3
[00000000] *** out: 3
[00000000] *** in: 2
[00000000] *** in: 1

```

19.7 Exceptions

19.7.1 Throwing exceptions

Use the `throw` keyword to *throw exceptions*, as shown below. Thrown exceptions will break the execution upward until they are caught, or until they reach the top-level — as in C++. Contrary to C++, exceptions reaching the top-level are printed, and won't abort the kernel — other and new connections will continue to execute normally.

```
throw 42;
[00000000:error] !!! 42
function inner() { throw "exn" } |
function outer() { inner() }|
// Exceptions propagate to parent call up to the top-level
outer();
[00000000:error] !!! exn
[00000000:error] !!!      called from: 3.20-26: inner
[00000000:error] !!!      called from: 4.1-7: outer
```

19.7.2 Catching exceptions

Exceptions are *caught* with the `try/catch` construct. It consists of a first block (the *try-block*), from which we want to catch exceptions, and one or more catch clauses to stop the exception (*catch-blocks*). Each catch clause defines a pattern against which the thrown exception is matched. If no pattern is specified, the catch clause matches systematically (equivalent to `catch (...)` in C++).

Exceptions thrown from the `try` block are matched sequentially against all catch clauses. The first matching clause is executed, and control jumps after the whole `try/catch` block. If no catch clause matches, the exceptions isn't stopped and continues upward.

```
function test(e)
{
  try
  { throw e; }
  catch (0)
  { echo("zero") }
  catch ([var x, var y])
  { echo(x + y) }
} | {};
test(0);
[00002126] *** zero
test([22, 20]);
[00002131] *** 42
test(51);
[00002143:error] !!! 51
[00002143:error] !!!      called from: 12.1-8: test
```


19.7.3 Inspecting exceptions

An [Exception](#) ([??sec:std-Exception](#)) is a regular object, on which introspection can be performed.

```
try
{
  Math.cos(3,1415);
}
catch (var e)
{
  echo("Exception type: %s" % e.type);
  if (e.isA(Exception.Arity))
  {
    echo("Routine: %s" % e.routine);
    echo("Number of effective arguments: %s" % e.effective);
  }
};
[00000132] *** Exception type: Arity
[00000133] *** Routine: cos
[00000134] *** Number of effective arguments: 2
```

19.8 Assertions

Assertions allow to embed consistency checks in the code. They are particularly useful when developing a program since they allow early catching of errors. Yet, they can be costly in production mode: the run-time cost of verifying every single assertion might be prohibitive. Therefore, as in C-like languages, assertions are disabled when [System.ndebug](#) is true, see [System](#) ([??sec:std-System](#)).

urbiscript supports assertions in two different ways: with a function-like syntax, which is adequate for single claims, and a block-like syntax, to group claims together.

19.8.1 Asserting an Expression

```
assert(true);
assert(42);
```

Failed assertions are displayed in a user friendly fashion: first the unevaluated assertion is displayed, then the effective values are reported.

```
function fail () { false }|;
assert (fail);
[00010239:error] !!! failed assertion: fail

function lazyFail { call.evalArgAt(0); false }|;
assert (lazyFail(1+2, "+" * 2));
[00010241:error] !!! failed assertion: lazyFail(1.'+'(2), "+".'*(2)) (! lazyFail(3, ?))
```

The following example is more realistic.

```

function areEqual
{
  var res = true;
  if (!call.args.empty)
  {
    var args = call.evalArgs;
    var a = args[0];
    for (var b : args.tail)
      if (a != b)
      {
        res = false;
        break;
      }
  };
  res
}|;
assert (areEqual);
assert (areEqual(1));
assert (areEqual(1, 0 + 1));
assert (areEqual(1, 1, 1+1));
[00001388:error] !!! failed assertion: areEqual(1, 1, 1.'+'(1)) (! areEqual(1, 1, 2))
assert (areEqual(1+2, 3+3, 4*6));
[00001393:error] !!! failed assertion: areEqual(1.'+'(2), 3.'+'(3), 4.*'(6)) (! areEqual(3, 6, 24))

```

Comparison operators are recognized, and displayed specially:

```

assert(1 == 1 + 1);
[00000002:error] !!! failed assertion: 1 == 1.'+'(1) (1 != 2)

```

Note however that if opposite comparison operators are absurd (i.e., if for instance `a == b` is not true, but `a != b` is not true either), then the message is unlikely to make sense.

19.8.2 Assertion Blocks

Groups of assertions are more readable when used with the `assert{exp1; exp2; ...}` construct. The (possibly empty) list of claims may be ended with a semicolon.

```

assert
{
  true;
  42;
  1 == 1 + 1;
};
[00000002:error] !!! failed assertion: 1 == 1.'+'(1) (1 != 2)

```

For sake of readability and compactness, this documentation shows assertion blocks as follows.

```

true;
42;
1 == 1 + 1;
[00000002:error] !!! failed assertion: 1 == 1.'+'(1) (1 != 2)

```

19.9 Parallel and event-based flow control

19.9.1 `at`

Using the `at` construct, one can arm code that will be triggered each time some condition is true.

The `at` construct is as follows:

```
at (condition)
    statement1
onleave
    statement2
```

The *condition* can be of two different kinds: *e?(args)* to catch when events are sent, or *exp* to catch each time a Boolean *exp* becomes true.

The `onleave statement2` part is optional. Note that, as is the case for the `if` statement, there must not be a semicolon after *statement1* if there is an `onleave` clause.

19.9.1.1 `at` on Events

See [Section 12.2](#) for an example of using `at` statements to watch events.

19.9.1.2 `at` on Boolean Expressions

The `at` construct can be used to watch a given Boolean expression.

```
var x = 0 |
var x_is_two = false |
at (x == 2)
    x_is_two = true
onleave
    x_is_two = false;

x = 3 |; assert(!x_is_two);
x = 2 |; assert( x_is_two);
x = 2 |; assert( x_is_two);
x = 3 |; assert(!x_is_two);
```

It can also wait for some condition to hold long enough: *exp ~ duration*, as a condition, denotes the fact that *exp* was true for *duration* seconds.

```
var x = 0 |
var x_was_two_for_two_seconds = false |
at (x == 2 ~ 2s)
    x_was_two_for_two_seconds = true
onleave
    x_was_two_for_two_seconds = false;

x = 2          | assert(!x_was_two_for_two_seconds);
sleep(1.5s)    | assert(!x_was_two_for_two_seconds);
sleep(1.5s)    | assert( x_was_two_for_two_seconds);
```

```
x = 3 |; sleep(0.1s);  assert(!x_was_two_for_two_seconds);

x = 2          |  assert(!x_was_two_for_two_seconds);
sleep(1.5s) |  assert(!x_was_two_for_two_seconds);
x = 3 |; x = 2 |; sleep (1s) |  assert(!x_was_two_for_two_seconds);
```

19.9.1.3 Scoping at at

`at` statements are not scoped. But, using a `Tag` (`??sec:std-Tag`) object, one can control them. In the following example, `Tag.scope` is used to label the `at` statement. When the function ends, the `at` is no longer active.

```
var x = 0 |
var x_is_two = false |;

{
  Tag.scope:
    at (x == 2)
      x_is_two = true
    onleave
      x_is_two = false;
  sleep(2s);
},
x = 2 |; assert(x_is_two);
x = 1 |; assert(!x_is_two);
sleep(3s);
x = 2 | assert(!x_is_two);
```

19.9.2 every

The `every` statement enables to execute a block of code repeatedly, with the given period.

```
// Print out a message every second.
timeout (2.1s)
  every (1s)
    echo("Are you still there?");
[00000000] *** Are you still there?
[00001000] *** Are you still there?
[00002000] *** Are you still there?
```

It exists in several flavors.

19.9.2.1 every|

The whole `every|` statement itself remains in foreground: statements attached after it with `;` or `|` will not be reached unless you `break` out of it. You may use `continue` to finish one iteration. In that case, the following iteration is not immediately started, it will be launched as expected, at the given period.

```

{
  var count = 4;
  var start = time;
  echo("before");
  every| (1s)
  {
    count -= 1;
    echo("begin: %s @ %1.0fs" % [count, time - start]);
    if (count == 2)
      continue;
    if (count == 0)
      break;
    echo("end:   " + count);
  };
  echo("after");
};
[00000597] *** before
[00000598] *** begin: 3 @ 0s
[00000599] *** end:   3
[00000698] *** begin: 2 @ 1s
[00000798] *** begin: 1 @ 2s
[00000799] *** end:   1
[00000898] *** begin: 0 @ 3s
[00000899] *** after

```

The `every|` flavor does not let iterations overlap. If an iteration takes too long, the following iterations are delayed. That is, the next iterations will start immediately after the end of the current one, and next iterations will occur normally from this point.

```

{
  var too_long = true|;

  var count = 5;
  // Every other iteration exceeds the period, and will delay the
  // following one.
  every| (1s)
  {
    if (! count --)
      break;

    if (too_long)
    {
      too_long = false;
      echo("Long in");
      sleep(1.5s);
      echo("Long out");
    }
    else
    {
      too_long = true;
      echo("Short");
    }
  };
};

```

```
};
[00000000] *** Long in
[00001500] *** Long out
[00001500] *** Short
[00002500] *** Long in
[00004000] *** Long out
[00004000] *** Short
```

The flow-control constructs `break` and `continue` are supported.

```
{
  var count = 0;
  every! (250ms)
  {
    count += 1;
    if (count == 2)
      continue;
    if (count == 4)
      break;
    echo(count);
  }
};

[00000000] *** 1
[00001500] *** 3
```

19.9.2.2 every,

The default flavor, `every`, launches the execution of the block in the background every given period. Iterations may overlap.

```
// If an iteration is longer than the given period, it will overlap
// with the next one.
timeout (2.8s)
  every (1s)
  {
    echo("In");
    sleep(1.5s);
    echo("Out");
  };

[00000000] *** In
[00001000] *** In
[00001500] *** Out
[00002000] *** In
[00002500] *** Out
```

19.9.3 for

The `for` loops come into several flavors, depending on the actual kind of `for` loop.

19.9.3.1 C-for,

This feature is experimental. It might be changed, or even removed. Feedback on its use would be appreciated.

`for`, is syntactic sugar for `while`,, see [Section 19.9.7.1](#).

```
for, (var i = 3; 0 < i; i -= 1)
{
    var j = i |
    echo ("in: i = %s, j = %s" % [i, j]);
    sleep(j/10);
    echo ("out: i = %s, j = %s" % [i, j]);
};
echo ("done");
[00000144] *** in: i = 3, j = 3
[00000145] *** in: i = 2, j = 2
[00000145] *** in: i = 1, j = 1
[00000246] *** out: i = 0, j = 1
[00000346] *** out: i = 0, j = 2
[00000445] *** out: i = 0, j = 3
[00000446] *** done
```

```
for, (var i = 9; 0 < i; i -= 1)
{
    var j = i;
    if (j % 2)
        continue
    else if (j == 4)
        break
    else
        echo ("%s: done" % j)
};
echo ("done");
[00000146] *** 8: done
[00000148] *** 6: done
[00000150] *** done
```

19.9.3.2 range-for& (:)

One can iterate concurrently over the members of a collection.

```
for& (var i: [0, 1, 2])
{
    echo (i * i);
    echo (i * i);
};
[00000000] *** 0
[00000000] *** 1
[00000000] *** 4
[00000000] *** 0
[00000000] *** 1
[00000000] *** 4
```

If an iteration executes `continue`, it is stopped; the other iterations are not affected.

```
for& (var i: [0, 1, 2])
{
  var j = i;
  if (j == 1)
    continue;
  echo (j);
};
[00020653] *** 0
[00021054] *** 2
```

If an iteration executes `break`, all the iterations including this one, are stopped.

```
for& (var i: [0, 1, 2])
{
  var j = i;
  echo (j);
  if (j == 1)
  { echo ("break");
    break;};
  sleep(1s);
  echo (j);
};
[00000001] *** 0
[00000001] *** 1
[00000001] *** 2
[00000002] *** break
```

19.9.3.3 for& (n)

Since `for& (n) body` is processed as `for& (var tmp: n) body`, which `tmp` a hidden variable, see [Section 19.9.3.2](#) for details.

19.9.4 loop,

This feature is experimental. It might be changed, or even removed. Feedback on its use would be appreciated.

This is syntactic sugar for `while, (true)`. In the following example, care must be taken that concurrent executions don't modify `n` simultaneously. This would happen had `;` been used instead of `|`.

```
{
  var n = 10|;
  var res = []|;
  loop,
  {
    n-- |
    res << n |
    if (n == 0)
```



```

        break
    };
    res.sort
}
==
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

```

19.9.5 waituntil

The `waituntil` construct is used to hold the execution until some condition is verified. Similarly to `at` (Section 19.9.1) and the other event-based constructs, `waituntil` may work on events, or on Boolean expressions.

19.9.5.1 waituntil on Events

When the execution flow enters a `waituntil`, the execution flow is held until the event is fired. Once caught, the event is consumed, another `waituntil` will require another event emission.

```

{
    var e = Event.new;
    {
        waituntil (e?);
        echo ("caught e");
    },
    e!;
[00021054] *** caught e
    e!;
};

```

In the case of lasting events (see `Event.trigger`), the condition remains verified as long as the event is “on”.

```

{
    var e = Event.new;
    e.trigger;
    {
        waituntil (e?);
        echo ("caught e");
    };
[00021054] *** caught e
    {
        waituntil (e?);
        echo ("caught e");
    };
[00021054] *** caught e
    {
        waituntil (e?);
        echo ("caught e");
    };
[00021054] *** caught e
};

```

The event specification may use pattern-matching to specify the accepted events.

```
{
  var e = Event.new;
  {
    waituntil (e?(1, var b));
    echo ("caught e(1, %s)" % b);
  },
  e!;
  e!(1);
  e!(2, 2);
  e!(1, 2);
[00021054] *** caught e(1, 2)
  e!(1, 2);
};
```

Events sent before do not release the construct.

```
{
  var e = Event.new;
  e!;
  {
    waituntil (e?);
    echo ("caught e");
  },
  e!;
[00021054] *** caught e
};
```

19.9.5.2 `waituntil` on Boolean Expressions

You may use any expression that evaluates to a truth value as argument to `waituntil`.

```
{
  var foo = Object.new;
  {
    waituntil (foo.hasLocalSlot("bar"));
    echo(foo.getLocalSlot("bar"));
  },
  var foo.bar = 123|;
};
[00021054] *** 123
```

19.9.6 `whenever`

The `whenever` construct really behaves like a never-ending `loop if` construct. It also works on events and Boolean expressions, and triggers each time the condition *becomes* verified.

```
whenever (condition)
  statement1
```

It supports an optional `else` clause, which is run whenever the condition changes “from true to false”.

```
whenever (condition)
  statement1
else
  statement2
```

The execution of a `whenever` clause is “instantaneous”, there is no mean to use ‘,’ to put it in background. It is also asynchronous with respect to the condition: the emission of an event is not held until all its watchers have completed their job.

19.9.6.1 `whenever` on Events

A `whenever` clause can be used to catch events with or without payloads.

```
var e = Event.new!;
whenever (e?)
  echo("e on")
else
  echo("e off");
[00000001] *** e off
[00000002] *** e off
[00000003] *** ...
e!;
[00000004] *** e on
[00000005] *** e off
[00000006] *** e off
[00000007] *** ...
e!(1) & e!(2);
[00000008] *** e on
[00000009] *** e on
[00000010] *** e off
[00000011] *** e off
[00000012] *** ...
```

The pattern-matching and guard on the payload is available.

```
var e = Event.new!;
whenever (e?("arg", var arg) if arg % 2)
  echo("e (%s) on" % arg)
else
  echo("e off");
e!("param", 23);
e!("arg", 52);
e!("arg", 23);
[00000001] *** e (23) on
[00000002] *** e off
[00000003] *** e off
[00000004] *** ...
e!("arg", 52);
e!("arg", 17);
[00000005] *** e (17) on
```

```
[00000006] *** e off
[00000007] *** e off
[00000008] *** ...
```

If the body of the `whenever` lasts for a long time, it is possible that two executions be run concurrently.

```
var e = Event.new|;
whenever (e?(var d))
{
  echo("e (%s) on begin" % d);
  sleep(d);
  echo("e (%s) on end" % d);
};

e!(0.3s) & e!(1s);
sleep(3s);
[00000202] *** e (1) on begin
[00000202] *** e (0.3) on begin
[00000508] *** e (0.3) on end
[00001208] *** e (1) on end
```

19.9.6.2 `whenever` on Boolean Expressions

A `whenever` construct will repeatedly evaluate its body as long as its condition holds. The number of evaluation of the bodies is typically non-deterministic, as not only does it depend on how long the condition holds, but also “how fast” the Urbi kernel runs.

```
var x = 0|;
var count = 0|;
var t = Tag.new|;
t:
  whenever (x % 2)
  {
    if (!count)
      echo("x is now odd (%s)" % x);
    count++;
  }
  else
  {
    if (!count)
      echo("x is now even (%s)" % x);
    count++;
  };

t:
  whenever (100 < count)
  {
    count = 0 |
    x++;
  };
waituntil(x == 4);
```

```
[00000769] *** x is now even (0)
[00000809] *** x is now odd (1)
[00000846] *** x is now even (2)
[00000886] *** x is now odd (3)
[00000924] *** x is now even (4)
t.stop;
```

19.9.7 While

19.9.7.1 while,

This feature is experimental. It might be changed, or even removed. Feedback on its use would be appreciated.

This construct provides a means to run concurrently multiple instances of statements. The semantics of

```
while, (condition)
    body;
```

is the same as

```
condition | body , condition | body , ...
```

Attention must be paid to the fact that the (concurrent) iterations share a common access to the environment, therefore if, for instance, you want to keep the value of some index variable, use a local variable inside the loop body:

```
{
    var i = 4 |
    while, (i)
    {
        var j = i -= 1;
        echo ("in: i = %s, j = %s" % [i, j]);
        sleep(j/10);
        echo ("out: i = %s, j = %s" % [i, j]);
    } |
    echo ("done");
} |
[00000144] *** in: i = 2, j = 3
[00000145] *** in: i = 1, j = 2
[00000145] *** in: i = 0, j = 1
[00000146] *** in: i = 0, j = 0
[00000146] *** out: i = 0, j = 0
[00000246] *** out: i = 0, j = 1
[00000346] *** out: i = 0, j = 2
[00000445] *** out: i = 0, j = 3
[00000446] *** done
```

As for the other flavors, `continue` skips the current iteration, and `break` ends the loop. Note that `break` stops all the running iterations. This semantics is likely to be changed to “`break`

ends the current iteration and stops the generation of others, but lets the other concurrent iterations finish”, so do not rely on this feature.

Control flow is passed to the following statement when all the iterations are done.

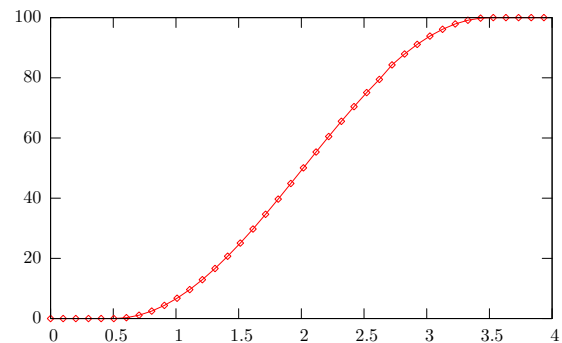
```
{
  var i = 10|
  while, (i)
  {
    var j = i -= 1;
    if (j % 2)
      continue
    else if (j == 4)
      break
    else
      echo("%s: done" % j)
  }|
  echo("done");
};
[00000146] *** 8: done
[00000148] *** 6: done
[00000150] *** done
```

19.10 Trajectories

In robotics, *trajectories* are often used: they are a means to change the value of a variable (actually, a slot) over time. This can be done using detached executions, for instance using a combination of [every](#) and [detach](#), but urbiscript provides syntactic sugar to this end.

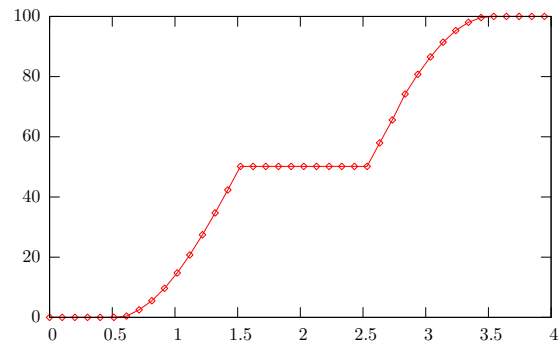
For instance the following drawing shows how the `y` variable is moved smoothly from its *initial value* (0) to its *target value* (100) in 3 seconds (the value given to the `smooth` attribute).

```
var y = 0;
{
  sleep(0.5s);
  y = 100 smooth:3s,
},
```



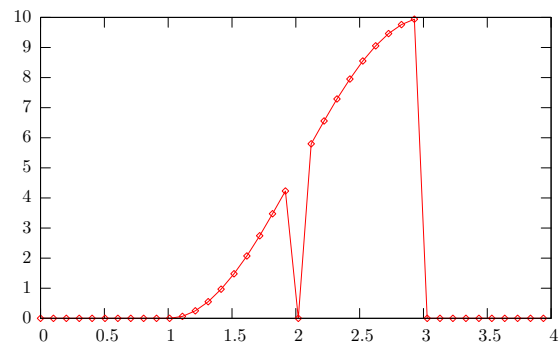
Trajectories can be frozen and unfrozen, using tags ([Section 11.3](#)). In that case, “time is suspended”, and the trajectory resumes as if the trajectory was never interrupted.

```
var y = 0;
{
  sleep(0.5s);
  assign: y = 100 smooth:2s,
  sleep(1s);
  assign.freeze;
  sleep(1s);
  assign.unfreeze;
},
```



When the target value is reached, the trajectory generator is detached from the variables: changes to the value of the variable no longer trigger the trajectory generator.

```
var y = 0;
{
  sleep(1s);
  assign: y = 10 smooth:2s,
  sleep(1s);
  y = 0;
  sleep(1s);
  y = 0;
},
```



See the specifications of [TrajectoryGenerator](#) ([??sec:std-TrajectoryGenerator](#)) for the list of supported trajectories.

19.11 Garbage collection and limitations

urbiscript provides automatic garbage collection. That is, you can create new objects and don't have to worry about reclaiming the memory when you're done with them. We use a reference counting algorithm for garbage collection: every object has a counter indicating how many references to it exist. When that counter drops to zero, nobody has a reference to the object anymore, and it is thus deleted.

```
{
  var x = List.new; // A new list is allocated
  x << 42;
};
// The list will be automatically freed, since there are no references to it left.
```

This is not part of the language interface, and we might change the garbage collecting system in the future. Therefore, do not rely on the current garbage collecting behavior, and especially not on the determinism of the destruction time of objects.

However, this implementation has a limitation you should be aware of: cycle of object references won't be properly reclaimed. Indeed if object A has a reference to B, and B has a reference to A, none of them will ever be reclaimed since they both have a reference pointing to them. As a consequence, avoid creating cycles in object references, or if you really have to, break the cycle manually before releasing your last reference to the object.

```
// Create a reference cycle
var A = Object.new;
var A.B = Object.new; // A refers to B
var A.B.A = A; // B refers back to A

removeSlot("A"); // delete our last reference to A
// Although we have no reference left to A or B,
// they won't be deleted since they refer to each other.
```

If you really need the cycle, this is how you could break it manually:

```
A.B.removeSlot("A"); // Break the cycle
removeSlot("A"); // Delete our last reference to A
// A will be deleted since it's not referred from anywhere.
// Since A held the last reference to B, B will be deleted too.
```


Chapter 20

urbiscript Standard Library

20.1 Barrier

Barrier is used to wait until another job raises a signal. This can be used to implement blocking calls waiting until a resource is available.

20.1.1 Prototypes

- `Object` (`??sec:std-Object`)

20.1.2 Construction

A **Barrier** can be created with no argument. Signals and wait calls done on this instance are restricted to this instance.

```
Barrier.new;  
[00000000] Barrier_0x25d2280
```

20.1.3 Slots

- `signal(payload)`
Wake up one of the job waiting for a signal. The *payload* is sent to the *wait* method. This method returns the number of job woken up.

```
do (Barrier.new)  
{  
  echo(wait) &  
  echo(wait) &  
  assert  
  {  
    signal(1) == 1;  
    signal(2) == 1;  
  }  
};  
[00000000] *** 1
```

```
[00000000] *** 2
```

- `signalAll(payload)`

Wake up all the jobs waiting for a signal. The *payload* is sent to all *wait* methods. Return the number of jobs woken up.

```
do (Barrier.new)
{
  echo(wait) &
  echo(wait) &
  assert
  {
    signalAll(1) == 2;
    signalAll(2) == 0;
  }
}!;
[00000000] *** 1
[00000000] *** 1
```

- `wait`

Block until a signal is received. The *payload* sent with the signal function is returned by the `wait` method.

```
do (Barrier.new)
{
  echo(wait) &
  signal(1)
}!;
[00000000] *** 1
```

20.2 Binary

A Binary object, sometimes called a *blob*, is raw memory, decorated with a user defined header.

20.2.1 Prototypes

- `Object (??sec:std-Object)`

20.2.2 Construction

Binaries are usually not made by users, but they are heavily used by the internal machinery when exchanging Binary UValues. A binary features some `content` and some `keywords`, both simple `Strings (??sec:std-String)`.

```
Binary.new("my header", "my content");
[00000001] BIN 10 my header
my content
```

Beware that the third line above (`my content`), was output by the system, although not preceded by a timestamp.

20.2.3 Slots

- `'+' (that)`

Return a new Binary whose keywords are those of `this` if not empty, otherwise those of `that`, and whose data is the concatenation of both.

```
Binary.new("0", "0") + Binary.new("1", "1")
  == Binary.new("0", "01");
Binary.new("", "0") + Binary.new("1", "1")
  == Binary.new("1", "01");
```

- `'==' (other)`

Whether keywords and data are equal.

```
Binary.new("0", "0") == Binary.new("0", "0");
Binary.new("0", "0") != Binary.new("0", "1");
Binary.new("0", "0") != Binary.new("1", "0");
```

- `asString`

Display using the syntactic rules of the UObject/UValue protocol. Incoming binaries must use a semicolon to separate the header part from the content, while outgoing binaries use a carriage-return.

```
assert(Binary.new("head", "content").asString
  == "BIN 7 head\ncontent");
var b = BIN 7 header;content;
[00000002] BIN 7 header
content
assert(b == Binary.new("header", "content"));
```

This syntax (`BIN size header; content`) is *partially* supported in urbiscript, but it is strongly discouraged. Rather, use the `\B(size)(data)` special escape (see [Section 19.1.6.6](#)):

```
Binary.new("head", "\B(7)(content)").asString
  == "BIN 7 head\ncontent";
```

- `data`

The data carried by the Binary.

```
Binary.new("head", "content").data == "content";
```

- `empty`

Whether the data is empty.

```
Binary.new("head", "").empty;
!Binary.new("head", "content").empty;
```

- **keywords**

The headers carried by the Binary.

```
Binary.new("head", "content").keywords == "head";
```

20.3 Boolean

There is no object `Boolean` in urbiscript, but two specific objects `true` and `false`. They are the result of all the comparison statements.

20.3.1 Prototypes

The objects `true` and `false` have the following prototype.

- `Singleton` (`??sec:std-Singleton`)

20.3.2 Construction

There are no constructors, use `true` and `false`. Since they are singletons, `clone` will return themselves, not new copies.

```
true;
!false;
2 < 6 === true;
true.new === true;
6 < 2 === false;
```

20.3.3 Truth Values

As in many programming languages, conditions may be more than only `true` and `false`. Whether some value is considered as true depends on the type of `this`. Actually, by default objects as considered “true”, objects evaluating to “false” are the exception:

- `false`, `nil`
`false`.

`void` raise an error.

`Float false` iff null (`Float` (`??sec:std-Float`)).

- `Dictionary`, `List`, `String`
`false` iff empty (`Dictionary` (`??sec:std-Dictionary`), `List` (`??sec:std-List`), `String` (`??sec:std-String`)).
- otherwise
`true`.

The method `Object.asBool` is in charge of converting some arbitrary value into a Boolean.

```
assert(Global.asBool == true);
assert(nil.asBool == false);
void.asBool;
[00000421:error] !!! unexpected void
```

20.3.4 Slots

- `'&&'` (*that*)

Short-circuiting logical and. If `this` is true evaluate and return *that*. If `this` is false, return itself without evaluating *that*.

```
(true && 2) == 2;
(false && 1 / 0) == false;
```

- `'||'` (*that*)

Short-circuiting logical or. If `this` is false evaluate and return *that*. If `this` is true, return itself without evaluating *that*.

```
(true || 1/0) == true;
(false || 2) == 2;
```

- `'!'`

Logical negation. If `this` is false return true and vice-versa.

```
!true == false;
!false == true;
```

- `asBool`

Identity.

```
true.asBool == true;
false.asBool == false;
```

20.4 CallMessage

Capturing a method invocation: its target and arguments.

20.4.1 Examples

20.4.1.1 Evaluating an argument several times

The following example implements a lazy function which takes an integer *n*, then arguments. The *n*-th argument is evaluated twice using `evalArgAt`.

```
function callTwice
{
  var n = call.evalArgAt(0);
  call.evalArgAt(n);
  call.evalArgAt(n)
} |;

// Call twice echo("foo").
callTwice(1, echo("foo"), echo("bar"));
[00000001] *** foo
[00000002] *** foo

// Call twice echo("bar").
callTwice(2, echo("foo"), echo("bar"));
[00000003] *** bar
[00000004] *** bar
```

20.4.1.2 Strict Functions

Strict functions do support `call`.

```
function strict(x)
{
  echo("Entering");
  echo("Strict: " + x);
  echo("Lazy:   " + call.evalArgAt(0));
} |;

strict({echo("1"); 1});
[00000011] *** 1
[00000013] *** Entering
[00000012] *** Strict: 1
[00000013] *** 1
[00000014] *** Lazy:   1
```

20.4.2 Slots

- **args**

The list of unevaluated arguments.

```
function args { call.args }|
assert
{
  args == [];
  args() == [];
  args({echo(111); 1}) == [Lazy.new(closure() {echo(111); 1})];
  args(1, 2) == [Lazy.new(closure () {1}),
                 Lazy.new(closure () {2})];
};
```

- **argsCount**

The number of arguments. Do not evaluate them.

```
function argsCount { call.argsCount }|;
assert
{
  argsCount == 0;
  argsCount() == 0;
  argsCount({echo(1); 1}) == 1;
  argsCount({echo(1); 1}, {echo(2); 2}) == 2;
};
```

- **code**

The body of the called function as a [Code \(??sec:std-Code\)](#).

```
function code { call.getSlot("code") }|
assert (code == getSlot("code"));
```

- **evalArgAt(*n*)**

Evaluate the *n*-th argument, and return its value. *n* must evaluate to an non-negative integer. Repeated invocations repeat the evaluation, see [Section 20.4.1.1](#).

```
function sumTwice
{
  var n = call.evalArgAt(0);
  call.evalArgAt(n) + call.evalArgAt(n)
}|;

function one () { echo("one"); 1 }|;

sumTwice(1, one, one + one);
[00000008] *** one
[00000009] *** one
[00000010] 2
sumTwice(2, one, one + one);
[00000011] *** one
[00000012] *** one
[00000011] *** one
[00000012] *** one
[00000013] 4

sumTwice(3, one, one);
[00000014:error] !!! evalArgAt: invalid index: 3
sumTwice(3.14, one, one);
[00000015:error] !!! evalArgAt: invalid index: 3.14
```

- **evalArgs**

Call [evalArgAt](#) for each argument, return the list of values.

```
function twice
{
```

```

    call.evalArgs + call.evalArgs
  }|;
twice({echo(1); 1}, {echo(2); 2});
[00000011] *** 1
[00000012] *** 2
[00000011] *** 1
[00000012] *** 2
[00000013] [1, 2, 1, 2]

```

- **message**

The name under which the function was called.

```

function myself { call.message }|
assert(myself == "myself");

```

- **sender**

The object *from which* the invocation was made (the *caller* in other languages). Not to be confused with **target**.

```

function Global.getSender { call.sender } |
function Global.callGetSender { getSender } |

assert
{
  // Call from the current Lobby, with the Lobby as target.
  getSender === lobby;
  // Call from the current Lobby, with Global as the target.
  Global.getSender === lobby;
  // Ask Lobby to call getSender.
  callGetSender === lobby;
  // Ask Global to call getSender.
  Global.callGetSender === Global;
};

```

- **target**

The object *on which* the invocation is made. In other words, the object that will be bound to **this** during the evaluation. Not to be confused with **sender**.

```

function Global.getTarget { call.target } |
function Global.callGetTarget { getTarget } |

assert
{
  // Call from the current Lobby, with the Lobby as target.
  getTarget === lobby;
  // Call from the current Lobby, with Global as the target.
  Global.getTarget === Global;
  // Ask Lobby to call getTarget.
  callGetTarget === lobby;
  // Ask Global to call getTarget.
  Global.callGetTarget === Global;
};

```



```
};
```

20.5 Channel

Returning data, typically asynchronous, with a label so that the “caller” can find it in the flow.

20.5.1 Prototypes

- `Object` (`??sec:std-Object`)

20.5.2 Construction

Channels are created like any other object. The constructor must be called with a string which will be the label.

```
var ch1 = Channel.new("my_label");
[00000201] Channel_0x7985810

ch1 << 1;
[00000201:my_label] 1

var ch2 = ch1;
[00000201] Channel_0x7985810

ch2 << 1/2;
[00000201:my_label] 0.5
```

20.5.3 Slots

- `'<<'` (*value*)
Send *value* to `this` tagged by its label if non-empty.

```
Channel.new("label") << 42;
[00000000:label] 42

Channel.new("") << 51;
[00000000] 51
```

- `echo(value)`
Same as `System.echo(value, name)`.

```
Channel.new("label").echo(42);
[00000000:label] *** 42

Channel.new("").echo("Foo");
[00000000] *** Foo
```

- **enabled**

Whether the Channel is enabled. Disabled Channels produce no output.

```
var c = Channel.new("");

c << "enabled";
[00000000] "enabled"

c.enabled = false|;
c << "disabled";

c.enabled = true|;
c << "enabled";
[00000000] "enabled"
```

- **Filter**

Filtering channel.

The Filter channel outputs text that can be parsed without error by the liburbi. It does this by filtering types not handled by the liburbi, and displaying them using [echo](#).

```
// Use a filtering channel on our lobby output.
topLevel = Channel.Filter.new("");
// liburbi knows about List, Dictionary, String and Float, so standard display.
[1, "foo", ["test" => 5]];
[00000001] [1, "foo", ["test" => 5]]
// liburbi does not know 'lobby', so it is escaped with echo:
lobby;
[00000002] *** Lobby_0xADDR
// The following list contains a function which is not handled by liburbi, so
// it gets escaped too.
[1, function () {}];
[00000003] *** [1, function () {}]
// Restore default display to see the difference.
topLevel = Channel.topLevel|;
// The echo is now gone.
[1, function () {}];
[00001758] [1, function () {}]
```

- **quote**

Whether the strings are output escaped (the default) instead of raw strings.

```
var d = Channel.new("");

assert(d.enabled);
d << "A \"String\"";
[00000000] "A \"String\""

d.quote = false|;
d << "A \"String\"";
[00000000] A "String"
```

- **name**

The name of the Channel, used to label the output.

```
assert
{
  Channel.new("").name == "";
  Channel.new("foo").name == "foo";
};
```

- **null**

A predefined stream whose **enabled** is **false**.

```
Channel.null << "Message";
```

- **topLevel**

A predefined stream for regular output. Strings are output escaped.

```
Channel.topLevel << "Message";
[00015895] "Message"
Channel.topLevel << "\"quote\"";
[00015895] "\"quote\""
```

- **warning**

A predefined stream for warning messages. Strings sent to it are not escaped.

```
Channel.warning << "Message";
[00015895:warning] Message
Channel.warning << "\"quote\"";
[00015895:warning] "quote"
```

20.6 Code

Functions written in urbiscript.

20.6.1 Prototypes

- [Comparable](#) (??sec:std-Comparable)
- [Executable](#) (??sec:std-Executable)

20.6.2 Construction

The keywords **function** and **closure** build Code instances.

```
function(){}.protos[0] === getSlot("Code");
closure(){}.protos[0] === getSlot("Code");
```

20.6.3 Slots

- Whether `this` and `that` are the same source code (actually checks that both have the same `asString`), and same closed values.

Closures and functions are different, even if the body is the same.

```
function () { 1 } == function () { 1 };
function () { 1 } != closure () { 1 };
closure () { 1 } != function () { 1 };
closure () { 1 } == closure () { 1 };
```

No form of equivalence is applied on the body, it must be the same.

```
function () { 1 + 1 } == function () { 1 + 1 };
function () { 1 + 2 } != function () { 2 + 1 };
```

Arguments do matter, even if in practice the functions are the same.

```
function (var ignored) {} != function () {};
function (var x) { x } != function (y) { y };
```

A lazy function cannot be equal to a strict one.

```
function () { 1 } != function { 1 };
```

If the functions capture different variables, they are different.

```
{
  var x;
  function Global.capture_x() { x };
  function Global.capture_x_again () { x };
  {
    var x;
    function Global.capture_another_x() { x };
  }
}|;
assert
{
  getSlot("capture_x") == getSlot("capture_x_again");
  getSlot("capture_x") != getSlot("capture_another_x");
};
```

If the functions capture different targets, they are different.

```
class Foo
{
  function makeFunction() { function () {} };
  function makeClosure() { closure () {} };
}|;

class Bar
{
  function makeFunction() { function () {} };
```

```
function makeClosure() { closure () {} };
}|;

assert
{
  Foo.makeFunction() == Bar.makeFunction();
  Foo.makeClosure() != Bar.makeClosure();
};
```

- `apply(args)`

Invoke the routine, with all the arguments. The target, `this`, will be set to `args[0]` and the remaining arguments will be given as arguments.

```
function (x, y) { x+y }.apply([nil, 10, 20]) == 30;
function () { this }.apply([123]) == 123;

// There is Object.apply.
1.apply([this]) == 1;
```

```
function () {}.apply([]);
[00000001:error] !!! apply: argument list must begin with 'this'

function () {}.apply([1, 2]);
[00000002:error] !!! apply: expected 0 argument, given 1
```

- `asString`

Conversion to `String (??sec:std-String)`.

```
closure () { 1 }.asString == "closure () {\n 1\n}";
function () { 1 }.asString == "function () {\n 1\n}";
```

- `bodyString`

Conversion to `String (??sec:std-String)` of the routine body.

```
closure () { 1 }.bodyString == "1";
function () { 1 }.bodyString == "1";
```

20.7 Comparable

Objects that can be compared for equality and inequality. See also `Orderable (??sec:std-Orderable)`.

This object, made to serve as prototype, provides a definition of `!=` based on `==`. `Object` provides a default implementation of `==` that bounces on the physical equality `===`.

```
class Foo : Comparable
{
  var value = 0;
  function init (v) { value = v; };
```

```

function '==' (lhs) { value == lhs.value; };
};
[00000000] Foo
Foo.new(1) == Foo.new(1);
[00000000] true
Foo.new(1) == Foo.new(2);
[00000000] false

```

20.7.1 Slots

- Whether ! (`this` != `that`).

```

class FortyTwo : Comparable
{
  function '!=' (that) { 42 != that };
}|;
assert
{
  FortyTwo != 51;
  FortyTwo == 42;
};

```

- !=(`that`)
Whether ! (`this` == `that`).

```

class FiftyOne : Comparable
{
  function '==' (that) { 51 == that };
}|;
assert
{
  FiftyOne == 51;
  FiftyOne != 42;
};

```

20.8 Container

This object is meant to be used as a prototype for objects that support `has` and `hasNot` methods. Any class using this prototype must redefine either `has`, `hasNot` or both.

20.8.1 Prototypes

- `Object` (`??sec:std-Object`)

20.8.2 Slots

- `has(e)`
!`hasNot(e)`. The indented semantics is “true when the container has a key (or item) matching `e`”. This is what `e in c` is mapped onto.

```

class NotCell : Container
{
  var val;
  function init(var v) { val = v };
  function hasNot(var v) { val != v };
};
var c = NotCell.new(23)|;
assert
{
  c.has(23);      23 in c;
  c.hasNot(3);    3 not in c;
};

```

- `hasNot(e)`
`!has(e)`. The indented semantics is “true when the container does not have a key (or item) matching *e*”.

```

class Cell : Container
{
  var val;
  function init(var v) { val = v };
  function has(var v) { val == v };
};
var d = Cell.new(23)|;
assert
{
  d.has(23);      23 in d;
  d.hasNot(3);    3 not in d;
};

```

20.9 Control

Control is designed as a namespace for control sequences. Some of these entities are used by the Urbi engine to execute some urbiscript features; in other words, users are not expected to use it, much less change it.

20.9.1 Prototypes

- `Object` (`??sec:std-Object`)

20.9.2 Slots

- `detach(exp)`

Detach the evaluation of the expression *exp* from the current evaluation. The *exp* is evaluated in parallel to the current code and keep the current tag which are attached to it. Return the spawned `Job` (`??sec:std-Job`). Same as calling `System.spawn`: `System.spawn(closure () { exp`

```

{
  var jobs = [];
  var res = [];
  for (var i : [0, 1, 2])
  {
    jobs << detach({ res << i; res << i }) |
    if (i == 2)
      break
  };
  assert (res == [0, 1, 0]);
  jobs
};
[00009120] [Job<shell_11>, Job<shell_12>, Job<shell_13>]

```

- **disown(*exp*)**

Same as **detach** except that tags used to tag the **disown** call are not inherited inside the expression. Return the spawned **Job** (**??sec:std-Job**). Same as calling **System.spawn: System.spawn(closure () { *exp* }, true**.

```

{
  var jobs = [];
  var res = [];
  for (var i : [0, 1, 2])
  {
    jobs << disown({ res << i; res << i }) |
    if (i == 2)
      break
  };
  jobs.each (function (var j) { j.waitForTermination });
  assert (res == [0, 1, 0, 2, 1, 2]);
  jobs
};
[00009120] [Job<shell_14>, Job<shell_15>, Job<shell_16>]

```

- **finally(*action*, *reaction*)**

Execute the *reaction* function at the end of the evaluation of the *action* function. This is equivalent to

```

{
  var t = Tag.new |
  at (t.leave?)
    reaction() |
  t: action()
}

```

- **persist(*expression*, *delay*)**

Return an object whose *val* slot evaluates to true if the *expression* has been continuously true for this *delay* and false otherwise.

This function is used to implement


```
at (condition ~ delay)
  action
```

as

```
var u = persist (condition, delay);
at (u.val)
  action
```

The `persist` action will be controlled by the same tags as the initial `at` block.

20.10 Date

This class is meant to record dates in time.

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

20.10.1 Prototypes

- `Orderable` (`??sec:std-Orderable`)
- `Comparable` (`??sec:std-Comparable`)

20.10.2 Construction

Without argument, newly constructed dates refer to the *epoch* (see [epoch](#)).

```
Date.new;
[00000001] 1970-01-01 01:00:00
```

With a numeric argument *s*, refers to the date that is *s* seconds after the epoch.

```
Date.new(1234567890);
[00023593] 2009-02-14 00:31:30
```

With a string argument *d*, refers to the date contained in *d*. The string should be formatted as ‘*yyyy-mm-dd hh:mm:ss*’ (see [asString](#)). *mm* and *ss* are optional. If the block ‘*hh:mm:ss*’ is absent, the behavior is undefined.

```
Date.new("2003-10-10 20:10:50");
[00086457] 2003-10-10 20:10:50
Date.new("2003-10-10 20:10");
[00091439] 2003-10-10 20:10:00
Date.new("2003-10-10 20");
[00094386] 2003-10-10 20:00:00
```

If you create a date with a numeric argument which cannot be interpreted as a date you should get an error message warning you that the value cannot be converted into a integer

```
Date.new(0.1);
[00095042:error] !!! new: bad numeric conversion: overflow or non empty fractional part: 0.1
```

20.10.3 Slots

- `'+' (that)`

Return the date which corresponds to waiting `Duration (??sec:std-Duration)` *that* after *this*.

```
Date.new(1000) + Duration.new(100) == Date.new(1100);
Date.new(1000) + Duration.new(-100) == Date.new(900);
```

- `'-' (that)`

Compute the difference between two dates. Return a `Duration (??sec:std-Duration)`.

```
Date.new(100) - Date.new(0) == Duration.new(100);
Date.new(0) - Date.new(100) == Duration.new(-100);
```

- `'==' (that)`

Equality test.

```
Date.new(11223344) == Date.new(11223344);
Date.new(11111111) != Date.new(22222222);
```

- `'<' (that)`

Order comparison.

```
Date.new(11111111) < Date.new(22222222);
```

- `asFloat`

Give the numeric value of the date that refers to the date that is *s* seconds after the epoch.

```
Date.new("2002-01-20 23:59:59").asFloat == 1011567599;
```

- `asString`

Present as `'yyyy-mm-dd hh:mm:ss'` where *yyyy* is the four-digit year, *mm* the two-digit month (from 1 to 12), *dd* the two-digit day in the month (from 1 to 31), *hh* the two-digit hour (from 0 to 23), *mn* the two-digit number of minutes in the hour (from 0 to 59), and *ss* the two-digit number of seconds in the minute (from 0 to 59).

```
Date.new(1234567890).asString == "2009-02-14 00:31:30";
```

- `epoch`

A fixed value, the “origin of times”: January 1st 1970, at midnight.

```
Date.epoch == Date;

Date.epoch == Date.new(0);
Date.epoch == Date.new();
```

- `now`
The current date.

```
Date.now;
[00000000] 2012-03-02 15:31:42
```

- `timestamp`
Synonym for `asFloat`

20.11 Dictionary

A *dictionary* is an *associative array*, also known as a *hash* in some programming languages. They are arrays whose indexes are strings.

In a way objects are dictionaries: one can use `setSlot`, `updateSlot`, and `getSlot`. This is unsafe since slots also contains value and methods that object depend upon to run properly.

20.11.1 Example

The following session demonstrates the features of the Dictionary objects.

```
var d = ["one" => 1, "two" => 2];
[00000001] ["one" => 1, "two" => 2]
for (var p : d)
  echo (p.first + " => " + p.second);
[00000003] *** one => 1
[00000002] *** two => 2
"three" in d;
[00000004] false
d["three"];
[00000005:error] !!! missing key: three
d["three"] = d["one"] + d["two"] | {};
"three" in d;
[00000006] true
d.getWithDefault("four", 4);
[00000007] 4
```

20.11.2 Prototypes

- `Comparable` (`??sec:std-Comparable`)
- `Container` (`??sec:std-Container`)
- `Object` (`??sec:std-Object`)
- `RangeIterable` (`??sec:std-RangeIterable`)

20.11.3 Construction

The Dictionary constructor takes arguments by pair (key, value).

```
Dictionary.new("one", 1, "two", 2);
[00000000] ["one" => 1, "two" => 2]
Dictionary.new;
[00000000] [ => ]
```

Yet you are encouraged to use the specific syntax for Dictionary literals:

```
["one" => 1, "two" => 2];
[00000000] ["one" => 1, "two" => 2]
[=>];
[00000000] [ => ]
```

An extra comma can be added at the end of the list.

```
[
  "one" => 1,
  "two" => 2,
];
[00000000] ["one" => 1, "two" => 2]
```

20.11.4 Slots

- '==' (*that*)

Whether *this* equals *that*. This suppose that elements contained inside the dictionary are Comparable.

```
[ => ] == [ => ];
["a" => 1, "b" => 2] == ["b" => 2, "a" => 1];
```

- '[]' (*key*)

Syntactic sugar for `get(key)`.

```
["one" => 1]["one"] == 1;
```

- '[]=' (*key*, *value*)

Syntactic sugar for `set(key, value)`, but returns *value*.

```
{
  var d = ["one" => "2"];
  assert
  {
    (d["one"] = 1) == 1;
    d["one"] == 1;
  };
};
```

- `asBool`

Negation of `empty`.

```
[=>].asBool == false;
["key" => "value"].asBool == true;
```

- `asList`

Return the contents of the dictionary as a `Pair` ([??sec:std-Pair](#)) list (*key*, *value*).

```
["one" => 1, "two" => 2].asList == [("one", 1), ("two", 2)];
```

Since `Dictionary` derives from `RangeIterable` ([??sec:std-RangeIterable](#)), it is easy to iterate over a `Dictionary` using a `range-for` ([Section 19.6.5.2](#)). No particular order is ensured.

```
{
  var res = [];
  for| (var entry: ["one" => 1, "two" => 2])
    res << entry.second;
  assert(res == [1, 2]);
};
```

- `asString`

A string representing the dictionary. There is no guarantee on the order of the output.

```
[=>].asString == "[ => ]";
["a" => 1, "b" => 2].asString == "[\"a\" => 1, \"b\" => 2]";
```

- `clear`

Empty the dictionary.

```
["one" => 1].clear.empty;
```

- `empty`

Whether the dictionary is empty.

```
[=>].empty == true;
["key" => "value"].empty == false;
```

- `erase(key)`

Remove the mapping for *key*.

```
["one" => 1, "two" => 2].erase("two") == ["one" => 1]
```

- `get(key)`

Return the value associated to *key*. A `Dictionary.KeyError` exception is thrown if the key is missing.

```

assert(["one" => 1, "two" => 2].get("one") == 1);
try
{
  ["one" => 1, "two" => 2].get("three");
  echo("never reached");
}
catch (var e if e.isA(Dictionary.KeyError))
{
  assert(e.key == "three")
};

```

- `getWithDefault(key, defaultValue)`

The value associated to *key* if it exists, *defaultValue* otherwise.

```

do (["one" => 1, "two" => 2])
{
  assert
  {
    getWithDefault("one", -1) == 1;
    getWithDefault("three", 3) == 3;
  };
}!;

```

- `has(key)`

Whether the dictionary has a mapping for *key*.

```

do (["one" => 1])
{
  assert(has("one"));
  assert(!has("zero"));
}!;

```

The infix operators `in` and `not in` use `has` (see [Section 19.1.8.6](#)).

```

"one" in ["one" => 1];
"two" not in ["one" => 1];

```

- `init(key1, value1, ...)`

Insert the mapping from *key1* to *value1* and so forth.

```

Dictionary.clone.init("one", 1, "two", 2);
[00000000] ["one" => 1, "two" => 2]

```

- `keys`

The list of all the keys. No particular order is ensured. Since [List \(??sec:std-List\)](#) features the same function, uniform iteration over a List or a Dictionary is possible.

```
{
  var d = ["one" => 1, "two" => 2];
  assert(d.keys == ["one", "two"]);
  assert({
    var res = [];
    for (var k: d.keys)
      res << d[k];
    res
  }
  == [1, 2]);
};
```

- `matchAgainst(handler, pattern)`

Pattern matching on members. See [Pattern \(??sec:std-Pattern\)](#).

```
{
  // Match a subset of the dictionary.
  ["a" => var a] = ["a" => 1, "b" => 2];
  // get the matched value.
  assert(a == 1);
};
```

- `set(key, value)`

Map *key* to *value* and return `this` so that invocations to `set` can be chained. The possibly existing previous mapping is overridden.

```
[=>].set("one", 2).set("one", 1);
[00000000] ["one" => 1]
```

- `size`

Number of element in the dictionary.

```
{
  var d = [=>];
  assert(d.size == 0);
  d["a"] = 0;
  assert(d.size == 1);
  d["b"] = 1;
  assert(d.size == 2);
  d["a"] = 2;
  assert(d.size == 2);
};
```

20.12 Directory

A *Directory* represents a directory of the file system.

20.12.1 Prototypes

- `Object` (`??sec:std-Object`)

20.12.2 Construction

A *Directory* can be constructed with one argument: the path of the directory using a `String` (`??sec:std-String`) or a `Path` (`??sec:std-Path`). It can also be constructed by the method `open` of `Path` (`??sec:std-Path`).

```
Directory.new(".");
[00000001] Directory(".")
Directory.new(Path.new("."));
[00000002] Directory(".")
```

20.12.3 Slots

- `asList`
The contents of the directory as a `Path` (`??sec:std-Path`) list. The various paths include the name of the directory `this`.
- `asString`
A `String` (`??sec:std-String`) containing the path of the directory.

```
Directory.new(".").asString == ".";
```

- `content`
The contents of the directory as a `String` (`??sec:std-String`) list. The strings include only the last component name; they do not contain the directory name of `this`.
- `fileCreated(name)`
Event launched when a file is created inside the directory. May not exist if not supported by your architecture.

```
if (Path.new("./dummy.txt").exists)
  File.new("./dummy.txt").remove;

{
  var d = Directory.new(".");
  waituntil(d.fileCreated?(var name));
  assert
  {
    name == "dummy.txt";
    Path.new(d.asString + "/" + name).exists;
  };
}
&
{
  sleep(100ms);
  File.create("./dummy.txt");
}
```



```
}|;
```

- `fileDeleted(name)`

Event launched when a file is deleted from the directory. May not exist if not supported by your architecture.

```
if (!Path.new("./dummy.txt").exists)
  File.create("./dummy.txt");

{
  var d = Directory.new(".");
  waituntil(d.fileDeleted?(var name));
  assert
  {
    name == "dummy.txt";
    !Path.new(d.asString + "/" + name).exists;
  };
}
&
{
  sleep(100ms);
  File.new("./dummy.txt").remove;
}|;
```

20.13 Duration

This class records differences between [Dates \(??sec:std-Date\)](#).

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

20.13.1 Prototypes

- [Float \(??sec:std-Float\)](#)

20.13.2 Construction

Without argument, a null duration.

```
Duration.new;
[00000001] Duration(0s)
Duration.new(1h);
[00023593] Duration(3600s)
```

Durations can be negative.

```
Duration.new(-1);
[00000001] Duration(-1s)
```

20.13.3 Slots

- `asFloat`
Return the duration as a `Float` (`??sec:std-Float`).

```
Duration.new(1000).asFloat == 1000;
```

- `asString`
Return the duration as a `String` (`??sec:std-String`).

```
Duration.new(1000).asString == "1000s";
```

- `seconds`
Return the duration as a `Float` (`??sec:std-Float`).

```
Duration.new(1000).seconds == 1000;
```

20.14 Event

An *event* can be “emitted” and “caught”, or “sent” and “received”. See also [Section 12.2](#).

20.14.1 Prototypes

- `Object` (`??sec:std-Object`)

20.14.2 Examples

There are several examples of uses of events in the documentation of event-based constructs. See `at` ([Section 19.9.1](#)), `waituntil` ([Section 19.9.5](#)), `whenever` ([Section 19.9.6](#)), and so forth. The tutorial chapter about event-based programming contains other examples, see [Chapter 12](#).

20.14.3 Construction

An `Event` is created like any other object, without arguments.

```
var e = Event.new;  
[00000001] Event_0x9ad8118
```

20.14.4 Slots

- `asEvent`
Return `this`.
- `'emit'`
Throw an `Event`. This function is called by the bang operator. It takes any number of arguments, passed to the receiver when the event is caught. An event can also be emitted

for a certain duration using `~`. The execution of `at` clauses etc., is asynchronous: the control flow might be released by the `emit` call before all the watchers have finished their execution.

- `onSubscribe`

This slot is not set by default. You can optionally assign an event to it. In this case, `onSubscribe` is triggered each time some code starts watching this event (by setting up an `at` or a `waituntil` on it for instance).

Throw a synchronized event. This call awaits that all functions that have to react to this event have returned. This function can have the same arguments as `emit`.

- `trigger`

This function is used to launch an event during an unknown amount of time. Calling this function launches and keeps the event triggered and returns an object whose `stop` method stops launching the event.

- `||(other)`

Logical or on events: a new Event that triggers whenever *this* or *other* triggers.

```
var e1 = Event.new|;
var e2 = Event.new|;
var e_or = e1 || e2|;
at (e_or?)
  echo("!");
e1!;
[00000004] *** !
e2!;
[00000005] *** !
```

- `'<<'(other)`

Watch an *other* event status and reproduce it on itself, return `this`. This operator is similar to an optimized `||=` operator. Do not make events watch for themselves, directly or indirectly.

```
var e3 = Event.new|;
var e4 = Event.new|;
var e_watch = Event.new << e3 << e4 |;
at (e_watch?)
  echo("!");
e3!;
[00000006] *** !
e4!;
[00000007] *** !
```

20.15 Exception

Exceptions are used to handle errors. More generally, they are a means to escape from the normal control-flow to handle exceptional situations.

The language support for throwing and catching exceptions (using `try/catch` and `throw`, see [Section 19.7](#)) work perfectly well with any kind of object, yet it is a good idea to throw only objects that derive from `Exception`.

20.15.1 Prototypes

- `Object` ([??sec:std-Object](#))

20.15.2 Construction

There are several types of exceptions, each of which corresponding to a particular kind of error. The top-level object, `Exception`, takes a single argument: an error message.

```
Exception.new("something bad has happened!");
[00000001] Exception 'something bad has happened!'
Exception.Arity.new("myRoutine", 1, 10, 23);
[00000002] Exception.Arity 'myRoutine: expected between 10 and 23 arguments, given 1'
```

20.15.3 Slots

`Exception` has many slots which are specific exceptions. See [Section 20.15.4](#) for their documentation.

- `backtrace`

The call stack at the moment the exception was thrown (not created), as a `List` ([??sec:std-List](#)) of `StackFrames` ([??sec:std-StackFrame](#)), from the innermost to the outermost call.

```
//#push 1 "file.u"
try
{
  function innermost () { throw Exception.new("Ouch") };
  function inner      () { innermost() };
  function outer      () { inner() };
  function outermost () { outer() };

  outermost();
}
catch (var e)
{
  assert
  {
    e.backtrace[0].location.asString == "file.u:4.27-37";
    e.backtrace[0].name == "innermost";

    e.backtrace[1].location.asString == "file.u:5.27-33";
    e.backtrace[1].name == "inner";

    e.backtrace[2].location.asString == "file.u:6.27-33";
    e.backtrace[2].name == "outer";
  }
}
```

```
e.backtrace[3].location.asString == "file.u:8.3-13";
e.backtrace[3].name == "outermost";
};
};
// #pop
```

- **location**

The location from which the exception was thrown (not created).

```
eval("1/0");
[00090441:error] !!! 1.1-3: /: division by 0
try
{
  eval("1/0");
}
catch (var e)
{
  assert (e.location.asString == "1.1-3");
};
```

- **message**

The error message provided at construction.

```
Exception.new("Ouch").message == "Ouch";
```

20.15.4 Specific Exceptions

- **ArgumentType.new(*routine*, *index*, *effective*, *expected*)**

Derives from [Type](#). The *routine* was called with a *index*-nth argument of type *effective* instead of *expected*.

```
Exception.ArgumentType.new("myRoutine", 1, "hisResult", "myException");
[00000003] Exception.ArgumentType 'myRoutine: unexpected "hisResult" for argument 1, expected a String'
```

- **Arity.new(*routine*, *effective*, *min*, *max* = void)**

The *routine* was called with an incorrect number of arguments (*effective*). It requires at least *min* arguments, and, if specified, at most *max*.

```
Exception.Arity.new("myRoutine", 1, 10, 23);
[00000004] Exception.Arity 'myRoutine: expected between 10 and 23 arguments, given 1'
```

- **BadInteger.new(*routine*, *fmt*, *effective*)**

The *routine* was called with an inappropriate integer (*effective*). Use the format *fmt* to create an error message from *effective*. Derives from [BadNumber](#).

```
Exception.BadInteger.new("myRoutine", "bad integer: %s", 12);
[00000005] Exception.BadInteger 'myRoutine: bad integer: 12'
```

- `BadNumber.new(routine, fmt, effective)`

The *routine* was called with an inappropriate number (*effective*). Use the format *fmt* to create an error message from *effective*.

```
Exception.BadNumber.new("myRoutine", "bad number: %s", 12.34);
[00000005] Exception.BadNumber 'myRoutine: bad number: 12.34'
```

- `Constness.new(msg)`

An attempt was made to change a constant value.

```
Exception.Constness.new;
[00000006] Exception.Constness 'cannot modify const slot'
```

- `FileNotFound.new(name)`

The file named *name* cannot be found.

```
Exception.FileNotFound.new("foo");
[00000007] Exception.FileNotFound 'file not found: foo'
```

- `ImplicitTagComponent.new(msg)`

An attempt was made to create an implicit tag, a component of which being undefined.

```
Exception.ImplicitTagComponent.new;
[00000008] Exception.ImplicitTagComponent 'invalid component in implicit tag'
```

- `Lookup.new(object, name)`

A failed name lookup was performed on *object* to find a slot named *name*. If `Exception.Lookup.fixSpelling` is true (which is the default), suggest what the user might have meant to use.

```
Exception.Lookup.new(Object, "GetSlot");
[00000009] Exception.Lookup 'lookup failed: Object'
```

- `MatchFailure.new`

A pattern matching failed.

```
Exception.MatchFailure.new;
[00000010] Exception.MatchFailure 'pattern did not match'
```

- `NegativeNumber.new(routine, effective)`

The *routine* was called with a negative number (*effective*). Derives from [BadNumber](#).

```
Exception.NegativeNumber.new("myRoutine", -12);
[00000005] Exception.NegativeNumber 'myRoutine: expected non-negative number, got -12'
```

- `NonPositiveNumber.new(routine, effective)`

The *routine* was called with a non-positive number (*effective*). Derives from [BadNumber](#).

```
Exception.NonPositiveNumber.new("myRoutine", -12);
[00000005] Exception.NonPositiveNumber 'myRoutine: expected positive number, got -12'
```

- `Primitive.new(routine, msg)`

The built-in *routine* encountered an error described by *msg*.

```
Exception.Primitive.new("myRoutine", "cannot do that");
[00000011] Exception.Primitive 'myRoutine: cannot do that'
```

- `Redefinition.new(name)`

An attempt was made to refine a slot named *name*.

```
Exception.Redefinition.new("foo");
[00000012] Exception.Redefinition 'slot redefinition: foo'
```

- `Scheduling.new(msg)`

Something really bad has happened with the Urbi task scheduler.

```
Exception.Scheduling.new("cannot schedule");
[00000013] Exception.Scheduling 'cannot schedule'
```

- `Syntax.new(loc, message, input)`

Declare a syntax error in *input*, at location *loc*, described by *message*. *loc* is the location of the syntax error, *location* is the place the error was thrown. They are usually equal, except when the errors are caught while using `System.eval` or `System.load`. In that case *loc* is really the position of the syntax error, while *location* refers to the location of the `System.eval` or `System.load` invocation.

```
Exception.Syntax.new(Location.new(Position.new("file.u", 14, 25)),
    "unexpected pouCharque", "file.u");
[00000013] Exception.Syntax 'syntax error: file.u:14.25: unexpected pouCharque'

try
{
    eval("1 / / 0");
}
catch (var e)
{
    assert
    {
        e.isA(Exception.Syntax);
        e.loc.asString == "1.5";
        e.input == "1 / / 0";
        e.message == "unexpected /";
    }
};
```

- `Type.new(effective, expected)`

A value of type *effective* was received, while a value of type *expected* was expected.

```
Exception.Type.new("hisResult", "myException");
[00000014] Exception.Type 'unexpected "hisResult", expected a String'
```

- **UnexpectedVoid.new**

An attempt was made to read the value of `void`.

```
Exception.UnexpectedVoid.new;
[00000015] Exception.UnexpectedVoid 'unexpected void'
var a = void;
a;
[00000016:error] !!! unexpected void
[00000017:error] !!! lookup failed: a
```

20.16 Executable

This class is used only as a common ancestor to [Primitive \(??sec:std-Primitive\)](#) and [Code \(??sec:std-Code\)](#).

20.16.1 Prototypes

- [Object \(??sec:std-Object\)](#)

20.16.2 Construction

There is no point in constructing an Executable.

20.16.3 Slots

- `asExecutable`
Return `this`.

20.17 File

20.17.1 Prototypes

- [Object \(??sec:std-Object\)](#)

20.17.2 Construction

Files may be created from a [String \(??sec:std-String\)](#), or from a [Path \(??sec:std-Path\)](#). The file must exist on the file system, and must be a file. You may use `create` to create a file that does not exist (or to override an existing one).


```
System.system("(echo 1; echo 2) >file.txt")|;
File.new("file.txt");
[00000001] File("file.txt")

File.new(Path.new("file.txt"));
[00000001] File("file.txt")
```

You may use [InputStream \(??sec:std-InputStream\)](#) and [OutputStream \(??sec:std-OutputStream\)](#) to read or write to Files.

20.17.3 Slots

- **asList**

Read the file, and return its content as a list of its lines.

```
System.system("(echo 1; echo 2) >file.txt")|;
assert(File.new("file.txt").asList == ["1", "2"]);
```

- **asPrintable**

```
System.system("(echo 1; echo 2) >file.txt")|;
assert(File.new("file.txt").asPrintable == "File(\"file.txt\")");
```

- **asString**

The name of the opened file.

```
System.system("(echo 1; echo 2) >file.txt")|;
assert(File.new("file.txt").asString == "file.txt");
```

- **content**

The content of the file as a [Binary \(??sec:std-Binary\)](#) object.

```
System.system("(echo 1; echo 2) >file.txt")|;
assert(File.new("file.txt").content.data == "1\n2\n");
```

- **create(name)**

If the file *name* exists, return a File to it, otherwise create an empty one, and return a File to it. See [OutputStream \(??sec:std-OutputStream\)](#) for methods to add content to the file.

```
System.system("(echo 1; echo 2) >file.txt")|;
assert
{
  File.create("file.txt").asPrintable == "File(\"file.txt\")";
  File.new("file.txt").content.data == "1\n2\n";
  File.create("new.txt").content.empty;
};
```

- `remove`

Remove the current file.

```
System.system("(echo 1; echo 2) >file.txt")|;
File.new("file.txt").remove;
assert(!Path.new("file.txt").exists);
```

- `rename(name)`

Rename the file to *name*. If the target exists, it is replaced by the opened file.

```
System.system("(echo 1; echo 2) >foo.txt")|;
File.new("foo.txt").rename("bar.txt");
assert
{
  !Path.new("foo.txt").exists;
  File.new("bar.txt").content.data == "1\n2\n";
};
```

20.18 Finalizable

Objects that derive from this object will execute their `finalize` routine right before being destroyed (reclaimed) by the system. It is comparable to a *destructor*.

20.18.1 Example

The following object is set up to die verbosely.

```
var obj =
  do (Finalizable.new)
  {
    function finalize ()
    {
      echo ("Ouch");
    }
  }|;
```

It is reclaimed by the system when it is no longer referenced by any other object.

```
var alias = obj|;
obj = nil|;
```

Here, the object is still alive, since `alias` references it. Once it no longer does, the object dies.

```
alias = nil|;
[00000004] *** Ouch
```

20.18.2 Prototypes

- `Object (??sec:std-Object)`

20.18.3 Construction

The constructor takes no argument.

```
Finalizable.new;  
[00000527] Finalizable_0x135360
```

Because of specific constraints of Finalizable, you cannot change the prototype of an object to make it “finalizable”: it *must* be an instance of Finalizable from its inception.

There, instead of these two invalid constructs,

```
class o1 : Finalizable.new  
{  
  function finalize()  
  {  
    echo("Ouch");  
  }  
};  
[00000008:error] !!! <empty>: cannot inherit from a Finalizable
```

```
class o2  
{  
  protos = [Finalizable];  
  function finalize()  
  {  
    echo("Ouch");  
  }  
};  
[00000010:error] !!! updateSlot: cannot inherit from a Finalizable
```

write:

```
var o3 =  
  do (Finalizable.new)  
  {  
    function finalize()  
    {  
      echo("Ouch");  
    }  
  }  
};
```

If you need multiple prototypes, do as follows.

```
class Global.Foo  
{  
  function init()  
  {  
    echo("1");  
  };  
};  
};  
  
class Global.FinalizableFoo  
{  
  addProto(Foo.new);  
  
  function 'new'()  
  {  
    var r = clone |  
    r.init |  
  }  
};
```

```

    Finalizable.new.addProto(r);
  };

  function init()
  {
    echo("2");
  };

  function finalize()
  {
    echo("3");
  };
}||;

var i = FinalizableFoo.new|;
[00000117] *** 1
[00000117] *** 2

i = nil;
[00000117] *** 3

```

20.18.4 Slots

- **finalize**
a simple function that takes no argument that will be evaluated when the object is reclaimed. Its return value is ignored.

```

Finalizable.new.setSlot("finalize", function() { echo("Ouch") })||;
[00033240] *** Ouch

```

20.19 Float

A Float is a floating point number. It is also used, in the current version of urbiscript, to represent integers.

20.19.1 Prototypes

- [Comparable](#) (??sec:std-Comparable)
- [Orderable](#) (??sec:std-Orderable)
- [RangeIterable](#) (??sec:std-RangeIterable)

20.19.2 Construction

The most common way to create fresh floats is using the literal syntax. Numbers are composed of three parts:

integral (mandatory) a non empty sequence of (decimal) digits;

fractional (optional) a period, and a non empty sequence of (decimal) digits;

exponent (optional) either ‘e’ or ‘E’, an optional sign (‘+’ or ‘-’), then a non-empty sequence of digits.

In other words, float literals match the `[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?` regular expression. For instance:

```
0 == 0000.0000;
// This is actually a call to the unary '+'.
+1 == 1;
0.123456 == 123456 / 1000000;
1e3 == 1000;
1e-3 == 0.001;
1.234e3 == 1234;
```

There are also some special numbers, `nan`, `inf` (see below).

```
Math.log(0) == -inf;
Math.exp(-inf) == 0;
(inf/inf).asString == "nan";
```

A null float can also be obtained with `Float`’s `new` method.

```
Float.new == 0;
```

20.19.3 Slots

- **abs**
Absolute value of the target.

```
(-5).abs == 5;
0 .abs == 0;
5 .abs == 5;
```

- **acos**
Arccosine of the target.

```
0.acos == Float.pi/2;
1.acos == 0;
```

- **asBool**
Whether non null.

```
0.asBool == false;
0.1.asBool == true;
(-0.1).asBool == true;
inf.asBool == true;
nan.asBool == true;
```

- **asFloat**
Return the target.

```
51.asFloat == 51;
```

- **asList**
Bounces to **seq**. It is therefore possible to use the various flavors of **for**-range loops on integers:

```
{  
  var res = [];  
  for (var i : 3)  
    res << i;  
  res  
}  
== [0, 1, 2];  
  
{  
  var res = [];  
  for|(var i : 3)  
    res << i;  
  res  
}  
== [0, 1, 2];  
  
{  
  var res = [];  
  for&(var i : 3)  
    res << i;  
  res.sort  
}  
== [0, 1, 2];
```

- **asin**
Arcsine of the target.

```
0.asin == 0;
```

- **asString**
Return a string representing the target.

```
42.asString == "42";
```

- **atan**
Return the arctangent of the target.

```
0.atan == 0;  
1.atan == Float.pi/4;
```

- `'bitand'(that)`

The bitwise-and between `this` and `that`.

```
(3 bitand 6) == 2;
```

- `'bitor'(that)`

Bitwise-or between `this` and `that`.

```
(3 bitor 6) == 7;
```

- `clone`

Return a fresh Float with the same value as the target.

```
var x = 0;
[00000000] 0
var y = x.clone;
[00000000] 0
x === y;
[00000000] false
```

- `compl`

The complement to 1 of the target interpreted as a 32 bits integer.

```
compl 0 == 4294967295;
compl 4294967295 == 0;
```

- `cos`

Cosine of the target.

```
0.cos == 1;
Float.pi.cos == -1;
```

- `each(fun)`

Call the functional argument *fun* on every integer from 0 to target - 1, sequentially. The number must be non-negative.

```
{
  var res = [];
  3.each(function (i) { res << 100 + i });
  res
}
== [100, 101, 102];

{
  var res = [];
  for(var x : 3) { res << x; sleep(20ms); res << (100 + x); };
  res
}
== [0, 100, 1, 101, 2, 102];
```

```
{
  var res = [];
  0.each (function (i) { res << 100 + i });
  res
}
== [];
```

- `each|(fun)`

Call the functional argument *fun* on every integer from 0 to target - 1, with tight sequentiality. The number must be non-negative.

```
{
  var res = [];
  3.'each|'(function (i) { res << 100 + i });
  res
}
== [100, 101, 102];

{
  var res = [];
  for|(var x : 3) { res << x; sleep(20ms); res << (100 + x); };
  res
}
== [0, 100, 1, 101, 2, 102];
```

- `each&(fun)`

Call the functional argument *fun* on every integer from 0 to target - 1, concurrently. The number must be non-negative.

```
{
  var res = [];
  for& (var x : 3) { res << x; sleep(30ms); res << (100 + x) };
  res
}
== [0, 1, 2, 100, 101, 102];
```

- `exp`

Exponential of the target.

```
1.exp;
[00000000] 2.71828
```

- `format(finfo)`

Format according to the [FormatInfo](#) (`??sec:std-FormatInfo`) object *finfo*. The precision, *finfo.precision*, sets the maximum number of digits after decimal point when in fixed or scientific mode, and in total when in default mode. Beware that 0 plays a special role, as it is not a “significant” digit.

Windows Issues

Under Windows the behavior differs slightly.

```
"%1.0d" % 0.1 == "0.1";
"%1.0d" % 1.1 == {if (System.Platform.isWindows) "1.1" else "1"};

"%1.0f" % 0.1 == "0";
"%1.0f" % 1.1 == "1";
```

- **inf**

Return the infinity.

```
Float.inf;
[00000000] inf
```

- **limits**

See [Float.limits](#) ([??sec:std-Float.limits](#)).

- **log**

The logarithm of the target.

```
0.log == -inf;
1.log == 0;
1.exp.log == 1;
```

- **max(*arg1*, ...)**

Bounces to [List.max](#) on [*this*, *arg1*, ...].

```
1.max == 1;
1.max(2, 3) == 3;
3.max(1, 2) == 3;
```

- **min(*arg1*, ...)**

Bounces to [List.min](#) on [*this*, *arg1*, ...].

```
1.min == 1;
1.min(2, 3) == 1;
3.min(1, 2) == 1;
```

- **nan**

The “not a number” special float value. More precisely, this returns the “quiet NaN”, i.e., it is propagated in the various computations, it does not raise exceptions.

```
Float.nan;
[00000000] nan
(Float.nan + Float.nan) / (Float.nan - Float.nan);
[00000000] nan
```

A NaN has one distinctive property over the other Floats: it is equal to no other float, not even itself. This behavior is mandated by the [IEEE 754-2008](#) standard.

```
{ var n = Float.nan; n === n};
{ var n = Float.nan; n != n};
```

- **pi**
 π .

```
Float.pi.cos ** 2 + Float.pi.sin ** 2 == 1;
```

- **random**

A random integer between 0 (included) and the target (excluded).

```
20.map(function (dummy) { 5.random });
[00000000] [1, 2, 1, 3, 2, 3, 2, 2, 4, 4, 4, 1, 0, 0, 0, 3, 2, 4, 3, 2]
```

- **round**

The target, rounded to the nearest integer.

```
1.6.round == 2;
1.4.round == 1;
```

- **seq**

The sequence of integers from 0 to [this](#)- 1 as a list. The number must be non-negative.

```
3.seq == [0, 1, 2];
0.seq == [];
```

- **sign**

Return 1 if [this](#) is positive, 0 if it is null, -1 otherwise.

```
(-1164).sign == -1;
0.sign      == 0;
(1164).sign == 1;
```

- **sin**

The sine of the target.

```
0.sin == 0;
```

- **sqr**

Square of the target.

```
32.sqr == 1024;
32.sqr == 32 ** 2;
```

- `sqrt`

The square root of the target.

```
1024.sqrt == 32;
1024.sqrt == 1024 ** 0.5;
```

- `srandom`

Initialized the seed used by the random function. As opposed to common usage, you should not use

```
{
  var now = Date.now.timestamp;
  now.srandom;
  var list1 = 20.map(function (dummy) { 5.random });
  now.srandom;
  var list2 = 20.map(function (dummy) { 5.random });
  assert
  {
    list1 == list2;
  }
};
```

- `tan`

Tangent of the target.

```
assert(0.tan == 0);
(Float.pi/4).tan;
[00000000] 1
```

- `times(fun)`

Call the functional argument *fun* **this** times.

```
3.times(function () { echo("ping")});
[00000000] *** ping
[00000000] *** ping
[00000000] *** ping
```

- `trunc`

Return the target truncated.

```
1.9.trunc == 1;
(-1.9).trunc == -1;
```

- `'^'(that)`

Bitwise exclusive or between **this** and *that*.

```
(3 ^ 6) == 5;
```

- `'>>'(that)`

`this` shifted by `that` bits towards the right.

```
4 >> 2 == 1;
```

- `'<'(that)`

Whether `this` is less than `b`. The other comparison operators (`<=`, `>`, ...) can thus also be applied on floats since Float inherits `Orderable` ([??sec:std-Orderable](#)).

```
0 < 1;
!(1 < 0);
!(0 < 0);
```

- `'<<'(that)`

`this` shifted by `that` bit towards the left.

```
4 << 2 == 16;
```

- `'-(that)`

`this` subtracted by `b`.

```
6 - 3 == 3;
```

- `'+(that)`

The sum of `this` and `that`.

```
1 + 1 == 2;
```

- `'/(that)`

The quotient of `this` divided by `that`.

```
50 / 10 == 5;
10 / 50 == 0.2;
```

- `'%(that)`

`this` modulo `b`.

```
50 % 11 == 6;
```

- `'*(that)`

Product of `this` by `that`.

```
2 * 3 == 6;
```

- `'**'(that)`

`this` to the `that` power ($this^{that}$).

```
2 ** 10 == 1024;
```

- `'=='` (*that*)

Whether `this` equals *that*.

```
1 == 1;  
!(1 == 2);
```

20.20 Float.limits

This singleton handles various limits related to the Float objects.

20.20.1 Slots

- `digits`

Number of digits (in `radix` base) in the mantissa.

```
Float.limits.digits;
```

- `digits10`

Number of digits (in decimal base) that can be represented without change.

```
Float.limits.digits10;
```

- `epsilon`

Machine epsilon (the difference between 1 and the least value greater than 1 that is representable).

```
1 != 1 + Float.limits.epsilon;  
1 == 1 + Float.limits.epsilon / 2;
```

- `max`

Maximum finite value.

```
Float.limits.max != Float.inf;  
Float.limits.max * 2 == Float.inf;
```

- `maxExponent`

Maximum integer value for the exponent that generates a normalized floating-point number.

```
Float.inf != Float.limits.radix ** (Float.limits.maxExponent - 1);  
Float.inf == Float.limits.radix ** Float.limits.maxExponent;
```

- `maxExponent10`

Maximum integer value such that 10 raised to that power generates a normalized finite floating-point number.

```
Float.inf != 10 ** Float.limits.maxExponent10;
Float.inf == 10 ** (Float.limits.maxExponent10 + 1);
```

- **min**
Minimum positive normalized value.

```
0 != Float.limits.min;
```

- **minExponent**
Minimum negative integer value for the exponent that generates a normalized floating-point number.

```
0 != Float.limits.radix ** Float.limits.minExponent;
```

- **minExponent10**
Minimum negative integer value such that 10 raised to that power generates a normalized floating-point number.

```
0 != 10 ** Float.limits.minExponent10;
```

- **radix**
Base of the exponent of the representation.

```
Float.limits.radix == 2;
```

20.21 FormatInfo

A *format info* is used when formatting a la `printf`. It store the formatting pattern itself and all the format information it can extract from the pattern.

20.21.1 Prototypes

- [Object](#) ([??sec:std-Object](#))

20.21.2 Construction

The constructor expects a string as argument, whose syntax is similar to `printf`'s. It is detailed below.

```
var f = FormatInfo.new("%+2.3d");
[00000001] %+2.3d
```

A formatting pattern must one of the following (brackets denote optional arguments):

- *%options spec*

- `%|options[spec]|`

options is a sequence of 0 or several of the following characters:

'-'	Left alignment.
'='	Centered alignment.
'+'	Show sign even for positive number.
' '	If the string does not begin with '+' or '-', insert a space before the converted string.
'0'	Pad with 0's (inserted after sign or base indicator).
'#'	Show numerical base, and decimal point.

spec is the conversion character and must be one of the following:

's'	Default character, prints normally
'd'	Case modifier: lowercase
'D'	Case modifier: uppercase
'x'	Prints in hexadecimal lowercase
'X'	Prints in hexadecimal uppercase
'o'	Prints in octal
'e'	Prints floats in scientific format
'E'	Prints floats in scientific format uppercase
'f'	Prints floats in fixed format

20.21.3 Slots

- **alignment**

Requested alignment: -1 for left, 0 for centered, 1 for right (default).

```
FormatInfo.new("%s").alignment == 1;
FormatInfo.new("%=s").alignment == 0;
FormatInfo.new("%-s").alignment == -1;
```

- **alt**

Whether the “alternative” display is requested ('#').

```
FormatInfo.new("%s").alt == false;
FormatInfo.new("%#s").alt == true;
```

- **group**

Separator to use for thousands. Corresponds to the *',' option*.

```
FormatInfo.new("%s").group == "";
FormatInfo.new("%'s").group == " ";
```

- **pad**

The padding character to use for alignment requests. Defaults to space.

```
FormatInfo.new("%s").pad == " ";
FormatInfo.new("%0s").pad == "0";
```

- **pattern**

The pattern given to the constructor.

```
FormatInfo.new("%#'12.8s").pattern == "%#'12.8s";
```

- **precision**

When formatting a [Float](#) ([??sec:std-Float](#)), the maximum number of digits after decimal point when in fixed or scientific mode, and in total when in default mode. When formatting other objects with spec-char 's', the conversion string is truncated to the precision first chars. The eventual padding to width is done after truncation.

```
FormatInfo.new("%s").precision == 6;
FormatInfo.new("%23.3s").precision == 3;
```

- **prefix**

The string to display before positive numbers. Defaults to empty.

```
FormatInfo.new("%s").prefix == "";
FormatInfo.new("% s").prefix == " ";
FormatInfo.new("%+s").prefix == "+";
```

- **spec**

The specification character, regardless of the case conversion requests.

```
FormatInfo.new("%s").spec == "s";
FormatInfo.new("%23.3s").spec == "s";
FormatInfo.new("%'X").spec == "x";
```

- **uppercase**

Case conversion: -1 for lower case, 0 for no conversion (default), 1 for conversion to uppercase. The value depends on the case of specification character, except for 's' which corresponds to 0.

```
FormatInfo.new("%s").uppercase == 0;
FormatInfo.new("%d").uppercase == -1;
FormatInfo.new("%D").uppercase == 1;
FormatInfo.new("%x").uppercase == -1;
FormatInfo.new("%X").uppercase == 1;
```

- **width**

Width requested for alignment.

```
FormatInfo.new("%s").width == 0;
FormatInfo.new("%10s").width == 10;
```


20.22 Formatter

A *formatter* stores format information of a format string like used in `printf` in the C library or in `boost::format`.

20.22.1 Prototypes

- `Object` ([??sec:std-Object](#))

20.22.2 Construction

Formatters are created with the format string. It cuts the string to separate regular parts of string and formatting patterns, and stores them.

```
Formatter.new("Name:%s, Surname:%s;");
[00000001] Formatter ["Name:", %s, ", Surname:", %s, ";"]
```

Actually, formatting patterns are translated into `FormatInfo` ([??sec:std-FormatInfo](#)).

20.22.3 Slots

- `asList`

Return the content of the *formatter* as a list of strings and `FormatInfo` ([??sec:std-FormatInfo](#)).

```
Formatter.new("Name:%s, Surname:%s;").asList.asString
== "[\"Name:\", %s, \", Surname:\", %s, \";\"]";
```

- `'%(args)`

Use [this](#) as format string and *args* as the list of arguments, and return the result (a `String` ([??sec:std-String](#))). The arity of the `Formatter` (i.e., the number of expected arguments) and the size of *args* must match exactly.

This operator concatenates regular strings and the strings that are result of `asString` called on members of *args* with the appropriate `FormatInfo` ([??sec:std-FormatInfo](#)).

```
Formatter.new("Name:%s, Surname:%s;") % ["Foo", "Bar"]
== "Name:Foo, Surname:Bar;";
```

If *args* is not a `List` ([??sec:std-List](#)), then the call is equivalent to calling —'

```
Formatter.new("%06.3f") % Math.pi
== "03.142";
```

Note that `String. '%'` provides a nicer interface to this operator:

```
"%06.3f" % Math.pi == "03.142";
```

It is nevertheless interesting to use the `Formatter` for performance reasons if the format is reused many times.

```

{
  // Some large database of people.
  var people =
    [
      ["Foo", "Bar" ],
      ["One", "Two" ],
      ["Un",  "Deux"],,];
  var f = Formatter.new("Name:%7s, Surname:%7s;");
  for (var p: people)
    echo (f % p);
};
[00031939] *** Name:   Foo, Surname:   Bar;
[00031940] *** Name:   One, Surname:   Two;
[00031941] *** Name:    Un, Surname:  Deux;

```

20.23 Global

Global is designed for the purpose of being global namespace. Since *Global* is a prototype of *Object* and all objects are an *Object*, all slots of *Global* are accessible from anywhere.

20.23.1 Prototypes

- [uobjects](#), see below.
- [Tag.tags](#) (see [Tag \(??sec:std-Tag\)](#))
- [Math \(??sec:std-Math\)](#)
- [System \(??sec:std-System\)](#)
- [Object \(??sec:std-Object\)](#)

20.23.2 Slots

- [Barrier](#)
See [Barrier \(??sec:std-Barrier\)](#).
- [Binary](#)
See [Binary \(??sec:std-Binary\)](#).
- [CallMessage](#)
See [CallMessage \(??sec:std-CallMessage\)](#).
- [cerr](#)
A predefined stream for error messages. Strings sent to it are not escaped, contrary to regular streams (see [output](#) for instance).

```
cerr << "Message";  
[00015895:error] Message  
cerr << "\"quote\"";  
[00015895:error] "quote"
```

- **Channel**
See [Channel](#) (??sec:std-Channel).
- **clog**
A predefined stream for log messages. Strings are output escaped.

```
clog << "Message";  
[00015895:clog] "Message"
```

- **Code**
See [Code](#) (??sec:std-Code).
- **Comparable**
See [Comparable](#) (??sec:std-Comparable).
- **cout**
A predefined stream for output messages. Strings are output escaped.

```
cout << "Message";  
[00015895:output] "Message"  
cout << "\"quote\"";  
[00015895:output] "\"quote\""
```

- **Date**
See [Date](#) (??sec:std-Date).
- **detach(*exp*)**
Bounce to [Control.detach](#), see [Control](#) (??sec:std-Control).
- **Dictionary**
See [Dictionary](#) (??sec:std-Dictionary).
- **Directory**
See [Directory](#) (??sec:std-Directory).
- **disown(*exp*)**
Bounce to [Control.disown](#), see [Control](#) (??sec:std-Control).
- **Duration**
See [Duration](#) (??sec:std-Duration).
- **echo(*value*, *channel* = "")**
Bounce to [lobby.echo](#), see [Lobby](#) (??sec:std-Lobby).

```

echo("111", "foo");
[00015895:foo] *** 111
echo(222, "");
[00051909] *** 222
echo(333);
[00055205] *** 333

```

- **evaluate**

This [UVar](#) ([??sec:std-UVar](#)) provides a synchronous interface to the Urbi engine: write to it to “send” an expression to compute it, and “read” it to get the result. This UVar is designed to be used from the C++; it makes little sense in urbiscript, use [System.eval](#) instead, if it is really required (see [Section 16.1](#)). Since the semantics of the assignment requires that it evaluates to the right-hand side argument, reading `evaluate` after the assignment is needed, which makes race conditions likely. To avoid this, use `|` (or better yet, do not use `evaluate` at all in urbiscript).

```

(evaluate = "1+2;") == "1+2;";
evaluate == 3;
{ evaluate = "1+2;" | evaluate } == 3;
{ evaluate = "var x = 1;"; x } == 1;

```

Errors raise an exception.

```

evaluate = "1/0;";
[00087671:error] !!! Exception caught while processing notify on Global.evaluate:
[00087671:error] !!! 1.1-3: /: division by 0
[00087671] "1/0;"

```

- **Event**

See [Event](#) ([??sec:std-Event](#)).

- **Exception**

See [Exception](#) ([??sec:std-Exception](#)).

- **Executable**

See [Executable](#) ([??sec:std-Executable](#)).

- **external**

An system object used to implement UObject support in urbiscript.

- **false**

See [Section 20.3.3](#).

- **File**

See [File](#) ([??sec:std-File](#)).

- **Finalizable**

See [Finalizable](#) ([??sec:std-Finalizable](#)).

- `Float`
See [Float](#) ([??sec:std-Float](#)).
- `FormatInfo`
See [FormatInfo](#) ([??sec:std-FormatInfo](#)).
- `Formatter`
See [Formatter](#) ([??sec:std-Formatter](#)).
- `getProperty(slotName, propName)`
This wrapper around [Object.getProperty](#) is actually a by-product of the existence of the [evaluate UVar](#) ([??sec:std-UVar](#)).
- `Global`
See [Global](#) ([??sec:std-Global](#)).
- `Group`
See [Group](#) ([??sec:std-Group](#)).
- `InputStream`
See [InputStream](#) ([??sec:std-InputStream](#)).
- `isdef(qualifiedIdentifier)`
Whether the *qualifiedIdentifier* is defined. It features some (fragile) magic to support an argument passed as a literal (`isdef(foo)`), not a string (`isdef("foo")`). It is not recommended to use this feature, which is provided for urbiscript compatibility. See [Object.hasLocalSlot](#) and [Object.hasSlot](#) for safer alternatives.

```

assert
{
  !isdef(a);
  !isdef(a.b);
  !isdef(a.b.c);
};

var a = Object.new|;
assert
{
  isdef(a);
  !isdef(a.b);
  !isdef(a.b.c);
};

var a.b = Object.new|;
assert
{
  isdef(a);
  isdef(a.b);
  !isdef(a.b.c);
};

```

```

var a.b.c = Object.new|;
assert
{
  isdef(a);
  isdef(a.b);
  isdef(a.b.c);
};

```

- Job
See [Job](#) (??sec:std-Job).
- Kernel1
See [Kernel1](#) (??sec:std-Kernel1).
- Lazy
See [Lazy](#) (??sec:std-Lazy).
- List
See [List](#) (??sec:std-List).
- Loadable
See [Loadable](#) (??sec:std-Loadable).
- Lobby
See [Lobby](#) (??sec:std-Lobby).
- Math
See [Math](#) (??sec:std-Math).
- methodToFunction(*name*)
Create a function from the method *name* so that calling the function with arguments (*a*, *b*, ...) is that same as calling *a.name(b, ...)*.

```

var uid_of = methodToFunction("uid")|;
assert
{
  uid_of(Object) == Object.uid;
  uid_of(Global) == Global.uid;
};
var '+_of' = methodToFunction("+")|;
assert
{
  '+_of'( 1, 2) == 1 + 2;
  '+_of'("1", "2") == "1" + "2";
  '+_of'([1], [2]) == [1] + [2];
};

```

- Mutex
See [Mutex](#) (??sec:std-Mutex).

- `nil`
See `nil` (`??sec:std-nil`).
- `Object`
See `Object` (`??sec:std-Object`).
- `Orderable`
See `Orderable` (`??sec:std-Orderable`).
- `OutputStream`
See `OutputStream` (`??sec:std-OutputStream`).
- `Pair`
See `Pair` (`??sec:std-Pair`).
- `Path`
See `Path` (`??sec:std-Path`).
- `Pattern`
See `Pattern` (`??sec:std-Pattern`).
- `persist(exp)`
Bounce to `Control.persist`, see `Control` (`??sec:std-Control`).
- `Position`
See `Position` (`??sec:std-Position`).
- `Primitive`
See `Primitive` (`??sec:std-Primitive`).
- `Process`
See `Process` (`??sec:std-Process`).
- `Profiling`
See `Profiling` (`??sec:std-Profiling`).
- `PseudoLazy`
See `PseudoLazy` (`??sec:std-PseudoLazy`).
- `PubSub`
See `PubSub` (`??sec:std-PubSub`).
- `RangeIterable`
See `RangeIterable` (`??sec:std-RangeIterable`).
- `Regexp`
See `Regexp` (`??sec:std-Regexp`).
- `Semaphore`
See `Semaphore` (`??sec:std-Semaphore`).

- **Server**
See [Server](#) ([??sec:std-Server](#)).
- **Singleton**
See [Singleton](#) ([??sec:std-Singleton](#)).
- **Socket**
See [Socket](#) ([??sec:std-Socket](#)).
- **String**
See [String](#) ([??sec:std-String](#)).
- **System**
See [System](#) ([??sec:std-System](#)).
- **Tag**
See [Tag](#) ([??sec:std-Tag](#)).
- **Timeout**
See [Timeout](#) ([??sec:std-Timeout](#)).
- **TrajectoryGenerator**
See [TrajectoryGenerator](#) ([??sec:std-TrajectoryGenerator](#)).
- **Triplet**
See [Triplet](#) ([??sec:std-Triplet](#)).
- **true**
See [Section 20.3.3](#).
- **Tuple**
See [Tuple](#) ([??sec:std-Tuple](#)).
- **UObject**
See [UObject](#) ([??sec:std-UObject](#)).
- **uobjects**
An object whose slots are all the [UObject](#) ([??sec:std-UObject](#)) bound into the system.
- **UValue**
See [UValue](#) ([??sec:std-UValue](#)).
- **UVar**
See [UVar](#) ([??sec:std-UVar](#)).
- **void**
See [void](#) ([??sec:std-void](#)).

- `wall(value, channel = "")`
Bounce to `lobby.wall`, see [Lobby \(??sec:std-Lobby\)](#).

```
wall("111", "foo");
[00015895:foo] *** 111
wall(222, "");
[00051909] *** 222
wall(333);
[00055205] *** 333
```

- `warn(message)`
Issue *message* on `Channel.warning`.

```
warn("cave canem");
[00015895:warning] *** cave canem
```

20.24 Group

A transparent means to send messages to several objects as if they were one.

20.24.1 Example

The following session demonstrates the features of the Group objects. It first creates the `Sample` family of object, makes a group of such object, and uses that group.

```
class Sample
{
  var value = 0;
  function init(v) { value = v; };
  function asString() { "<" + value.asString + ">"; };
  function timesTen() { new(value * 10); };
  function plusTwo() { new(value + 2); };
};
[00000000] <0>

var group = Group.new(Sample.new(1), Sample.new(2));
[00000000] Group [<1>, <2>]
group << Sample.new(3);
[00000000] Group [<1>, <2>, <3>]
group.timesTen.plusTwo;
[00000000] Group [<12>, <22>, <32>]

// Bouncing getSlot and updateSlot.
group.value;
[00000000] Group [1, 2, 3]
group.value = 10;
[00000000] Group [10, 10, 10]

// Bouncing to each&.
var sum = 0|
```

```
for& (var v : group)
  sum += v.value;
sum;
[00000000] 30
```

20.24.2 Prototypes

- `Object (??sec:std-Object)`

20.24.3 Construction

Groups are created like any other object. The constructor can take members to add to the group.

```
Group.new;
[00000000] Group []
Group.new(1, "two");
[00000000] Group [1, "two"]
```

20.24.4 Slots

- `add(member, ...)`
Add members to `this` group, and return `this`.
- `asString`
Report the members.
- `each(action)`
Apply *action* to all the members, in sequence, then return the Group of the results, in the same order. Allows to iterate over a Group via `for`.
- `each&(action)`
Apply *action* to all the members, concurrently, then return the Group of the results. The order is *not* necessarily the same. Allows to iterate over a Group via `for&`.
- `fallback`
This function is called when a method call on `this` failed. It bounces the call to the members of the group, collects the results returned as a group. This allows to chain grouped operation in a row. If the dispatched calls return `void`, returns a single `void`, not a “group of void”.
- `getProperty(slot, prop)`
Bounced to the members so that `this.slot->prop` actually collects the values of the property *prop* of the slots *slot* of the group members.
- `hasProperty(name)`
Bounced to the members.

- `remove(member, ...)`
Remove members from `this` group, and return `this`.
- `setProperty(slot, prop, value)`
Bounced to the members so that `this.slot->prop = value` actually updates the value of the property `prop` in the slots `slot` of the group members.
- `updateSlot(name, value)`
Bounced to the members so that `this.name = value` actually updates the value of the slot `name` in the group members.
- `'<<' (member)`
Syntactic sugar for `add`.

20.25 InputStream

InputStreams are used to read (possibly binary) files by hand. `File` ([??sec:std-File](#)) provides means to swallow a whole file either as a single large string, or a list of lines. `InputStream` provides a more fine-grained interface to read files.

20.25.1 Prototypes

- `Object` ([??sec:std-Object](#))

Windows Issues

Beware that because of limitations in the current implementation, one cannot safely read from two different files at the same time under Windows.

20.25.2 Construction

An `InputStream` is a reading-interface to a file, so its constructor requires a `File` ([??sec:std-File](#)).

```
System.system("(echo 1; echo 2) >file.txt")|;
InputStream.new(File.new("file.txt"));
[00000001] InputStream_0x827000
```

20.25.3 Slots

- `get`
Get the next available byte as a `Float` ([??sec:std-Float](#)), or `void` if the end of file was reached.

```
System.system("(echo 1; echo 2) >file.txt")|;
assert(
{
  var res = [];
```

```

var i = InputStream.new(File.new("file.txt"));
var x;
while (!(x = i.get.acceptVoid).isVoid)
  res << x;
res;
}
==
[49, 10, 50, 10]);

```

- `getChar`

Get the next available byte as a `String` (`??sec:std-String`), or `void` if the end of file was reached.

```

System.system("(echo 1; echo 2) >file.txt");
assert(
{
  var res = [];
  var i = InputStream.new(File.new("file.txt"));
  var x;
  while (!(x = i.getChar.acceptVoid).isVoid)
    res << x;
  res;
}
==
["1", "\n", "2", "\n"]);

```

- `getLine`

Get the next available line as a `String` (`??sec:std-String`), or `void` if the end of file was reached.

```

System.system("(echo 1; echo 2) >file.txt");
assert(
{
  var res = [];
  var i = InputStream.new(File.new("file.txt"));
  var x;
  while (!(x = i.getLine.acceptVoid).isVoid)
    res << x;
  res;
}
==
["1", "2"]);

```

20.26 IoService

A *IoService* is used to manage the various operations of a set of `Socket` (`??sec:std-Socket`).

All `Socket` (`??sec:std-Socket`) and `Server` (`??sec:std-Server`) are by default using the default `IoService` (`??sec:std-IoService`) which is polled regularly by the system.

20.26.1 Example

Using a different `IoService` (`??sec:std-IoService`) is required if you need to perform synchronous read operations.

The `Socket` (`??sec:std-Socket`) must be created by the `IoService` (`??sec:std-IoService`) that will handle it using its `makeSocket` function.

```
var io = IoService.new|;
var s = io.makeSocket|;
```

You can then use this socket like any other.

```
// Make a simple hello server.
var serverPort = 0|
do(Server.new)
{
  listen("127.0.0.1", "0");
  lobby.serverPort = port;
  at(connection?(var s))
  {
    s.write("hello");
  }
}|;
// Connect to it using our socket.
s.connect("0.0.0.0", serverPort);
at(s.received?(var data))
  echo("received something");
s.write("1;");
```

... except that nothing will be read from the socket unless you call one of the `poll` functions of `io`.

```
sleep(200ms);
s.isConnected(); // Nothing was received yet
[00000001] true
io.poll();
[00000002] *** received something
sleep(200ms);
```

20.26.2 Prototypes

- `Object` (`??sec:std-Object`)

20.26.3 Construction

A `IoService` is constructed with no argument.

20.26.4 Slots

- `makeServer`
Create and return a new `Server` (`??sec:std-Server`) using this `IoService` (`??sec:std-IoService`).

- `makeSocket`
Create and return a new `Socket` (`??sec:std-Socket`) using this `IoService` (`??sec:std-IoService`).
- `poll`
Handle all pending socket operations(read, write, accept) that can be performed without waiting.
- `pollFor(duration)`
Will block for *duration* seconds, and handle all ready socket operations during this period.
- `pollOneFor(duration)`
Will block for at most *duration*, and handle the first ready socket operation and immediately return.

20.27 Job

Jobs are independent threads of executions. Jobs can run concurrently. They can also be managed using `Tags` (`??sec:std-Tag`).

20.27.1 Prototypes

- `Object` (`??sec:std-Object`)

20.27.2 Construction

A Job is typically constructed via `Control.detach`, `Control.disown`, or `System.spawn`.

```
detach(sleep(10));
[00202654] Job<shell_4>

disown(sleep(10));
[00204195] Job<shell_5>

spawn (function () { sleep(10) }, false);
[00274160] Job<shell_6>
```

20.27.3 Slots

- `asJob`
Return `this`.
- `asString`
The string `Job<name>` where *name* is the name of the job.
- `backtrace`
The current backtrace of the job as a list of `StackFrames` (`??sec:std-StackFrame`).

```

// #push 1 "file.u"
var s = detach(sleep(10));
// Leave some time for s to be started.
sleep(1);
assert
{
  s.backtrace[0].asString == "file.u:1.16-24: sleep";
  s.backtrace[1].asString == "file.u:1.9-25: detach";
};
// #pop

```

- **clone**
Cloning a job is impossible since Job is considered as being an atom.
- **dumpState**
Pretty-print the state of the job.

```

// #push 1 "file.u"
var t = detach(sleep(10));
// Leave some time for s to be started.
sleep(1);
t.dumpState;
[00004295] *** Job: shell_10
[00004295] ***   State: sleeping
[00004295] ***   Tags:
[00004295] ***     Tag<Lobby_1>
[00004297] ***   Backtrace:
[00004297] ***     file.u:1.16-24: sleep
[00004297] ***     file.u:1.9-25: detach
// #pop

```

- **name**
The name of the job.

```

detach(sleep(10)).name;
[00004297] "shell_5"

```

- **setSideEffectFree(*value*)**
If *value* is true, mark the current job as side-effect free. It indicates whether the current state may influence other parts of the system. This is used by the scheduler to choose whether other jobs need scheduling or not. The default value is false.
- **status**
The current status of the job (starting, running, ...), and its properties (frozen, ...).
- **tags**
The list of [Tags](#) ([??sec:std-Tag](#)) that manage this job.
- **terminate**
Kill this job.

```
var r = detach({ sleep(1s); echo("done") })|;
assert (r in jobs);
r.terminate;
assert (r not in jobs);
sleep(2s);
```

- **timeShift**

Get the total amount of time during which we were frozen.

```
tag: r = detach({ sleep(3); echo("done") })|;
tag.freeze();
sleep(2);
tag.unfreeze();
Math.round(r.timeShift);
[00000001] 2
```

- **waitForChanges**

Resume the scheduler, putting the current Job in a waiting status. The scheduler may reschedule the job immediately.

- **waitForTermination**

Wait for the job to terminate before resuming execution of the current one. If the job has already terminated, return immediately.

20.28 Kernel1

This object plays the role of a name-space in which obsolete functions from urbiscript 1.0 are provided for backward compatibility. Do not use these functions, scheduled for removal.

20.28.1 Prototypes

- [Singleton](#) ([??sec:std-Singleton](#))

20.28.2 Construction

Since it is a [Singleton](#) ([??sec:std-Singleton](#)), you are not expected to build other instances.

20.28.3 Slots

- **commands**
Ignored for backward compatibility.
- **connections**
Ignored for backward compatibility.
- **copy(*binary*)**
Obsolete syntax for *binary*.copy, see [Binary](#) ([??sec:std-Binary](#)).


```
// copy.
var a = BIN 10;0123456789
[00000001] BIN 10
0123456789

var b = Kernel1.copy(a);
[00000003:warning] *** 'copy(binary)' is deprecated, use 'binary.copy'
[00000004] BIN 10
0123456789

echo (b);
[00000005] *** BIN 10
0123456789
```

- **devices**
Ignored for backward compatibility.
- **events**
Ignored for backward compatibility.
- **functions**
Ignored for backward compatibility.
- **isvoid(*obj*)**
Obsolete syntax for *obj.isVoid*, see [Object \(??sec:std-Object\)](#).
- **noop**
Do nothing. Use {} instead.
- **ping**
Return time verbosely, see [System \(??sec:std-System\)](#).

```
Kernel1.ping;
[00000421] *** pong time=0.12s
```

- **reset**
Ignored for backward compatibility.
- **runningcommands**
Ignored for backward compatibility.
- **seq(*number*)**
Obsolete syntax for *number.seq*, see [Float \(??sec:std-Float\)](#).
- **size(*list*)**
Obsolete syntax for *list.size*, see [List \(??sec:std-List\)](#).

```
Kernel1.size([1, 2, 3]) == [1, 2, 3].size;
[00000002:warning] *** 'size(list)' is deprecated, use 'list.size'
```

- `strict`
Ignored for backward compatibility.
- `strlen(string)`
Obsolete syntax for `string.length`, see [String \(??sec:std-String\)](#).

```
Kernel1 strlen("123") == "123".length;
[00000002:warning] *** 'strlen(string)' is deprecated, use 'string.length'
```

- `taglist`
Ignored for backward compatibility.
- `undefall`
Ignored for backward compatibility.
- `unstrict`
Ignored for backward compatibility.
- `uservars`
Ignored for backward compatibility.
- `vars`
Ignored for backward compatibility.

20.29 Lazy

Lazies are objects that hold a lazy value, that is, a not yet evaluated value. They provide facilities to evaluate their content only once (*memoization*) or several times. Lazy are essentially used in call messages, to represent lazy arguments, as described in [Section 20.4](#).

20.29.1 Examples

20.29.1.1 Evaluating once

One usage of lazy values is to avoid evaluating an expression unless it's actually needed, because it's expensive or has undesired side effects. The listing below presents a situation where an expensive-to-compute value (`heavy_computation`) might be needed zero, one or two times. The objective is to save time by:

- Not evaluating it if it's not needed.
- Evaluating it only once if it's needed once or twice.

We thus make the wanted expression lazy, and use the `value` method to fetch its value when needed.

```
// This function supposedly performs expensive computations.
function heavy_computation()
{
    echo("Heavy computation");
    return 1 + 1;
};

// We want to do the heavy computations only if needed,
// and make it a lazy value to be able to evaluate it "on demand".
var v = Lazy.new(closure () { heavy_computation() });
[00000000] heavy_computation()
/* some code */;
// So far, the value was not needed, and heavy_computation
// was not evaluated.
/* some code */;
// If the value is needed, heavy_computation is evaluated.
v.value;
[00000000] *** Heavy computation
[00000000] 2
// If the value is needed a second time, heavy_computation
// is not reevaluated.
v.value;
[00000000] 2
```

20.29.1.2 Evaluating several times

Evaluating a lazy several times only makes sense with lazy arguments and call messages. See example with call messages in [Section 20.4.1.1](#).

20.29.2 Caching

[Lazy \(??sec:std-Lazy\)](#) is meant for functions without argument. If you need *caching* for functions that depend on arguments, it is straightforward to implement using a [Dictionary \(??sec:std-Dictionary\)](#). In the future urbiscript might support dictionaries whose indices are not only strings, but in the meanwhile, convert the arguments into strings, as the following sample object demonstrates.

```
class UnaryLazy
{
    function init(f)
    {
        results = [ => ];
        func = f;
    };
    function value(p)
    {
        var sp = p.asString;
        if (results.has(sp))
            return results[sp];
        var res = func(p);
```

```

    results[sp] = res |
    res
};
var results;
var func;
} |
// The function to cache.
var inc = function(x) { echo("incing " + x) | x+1 } |
// The function with cache.
// Use "getSlot" to get the unevaluated function.
var p = UnaryLazy.new(getSlot("inc"));
[00062847] UnaryLazy_0x78b750
p.value(1);
[00066758] *** incing 1
[00066759] 2
p.value(1);
[00069058] 2
p.value(2);
[00071558] *** incing 2
[00071559] 3
p.value(2);
[00072762] 3
p.value(1);
[00074562] 2

```

20.29.3 Prototypes

- [Comparable](#) (??sec:std-Comparable)

20.29.4 Construction

Lazies are seldom instantiated manually. They are mainly created automatically when a lazy function call is made (see [Section 19.3.4](#)). One can however create a lazy value with the standard `new` method of `Lazy`, giving it an argument-less function which evaluates to the value made lazy.

```

Lazy.new(closure () { /* Value to make lazy */ 0 });
[00000000] 0

```

20.29.5 Slots

- Whether `this` and `that` are the same source code and value (an unevaluated `Lazy` is never equal to an evaluated one).

```

Lazy.new(closure () { 1 + 1 }) == Lazy.new(closure () { 1 + 1 });
Lazy.new(closure () { 1 + 2 }) != Lazy.new(closure () { 2 + 1 });

```

```

{
  var l1 = Lazy.new(closure () { 1 + 1 });
  var l2 = Lazy.new(closure () { 1 + 1 });
  assert (l1 == l2);
}

```

```

11.eval;
assert (11 != 12);
12.eval;
assert (11 == 12);
};

```

- **asString**

The conversion to **String** (**??sec:std-String**) of the body of a non-evaluated argument.

```

Lazy.new(closure () { echo(1); 1 }).asString == "echo(1);\n1";

```

- **eval**

Force the evaluation of the held lazy value. Two calls to **eval** will systematically evaluate the expression twice, which can be useful to duplicate its side effects.

- **value**

Return the held value, potentially evaluating it before. **value** performs memoization, that is, only the first call will actually evaluate the expression, subsequent calls will return the cached value. Unless you want to explicitly trigger side effects from the expression by evaluating it several time, this should be preferred over **eval** to avoid evaluating the expression several times uselessly.

20.30 List

Lists implement potentially-empty ordered (heterogeneous) collections of elements.

20.30.1 Prototypes

- **Container** (**??sec:std-Container**)
- **RangeIterable** (**??sec:std-RangeIterable**)
- **Orderable** (**??sec:std-Orderable**)

20.30.2 Construction

List can be created with their literal syntax: a possibly empty sequence of expressions in square brackets, separated by commas. Non-empty list may actually *terminate* with a comma, rather than *separate*; in other words, an optional trailing comma is accepted.

```

[]; // The empty list
[00000000] []
[1, "2", [3],,];
[00000000] [1, "2", [3]]

```

20.30.3 Slots

- `argMax(fun = function(a, b)`
`a ; b)` The index of the (leftmost) “largest” member based on the comparison function *fun*.

```
[1].argMax == 0;
[1, 2].argMax == 1;
[1, 2, 2].argMax == 1;
[2, 1].argMax == 0;
[2, -1, 3, -4].argMax == 2;

[2, -1, 3, -4].argMax (function (a, b) { a.abs < b.abs }) == 3;
```

The list cannot be empty.

```
[] .argMax;
[00000001:error] !!! list cannot be empty
```

- `argMin(fun = function(a, b)`
`a ; b)` The index of the (leftmost) “smallest” member based on the comparison function *fun*.

```
[1].argMin == 0;
[1, 2].argMin == 0;
[1, 2, 1].argMin == 0;
[2, 1].argMin == 1;
[2, -1, 3, -4].argMin == 3;

[2, -1, 3, -4].argMin (function (a, b) { a.abs < b.abs }) == 1;
```

The list cannot be empty.

```
[] .argMin;
[00000001:error] !!! list cannot be empty
```

- `asBool`
Whether not empty.

```
[] .asBool == false;
[1].asBool == true;
```

- `asList`
Return the target.

```
{
  var l = [0, 1, 2];
  assert (l.asList === l);
};
```

- **asString**

A string describing the list. Uses `asPrintable` on its members, so that, for instance, strings are displayed with quotes.

```
[0, [1], "2"].asString == "[0, [1], \"2\"]";
```

- **back**

The last element of the target. An error if the target is empty.

```
assert([0, 1, 2].back == 2);
[].back;
[00000000:error] !!! back: cannot be applied onto empty list
```

- **clear**

Empty the target.

```
var x = [0, 1, 2];
[00000000] [0, 1, 2]
assert(x.clear == []);
```

- **each(*fun*)**

Apply the given functional value *fun* on all members, sequentially.

```
[0, 1, 2].each(function (v) {echo (v * v); echo (v * v)});
[00000000] *** 0
[00000000] *** 0
[00000000] *** 1
[00000000] *** 1
[00000000] *** 4
[00000000] *** 4
```

- **eachi(*fun*)**

Apply the given functional value *fun* on all members sequentially, additionally passing the current element index.

```
["a", "b", "c"].eachi(function (v, i) {echo ("%s: %s" % [i, v])});
[00000000] *** 0: a
[00000000] *** 1: b
[00000000] *** 2: c
```

- **'each&'(*fun*)**

Apply the given functional value on all members simultaneously.

```
[0, 1, 2].'each&'(function (v) {echo (v * v); echo (v * v)});
[00000000] *** 0
[00000000] *** 1
[00000000] *** 4
[00000000] *** 0
[00000000] *** 1
[00000000] *** 4
```

- `empty`

Whether the target is empty.

```
[].empty;
! [1].empty;
```

- `filter(fun)`

The list of all the members of the target that verify the predicate *fun*.

```
do ([0, 1, 2, 3, 4, 5])
{
  assert
  {
    // Keep only odd numbers.
    filter(function (v) {v % 2 == 1}) == [1, 3, 5];
    // Keep all.
    filter(function (v) { true })      == this;
    // Keep none.
    filter(function (v) { false })     == [];
  };
};
```

- `foldl(action, value)`

Fold, also known as *reduce* or *accumulate*, computes a result from a list. Starting from *value* as the initial result, apply repeatedly the binary *action* to the current result and the next member of the list, from left to right. For instance, if *action* were the binary addition and *value* were 0, then folding a list would compute the sum of the list, including for empty lists.

```
[] .foldl(function (a, b) { a + b }, 0) == 0;
[1, 2, 3] .foldl(function (a, b) { a + b }, 0) == 6;
[1, 2, 3] .foldl(function (a, b) { a - b }, 0) == -6;
```

- `front`

Return the first element of the target. An error if the target is empty.

```
assert([0, 1, 2].front == 0);
[].front;
[00000000:error] !!! front: cannot be applied onto empty list
```

- `has(that)`

Whether one of the members of the target equals the argument.

```
[0, 1, 2].has(1);
! [0, 1, 2].has(5);
```

The infix operators `in` and `not in` use `has` (see [Section 19.1.8.6](#)).


```

1 in    [0, 1];
2 not in [0, 1];
!(2 in  [0, 1]);
!(1 not in [0, 1]);

```

- `hasSame(that)`

Whether one of the members of the target is physically equal to *that*.

```

var y = 1;
assert
{
  [0, y, 2].hasSame(y);
  ![0, y, 2].hasSame(1);
};

```

- `head`

Synonym for `front`.

```

assert([0, 1, 2].head == 0);
[].head;
[00000000:error] !!! head: cannot be applied onto empty list

```

- `insertBack(that)`

Insert the given element at the end of the target.

```

do ([0, 1])
{
  assert
  {
    insertBack(2) == [0, 1, 2];
    this          == [0, 1, 2];
  };
};

```

- `insert(where, what)`

Insert *what* before the value at index *where*, return `this`.

```

do ([0, 1])
{
  assert
  {
    insert(0, 10) == [10, 0, 1];
    this          == [10, 0, 1];
    insert(2, 20) == [10, 0, 20, 1];
    this          == [10, 0, 20, 1];
  };
  insert(4, 30);
};
[00044239:error] !!! insert: invalid index: 4

```

The index must be valid, to insert past the end, use [insertBack](#).

```
[].insert(0, "foo");
[00044239:error] !!! insert: invalid index: 0
```

- `insertFront(that)`

Insert the given element at the beginning of the target.

```
do ([1, 2])
{
  assert
  {
    insertFront(0) == [0, 1, 2];
    this           == [0, 1, 2];
  };
};
```

- `join(sep = "", prefix = "", suffix = "")`

Bounce to [String.join](#).

```
["", "ob", ""].join      == "ob";
["", "ob", ""].join("a") == "aoba";
["", "ob", ""].join("a", "B", "b") == "Baobab";
```

- `keys`

The list of valid indexes. This allows uniform iteration over a [Dictionary](#) ([??sec:std-Dictionary](#)) or a [List](#) ([??sec:std-List](#)).

```
{
  var l = ["a", "b", "c"];
  assert
  {
    l.keys == [0, 1, 2];
    {
      var res = [];
      for (var k: l.keys)
        res << l[k];
      res
    }
    == l;
  };
};
```

- `map(fun)`

Apply the given functional value on every member, and return the list of results.

```
[0, 1, 2, 3].map(function (v) { v % 2 == 0 })
== [true, false, true, false];
```

- `max(fun = function(a, b)`
`a | b)` Return the “largest” member based on the comparison function *fun*.

```
[1].max == 1;
[1, 2].max == 2;
[2, 1].max == 2;
[2, -1, 3, -4].max == 3;

[2, -1, 3, -4].max (function (a, b) { a.abs < b.abs }) == -4;
```

The list cannot be empty.

```
[] .max;
[00000001:error] !!! list cannot be empty
```

- `min(fun = function(a, b)`
`a | b)` Return the “smallest” member based on the comparison function *fun*.

```
[1].min == 1;
[1, 2].min == 1;
[2, 1].min == 1;
[2, -1, 3, -4].min == -4;

[2, -1, 3, -4].min (function (a, b) { a.abs < b.abs }) == -1;
```

The list cannot be empty.

```
[] .min;
[00000001:error] !!! list cannot be empty
```

- `range(begin, end = nil)`
Return a sub-range of the list, from the first index included to the second index excluded.
An error if out of bounds. Negative indices are valid, and number from the end.

If *end* is `nil`, calling `range(n)` is equivalent to calling `range(0, n)`.

```
do ([0, 1, 2, 3])
{
  assert
  {
    range(0, 0) == [];
    range(0, 1) == [0];
    range(1) == [0];
    range(1, 3) == [1, 2];

    range(-3, -2) == [1];
    range(-3, -1) == [1, 2];
    range(-3, 0) == [1, 2, 3];
    range(-3, 1) == [1, 2, 3, 0];
    range(-4, 4) == [0, 1, 2, 3, 0, 1, 2, 3];
  };
};
```

```
[].range(1, 3);
[00428697:error] !!! range: invalid index: 1
```

- **remove(*val*)**

Remove all elements from the target that are equal to *val*, return *this*.

```
var c = [0, 1, 0, 2, 0, 3]||;
assert
{
  c.remove(0) === c;   c == [1, 2, 3];
  c.remove(42) === c;  c == [1, 2, 3];
};
```

- **removeBack**

Remove and return the last element of the target. An error if the target is empty.

```
var t = [0, 1, 2];
[00000000] [0, 1, 2]
assert(t.removeBack == 2);
assert(t == [0, 1]);
[].removeBack;
[00000000:error] !!! removeBack: cannot be applied onto empty list
```

- **removeById(*that*)**

Remove all elements from the target that physically equals *that*.

```
var d = 1|;
var e = [0, 1, d, 1, 2]||;
assert
{
  e.removeById(d) == [0, 1, 1, 2];
  e == [0, 1, 1, 2];
};
```

- **removeFront**

Remove and return the first element from the target. An error if the target is empty.

```
var g = [0, 1, 2]||;
assert
{
  g.removeFront == 0;
  g == [1, 2];
};
[].removeFront;
[00000000:error] !!! removeFront: cannot be applied onto empty list
```

- **reverse**

Return the target with the order of elements inverted.

```
[0, 1, 2].reverse == [2, 1, 0];
```

- **size**

Return the number of elements in the target.

```
[0, 1, 2].size == 3;
[].size == 0;
```

- **sort**

Return the target, sorted with respect to the < criteria.

```
[1, 0, 3, 2].sort == [0, 1, 2, 3];
```

- **subset(*that*)**

Whether the members of **this** are members of *that*.

```
{}.subset({});
{}.subset([1, 2, 3]);
[3, 2, 1].subset([1, 2, 3]);
[1, 3].subset([1, 2, 3]);
[1, 1].subset([1, 2, 3]);
! [3].subset({});
! [3, 2].subset([1, 2]);
! [1, 2, 3].subset([1, 2]);
```

- **tail**

Return the target, minus the first element. An error if the target is empty.

```
assert([0, 1, 2].tail == [1, 2]);
[].tail;
[00000000:error] !!! tail: cannot be applied onto empty list
```

- **zip(*fun*, *other*)**

Zip **this** list and the *other* list with the *fun* function, and return the list of results.

```
[1, 2, 3].zip(closure (x, y) { (x, y) }, [4, 5, 6])
  == [(1, 4), (2, 5), (3, 6)];
[1, 2, 3].zip(closure (x, y) { x + y }, [4, 5, 6])
  == [5, 7, 9];
```

- **'=='(*that*)**

Check whether all elements in the target and *that*, are equal two by two.

```
[0, 1, 2] == [0, 1, 2];
!([0, 1, 2] == [0, 0, 2]);
```

- **'[]'(*n*)**

Return the *n*th member of the target (indexing is zero-based). If *n* is negative, start from the end. An error if out of bounds.

```

assert(["0", "1", "2"][0] == "0");
assert(["0", "1", "2"][2] == "2");
["0", "1", "2"][3];
[00007061:error] !!! [: invalid index: 3

assert(["0", "1", "2"][-1] == "2");
assert(["0", "1", "2"][-3] == "0");
["0", "1", "2"][-4];
[00007061:error] !!! [: invalid index: -4

```

- `'[]='(index, value)`

Assign *value* to the element of the target at the given *index*.

```

var f = [0, 1, 2];
[00000000] [0, 1, 2]
f[1] = 42;
[00000000] 42
assert(f == [0, 42, 2]);

```

- `'*'(n)`

Return the target, concatenated *n* times to itself.

```

[0, 1] * 0 == [];
[0, 1] * 3 == [0, 1, 0, 1, 0, 1];

```

n must be a non-negative integer.

```

[0, 1] * -2;
[00000063:error] !!! *: expected non-negative integer, got -2

```

Note that since it is the very same list which is repeatedly concatenated (the content is not cloned), side-effects on one item will reflect on “all the items”.

```

var l = [[]] * 3;
[00000000] [[], [], []]
l[0] << 1;
[00000000] [1]
l;
[00000000] [[1], [1], [1]]

```

- `'+'(other)`

Return the concatenation of the target and the *other* list.

```

[0, 1] + [2, 3] == [0, 1, 2, 3];
[] + [2, 3] == [2, 3];
[0, 1] + [] == [0, 1];
[] + [] == [];

```

- `'-'(other)`

Return the target without the elements that are equal to any element in the *other* list.

```
[0, 1, 0, 2, 3] - [1, 2] == [0, 0, 3];
[0, 1, 0, 1, 0] - [1, 2] == [0, 0, 0];
```

- '*<<*' (*that*)

A synonym for `insertBack`.

- '*<*' (*other*)

Return whether `this` is less than the *other* list. This is the lexicographic comparison: `this` is “less than” if one of its member is “less than” the corresponding member of *other*:

```
[0, 0, 0] < [0, 0, 1];
[0, 1, 2] < [0, 2, 1];

!([0, 1, 2] < [0, 1, 2]);
!([0, 1, 2] < [0, 0, 2]);
```

or *other* is a prefix (strict) of `this`:

```
[] < [0];          !([0] < []);
[0, 1] < [0, 1, 2]; !([0, 1, 2] < [0, 1]);
!([0, 1, 2] < [0, 1, 2]);
```

Since `List` derives from `Orderable` ([??sec:std-Orderable](#)), the other order-based operators are defined.

```
[] <= [];
[] <= [0, 1, 2];
[0, 1, 2] <= [0, 1, 2];

[] >= [];
[0, 1, 2] >= [];
[0, 1, 2] >= [0, 1, 2];
[0, 1, 2] >= [0, 0, 2];

!([0] > []);
[0, 1, 2] > [];
!([0, 1, 2] > [0, 1, 2]);
[0, 1, 2] > [0, 0, 2];
```

20.31 Loadable

Loadable objects can be switched on and off — typically physical devices.

20.31.1 Prototypes

- `Object` ([??sec:std-Object](#))

20.31.2 Example

The intended use is rather as follows:

```
class Motor: Loadable
{
  var val = 0;
  function go(var d)
  {
    if (load)
      val += d
    else
      echo("cannot advance, the motor is off");
  };
};
[00000002] Motor

var m = Motor.new;
[00000003] Motor_0xADDR

m.load;
[00000004] false

m.go(1);
[00000006] *** cannot advance, the motor is off

m.on;
[00000007] Motor_0xADDR

m.go(123);
m.val;
[00000009] 123
```

20.31.3 Construction

Loadable can be constructed, but it hardly makes sense. This object should serve as a prototype.

20.31.4 Slots

- `load`
The current status.
- `off(val)`
Set `load` to false and return `this`.

```
do (Loadable.new)
{
  assert
  {
    !load;
    off === this;
    !load;
  }
```



```

    on === this;
    load;
    off === this;
    !load;
  };
};

```

- `on(val)`
Set `load` to `true` and return `this`.

```

do (Loadable.new)
{
  assert
  {
    !load;
    on === this;
    load;
    on === this;
    load;
  };
};

```

- `toggle`
Set `load` from `true` to `false`, and vice-versa. Return `val`.

```

do (Loadable.new)
{
  assert
  {
    !load;
    toggle === this;
    load;
    toggle === this;
    !load;
  };
};

```

20.32 Lobby

A *lobby* is the local environment for each (remote or local) connection to an Urbi server.

20.32.1 Prototypes

- `Channel.topLevel`, an instance of `Channel` (`??sec:std-Channel`) with an empty Channel name.

20.32.2 Construction

A lobby is implicitly created at each connection. At the top level, `this` is a *Lobby*.

```
this.protos;
[00000001] [Lobby]
this.protos[0].protos;
[00000003] [Channel_0xADDR]
```

Lobbies cannot be cloned, they must be created using `create`.

```
Lobby.new;
[00000177:error] !!! new: 'Lobby' objects cannot be cloned
Lobby.create;
[00000174] Lobby_0x126450
```

20.32.3 Examples

Since every lobby is-a `Channel` (`??sec:std-Channel`), one can use the methods of `Channel`.

```
lobby << 123;
[00478679] 123
lobby << "foo";
[00478679] "foo"
```

20.32.4 Slots

- **authors**
Credit the authors of Urbi SDK.
- **banner**
Internal. Display Urbi SDK banner.

```
lobby.banner;
[00000987] *** *****
[00000990] *** Urbi SDK version 2.0.3 rev. d6a568d
[00001111] *** Copyright (C) 2005-2010 Gostai S.A.S.
[00001111] ***
[00001111] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00001112] *** be used under certain conditions. Type 'license;',
[00001112] *** 'authors;', or 'copyright;' for more information.
[00001112] ***
[00001112] *** Check our community site: http://www.urbiforge.org.
[00001112] *** *****
```

- **connected**
Whether `this` is connected.

```
connected;
```

- **connectionTag**
The tag of all code executed in the context of `this`. This tag applies to `this`, but the top-level loop is immune to `Tag.stop`, therefore `connectionTag` controls every thing that was launched from this lobby, yet the lobby itself is still usable.

```

every (1s) echo(1), sleep(0.5s); every (1s) echo(2),
sleep(1.2s);
connectionTag.stop;
[00000507] *** 1
[00001008] *** 2
[00001507] *** 1
[00002008] *** 2

"We are alive!";
[00002008] "We are alive!"

every (1s) echo(3), sleep(0.5s); every (1s) echo(4),
sleep(1.2s);
connectionTag.stop;
[00003208] *** 3
[00003710] *** 4
[00004208] *** 3
[00004710] *** 4

"and kicking!";
[00002008] "and kicking!"

```

Of course, a background job may stop a foreground one.

```

{ sleep(1.2s); connectionTag.stop; },
// Note the `;`, this is a foreground statement.
every (1s) echo(5);
[00005008] *** 5
[00005508] *** 5

"bye!";
[00006008] "bye!"

```

- `copyright(deep = true)`

Display the copyright of Urbi SDK. Include copyright information about sub-components if *deep*.

```

lobby.copyright(false);
[00006489] *** Urbi SDK version 2.1-rc-2 rev. 35f41b9
[00006489] *** Copyright (C) 2005-2010 Gostai S.A.S.

lobby.copyright(true);
[00020586] *** Urbi SDK version 2.1-rc-2 rev. 35f41b9
[00020593] *** Copyright (C) 2005-2010 Gostai S.A.S.
[00020593] ***
[00020593] *** Urbi SDK Remote version preview/1.6/rc-1 rev. 7bfac02
[00020593] *** Copyright (C) 2005-2010 Gostai S.A.S.
[00020593] ***
[00020593] *** Libport version releases/1.0 rev. f52545a
[00020593] *** Copyright (C) 2005-2010 Gostai S.A.S.

```

- **create**

Instantiate a new Lobby.

```
Lobby.create;
```

- **echo(*value*, *channel* = "")**

Send *value.asString* to *this*, prefixed by the **String** (*sec:std-String*) *channel* name if specified. This is the preferred way to send informative messages (prefixed with ‘***’).

```
lobby.echo("111", "foo");
[00015895:foo] *** 111
lobby.echo(222, "");
[00051909] *** 222
lobby.echo(333);
[00055205] *** 333
```

- **echoEach(*list*, *channel* = "")**

Apply *echo(m, channel)* for each member *m* of *list*.

```
lobby.echo([1, "2"], "foo");
[00015895:foo] *** [1, "2"]

lobby.echoEach([1, "2"], "foo");
[00015895:foo] *** 1
[00015895:foo] *** 2

lobby.echoEach([], "foo");
```

- **instances**

A list of the currently alive lobbies. It contains at least the Lobby object itself, and the current **lobby**.

```
lobby in Lobby.instances;
Lobby in Lobby.instances;
```

- **license**

Display the end user license agreement of the Urbi SDK.

```
lobby.license;
[00000000] *** END USER LICENSE AGREEMENT (1.2)
[00000000] ***
[00000000] *** PLEASE READ THIS AGREEMENT CAREFULLY. BY USING ALL OR ANY PORTION OF
[00000000] *** THE SOFTWARE YOU ("YOU" AND "LICENSEE") ACCEPT THE FOLLOWING TERMS
[00000000] *** FROM GOSTAI S.A.S, FRENCH CORPORATION ("GOSTAI"), REGISTERED AT
[00000000] *** 489 244 624 RCS PARIS. YOU AGREE TO BE BOUND BY ALL THE TERMS AND
[00000000] *** CONDITIONS OF THIS AGREEMENT. YOU AGREE THAT IT IS ENFORCEABLE AS IF
[00000000] *** IT WERE A WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU. IF YOU DO NOT
[00000000] *** AGREE TO THE TERMS OF THIS AGREEMENT YOU MUST NOT USE THE SOFTWARE.
[00000000] ***
[00000000] *** [...]
```

- `lobby`
Return the current lobby, i.e., `this`.

```
Lobby.lobby == this;
```

- `onDisconnect(lobby)`
Event launched when `this` has disconnected.
- `quit`
Shut this lobby down, i.e., close the connection. The server is still running, see `System.shutdown` to quit the server.
- `receive(value)`
This is low-level routine. Pretend the `String (??sec:std-String) value` was received from the connection. There is no guarantee that `value` will be the next program block that will be processed: for instance, if you load a file which, in its middle, uses `lobby.receive("foo")`, then `"foo"` will be appended after the end of the file.

```
Lobby.create.receive("12;");  
[00478679] 12
```

- `send(value, channel = "")`
This is low-level routine. Send the `String (??sec:std-String) value` to `this`, prefixed by the `String (??sec:std-String) channel` name if specified.

```
lobby.send("111", "foo");  
[00015895:foo] 111  
lobby.send("222", "");  
[00051909] 222  
lobby.send("333");  
[00055205] 333
```

- `thanks`
Credit the contributors of Urbi SDK.
- `wall(value, channel = "")`
Perform `echo(value, channel)` on all the existing lobbies (except Lobby itself).

```
Lobby.wall("111", "foo");  
[00015895:foo] *** 111
```

- `write(value)`
This is low-level routine. Send the `String (??sec:std-String) value` to the connection. Note that because of buffering, the output might not be visible before an end-of-line character is output.

```

lobby.write("[");
lobby.write("999999999:");
lobby.write("myTag] ");
lobby.write("Hello, World!");
lobby.write("\n");
[999999999:myTag] Hello, World!

```

20.33 Location

This class aggregates two Positions and provides a way to print them as done in error messages.

20.33.1 Prototypes

- `Object` (`??sec:std-Object`)

20.33.2 Construction

Without argument, a newly constructed Location has its Positions initialized to the first line and the first column.

```

Location.new;
[000000001] 1.1

```

With a Position argument *p*, the Location will clone the Position into the begin and end Positions.

```

Location.new(Position.new("file.u",14,25));
[000000001] file.u:14.25

```

With two Positions arguments *begin* and *end*, the Location will clone both Positions into its own fields.

```

Location.new(Position.new("file.u",14,25), Position.new("file.u",14,35));
[000000001] file.u:14.25-34

```

20.33.3 Slots

- `'=='` (*other*)

Compare the begin and end Position.

```

{
  var p1 = Position.new("file.u",14,25);
  var p2 = Position.new("file.u",16,35);
  var p3 = Position.new("file.u",18,45);
  assert {
    Location.new(p1, p3) != Location.new(p1, p2);
    Location.new(p1, p3) == Location.new(p1, p3);
    Location.new(p1, p3) != Location.new(p2, p3);
  };
}

```

```
};
```

- **asString**

Present Locations with less variability as possible as either:

- `'file:ll.cc'`
- `'file:ll.cc-cc'`
- `'file:ll.cc-ll.cc'`

or the same without file name when the file name is not defined.

```
Location.new(Position.new("file.u",14,25)).asString == "file.u:14.25";
Location.new(Position.new(14,25)).asString == "14.25";

Location.new(
  Position.new("file.u",14,25),
  Position.new("file.u",14,35)
).asString == "file.u:14.25-34";

Location.new(
  Position.new(14,25),
  Position.new(14,35)
).asString == "14.25-34";

Location.new(
  Position.new("file.u",14,25),
  Position.new("file.u",15,35)
).asString == "file.u:14.25-15.34";

Location.new(
  Position.new(14,25),
  Position.new(15,35)
).asString == "14.25-15.34";
```

- **begin**

The begin Position used by the Location. Modifying a copy of this field does not modify the Location.

```
Location.new(
  Position.new("file.u",14,25),
  Position.new("file.u",16,35)
).begin == Position.new("file.u",14,25);
```

- **end**

The end Position used by the Location. Modifying a copy of this field does not modify the Location.

```
Location.new(
  Position.new("file.u",14,25),
  Position.new("file.u",16,35)
).end == Position.new("file.u",16,35);
```

20.34 Math

This object is actually meant to play the role of a name-space in which the mathematical functions are defined with a more conventional notation. Indeed, in an object-oriented language, writing `pi.cos` makes perfect sense, yet `cos(pi)` is more usual.

20.34.1 Prototypes

- [Singleton \(??sec:std-Singleton\)](#)

20.34.2 Construction

Since it is a [Singleton \(??sec:std-Singleton\)](#), you are not expected to build other instances.

20.34.3 Slots

- `abs(float)`
Bounce to `float.abs`.

```
Math.abs(1) == 1;
Math.abs(-1) == 1;
Math.abs(0) == 0;
Math.abs(3.5) == 3.5;
```

- `acos(float)`
Bounce to `float.acos`.

- `asin(float)`
Bounce to `float.asin`.

- `atan(float)`
Bounce to `float.atan`.

```
Math.atan(1) ~= pi/4;
```

- `atan2(x, y)`
Bounce to `x.atan2(y)`.

```
Math.atan2(2, 2) ~= pi/4;
Math.atan2(-2, 2) ~= -pi/4;
```

- `cos(float)`
Bounce to `float.cos`.

```
Math.cos(0) == 1;
Math.cos(pi) ~= -1;
```


- `exp(float)`
Bounce to `float.exp`.

- `inf`
Bounce to `Float.inf`.

- `log(float)`
Bounce to `float.log`.

```
Math.log(1) == 0;
```

- `max(arg1, ...)`
Bounce to `[arg1, ...].max`, see `List.max`.

```
max( 100, 20, 3 ) == 100;  
max("100", "20", "3") == "100";
```

- `min(arg1, ...)`
Bounce to `[arg1, ...].min`, see `List.min`.

```
min( 100, 20, 3 ) == 3;  
min("100", "20", "3") == "100";
```

- `nan`
Bounce to `Float.nan`.

- `pi`
Bounce to `Float.pi`.

- `random(float)`
Bounce to `float.random`.

- `round(float)`
Bounce to `float.round`.

```
Math.round(1) == 1;  
Math.round(1.1) == 1;  
Math.round(1.49) == 1;  
Math.round(1.5) == 2;  
Math.round(1.51) == 2;
```

- `sign(float)`
Bounce to `float.sign`.

```
Math.sign(2) == 1;  
Math.sign(-2) == -1;  
Math.sign(0) == 0;
```

- `sin(float)`
Bounce to `float.sin`.

```
Math.sin(0) == 0;
Math.sin(pi) ~= 0;
```

- `sqr(float)`
Bounce to `float.sqr`.

```
Math.sqr(2.2) ~= 4.84;
```

- `sqrt(float)`
Bounce to `float.sqrt`.

```
Math.sqrt(4) == 2;
```

- `srandom(float)`
Bounce to `float.srandom`.

- `tan(float)`
Bounce to `float.tan`.

```
Math.tan(pi/4) ~= 1;
```

- `trunc(float)`
Bounce to `float.trunc`.

```
Math.trunc(1) == 1;
Math.trunc(1.1) == 1;
Math.trunc(1.49) == 1;
Math.trunc(1.5) == 1;
Math.trunc(1.51) == 1;
```

20.35 Mutex

Mutex allow to define critical sections.

20.35.1 Prototypes

- [Tag \(??sec:std-Tag\)](#)

20.35.2 Construction

A *Mutex* can be constructed like any other *Tag* but without name.

```
var m = Mutex.new;
[00000000] Mutex_0x964ed40
```

You can define critical sections by tagging your code using the Mutex.

```
var m = Mutex.new|;
m: echo("this is critical section");
[00000001] *** this is critical section
```

As a critical section, two pieces of code tagged by the same “Mutex” will never be executed at the same time.

20.35.3 Slots

- `asMutex`
Return `this`.

```
var m1 = Mutex.new|;
assert
{
  m1.asMutex === m1;
};
```

20.36 nil

The special entity `nil` is an object used to denote an empty value. Contrary to `void` ([??sec:std-void](#)), it is a regular value which can be read.

20.36.1 Prototypes

- [Singleton](#) ([??sec:std-Singleton](#))

20.36.2 Construction

Being a singleton, `nil` is not to be constructed, just used.

```
nil == nil;
```

20.36.3 Slots

- `isNil`
Whether `this` is `nil`. I.e., true. See also [Object.isNil](#).

```
nil.isNil;
!Object.isNil;
```

20.37 Object

All objects in urbiscript must have [Object](#) ([??sec:std-Object](#)) in their parents. [Object](#) ([??sec:std-Object](#)) is done for this purpose so that it come with many primitives that are mandatory for all object in urbiscript.

20.37.1 Prototypes

- [Orderable](#) ([??sec:std-Orderable](#))
- [Global](#) ([??sec:std-Global](#))

20.37.2 Construction

Fresh object can be instantiated by cloning `Object` itself.

```
Object.new;  
[00000421] Object_0x00000000
```

The keyword `class` also allows to define objects which are intended to serve as prototype of a family of objects, similarly to classes in traditional object-oriented programming languages (see [Section 9.4](#)).

```
{  
  class Foo  
  {  
    var attr = 23;  
  };  
  assert  
  {  
    Foo.localSlotNames == ["asFoo", "attr", "type"];  
    Foo.asFoo === Foo;  
    Foo.attr == 23;  
    Foo.type == "Foo";  
  };  
};
```

20.37.3 Slots

- `acceptVoid`
Return `this`. See [void](#) ([??sec:std-void](#)) to know why.

```
{  
  var o = Object.new;  
  assert(o.acceptVoid === o);  
};
```

- `addProto(proto)`
Add *proto* into the list of prototypes of `this`. Return `this`.

```
do (Object.new)  
{  
  assert  
  {  
    addProto(Orderable) === this;  
    protos == [Orderable, Object];  
  };  
};
```

- `allProto`

Return a list with `this`, all parents of `this`, the parents of the parents of `this`,...

```
123.allProtos.size == 12;
```

- `allSlotNames`

Deprecated alias for `slotNames`.

```
Object.allSlotNames == Object.slotNames;
```

- `apply(args)`

“Invoke `this`”. The size of the argument list, `args`, must be one. This argument is ignored. This function exists for compatibility with `Code.apply`.

```
Object.apply([this]) === Object;
Object.apply([1])    === Object;
```

- `as(type)`

Convert `this` to `type`. This is syntactic sugar for `asType` when `Type` is the type of `type`.

```
12.as(Float) == 12;
"12".as(Float) == 12;
12.as(String) == "12";
Object.as(Object) === Object;
```

- `asBool`

Whether `this` is “true”, see [Section 20.3.3](#).

```
assert(Global.asBool == true);
assert(nil.asBool == false);
void.asBool;
[00000421:error] !!! unexpected void
```

- `bounce(name)`

Return `this.name` transformed from a method into a function that takes its target (its “`this`”) as first and only argument. `this.name` must take no argument.

```
{ var myCos = Object.bounce("cos"); myCos(0) } == 0.cos;
{ var myType = bounce("type"); myType(Object); } == "Object";
{ var myType = bounce("type"); myType(3.14); } == "Float";
```

- `callMessage(msg)`

Invoke the `CallMessage (??sec:std-CallMessage)` `msg` on this.

- `clone`

Clone `this`, i.e., create a fresh, empty, object, which sole prototype is `this`.

```
Object.clone.protos == [Object];
Object.clone.localSlotNames == [];
```

- `cloneSlot(from, to)`

Set the new slot *to* using a clone of *from*. This can only be used into the same object.

```
var foo = Object.new |;
cloneSlot("foo", "bar") |;
assert(!(foo === bar));
```

- `copySlot(from, to)`

Same as `cloneSlot`, but the slot aren't cloned, so the two slot are the same.

```
var moo = Object.new |;
cloneSlot("moo", "loo") |;
assert(!(moo === loo));
```

- `createSlot(name)`

Create an empty slot (which actually means it is bound to `void`) named *name*. Raise an error if *name* was already defined.

```
do (Object.new)
{
  assert(!hasLocalSlot("foo"));
  assert(createSlot("foo").isVoid);
  assert(hasLocalSlot("foo"));
}|;
```

- `dump(depth)`

Describe `this`: its prototypes and slots. The argument *depth* specifies how recursive the description is: the greater, the more detailed. This method is mostly useful for debugging low-level issues, for a more human-readable interface, see also `inspect`.

```
do (2) { var this.attr = "foo"; this.attr->prop = "bar" }.dump(0);
[00015137] *** Float_0x240550 {
[00015137] ***   /* Special slots */
[00015137] ***   protos = Float
[00015137] ***   value = 2
[00015137] ***   /* Slots */
[00015137] ***   attr = String_0x23a750 <...>
[00015137] ***   /* Properties */
[00015137] ***   prop = String_0x23a7a0 <...>
[00015137] *** }
do (2) { var this.attr = "foo"; this.attr->prop = "bar" }.dump(1);
[00020505] *** Float_0x240550 {
[00020505] ***   /* Special slots */
[00020505] ***   protos = Float
[00020505] ***   value = 2
[00020505] ***   /* Slots */
[00020505] ***   attr = String_0x23a750 {
[00020505] ***     /* Special slots */
[00020505] ***     protos = String
[00020505] ***     /* Slots */
[00020505] ***   }
[00020505] *** }
```

```
[00020505] ***      /* Properties */
[00020505] ***      prop = String_0x239330 {
[00020505] ***          /* Special slots */
[00020505] ***          protos = String
[00020505] ***          /* Slots */
[00020505] ***      }
[00020505] *** }
```

- `getPeriod`
Deprecated. Use `System.period` instead.
- `getProperty(slotName, propName)`
Return the value of the `propName` property associated to the slot `slotName` if defined, void otherwise.

```
const var myPi = 3.14|;
assert
{
  getProperty("myPi", "constant");
  getProperty("myPi", "foobar").isVoid;
};
```

- `getLocalSlot(name)`
The value associated to `name` in `this`, excluding its ancestors (contrary to `getSlot`).

```
var a = Object.new|;

// Local slot.
var a.slot = 21|;
assert
{
  a.locateSlot("slot") === a;
  a.getLocalSlot("slot") == 21;
};

// Inherited slot are not looked-up.
assert { a.locateSlot("init") == Object };
a.getLocalSlot("init");
[00041066:error] !!! lookup failed: init
```

- `getSlot(name)`
The value associated to `name` in `this`, possibly after a look-up in its prototypes (contrary to `getLocalSlot`).

```
var b = Object.new|;
var b.slot = 21|;

assert
{
  // Local slot.
  b.locateSlot("slot") === b;
```

```

b.getSlot("slot") == 21;

// Inherited slot.
b.locateSlot("init") === Object;
b.getSlot("init") == Object.getSlot("init");
};

// Unknown slot.
assert { b.locateSlot("ENOENT") == nil; };
b.getSlot("ENOENT");
[00041066:error] !!! lookup failed: ENOENT

```

- `hasLocalSlot(slot)`

Whether `this` features a slot `slot`, locally, not from some ancestor. See also `hasSlot`.

```

class Base      { var this.base = 23; } |;
class Derive: Base { var this.derive = 43 } |;
assert(Derive.hasLocalSlot("derive"));
assert(!Derive.hasLocalSlot("base"));

```

- `hasProperty(slotName, propName)`

Whether the slot `slotName` of `this` has a property `propName`.

```

const var halfPi = pi / 2|;
assert
{
  hasProperty("halfPi", "constant");
  !hasProperty("halfPi", "foobar");
};

```

- `hasSlot(slot)`

Whether `this` has the slot `slot`, locally, or from some ancestor. See also `hasLocalSlot`.

```

Derive.hasSlot("derive");
Derive.hasSlot("base");
!Base.hasSlot("derive");

```

- `'$id'`

- `inspect(deep = false)`

Describe `this`: its prototypes and slots, and their properties. If `deep`, all the slots are described, not only the local slots. See also `dump`.

```

do (2) { var this.attr = "foo"; this.attr->prop = "bar"}.inspect;
[00001227] *** Inspecting 2
[00001227] *** ** Prototypes:
[00001227] ***   0
[00001227] *** ** Local Slots:
[00001228] ***   attr : String

```



```
[00001228] ***      Properties:
[00001228] ***      prop : String = "bar"
```

- `isA(obj)`

Return true if `this` has `obj` in his parents, false otherwise.

```
Float.isA(Orderable);
!(String.isA(Float));
```

- `isNil`

Return true if `this` is `nil` (`??sec:std-nil`), false otherwise.

```
nil.isNil;
!(0.isNil);
```

- `isProto`

Return true if `this` is a prototype, false otherwise;

```
Float.isProto;
!(42.isProto);
```

- `isVoid`

Return true if `this` is void. See `void` (`??sec:std-void`).

```
void.isVoid;
! 42.isVoid;
```

- `localSlotNames`

Return a list with the names of the local slots of `this`, not including those of its ancestors.
See also `slotNames`.

```
var top = Object.new|;
var top.top1 = 1|;
var top.top2 = 2|;
var bot = top.new|;
var bot.bot1 = 10|;
var bot.bot2 = 20|;
assert
{
  top.localSlotNames == ["top1", "top2"];
  bot.localSlotNames == ["bot1", "bot2"];
};
```

- `locateSlot(slot)`

Return `nil` if `this` don't have the slot `slot`. Otherwise it returns the first lowest owner of `slot` of `this`.

```
locateSlot("init") == Channel;
locateSlot("doesNotExist").isNil;
```

- `print`
Send `print` to the `Channel.topLevel` channel.

```
1.print;
[00001228] 1
[1, "12"].print;
[00001228] [1, "12"]
```

- `protos`
Return the list of prototypes of `this`.

```
12.protos == [0];
```

- `properties(slotName)`
Return a dictionary of the properties of slot `slotName`. Raise an error if the slot does not exist.

```
2.properties("foo");
[00238495:error] !!! lookup failed: foo
do (2) { var foo = "foo" }.properties("foo");
[00238501] ["constant" => false]
do (2) { var foo = "foo" ; foo->bar = "bar" }.properties("foo");
[00238502] ["bar" => "bar", "constant" => false]
```

- `removeProperty(slotName, propName)`
Remove the property `propName` from the slot `slotName`. Raise an error if the slot does not exist, do nothing if the property does not exist.

```
do (2)
{
  var foo = "foo";
  foo->bar = "bar";
  removeProperty("foo", "bar");
}.properties("foo");
[00238502] ["constant" => false]

2.removeProperty("foo", "bar");
[00000072:error] !!! lookup failed: foo

do (2)
{
  var foo = "foo";
  removeProperty("foo", "bar");
}||;
```

- `removeProto(proto)`
Remove `proto` from the list of prototypes of `this`, and return `this`. Do nothing if `proto` is not a prototype of `this`.

```
do (Object.new)
{
  assert
  {
    addProto(Orderable);
    removeProto(123) === this;
    protos == [Orderable, Object];
    removeProto(Orderable) === this;
    protos == [Object];
  };
}
|;
```

- `removeSlot(slot)`

Remove *slot* from the (local) list of slots of *this*, and return *this*. Do nothing if *slot* does not exist.

```
{
  var base = Object.new;
  var base.slot = "base";

  var derive = Base.new;
  var derive.slot = "derive";

  do (derive)
  {
    assert
    {
      removeSlot("no such slot") === this;
      removeSlot("slot") === this;
      localSlotNames == [];
      base.slot == "base";
      removeSlot("slot") === this;
      base.slot == "base";
    };
  };
}
|;
```

- `setConstSlot`

Like `setSlot` but the created slot is const.

```
assert(setConstSlot("fortyTwo", 42) == 42);
fortyTwo = 51;
[00000000:error] !!! cannot modify const slot
```

- `setProperty(slotName, propName, value)`

Set the property *propName* of slot *slotName* to *value*. Raise an error in *slotName* does not exist. Return *value*. This is what *slotName*->*propName* = *value* actually performs.

```
do (Object.new)
{
```

```

var slot = "slot";
var value = "value";
assert
{
  setProperty("slot", "prop", value) === value;
  "prop" in properties("slot");
  getProperty("slot", "prop") === value;
  slot->prop === value;
  setProperty("slot", "noSuchProperty", value) === value;
};
}|;
setProperty("noSuchSlot", "prop", "12");
[00000081:error] !!! lookup failed: noSuchSlot

```

- **setProtos(*protos*)**

Set the list of prototypes of **this** to *protos*. Return void.

```

do (Object.new)
{
  assert
  {
    protos == [Object];
    setProtos([Orderable, Object]).isVoid;
    protos == [Orderable, Object];
  };
}|;

```

- **setSlot(*name*, *value*)**

Create a slot *name* mapping to *value*. Raise an error if *name* was already defined. This is what **var** *name* = *value* actually performs.

```

Object.setSlot("theObject", Object) === Object;
Object.theObject === Object;
theObject === Object;

```

If the current job is in redefinition mode, **setSlot** on an already defined slot is not an error and overwrites the slot like **updateSlot** would. See the **redefinitionMode** method in **System** ([??sec:std-System](#)).

- **slotNames**

Return a list with the slot names of **this** and its ancestors.

```

Object.localSlotNames
  .subset(Object.slotNames);
Object.protos.foldl(function (var r, var p) { r + p.localSlotNames },
  [])
  .subset(Object.slotNames);

```

- **type**

The name of the type of **this**. The **class** construct defines this slot to the name of the class ([Section 9.4](#)). This is used to display the name of “instances”.

```

class Example {};
[00000081] Example
assert
{
    Example.type == "Example";
};
Example.new;
[00000081] Example_0x6fb2720

```

- `uid`

Returns the unique id of `this`.

```

{
    var foo = Object.new;
    var bar = Object.new;
    assert
    {
        foo.uid == foo.uid;
        foo.uid != bar.uid;
    };
};

```

- `unacceptvoid`

Return `this`. See `void (??sec:std-void)` to know why.

```

{
    var o = Object.new|
    assert(o.unacceptVoid === o);
};

```

- `updateSlot(name, value)`

Map the existing slot named *name* to *value*. Raise an error if *name* was not defined.

```

Object.setSlot("one", 1)    == 1;
Object.updateSlot("one", 2) == 2;
Object.one                  == 2;

```

- `'&&'(that)`

Short-circuiting logical and. If `this` evaluates to true evaluate and return *that*, otherwise return `this` without evaluating *that*.

```

(0 && "foo") == 0;
(2 && "foo") == "foo";

("" && "foo") == "";
("foo" && "bar") == "bar";

```

- `'||'(that)`

Short-circuiting logical or. If `this` evaluates to false evaluate and return *that*, otherwise return `this` without evaluating *that*.

```
(0 || "foo") == "foo";
(2 || 1/0) == 2;

("" || "foo") == "foo";
("foo" || 1/0) == "foo";
```

- '!',

Logical negation. If `this` evaluates to false return `true` and vice-versa.

```
!1 == false;
!0 == true;

!"foo" == false;
!"" == true;
```

20.38 Orderable

Objects that have a concept of “less than”. See also [Comparable \(??sec:std-Comparable\)](#).

This object, made to serve as prototype, provides a definition of `<` based on `>`, and vice versa; and definition of `<=`/`>=` based on `</>==`. You **must** define either `<` or `>`, otherwise invoking either method will result in endless recursions.

```
class Foo : Orderable
{
  var value = 0;
  function init (v) { value = v; };
  function '<' (lhs) { value < lhs.value; };
  function asString() { "<" + value.asString + ">"; };
};
[00000000] <0>
var one = Foo.new(1);
[00000001] <1>
var two = Foo.new(2);
[00000002] <2>

assert( (one <= one) && (one <= two) && !(two <= one));
assert(!(one > one) && !(one > two) && (two > one));
assert( (one >= one) && !(one >= two) && (two >= one));
```

20.39 OutputStream

OutputStreams are used to write (possibly binary) files by hand.

20.39.1 Prototypes

- [Object \(??sec:std-Object\)](#)

20.39.2 Construction

An `OutputStream` is a writing-interface to a file; its constructor requires a `File` ([??sec:std-File](#)). If the file already exists, content is *appended* to it. Remove the file beforehand if you want to override its content.

```
System.system("(echo 1; echo 2) >file.txt")|;
OutputStream.new(File.new("file.txt"));
[00000001] OutputStream_0x827000

OutputStream.new(File.create("new.txt"));
[00000001] OutputStream_0x827000
```

20.39.3 Slots

- Output `this.asString`. Return `this` to enable chains of calls.

```
{
  {
    var o = OutputStream.new(File.create("fresh.txt"));
    o << 1 << "2" << [3, [4]];
  };
  File.new("fresh.txt").content.data;
}
==
"12[3, [4]]";
```

- `close`
Flush the buffers, and close the file.

```
OutputStream.new(File.create("file.txt")).close.isVoid;
```

- `putByte(byte)`
In order to private efficient input/output operations, *buffers* are used. As a consequence, what is put into a stream might not be immediately saved on the actual file. To *flush* a buffer means to dump its content to the file.

```
OutputStream.new(File.create("file.txt")).flush.isVoid;
```

20.40 Pair

A *pair* is a container storing two objects, similar in spirit to `std::pair` in C++.

20.40.1 Prototype

- `Tuple` ([??sec:std-Tuple](#))

20.40.2 Construction

A *Pair* is constructed with two arguments.

```
Pair.new(1, 2);
[00000001] (1, 2)

Pair.new;
[00000003:error] !!! Pair.init: expected 2 arguments, given 0

Pair.new(1, 2, 3, 4);
[00000003:error] !!! Pair.init: expected 2 arguments, given 4
```

20.40.3 Slots

- **first**
Return the first member of the pair.

```
Pair.new(1, 2).first == 1;
Pair[0] === Pair.first;
```

- **second**
Return the second member of the pair.

```
Pair.new(1, 2).second == 2;
Pair[1] === Pair.second;
```

20.41 Path

A *Path* points to a file system entity (directory, file and so forth).

20.41.1 Prototypes

- [Comparable](#) ([??sec:std-Comparable](#))
- [Orderable](#) ([??sec:std-Orderable](#))

20.41.2 Construction

A *Path* is constructed with the string that points to the file system entity. This path can be relative or absolute.

```
Path.new("/path/file.u");
[00000001] Path("/path/file.u")
```

Some minor simplifications are made, such as stripping useless ‘./’ occurrences.

```
Path.new("././////./foo/");
[00000002] Path("foo")
```


20.41.3 Slots

- `absolute`

Whether `this` is absolute.

```
Path.new("/abs/path").absolute;
!Path.new("rel/path").absolute;
```

- `asList`

List of names used in path (directories and possibly file), from bottom up. There is no difference between relative path and absolute path.

```
Path.new("/path/to/file.u").asList == ["path", "to", "file.u"];
Path.new("/path").asList == Path.new("path").asList;
```

- `asPrintable`

```
Path.new("file.txt").asPrintable == "Path(\"file.txt\")";
```

- `asString`

The name of the file.

```
Path.new("file.txt").asString == "file.txt";
```

- `basename`

Base name of the path.

```
Path.new("/absolute/path/file.u").basename == "file.u";
Path.new("relative/path/file.u").basename == "file.u";
```

- `cd`

Change current working directory to `this`. Return the new current working directory as a `Path`.

- `cwd`

The current working directory.

```
{
  // Save current directory.
  var pwd = Path.cwd|
  // Go into '/'.
  var root = Path.new("/").cd;
  // Current working directory is '/'.
  assert(Path.cwd == root);
  // Go back to the directory we were in.
  assert(pwd.cd == pwd);
};
```

- `dirname`

Directory name of the path.

```
Path.new("/abs/path/file.u").dirname == Path.new("/abs/path");
Path.new("rel/path/file.u").dirname == Path.new("rel/path");
```

- `exists`

Whether something (a file, a directory, ...) exists where `this` points to.

```
Path.cwd.exists;
Path.new("/").exists;
!Path.new("/this/path/does/not/exists").exists;
```

- `isDir`

Whether `this` is a directory.

```
Path.cwd.isDir;
```

- `isReg`

Whether `this` is a regular file.

```
!Path.cwd.isReg;
```

- `open`

Open `this`. Return either a *Directory* or a *File* according the type of `this`. See [File \(??sec:std-File\)](#) and [Directory \(??sec:std-Directory\)](#).

- `readable`

Whether `this` is readable. Throw if does not even exist.

```
Path.new(".").readable;
```

- `writable`

Whether `this` is writable. Throw if does not even exist.

```
Path.new(".").writable;
```

- `'/' (rhs)`

Create a new *Path* that is the concatenation of `this` and *rhs*. *rhs* can be a *Path* or a *String* and cannot be absolute.

```
assert(Path.new("/foo/bar") / Path.new("baz/qux/quux")
      == Path.new("/foo/bar/baz/qux/quux"));
Path.cwd / Path.new("/tmp/foo");
[00000003:error] !!! /: Rhs of concatenation is absolute: /tmp/foo
```

- '==' (*that*)

Same as comparing the string versions of *this* and *that*. Beware that two paths may be different and point to the very same location.

```
Path.new("/a") == Path.new("/a");
!(Path.new("/a") == Path.new("a") );
```

- '<' (*that*)

Same as comparing the string versions of *this* and *that*.

```
Path.new("/a") < Path.new("/a/b");
!(Path.new("/a/b") < Path.new("/a") );
```

20.42 Pattern

Pattern class is used to make correspondences between a pattern and another **Object**. The visit is done either on the pattern or on the element against which the pattern is compared.

Patterns are used for the implementation of the pattern matching. So any class made compatible with the pattern matching implemented by this class will allow you to use it implicitly in your scripts.

```
[1, var a, var b] = [1, 2, 3];
[00000000] [1, 2, 3]
a;
[00000000] 2
b;
[00000000] 3
```

20.42.1 Prototypes

- **Object** (??sec:std-Object)

20.42.2 Construction

A **Pattern** can be created with any object that can be matched.

```
Pattern.new([1]); // create a pattern to match the list [1].
[00000000] Pattern_0x189ea80
Pattern.new(Pattern.Binding.new("a")); // match anything into "a".
[00000000] Pattern_0x18d98b0
```

20.42.3 Slots

- **Binding**

A class used to create pattern variables.

```
Pattern.Binding.new("a");
[00000000] var a
```

- **bindings**

A Dictionary filled by the match function for each **Binding** contained inside the pattern.

```
{
  var p = Pattern.new([Pattern.Binding.new("a"), Pattern.Binding.new("b")]);
  assert (p.match([1, 2]));
  p.bindings
};
[00000000] ["a" => 1, "b" => 2]
```

- **match(*value*)**

Use *value* to unify the current pattern with this value. Return the status of the match.

- If the match is correct, then the *bindings* member will contain the result of every matched values.
- If the match is incorrect, then the *bindings* member should not be used.

If the pattern contains multiple **Binding** with the same name, then the behavior is undefined.

```
Pattern.new(1).match(1);
Pattern.new([1, 2]).match([1, 2]);
! Pattern.new([1, 2]).match([1, 3]);
! Pattern.new([1, 2]).match([1, 2, 3]);
Pattern.new(Pattern.Binding.new("a")).match(0);
Pattern.new([1, Pattern.Binding.new("a")]).match([1, 2]);
! Pattern.new([1, Pattern.Binding.new("a")]).match(0);
```

- **matchPattern(*pattern*, *value*)**

This function is used as a callback function to store all bindings in the same place. This function is useful inside objects that implement a **match** or **matchAgainst** function that need to continue the match deeper. Return the status of the match (a Boolean).

The *pattern* should provide a method **match(*handler*, *value*)** otherwise the value method **matchAgainst(*handler*, *pattern*)** is used. If none are provided the '==' operator is used.

To see how to use it, you can have a look at the implementation of **List.matchAgainst**.

- **pattern**

The pattern given at the creation.

```

Pattern.new(1).pattern == 1;
Pattern.new([1, 2]).pattern == [1, 2];
{
  var pattern = [1, Pattern.Binding.new("a")];
  Pattern.new(pattern).pattern === pattern
};

```

20.43 Position

This class is used to handle file locations with a line, column and file name.

20.43.1 Prototypes

- `Object` (??sec:std-Object)

20.43.2 Construction

Without argument, a newly constructed Position has its fields initialized to the first line and the first column.

```

Position.new;
[00000001] 1.1

```

With a position argument p , the newly constructed Position is a clone of p .

```

Position.new(Position.new(2, 3));
[00000001] 2.3

```

With two float arguments l and c , the newly constructed Position has its line and column defined and an empty file name.

```

Position.new(2, 3);
[00000001] 2.3

```

With three arguments f , l and c , the newly constructed Position has its file name, line and column defined.

```

Position.new("file.u", 2, 3);
[00000001] file.u:2.3

```

20.43.3 Slots

- `'+'(n)`

Return a new Position which is shifted from n columns to the right. The minimal value of the new position column is 1.

```

Position.new(2, 3) + 2 == Position.new(2, 5);
Position.new(2, 3) + -4 == Position.new(2, 1);

```

- `'-'(n)`

Return a new Position which is shifted from *n* columns to the left. The minimal value of the new Position column is 1.

```
Position.new(2, 3) - 1 == Position.new(2, 2);
Position.new(2, 3) - -4 == Position.new(2, 7);
```

- `'=='(other)`

Compare the lines and columns of two Positions.

```
Position.new(2, 3) == Position.new(2, 3);
Position.new("a.u", 2, 3) == Position.new("b.u", 2, 3);
Position.new(2, 3) != Position.new(2, 2);
```

- `'<'(other)`

Order comparison of lines and columns.

```
Position.new(2, 3) < Position.new(2, 4);
Position.new(2, 3) < Position.new(3, 1);
```

- `asString`

Present as `'file:line.column'`, the file name is omitted if it is not defined.

```
Position.new("file.u", 2, 3);
[00000001] file.u:2.3
```

- `column`

Field which give access to the column number of the Position.

```
Position.new(2, 3).column == 3;
```

- `columns(n)`

Identical to `'+'(n)`.

```
Position.new(2, 3).columns(2) == Position.new(2, 5);
Position.new(2, 3).columns(-4) == Position.new(2, 1);
```

- `file`

The Path of the Position file.

```
Position.new("file.u", 2, 3).file == Path.new("file.u");
Position.new(2, 3).file == nil;
```

- `line`

Field which give access to the line number of the Position.

```
Position.new(2, 3).line == 2;
```

- `lines(n)`
Add *n* lines and reset the column number to 1.

```
Position.new(2, 3).lines(2) == Position.new(4, 1);
Position.new(2, 3).lines(-1) == Position.new(1, 1);
```

20.44 Primitive

C++ routine callable from urbiscript.

20.44.1 Prototypes

- [Executable](#) ([??sec:std-Executable](#))

20.44.2 Construction

It is not possible to construct a Primitive.

20.44.3 Slots

- `apply(args)`
Invoke a primitive. The argument list, *args*, must start with the target.

```
Float.getSlot("+").isA(Global.getSlot("Primitive"));
Float.getSlot("+").apply([1, 2]) == 3;

String.getSlot("+").isA(Global.getSlot("Primitive"));
String.getSlot("+").apply(["1", "2"]);
```

- `asPrimitive`
Return `this`.

```
Float.getSlot("+").asPrimitive === Float.getSlot("+");
```

20.45 Process

A Process is a separated task handled by the underneath operating system.

Windows Issues

Process is not yet supported under Windows.

20.45.1 Prototypes

- [Object](#) ([??sec:std-Object](#))

20.45.2 Example

The following examples runs the `cat` program, a Unix standard command that simply copies on its (standard) output its (standard) input.

```
var p = Process.new("cat", []);  
[00000004] Process cat
```

Just created, this process is not running yet. Use `run` to launch it.

```
p.status;  
[00000005] not started  
  
p.run;  
p.status;  
[00000006] running
```

Then we feed its input, named `stdin` in the Unix tradition, and close its input.

```
p.stdin << "1\n" |  
p.stdin << "2\n" |  
p.stdin << "3\n" |;  
  
p.status;  
[00000007] running  
  
p.stdin.close;
```

At this stage, the status of the process is unknown, as it is running asynchronously. If it has had enough time to “see” that its input is closed, then it will have finished, otherwise we might have to wait for awhile. The method `join` means “wait for the process to finish”.

```
p.join;  
  
p.status;  
[00000008] exited with status 0
```

Finally we can check its output.

```
p.stdout.asList;  
[00000009] ["1", "2", "3"]
```

20.45.3 Construction

A `Process` needs a program name to run and a possibly-empty list of command line arguments. Calling `run` is required to execute the process.

```
Process.new("cat", []);  
[00000004] Process cat  
  
Process.new("cat", ["--version"]);  
[00000004] Process cat
```


20.45.4 Slots

- `asProcess`
Return `this`.

```
do (Process.new("cat", []))
{
  assert (asProcess == this);
};
```

- `asString`
Process and the name of the program.

```
Process.new("cat", ["--version"]).asString
== "Process cat";
```

- `done`
Whether the process has completed its execution.

```
do (Process.new("sleep", ["1"]))
{
  assert (!done);
  run;
  assert (!done);
  join;
  assert (done);
};
```

- `join`
Wait for the process to finish. Changes its status.

```
do (Process.new("sleep", ["2"]))
{
  var t0 = System.time;
  assert (status.asString == "not started");
  run;
  assert (status.asString == "running");
  join;
  assert (t0 + 2s <= System.time);
  assert (status.asString == "exited with status 0");
};
```

- `kill`
If the process is not `done`, interrupt it (with a `SIGKILL` in Unix parlance). You still have to wait for its termination with `join`.

```
do (Process.new("sleep", ["1"]))
{
  run;
  kill;
  join;
```

```

    assert (done);
    assert (status.asString == "killed by signal 9");
  }|;

```

- **name**

The (base) name of the program the process runs.

```

Process.new("cat", ["--version"]).name == "cat";

```

- **run**

Launch the process. Changes it status. A process can only be run once.

```

do (Process.new("sleep", ["1"]))
{
  assert (status.asString == "not started");
  run;
  assert (status.asString == "running");
  join;
  assert (status.asString == "exited with status 0");
  run;
}|;
[00021972:error] !!! run: Process was already run

```

- **runTo**

- **status**

An object whose slots describe the status of the process.

- **stderr**

An [InputStream](#) ([??sec:std-InputStream](#)) (the output of the Process is an input for Urbi) to the standard error stream of the process.

```

do (Process.new(System.programName, ["--", "--no-such-option"]))
{
  run;
  join;
  assert
  {
    stderr.asList ==
    [System.programName + ": invalid option: --no-such-option",
     "Try '" + System.programName + " --help' for more information."];
  };
}|;

```

- **stdin**

An [OutputStream](#) ([??sec:std-OutputStream](#)) (the input of the Process is an output for Urbi) to the standard input stream of the process.

```
do (Process.new(System.programName, ["--version"]))
{
  run;
  join;
  assert
  {
    stdout.asList[1] == "Copyright (C) 2004-2010 Gostai S.A.S..";
  };
} |;
```

- **stdout**

An [InputStream](#) ([??sec:std-InputStream](#)) (the output of the Process is an input for Urbi) to the standard output stream of the process.

```
do (Process.new("cat", []))
{
  run;
  stdin << "Hello, World!\n";
  stdin.close;
  join;
  assert (stdout.asList == ["Hello, World!"]);
} |;
```

20.46 Profiling

Profiling is useful to get an idea of the efficiency of some small pieces of code.

20.46.1 Prototypes

- [Object](#) ([??sec:std-Object](#))

20.46.2 Construction

A **Profiling** can be created with two arguments. The first argument is the expression which has to be profiled and the second is the number of iteration it should be run.

Creating a **Profiling** session prints the result of the profiled expression, the number of iterations, the number of cycles and the time of the evaluation. The number of cycles corresponds to the number of time the job is scheduled.

```
Profiling.new({1| 2| 3| 4}, 10000);
[00000000] Profiling information
Expression:      1 | 2 | 3 | 4
Iterations:      10000
Cycles:          10000
Total time:      1.00098 s
Single iteration: 0.000100098 s
                  1 cycles
```

```

Profiling.new({1; 2; 3; 4}, 10000);
[00000000] Profiling information
  Expression:      1;
2;
3;
4
  Iterations:      10000
  Cycles:          40000
  Total time:      1.45856 s
  Single iteration: 0.000145856 s
                  4 cycles

```

20.47 PseudoLazy

20.48 PubSub

PubSub provides an abstraction over `Barrier` [Barrier](#) ([??sec:std-Barrier](#)) to queue signals for each subscriber.

20.48.1 Prototypes

- [Object](#) ([??sec:std-Object](#))

20.48.2 Construction

A PubSub can be created with no arguments. Values can be published and read by each subscriber.

```

var ps = PubSub.new;
[00000000] PubSub_0x28c1bc0

```

20.48.3 Slots

- `publish(ev)`
Queue the value *ev* to the queue of each subscriber. This method returns the value *ev*.

```

{
  var sub = ps.subscribe;
  assert
  {
    ps.publish(2) == 2;
    sub.getOne == 2;
  };
  ps.unsubscribe(sub)
}|;

```

- **subscribe**

Create a [Subscriber](#) and insert it inside the list of subscribers.

```
var sub = ps.subscribe |
ps.subscribers == [sub];
[00000000] true
```

- **Subscriber**

See [PubSub.Subscriber](#) ([??sec:std-PubSub.Subscriber](#)).

- **subscribers**

Field containing the list of [Subscriber](#) which are watching published values. This field only exists in instances of PubSub.

- **unsubscribe(sub)**

Remove a subscriber from the list of subscriber watching the published values.

```
ps.unsubscribe(sub) |
ps.subscribers;
[00000000] []
```

20.49 PubSub.Subscriber

Subscriber is created by `PubSub.subscribe`. It provides methods to access to the list of values published by PubSub instances.

20.49.1 Prototypes

- [Object](#) ([??sec:std-Object](#))

20.49.2 Construction

A `PubSub.Subscriber` can be created with a call to `PubSub.subscribe`. This way of creating a Subscriber adds the subscriber as a watcher of values published on the instance of PubSub.

```
var ps = PubSub.new |;
var sub = ps.subscribe;
[00000000] Subscriber_0x28607c0
```

20.49.3 Slots

- **getOne**

Block until a value is accessible and return it. If a value is already queued, then the method returns it without blocking.

```
echo(sub.getOne) &
ps.publish(3);
[00000000] *** 3
```

- `getAll`
Block until a value is accessible. Return the list of queued values. If the values are already queued, then return them without blocking.

```
ps.publish(4) |
ps.publish(5) |
echo(sub.getAll);
[00000000] *** [4, 5]
```

20.50 RangeIterable

This object is meant to be used as a prototype for objects that support an `asList` method, to use range-based `for` loops (Section 19.6.5.2).

20.50.1 Prototypes

- `Object` (`??sec:std-Object`)

20.50.2 Slots

- `all(fun)`
Return whether all the members of the target verify the predicate *fun*.

```
// Are all elements positive?
! [-2, 0, 2, 4].all(function (e) { e > 0 });
// Are all elements even?
[-2, 0, 2, 4].all(function (e) { e % 2 == 0 });
```

- `any(fun)`
Whether at least one of the members of the target verifies the predicate *fun*.

```
// Is there any even element?
! [-3, 1, -1].any(function (e) { e % 2 == 0 });
// Is there any positive element?
[-3, 1, -1].any(function (e) { e > 0 });
```

- `each(fun)`
Apply the given functional value *fun* on all “members”, sequentially. Corresponds to range-`for` loops.

```
class range : RangeIterable
{
    var asList = [10, 20, 30];
} |;
for (var i : range)
    echo (i);
[00000000] *** 10
[00000000] *** 20
[00000000] *** 30
```

- `'each&'(fun)`

Apply the given functional value *fun* on all “members”, in parallel, starting all the computations simultaneously. Corresponds to range-`for&` loops.

```
{
  var res = [];
  for& (var i : range)
    res << i;
  assert(res.sort == [10, 20, 30]);
};
```

- `'each|'(fun)`

Apply the given functional value *fun* on all “members”, with tight sequentially. Corresponds to range-`for|` loops.

```
{
  var res = [];
  for| (var i : range)
    res << i;
  assert(res == [10, 20, 30]);
};
```

20.51 Regexp

A `Regexp` is an object which allow you to match strings with a regular expression.

20.51.1 Prototypes

- `Container` (`??sec:std-Container`)
- `Object` (`??sec:std-Object`)

20.51.2 Construction

A `Regexp` is created with the regular expression once and for all, and it can be used many times to match with other strings.

```
Regexp.new(".");
[00000001] Regexp(". ")
```

urbiscript supports Perl regular expressions, see [the perlre man page](#). Expressions cannot be empty.

20.51.3 Slots

- `asPrintable`

A string that shows that `this` is a `Regexp`, and its value.

```

Regexp.new("abc").asPrintable == "Regexp(\"abc\")";
Regexp.new("\\d+(\\.\\d+)?").asPrintable == "Regexp(\"\\\\d+(\\\\\\.\\\\d+)?\")";

```

- `asString`

The regular expression that was compiled.

```

Regexp.new("abc").asString == "abc";
Regexp.new("\\d+(\\.\\d+)?").asString == "\\d+(\\.\\d+)?";

```

- `has(str)`

An experimental alias to `match`, so that the infix operators `in` and `not in` can be used (see [Section 19.1.8.6](#)).

```

"23.03" in Regexp.new("^\\d+\\.\\d+$");
"-3.14" not in Regexp.new("^\\d+\\.\\d+$");

```

- `match(str)`

Whether `this` matches `str`.

```

// Ordinary characters
var r = Regexp.new("oo")|
assert
{
  r.match("oo");
  r.match("foobar");
  !r.match("bazquux");
};

// ^, anchoring at the beginning of line.
r = Regexp.new("^oo")|
assert
{
  r.match("oops");
  !r.match("woot");
};

// $, anchoring at the end of line.
r = Regexp.new("oo$")|
assert
{
  r.match("foo");
  !r.match("mooh");
};

// *, greedy repetition, 0 or more.
r = Regexp.new("fo*bar")|
assert
{
  r.match("fbar");
  r.match("fooooooobar");
};

```



```

    !r.match("far");
  };

  // (), grouping.
  r = Regexp.new("f(oo)*bar")|
  assert
  {
    r.match("foooobar");
    !r.match("foobar");
  };

```

20.52 StackFrame

This class is meant to record backtrace (see [Exception.backtrace](#)) information.

For convenience, all snippets of code are supposed to be run after these function definitions. In this code, the `getStackFrame` function is used to get the first `StackFrame` of an exception backtrace. Backtrace of [Exception \(??sec:std-Exception\)](#) are filled with `StackFrames` when the is thrown.

```

// #push 1 "foo.u"
function inner () { throw Exception.new("test") }|;

function getStackFrame()
{
  try
  {
    inner
  }
  catch(var e)
  {
    e.backtrace[0]
  };
}|;
// pop

```

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

20.52.1 Construction

`StackFrame` are not made to be manually constructed. The initialization function expect 2 arguments, which are the name of the called function and the [Location \(??sec:std-Location\)](#) from which it has been called.

```

StackFrame.new("inner",
  Location.new(
    Position.new("foo.u", 7, 5),
    Position.new("foo.u", 7, 10)
  )
)

```

```
);
[00000001] foo.u:7.5-9: inner
```

20.52.2 Slots

- **name**
String (??sec:std-String), representing the name of the called function.

```
getStackFrame.name;
[00000002] "inner"
```

- **location**
Location (??sec:std-Location) of the function call.

```
getStackFrame.location;
[00000003] foo.u:7.5-9
```

- **asString**
Clean display of the call location.

```
getStackFrame;
[00000004] foo.u:7.5-9: inner
```

20.53 Semaphore

Semaphore are useful to limit the number of access to a limited number of resources.

20.53.1 Prototypes

- **Object** (??sec:std-Object)

20.53.2 Construction

A **Semaphore** can be created with as argument the number of processes allowed to enter critical sections at the same time.

```
Semaphore.new(1);
[00000000] Semaphore_0x8c1e80
```

20.53.3 Slots

- **criticalSection**(**function** ()
;code;) Put the piece of *code* inside a critical section which can be executed simultaneously at most the number of time given at the creation of the **Semaphore**. This method is similar to a call to **acquire** and a call to **release** when the code ends by any means.

```

{
  var s = Semaphore.new(1);
  for& (var i : [0, 1, 2, 3])
  {
    s.criticalSection(function () {
      echo("start " + i);
      echo("end " + i);
    })
  }
};
[00000000] *** start 0
[00000000] *** end 0
[00000000] *** start 1
[00000000] *** end 1
[00000000] *** start 2
[00000000] *** end 2
[00000000] *** start 3
[00000000] *** end 3

{
  var s = Semaphore.new(2);
  for& (var i : [0, 1, 2, 3])
  {
    s.criticalSection(function () {
      echo("start " + i);

      // Illustrate that processes can be intertwined
      sleep(i * 100ms);

      echo("end " + i);
    })
  }
};
[00000000] *** start 0
[00000000] *** start 1
[00000000] *** end 0
[00000000] *** start 2
[00000000] *** end 1
[00000000] *** start 3
[00000000] *** end 2
[00000000] *** end 3

```

- **acquire**
Wait to enter a critical section delimited by the execution of **acquire** and **release**. Enter the critical section when the number of processes inside it goes below the maximum allowed.
- **p**
Historical synonym for **acquire**.
- **release**

Leave a critical section delimited by the execution of [acquire](#) and [release](#).

```
{
  var s = Semaphore.new(1);
  for& (var i : [0, 1, 2, 3])
  {
    s.acquire;
    echo("start " + i);
    echo("end " + i);
    s.release;
  }
};
[00000000] *** start 0
[00000000] *** end 0
[00000000] *** start 1
[00000000] *** end 1
[00000000] *** start 2
[00000000] *** end 2
[00000000] *** start 3
[00000000] *** end 3
```

- **v**
Historical synonym for [release](#).

20.54 Server

A *Server* can listen to incoming connections. See [Socket](#) ([??sec:std-Socket](#)) for an example.

20.54.1 Prototypes

- [Object](#) ([??sec:std-Object](#))

20.54.2 Construction

A **Server** is constructed with no argument. At creation, a new **Server** has its own slot **connection**. This slot is an event that is launched when a connection establishes.

```
var s = Server.new|
s.localSlotNames;
[00000001] ["connection"]
```

20.54.3 Slots

- **connection**
The event launched at each incoming connection. This event is launched with one argument: the socket of the established connection. This connection uses the same [IoService](#) ([??sec:std-IoService](#)) as the server.

```

at (s.connection?(var socket))
{
  // This code is run at each connection. 'socket' is the incoming
  // connection.
};

```

- `getIoService`
Return the `IoService` (`??sec:std-IoService`) used by this socket. Only the default `IoService` is automatically polled.

- `host`
The host on which `this` is listening. Raise an error if `this` is not listening.

```

Server.host;
[00000003:error] !!! host: server not listening

```

- `listen(host, port)`
Listen incoming connections with `host` and `port`.

- `port`
The port on which `this` is listening. Raise an error if `this` is not listening.

```

Server.port;
[00000004:error] !!! port: server not listening

```

- `sockets`
The list of the sockets created at each incoming connection.

20.55 Singleton

A *singleton* is a prototype that cannot be cloned. All prototypes derived of `Singleton` are also singletons.

20.55.1 Prototypes

- `Object` (`??sec:std-Object`)

20.55.2 Construction

To be a singleton, the object must have `Singleton` as a prototype. The common way to do this is `var s = Singleton.new`, but this does not work : `s` is not a new singleton, it is the `Singleton` itself since it cannot be cloned. There are two other ways:

```

// Defining a new class and specifying Singleton as a parent.
class NewSingleton1: Singleton
{
  var asString = "NewSingleton1";
}

```

```

}|
var s1 = NewSingleton1.new;
[00000001] NewSingleton1
assert(s1 === NewSingleton1);
assert(NewSingleton1 !== Singleton);

// Create a new Object and set its prototype by hand.
var NewSingleton2 = Object.new|
var NewSingleton2.asString = "NewSingleton2"|
NewSingleton2.protos = [Singleton]|
var s2 = NewSingleton2.new;
[00000001] NewSingleton2
assert(s2 === NewSingleton2);
assert(NewSingleton2 !== Singleton);

```

20.55.3 Slots

- clone
Return `this`.
- 'new'
Return `this`.

20.56 Socket

A *Socket* can manage asynchronous input/output network connections.

20.56.1 Example

The following example demonstrates how both the `Server` (`??sec:std-Server`) and `Socket` (`??sec:std-Socket`) object work.

This simple example will establish a dialog between `server` and `client`. The following object, `Dialog`, contains the script of this exchange. It is put into `Global` so that both the server and client can read it. `Dialog.reply(var s)` returns the reply to a message `s`.

```

class Global.Dialog
{
  var lines =
  [
    "Hi!",
    "Hey!",
    "Hey you doin'?",
    "Whazaaa!",
    "See ya.",
  ];

  function reply(var s)
  {
    for (var i: lines.size - 1)

```

```

        if (s == lines[i])
            return lines[i + 1];
        "off";
    }
}!;

```

The server, an instance of `Server` (`??sec:std-Server`), expects incoming connections, notified by the socket's `connection?` event. Once the connection establish, it listens to the `socket` for incoming messages, notified by the `received?` event. Its reaction to this event is to send the following line of the dialog. At the end of the dialog, the socket is disconnected.

```

var server =
do (Server.new)
{
    at (connection?(var socket))
        at (socket.received?(var data))
        {
            var reply = Dialog.reply(data);
            socket.write(reply);
            echo("server: " + reply);
            if (reply == "off")
                socket.disconnect;
        }
};
}!;

```

The client, an instance of `Socket` (`??sec:std-Socket`) expects incoming messages, notified by the `received?` event. Its reaction is to send the following line of the dialog.

```

var client =
do (Socket.new)
{
    at (received?(var data))
    {
        var reply = Dialog.reply(data);
        write(reply);
        echo("client: " + reply);
    }
};
}!;

```

As of today, urbiscript's socket machinery requires to be regularly polled.

```

every (100ms)
    Socket.poll,

```

The server is then activated, listening to incoming connections on a port that will be chosen by the system amongst the free ones.

```

server.listen("localhost", "0");
clog << "connecting to %s:%s" % [server.host, server.port];

```

The client connects to the server, and initiates the dialog.

```

client.connect(server.host, server.port);
echo("client: " + Dialog.lines[0]);

```

```
client.write(Dialog.lines[0]);
[00000003] *** client: Hi!
```

Because this dialog is asynchronous, the easiest way to wait for the dialog to finish is to wait for the `disconnected?` event.

```
waituntil(client.disconnected?);
[00000004] *** server: Hey!
[00000005] *** client: Hey you doin'?
[00000006] *** server: Whazaaa!
[00000007] *** client: See ya.
[00000008] *** server: off
```

20.56.2 Prototypes

- `Object (??sec:std-Object)`

20.56.3 Construction

A `Socket` is constructed with no argument. At creation, a new `Socket` has four own slots: `connected`, `disconnected`, `error` and `received`.

```
var s = Socket.new|
```

20.56.4 Slots

- `connect(host, port)`
Connect `this` to `host` and `port`. The `port` can be either an integer, or a string that denotes symbolic ports, such as `"smtp"`, or `"ftp"` and so forth.
- `connected`
Event launched when the connection is established.
- `connectSerial(device, baudRate)`
Connect `this` to the serial port `device`, with given `baudRate`.
- `disconnect`
Close the connection.
- `disconnected`
Event launched when a disconnection happens.
- `error`
Event launched when an error happens. This event is launched with the error message in argument. The event `disconnected` is also always launched.
- `getIoService`
Return the `IoService (??sec:std-IoService)` used by this socket. Only the default `IoService` is automatically polled.

- `host`
The remote host of the connection.
- `isConnected`
Whether `this` is connected.
- `localhost`
The local host of the connection.
- `localPort`
The local port of the connection.
- `poll`
Call `getIoService.poll()`. This method is called regularly every `pollInterval` on the `Socket` object. You do not need to call this function on your sockets unless you use your own `IoService` ([??sec:std-IoService](#)).
- `pollInterval`
Each `pollInterval` amount of time, `poll` is called. If `pollInterval` equals zero, `poll` is not called.
- `port`
The remote port of the connection.
- `received`
Event launched when `this` has received data. The data is given by argument to the event.
- `write(data)`
Sends *data* through the connection.
- `syncWrite(data)`
Similar to `write`, but forces the operation to complete synchronously. Synchronous and asynchronous write operations cannot be mixed.

20.57 String

A *string* is a sequence of characters.

20.57.1 Prototypes

- [Comparable](#) ([??sec:std-Comparable](#))
- [Orderable](#) ([??sec:std-Orderable](#))
- [RangeIterable](#) ([??sec:std-RangeIterable](#))

20.57.2 Construction

Fresh Strings can easily be built using the literal syntax. Several escaping sequences (the traditional ones and urbiscript specific ones) allow to insert special characters. Consecutive string literals are merged together. See [Section 19.1.6.6](#) for details and examples.

A null String can also be obtained with `String`'s `new` method.

```
String.new == "";
String == "";
"123".new == "123";
```

20.57.3 Slots

- `asFloat`

If the whole content of `this` is an integer, return its value, otherwise return an error.

```
assert("23.03".asFloat == 23.03);
"123abc".asFloat;
[00000001:error] !!! asFloat: unable to convert to float: "123abc"
```

- `asList`

Return a List of one-letter Strings that, concatenated, equal `this`. This allows to use `for` to iterate over the string.

```
assert("123".asList == ["1", "2", "3"]);
for (var v : "123")
  echo(v);
[00000001] *** 1
[00000001] *** 2
[00000001] *** 3
```

- `asPrintable`

Return `this` as a literal (escaped) string.

```
"foo".asPrintable == "\"foo\"";
"foo".asPrintable.asPrintable == "\"\\\"foo\\\"\"";
```

- `asString`

Return `this`.

```
"\"foo\"".asString == "\"foo\"";
```

- `closest(set)`

Return the string in `set` that is the closest (in the sense of [distance](#)) to `this`. If there is no convincing match, return `nil`.

```
"foo".closest(["foo", "baz", "qux", "quux"]) == "foo";
"bar".closest(["foo", "baz", "qux", "quux"]) == "baz";
"F00".closest(["foo", "bar", "baz"])         == "foo";
"qux".closest(["foo", "bar", "baz"])          == nil;
```

- `distance(other)`

Return the [Damerau-Levenshtein distance](#) between `this` and `other`. The more alike the strings are, the smaller the distance is.

```
"foo".distance("foo") == 0;
"bar".distance("baz") == 1;
"foo".distance("bar") == 3;
```

- `fresh`

Return a String that has never been used as an identifier, prefixed by `this`. It can safely be used with `Object.setSlot` and so forth.

```
String.fresh == "_5";
"foo".fresh == "foo_6";
```

- Character handling functions

Here is a map of how the original 127-character ASCII set is considered by each function (a • indicates that the function returns true if all characters of `this` are on the row).

ASCII values	Characters	isctrl	isspace	isupper	islower	isalpha	isdigit	isxdigit	isalnum	ispunct	isgraph	print
0x00 .. 0x08		•										
0x09 .. 0x0D	\t, \f, \v, \n, \r	•	•									
0x0E .. 0x1F		•										
0x20	space (')		•									•
0x21 .. 0x2F	!"#\$%&'()*+,-./									•	•	•
0x30 .. 0x39	0-9						•	•	•		•	•
0x3a .. 0x40	;<=>?@									•	•	•
0x41 .. 0x46	A-F			•		•		•	•		•	•
0x47 .. 0x5A	G-Z			•		•			•		•	•
0x5B .. 0x60	[\] ^ { } _ ' `									•	•	•
0x61 .. 0x66	a-f				•	•		•	•		•	•
0x67 .. 0x7A	g-z				•	•			•		•	•
0x7B .. 0x7E	{ } ~									•	•	•
0x7F	(DEL)	•										

```
"".isDigit;
"0123456789".isDigit;
!"a".isDigit;
```

```
"".isLower;
"lower".isLower;
! "Not Lower".isLower;

"".isUpper;
"UPPER".isUpper;
! "Not Upper".isUpper;
```

- `join(list, prefix, suffix)`

Glue the result of `asString` applied to the members of *list*, separated by *this*, and embedded in a pair *prefix/suffix*.

```
"|".join([1, 2, 3], "(", ")") == "(1|2|3)";
", ".join([1, [2], "3"], "[", "]") == "[1, [2], 3]";
```

- `replace(from, to)`

Replace every occurrence of the string *from* in *this* by *to*, and return the result. *this* is not modified.

```
"Hello, World!".replace("Hello", "Bonjour")
    .replace("World!", "Monde !") ==
    "Bonjour, Monde !";
```

- `size`

Return the size of the string.

```
"foo".size == 3;
"".size == 0;
```

- `split(sep = [" ", "\t", "\n", "\r"], lim = -1, keepSep = false, keepEmpty = true)`

Split *this* on the separator *sep*, in at most *lim* components, which include the separator if *keepSep*, and the empty components of *keepEmpty*. Return a list of strings.

The separator, *sep*, can be a string.

```
"a,b;c".split(",") == ["a", "b;c"];
"a,b;c".split(";") == ["a,b", "c"];
"foobar".split("x") == ["foobar"];
"foobar".split("ob") == ["fo", "ar"];
```

It can also be a list of strings.

```
"a,b;c".split(["", ";"]) == ["a", "b", "c"];
```

By default splitting is performed on white-spaces:

```
" abc def\tghi\n".split == ["abc", "def", "ghi"];
```

Splitting on the empty string stands for splitting between each character:

```
"foobar".split("") == ["f", "o", "o", "b", "a", "r"];
```

The limit *lim* indicates a maximum number of splits that can occur. A negative number corresponds to no limit:

```
"a:b:c".split(":", 1) == ["a", "b:c"];
"a:b:c".split(":", -1) == ["a", "b", "c"];
```

keepSep indicates whether to keep delimiters in the result:

```
"aaa:bbb;ccc".split([":", ";"], -1, false) == ["aaa", "bbb", "ccc"];
"aaa:bbb;ccc".split([":", ";"], -1, true) == ["aaa", ":", "bbb", ";", "ccc"];
```

keepEmpty indicates whether to keep empty elements:

```
"foobar".split("o") == ["f", "", "bar"];
"foobar".split("o", -1, false, true) == ["f", "", "bar"];
"foobar".split("o", -1, false, false) == ["f", "bar"];
```

- **toLowerCase**

Make lower case every upper case character in *this* and return the result. *this* is not modified.

```
"Hello, World!".toLowerCase == "hello, world!";
```

- **toUpperCase**

Make upper case every lower case character in *this* and return the result. *this* is not modified.

```
"Hello, World!".toUpperCase == "HELLO, WORLD!";
```

- **'==' (that)**

Whether *this* and *that* are the same string.

```
" " == " ";      !(" " != " ");
!(" " == "\0");  " " != "\0";

"0" == "0";      !("0" != "0");
!("0" == "1");   "0" != "1";
!("1" == "0");   "1" != "0";
```

- **'%' (args)**

It is an equivalent of `Formatter.new(this) % args`. See [Formatter \(??sec:std-Formatter\)](#).

```
"%s + %s = %s" % [1, 2, 3] == "1 + 2 = 3";
```

- **'*' (n)**

Concatenate *this* *n* times.

```
"foo" * 0 == "";
"foo" * 1 == "foo";
"foo" * 3 == "foofoofoo";
```

- `'+'(other)`

Concatenate `this` and `other.asString`.

```
"foo" + "bar" == "foobar";
"foo" + "" == "foo";
"foo" + 3 == "foo3";
"foo" + [1, 2, 3] == "foo[1, 2, 3]";
```

- `'<'(other)`

Whether `this` is lexicographically before `other`, which must be a String.

```
"" < "a";
!("a" < "");
"a" < "b";
!("a" < "a");
```

- `'[]'(from)`

`'[]'(from, to)`

Return the sub-string starting at `from`, up to and not including `to` (which defaults to `to + 1`).

```
"foobar"[0, 3] == "foo";
"foobar"[0] == "f";
```

- `'[]='(from, other)`

`'[]='(from, to, other)`

Replace the sub-string starting at `from`, up to and not including `to` (which defaults to `to + 1`), by `other`. Return `other`.

Beware that this routine is imperative: it changes the value of `this`.

```
var s1 = "foobar" | var s2 = s1 |
assert((s1[0, 3] = "quux") == "quux");
assert(s1 == "quuxbar");
assert(s2 == "quuxbar");
assert((s1[4, 7] = "") == "");
assert(s2 == "quux");
```

20.58 System

Details on the architecture the Urbi server runs on.

20.58.1 Prototypes

- `Object` (`??sec:std-Object`)

20.58.2 Slots

- `_exit(status)`
Shut the server down brutally: the connections are not closed, and the resources are not explicitly released (the operating system reclaims most of them: memory, file descriptors and so forth). Architecture dependent.

- `aliveJobs`
The number of detached routines currently running.

```
{
  var nJobs = aliveJobs;
  for (var i: [1s, 2s, 3s])
    detach({sleep(i)});
  sleep(0.5s);
  assert(aliveJobs - nJobs == 3);
  sleep(1s);
  assert(aliveJobs - nJobs == 2);
  sleep(1s);
  assert(aliveJobs - nJobs == 1);
  sleep(1s);
  assert(aliveJobs - nJobs == 0);
};
```

- `arguments`
The list of the command line arguments passed to the user script. This is especially useful in scripts.

```
$ cat >echo <<EOF
#! /usr/bin/env urbi
System.arguments;
shutdown;
EOF
$ chmod +x echo
$ ./echo 1 2 3
[00000172] ["1", "2", "3"]
$ ./echo -x 12 -v "foo"
[00000172] ["-x", "12", "-v", "foo"]
```

- `'assert'(assertion)`
Unless `ndebug` is true, throw an error if *assertion* is not verified. See also the assertion support in urbiscript, [Section 19.8](#).

```
'assert'(true);
'assert'(42);
'assert'(1 == 1 + 1);
[00000002:error] !!! failed assertion: 1.'=='(1.'+'(1))
```

- `assert_(assertion, message)`

If *assertion* does not evaluate to true, throw the failure *message*.

```
assert_(true,      "true failed");
assert_(42,        "42 failed");
assert_(1 == 1 + 1, "one is not two");
[00000001:error] !!! failed assertion: one is not two
```

- `assert_op(operator, lhs, rhs)`

Deprecated, use `assert` instead, see [Section 19.8](#).

- `backtrace`

Display the call stack on the channel `backtrace`.

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

```
//#push 100 "foo.u"
function innermost () { backtrace }|;
function inner ()     { innermost }|;
function outer ()     { inner }|;
function outermost () { outer }|;
outermost;
[00000013:backtrace] innermost (foo.u:101.25-33)
[00000014:backtrace] inner (foo.u:102.25-29)
[00000015:backtrace] outer (foo.u:103.25-29)
[00000016:backtrace] outermost (foo.u:104.1-9)
//#pop
```

- `cycle`

The number of execution cycles since the beginning.

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

```
{
  var first = cycle ; var second = cycle ;
  assert(first + 1 == second);
  first = cycle | second = cycle ;
  assert(first == second);
};
```

- `eval(source)`

Evaluate the urbiscript *source*, and return its result. The *source* must be complete, yet the terminator (e.g., ‘;’) is not required.

```
eval("1+2") == 1+2;
eval("\x\" * 10") == "x" * 10;
eval("eval(\"1\")") == 1;
eval("{ var x = 1; x + x; }") == 2;
```


The evaluation is performed in the context of the current object ([this](#)), in particular, to create local variables, create scopes.

```
// Create a slot in the current object.
eval("var x = 23;") == 23;
x == 23;
```

Exceptions are thrown on error.

```
eval("1/0");
[00008316:error] !!! 1.1-3: /: division by 0
try
{
  eval ("1/0")
}
catch (var e)
{
  assert
  {
    e.isA(Exception.Primitive);
    e.location.asString == "1.1-3";
    e.routine           == "/";
    e.message           == "division by 0";
  }
};
```

- `getenv(name)`

Return the value of the environment variable *name* as a [String](#) ([??sec:std-String](#)) if set, [nil](#) ([??sec:std-nil](#)) otherwise. See also [setenv](#) and [unsetenv](#).

```
getenv("UndefinedEnvironmentVariable").isNil;
!getenv("PATH").isNil;
```

- `load(file)`

Look for *file* in the Urbi path ([Section 18.1](#)), and load it. Throw a [Exception.FileNotFoundException](#) error if the file cannot be found. Return the last value of the file.

```
// Create the file '123.u' that contains exactly '123;'.
System.system("echo '123;' >123.u") == 0;
load("123.u") == 123;
```

- `loadFile(file)`

Load the urbiscript file *file*. Throw a [Exception.FileNotFoundException](#) error if the file cannot be found. Return the last value of the file.

```
// Create the file '123.u' that contains exactly '123;'.
System.system("echo '123;' >123.u") == 0;
loadFile("123.u") == 123;
```

- `loadLibrary(library)`
Load the library *library*, to be found in the `URBI_UOBJECT_PATH` search-path (see [Section 18.1](#)), or the default UObject path. The *library* may be a [String \(??sec:std-String\)](#) or a [Path \(??sec:std-Path\)](#). The C++ symbols are made available to the other C++ components. See also [loadModule](#).
- `loadModule(module)`
Load the UObject *module*. Same as [loadLibrary](#), except that the low-level C++ symbols are not made “global” (in the sense of the shared library loader).
- `lobbies`
Bounce to [Lobby.instances](#).
- `lobby`
Bounce to [Lobby.lobby](#).
- `maybeLoad(file, channel = Channel.null)`
Look for *file* in the Urbi path ([Section 18.1](#)). If the file is found announce on *Channel* that *file* is about to be loaded, and load it.

```
// Create the file ‘‘123.u’’ that contains exactly ‘‘123;’’.
System.system("echo '123;' >123.u") == 0;
maybeLoad("123.u") == 123;
maybeLoad("u.123").isVoid;
```

- `ndebug`
If true, do not evaluate the assertions. See [Section 19.8](#).

```
function one() { echo("called!"); 1 }|;
assert(!System.ndebug);

assert(one);
[00000617] *** called!

// Beware of copy-on-write.
System.ndebug = true|;
assert(one);

System.ndebug = false|;
assert(one);
[00000622] *** called!
```

- `PackageInfo`
See [System.PackageInfo \(??sec:std-System.PackageInfo\)](#).
- `period`
The *period* of the Urbi kernel. Influences the trajectories ([TrajectoryGenerator \(??sec:std-TrajectoryGenerator\)](#)), and the UObject monitoring. Defaults to 20ms.

```
System.period == 20ms;
```

- Platform

See [System.Platform](#) ([??sec:std-System.Platform](#)).

- programName

The path under which the Urbi process was called. This is typically ‘.../urbi’ ([Section 18.3](#)) or ‘.../urbi-launch’ ([Section 18.5](#)).

```
Path.new(System.programName).basename
  in ["urbi", "urbi.exe", "urbi-launch", "urbi-launch.exe"];
```

- reboot

Restart the Urbi server. Architecture dependent.

- redefinitionMode

Switch the current job in redefinition mode until the end of the current scope. While in redefinition mode, setSlot on already existing slots will overwrite the slot instead of erring.

```
var Global.x = 0;
[00000001] 0
{
  System.redefinitionMode;
  // Not an error
  var Global.x = 1;
  echo(Global.x);
};
[00000002] *** 1
// redefinitionMode applies only to the scope.
var Global.x = 0;
[00000003:error] !!! slot redefinition: x
```

- scopeTag

Bounce to [Tag.scope](#).

- searchFile(*file*)

Look for *file* in the Urbi path ([Section 18.1](#)) and return its [Path](#) ([??sec:std-Path](#)). Throw a [Exception.FileNotFoundException](#) error if the file cannot be found.

```
System.system("echo '123;' >123.u") == 0;
searchFile("123.u") == Path.cwd / Path.new("123.u");
```

- setenv(*name*, *value*)

Set the environment variable *name* to *value.asString*, and return this value. See also [getenv](#) and [unsetenv](#).

Windows Issues

Under Windows, setting to an empty value is equivalent to making undefined.

```
setenv("MyVar", 12) == "12";
getenv("MyVar") == "12";

// A child process that uses the environment variable.
System.system("exit $MyVar") >> 8 ==
  {if (Platform.isWindows) 0 else 12};
setenv("MyVar", 23) == "23";
System.system("exit $MyVar") >> 8 ==
  {if (Platform.isWindows) 0 else 23};

// Defining to empty is defining, unless you are on Windows.
setenv("MyVar", "") == "";
getenv("MyVar").isNil == Platform.isWindows;
```

- **shiftedTime**

Return the number of seconds elapsed since the Urbi server was launched. Contrary to **time**, time spent in frozen code is not counted.

```
{ var t0 = shiftedTime | sleep(1s) | shiftedTime - t0 }.round ~= 1;

1 ==
{
  var t = Tag.new|;
  var t0 = time|;
  var res;
  t: { sleep(1s) | res = shiftedTime - t0 },
  t.freeze;
  sleep(1s);
  t.unfreeze;
  sleep(1s);
  res.round;
};
```

- **shutdown**

Have the Urbi server shut down. All the connections are closed, the resources are released. Architecture dependent.

- **sleep(*duration*)**

Suspend the execution for *duration* seconds. No CPU cycle is wasted during this wait.

```
(time - {sleep(1s); time}).round == -1;
```

- **spawn(*function*, *clear*)**

Run the *function*, with fresh tags if *clear* is true, otherwise under the control of the current tags. Return the spawn **Job** (**??sec:std-Job**).

```

var jobs = [];
var res = [];
for (var i : [0, 1, 2])
{
  jobs << System.spawn(closure () { res << i; res << i },
                      true) |

  if (i == 2)
    break
}|
jobs;
[00009120] [Job<shell_11>, Job<shell_12>, Job<shell_13>]
// Wait for the jobs to be done.
jobs.each (function (var j) { j.waitForTermination });
assert (res == [0, 1, 0, 2, 1, 2]);

```

```

jobs = [];
res = [];
for (var i : [0, 1, 2])
{
  jobs << System.spawn(closure () { res << i; res << i },
                      false) |

  if (i == 2)
    break
}|
jobs;
[00009120] [Job<shell_14>, Job<shell_15>, Job<shell_16>]
// Give some time to get the output of the detached expressions.
sleep(100ms);
assert (res == [0, 1, 0]);

```

- `system(command)`

Ask the operating system to run the *command*. This is typically used to start new processes. The exact syntax of *command* depends on your system. On Unix systems, this is typically `‘/bin/sh’`, while Windows uses `‘command.exe’`.

Return the exit status.

Windows Issues

Under Windows, the exit status is always 0.

```

System.system("exit 0") == 0;
System.system("exit 23") >> 8
== { if (System.Platform.isWindows) 0 else 23 };

```

- `time`

Return the number of seconds elapsed since the Urbi server was launched. In presence of a frozen `Tag (??sec:std-Tag)`, see also `shiftedTime`.

```
{ var t0 = time | sleep(1s) | time - t0 }.round =~ 1;

2 ==
{
  var t = Tag.new|;
  var t0 = time|;
  var res;
  t: { sleep(1s) | res = time - t0 },
  t.freeze;
  sleep(1s);
  t.unfreeze;
  sleep(1s);
  res.round;
};
```

- `unsetenv(name)`
 Undefine the environment variable *name*, return its previous value. See also `getenv` and `setenv`.

```
setenv("MyVar", 12) == "12";
!getenv("MyVar").isNil;
unsetenv("MyVar") == "12";
getenv("MyVar").isNil;
```

20.59 System.PackageInfo

Information about Urbi SDK and its components.

- `copyrightHolder`
 The Urbi SDK copyright holder.

```
System.PackageInfo.copyrightHolder == "Gostai S.A.S.";
```

- `copyrightYears`
 The Urbi SDK copyright years.

```
System.PackageInfo.copyrightYears == "2005-2010";
```

20.60 System.Platform

A description of the platform (the computer) the server is running on.

- `host`
 The type of system Urbi SDK runs on. Composed of the CPU, the vendor, and the OS.

```
System.Platform.host ==
  "%s-%s-%s" % [System.Platform.hostCpu,
                 System.Platform.hostVendor,
                 System.Platform.hostOs];
```

- **hostCpu**

The CPU type of system Urbi SDK runs on. The following values are those for which Gostai provides binary builds.

```
System.Platform.hostCpu in ["i386", "i686", "x86_64"];
```

- **hostOs**

The OS type of system Urbi SDK runs on. For instance `darwin9.8.0` or `linux-gnu` or `mingw32`.

- **hostVendor**

The vendor type of system Urbi SDK runs on. The following values are those for which Gostai provides binary builds.

```
System.Platform.hostVendor in ["apple", "pc", "unknown"];
```

- **isWindows**

Whether running under Windows.

```
System.Platform.isWindows in [true, false];
```

- **kind**

Either `"POSIX"` or `"WIN32"`.

```
System.Platform.kind in ["POSIX", "WIN32"];
```

20.61 Tag

A *tag* is an object meant to label blocks of code in order to control them externally. Tagged code can be frozen, resumed, stopped... See also [Section 11.3](#).

20.61.1 Examples

20.61.1.1 Stop

To *stop* a tag means to kill all the code currently running that it labels. It does not affect “newcomers”.

```

var t = Tag.new|;
var t0 = time|;
t: every(1s) echo("foo"),
sleep(2.2s);
[00000158] *** foo
[00001159] *** foo
[00002159] *** foo

t.stop;
// Nothing runs.
sleep(2.2s);

t: every(1s) echo("bar"),
sleep(2.2s);
[00000158] *** bar
[00001159] *** bar
[00002159] *** bar

t.stop;

```

`System.stop` can be used to inject a return value to a tagged expression.

```

var t = Tag.new|;
var res;
detach(res = { t: every(1s) echo("computing") }|);
sleep(2.2s);
[00000001] *** computing
[00000002] *** computing
[00000003] *** computing

t.stop("result");
assert(res == "result");

```

Be extremely cautious, the precedence rules can be misleading: `var = tag: exp` is read as `(var = tag): exp` (i.e., defining `var` as an alias to `tag` and using it to tag `exp`), not as `var = { tag: exp }`. Contrast the following example, which is most probably an error from the user, with the previous, correct, one.

```

var t = Tag.new("t")|;
var res;
res = t: every(1s) echo("computing"),
sleep(2.2s);
[00000001] *** computing
[00000002] *** computing
[00000003] *** computing

t.stop("result");
assert(res == "result");
[00000004:error] !!! failed assertion: res == "result" (Tag<t> != "result")

```


20.61.1.2 Block/unblock

To *block* a tag means:

- Stop running pieces of code it labels (as with `stop`).
- Ignore new pieces of code it labels (this differs from `stop`).

One can *unblock* the tag. Contrary to `freeze/unfreeze`, tagged code does not resume the execution.

```
var ping = Tag.new("ping");
ping:
  every (1s)
    echo("ping"),
assert(!ping.blocked);
sleep(2.1s);
[00000000] *** ping
[00002000] *** ping
[00002000] *** ping

ping.block;
assert(ping.blocked);

ping:
  every (1s)
    echo("pong"),

// Neither new nor old code runs.
ping.unblock;
assert(!ping.blocked);
sleep(2.1s);

// But we can use the tag again.
ping:
  every (1s)
    echo("ping again"),
sleep(2.1s);
[00004000] *** ping again
[00005000] *** ping again
[00006000] *** ping again
```

As with `stop`, one can force the value of stopped expressions.

```
["foo", "foo", "foo"]
==
{
  var t = Tag.new;
  var res = [];
  for (3)
    detach(res << {t: sleep(inf)});
  t.block("foo");
  res;
};
```

20.61.1.3 Freeze/unfreeze

To *freeze* a tag means holding the execution of code it labels. This applies to code already being run, and “arriving” pieces of code.

```
var t = Tag.new|;
var t0 = time|;
t: every(1s) echo("time    : %.0f" % (time - t0)),
sleep(2.2s);
[00000158] *** time    : 0
[00001159] *** time    : 1
[00002159] *** time    : 2

t.freeze;
assert(t.frozen);
t: every(1s) echo("shifted: %.0f" % (shiftedTime - t0)),
sleep(2.2s);
// The tag is frozen, nothing is run.

// Unfreeze the tag: suspended code is resumed.
// Note the difference between "time" and "shiftedTime".
t.unfreeze;
assert(!t.frozen);
sleep(2.2s);
[00004559] *** shifted: 2
[00005361] *** time    : 5
[00005560] *** shifted: 3
[00006362] *** time    : 6
[00006562] *** shifted: 4
```

20.61.1.4 Scope tags

Scopes feature a `scopeTag`, i.e., a tag which will be stop when the execution reaches the end of the current scope. This is handy to implement cleanups, how ever the scope was exited from.

```
{
  var t = scopeTag;
  t: every(1s)
    echo("foo"),
  sleep(2.2s);
};
[00006562] *** foo
[00006562] *** foo
[00006562] *** foo

{
  var t = scopeTag;
  t: every(1s)
    echo("bar"),
  sleep(2.2s);
  throw 42;
};
[00006562] *** bar
```

```
[00006562] *** bar
[00006562] *** bar
[00006562:error] !!! 42
sleep(2s);
```

20.61.1.5 Enter/leave events

Tags provide two events, **enter** and **leave**, that trigger whenever flow control enters or leaves statements the tag.

```
var t = Tag.new("t");
[00000000] Tag<t>

at (t.enter?)
  echo("enter");
at (t.leave?)
  echo("leave");

t: {echo("inside"); 42};
[00000000] *** enter
[00000000] *** inside
[00000000] *** leave
[00000000] 42
```

This feature is fundamental; it is a concise and safe way to ensure code will be executed upon exiting a chunk of code (like **RAII** in C++ or **finally** in Java). The exit code will be run no matter what the reason for leaving the block was: natural exit, exceptions, flow control statements like **return** or **break**, ...

For instance, suppose we want to make sure we turn the gas off when we're done cooking. Here is the *bad* way to do it:

```
{
  function cook()
  {
    turn_gas_on();

    // Cooking code ...

    turn_gas_off();
  }

  enter_the_kitchen();
  cook();
  leave_the_kitchen();
};
```

This is wrong because there are several situations where we could leave the kitchen with gas still turned on. Consider the following cooking code:

```
{
  function cook()
  {
```

```

    turn_gas_on();

    if (meal_ready)
    {
        echo("The meal is already there, nothing to do!");
        // Oops ...
        return
    };

    for (var ingredient in recipe)
        if (ingredient not in kitchen)
            // Oops ...
            throw Exception("missing ingredient: %s" % ingredient)
        else
            put_ingredient();

    // ...

    turn_gas_off();
}

enter_the_kitchen();
cook();
leave_the_kitchen();
};

```

Here, if the meal was already prepared, or if an ingredient is missing, we will leave the `cook` function without executing the `turn_gas_off` statement, through the `return` statement or the exception. The right way to ensure gas is necessarily turned off is:

```

{
    function cook()
    {
        var with_gas = Tag.new("with_gas");

        at (with_gas.enter?)
            turn_gas_on();
        at (with_gas.leave?)
            turn_gas_off();

        with_gas: {
            // Cooking code. Even if exception are thrown here or return is called,
            // the gas will be turned off.
        }
    }
}

enter_the_kitchen();
cook();
leave_the_kitchen();
};

```

20.61.1.6 Begin/end

The `begin` and `end` methods enable to monitor when code is executed. The following example illustrates the proper use of `enter` and `leave` events ([Section 20.61.1.5](#)), which are used to implement this feature.

```
var mytag = Tag.new("mytag");
[00000000] Tag<mytag>

mytag.begin: echo(1);
[00000000] *** mytag: begin
[00000000] *** 1

mytag.end: echo(2);
[00000000] *** 2
[00000000] *** mytag: end

mytag.begin.end: echo(3);
[00000000] *** mytag: begin
[00000000] *** 3
[00000000] *** mytag: end
```

20.61.2 Construction

As any object, tags are created using `new` to create derivatives of the `Tag` object. The name is optional, it makes easier to display a tag and remember what it is.

```
// Anonymous tag.
var t1 = Tag.new;
[00000001] Tag<tag_8>

// Named tag.
var t2 = Tag.new("cool name");
[00000001] Tag<cool name>
```

20.61.3 Slots

- `begin`
A sub-tag that prints out "tag_name: begin" each time flow control enters the tagged code. See [Section 20.61.1.6](#).
- `block(result = void)`
Block any code tagged by `this`. Blocked tags can be unblocked using `unblock`. If some `result` was specified, let stopped code return `result` as value. See [Section 20.61.1.2](#).
- `blocked`
Whether code tagged by `this` is blocked. See [Section 20.61.1.2](#).

- **end**
A sub-tag that prints out "tag_name: end" each time flow control leaves the tagged code. See [Section 20.61.1.6](#).
- **enter**
An event triggered each time the flow control enters the tagged code. See [Section 20.61.1.5](#).
- **freeze**
Suspend code tagged by [this](#), already running or forthcoming. Frozen code can be later unfrozen using [unfreeze](#). See [Section 20.61.1.3](#).
- **frozen**
Whether the tag is frozen. See [Section 20.61.1.3](#).
- **leave**
An event triggered each time flow control leaves the tagged code. See [Section 20.61.1.5](#).
- **scope**
Return a fresh Tag whose **stop** will be invoked at the end of the current scope. This function is likely to be removed. See [Section 20.61.1.4](#).
- **stop(result = void)**
Stop any code tagged by [this](#). If some *result* was specified, let stopped code return *result* as value. See [Section 20.61.1.1](#).
- **tags**
All the undeclared tags are created as slots in this object. Using this feature is discouraged.

```
{
  assert ("brandNewTag" not in Tag.tags.localSlotNames);
  brandNewTag: 1;
  assert ("brandNewTag" in Tag.tags.localSlotNames);
  assert (Tag.tags.brandNewTag.isA(Tag));
};
```
- **unblock**
Unblock [this](#). See [Section 20.61.1.2](#).
- **unfreeze**
Unfreeze code tagged by [this](#). See [Section 20.61.1.3](#).

20.61.4 Hierarchical tags

Tags can be arranged in a parent/child relationship: any operation done on a tag — freezing, stopping, ... is also performed on its descendants. Another way to see it is that tagging a piece of code with a child will also tag it with the parent. To create a child Tag, simply clone its parent.

```

var parent = Tag.new |
var child = parent.clone |

// Stopping parent also stops children.
{
  parent: {sleep(100ms); echo("parent")},
  child: {sleep(100ms); echo("child")},
  parent.stop;
  sleep(200ms);
  echo("end");
};
[00000001] *** end

// Stopping child has no effect on parent.
{
  parent: {sleep(100ms); echo("parent")},
  child: {sleep(100ms); echo("child")},
  child.stop;
  sleep(200ms);
  echo("end");
};
[00000002] *** parent
[00000003] *** end

```

Hierarchical tags are commonly laid out in slots so as to reflect their tag hierarchy.

```

var a = Tag.new;
var a.b = a.clone;
var a.b.c = a.b.clone;

a:    foo; // Tagged by a
a.b:  bar; // Tagged by a and b
a.b.c: baz; // Tagged by a, b and c

```

20.62 Timeout

Timeout objects can be used as [Tags \(??sec:std-Tag\)](#) to execute some code in limited time.

20.62.1 Prototypes

- [Tag \(??sec:std-Tag\)](#)

20.62.2 Construction

At construction, a Timeout takes a duration, and a [Boolean \(??sec:std-Boolean\)](#) stating whether an exception should be thrown on timeout (by default, it does).

```

Timeout.new(300ms);
[00000000] Timeout_0x953c1e0
Timeout.new(300ms, false);

```

```
[00000000] Timeout_0x953c1e0
```

20.62.3 Examples

Use it as a tag:

```
var t = Timeout.new(300ms);
[00000000] Timeout_0x133ec0
t:{
  echo("This will be displayed.");
  sleep(500ms);
  echo("This will not.");
};
[00000000] *** This will be displayed.
[00000007:error] !!! Timeout_0x133ec0 has timed out.
```

The same Timeout, `t` can be reused. It is armed again each time it is used to tag some code.

```
t: { echo("Open"); sleep(1s); echo("Close"); };
[00000007] *** Open
[00000007:error] !!! Timeout_0x133ec0 has timed out.
t: { echo("Open"); sleep(1s); echo("Close"); };
[00000007] *** Open
[00000007:error] !!! Timeout_0x133ec0 has timed out.
```

Even if exceptions have been disabled, you can check whether the count-down expired with `timedOut`.

```
t:sleep(500ms);
[00000007:error] !!! Timeout_0x133ec0 has timed out.
if (t.timedOut)
  echo("The Timeout expired.");
[00000000] *** The Timeout expired.
```

20.62.4 Slots

- `launch`
Fire `this`.

20.63 TrajectoryGenerator

The trajectory generators change the value of a given variable from an *initial value* to a *target value*. They can be *open-loop*, i.e., the intermediate values depend only on the initial and/or target value of the variable; or *closed-loop*, i.e., the intermediate values also depend on the current value of the variable.

Open-loop trajectories are insensitive to changes made elsewhere to the variable. Closed-loop trajectories *are* sensitive to changes made elsewhere to the variable — for instance when the human physically changes the position of a robot's motor.

Trajectory generators are not made to be used directly, rather use the “continuous assignment” syntax ([Section 19.10](#)).

20.63.1 Prototypes

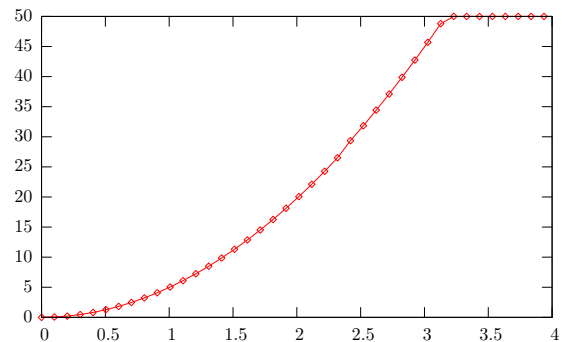
- `Object` (`??sec:std-Object`)

20.63.2 Examples

20.63.2.1 Accel

The `Accel` trajectory reaches a target value at a fixed acceleration (`accel` attribute).

```
var y = 0;
y = 50 accel:10,
```

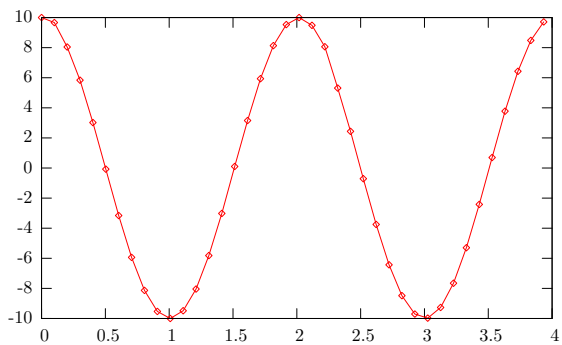


20.63.2.2 Cos

The `Cos` trajectory implements a cosine around the target value, given an amplitude (`ampli` attribute) and period (`cos` attribute).

This trajectory is not “smooth”: the initial value of the variable is not taken into account.

```
var y = 0;
y = 0 cos:2s ampli:10,
```

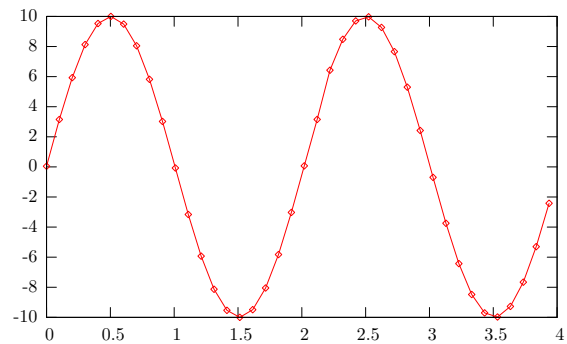


20.63.2.3 Sin

The `Sin` trajectory implements a sine around the target value, given an amplitude (`ampli` attribute) and period (`sin` attribute).

This trajectory is not “smooth”: the initial value of the variable is not taken into account.

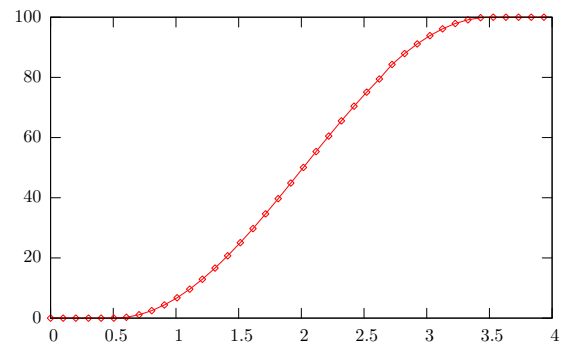
```
var y = 0;
y = 0 sin:2s ampli:10,
```



20.63.2.4 Smooth

The **Smooth** trajectory implements a sigmoid. It changes the variable from its current value to the target value “smoothly” in a given amount of time (**smooth** attribute).

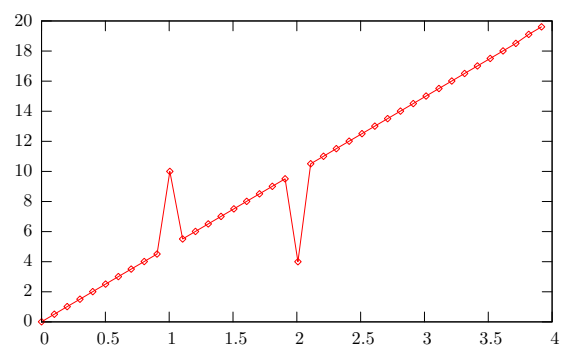
```
var y = 0;
{
  sleep(0.5s);
  y = 100 smooth:3s,
},
```



20.63.2.5 Speed

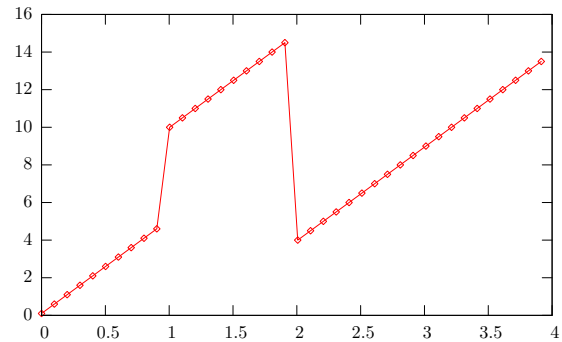
The **Speed** trajectory changes the value of the variable from its current value to the target value at a fixed speed (the **speed** attribute).

```
var y = 0;
assign: y = 20 speed: 5,
{
  sleep(1s);
  y = 10;
  sleep(1s);
  y = 4;
},
```



If the **adaptive** attribute is set to true, then the duration of the trajectory is constantly reevaluated.

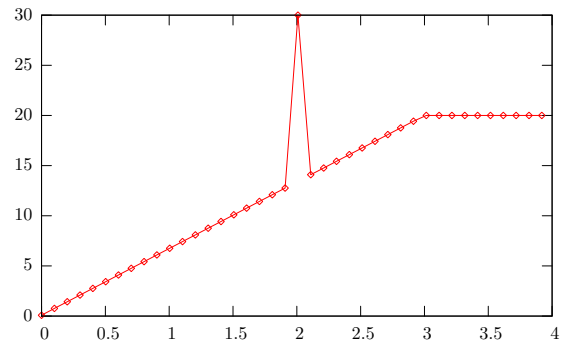
```
var y = 0;
assign: y = 20 speed: 5 adaptive: true,
{
  sleep(1s);
  y = 10;
  sleep(1s);
  y = 4;
},
```



20.63.2.6 Time

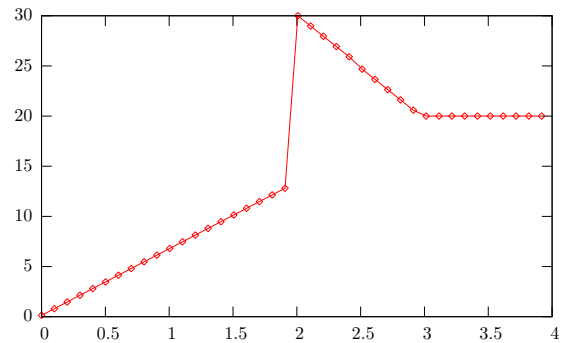
The **Time** trajectory changes the value of the variable from its current value to the target value within a given duration (the **time** attribute).

```
var y = 0;
assign: y = 20 time:3s,
{
  sleep(2s);
  y = 30;
},
```



If the **adaptive** attribute is set to true, then the duration of the trajectory is constantly reevaluated.

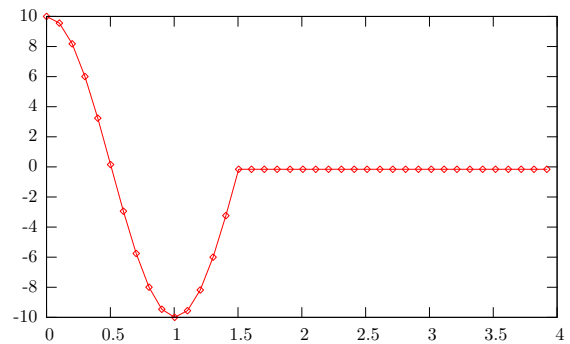
```
var y = 0;
assign: y = 20 time:3s adaptive: true,
{
  sleep(2s);
  y = 30;
},
```



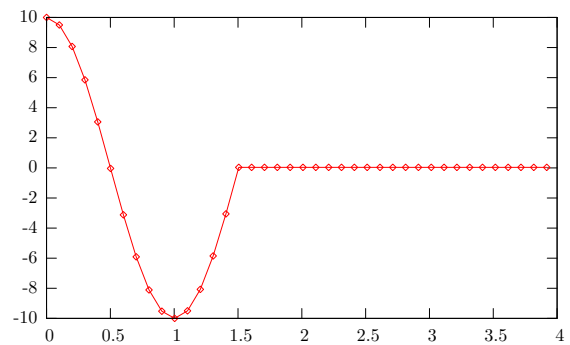
20.63.2.7 Trajectories and Tags

Trajectories can be managed using **Tags** ([??sec:std-Tag](#)). Stopping or blocking a tag that manages a trajectory kill the trajectory.

```
var y = 0;
assign: y = 0 cos:2s ampli:10,
{
  sleep(1.5s);
  assign.stop;
},
```

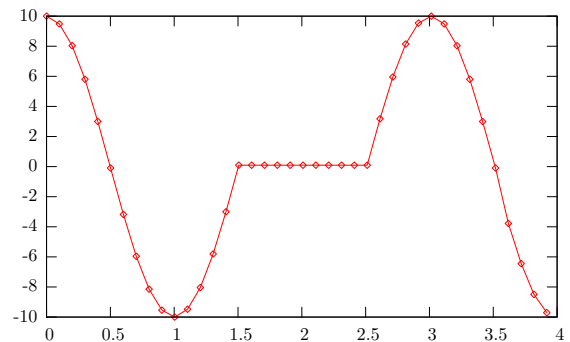


```
var y = 0;
assign: y = 0 cos:2s ampli:10,
{
  sleep(1.5s);
  assign.block;
  sleep(1s);
  assign.unblock;
},
```



When a trajectory is frozen, its local time is frozen too, the movement proceeds from where it was rather than from where it would have been had it been not frozen.

```
var y = 0;
assign: y = 0 cos:2s ampli:10,
{
  sleep(1.5s);
  assign.freeze;
  sleep(1s);
  assign.unfreeze;
},
```



20.63.3 Construction

You are not expected to construct trajectory generators by hand, using modifiers is the recommended way to construct trajectories. See [Section 19.10](#) for details about trajectories, and see [Section 20.63.2](#) for an extensive set of examples.

20.63.4 Slots

- **Accel**
This class implements the **Accel** trajectory ([Section 20.63.2.1](#)). It derives from **OpenLoop**.
- **ClosedLoop**
This class factors the implementation of the *closed-loop* trajectories. It derives from **TrajectoryGenerator**.
- **OpenLoop**
This class factors the implementation of the *open-loop* trajectories. It derives from **TrajectoryGenerator**.
- **Sin**
This class implements the **Cos** and **Sin** trajectories ([Section 20.63.2.2](#), [Section 20.63.2.3](#)). It derives from **OpenLoop**.
- **Smooth**
This class implements the **Smooth** trajectory ([Section 20.63.2.4](#)). It derives from **OpenLoop**.
- **SpeedAdaptive**
This class implements the **Speed** trajectory when the **adaptive** attribute is given ([Section 20.63.2.5](#)). It derives from **ClosedLoop**.
- **Time**
This class implements the non-adaptive **Speed** and **Time** trajectories ([Section 20.63.2.5](#), [Section 20.63.2.6](#)). It derives from **OpenLoop**.
- **TimeAdaptive**
This class implements the **Time** trajectory when the **adaptive** attribute is given ([Section 20.63.2.6](#)). It derives from **ClosedLoop**.

20.64 Triplet

A *triplet* (or *triple*) is a container storing three objects.

20.64.1 Prototype

- **Tuple** ([??sec:std-Tuple](#))

20.64.2 Construction

A **Triplet** is constructed with three arguments.

```
Triplet.new(1, 2, 3);
[00000001] (1, 2, 3)

Triplet.new(1, 2);
[00000003:error] !!! Triplet.init: expected 3 arguments, given 2
```

```
Triplet.new(1, 2, 3, 4);
[00000003:error] !!! Triplet.init: expected 3 arguments, given 4
```

20.64.3 Slots

- **first**

Return the first member of the pair.

```
Triplet.new(1, 2, 3).first == 1;
Triplet[0] === Triplet.first;
```

- **second**

Return the second member of the triplet.

```
Triplet.new(1, 2, 3).second == 2;
Triplet[1] === Triplet.second;
```

- **third**

Return the third member of the triplet.

```
Triplet.new(1, 2, 3).third == 3;
Triplet[2] === Triplet.third;
```

20.65 Tuple

A *tuple* is a container storing a fixed number of objects. Examples include **Pair** ([??sec:std-Pair](#)) and **Triplet** ([??sec:std-Triplet](#)).

20.65.1 Prototype

- **Object** ([??sec:std-Object](#))

20.65.2 Construction

The **Tuple** object is not meant to be instantiated, its main purpose is to share code for its descendants, such as **Pair** ([??sec:std-Pair](#)). Yet it accepts its members as a list.

```
var t = Tuple.new([1, 2, 3]);
[00000000] (1, 2, 3)
```

The output generated for a **Tuple** can also be used to create a **Tuple**. Expressions are put inside parentheses and separated by commas. One extra comma is allowed after the last element. To avoid confusion between a 1 member **Tuple** and a parenthesized expression, the extra comma must be added. **Tuple** with no expressions are also accepted.

```
// not a Tuple
(1);
[000000000] 1

// Tuples
();
[000000000] ()
(1,);
[000000000] (1,)
(1, 2);
[000000000] (1, 2)
(1, 2, 3, 4,);
[000000000] (1, 2, 3, 4)
```

20.65.3 Slots

- **asString**
The string `'(first, second, ..., last)'`, using `asPrintable` to convert members to strings.

```
()asString == "()";
(1,)asString == "(1,)";
(1, 2).asString == "(1, 2)";
(1, 2, 3, 4,).asString == "(1, 2, 3, 4)";
```

- **matchAgainst(handler, pattern)**
Pattern matching on members. See [Pattern \(??sec:std-Pattern\)](#).

```
{
  // Match a tuple.
  (var first, var second) = (1, 2);
  assert { first == 1; second == 2 };
};
```

- **size**
Number of members.

```
()size == 0;
(1,)size == 1;
(1, 2, 3, 4).size == 4;
(1, 2, 3, 4,).size == 4;
```

- **'[]'(index)**
Return the *index*-th element. *index* must be in bounds.

```
(1, 2, 3)[0] == 1;
(1, 2, 3)[1] == 2;
```

- `'[]='(index, value)`

Set (and return) the *index*-th element to *value*. *index* must be in bounds.

```
{
  var t = (1, 2, 3);
  assert
  {
    (t[0] = 2) == 2;
    t == (2, 2, 3);
  };
};
```

- `'<'(other)`

Lexicographic comparison between two tuples.

```
(0, 0) < (0, 1);
(0, 0) < (1, 0);
(0, 1) < (1, 0);
```

- `'==(other)`

Whether `this` and `other` have the same contents (equality-wise).

```
(1, 2) == (1, 2);
!((1, 1) == (2, 2));
```

- `'*(value)`

Return a Tuple in which all elements of `this` are multiplied by a *value*.

```
(0, 1, 2, 3) * 3 == (0, 3, 6, 9);
(1, "foo") * 2 == (2, "foofoo");
```

- `'+'(other)`

Return a Tuple in which each element of `this` is summed with its corresponding element in the *other* Tuple.

```
(0, 1) + (2, 3) == (2, 4);
(1, "foo") + (2, "bar") == (3, "foobar");
```

20.66 UObject

UObject is used by the UObject API (see [Part I](#)) to represent a bound C++ instance.

All the UObjects are copied under an unique name as slots of `uobjects`.

20.66.1 Prototypes

- [Object](#) ([??sec:std-Object](#))

20.66.2 Slots

- `__uobjectName`
Unique name assigned to this object. This is also the slot name of [Global.uobjects](#) containing this `UObject`.

20.67 UValue

The `UValue` object is used internally by the `UObject` API and is mostly hidden from the user.

20.68 UVar

This class is used internally by the `UObject` middleware (see [Part I](#)) to represent a variable that can be hooked in C++. Each slot on which a C++ `urbi::UVar` exists contains an instance of this class.

Instances of `UVar` are mostly transparent, they appear as the value they contain. Thus, since the `UVar` evaluates to the contained value, you must use `getSlot` to manipulate the `UVar` itself.

20.68.1 Construction

To instantiate a new `UVar`, pass the owner object and the slot name to the constructor.

```
UVar.new(Global, "x") |
Global.x = 5;
[000000001] 5
x;
[000000002] 5
```

20.68.2 Prototypes

- [Object](#) ([??sec:std-Object](#))

20.68.3 Slots

- `copy(targetObject, targetName)`
Copy the `UVar` to the slot `targetName` of object `targetObject`. Since the `UVar` is using properties, you must use this method to copy it to an other location.
- `notifyChange(handle, onChange)`
Similar to the C++ `UNotifyChange` (see [Section 3.5](#)), register `onChange` and call it each time this `UVar` is written to. `handle` is a `WeakPointer` used to automatically unregister the callback. You can use the global `uobjects.handle`.

```

UVar.new(Global, "y")|
Global.getSlot("y").notifyChange(uobjects_handle, closure() {
  echo("The value is now " + Global.y)
})|
Global.y = 12;
[00000001] *** The value is now 12
[00000002] 12

```

- `notifyAccess(handle, onAccess)`
Similar to the C++ `UNotifyAccess`, calls *onAccess* each time the `UVar` is accessed (read).
- `owned`
True if the `UVar` is in owned mode, that is if it contains both a sensor and a command value.

20.69 void

The special entity `void` is an object used to denote “no value”. It has no prototype and cannot be used as a value. In contrast with `nil` ([??sec:std-nil](#)), which is a valid object, `void` denotes a value one is not allowed to read.

20.69.1 Prototypes

None.

20.69.2 Construction

`void` is the value returned by constructs that return no value.

```

void.isVoid;
{}.isVoid;
{if (false) 123}.isVoid;

```

20.69.3 Slots

- `acceptVoid`
Trick `this` so that, even if it is `void` it can be used as a value. See also `unacceptVoid`.

```

void.foo;
[00096374:error] !!! unexpected void
void.acceptVoid.foo;
[00102358:error] !!! lookup failed: foo

```

- `isVoid`
Whether `this` is `void`. Therefore, return `true`.

```
void.isVoid;  
void.acceptVoid.isVoid;  
! 123.isVoid;
```

- **unacceptVoid**

Remove the magic from `this` that allowed to manipulate it as a value, even if it `void`. See also `acceptVoid`.

```
void.acceptVoid.unacceptVoid.foo;  
[00096374:error] !!! unexpected void
```


Chapter 21

Communication with ROS

This chapter is not an introduction to using ROS from Urbi, see [Chapter 13](#) for a tutorial.

Urbi provides a set of tools to communicate with ROS (Robot Operating System). For more information about ROS, please refer to <http://www.ros.org>. Urbi, acting as a ROS node, is able to interact with the ROS world.

Requirements You need to have installed ROS (possibly a recent version), and compiled all of the common ROS tools (`rxconsole`, `roscore`, `roscpp`, ...).

You also need to have a few environment variables set, normally provided with ROS installation: `ROS_ROOT`, `ROS_MASTER_URI` and `ROS_PACKAGE_PATH`.

Usage The classes are implemented as uobjects (see [Part I](#)), [Ros.Service](#) ([??sec:std-Ros.Service](#)), [Ros.Topic](#) ([??sec:std-Ros.Topic](#)), and [Ros](#) ([??sec:std-Ros](#)).

This module will be loaded automatically if `ROS_ROOT` is detected in your environment. If `roscore` is not launched, you will be noticed by a warning and Urbi will check regularly for `roscore`.

If for any reason you need to load this module manually, use:

```
loadModule("urbi/ros");
```

21.1 Ros

This object provides some handy tools to know the status of `roscore`, to list the different nodes, topics, services, It also serves as a namespace entry point for ROS entities, such as [Ros.Topic](#) ([??sec:std-Ros.Topic](#)) and so forth.

21.1.1 Construction

There is no construction, since this class only provides a set of tools related to ROS in general, or the current node (which is unique per instance of Urbi).

21.1.2 Slots

- **checkMaster**
Whether `roscore` is accessible.
- **name**
The name of the current node associated with Urbi (as a string). The name of an Urbi node is generally composed of `‘/urbi_+random’` sequence.

- **nodes**
A dictionary containing all the nodes known by `roscore`. Each key of the dictionary is a node name. The associated value is an other dictionary, with the following keys: `publish`, `subscribe`, `services`. Each of these keys is associated with a list of topics or services.

```
"/rosout" in Ros.nodes;
Ros.name in Ros.nodes;
```

- **Service**
The `Ros.Service` (`??sec:std-Ros.Service`) class.
- **services**
A dictionary containing all the services known by `roscore`. Each key is the name of a service, and the associated value is a list of nodes that provide this service.

```
//#roscore
var services = Ros.services|
var name = Ros.name|;
assert
{
    "/rosout/get_loggers" in services;
    "/rosout/set_logger_level" in services;
    (name + "/get_loggers") in services;
    (name + "/set_logger_level") in services;
};
```

- **Topic**
The `Ros.Topic` (`??sec:std-Ros.Topic`) class.
- **topics**
A dictionary containing all the topics advertised to `roscore`. Each key is the name of a topic, and the associated value is an other dictionary, with the following keys: `subscribers`, `publishers`, `type`.

```
var topics = Ros.topics|;
topics.keys;
[03316144] ["/rosout_agg"]
topics["/rosout_agg"];
[03325634] ["publishers" => ["/rosout"], "subscribers" => [], "type" => "roslib/Log"]
```

21.2 Ros.Topic

This `UObject` provides a handy way to communicate through ROS topics, by subscribing to existent topics or advertising to them.

21.2.1 Construction

To create a new topic, call `Ros.topic.new` with a string (for the name of the topic you want to subscribe to / on which you want to advertise).

The topic name can only contain alphanumerical characters, `'/'` and `'_'`, and cannot be empty. If the topic name is invalid, an exception is thrown and the topic is not created.

Until you decide what you want to do with your topic ([subscribe](#) or [advertise](#)), you are free to call `init` to change its name.

21.2.2 Slots

Some slots on this `UObject` have no interest once the type of instance is determined. For example, you cannot call `unsubscribe` if you advertise, and in the same way you cannot call `publish` if you subscribed to a topic.

21.2.2.1 Common

- **name**

The name of the topic you provided in `init`.

```
Ros.Topic.new("/test").name;
[00011644] "/test"
```

- **subscriberCount**

The number of subscribers for the topic given in `init`; 0 if the topic is not registered.

- **structure**

Once [advertise](#) or [subscribe](#) has been called, this slot contains a template of the message type, with default values for each type (empty strings, zeros, ...). This template is made of dictionaries ([Dictionary](#) (`??sec:std-Dictionary`)).

```
var logTopic = Ros.Topic.new("/rosout_agg")|
logTopic.subscribe|
logTopic.structure.keys;
[00133519] ["file", "function", "header", "level", "line", "msg", "name", "topics"]
```

21.2.2.2 Subscription

- **subscribe**

Subscribe to the provided topic. Throw an exception it doesn't exist, or if the type advertised by ros for this topic does not exist. From the call of this method, a direct

connection is made between the advertiser and the subscriber which starts to deserialize received messages.

- **unsubscribe**
Unsubscribe from a previously subscribed channel, and set the state of the instance as if `init` was called but not subscribe.
- **onMessage**
Event triggered when a new message is received from a subscribed channel, the payload of this event contains the message.

```
var t = Ros.Topic.new("/test")|;
msg: at (t.onMessage?(var m)) echo(m);
t.subscribe;
```

21.2.2.3 Advertising

- **advertise(*type*)**
Tells ROS that this node will advertise on the topic given in `init`, with the type *type*. *type* must be a valid ROS type, such as `roslib/Log`. If *type* is an empty string, the method will try to check whether or not a node is already advertising on this topic, and its type.

```
var stringPub = Ros.Topic.new("/mytest")|;
stringPub.advertise("std_msgs/String");
stringPub.structure;
[00670809] ["data" => ""]
```

- **onConnect**
This event is triggered when the current instance is advertising on a topic and a node subscribes to this topic. The payload of this event is the name of the node that subscribed.

```
var p = Ros.Topic.new("/test/publisher")|;
con: at (p.onConnect?(var n))
  echo(n + " has subscribed to " + p.name);
p.advertise("roslib/Log");
```

- **onDisconnect**
This event is triggered when the current instance is advertising on a topic, and a node unsubscribes from this topic. The payload contains the name of the node that unsubscribed.
- **publish(*message*)**
Publish *message* to the current topic. This method is only usable if `advertise` has been called previously. The message must follow the same structure as the one in the slot `structure`, or it will throw an exception telling which key is missing / wrong in the dictionary.
- **'<<'(*message*)**
An alias for `publish`.


```
stringPub << ["data" => "Hello world!"];
```

- `unadvertise`

Tells ROS that we stop publishing on this topic. The `UObject` then gets back to a state where it could call `init`, `subscribe` or `advertise`.

21.2.3 Example

This is a typical example of the creation of a publisher, a subscriber, and message transmission between both of them.

First we need to declare our Publisher, the topic name is `/example`, and the type of message that will be sent on this topic is `"std_msgs/String"`. This type contains a single field called `"data"`, holding a string. We also set up handlers for `onConnect` and `onDisconnect` to be noticed when someone subscribes to us.

```
var publisher = Ros.Topic.new("/example")|
con: at (publisher.onConnect?(var name))
  echo(name[0,5] + " is now listening on " + publisher.name);
dcon: at (publisher.onDisconnect?(var name))
  echo(name[0,5] + " is no longer listening on " + publisher.name);
publisher.advertise("std_msgs/String");
```

Then we subscribe to the freshly created topic, and for each message, we display the `"data"` section (which is the content of the message). Due to the previous `at` above, a message is displayed at subscription time.

```
var subscriber = Ros.Topic.new("/example")|
msg: at (subscriber.onMessage?(var m))
  echo(m["data"]);
subscriber.subscribe;
[00026580] *** /urbi is now listening on /example

// The types should be the same, ensure that.
assert { subscriber.structure == publisher.structure };
```

Now we can send a message, and get it back through the `at` in the section above. To do this we first copy the template structure and then fill the field `"data"` with our message.

```
var message = publisher.structure.new;
[00098963] ["data" => ""]
message["data"] = "Hello world!";

// publish the message.
publisher << message;
// Leave some time to asynchronous communications before shutting down.
sleep(200ms);
[00098964] *** Hello world!

subscriber.unsubscribe;
[00252566] *** /urbi is no longer listening on /example
```

21.3 Ros.Service

This `UObject` provides a handy way to call services provided by other ROS nodes.

21.3.1 Construction

To create a new instance of this object, call `Ros.Service.new` with a string representing which service you want to use, and a Boolean stating whether the connection between you and the service provider should be kept opened (pass `true` for better performances on multiple requests).

The service name can only contain alphanumerical characters, `'/'`, `'_'`, and cannot be empty. If the service name is invalid, an exception is thrown, and the Object is not created.

Then if the service does not exist, an other exception is thrown. Since the initialization is asynchronous internally, you need to wait for `yourService.initialized` to be true to be able to call `request`.

21.3.2 Slots

- **initialized**
Boolean. Whether the method `request` is ready to be called, and whether `resStruct` and `reqStruct` are filled.

```
var logService = Ros.Service.new("/rosout/get_loggers", false);
waituntil(logService.initialized);
```

- **name**
The name of the current service.

```
logService.name;
[00036689] "/rosout/get_loggers"
```

- **resStruct**
Get the template of the response message type, with default values for each type (empty strings, zeros, ...). This template is made of dictionaries.

```
logService.resStruct;
[00029399] ["loggers" => {}]
```

- **reqStruct**
Get the template of the request message type, with default values for each type (empty strings, zeros, ...). This template is made of dictionaries.

```
logService.reqStruct;
[00029399] [ => ]
```

- **request(req)**
Synchronous call to the service, providing `req` as your request (following the structure of `reqStruct`). The returned value is a dictionary following the structure of `resStruct`, containing the response to your request.

```
for (var item in logService.request(=>)["loggers"])
  echo(item);
[00236349] *** ["level" => "INFO", "name" => "ros"]
[00236349] *** ["level" => "INFO", "name" => "ros.roscpp"]
[00236350] *** ["level" => "WARN", "name" => "ros.roscpp.superdebug"]
[00236350] *** ["level" => "DEBUG", "name" => "roscpp_internal"]
```


Chapter 22

Gostai Standard Robotics API

This section aims at clarifying the naming conventions in Urbi Engines for standard hardware/-software devices and components implemented as UObject and the corresponding methods/attributes/events to access them. The list of available hardware types and software component is increasing and this document will be updated accordingly. Please contact us directly, should you be working on a component not described or closely related to one described here:

standard@gostai.com

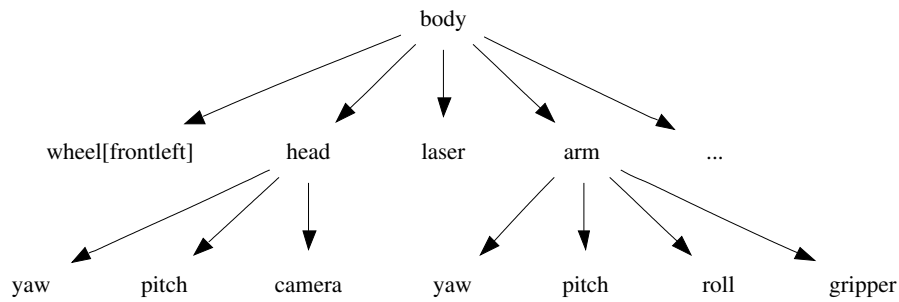
Any implementation of an Urbi server must comply with the latest version of this standard to get the “Urbi Ready” certification from Gostai S.A.S.

Gostai S.A.S. is currently the only authority which has the ability to deliver an “Urbi Ready” certification.

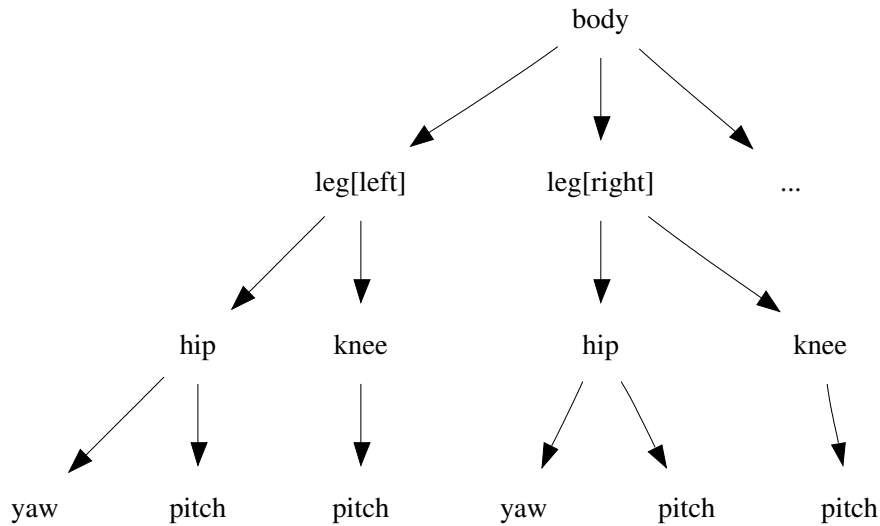
“Urbi Ready” and the associated logo are trademarks of Gostai S.A.S. and should not be used or displayed in any way without an explicit written agreement from Gostai.

22.1 The Structure Tree

The robot will be described as a set of *components* organized in a hierarchical structure called the *structure tree*. The relationship between a component and a sub-component in the tree is a ‘part-of’ inclusion relationship. From the point of view of Urbi, each component in the tree is an object, and it contains attributes pointing to its sub-components. Here is an example illustrating a part of a hierarchy that could be found with a wheeled robot with a gripper:



And here is another example for an humanoid robot:



The leaves of the tree are called *devices*, and they usually match physical devices in the robot: motors, sensors, lights, camera, etc. Inside Urbi, the various objects corresponding to the tree components are accessed by following the path of objects inclusions, like in the example below (shortcuts will be described later):

```
body.leg[right].hip.tilt;
body.arm.grip;
body.laser;
// ...
```

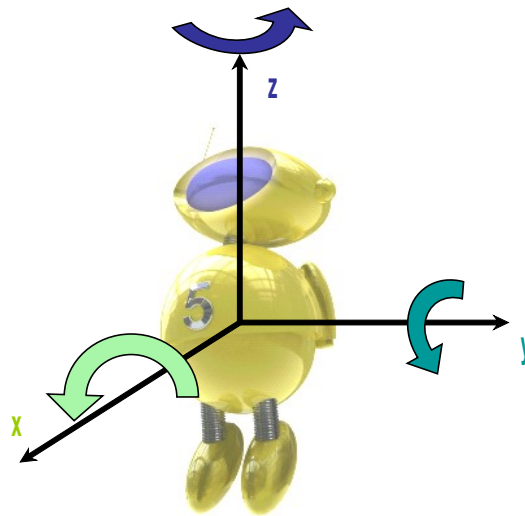
The structure tree should not be mistaken for a representation of the kinematics chain of the robot. The kinematics chain is built from a subset of the devices corresponding to motor devices, and it represents spatial connections between them. Except for these motor devices, the structure tree components do not have a direct counterpart in the kinematics chain, or, if

they do, it is as a subset of the kinematics chain (for example, `leg[right]` is a subset of the whole kinematics chain).

The goal of this standard is to provide guidelines on how to define the components and the structure tree, knowing the kinematics chain of the robot.

22.2 Frame of Reference

In many cases, it will be necessary to refer to an absolute frame of reference attached to the robot body. To avoid ambiguities, the standard frame of reference will have the following definition:



Origin the center of mass of the robot

X axis oriented towards the front of the robot. If there is a camera, the front is defined by the default direction of the camera, otherwise the front will be seen as the natural frontal orientation for a mobile robot (the direction of “forward” movement). If the robot is not naturally oriented, the X axis will be chosen to match the main axis of symmetry of the robot body and it will be oriented towards the smallest side, typically the top of a cone for example. In case of a perfectly symmetrical body, the X axis can be chosen arbitrarily but a clear mark should be made visible on the robot body to indicate it.

Z axis oriented in the opposite direction from the gravity. If there is no gravity or natural up/down orientation in the environment or normal operation mode of the robot, the Z axis should be chosen in the direction of the main axis of symmetry in the orthogonal plane defined by the X axis, oriented towards the smallest side. In case of a perfectly symmetrical plane, the Z axis can be chosen arbitrarily but a clear mark should be made visible on the robot body to indicate it.

Y axis oriented to make a right-handed coordinate system.

The axes are oriented in a counter-clockwise direction, as depicted in the illustration above.

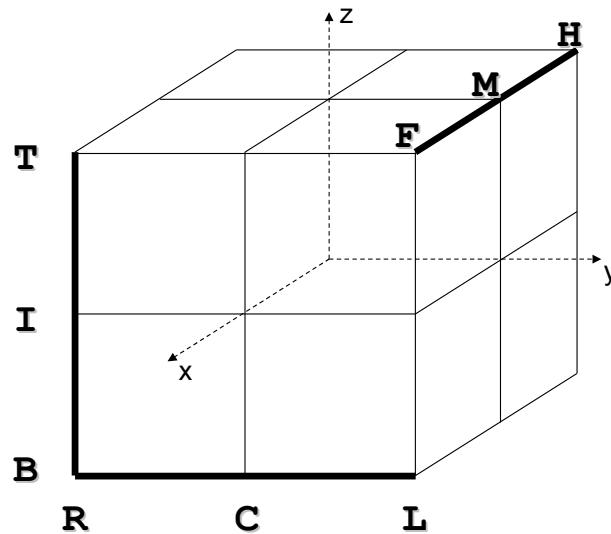
22.3 Component naming

Each component A, which is a sub-component of component B has a name, distinct from the name of all the other components at the same level. This name is a generic designation of what A represents, such as “leg”, “head”, or “finger”.

Using the correct name for each component is a critical part of this standard. No formal rule can be given to find this name for any possible robot configuration. However, this document includes a table covering many different possible cases. We recommend that robot manufacturers pick from this table the name that fits the most the description of their component.

22.4 Localization

When two identical components A1 and A2, such as the two legs of an humanoid robots, are present in the same sub-component B, an extra node is inserted in the hierarchy to differentiate them. This node is of the **Localizer** type, and provides a `[]` operator, taking a **Localization** argument, used to access each of the identical sub-components. The Urbi SDK provides an implementation for the **Localizer** and **Localization** classes. When possible, localization should be simple geometrical qualifier like *right/center/left*, *front/middle/back* or *up/in-between/down*. Note that “right” or “front” are understood here from the point of view of a man standing and looking in the direction of the X-axis of the robot, and *up/down* matches the Z-axis, as depicted in the figure below:



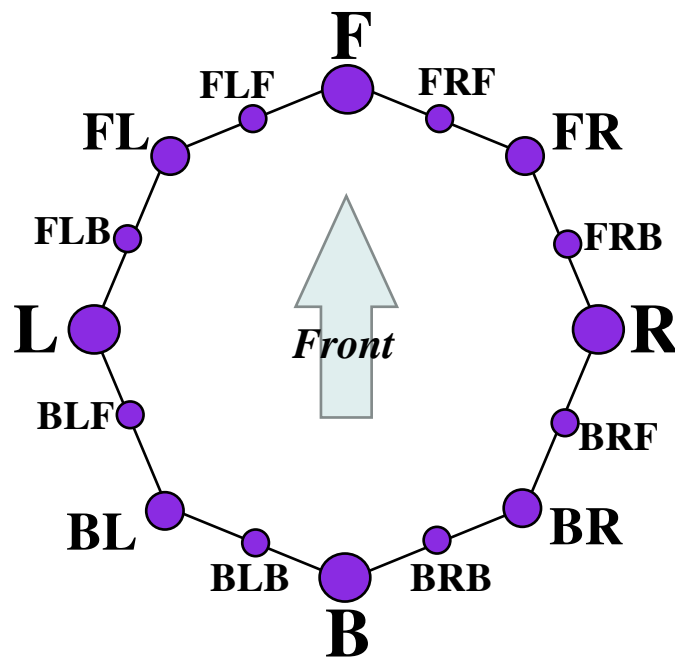
Several geometric qualifiers can be used at the same time to further refine the position. In this case, multiple Localizer nodes are used. As a convention, height information (U/I/D) comes first, followed by depth information (F/M/H), and then side information (R/C/L).

```
// Front-left wheel of a four-wheeled robot:
robot.body.wheel[front][left];
// Front laser of a robot equipped with multiple sonars:
```



```
robot.body.laser[front];
// Left camera from a robot with a stereo camera at the end of an arm:
robot.body.arm.camera[left];
// Top-left LED of the left eye.
robot.body.head.eye[left].led[up][left].val = 1;
// Touch sensor at the end of the front-left leg of a four-legged robot:
robot.body.leg[front][left].foot.touch;
```

You can further qualify a side+depth localization with an additional F/B side information. This can be used in the typical layout below:



This dual positioning using side+depth can also be used to combine side+height or height+depth information.

Layouts with a sequence of three or more identical components can use numbers as their Localization, starting from 0. The smaller the number, the closer to the front, up, or left. For instance, an insectoid robot with 3 legs on each side will use `robot.body.leg[left][0]` to address the frontleft leg.

Layouts with identical components arranged in a circle can also use numeric localization. The component with index 0 should be the uppermost, or front-most if non applicable. Index should increase counterclockwise.

Some components like spines or tails are highly articulated with a set of identical sub-components. When talking about these sub-components, the above localization should be replaced by an array with a numbering starting at 0. The smaller the number, the closer the

sub-component is to the robot main body. For surface-like sub-components, like skin touch sensors, the array can be two dimensional.

Other possible localization for sensors are the X, Y and Z axis themselves, like for example for an accelerometer or a gyro sensor, available in each of the three directions.

```
robot.body.accel[x]; // accelerometer in the x direction
```

Examples of component names including localization:

```
leg[right], leg[left];
finger[right], finger[center], finger[left]; // three-finger hand
joint[0], joint[1] ... joint[5] // from tail
touch[478][124] // from skin
accel[x], accel[y], accel[z]; // typical accelerometer
gyro[x], gyro[y], gyro[z]; // typical gyro sensor
```

22.5 Interface

An “interface” describes some aspects of a type of device, by specifying the slots and methods that implementations must provide. Each child node of the component hierarchy should implement at least one interface.

For example, for a joint, we can have a “Swivel” interface, used to define patella joints. For the robot body itself, we have a “Mobile” interface describing mobile robots, which includes some standard way of requesting a move forward, a turn, etc.

In short, interfaces are standard Urbi objects that components can inherit from to declare that they have some functionalities.

The following pages describe a few of the most standard interfaces. Each device in the component hierarchy which falls within the category of an interface should implement it.

Each interface defines slots, which can be functions, events or plain data. Some of those slots are optional: they are grayed and put around square brackets in the following table.

22.5.1 Identity

Contains information about the robot identity.

- **type**
This describes the robot category among: humanoid, four-legged, wheeled, industrial arm. It gives a general idea of the robot family, but does not replace a more systematic probe of available services by investigating the list of attributes of the object.
- **name**
Name of the robot.
- **model**
Model of the robot.
- **serial**
Serial number (if available).

22.5.2 Network

Contains information about the network identification of the robot.

- **IP**
IP address of the robot.

22.5.3 Motor

This interface is used to describe a generic motor controller.

- **val**
This slot is a generic pointer to a more specific slot describing the motor position, like **position** or **angle**, depending on the type of motor. It is mandatory in the Urbi Ready standard as a universal proxy to control an actuator. The more specific slot is described in a subclass of **Motor**.
- **PGain?**
Controls the P gain of the PID controller.
- **IGain?**
Controls the I gain of the PID controller.
- **DGain?**
Controls the D gain of the PID controller.

22.5.4 LinearMotor (subclass of Motor)

This interface is used to describe a linear motor controller.

A wheel can fall in this category, if the reported position is the distance traveled by a point at the surface of the wheel.

- **position**
Position of the motor in meters. Pointed to by the **val** slot.
- **force**
Intensity of the measured or estimated force applied on a linear motor.

22.5.5 LinearSpeedMotor (subclass of Motor)

Motor similar to **LinearMotor**, but controlled by its translation speed.

- **speed**
Translation speed in meters per second. Pointed to by the **val** slot.

22.5.6 RotationalMotor (subclass of Motor)

This interface is used to describe a position-controlled rotational motor controller.

- **angle**
Angle of the motor in radian, modulo 2π . Pointed to by the **val** slot.
- **turn**
Absolute angular position of the motor, expressed in number of turns.
- **torque**
Intensity of the measured or estimated torque applied on the motor.

22.5.7 RotationalSpeedMotor (subclass of Motor)

Interface describing a motor similar to **RotationalMotor** controlled by its rotation speed.

- **speed**
Rotation speed in radians per second.

22.5.8 Sensor

This interface is used to describe a generic sensor.

- **val**
This slot is a generic pointer to a more specific slot describing the sensor value, like **distance** or **temperature**, depending on the type of sensor. It is mandatory in the Urbi Ready standard as a universal proxy to read a sensor. The more specific slot is described in a subclass of **Sensor**.

22.5.9 DistanceSensor (subclass of Sensor)

This interface is used to describe a distance sensor (infrared, laser, ultrasonic...).

- **distance**
Measured distance expressed in meters. Pointed to by the **val** slot.

22.5.10 TouchSensor (subclass of Sensor)

This interface is used to describe a touch pressure sensor (contact, induction,...).

- **pressure**
Intensity of the pressure put on the touch sensor. Can be 0/1 for simple buttons or expressed in Pascal units. Pointed to by the **val** slot.

22.5.11 AccelerationSensor (subclass of Sensor)

This interface is used to describe an accelerometer.

- **acceleration**
Acceleration expressed in m/s^2 . Pointed to by the **val** slot.

22.5.12 GyroSensor (subclass of Sensor)

This interface is used to describe an gyrometer.

- **speed**
Rotational speed in rad/s . Pointed to by the **val** slot.

22.5.13 TemperatureSensor (subclass of Sensor)

This interface is used to describe a temperature sensor.

- **temperature**
Measured temperature in Celsius degrees. Pointed to by the **val** slot.

22.5.14 Laser (subclass of Sensor)

Interface for a scanning laser rangefinder, or other similar technologies.

- **val**
Last scan result. Can be either a list of ufloat, or a binary containing a packed array of doubles.
- **angleMin**
Start scan angle in radians, relative to the front of the device.
- **angleMax**
End scan angle in radians, relative to the front of the device.
- **resolution**
Angular resolution of the scan, in radians.
- **distanceMin**
Minimum measurable distance.
- **distanceMax**
Maximum measurable distance.
- **rate?**
Number of scans per second.

Depending on the implementation, some of the parameters can be read-only, or can only accept a few possible values. In that case it is up to the implementer to select the closest possible value to what the user entered. It is the responsibility of the user to read the parameter after setting it to check what value will actually be used by the implementation.

22.5.15 Mobile

Mobile robots all share this generic interface to provide high order level motion control capabilities.

- `go(x)`
Move approximately *x* meters forward if *x* is positive, backward otherwise.
- `turn(x)`
Turn right approximately *x* radians. *x* can be a positive or negative value.

22.5.16 Tracker

Camera-equipped robots can sometimes move the orientation of the field of view horizontally and vertically, which is a very important feature for many applications. In that case, this interface abstracts how such motion can be achieved, whether it is done with a pan/tilt camera or with whole body motion or a combination of both.

- `yaw`
Rotational articulation around the Z axis in the robot, expressed in radians.
- `pitch`
Rotational articulation around the Y axis in the robot, expressed in radians.

22.5.17 VideoIn

The VideoIn interface groups every information relative to cameras or any image sensor.

- `val`
Image represented as a `Binary` value.
- `xfov`
The x field of view of the camera expressed in radians.
- `yfov`
The y field of view of the camera expressed in radians.
- `height`
Height of the image in the current resolution, expressed in pixels.
- `width`
Width of the image in the current resolution, expressed in pixels
- `format?`
Format of the image, expressed as an integer in the enum `urbi::UImageFormat`. See below for more information.

- **exposure?**
Exposure duration, expressed in seconds. 0 if non applicable.
- **wb?**
White balance (expressed with an integer value depending on the camera documentation). 0 if non applicable.
- **gain?**
Camera gain amplification (expressed as a coefficient between 0 and infinity). 1 if non applicable.
- **resolution?**
Image resolution, expressed as an integer. 0 corresponds to the maximal resolution of the camera. Successive values correspond to all the supported image sizes in decreasing order. Once modified, the effective resolution in X/Y can be checked with the width and height slots.
- **quality?**
If the image is in the jpeg format, this slot sets the compression quality, from 0 (best compression, worst quality) to 100 (best quality, bigger image).

The image sensor is expected to use the cheapest way in term of CPU and/or energy consumption to produce images of the requested format. Implementations linked to a physical image sensor do not have to implement all the possible formats. In this case, the format closest to what was requested must be used. A generic image conversion object will be provided. In order to avoid duplicate image conversions when multiple unrelated behaviors need the same format, it is recommended that this object be instantiated in a slot of the VideoIn object named after the format it converts to:

```
if (!robot.body.head.camera.hasSlot("jpeg"))
{
    var robot.body.head.camera.jpeg =
        ImageConversion.new(robot.body.head.camera.getSlot("val"));
    robot.body.head.camera.jpeg.format = 3;
}
```

22.5.18 AudioOut

The AudioOut interface groups every information relative to speakers.

- **val**
The speaker value, expressed as a binary, in the format given by the binary header during the assignment.

Speakers are write-only devices, so there is not much sense in reading the content of this attribute. At best, it returns the remaining sound to be played if it is not over yet, but this is not a requirement.

- **remain**
The amount of time remaining to play in the speaker sound buffer (expressed in *ms* as a default unit).
- **playing**
This is a Boolean value which is true when there is a sound currently playing (the buffer is not empty)
- **volume?**
Volume of the play back, in decibels.

22.5.19 AudioIn

The AudioIn interface groups every information relative to microphones.

- **val**
Binary value corresponding to the sound heard, expressed in the current unit (wav, mp3...).
The unit can be changed like any other regular unit in Urbi.
The content is the sound heard by the microphone since the last update event.
- **duration**
Amount of sound in the val attribute, expressed in *ms*.
- **gain?**
Microphone gain amplification (expressed between 0 and 1).

22.5.20 BlobDetector

Ball detectors, marker detectors and various feature-based detectors should all share a similar interface. They extract a part of the image that fits some criteria and define a *blob* accordingly. Here are the typical slots expected:

- **x**
The x position of the center of the blob in the image
- **y**
The y position of the center of the blob in the image
- **ratio**
The size of the blob expressed as a normalized image size: 1 = full image, 0 = nothing.
- **visible**
A Boolean expressing whether there is a blob in the image or not (see [threshold](#)).
- **threshold**
The minimum value of ratio to decide that the blob is visible.

- **orientation?**
Angle of the main ellipsoid axis of the blob (0 = horizontal), expressed in radians.
- **elongation?**
Ratio between the main and the second diameter of the blob enveloping ellipse.

22.5.21 TextToSpeech

Text to speech allows to read text using a speech synthesizer. Default implementations should use the **speaker** component (or alias) as their default sound output.

- **lang?**
The language used, in international notation (fr, en, it...): ISO 639
- **speed?**
How fast the voice should go. A positive number, with 1 standing for “regular speed”.
- **pitch?**
Voice pitch. A positive number, with 1 standing the regular pitch.
- **gender?**
Gender of the speaker (0:male/1:female).
- **age?**
Age of the speaker, if applicable.
- **voice?**
Most TTS engines propose several voices, this attribute allows picking one. It’s a string identifier specific to the TTS developer.
- **say(s)**
Speak the sentence given in parameter *s*.
- **voicexml?(s)**
Speak the text *s* expressed as a VoiceXML string.
- **script?(s)**
Speak the text *s* augmented by script markups (see specific Gostai documentation) to generate Urbi events.

22.5.22 SpeechRecognizer

Speech recognition allows to transform a stream of sound into a text using various speech recognition algorithms. Implementations should use the **micro** component as their default sound input.

- **lang?**
The language used, in international notation (fr, en, it...): ISO 639.

- **hear(s)**

This event has one parameter which is the string describing what the speech engine has recognized (can be a word or a sentence).

22.5.23 Led

Simple uni-color Led.

- **val**

Led intensity between 0 and 1.

22.5.24 RGBLed (subclass of Led)

Tri-color led.

- **r**

Intensity of the red component, between 0 and 1.

- **g**

Intensity of the green component, between 0 and 1.

- **b**

Intensity of the blue component, between 0 and 1.

22.5.25 Battery

Power source of any kind.

- **remain**

Estimation of the remaining energy between 0 and 1.

- **capacity**

Storage capacity in Amp.Hour.

- **current**

Current current consumption in Amp.

- **voltage**

Current voltage in Volt.

22.6 Standard Components

Standard components correspond to components typically found in wheeled robots, humanoid or animaloid robots, or in industrial arms. This section provides a list of such components. Whenever possible, robots compatible with the Gostai Standard Robotics API should name all the components in the hierarchy using this list.

22.6.1 Yaw/Pitch/Roll orientation

Note that Gostai Standard Robotics API considers the orientation to be a component name, and not a localizer. So one would write `head.yaw` and not `head.joint[yaw]` to refer to a rotational articulation of the head around the Z axis.

It is not always clear which rotational direction corresponds to the yaw, pitch or roll components (listed in the table below). This is a quick guideline to help determine the proper association.

Let us consider the robot in its resting, most prototypical position, like “standing” on two or four legs for a humanoid or animaloid, and let all members “naturally” fall under gravity. When gravity has no effect on a certain joint (because it is in the orthogonal plan to Z, for example), the medium position between `rangemin` and `rangemax` should be used. The body position achieved will be considered as a reference. Then for each component that is described in terms of yaw/pitch/roll sub-decomposition, the association will be as follow:

yaw rotational articulation around the Z axis in the robot.

pitch rotational articulation around the Y axis in the robot.

roll rotational articulation around the X axis in the robot.

When there is no exact match with the X/Y/Z axis, the closest match, or the default remaining available axis, should be selected to determine the yaw/pitch/roll meaning.

22.6.2 Standard Component List

The following table summarizes the currently referenced standard components, with a description of potential components that they could be sub-component of, a description of potential components they may contain, and a list of relevant interfaces. This table should be seen as a quick reference guide to identify available components in a given robot.

Name	Description	Sub. of	Contains	Facets
robot	This is the main component that represents an abstraction of the robot, and the root node of the whole component hierarchy.		body	Identity Network Mobile Tracker
body	This is the main component that contains every piece of hardware in the robot. This includes all primary kinematics sub-chains (arms, legs, neck, head, etc) and non-localized sensor arrays, typically body skin or heat detectors. Localized sensors, like fingertips touch sensors, will typically be found attached to the finger component they belong and not directly to the body.	robot	wheel arm leg neck head wheel tail skin torso ...	

Name	Description	Sub. of	Contains	Facets
wheel	Wheel attached to its parent component.	body		Rotational-Motor
leg	Legs are found in humanoid or animaloid robots and correspond to part of the kinematics chain that are attached to the main body by one extremity only and which do touch the ground in normal operation mode (unlike arms). A typical configuration for humanoids contains a hip, a knee and an ankle. If the leg is more segmented, the leg can be described with a simple array of joints.	body	hip knee ankle foot joint	
arm	Unlike legs, an arm's extremity does not always touch the ground in normal operating mode. This applies to humanoid robots or single-arm industrial robots. Arms supersede legs in the nomenclature: if a body part behaves alternatively like an arm and like a leg, it will be considered as an arm.	body	shoulder elbow wrist hand grip joint	
shoulder	The shoulder is the upper part of the arm. It can have one, two or three degrees of freedom and is the closest part of the arm relative to the body.	arm	yaw pitch roll	
elbow	Separates the upper arm and the lower arm, this is usually a single rotational axis.	arm	pitch	
wrist	Connects the hand and the lower part of the arm. Usually three degrees of freedom axis.	arm	yaw pitch roll	
hand	The hand is an extension of the arm that usually holds fingers. It's not the wrist, which is articulated and between the arm and the hand.	arm	finger	
finger	Fingers are a series of articulated motors at the extremity of the arm, and connected to the hand. They are usually localized with arrays and/or lateral localization respective to the hand.	hand	touch	Motor

Name	Description	Sub. of	Contains	Facets
grip	Simple two-fingers system.	arm hand	touch	Motor
hip	The hip is the upper part of the leg and connects it to the main body. It can have one, two or three degrees of freedom.	leg	yaw pitch roll	
knee	Separates the upper leg and the lower leg, this is usually a single rotational axis.	leg	pitch	
ankle	Connects the foot and the lower part of the leg. Usually three degrees of freedom axis.	leg	yaw pitch roll	
foot	The foot is an extension of the leg that usually holds toes. It's not the ankle, which is articulated and between the leg and the foot. The foot can also contain touch sensors in simple configurations.	leg	touch	
toe	Like fingers, but attached to the foot.	foot	touch	Motor
neck	The neck corresponds to a degree of freedom not part of the head, but relative to the rigid connection between the head and the main body.	body	yaw pitch roll	
tail	A tail is a series of articulated motors at the back of the robot.	body	joint	
head	The head main pivotal axis.	body neck	camera mouth ear lip eye eyebrow	
mouth	The robot mouth (open/close)	head	lip	Motor
ear	Ears may have degrees of freedom in certain robots.	head		Motor
joint	Generic articulation in the robot.	tail arm leg lip		Motor
yaw	Rotational articulation around the Z axis in the robot. See Section 22.6.1 .	body neck knee ankle shoulder elbow wrist torso		Rotational- Motor Rotational- Speed- Motor

Name	Description	Sub. of	Contains	Facets
pitch	Rotational articulation around the Y axis in the robot. See Section 22.6.1 .	body neck knee ankle shoulder elbow wrist torso		Rotational- Motor Rotational- Speed- Motor
roll	Rotational articulation around the X axis in the robot. See Section 22.6.1 .	body neck knee ankle shoulder elbow wrist torso		Rotational- Motor Rotational- Speed- Motor
x	Translational movement along the X axis.	body arm		Linear- Motor Linear- Speed- Motor
y	Translational movement along the Y axis.	body arm		Linear- Motor Linear- Speed- Motor
z	Translational movement along the Z axis.	body arm		Linear- Motor Linear- Speed- Motor
lip	Corresponds to animated lips.	mouth	joint	Motor
eye	Corresponds to the eyeball pivotal axis.	head	camera	
eyebrow	Some robots will have eyebrows with generally one or several degrees of freedom.	head	joint	Motor
torso	This corresponds to a pivotal or rotational axis in the middle of the main body.	body	yaw pitch roll	
spine	This is a more elaborated version of “torso”, with a series of articulations to give several degrees of freedom in the back of the robot.	torso	joint	

Name	Description	Sub. of	Contains	Facets
clavicle	This is not to be mixed up with the “top of the arm” body part. It is an independent degree of freedom that can be used to bring the two arms closer in a sort of “shoulder raising” movement.	body		Motor
touch	Touch sensor.	finger grip foot toe		TouchSensor
gyro	Gyrometer sensor.	body		GyroSensor
accel	Accelerometer sensor.	body		Accel- eration- Sensor
camera	Camera sensor. If several cameras are available, localization shall apply; however there must always be an alias from camera to one of the effective cameras (like cameraR or cameraL).	head body		VideoIn
speaker	Speaker device. If several speakers are available, localization shall apply; however there must always be an alias from speaker to one of the effective speakers (like speakerR or speakerL).	head body		AudioIn
micro	Microphone devices. If several microphones are available, localization shall apply; however there must always be an alias from micro to one of the effective microphones (like microR or microL).	head body		AudioOut
speech	Speech recognition component.	robot		Speech- Recognizer
voice	Voice synthesis component.	robot		TextTo- Speech

22.7 Compact notation

Components are usually identified with their full-length name, which is the path to access them inside the structure tree. For convenience and backward compatibility with pre-2.0 versions of Urbi, there is also a compact notation available. We will describe here how to construct the compact notation starting from the full name and the structure tree.

Full name	Compact name
robot.body.armR.elbow	elbowR
robot.body.head.yaw	headYaw
robot.body.legL.knee.pitch	kneeL
robot.body.armR.hand.finger[3][2]	fingerR[3][2]

Full name	Compact name
<code>robot.body.armL.hand.fingerR</code>	<code>fingerLR</code>

The rule is to move every localization qualifier at the end of the compact notation, in the order where they appear in the full-length name. The remaining component names should then be considered one by one to see if they are needed to remove ambiguities. If they are not, like typically the robot or body components which are shared with almost every other full-length name, they can be ignored. If finally several component names have to be kept, they should be separated by using upper case letters for the first character instead of a dot, like in Java-style notation.

Example 1 (`robot.body.armL.hand.fingerR`)

1. Move all localization at the end: *robot.body.arm.hand.fingerLR*
2. The full name remaining is: *robot.body.arm.hand.finger*
3. *finger* should be kept, *hand*, *arm*, *body* and *robot* are not necessary since every finger component will always be attached only to a hand, itself attached to an arm and a body and a robot.
4. The result is *fingerLR*

Example 2 (`robot.body.head.yaw`)

1. No localization to move
2. *yaw* must be kept because *head* also have a *pitch* sub-component and
3. *head* must also be kept to avoid ambiguity with other components having a *yaw* sub-component.
4. The result is *headYaw*

Example 3 (`robot.body.legL.knee.pitch`)

1. Move all localization at the end: *robot.body.leg.knee.pitchL*
2. *pitch* is not necessary because *knee* has only a *pitch*, so *knee* will be kept only
3. The result is *kneeL*

22.8 Support classes

The Urbi SDK provides a few support urbiscript classes to help you build the component hierarchy. You can access to those classes by including the files ‘`urbi/naming-standard.u`’ and ‘`urbi/component.h`’.

22.8.1 Interface

The **Interface** class contains urbiscript objects for all the interfaces defined in this document. Implementations must inherit from the correct interface.

```
// Instantiate a camera.
var cam = myCamera.new();
// Make it inherit from VideoIn.
cam.addProto(Interface.VideoIn);
```

The `Interface.interfaces` method can be called to get a list of all the interfaces an object implements.

22.8.2 Component

The **Component** class can be used to create intermediate nodes of the hierarchy. It provides the following methods:

- `addComponent(name)`
Add a new sub-component to the current component. *name* can be the name of the new component to create, or an instance of **Component**.
- `addDevice(name, value)`
Add device *value* as sub-component, under the name *name*. The device must inherit from at least one **Interface**.
- `makeCompactNames`
This function must be called once on the root node (**robot**) after the hierarchy is completed. It automatically computes the short name of all the devices, and insert them as slots of the **Global** object.
- `dump`
Display a hierarchical view of the component hierarchy.
- `flatDump`
Display all the devices in the hierarchy, sorted by the **Interface** they implement.

22.8.3 Localizer

The **Localizer** class is a special type of **Component** that stores other components based on their localization. It provides a `[]` operator that takes a **Localization**, such as `top`, `left`, `front`, and that can be used to set and get the **Component** or device associated with that **Localization**.

Note that the `[]` function is using a mechanisms to automatically look for its argument as a slot of **Localizer**. As a consequence, you cannot pass a variable to this function, but only one of the constant **Localization**. To pass a variable, use the `get(loc)` or the `set(loc, value)` function.

The following example illustrates a typical instantiation sequence:

```
// Create the top-level node.
var Global.robot = Component.new("robot");
robot.addComponent("head");
var cam = MyCamera.new;
cam.addProto(Interface.VideoIn);
robot.head.addDevice("camera", cam);
// Add two wheels
robot.addComponent(Localizer.new("wheel"));
robot.wheel[left] = MyWheel.new(0).addProto(Interface.RotationalMotor);
robot.wheel[right] = MyWheel.new(1).addProto(Interface.RotationalMotor);
// Implement the Mobile facet in urbiscript:
var robot.go = function(d)
{
  robot.wheel.val = robot.wheel.val + d / wheelRadius adaptive:1
};
var robot.turn = function(r)
{
  var v = r * wheelDistance / wheelRadius;
  robot.wheel[left].val = robot.wheel[left].val + v adaptive:1 &
  robot.wheel[right].val = robot.wheel[right].val - v adaptive:1
};
robot.addProto(Interface.Mobile);
robot.makeCompactNames;
// Let us see the result:
robot.flatDump;
[00010130] *** Mobile: robot
[00010130] *** RotationalMotor: wheelL wheelR
[00010130] *** VideoIn: camera
```

Part V

Tables and Indexes

About This Part

This part contains material about the document itself.

Chapter 23 — Notations

Conventions used in the type-setting of this document.

Chapter 25 — Licenses

Licenses of components used in Urbi SDK.

Chapter 24 — Release Notes

Release notes of Urbi SDK.

Chapter 26 — Glossary

Definition of the terms used in this document.

Chapter 23

Notations

This chapter introduces the various notations that are used in the document.

23.1 Words

- *name*
Depending on the context, a *variable* name (i.e., an identifier in C++ or urbiscript), or a *meta-variable* name. A meta-variable denotes a place where some syntactic construct may be entered. For instance, in `while (expression) statement`, *expression* and *statement* do not denote two variable names, but two placeholders which can be filled with an arbitrary expression, and an arbitrary statement. For instance:
`while (!tasks.empty) { tasks.removeFront.process }.`
- *name*
An environment variable name, e.g., `PATH`.
- *code*
A piece of urbiscript or C++ code.
- *'name'*
A file name.

23.2 Frames

23.2.1 Shell Sessions

Interactive sessions with a (Unix) shell are represented as follows.

```
$ echo toto
toto
```

The user entered `'echo toto'`, and the system answered `'toto'`. `'$'` is the *prompt*: it starts the lines where the system invites the user to enter her commands.

23.2.2 urbiscript Sessions

Interactive sessions with Urbi are represented as follows.

```
echo("toto");
[00000001] *** toto
```

Contrary to shell interaction (see [Section 23.2.1](#)), there is no prompt that marks the user-entered lines (here `echo("toto")`); but, on the contrary, answers from the Urbi server start with a label that includes a timestamp (here '00000001'), and possibly a channel name, 'output' in the following example.

```
cout << "toto";
[00000002:output] "toto"
```

23.2.3 urbiscript Assertions

urbiscript features assertions blocks, see [Section 19.8](#). The following assertion frame:

```
1 == 2 / 2;
1 < 2;
true;
"foobar"[0, 3] == "foo";
[1, 2, 3].map (function (a) { a * a }) == [1, 4, 9];
[ => ].empty;
```

actually denotes the following assertion-block in an urbiscript-session frame:

```
assert
{
  true;
  1 < 2;
  1 + 2 * 3 == 7;
  "foobar"[0, 3] == "foo";
  [1, 2, 3].map (function (a) { a * a }) == [1, 4, 9];
  [ => ].empty;
};
```

23.2.4 C++ Code

C++ source code is presented in frames as follows.

```
class Int
{
public:
  Foo(int v = 0)
    : val_(v)
  {}

  void operator(int v)
  {
    std::swap(v, val_);
```



```
    return v;
}

int operator() const
{
    return val_;
}

private:
    int val_;
};
```


Chapter 24

Release Notes

This chapter lists the user-visible changes in Urbi SDK releases.

24.1 Urbi SDK 2.1

Released on 2010-07-08.

24.1.1 Fixes

- [Lobby.connectionTag](#) monitors the jobs launched from the lobby, but can no longer kill the lobby itself.
- ‘123foo’ is no longer accepted as a synonym to ‘123 foo’. As a consequence, in case you were using `x = 123cos: 1`, convert it to `x = 123 cos: 1`.
- Some old tools that no longer make sense in Urbi SDK 2.0 have been removed: `umake-engine`, `umake-fullengine`, `umake-lib`, `umake-remote`. Instead, use `umake`, see [Section 18.7](#).
- On Windows `urbi-launch` could possibly miss module files to load if the extension (‘.dll’) was not specified. One may now safely, run ‘`urbi-launch my-module`’ (instead of ‘`urbi-launch my-module.dll`’ or ‘`urbi-launch my-module.so`’) on all the platforms.

24.1.2 New Features

- [Regex.asPrintable](#), [Regex.asString](#), [Regex.has](#).
- [System.Platform.host](#), [System.Platform.hostAlias](#), [System.Platform.hostCpu](#), [System.Platform.hostVendor](#).
- `UObject` `init` method and methods bound by `notifyChange` no longer need to return an `int`.
- [Channel.Filter](#), a [Channel](#) (`??sec:std-Channel`) that outputs text that can be parsed without error by the `liburbi`.

- `RangeIterable.all`, `RangeIterable.any`, moved from `List` ([??sec:std-List](#)).
- Support for `ROS`, the Robot Operating System. See [Chapter 13](#) for an introduction, and [Chapter 21](#) for the details.
- `Lobby.lobby` and `Lobby.instances`, bounced to from `System.lobby` and `System.lobbies`.
- `Tag.scope`, bounced to from `System.scopeTag`.

24.1.3 Optimization

- The main loop was reorganized to factor all socket polling in a single place: latency of `Socket` ([??sec:std-Socket](#)) is greatly reduced.

24.1.4 Documentation

- `Lobby.authors`, `Lobby.thanks`.
- `System.PackageInfo` ([??sec:std-System.PackageInfo](#)).
- `System.spawn`.
- LEGO Mindstorms NXT support ([??](#)).
- Pioneer D-3X support ([??](#)).
- Support for Segway RMP ([??](#)).

24.2 Urbi SDK 2.0.3

Released on 2010-05-28.

24.2.1 New Features

- `Container` ([??sec:std-Container](#)), prototype for `Dictionary` ([??sec:std-Dictionary](#)), `List` ([??sec:std-List](#)) derive.
- `e not in c` is mapped onto `c.hasNot(e)` instead of `!c.has(e)`.
- `Float.limits` ([??sec:std-Float.limits](#))
- `Job.asString`
- `IoService` ([??sec:std-IoService](#))
- `Event.<<<`
- `List.argmax`, `List.argmin`, `List.zip`

- `Tuple.'``+``'`
- `Tuple.'``*``'`
- Assertion failures are more legible:

```
var one = 1|;
var two = 2|;
assert (one == two);
[00000002:error] !!! failed assertion: one == two (1 != 2)
```

instead of

```
assert (one == two);
[00000002:error] !!! failed assertion: one.'=='(two)
```

previously. As a consequence, `System.assert_op` is deprecated. The never documented following slots have been removed from `System` ([??sec:std-System](#)): `assert_eq`, `assert_ge`, `assert_gt`, `assert_le`, `assert_lt`, `assert_meq`, `assert_mne`, `assert_ne`.

24.2.2 Fixes

- `List.'``<``'` and `Tuple.'``<``'` implement true lexicographic order: `[0, 4] < [1, 3]` is true. List comparison used to implement member-wise comparison; the previous assertion was not verified because `4 < 3` is not true.
- `Mutex.asMutex` is fixed.
- `Directory` ([??sec:std-Directory](#)) events were not launched if a `Directory` ([??sec:std-Directory](#)) had already been created on the same `Path` ([??sec:std-Path](#)).
- `waituntil` no longer ignores pattern guards.

24.2.3 Documentation

- `Bioid` ([??](#)).
- Garbage collection ([Section 19.11](#)).
- Structural Pattern matching ([Section 19.5](#)).
- `CallMessage.sender` and `CallMessage.target`.
- `Dictionary.asString`.
- `Directory.fileCreated` and `Directory.fileDeleted`.
- `List.max`, `List.min`.
- `Mutex.asMutex`.
- `Object.localSlotNames`.

24.3 Urbi SDK 2.0.2

Released on 2010-05-06.

24.3.1 urbiscript

- `Control.detach` and `Control.disown` return the `Job` (`??sec:std-Job`).

24.3.2 Fixes

- ‘make install’ failures are addressed.
- `freezeif` can be used more than once inside a scope.

24.3.3 Documentation

- `StackFrame` (`??sec:std-StackFrame`)
- `String.split`

24.4 Urbi SDK 2.0.1

Released on 2010-05-03.

24.4.1 urbiscript

- Minor bug fixes.
- The short option ‘-v’ is reserved for ‘--verbose’. Tools that mistakenly used ‘-V’ for ‘--verbose’ and ‘-v’ for ‘--version’ have been corrected (short options are swapped). Use long options in scripts, not short options.
- `Lobby.echoEach`: new.
- `String.closest`: new.
- `Tuple.size`: new.

24.4.2 Documentation

- How to build Urbi SDK ([Chapter 17](#)).
- Hyperlinks to slots (e.g., `Float.asString`).

24.4.2.1 Fixes

- Closures enclose the lobby. Now slots of the lobby in which the closure has been defined are visible in functions called from the closure.

24.5 Urbi SDK 2.0

Released on 2010-04-09.

24.5.1 urbiscript

24.5.1.1 Changes

- `GlobalTags` is renamed as `Tag.tags`.
- `GlobalTask` is renamed as `Global.Job`.
- `GlobaltopLevel` is renamed as `Channel.topLevel`.
- `Globaloutput`, `Globalerror` are removed, they were deprecated in favor of `Global.cout` and `Global.cerr`.
- `Object.getPeriod` is deprecated in favor of `System.period`.
- As announced long ago, and as displayed by warnings, `Object.slotNames` now returns all the slot names, ancestors included. Use `Object.localSlotNames` to get the list of the names of the slot the object owns.
- `Semaphore.acquire` and `Semaphore.release` are promoted over `Semaphore.p` and `Semaphore.v`.

24.5.1.2 New features

- Dictionary can now be created with literals.

Syntax	Semantics
[<code>=></code>]	<code>Dictionary.new</code>
[<code>"a"</code> <code>=></code> 1, <code>"b"</code> <code>=></code> 2, <code>"c"</code> <code>=></code> 3]	<code>Dictionary.new("a", 1, "b", 2, "c", 3)</code>

- `Float.srandom`
- `List.subset`
- `Object.getLocalSlot`.
- String escapes accept one- and two-digit octal numbers. For instance `"\0"`, `"\00"` and `"\000"` all denote the same value.
- Tuple can now be created with literals.

Syntax	Semantics
<code>()</code>	<code>Tuple.new([])</code>
<code>(1,)</code>	<code>Tuple.new([1])</code>
<code>(1, 2, 3)</code>	<code>Tuple.new([1, 2, 3])</code>

- `Location.'=='`.
- `type` replaces `'$type'`

24.5.2 UObjects

- Remote timers (`USetUpdate`, `USetTimer`) are now handled locally instead of by the kernel.
- `UVars` can be copied using the `UVar.copy` method.
- New `UEvent` class, similar to `UVar`. Can be used to emit events.
- Added support for dictionaries: new `UDictionary` structure in the `UValue` union.

24.5.3 Documentation

- [Barrier](#) ([??sec:std-Barrier](#))
- [Date.now](#)
- [Float.srandom](#)
- [Pattern](#) ([??sec:std-Pattern](#))
- [PubSub](#) ([??sec:std-PubSub](#))
- [PubSub.Subscriber](#) ([??sec:std-PubSub.Subscriber](#))
- [Profiling](#) ([??sec:std-Profiling](#))
- [Semaphore](#) ([??sec:std-Semaphore](#))
- Trajectories
- [TrajectoryGenerator](#) ([??sec:std-TrajectoryGenerator](#))
- `urbi-image`
- `waituntil` clauses
- `whenever` clauses

24.6 Urbi SDK 2.0 RC 4

Released on 2010-01-29.

24.6.1 urbiscript

24.6.1.1 Changes

- '\$id' replaces id
- '\$type' replaces type
- List derives from Orderable.

24.6.1.2 New objects

- [Location](#) (`??sec:std-Location`)
- [Position](#) (`??sec:std-Position`)

24.6.1.3 New features

- [File.remove](#)
- [File.rename](#)

24.6.2 UObjects

- The UObject API is now thread-safe: All UVar and UObject operations can be performed from any thread.
- You can request bound functions to be executed asynchronously in a different thread by using `UBindThreadedFunction` instead of `UBindFunction`.

24.7 Urbi SDK 2.0 RC 3

Released on 2010-01-13.

24.7.1 urbiscript

24.7.1.1 Fixes

- `local.u` works as expected.

24.7.1.2 Changes

- [Lobby.quit](#) replaces `Systemquit`.
- [Socket.connect](#) accepts integers
- UObject remote `notifyChange` on `USensor` variable now works as expected.
- UObject timers can now be removed with `UObject::removeTimer()`.

24.7.2 Documentation

- Socket provides a complete example.
- The Naming Standard documents the support classes provided to ease creation of the component hierarchy.

24.8 Urbi SDK 2.0 RC 2

Released on 2009-11-30.

This release candidate includes many fixes and improvements that are not reported below. The following list is by no means exhaustive.

24.8.1 Optimization

The urbiscript engine was considerably optimized in both space and time.

24.8.2 urbiscript

24.8.2.1 New constructs

- `assert { claim1; claim2;... };`
- `every|`
- `break` and `continue` are supported in `every|` loops.
- `for(num)` and `for(var i: set)` support the `for&`, `for|` and `for;` flavors.
- `for(init; cond; inc)` supports the `for|` and `for;` flavors.
- non-empty lists of expressions in list literals, in function calls, and non-empty lists of function formal arguments may end with a trailing optional comma. For instance:

```
function binList(a, b,) { [a, b,] } | binList(1, 2,)
```

is equivalent to

```
function binList(a, b) { [a, b] } | binList(1, 2)
```

- consecutive string literals are joined into a unique string literal, as in C++.

24.8.2.2 New objects

- `Component` (`??sec:std-Component`), `Localizer` (`??sec:std-Localizer`), `Interface` (`??sec:std-Interface`): naming standard infrastructure classes.
- `Date` (`??sec:std-Date`)

- `Directory` (`??sec:std-Directory`)
- `File` (`??sec:std-File`)
- `Finalizable` (`??sec:std-Finalizable`): objects that call `finalize()` when destroyed.
- `InputStream` (`??sec:std-InputStream`)
- `Mutex` (`??sec:std-Mutex`)
- `OutputStream` (`??sec:std-OutputStream`)
- `Process` (`??sec:std-Process`): Start and monitor child processes.
- `Regexp` (`??sec:std-Regexp`)
- `Server` (`??sec:std-Server`): TCP/UDP server socket.
- `Socket` (`??sec:std-Socket`): TCP/UDP client socket.
- `Timeout` (`??sec:std-Timeout`)
- `WeakDictionary`, `WeakPointer`: Store dictionary of objects without increasing their reference count.

24.8.2.3 New features

- `asBool`
- `Lobby.wall`
- `Dictionary.size`
- `Global.evaluate`
- `Group.each`, `Group.each&`
- `Lobby.onDisconnect` (actually, an event), `Lobby.remoteIP` `Lobby.create`, `Lobby.received`, `Lobby.resendBanner`
- `Object.inspect`
- `String.fromAscii`, `String.replace`, `String.toAscii`
- `System`: `_exit`, `assert_eq`, `System.system`, `System.terminate`

24.8.2.4 Fixes

- at constructs do not leak local variables anymore.
- Each tag now has its enter and leave events.
- `File.content` reads the whole file.
- Invalid assignments such as `f(x) = n` are now refused as expected.

24.8.2.5 Deprecations

- `ownsSlotObject` (`??sec:std-ownsSlot`) is deprecated in favor of `hasSlotObject` (`??sec:std-hasSlot`)/`hasLocalSlotObject` (`??sec:std-hasLocalSlot`).
- `slotNamesObject` (`??sec:std-slotNames`) is deprecated in favor of `allSlotNamesObject` (`??sec:std-allSlotNames`)/`localSlotNamesObject` (`??sec:std-localSlotNames`).

24.8.2.6 Changes

- empty strings, dictionaries and lists are now evaluated as "false" in conditions.
- `Dictionary.asString` does not sort the keys.
- `Dictionary.'[]='` returns the assigned value, not the dictionary.
- `Dictionary.'[]'` raises an exception if the key is missing.
- Constants is merged into Math.
- `every` no longer goes in background. Instead of:

```
every (1s) echo("foo");
```

write (note the change in the separator)

```
every (1s) echo("foo"),
```

or

```
detach({ every (1s) echo("foo"); });
```

- Tag: begin and end now simply print the tag name followed by 'begin' or 'end'.
- System-code is now hidden from the backtraces.
- `Code.apply`: the call message can be changed by passing it as an extra argument.

24.8.3 UObjects

- Handle UObject destruction. To remove an UObject, call the urbiscript `destroy` method. The corresponding C++ instance will be deleted.
- Add `UVar::unnotify()`. When called, it removes all `UNotifyChange` registered with the `UVar`.
- Bound functions using `UBindFunction` can now take arguments of type `UVar&` and `UObject*`. The recommended method to pass UVars from urbiscript is now to use `'camera.getSlot("val")'` instead of `'"camera.val"'`.
- Add a 0-copy mode for UVars: If `'UVar::enableBypass(true)'` is called on an `UVar`, `notifyChange` on this `UVar` can recover the not-copied data by using `UVar.get()`, returning an `UValue&`. However, the data is only accessible from within `notifyChange`: reading the `UVar` directly will return `nil`.
- Add support for the `changed!` event on UVars. Code like:

```
at (headTouch.val->changed? if headTouch.val)
  tts.say("ouch");
```

will now work. This hook costs one `at` per `UVar`, and can be disabled by setting `UVar.hookChanged` to false.

- Add a statistics-gathering tool. Enable it by calling `'uobjects.enableStats(true)'`. Reset counters by calling `'uobjects.clearStats'`. `'uobjects.getStats'` will return a dictionary of all bound C++ function called, including timer callbacks, along with the average, min, max call durations, and the number of calls.
- When code registered by a `notifyChange` throws, the exception is intercepted to protect other unrelated callbacks. The throwing callback gets removed from the callback list, unless the `removeThrowingCallbacks` on the `UVar` is false.
- the environment variable `URBI_UOBJECT_PATH` is used by `urbi-launch` and `urbiscript`'s `load-Module` to find `uobjects`.
- fixed multiple notifications of event trigger in remote UObjects.
- Many other bug fixes and performance improvements.
- an exception is now thrown if the C++ `init` method failed.

24.8.4 Documentation

The documentation was fixed, completed, and extended. Its layout was also improved. Changes include, but are not limited to:

- various programs: `urbi`, `urbi-launch`, `urbi-send` etc. ([Chapter 18](#)).

- environment variables: `URBI_UOBJECT_PATH`, `URBI_PATH`, `URBI_ROOT` ([Section 18.1](#)).
- special files `'global.u'`, `'local.u'` ([Section 18.2](#)).
- k1-to-k2: Conversion idioms from urbiscript 1 to urbiscript 2 ([Chapter 16](#)).
- FAQ ([Chapter 15](#))
 - stack exhaustion
 - at and waituntil: performance considerations
- Specifications:
 - completion of the definition of the control flow constructs ([every](#), [everyl](#), [if](#), [for](#), [loop](#))
 - tools (`umake`, `umake-shared`, `umake-deepclean`, `urbi`, `urbi-launch`, `urbi-send`).
 - [Boolean](#) (`??sec:std-Boolean`)
 - [Channel](#) (`??sec:std-Channel`)
 - [Date](#) (`??sec:std-Date`)
 - [Dictionary](#) (`??sec:std-Dictionary`)
 - [Exception](#) (`??sec:std-Exception`)
 - [File](#) (`??sec:std-File`)
 - [Kernel1](#) (`??sec:std-Kernel1`)
 - [InputStream](#) (`??sec:std-InputStream`)
 - [Lazy](#) (`??sec:std-Lazy`)
 - [Math](#) (`??sec:std-Math`)
 - [Mutex](#) (`??sec:std-Mutex`)
 - [Regexp](#) (`??sec:std-Regexp`)
 - [Object](#) (`??sec:std-Object`)
 - [OutputStream](#) (`??sec:std-OutputStream`)
 - [Pair](#) (`??sec:std-Pair`)
 - [String](#) (`??sec:std-String`)
 - [Tag](#) (`??sec:std-Tag`)
 - [Timeout](#) (`??sec:std-Timeout`)
- tutorial:
 - `uobjects`

24.8.5 Various

- Text files are converted to DOS end-of-lines for Windows packages.
- `urbi-send` supports `--quit`.
- The files `'global.u'`/`'local.u'` replace `'URBI.INI'`/`'CLIENT.INI'`.
- `urbi` supports `--quiet` to inhibit the banner.

24.9 Urbi SDK 2.0 RC 1

Released on 2009-04-03.

24.9.1 Auxiliary programs

- `urbi-send` no longer displays the server version banner, unless given `'-b'`/`--banner`.
- `urbi-console` is now called simply `urbi`.
- `urbi.bat` should now work out of the box under windows.

24.9.2 urbiscript

24.9.2.1 Syntax of events

The keyword `emit` is deprecated in favor of `!`.

Deprecated	Updated
<code>emit e;</code>	<code>e!;</code>
<code>emit e(a);</code>	<code>e!(a);</code>
<code>emit e ~ 1s;</code>	<code>e! ~ 1s;</code>
<code>emit e(a) ~ 1s;</code>	<code>e!(a) ~ 1s;</code>

The `?` construct is changed for symmetry.

Deprecated	Updated
<code>at (?e)</code>	<code>at (e?)</code>
<code>at (?e(var a))</code>	<code>at (e?(var a))</code>
<code>at (?e(var a) if 0 <= a)</code>	<code>at (e?(var a) if 0 <= a)</code>
<code>at (?e(2))</code>	<code>at (e?(2))</code>

This syntax for sending and receiving is traditional and can be found in various programming languages.

24.9.2.2 Changes

- `System.Platform` (`??sec:std-System.Platform`) enhances former `System.platform`. Use `System.Platform.kind` instead of `System.platform`.

24.9.2.3 Fixes

- Under some circumstances successful runs could report "at job handler exited with exception TerminateException". This is fixed.
- Using `waituntil` on an event with no payload (i.e., `waituntil(e?) ...;`) will not cause an internal error anymore.

24.9.3 URBI Remote SDK

The API for plugged-in UObjects is not thread safe, and never was: calls to the API must be done only in the very same thread that runs the Urbi code. Assertions (run-time failures) are now triggered for invalid calls.

24.9.4 Documentation

Extended documentation on: [Comparable](#) (`??sec:std-Comparable`), [Orderable](#) (`??sec:std-Orderable`).

24.10 Urbi SDK 2.0 beta 4

Released on 2009-03-03.

24.10.1 Documentation

An initial sketch of documentation (a tutorial, and the language and library specifications) is included.

24.10.2 urbiscript

24.10.2.1 Bug fixes

- Bitwise operations.
The native "long unsigned int" type is now used for all the bitwise operations (`&`, `|`, `^`, `compl`, `<<`, `>>`). As a consequence it is now an error to pass negative operands to these operations.
- [System.PackageInfo](#).
This new object provides version information about the Urbi package. It is also used to ensure that the initialization process uses matching Urbi and C++ files. This should prevent accidental mismatches due to incomplete installation processes.
- Precedence of operator `**`.
In conformance with the usage in mathematics, the operator `**` now has a stronger precedence than the unary operators. Therefore, as in Perl, Python and others, `'-2 ** 2 == -4'` whereas it used to be equal to `'4'` before (as with GNU bc).

- `whenever` now properly executes the else branch when the condition is false. It used to wait for the condition to be verified at least once before.

24.10.2.2 Changes

- `String.asFloat`

This new method has been introduced to transform a string to a float. It raises a `PrimitiveError` exception if the conversion fails:

```
"2.1".asFloat;
[00000002] 2.1
"2.0a".asFloat;
[00000003:error] !!! asFloat: unable to convert to float: "2.0a"
```

24.10.3 Programs

24.10.3.1 Environment variables

The environment variable `URBI_ROOT` denotes the directory which is the root of the tree into which Urbi was installed. It corresponds to the "prefix" in GNU Autoconf parlance, and defaults to `/usr/local` under Unix. urbiscript library files are expected to be in `URBI_ROOT/share/gostai/urbi`.

The environment variable `URBI_PATH`, which allows to specify a colon-separated list of directories into which urbiscript files are looked-up, may extend or override `URBI_ROOT`. Any superfluous colon denotes the place where the `URBI_ROOT` path is taken into account.

24.10.3.2 Scripting

To enable writing (batch) scripts seamlessly in Urbi, `urbi-console -f/--fast` is now renamed as `-F/--fast`. Please, never use short options in batch programs, as they are likely to change.

Two new option pairs, `-e/--expression` and `-f/--file`, plus the ability to reach the command line arguments from Urbi make it possible to write simple batch Urbi programs. For instance:

```
$ cat demo
#!/usr/bin/env urbi-console
cout << System.arguments;
shutdown;

$ ./demo 1 2 3 | grep output
[00000004:output] ["1", "2", "3"]
```

24.10.3.3 urbi-console

`urbi-console` is now a simple wrapper around `urbi-launch`. Running

```
urbi-console arg1 arg2...
```

is equivalent to running

```
urbi-launch --start -- arg1 arg2...
```

24.10.3.4 Auxiliary programs

The command line interface of `urbi-sendbin` has been updated. `urbi-send` now supports `'-e'/'--expression'` and `'-f'/'--file'`. For instance

```
$ urbi-send -e 'var x;' -e "x = $value;" -e 'shutdown;'
```

24.11 Urbi SDK 2.0 beta 3

Released on 2009-01-05.

24.11.1 Documentation

A new document, `'FAQ.txt'`, addresses the questions most frequently asked by our users during the beta-test period.

24.11.2 urbiscript

24.11.2.1 Fixes

- If a file loaded from `'URBI.INI'` cannot be found, it is now properly reported.

24.11.2.2 Changes

new syntax revamped

The syntax `"new myObject(myArgs)"` has been deprecated and now gives a warning. The recommended `"myObject.new(myArgs)"` is suggested.

delete has been removed

`delete` was never the right thing to do. A local variable should not be deleted, and a slot can be removed using `Object.removeSlot`. The construct `"delete object"` has been removed from the language.

`__HERE__`

The new `__HERE__` pseudo-symbol gives the current position. It features three self explanatory slots: `"file"`, `"line"`, and `"column"`.

Operator "()"

It is now possible to define the "()" operator on objects and have it called as soon as at least one parameter is given:

```
class A {
  function '()' (x) { echo("A called with " + x) };
}|;
A;
[00000001] A
A();
[00000002] A
A(42);
[00000003] *** A called with 42
```

"catch(jtype_i jname_i)" syntax removed

The "catch(jtype_i jname_i)", which was used to catch exceptions if and only if they inherited jtype_i, has been removed. This behavior can be obtained with the more general guard system:

```
catch (var e if e.isA(<type>))
{
  ...
}
```

Pattern matching and guards in catch blocks

Exception can now be filtered thanks to pattern matching, just like events. Moreover, the pattern can be followed by the "if" keyword and an arbitrary guard. The block will catch the exception only if the guard is true.

```
try
{ ... }
catch ("foo") // Catch only the "foo" string
{ ... }
catch (var x if x.isA(Float) && x > 10) // Catch all floats greater than 10
{ ... }
catch (var e) // Catch any other exception
{ ... }
```

Parsing of integer literals

The parser could not read integer literals greater than 2**31-1. This constraint has been alleviated, and Urbi now accepts integer literals up to 2**63-1.

Display of integer literals

Some large floating point values could not be displayed correctly at the top level of the interpreter. This limitation has been removed.

Variables binding in event matching

Parentheses around variables bindings ("var x") are no longer required in event matching:

```
at (?myEvent(var x, var y, 1))
```

instead of:

```
at (?myEvent((var x), (var y), 1))
```

Waituntil and bindings

Bindings performed in "waituntil" constructs are now available in its context:

```
waituntil(?event(var x));
// x is available
echo (x);
```

"List.insert" method

Now uses an index as its first argument and inserts the given element before the index position:

```
["a", "b", "c"].insert(1, "foo");
[00000001] ["a", "foo", "b", "c"]
```

"List.sort" method

Now takes an optional argument, which is a function to call instead of the "i" operator. Here are two examples illustrating how to sort strings, depending on whether we want to be case-sensitive (the default) or not:

```
["foo", "bar", "Baz"].sort;
[00000001] ["Baz", "bar", "foo"]
["foo", "bar", "Baz"].sort(function(x, y) {x.toLower < y.toLower});
[00000002] ["bar", "Baz", "foo"]
```

"System.searchPath" method

It is now possible to get the search path for files such as 'urbi.u' or 'URBI.INI' by using `System.searchPath`.

"System.getenv" method

Now returns "nil" if a variable cannot be found in the environment instead of "void". This allows you do to things such as:

```
var ne = System.getenv("nonexistent");
if (!ne.isNil) do_something(ne);
```

while previously you had to retrieve the environment variable twice, once to check for its existence and once to get its content.

”Control.disown”

It is now possible to start executing code in background while dropping all the tags beforehand, including the connection tag. The code will still continue to execute after the connection that created it has died.

”Object.removeSlot”

Now silently accepts non-existing slot names instead of signaling an error.

”Semaphore.criticalSection”

It is now possible to define a critical section associated with a semaphore. The **Semaphore.acquire** method will be called at the beginning, and if after that the operation is interrupted by any means the **Semaphore.release** operation will be called before going on. If there are no interruption, the **Semaphore.release** operation will also be called at the end of the callback:

```
var s = \refSlot[Semaphore]{new}(1);
s.criticalSection(function () { echo ("In the critical section") });
```

”System.stats” Its output is now expressed in seconds rather than milliseconds, for consistency with the rest of the kernel.

24.11.3 UObjects

void The error message given to the user trying to cast a void UVar has been specialized.

Remote bound methods can now return void.

Coroutine interface The functions `yield()`, `yield_until()`, and `yield_until_things_changed()` have been added to the UObject API. They allow the user to write plugin UObject code that behaves like any other coroutine in the kernel: if `yield()` is called regularly, the kernel can continue to work while the user code runs. Meaningful implementation for these functions is provided also in remote mode: calling `yield()` will allow the UObject remote library to process pending messages from within the user callback.

Remote UObject initialization Remote UObject instantiation is now atomic: the API now ensures that all variables and functions bound from the UObject constructor and `init` are visible as soon as the UObject itself is visible. Code like:

```
waituntil(uobjects.hasSlot("MyRemote")) | var m = \refSlot[MyRemote]{new}();
```

is now safe.

24.11.4 Auxiliary programs

urbi-launch Now, options for `urbi-launch` are separated from options to give to the underlying program (in remote and start modes) by using `--`. Use `urbi-launch --help` to get the full usage information.

24.12 Urbi SDK 2.0 beta 2

Released on 2008-11-03.

24.12.1 urbiscript

”object” and ”from” as identifiers

”object” and ”from” are now regular identifiers and can be used as other names. For example, it is now legal to declare:

```
var object = 1|;  
var from = 1|;
```

Hexadecimal literals It is now possible to enter (integral) hexadecimal numbers by prefixing them with ”0x”, as in:

```
0x2a;  
[00000001] 42
```

Only integral numbers are supported.

24.12.2 Standard library

String.asList

”String” now has a ”asList” method, which can be used transparently to iterate over the characters of a string:

```
for (var c: "foo") echo (c);  
[00000001] *** f  
[00000002] *** o  
[00000003] *** o
```

String.split method Largely improved.

”min” and ”max”

It is now possible to call ”min” and ”max” on a list. By default, the ”i” comparison operator is used, but one explicit ‘lower than’ function can be provided as ”min” or ”max” argument should one be needed. Here is an example on how to compare strings in case-sensitive and case-insensitive modes:

```
[ "the", "brown", "Fox" ].min;  
[00000001] "Fox"  
[ "the", "brown", "Fox" ].min(function (l, r) { l.toLower < r.toLower });  
[00000002] "brown"
```

Global functions ”min” and ”max” taking an arbitrary number of arguments have also been defined. In this case, the default ”i” operator is used for comparison:

```
min(3, 2, 17);
[00000001] 2
```

Negative indices

It is now possible to use negative indices when taking list elements. For example, -1 designates the last element, and -2 the one before that.

```
["a", "b", "c"][-1];
[00000001] "c"
```

Tag names

Tags were displayed as `Tag_0xADDR` which did not make their "name" slot apparent. They are now displayed as `"Tag{name}"`:

```
Tag.new;
[00000001] Tag<tag_1>
Tag.new("mytag");
[00000002] Tag<mytag>
```

"every" and exceptions

If an exception is thrown and not caught during the execution of an "every" block, the "every" expression is stopped and the exception displayed.

24.12.3 UObjects

"UVar::type()" method

It is now possible to get the type of a "UVar" by calling its "type()" method, which returns a "UDataType" (see `'urbi/uvalue.hh'` for the types declarations).

24.12.4 Run-time

Stack exhaustion check on Windows

As was done on Linux already, stack exhaustion condition is detected on Windows, for example in the case of an infinite recursion. In this case, `SchedulingError` will be raised and can be caught.

Errors from the trajectory generator are propagated

If the trajectory generator throws an exception, for example because it cannot assign the result of its computation to a non-existent variable, the error is propagated and the generator is stopped:

```
xx = 20 ampli:5 sin:10s;
[00002140:error] !!! lookup failed: xx
```

24.12.5 Bug fixes

Support for Windows shares

Previous versions of the kernel could not be launched from a Windows remote directory whose name is starting with two slashes such as `'//share/some/dir'`.

Implement `"UVar::syncValue()"` in plugged uobjects

Calling `"syncValue()"` on a `"UVar"` from a plugged `UObject` resulted in a link error. This method is now implemented, but does nothing as there is nothing to do. However, its presence is required to be able to use the same `UObject` in both remote and engine modes.

`"isdef"` works again

The support for k1 compatibility function `"isdef"` was broken in the case of composed names or variables whose content was `"void"`. Note that we do not recommend using `"isdef"` at all. Slots related methods such as `"getSlot"`, `"hasSlot"`, `"locateSlot"`, or `"slot-Names"` have much cleaner semantics.

`"__name"` macro

In some cases, the `__name` macro could not be used with plugged uobjects, for example in the following expression:

```
send(__name + ".val = 1;");
```

This has been fixed. `__name` contains a valid slot name of `uobjects`.

24.12.6 Auxiliary programs

The sample programs demonstrating the SDK Remote, i.e., how to write a client for the Urbi server, have been renamed from `urbi*` to `urbi-*`. For instance `urbisend` is now spelled `urbi-send`.

Besides, their interfaces are being overhauled to be more consistent with the Urbi command-line tool-box. For instance while `urbisend` used to require exactly two arguments (host-name, file to send), it now supports options (e.g., `'--help'`, `'--port'` to specify the port etc.), and as many files as provided on the command line.

Chapter 25

Licenses

Part of the Urbi SDK is based on software distributed under the following licenses. Other licenses are included below for information; each section explains why its corresponding license is included here.

25.1 BSD License

In order to be included in Urbi SDK, contributors can use this license for their patches, see [Section 15.5.2](#).

This license is also known as the “modified BSD License”, see the [License List](#) maintained by the Free Software Foundation.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE

POSSIBILITY OF SUCH DAMAGE.

25.2 Expat License

In order to be included in Urbi SDK, contributors can use this license for their patches, see [Section 15.5.2](#).

This license is also known as the “MIT License”, see the [License List](#) maintained by the Free Software Foundation.

```
Copyright (c) <year> <copyright holders>
```

```
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
```

```
The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

25.3 Independent JPEG Group’s Software License

Urbi SDK uses the Independent JPEG Group’s software, better known as libjpeg. Its license requires us to state that:

This software is based in part on the work of the Independent JPEG Group.

The license below is the relevant part of the ‘COPYRIGHT’ file in the top-level directory of libjpeg.

```
The authors make NO WARRANTY or representation, either express or implied,
with respect to this software, its quality, accuracy, merchantability, or
fitness for a particular purpose. This software is provided "AS IS", and you,
its user, assume the entire risk as to its quality and accuracy.
```

```
This software is copyright (C) 1991-1998, Thomas G. Lane.
All Rights Reserved except as specified below.
```

Permission is hereby granted to use, copy, modify, and distribute this software (or portions thereof) for any purpose, without fee, subject to these conditions:

- (1) If any part of the source code for this software is distributed, then this README file must be included, with this copyright and no-warranty notice unaltered; and any additions, deletions, or changes to the original files must be clearly indicated in accompanying documentation.
- (2) If only executable code is distributed, then the accompanying documentation must state that "this software is based in part on the work of the Independent JPEG Group".
- (3) Permission for use of this software is granted only if the user accepts full responsibility for any undesirable consequences; the authors accept NO LIABILITY for damages of any kind.

These conditions apply to any software derived from or based on the IJG code, not just to the unmodified library. If you use our work, you ought to acknowledge us.

Permission is NOT granted for the use of any IJG author's name or company name in advertising or publicity relating to this software or products derived from it. This software may be referred to only as "the Independent JPEG Group's software".

We specifically permit and encourage the use of this software as the basis of commercial products, provided that all warranty or liability claims are assumed by the product vendor.

25.4 Libcoroutine License

Urbi uses the libcoroutine from the *Io* language. See <http://www.iolanguage.com/>.

(This is a BSD License)

Copyright (c) 2002, 2003 Steve Dekorte
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the author nor the names of other contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

25.5 OpenSSL License

LICENSE ISSUES

=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License

```
/* =====
 * Copyright (c) 1998-2007 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 *    endorse or promote products derived from this software without
 *    prior written permission. For written permission, please contact
 *    openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 *    nor may "OpenSSL" appear in their names without prior written
```

```

*   permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*   acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT 'AS IS' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com). This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

Original SSLeay License
-----

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to. The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions

```

```

* are met:
* 1. Redistributions of source code must retain the copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*    must display the following acknowledgement:
*    "This product includes cryptographic software written by
*      Eric Young (eay@cryptsoft.com)"
*    The word 'cryptographic' can be left out if the routines from the library
*    being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*    the apps directory (application code) you must include an acknowledgement:
*    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG 'AS IS' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/

```

25.6 Urbi Open Source Contributor Agreement

The following text is not a license, but it's one way for Urbi SDK contributors to enable us to use their code. See [Section 15.5.2](#). This document itself is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License <http://creativecommons.org/licenses/by-sa/3.0/>.

IMPORTANT - PLEASE READ CAREFULLY BEFORE SIGNING

These terms apply to your contribution of materials to the Urbi Open Source project owned and managed by us (Gostai S.A.S), and set out the intellectual property rights you grant to us in the contributed materials. If this contribution is on behalf of a company, the term 'you' will also mean the company you identify below. If you agree to be bound by these terms, fill in the information requested below and provide your signature.

1. The term “contribution” means any source code, object code, patch, tool, creation, images, sound, sample, graphic, specification, manual, documentation, or any other material posted or submitted by you to the Urbi Open Source project, in human or machine readable form. For the avoidance of doubt: is considered as a “contribution” any material that you will send to us via one of the email addresses of the Gostai employees or owned projects (for example `project-contrib@gostai.com` as displayed on the project web page), or via pushing to any of our public git/svn or other code repositories.
 - you hereby assign to us joint ownership, and to the extent that such assignment is or becomes invalid, ineffective or unenforceable, you hereby grant to us a perpetual, irrevocable, non-exclusive, worldwide, no-charge, royalty-free, unrestricted license to exercise all rights under those copyrights, including a license to use, reproduce, prepare derivative works of, publicly display, publicly perform and distribute. This also includes, at our option, the right to sublicense these same rights to third parties through multiple levels of sublicensees or other licensing arrangements;
 - you agree that each of us can do all things in relation to your contribution as if each of us were the sole owners, and if one of us makes a derivative work of your contribution, the one who makes the derivative work (or has it made) will be the sole owner of that derivative work;
 - you agree that you will not assert any moral rights in your contribution against us, our licensees or transferees;
 - you agree that we may register a copyright in your contribution and exercise all ownership rights associated with it; and
 - you agree that neither of us has any duty to consult with, obtain the consent of, pay or render an accounting to the other for any use or distribution of your contribution.
2. With respect to any patents you own, or that you can license without payment to any third party, you hereby grant to us a perpetual, irrevocable, non-exclusive, worldwide, no-charge, royalty-free license to:

- make, have made, use, sell, offer to sell, import, and otherwise transfer your contribution in whole or in part, alone or in combination with or included in any product, work or materials arising out of the project to which your contribution was submitted, and
 - at our option, to sublicense these same rights to third parties through multiple levels of sublicensees or other licensing arrangements.
3. Except as set out above, you keep all right, title, and interest in your contribution. The rights that you grant to us under these terms are effective on the date you first submitted a contribution to us, even if your submission took place before the date you sign these terms.
 4. With respect to your contribution, you represent that:
 - it is an original work and that you can legally grant the rights set out in these terms;
 - it does not to the best of your knowledge violate any third party's copyrights, trademarks, patents, or other intellectual property rights; and
 - you are authorized to sign this contract on behalf of your company (if identified below).
 5. These terms will be governed in all respects by French laws and the French courts only shall have jurisdiction in relation to them.

If available, please list your user name(s) (or "login") and the name of the project(s) (or project website(s) or repository) for which you would like to contribute materials.

Your user name:

Project name, website or repository:

Your user name:

Project name, website or repository:

Your contact information (Please print clearly):

Your name:

Your company's name (if applicable):

Mailing address:

Telephone, Fax and Email:

Your signature:

Date:

To deliver these terms to us, scan and email, or fax a signed copy to us using the contrib@gostai.com email address or fax number set out on the appropriate project website.

Chapter 26

Glossary

This chapter aggregates the definitions used in the this document.

Bioloid The Robotis Bioloid is a hobbyist and educational robot kit produced by the Korean robot manufacturer Robotis. The Bioloid platform consists of components and small, modular servomechanisms called Dynamixels, which can be used in a daisy-chained fashion to construct robots of various configurations, such as wheeled, legged, or humanoid robots. The Bioloid system is thus comparable to the LEGO Mindstorms and VEXplorer kits.

Gostai Console This tool provide a graphical user interface for Windows users to a remote Urbi server (see [Figure 26.1](#)). Unix users, GNU/Linux or Mac OS X, can use the traditional `telnet` tool. Windows users are invited to use Gostai Console instead. See [Chapter 2](#).

Gostai Lab This tool (see [Figure 26.2](#)), which includes the features of Gostai Console, allows to build easily elaborate remote controller for robots. It provides various widgets to visualize data from the robot (including video and sound), and to modify the state of the robot.

Gostai Studio This tool (see [Figure 26.3](#)), includes all the features of Gostai Console and Gostai Lab. It is a high-level Integrated Development Environment for Urbi. Its formalism is based on *Hierarchical Finite State Machines*.

RMP The Segway Robotic Mobility Platform is a robotic platform based on the Segway Personal Transporter. See <http://rmp.segway.com>.

ROS Robot Operating System, developed by Willow Garage. It is an abstraction layer on top of the genuine operating system (such as GNU/Linux) that provides hardware abstraction, device control, common algorithms, message-passing between processes, and package management.

Spykee The Spykee is a WiFi-enabled robot built by Meccano (known as Erector in the United States). It is equipped with a camera, speaker, microphone, and moves using two tracks. See <http://www.spykeeworld.com>.

urbi-console Former name of “Gostai Console”. See that item.

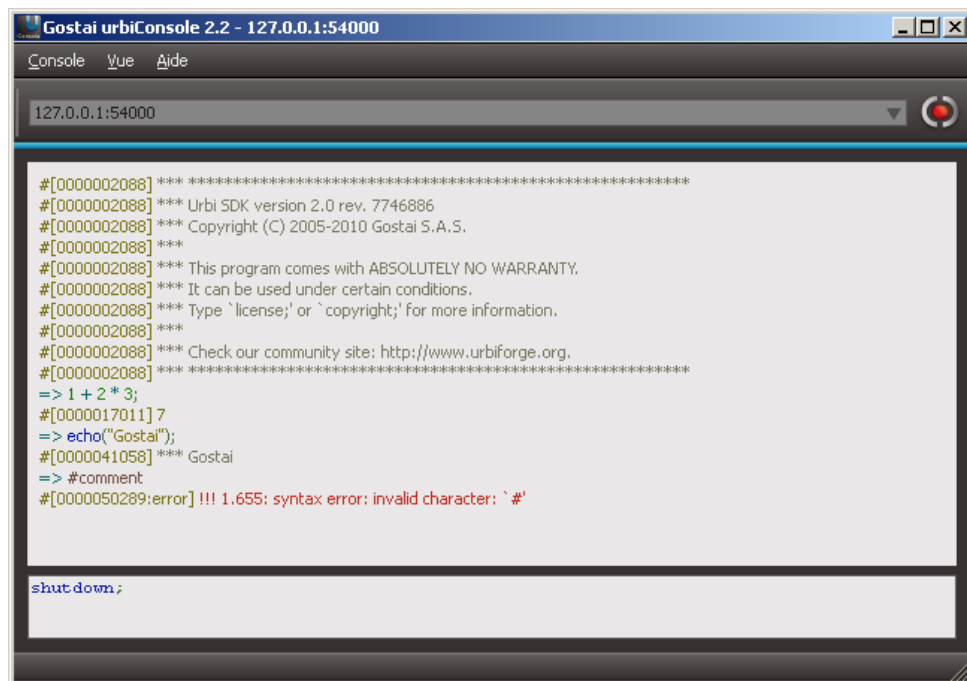


Figure 26.1: Gostai Console

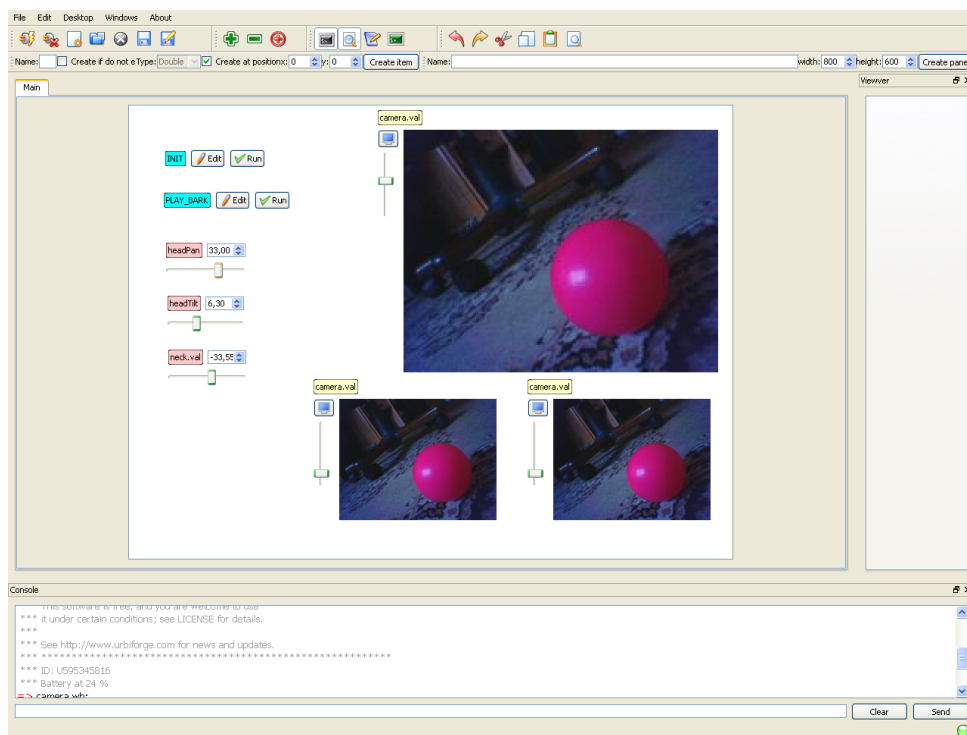


Figure 26.2: Gostai Lab

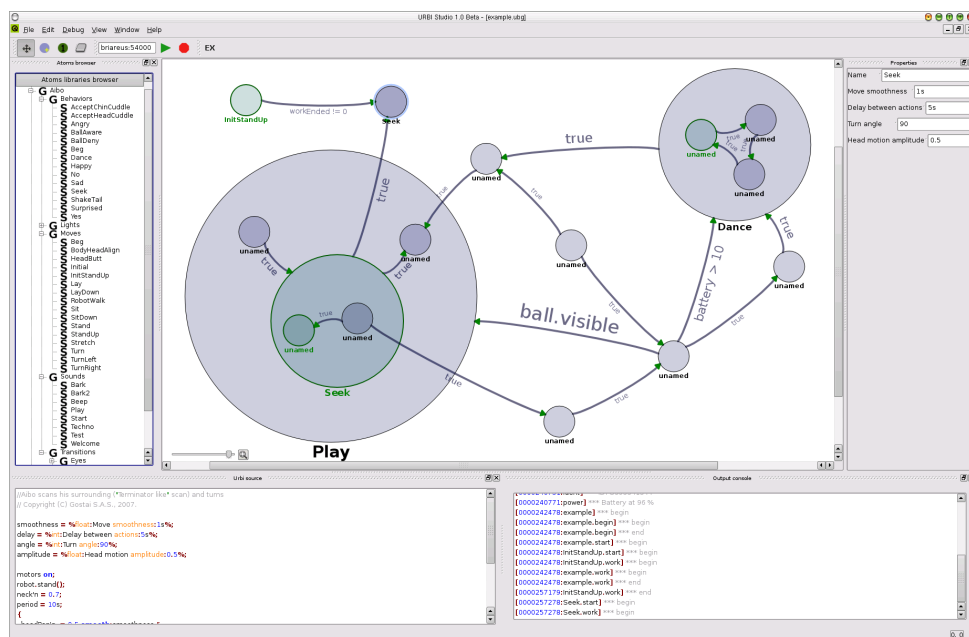


Figure 26.3: Gostai Studio

Chapter 27

Index

'&&', 221, 317
'*', 260, 294, 349, 376
'**', 260
'+', 219, 234, 260, 294, 325, 350, 376
'-', 234, 260, 294, 326
'/', 260, 322
'<', 234, 260, 295, 323, 326, 350, 376
'<<', 225, 243, 260, 275, 295, 384
'==', 219, 234, 236, 261, 293, 302, 323, 326, 349, 376
'>>', 260
'[]', 236
'[]=', 236
'^', 259
'assert', 351
'bitand', 255
'bitor', 255
'each&', 287
'emit', 242
'new', 342
_exit, 351

abs, 253, 304
absolute, 321
Accel, 373
Accel, 369
accel, 407
acceleration, 397
AccelerationSensor, 397
acceptVoid, 308, 378
accumulate, 288

acos, 253, 304
acquire, 339
add, 274
addComponent, 409
addDevice, 409
addProto, 308
advertise, 384
age?, 401
alignment, 263
aliveJobs, 351
all, 334
'all.stamp', 148
allProto, 309
allSlotNames, 130, 309
alt, 263
angle, 167
angle, 396
angleMax, 397
angleMin, 397
ankle, 405
any, 334
apply, 229, 309, 327
argMax, 286
argMin, 286
args, 222
argsCount, 223
arguments, 351
arm, 404
as, 309
asBool, 221, 237, 253, 286, 309

ASCII, 165
 asEvent, 242
 asExecutable, 248
 asFloat, 234, 242, 254, 346
 asin, 254, 304
 asJob, 278
 asList, 237, 240, 249, 254, 265, 286, 321, 346
 asMutex, 307
 asPrimitive, 327
 asPrintable, 249, 321, 335, 346
 asProcess, 329
 assert_, 352
 assert_op, 352
 assertion, 201
 assignment, 176
 associative array, 235
 asString, 219, 229, 234, 237, 240, 242, 249, 254, 274, 278, 285, 287, 303, 321, 326, 329, 336, 338, 346, 375
 at, 203
 atan, 254, 304
 atan2, 304
 attribute, 214
 AudioIn, 400
 AudioOut, 399
 authors, 298

 b, 402
 back, 287
 backtrace, 244, 278, 352
 banner, 298
 Barrier, 217, 266
 basename, 321
 Battery, 402
 begin, 303, 365
 Binary, 218, 266
 Binding, 323
 bindings, 324
 blob, 218, 400
 BlobDetector, 400
 block, 361
 catch-block, 200
 try-block, 200
 block, 365
 blocked, 365
 body, 403
 bodyString, 229
 Boolean, 220
 bootstrap, 141
 bounce, 309
 buffer, 319
 build, 141

 ‘-C’, 163
 ‘-c’, 160, 163
 caching, 283
 caller, 224
 CallMessage, 221, 266
 callMessage, 309
 camera, 407
 capacity, 402
 cd, 321
 cerr, 266
 Channel, 225, 267
 checkMaster, 382
 class, 308
 clavicle, 407
 ‘--clean’, 163
 clear, 237, 287
 ‘CLIENT.INI’, 156
 clog, 267
 clone, 255, 279, 309, 342
 cloneSlot, 310
 close, 319
 ClosedLoop, 373
 closest, 346
 closure, 82
 closure (lexical), 185
 Code, 227, 267
 code, 223
 column, 326
 columns, 326
 commands, 280
 comment, 61, 165
 Comparable, 229, 267
 compl, 255
 Component, 409
 component, 389

- connect, 344
- connected, 298, 344
- connection, 340
- connections, 280
- connectionTag, 298
- connectSerial, 344
- constructor, 130
- Container, 230
- content, 240, 249
- Control, 231
- Control.detach, 99
- copy, 280, 377
- copy on write, 188
- copyright, 299
- copyrightHolder, 358
- copyrightYears, 358
- copySlot, 310
- '--core=*core*', 163
- Cos, 369
- cos, 255, 304
- cout, 267
- create, 249, 300
- createSlot, 310
- criticalSection, 338
- current, 402
- '--customize=*file*', 160
- cwd, 321
- cycle, 352
- '-D', 162
- '-d', 157, 159, 160
- data, 219
- Date, 233, 267
- '--debug', 162
- '--debug=*level*', 157, 160
- '--deep-clean', 163
- delete, 124
- destructor, 250
- detach, 231, 267
- '--device=*device*', 159
- devices, 281
- DGain?, 395
- dictionaries, 167
- Dictionary, 235, 267
- dictionary, 235
- digits, 261
- digits10, 261
- Directory, 239, 240, 322
- Directory, 239, 267
- dirname, 322
- '--disable-automain', 163
- disconnect, 344
- disconnected, 344
- disown, 232, 267
- distance, 347, 396
- distanceMax, 397
- distanceMin, 397
- DistanceSensor, 396
- 'doc/tests/test-suite.log', 149
- done, 329
- dump, 310, 409
- dumpState, 279
- Duration, 241, 267
- duration, 169
- duration, 400
- '-e', 158, 161
- each, 255, 274, 287, 334
- each&, 256, 274
- eachi, 287
- ear, 405
- echo, 225, 267, 300
- echoEach, 300
- elbow, 404
- elongation?, 401
- empty, 219, 237, 288
- enabled, 226
- encoding, 165
- end, 303, 366
- enter, 366
- epoch, 233
- epoch, 234
- epsilon, 261
- erase, 237
- error, 344
- eval, 285, 352
- evalArgAt, 223
- evalArgs, 223

- evaluate, 268
- Event, 242, 268
- event, 131, 242
- events, 281
- every, 204
- Exception, 243, 268
- exception
 - catching, 200
 - throwing, 200
- Executable, 248, 268
- 'executables.stamp', 148
- exists, 322
- exp, 256, 305
- exposure?, 399
- '--expression=*exp*', 158
- '--expression=*script*', 161
- external, 268
- eye, 406
- eyebrow, 406

- '-F', 157, 159
- '-f', 158, 161
- fallback, 274
- false, 220, 268
- '--fast', 157
- File, 322
- File, 248, 268
- file, 326
- '--file=*file*', 158, 161
- fileCreated, 240
- fileDeleted, 241
- Filter, 226
- filter, 288
- Finalizable, 250, 268
- finalize, 252
- finally, 232
- finger, 404
- first, 320, 374
- flatDump, 409
- Float, 252, 269
- Float.limits, 261
- flush, 319
- foldl, 288
- foot, 405

- force, 395
- format, 256
- format info, 262
- '--format=*format*', 159
- format?, 398
- FormatInfo, 262, 269
- Formatter, 265, 269
- formatter, 265
- freeze, 362
- freeze, 366
- fresh, 347
- front, 288
- frozen, 366
- function, 182
 - lazy, 184
 - return value, 183
 - strict, 184
- functions, 281

- g, 402
- gain?, 399, 400
- garbage collection, 138
- GD_LEVEL, 156
- gender?, 401
- get, 237, 275
- getAll, 334
- getChar, 276
- getenv, 353
- getIoService, 341, 344
- getLine, 276
- getLocalSlot, 311
- getOne, 333
- getPeriod, 311
- getProperty, 269, 274, 311
- getSlot, 311
- getWithDefault, 238
- Global, 266
- Global, 266, 269
- 'global.u', 156
- go, 398
- grip, 405
- Group, 269, 273
- group, 263
- guard, 191

- gyro, 407
- GyroSensor, 397
- ‘-H’, 158, 159, 161, 163
- ‘-h’, 157, 159–162
- hand, 404
- has, 230, 238, 288, 336
- hash, 235
- hasLocalSlot, 312
- hasNot, 231
- hasProperty, 274, 312
- hasSame, 289
- hasSlot, 312
- head, 289, 405
- hear, 402
- height, 398
- ‘--help’, 157, 159–162
- Hierarchical Finite State Machines, 449
- hip, 405
- host, 341, 345, 358
- ‘--host=address’, 158
- ‘--host=host’, 159, 161, 163
- hostCpu, 359
- hostOs, 359
- hostVendor, 359
- ‘-i’, 158
- identifier, 166
- Identity, 394
- IGain?, 395
- in, 178
- inf, 257, 305
- init, 238
- initial value, 214, 368
- initialized, 386
- InputStream, 269, 275
- insert, 289
- insertBack, 289
- insertFront, 290
- inspect, 312
- instances, 300
- ‘--interactive’, 158
- Interface, 409
- Interface.AccelerationSensor, 397
- Interface.AudioIn, 400
- Interface.AudioOut, 399
- Interface.Battery, 402
- Interface.BlobDetector, 400
- Interface.DistanceSensor, 396
- Interface.GyroSensor, 397
- Interface.Identity, 394
- Interface.Laser, 397
- Interface.Led, 402
- Interface.LinearMotor, 395
- Interface.LinearSpeedMotor, 395
- Interface.Mobile, 398
- Interface.Motor, 395
- Interface.Network, 395
- Interface.RGBLed, 402
- Interface.RotationalMotor, 396
- Interface.RotationalSpeedMotor, 396
- Interface.Sensor, 396
- Interface.SpeechRecognizer, 401
- Interface.TemperatureSensor, 397
- Interface.TextToSpeech, 401
- Interface.TouchSensor, 396
- Interface.Tracker, 398
- Interface.VideoIn, 398
- Io, 443
- IoService, 276
- IoService, 276
- IP, 395
- isA, 313
- isConnected, 345
- isdef, 269
- isDir, 322
- isNil, 307, 313
- isProto, 313
- isReg, 322
- isVoid, 313, 378
- isvoid, 281
- isWindows, 359
- ‘-j’, 159, 163
- Job, 270, 278
- ‘--jobs=jobs’, 163
- join, 290, 329, 348
- joint, 405

- `--jpeg=factor`, 159
- `-k`, 163
- Kernel1, 270, 280
- `--kernel=dir`, 163
- keys, 238, 290
- keyword, 167
- keywords, 220
- kill, 329
- kind, 359
- knee, 405
- `-l`, 163
- lang?, 401
- Laser, 397
- launch, 368
- Lazy, 270, 282
- lazy, 282
- leave, 366
- Led, 402
- leg, 404
- `libraries.stamp`, 148
- `--library`, 163
- license, 300
- limits, 257
- line, 326
- LinearMotor, 395
- LinearSpeedMotor, 395
- lines, 327
- lip, 406
- List, 270, 285
- list, 132, 170
- listen, 341
- load, 296, 353
- Loadable, 270, 295
- loadFile, 353
- loadLibrary, 354
- loadModule, 354
- lobbies, 354
- Lobby, 297
- Lobby, 270, 297
- lobby, 126, 297
- lobby, 301, 354
- `local.u`, 156
- localhost, 345
- Localizer, 409
- localPort, 345
- localSlotNames, 130, 313
- locateSlot, 313
- Location, 302
- location, 245, 338
- log, 257, 305
- loop
 - closed-loop, 368, 373
 - open-loop, 368, 373
- `-m`, 163
- makeCompactNames, 409
- makeServer, 277
- makeSocket, 278
- map, 290
- match, 324, 336
- matchAgainst, 239, 375
- matchPattern, 324
- Math, 270, 304
- max, 257, 261, 291, 305
- maxExponent, 261
- maxExponent10, 261
- maybeLoad, 354
- memoization, 282
- message, 64, 189
 - call, 184
- message, 224, 245
- meta-variable, 415
- methodToFunction, 270
- micro, 407
- min, 257, 262, 291, 305
- minExponent, 262
- minExponent10, 262
- Mobile, 398
- model, 394
- Motor, 395
- mouth, 405
- Mutex, 306
- Mutex, 270, 306
- name, 227, 279, 330, 338, 382, 383, 386, 394
- nan, 257, 305

- ndebug, 354
- neck, 405
- Network, 395
- nil, 271, 307
- nodes, 382
- noop, 281
- not in, 178
- notifyAccess, 378
- notifyChange, 377
- now, 235
- null, 227
- ‘-o’, 159, 163
- Object, 266
- Object, 271, 307
- object, 67, 186
- off, 296
- on, 297
- onConnect, 384
- onDisconnect, 301, 384
- onMessage, 384
- onSubscribe, 243
- open, 322
- OpenLoop, 373
- operator, 175
 - arithmetics, 175
 - bitwise, 176
 - Boolean, 176
 - comparison, 177
 - subscript, 178
- Orderable, 271, 318
- orientation?, 401
- ‘--output=*file*’, 159, 163
- OutputStream, 271, 318
- owned, 378
- ‘-P’, 158, 159, 161, 163
- p, 339
- ‘-p’, 159, 160, 163
- PackageInfo, 354
- pad, 263
- Pair, 271, 319
- pair, 319
- ‘--param-mk=*file*’, 163
- Path, 320, 322
- Path, 271, 320
- Pattern, 271, 323
- pattern, 264, 324
- pattern matching, 190
- period, 354
- period, 354
- ‘--period=*period*’, 159
- persist, 232, 271
- PGain?, 395
- pi, 258, 305
- ping, 281
- pitch, 398, 406
- pitch?, 401
- Platform, 355
- playing, 400
- ‘--plugin’, 160
- plugin mode, 132
- poll, 278, 345
- pollFor, 278
- pollInterval, 345
- pollOneFor, 278
- port, 341, 345
- ‘--port-file=*file*’, 158, 159, 161
- ‘--port=*port*’, 158, 159, 161
- Position, 271, 325
- position, 395
- precision, 264
- prefix, 264
- ‘--prefix=*dir*’, 163
- pressure, 396
- Primitive, 271, 327
- print, 314
- Process, 271, 327
- Profiling, 271, 331
- programName, 355
- prompt, 415
- properties, 314
- protos, 130, 314
- prototype, 89, 187
- PseudoLazy, 271, 332
- publish, 332, 384
- PubSub, 271, 332

PubSub.Subscriber, 333
putByte, 319

‘-Q’, 161
‘-q’, 157, 162
quality?, 399
‘--quiet’, 157, 162
quit, 301
‘--quit’, 161
quote, 226

‘-R’, 159
r, 402
‘-r’, 159, 160
radix, 262
random, 258, 305
range, 291
RangeIterable, 271, 334
rate?, 397
ratio, 400
readable, 322
reboot, 355
receive, 301
received, 345
‘--reconstruct’, 159
redefinitionMode, 355
reduce, 288
Regexp, 271, 335
release, 339
relocatable, 117
remain, 400, 402
‘--remote’, 160
remote mode, 133
remove, 250, 275, 292
removeBack, 292
removeById, 292
removeFront, 292
removeProperty, 314
removeProto, 314
removeSlot, 315
rename, 250
replace, 348
reqStruct, 386
request, 386

reset, 281
resolution, 397
‘--resolution=*resolution*’, 159
resolution?, 399
resStruct, 386
reverse, 292
RGBLed, 402
robot, 403
roll, 406
root, 118
Ros, 381
Ros.Service, 386
Ros.Topic, 383
RotationalMotor, 396
RotationalSpeedMotor, 396
round, 258, 305
routine, 189
run, 330
runningcommands, 281
runtime path, 122
runTo, 330

‘-s’, 157, 159, 160, 163
say, 401
‘--scale=*factor*’, 159
scientific notation, 169
scope, 63, 179
scope, 366
scopeTag, 355, 362
script?, 401
‘sdk-remote/libport/test-suite.log’, 148
‘sdk-remote/src/tests/test-suite.log’, 149
search-path, 155
searchFile, 355
second, 320, 374
seconds, 242
Semaphore, 271, 338
send, 301
sender, 224
Sensor, 396
seq, 258, 281
serial, 394
Server, 340
Server, 272, 340

- Service, 382
- services, 382
- set, 239
- setConstSlot, 315
- setenv, 355
- setProperty, 275, 315
- setProtos, 316
- setSideEffectFree, 279
- setSlot, 316
- '--shared', 163
- shiftedTime, 356
- shoulder, 404
- shutdown, 356
- sign, 258, 305
- signal, 217
- signalAll, 218
- Sin, 373
- Sin, 369
- sin, 258, 306
- Singleton, 272, 341
- singleton, 341
- size, 239, 281, 293, 348, 375
- sleep, 356
- slot, 67, 186
- slotNames, 316
- Smooth, 373
- Smooth, 370
- Socket, 342
- Socket, 272, 342
- sockets, 341
- sort, 293
- spawn, 356
- speaker, 407
- spec, 264
- speech, 407
- SpeechRecognizer, 401
- Speed, 370
- speed, 395–397
- speed?, 401
- SpeedAdaptive, 373
- spine, 406
- sqr, 258, 306
- sqrt, 259, 306
- srandom, 259, 306
- stack size, 157
- '--stack-size=size', 157
- StackFrame, 337
- '--start', 160
- status, 279, 330
- stderr, 330
- stdin, 330
- stdout, 331
- stop, 359
- stop, 366
- strict, 282
- String, 272, 345
- string, 171, 322, 345
- strlen, 282
- structural pattern matching, 190
- structure, 383
- structure tree, 389
- subscribe, 333, 383
- Subscriber, 333
- subscriberCount, 383
- subscribers, 333
- subset, 293
- syncline, 166
- syncWrite, 345
- System, 272, 350
- system, 357
- System.PackageInfo, 358
- System.Platform, 358
- Tag, 272, 359
- tag, 131, 359
- taglist, 282
- tags, 279, 366
- tail, 293, 405
- tan, 259, 306
- target, 64
- target, 224
- target value, 214, 368
- temperature, 397
- TemperatureSensor, 397
- terminate, 279
- 'tests/test-suite.log', 149
- TextToSpeech, 401

- thanks, 301
- third, 374
- thread-safety, 132
- threshold, 400
- Time, 373
- Time, 371
- time, 357
- TimeAdaptive, 373
- Timeout, 272, 367
- times, 259
- timeShift, 280
- timestamp, 235
- toe, 405
- toggle, 297
- toLower, 349
- Topic, 382
- topics, 382
- topLevel, 227
- torque, 396
- torso, 406
- touch, 407
- TouchSensor, 396
- toUpper, 349
- Tracker, 398
- trajectory, 214
- TrajectoryGenerator, 272, 368
- trigger, 243
- triple, 373
- Triplet, 272, 373
- triplet, 373
- true, 220, 272
- trunc, 259, 306
- Tuple, 272, 374
- tuple, 374
- tuples, 172
- turn, 396, 398
- type, 316, 394
- uid, 317
- unacceptVoid, 379
- unacceptvoid, 317
- unadvertise, 385
- unblock, 361
- unblock, 366
- undefall, 282
- unfreeze, 366
- unsetenv, 358
- unstrict, 282
- unsubscribe, 333, 384
- UObject, 3, 132
- UObject, 272, 376
- uobjects, 272
- updateSlot, 275, 317
- uppercase, 264
- urbi, 27
- urbi-launch, 27
- 'URBI.INI', 156
- 'urbi.stamp', 148
- URBI_CONSOLE_MODE, 147
- URBI_DESUGAR, 147
- URBI_IGNORE_URBI_U, 147
- URBI_LAUNCH, 147
- URBI_NO_ICE_CATCHER, 147
- URBI_PARSER, 147
- URBI_PATH, 156
- URBI_REPORT, 147
- URBI_ROOT, 156
- URBI_ROOT_LIB*name*, 147
- URBI_SCANNER, 147
- URBI_TOPLEVEL, 147
- URBI_UOBJECT_PATH, 156
- uservars, 282
- UTF-8, 165
- UValue, 272, 377
- UVar, 132
- UVar, 272, 377
- '-V', 162
- v, 340
- '-v', 162
- val, 395–400, 402
- value, 285
- variable, 63, 415
 - local, 181
- variadic, 178
- vars, 282
- '--verbose', 162
- '--version', 157, 159–162

VideoIn, 398
visible, 400
voice, 407
voice?, 401
voicexml?, 401
void, 272, 378
voltage, 402
volume?, 400

'-w', 158
wait, 218
waitForChanges, 280
waitForTermination, 280
wall, 273, 301
warn, 273
warning, 227
wb?, 399
wheel, 404
width, 264, 398
wrist, 404
writable, 322
write, 301, 345

x, 400, 406
xfov, 398

y, 400, 406
yaw, 398, 405
yfov, 398

z, 406
zip, 293