

The Urbi Software Development Kit

Version 3.0.0

January 31, 2013
(Revision 3.0)

Urbi is an open source software component platform. It features a C++/Java middleware, UObject, to interface components, and an innovative scripting language, urbiscript, with built-in support for parallel and event-based programming.

Chapter 1

Introduction

Urbi SDK is a fully-featured environment to orchestrate complex organizations of components. It is an open source framework for complex systems. It relies on a middleware architecture that coordinates components named UObjects. It also features urbiscript, a scripting language tailored to write orchestration programs.

1.1 Urbi and UObjects

Urbi makes the orchestration of independent and 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.

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 The Big Picture

The [Figure 1.1](#) shows the architecture of Urbi. Let’s browse it bottom up.

At the lowest level, Urbi requires a (possibly very limited) embedded computer. This is the case for most robots today, but on occasion, some device cannot even run reasonably small pieces of code. In that case, Urbi can still be used, but then the robot is actually remote-controlled from a computer running Urbi.

Right on top of the hardware, is running the *Operating System*. Urbi supports the major OSes; it was also ported on top of real-time OSes such as Xenomai, and on specific OSes such as Aperios, Sony’s proprietary system running its Aibo robotic dog.

The *Urbi Runtime*, which is the actual core of the system, also known as the *engine* or the *kernel*, is interfacing the OS with the rest of the Urbi world, urbiscript and UObjects.

UObjects are used to bind hardware or software components, such as actuators and sensors on the one hand, and voice synthesis or face recognition on the other hand. They can be run locally on the robot, or on a remote, more powerful, computer.

To orchestrate all the components, urbiscript is a programming language of choice (see below).

1.3 Urbi and urbiscript

urbiscript is a programming language primarily designed to handle concurrent programming. 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, C++, Java, or JavaScript, 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.

1.4 Genesis

Urbi was 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. Gostai has been acquired by Aldebaran Robotics in 2012. Urbi is now a community-driven open source project available at <https://github.com/urbiforge/urbi>.

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 in Gostai was 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 Many people contributed significantly to Urbi, including Alexandre Morgand, Romain Bezut, Thomas Moulard, Clément Moussu, Nicolas Pierron.

1.5 Outline

This multi-part document provides a complete guide to Urbi. See [Chapter 29](#) for the various notations that are used in the document.

Part IV — 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 I](#)), or the reference manual ([Chapter 21](#)).

[Chapter 25 — Quick Start](#)

This chapter, self-contained, shows the potential of Urbi used as a middleware.

[Chapter 26 — The UObject API](#)

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

[Chapter 27 — The UObject Java API](#)

UObjects can also be written in Java. This section demonstrates it all.

[Chapter 28 — Use Cases](#)

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

Part I — 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 IV](#)). The reference manual contains a terse and complete definition of the Urbi environment ([Part III](#)).

[Chapter 4 — First Steps](#)

First contacts with urbiscript.

[Chapter 5 — Basic Objects, Value Model](#)

A quick introduction to objects and values.

[Chapter 6 — Flow Control Constructs](#)

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

Chapter 7 — Advanced Functions and Scoping

Details about functions, scopes, and lexical closures.

Chapter 8 — 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 II — 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 binary packages.

Chapter 15 — Frequently Asked Questions

Some answers to common questions.

Chapter 16 — Urbi Guideline

Based on our own experience, and code that users have submitted to us, we suggest a programming guideline for Urbi SDK.

Chapter 17 — 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 19 — Building Urbi SDK

Building Urbi SDK from the sources. How to install it, how to check it and so forth. This chapter is meant only for people who want to *build* Urbi SDK, which is orders of magnitude more delicate than simply installing the binary packages from Gostai.

Part III — Urbi SDK Reference Manual

This part defines the specifications of the urbiscript language. 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. For an urbiscript Tutorial, see [Part I](#).

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

Chapter 20 — 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 21 — urbiscript Language Reference Manual

Core constructs of the language and their semantics.

Chapter 22 — urbiscript Standard Library

The classes and methods provided in the standard library.

Chapter 23 — 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>.

Chapter 24 — Urbi 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 29 — Notations

Conventions used in the type-setting of this document.

Chapter 30 — Grammar

Grammar of the urbiscript language.

Chapter 32 — Licenses

Licenses of components used in Urbi SDK.

Chapter 31 — Release Notes

Also known as the Urbi ChangeLog, or Urbi NEWS, this chapter lists the user-visible changes in Urbi SDK releases.

Chapter 33 — Bibliography

References to other documents such as documentation, scientific papers, etc.

Chapter 34 — Glossary

Definition of the terms used in this document.

Chapter 35 — List of Tables

Index of all the tables: list of keywords, operators, etc.

Chapter 36 — List of Figures

Index of all the figures: snapshots, schema, etc.

1.6 Documentation

- Pdf documentation is available here :

<https://github.com/urbiforge/urbi/blob/master/doc/urbi-sdk.pdf>

- An online version has been made available at:

<http://urbi.jcbaillie.net/doc>

- To build the doc, run the ./doc.sh script (the result should be in build-doc folder then)

- The HTML archive version is here (just uncompress an open locally):

<https://github.com/urbiforge/urbi/blob/master/doc/urbi-sdk.html.tar.gz>

- To compile Urbi, please check the README file on the github repository:

<https://github.com/urbiforge/urbi>

- You may be interested to use the Urbi docker container located here:

<https://hub.docker.com/r/urbiforge/urbi>

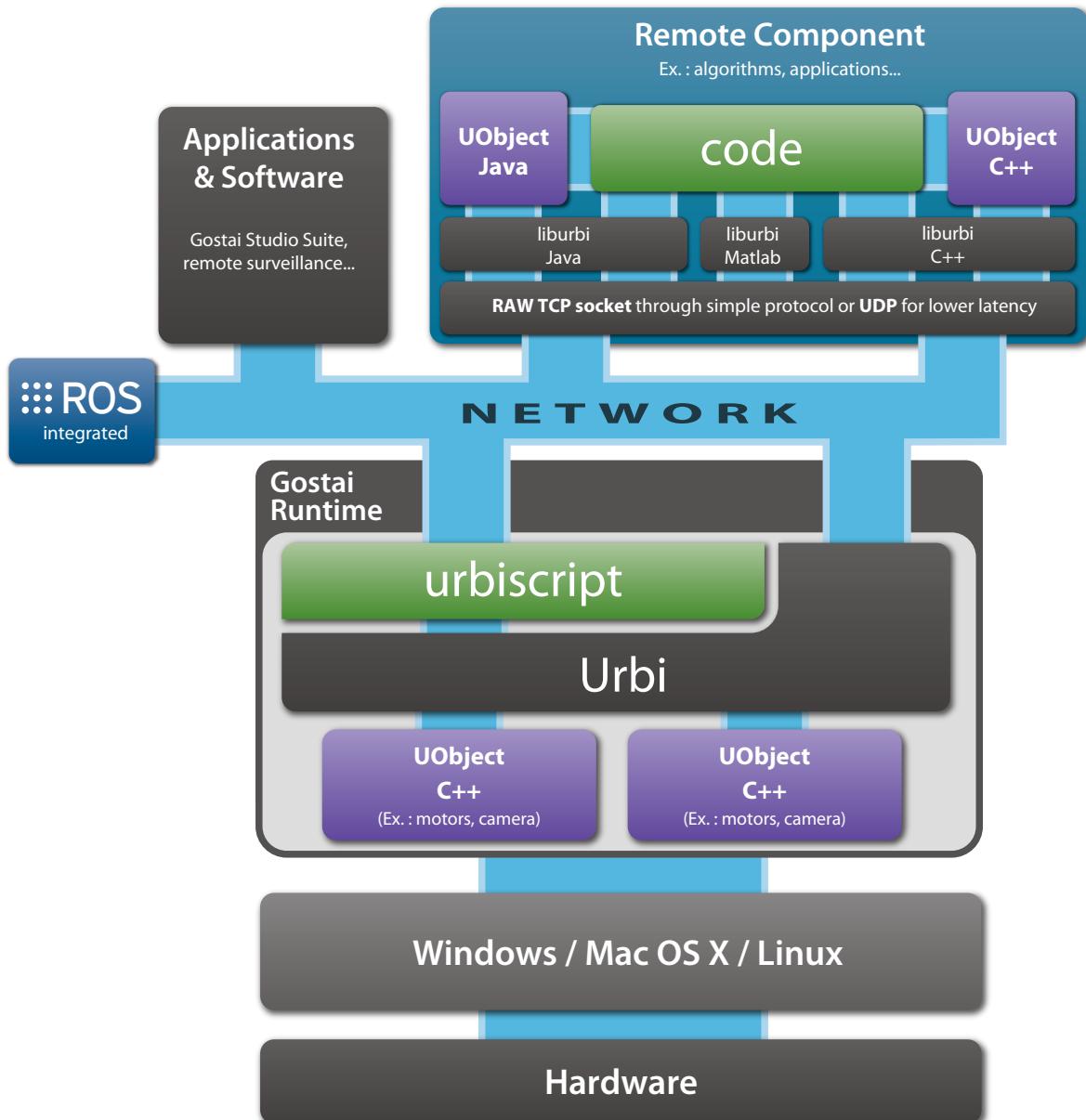


Figure 1.1: A Bird-View of the Urbi Architecture

Chapter 2

Contents

1	Introduction	3
1.1	Urbi and UObjects	3
1.2	The Big Picture	3
1.3	Urbi and urbscript	4
1.4	Genesis	4
1.5	Outline	5
1.6	Documentation	7
2	Contents	9
3	Getting Started	29
I urbscript User Manual		33
4	First Steps	37
4.1	Comments	37
4.2	Literal values	37
4.3	Function calls	38
4.4	Variables	39
4.5	Scopes	39
4.6	Method calls	39
4.7	Function definition	40
4.8	Conclusion	41
5	Basic Objects, Value Model	43
5.1	Objects in urbscript	43
5.2	Methods	46
5.3	Everything is an object	47
5.4	The urbscript values model	47
5.5	Conclusion	49
6	Flow Control Constructs	51
6.1	if	51
6.2	while	51
6.3	for	52
6.4	switch	52
6.5	do	52

7 Advanced Functions and Scoping	55
7.1 Scopes as expressions	55
7.2 Advanced scoping	55
7.3 Local functions	56
7.4 Lexical closures	56
8 Objective Programming, urbiscript Object Model	57
8.1 Prototype-Based Programming in urbiscript	57
8.2 Prototypes and Slot Lookup	58
8.3 Copy on Write	60
8.4 Defining Pseudo-Classes	60
8.5 Constructors	62
8.6 Operators	62
8.7 Properties	62
8.7.1 Features of Values	63
8.7.2 Features of Slots	63
8.8 Getters and setters	64
8.8.1 When to use properties?	64
9 Organizing your code using package and import	67
9.1 Why object lookup is not enough	67
9.2 Import and package	67
10 Functional Programming	69
10.1 First class functions	69
10.2 Lambda functions	69
10.3 Lazy arguments	70
11 Parallelism, Concurrent Flow Control	73
11.1 Parallelism operators	73
11.2 Detach	74
11.3 Tags for parallel control flows	75
11.4 timeout, freezeif and stopif	76
11.4.1 timeout	77
11.4.2 stopif and freezeif	77
11.5 Advanced example with parallelism and tags	77
12 Event-based Programming	79
12.1 Watchdog constructs	79
12.2 Events	80
12.2.1 Emitting Events	80
12.2.2 Emitting events with a payload	80
12.2.3 waituntil	82
13 Urbi for ROS Users	83
13.1 Communication on topics	83
13.1.1 Starting a process from Urbi	83
13.1.2 Listening to Topics	83
13.1.3 Advertising on Topics	84
13.1.3.1 Simple Talker	84
13.1.3.2 Turtle Simulation	85
13.2 Using Services	86
13.3 Image Publisher from ROS to Urbi	86
13.4 Image Subscriber from Urbi to ROS	88

13.5 Remote communication	89
II Guidelines and Cook Books	91
14 Installation	95
14.1 Download	95
14.1.1 Download Urbi 3.0.0	95
14.1.2 Download a Specific Version of Urbi	95
14.2 Install & Check	96
14.2.1 GNU/Linux and Mac OS X	96
14.2.2 Windows	97
15 Frequently Asked Questions	99
15.1 Build Issues	99
15.1.1 Complaints about ‘+=’	99
15.1.2 error: ‘<anonymous>’ is used uninitialized in this function	99
15.1.3 AM.LANGINFO.CODESET	99
15.1.4 configure: error: The Java VM java failed	99
15.1.5 ‘make check’ fails	100
15.1.6 check: error: Unable to load native library: libjava.jnilib	100
15.2 Troubleshooting	100
15.2.1 error while loading shared libraries: libport.so	100
15.2.2 Error 1723: “A DLL required for this install to complete could not be run.”	100
15.2.3 When executing a program, the message “The system cannot execute the specified program.” is raised	101
15.2.4 When executing a program, the message “This application has failed to start” is raised	101
15.2.5 The server dies with “stack exhaustion”	101
15.2.6 ‘myuobject: file not found’. What can I do?	101
15.2.6.1 Getting a better diagnostic	101
15.2.6.2 GNU/Linux	101
15.2.6.3 Mac OS X	102
15.2.6.4 Windows	103
15.3 urbiscript	104
15.3.1 Objects lifetime	104
15.3.1.1 How do I create a new Object derivative?	104
15.3.1.2 How do I destroy an Object?	104
15.3.2 Slots and variables	105
15.3.2.1 Is the lobby a scope?	105
15.3.2.2 How do I add a new slot in an object?	108
15.3.2.3 How do I modify a slot of my object?	109
15.3.2.4 How do I create or modify a local variable?	109
15.3.2.5 How do I make a constructor?	109
15.3.2.6 How do I call a constructor of a parent class?	109
15.3.2.7 How can I manipulate the list of prototypes of my objects?	110
15.3.2.8 How can I know the slots available for a given object?	110
15.3.2.9 How do I create a new function?	110
15.3.3 Tags	111
15.3.3.1 How do I create a tag?	111
15.3.3.2 How do I stop a tag?	111
15.3.3.3 Can tagged statements return a value?	111
15.3.4 Events	111

15.3.4.1 How do I create an event?	111
15.3.4.2 How do I emit an event?	111
15.3.4.3 How do I catch an event?	111
15.3.5 Standard Library	112
15.3.5.1 How can I iterate over a list?	112
15.4 UObjects	112
15.4.1 Is the UObject API Thread-Safe?	112
15.4.1.1 Plugin mode	112
15.4.1.2 Remote mode	112
15.5 Miscellaneous	113
15.5.1 What has changed since the latest release?	113
15.5.2 How can I contribute to the code?	113
15.5.3 How do I report a bug?	114
16 Urbi Guideline	115
16.1 urbiscript Programming Guideline	115
16.1.1 Prefer Expressions to Statements	115
16.1.2 Avoid <code>return</code>	115
17 Migration from urbiscript 1 to urbiscript 2	117
17.1 <code>\$(Foo)</code>	117
17.2 <code>delete Foo</code>	117
17.3 <code>emit Foo</code>	118
17.4 <code>eval(Foo)</code>	118
17.5 <code>foreach</code>	118
17.6 <code>group</code>	118
17.7 <code>loopn</code>	119
17.8 <code>new Foo</code>	119
17.9 <code>self</code>	119
17.10 <code>stop Foo</code>	119
17.11 <code># line</code>	119
17.12 <code>tag+end</code>	119
18 Migration from urbiscript 2 to urbiscript 3	121
18.1 Automatic function evaluation without parenthesis	121
18.2 <code>getSlot</code> and <code>setSlot</code>	121
18.3 Packages	121
19 Building Urbi SDK	123
19.1 Building	123
19.1.1 Getting and installing qibuild	123
19.1.2 Setting up an urbi worktree	123
19.1.3 Building	123
19.1.4 Modifying the grammar or the AST	124
19.1.5 Installing	124
19.2 Run	124
19.3 Check	125
19.4 Debug	125
19.4.1 GDB commands	125

III Urbi SDK Reference Manual	129
20 Programs	133
20.1 Environment Variables	133
20.1.1 Search Path Variables	133
20.1.2 Environment Variables	133
20.2 Special Files	135
20.3 <code>urbi</code> — Running an Urbi Server	135
20.3.1 Options	135
20.3.2 Quitting	137
20.4 <code>urbi-image</code> — Querying Images from a Server	137
20.4.1 Options	137
20.5 <code>urbi-launch</code> — Running a UObject	138
20.5.1 Invoking <code>urbi-launch</code>	138
20.5.2 Examples	139
20.5.3 UObject Module Options	139
20.6 <code>urbi-launch-java</code> — Running a Java UObject	140
20.6.1 Invoking <code>urbi-launch-java</code>	140
20.7 <code>urbi-ping</code> — Checking the Delays with a Server	141
20.7.1 Options	141
20.8 <code>urbi-send</code> — Sending urbiscript Commands to a Server	142
20.9 <code>urbi-sound</code> — Querying Sounds from a Server	142
20.9.1 Options	143
20.10 <code>umake</code> — Compiling UObject Components	143
20.10.1 Invoking <code>umake</code>	143
20.10.2 <code>umake</code> Wrappers	145
21 urbiscript Language Reference Manual	147
21.1 Syntax	147
21.1.1 Characters, encoding	147
21.1.2 Comments	147
21.1.3 Synclines	148
21.1.4 Identifiers	148
21.1.5 Keywords	149
21.1.6 Literals	149
21.1.6.1 Angles	149
21.1.6.2 Dictionaries	149
21.1.6.3 Durations	149
21.1.6.4 Floats	150
21.1.6.5 Lists	151
21.1.6.6 Strings	151
21.1.6.7 Tuples	152
21.1.6.8 Pseudo classes	152
21.1.7 Statement Separators	154
21.1.7.1 ';'	154
21.1.7.2 ','	154
21.1.7.3 ' '	155
21.1.7.4 '&'	155
21.1.8 Operators	155
21.1.8.1 Arithmetic operators	156
21.1.8.2 Assignment operators	156
21.1.8.3 Increment/decrement Operators	158
21.1.8.4 Bitwise operators	158

21.1.8.5 Logical operators	159
21.1.8.6 Comparison operators	160
21.1.8.7 Container operators	161
21.1.8.8 Object operators	161
21.1.8.9 All operators summary	161
21.2 Scopes and local variables	161
21.2.1 Scopes	161
21.2.2 Local variables	162
21.3 Functions	163
21.3.1 Function Definition	163
21.3.2 Arguments	164
21.3.2.1 Default value	164
21.3.2.2 Argument typing	164
21.3.3 Return value	165
21.3.4 Call messages	166
21.3.5 Strictness	166
21.3.6 Closures	167
21.3.7 Variadic functions	168
21.4 Objects	168
21.4.1 Slots	169
21.4.1.1 Manipulation	169
21.4.1.2 Syntactic Sugar	169
21.4.2 Properties	169
21.4.2.1 Manipulation	169
21.4.3 Direct slot access	170
21.4.3.1 Getting and setting slots	170
21.4.3.2 Properties	170
21.4.4 Other features	170
21.4.5 Prototypes	170
21.4.5.1 Manipulation	170
21.4.5.2 Inheritance	171
21.4.5.3 Copy on write	171
21.4.6 Sending messages	172
21.5 Enumeration types	173
21.6 Structural Pattern Matching	173
21.6.1 Basic Pattern Matching	174
21.6.2 Variable	174
21.6.3 Guard	175
21.7 Imperative flow control	176
21.7.1 break	176
21.7.2 continue	176
21.7.3 do	177
21.7.4 for	177
21.7.5 C-like for	178
21.7.5.1 for;	178
21.7.5.2 for 	179
21.7.6 Range-for	179
21.7.6.1 Range-for;	179
21.7.6.2 Range-for 	180
21.7.7 Anonymous range-for	180
21.7.7.1 Anonymous Range-for;	180
21.7.7.2 Anonymous Range-for 	180
21.7.8 if	180

21.7.9 <code>loop</code>	181
21.7.9.1 <code>loop;</code>	181
21.7.9.2 <code>loop </code>	181
21.7.10 <code>switch</code>	182
21.7.11 <code>while</code>	182
21.7.11.1 <code>while;</code>	183
21.7.11.2 <code>while </code>	183
21.8 Exceptions	184
21.8.1 Throwing exceptions	184
21.8.2 Catching exceptions	184
21.8.3 Inspecting exceptions	185
21.8.4 Finally	186
21.8.4.1 Regular execution	186
21.8.4.2 Control-flow	186
21.8.4.3 Tags	187
21.8.4.4 Exceptions	187
21.9 Assertions	188
21.9.1 Asserting an Expression	188
21.9.2 Assertion Blocks	189
21.10 Parallel Flow Control	190
21.10.1 Tagging	190
21.10.2 <code>every</code>	191
21.10.2.1 <code>every;</code>	191
21.10.2.2 <code>every </code>	192
21.10.2.3 <code>every,</code>	192
21.10.3 <code>C-for</code> ,	192
21.10.4 <code>Range-for&</code>	193
21.10.5 <code>Anonymous range-for&</code>	194
21.10.6 <code>loop,</code>	194
21.10.7 <code>timeout</code>	194
21.10.8 <code>stopif</code>	196
21.10.9 <code>freezeif</code>	196
21.10.10 <code>while,</code>	196
21.11 Event Handling	197
21.11.1 <code>at</code>	197
21.11.1.1 <code>at on Events</code>	198
21.11.1.2 <code>at on Boolean Expressions</code>	198
21.11.1.3 Synchronous and asynchronous <code>at</code>	199
21.11.1.4 Execution Context	200
21.11.1.5 Tags and <code>at</code>	201
21.11.1.6 Scoped <code>at</code>	202
21.11.2 <code>waituntil</code>	202
21.11.2.1 <code>waituntil on Events</code>	202
21.11.2.2 <code>waituntil on Boolean Expressions</code>	203
21.11.3 <code>watch</code>	204
21.11.4 <code>whenever</code>	204
21.11.4.1 <code>whenever on Events</code>	205
21.11.4.2 <code>whenever on Boolean Expressions</code>	206
21.12 Trajectories	207
21.13 Garbage Collection and Limitations	208

22 urbiscript Standard Library	215
22.1 Barrier	216
22.1.1 Prototypes	216
22.1.2 Construction	216
22.1.3 Slots	216
22.2 Binary	218
22.2.1 Prototypes	218
22.2.2 Construction	218
22.2.3 Slots	218
22.3 Boolean	220
22.3.1 Truth Values	220
22.3.2 Prototypes	220
22.3.3 Construction	220
22.3.4 Slots	220
22.4 CallMessage	222
22.4.1 Examples	222
22.4.1.1 Evaluating an argument several times	222
22.4.1.2 Strict Functions	222
22.4.2 Prototypes	222
22.4.3 Slots	222
22.5 Channel	226
22.5.1 Prototypes	226
22.5.2 Construction	226
22.5.3 Slots	226
22.6 Code	229
22.6.1 Prototypes	229
22.6.2 Construction	229
22.6.3 Slots	229
22.7 Comparable	232
22.7.1 Example	232
22.7.2 Prototypes	232
22.7.3 Slots	232
22.8 Container	233
22.8.1 Prototypes	233
22.8.2 Slots	233
22.9 Control	234
22.9.1 Prototypes	234
22.9.2 Slots	234
22.10 Date	236
22.10.1 Prototypes	236
22.10.2 Construction	236
22.10.3 Slots	236
22.11 Dictionary	240
22.11.1 Example	240
22.11.2 Hash values	240
22.11.3 Prototypes	240
22.11.4 Construction	240
22.11.5 Slots	241
22.12 Directory	246
22.12.1 Prototypes	246
22.12.2 Construction	246
22.12.3 Slots	246
22.13 Duration	252

22.13.1 Prototypes	252
22.13.2 Construction	252
22.13.3 Slots	252
22.14 Enumeration	253
22.14.1 Examples	253
22.14.2 Prototypes	253
22.14.3 Construction	253
22.14.4 Slots	253
22.15 Event	255
22.15.1 Examples	255
22.15.2 Synchronicity of Event Handling	255
22.15.3 Sustained Events	255
22.15.4 Prototypes	256
22.15.5 Construction	256
22.15.6 Slots	256
22.16 Exception	260
22.16.1 Prototypes	260
22.16.2 Construction	260
22.16.3 Slots	260
22.16.3.1 General Features	260
22.16.3.2 Specific Exceptions	261
22.17 Executable	265
22.17.1 Prototypes	265
22.17.2 Construction	265
22.17.3 Slots	265
22.18 File	266
22.18.1 Prototypes	266
22.18.2 Construction	266
22.18.3 Slots	266
22.19 Finalizable	270
22.19.1 Example	270
22.19.2 Prototypes	270
22.19.3 Construction	270
22.19.4 Slots	271
22.20 Float	272
22.20.1 Prototypes	272
22.20.2 Construction	272
22.20.3 Slots	273
22.21 Float.limits	281
22.21.1 Prototypes	281
22.21.2 Slots	281
22.22 FormatInfo	283
22.22.1 Prototypes	283
22.22.2 Construction	283
22.22.3 Slots	284
22.23 Formatter	287
22.23.1 Prototypes	287
22.23.2 Construction	287
22.23.3 Slots	287
22.24 Global	289
22.24.1 Prototypes	289
22.24.2 Slots	289
22.25 Group	295

22.25.1 Example	295
22.25.2 Prototypes	295
22.25.3 Construction	295
22.25.4 Slots	295
22.26 Hash	298
22.26.1 Prototypes	298
22.26.2 Construction	298
22.26.3 Slots	298
22.27 InputStream	299
22.27.1 Prototypes	299
22.27.2 Construction	299
22.27.3 Slots	299
22.28 IoService	302
22.28.1 Example	302
22.28.2 Prototypes	302
22.28.3 Construction	302
22.28.4 Slots	302
22.29 Job	304
22.29.1 Prototypes	304
22.29.2 Construction	304
22.29.3 Slots	304
22.30 Kernel1	309
22.30.1 Prototypes	309
22.30.2 Construction	309
22.30.3 Slots	309
22.31 Lazy	311
22.31.1 Examples	311
22.31.1.1 Evaluating once	311
22.31.1.2 Evaluating several times	311
22.31.2 Caching	311
22.31.3 Prototypes	312
22.31.4 Construction	312
22.31.5 Slots	312
22.32 List	314
22.32.1 Prototypes	314
22.32.2 Construction	314
22.32.3 Slots	314
22.33 Loadable	325
22.33.1 Example	325
22.33.2 Prototypes	325
22.33.3 Construction	325
22.33.4 Slots	325
22.34 Lobby	327
22.34.1 Examples	327
22.34.2 Prototypes	327
22.34.3 Construction	327
22.34.4 Slots	327
22.35 Location	331
22.35.1 Prototypes	331
22.35.2 Construction	331
22.35.3 Slots	331
22.36 Logger	333
22.36.1 Examples	333

22.36.2 Existing Categories	333
22.36.3 Prototypes	333
22.36.4 Construction	334
22.36.5 Slots	334
22.37 Math	338
22.37.1 Prototypes	338
22.37.2 Construction	338
22.37.3 Slots	338
22.38 Matrix	341
22.38.1 Prototypes	341
22.38.2 Construction	341
22.38.3 Slots	342
22.39 Mutex	352
22.39.1 Prototypes	352
22.39.2 Construction	352
22.39.3 Slots	353
22.40 nil	354
22.40.1 Prototypes	354
22.40.2 Construction	354
22.40.3 Slots	354
22.41 Object	355
22.41.1 Prototypes	355
22.41.2 Construction	355
22.41.3 Slots	355
22.42 Orderable	369
22.42.1 Example	369
22.42.2 Prototypes	369
22.42.3 Slots	369
22.43 OutputStream	370
22.43.1 Prototypes	370
22.43.2 Construction	370
22.43.3 Slots	370
22.44 Pair	372
22.44.1 Prototypes	372
22.44.2 Construction	372
22.44.3 Slots	372
22.45 Path	373
22.45.1 Prototypes	373
22.45.2 Construction	373
22.45.3 Slots	373
22.46 Pattern	377
22.46.1 Prototypes	377
22.46.2 Construction	377
22.46.3 Slots	377
22.47 Position	379
22.47.1 Prototypes	379
22.47.2 Construction	379
22.47.3 Slots	379
22.48 Primitive	381
22.48.1 Prototypes	381
22.48.2 Construction	381
22.48.3 Slots	381
22.49 Process	382

22.49.1 Example	382
22.49.2 Prototypes	382
22.49.3 Construction	382
22.49.4 Slots	383
22.50 Profile	385
22.50.1 Example	385
22.50.1.1 Basic profiling	385
22.50.1.2 Asynchronous profiling	385
22.50.2 Prototypes	386
22.50.3 Construction	387
22.50.4 Slots	387
22.51 Profile.Function	388
22.51.1 Prototypes	388
22.51.2 Construction	388
22.51.3 Slots	388
22.52 PseudoLazy	390
22.52.1 Prototypes	390
22.52.2 Slots	390
22.53 PubSub	391
22.53.1 Prototypes	391
22.53.2 Construction	391
22.53.3 Slots	391
22.54 PubSub.Subscriber	392
22.54.1 Prototypes	392
22.54.2 Construction	392
22.54.3 Slots	392
22.55 RangeIterable	393
22.55.1 Prototypes	393
22.55.2 Slots	393
22.56 Regexp	395
22.56.1 Prototypes	395
22.56.2 Construction	395
22.56.3 Slots	395
22.57 Semaphore	398
22.57.1 Prototypes	398
22.57.2 Construction	398
22.57.3 Slots	398
22.58 Serializables	400
22.58.1 Prototypes	400
22.58.2 Slots	400
22.59 Server	401
22.59.1 Prototypes	401
22.59.2 Construction	401
22.59.3 Slots	401
22.60 Singleton	402
22.60.1 Prototypes	402
22.60.2 Construction	402
22.60.3 Slots	402
22.61 Slot	403
22.61.1 Accessing the slot object	403
22.61.2 Key features	403
22.61.3 Split mode	403
22.61.4 Construction	403

22.61.5 Prototypes	403
22.61.6 Slots	403
22.62 Socket	410
22.62.1 Example	410
22.62.2 Prototypes	412
22.62.3 Construction	412
22.62.4 Slots	412
22.63 StackFrame	414
22.63.1 Prototypes	414
22.63.2 Construction	414
22.63.3 Slots	414
22.64 Stream	416
22.64.1 Prototypes	416
22.64.2 Construction	416
22.64.3 Slots	416
22.65 String	417
22.65.1 Prototypes	417
22.65.2 Construction	417
22.65.3 Slots	417
22.66 Subscription	425
22.66.1 Prototypes	425
22.66.2 Construction	425
22.66.3 Slots	425
22.67 System	426
22.67.1 Prototypes	426
22.67.2 Slots	426
22.68 System.PackageInfo	436
22.68.1 Prototypes	436
22.68.2 Slots	436
22.69 System.Platform	439
22.69.1 Prototypes	439
22.69.2 Slots	439
22.70 Tag	440
22.70.1 Examples	440
22.70.1.1 Stop	440
22.70.1.2 Block/unblock	440
22.70.1.3 Freeze/unfreeze	441
22.70.1.4 Scope tags	442
22.70.1.5 Enter/leave events	442
22.70.1.6 Begin/end	444
22.70.2 Hierarchical tags	444
22.70.3 Prototypes	445
22.70.4 Construction	445
22.70.5 Slots	445
22.71 Timeout	447
22.71.1 Examples	447
22.71.2 Prototypes	447
22.71.3 Construction	447
22.71.4 Slots	448
22.72 Traceable	449
22.72.1 Prototypes	449
22.72.2 Slots	449
22.73 TrajectoryGenerator	450

22.73.1 Examples	450
22.73.1.1 Accel	450
22.73.1.2 Cos	450
22.73.1.3 Sin	450
22.73.1.4 Smooth	451
22.73.1.5 Speed	451
22.73.1.6 Time	452
22.73.1.7 Trajectories and Tags	452
22.73.2 Prototypes	453
22.73.3 Construction	453
22.73.4 Slots	453
22.74 Triplet	455
22.74.1 Prototypes	455
22.74.2 Construction	455
22.74.3 Slots	455
22.75 Tuple	456
22.75.1 Prototypes	456
22.75.2 Construction	456
22.75.3 Slots	456
22.76 UObject	458
22.76.1 Prototypes	458
22.76.2 Slots	458
22.77 uobjects	459
22.77.1 Prototypes	459
22.77.2 Slots	459
22.78 UValue	460
22.78.1 Prototypes	460
22.78.2 Slots	460
22.79 UValueSerializable	461
22.79.1 Example	461
22.79.2 Prototypes	461
22.79.3 Slots	461
22.80 Vector	462
22.80.1 Prototypes	462
22.80.2 Construction	462
22.80.3 Slots	462
22.81 void	468
22.81.1 Prototypes	468
22.81.2 Construction	468
22.81.3 Slots	468
23 Communication with ROS	469
23.1 Ros	469
23.1.1 Construction	469
23.1.2 Slots	469
23.2 Ros.Topic	470
23.2.1 Construction	470
23.2.2 Slots	471
23.2.2.1 Common	471
23.2.2.2 Subscription	471
23.2.2.3 Advertising	471
23.2.3 Example	473
23.3 Ros.Service	474

23.3.1 Construction	474
23.3.2 Slots	474
24 Urbi Standard Robotics API	475
24.1 The Structure Tree	475
24.2 Frame of Reference	476
24.3 Component naming	477
24.4 Localization	477
24.5 Interface	479
24.5.1 AudioIn	479
24.5.2 AudioOut	479
24.5.3 Battery	480
24.5.4 BlobDetector	480
24.5.5 Identity	480
24.5.6 Led	481
24.5.6.1 RGBLed	481
24.5.7 Mobile	481
24.5.7.1 Blocking API	481
24.5.7.2 Speed-control API	482
24.5.7.3 Safety	482
24.5.7.4 State	482
24.5.8 Motor	482
24.5.8.1 LinearMotor	483
24.5.8.2 LinearSpeedMotor	483
24.5.8.3 RotationalMotor	483
24.5.8.4 RotationalSpeedMotor	483
24.5.9 Network	483
24.5.10 Sensor	483
24.5.10.1 AccelerationSensor	484
24.5.10.2 DistanceSensor	484
24.5.10.3 GyroSensor	484
24.5.10.4 Laser	484
24.5.10.5 TemperatureSensor	484
24.5.10.6 TouchSensor	485
24.5.11 SpeechRecognizer	485
24.5.12 TextToSpeech	485
24.5.13 Tracker	486
24.5.14 VideoIn	486
24.6 Standard Components	487
24.6.1 Yaw/Pitch/Roll orientation	487
24.6.2 Standard Component List	487
24.7 Compact notation	491
24.8 Support classes	492
24.8.1 Interface	492
24.8.2 Component	492
24.8.3 Localizer	493
IV Urbi and UObjects User Manual	495
25 Quick Start	499
25.1 UObject Basics	499
25.1.1 The Objects to Bind into Urbi	499

25.1.2 Wrapping into an UObject	500
25.1.3 Running Components	502
25.1.3.1 Compiling	502
25.1.3.2 Running UObject s	503
25.2 Using urbiscript	504
25.2.1 The urbiscript Scripting Language	504
25.2.2 Concurrency	505
25.2.2.1 First Attempt	505
25.2.2.2 Second Attempt: Threaded Functions	506
25.3 Conclusion	506
26 The UObject API	507
26.1 Compiling UObject s	507
26.1.1 Compiling with qibuild	507
26.1.2 Compiling by hand	508
26.1.3 The <code>umake-*</code> family of tools	508
26.1.4 Using the Visual C++ Wizard	509
26.2 Creating a class, binding variables and functions	510
26.3 Creating new instances	511
26.4 Binding functions	511
26.4.1 Simple binding	511
26.4.2 Multiple bindings	511
26.4.3 Asynchronous binding	511
26.5 Notification of a variable change or access	512
26.5.1 Threaded notification	512
26.6 Data-flow based programming: exchanging UVars	513
26.7 Data-flow based programming: InputPort	513
26.7.1 Customizing data-flow links	514
26.8 Timers	514
26.9 The special case of sensor/actuator variables	514
26.10 Using Urbi variables	515
26.11 Emitting events	515
26.12 UObject and Threads	515
26.13 Using binary types	516
26.13.1 UVar conversion and memory management	516
26.13.2 Binary conversion	516
26.13.3 0-copy mode	517
26.14 Direct communication between UObject s	518
26.15 Using hubs to group objects	518
26.16 Sending urbiscript code	518
26.17 Using RTP transport in remote mode	518
26.17.1 Enabling RTP	519
26.17.2 Per-UVar control of RTP mode	519
26.18 Extending the cast system	519
26.18.1 Principle	519
26.18.2 Casting simple structures	519
27 The UObject Java API	521
27.1 Compiling and running UObject s	521
27.1.1 Compiling and running by hand	521
27.1.2 The <code>umake-java</code> and <code>urbi-launch-java</code> tools	522
27.2 Creating a class, binding variables and functions	522
27.3 Creating new instances	524

27.4 Binding functions	525
27.5 Notification of a variable change or access	526
27.6 Timers	526
27.7 Using Urbi variables	526
27.8 Sending Urbi code	526
27.9 Providing a main class or not	527
27.10 Import the examples with Eclipse	527
27.11 Run the UObject Java examples	530
28 Use Cases	533
28.1 Writing a Servomotor Device	533
28.1.1 Caching	535
28.1.2 Using Timers	536
28.2 Using Hubs to Group Objects	536
28.2.1 Alternate Implementation	538
28.3 Writing a Camera Device	538
28.3.1 Optimization in Plugin Mode	540
28.4 Writing a Speaker or Microphone Device	540
28.5 Writing a Softdevice: Ball Detection	540
V Tables and Indexes	543
29 Notations	547
29.1 Words	547
29.2 Frames	547
29.2.1 C++ Code	547
29.2.2 Grammar Excerpts	548
29.2.3 Java Code	548
29.2.4 Shell Sessions	549
29.2.5 urbiscript Sessions	549
29.2.6 urbiscript Assertions	549
30 Grammar	551
31 Release Notes	553
31.1 Urbi SDK 3.0	553
31.1.1 Major changes	553
31.1.2 Fixes	553
31.1.3 Changes	554
31.1.3.1 urbiscript	554
31.1.3.2 urbiscript Standard Library	555
31.1.3.3 Miscellaneous	556
31.1.4 New features	556
31.1.4.1 urbiscript	556
31.1.4.2 urbiscript Standard Library	557
31.1.4.3 Miscellaneous	558
31.1.5 Documentation	558
31.2 Urbi SDK 2.7.5	559
31.2.1 Fixes	559
31.2.2 Changes	559
31.2.3 New Feature	560
31.2.4 Documentation	560

31.3 Urbi SDK 2.7.4	560
31.3.1 Fixes	560
31.3.2 Changes	561
31.3.3 Documentation	561
31.4 Urbi SDK 2.7.3	561
31.4.1 Fixes	561
31.4.2 Changes	562
31.4.3 Documentation	562
31.5 Urbi SDK 2.7.2	562
31.5.1 Fixes	562
31.6 Urbi SDK 2.7.1	562
31.6.1 Fixes	562
31.6.2 Changes	562
31.6.3 Documentation	562
31.7 Urbi SDK 2.7	563
31.7.1 Changes	563
31.7.2 New Features	563
31.7.3 Documentation	564
31.8 Urbi SDK 2.6	564
31.8.1 Fixes	564
31.8.2 Optimizations	564
31.8.3 New Features	565
31.8.4 Documentation	565
31.9 Urbi SDK 2.5	565
31.9.1 Fixes	565
31.9.2 New Features	565
31.9.3 Changes	566
31.9.4 Documentation	567
31.10 Urbi SDK 2.4	567
31.10.1 Fixes	567
31.10.2 New Features	568
31.10.3 Documentation	569
31.11 Urbi SDK 2.3	569
31.11.1 Fixes	569
31.11.2 New Features	569
31.12 Urbi SDK 2.2	570
31.12.1 Fixes	570
31.12.2 New Features	571
31.12.3 Documentation	572
31.13 Urbi SDK 2.1	572
31.13.1 Fixes	572
31.13.2 New Features	572
31.13.3 Optimization	573
31.13.4 Documentation	573
31.14 Urbi SDK 2.0.3	573
31.14.1 New Features	573
31.14.2 Fixes	574
31.14.3 Documentation	574
31.15 Urbi SDK 2.0.2	574
31.15.1 urbascript	574
31.15.2 Fixes	574
31.15.3 Documentation	574
31.16 Urbi SDK 2.0.1	574

31.16.1 urbiscript	575
31.16.2 Documentation	575
31.16.3 Fixes	575
31.17 Urbi SDK 2.0	575
31.17.1 urbiscript	575
31.17.1.1 Changes	575
31.17.1.2 New features	575
31.17.2 UObjects	576
31.17.3 Documentation	576
31.18 Urbi SDK 2.0 RC 4	576
31.18.1 urbiscript	577
31.18.1.1 Changes	577
31.18.1.2 New objects	577
31.18.1.3 New features	577
31.18.2 UObjects	577
31.19 Urbi SDK 2.0 RC 3	577
31.19.1 urbiscript	577
31.19.1.1 Fixes	577
31.19.1.2 Changes	577
31.19.2 Documentation	577
31.20 Urbi SDK 2.0 RC 2	578
31.20.1 Optimization	578
31.20.2 urbiscript	578
31.20.2.1 New constructs	578
31.20.2.2 New objects	578
31.20.2.3 New features	579
31.20.2.4 Fixes	579
31.20.2.5 Deprecations	579
31.20.2.6 Changes	579
31.20.3 UObjects	580
31.20.4 Documentation	580
31.20.5 Various	581
31.21 Urbi SDK 2.0 RC 1	581
31.21.1 Auxiliary programs	581
31.21.2 urbiscript	582
31.21.2.1 Changes	582
31.21.2.2 Fixes	582
31.21.3 URBI Remote SDK	582
31.21.4 Documentation	582
31.22 Urbi SDK 2.0 beta 4	582
31.22.1 Documentation	582
31.22.2 urbiscript	582
31.22.2.1 Bug fixes	582
31.22.2.2 Changes	583
31.22.3 Programs	583
31.22.3.1 Environment variables	583
31.22.3.2 Scripting	583
31.22.3.3 urbi-console	583
31.22.3.4 Auxiliary programs	583
31.23 Urbi SDK 2.0 beta 3	584
31.23.1 Documentation	584
31.23.2 urbiscript	584
31.23.2.1 Fixes	584

31.23.2.2 Changes	584
31.23.3 UObjects	586
31.23.4 Auxiliary programs	586
31.24 Urbi SDK 2.0 beta 2	587
31.24.1 urbiscript	587
31.24.2 Standard library	587
31.24.3 UObjects	588
31.24.4 Run-time	588
31.24.5 Bug fixes	588
31.24.6 Auxiliary programs	589
32 Licenses	591
32.1 Boost Software License 1.0	591
32.2 BSD License	591
32.3 Expat License	592
32.4 gnu.bytecode	592
32.5 ICU License	593
32.6 Independent JPEG Group's Software License	594
32.7 Libcoroutine License	594
32.8 OpenSSL License	595
32.9 ROS	597
32.10 Urbi Open Source Contributor Agreement	599
33 Bibliography	601
34 Glossary	603
35 List of Tables	607
36 List of Figures	609
37 Index	611

Chapter 3

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 version 3.x.y
```

Shell Session

There are several means to interact with a server spawned by ***urbi***, see [Section 20.3](#) 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!
```

Shell Session

Listing 3.1: 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!
```

Shell Session

Listing 3.2: 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  
[00000825] *** Urbi version 3.x.y  
1+2;  
[00001200] 3  
shutdown;
```

Shell Session

Listing 3.3: 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. 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.

Shell Session

```
$ urbi --port 54000 &
[1] 77024
$ telnet localhost 54000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[00004816] *** Urbi version 3.x.y
12345679*8;
[00018032] 98765432
quit;
Connection closed by foreign host.
```

Listing 3.4: An interactive session under Unix.

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

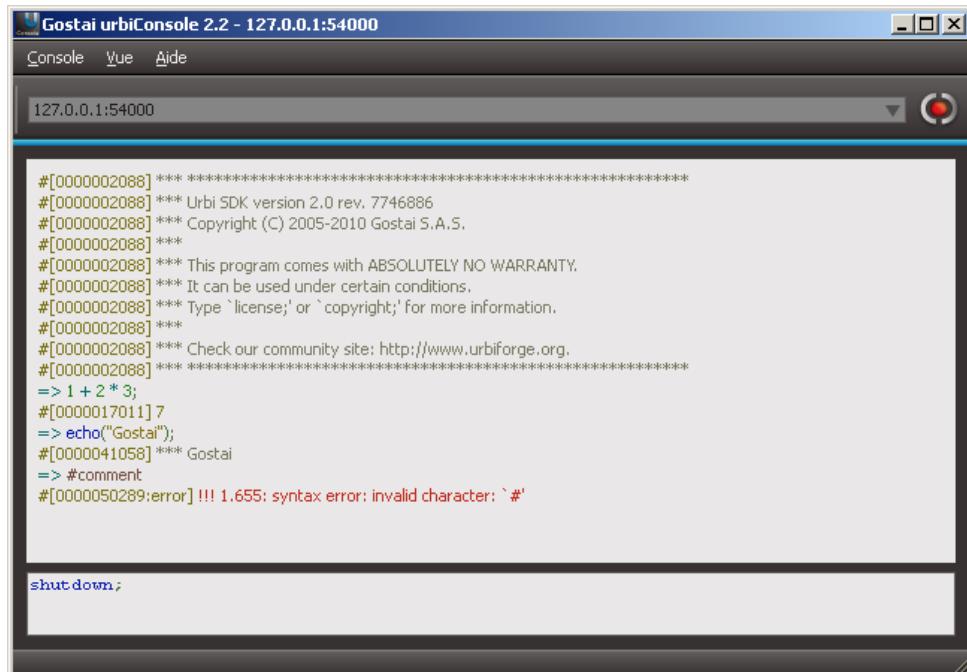
Shell Session

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

Listing 3.5: Starting an interactive session under Windows.

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

Then, the interaction proceeds in the `Gostai Console` 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 20.8](#)) 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;"
```

Shell
Session

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 using the command `shutdown`, and spawn a new one. In interactive mode you can also use the shortcut sequence Ctrl-D, like in many other interpreters.

In case of a foreground task preventing you to execute other commands, you can use Ctrl-C to kill the foreground task, clear queued commands and restore interactive mode.

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

Enjoy!

Part I

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 IV](#)). The reference manual contains a terse and complete definition of the Urbi environment ([Part III](#)).

[Chapter 4 — First Steps](#)

First contacts with urbiscript.

[Chapter 5 — Basic Objects, Value Model](#)

A quick introduction to objects and values.

[Chapter 6 — Flow Control Constructs](#)

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

[Chapter 7 — Advanced Functions and Scoping](#)

Details about functions, scopes, and lexical closures.

[Chapter 8 — 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 4

First Steps

This section expects that you already know how to run `urbi`. If not, please first see [Chapter 3](#).

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.

4.1 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 `*/`. Contrary to C/C++, this type of comments does nest.
- 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  
  
"foo" /* You /* can /* nest */ */ comments. */ "bar";  
[00000000] "foobar"
```

urbiscript
Session

[Chapter 3](#) introduced some of the conventions used in this document: frames such as the previous one denote “urbiscript sessions”, i.e., dialogs between Urbi and you. The output is prefixed by a number between 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. Since these timestamps are irrelevant in documentation, they will often be filled with zeroes. More details about the typesetting of this document (and the other kinds of frames) can be found in [Chapter 29](#).

4.2 Literal values

Several special kinds of “values” can be entered directly with a specific syntax. They are called *literals*, or sometimes *manifest values*. We just met a first kind of literals: integers. There are several others, such as:

- **floats**: floating point numbers.

urbiscript
Session

```
42; // Integer literal.
[00000000] 42

3.14; // Floating point number literal.
[00000000] 3.14
```

- **strings**: character strings.

urbiscript
Session

```
"string";
[00000000] "string"
```

- **lists**: ordered collection of values.

urbiscript
Session

```
[1, 2, "a", "b"];
[00000000] [1, 2, "a", "b"]
```

- **dictionaries**: unordered collection of associations.

urbiscript
Session

```
["a" => 1, "b" => 2, 3 => "three"];
[00000000] [3 => "three", "a" => 1, "b" => 2]
```

- **nil**: neutral value, or value placeholder. Think of it as the value that fits anywhere.

urbiscript
Session

```
nil;
```

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

urbiscript
Session

```
void;
```

These examples highlight some points:

- **Lists** and **dictionaries** in urbiscript are heterogeneous. That is, they can hold values of different types.
- The printing of **nil** and **void** is empty.
- There are many hyperlinks in this document: clicking on names such as [Dictionary](#) will drive you immediately to its specifications. This is also true for slots, such as [String.size](#).

4.3 Function calls

You can call functions with the classical, mathematical notation.

urbiscript
Session

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

urbiscript
Session

```
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.

4.4 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
```

urbiscript
Session

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
```

urbiscript
Session

4.5 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
x;
[00000073:error] !!! lookup failed: x
```

urbiscript
Session

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

4.6 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"]
"foo".length();
[00000000] 3
// Method call can be chained
"".length().cos();
[00000000] 1
```

urbiscript
Session

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

4.7 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.

urbiscript
Session

```
// 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:

urbiscript
Session

```
// 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. When a statement is “missing”, an empty statement `({})` is inserted. So

`code|`; is actually equivalent to `code | {};`, which means “run `code`, then run `{}` and return its value”. Since the value of `{}` is `void`, which is not displayed, this is a means to discard the result of a computation, and avoid that something is printed. Contrast the two following function definitions.

```
function sum(a, b, c)
{
    return a + b + c;
};
[00003553] function (var a, var b, var c) { return a.'+'(b).+''(c) }
function sum2(a, b, c)
{
    return a + b + c;
}|;
sum(20, 2, 20);
[00003556] 42
```

urbiscript
Session

The `return` keyword breaks the control flow of a function (similarly to the way `break` interrupts a loop) and returns the control flow to the caller. It accepts an optional argument, the value to return to the caller.

In urbiscript, if no `return` statement is executed, the value of the last expression is returned. Actually, refrained from using `return` when you don't need it, it is both less readable (once you get used to this programming style), and less efficient (Section 16.1.2).

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

urbiscript
Session

4.8 Conclusion

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

Chapter 5

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 8](#).

5.1 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
```

urbiscript
Session

The `localSlotNames` method lists the names of the slots of an object ([Object](#)).

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

urbiscript
Session

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"
```

urbiscript
Session

It's as simple as this. The `inspect` method provides a convenient short-hand to discover an object ([Object](#)).

```
Foo.inspect();
[00000000] *** Inspecting Foo
[00000000] *** ** Prototypes:
[00000000] ***     Object
[00000000] *** ** Local Slots:
[00000000] ***     a : Float
[00000000] ***     asFoo : Code
[00000000] ***     b : String
[00000000] ***     type : String
```

urbiscript
Session

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`.

urbiscript
Session

```
// 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 `setSlotValue` method, which takes the slot name and its value.

urbiscript
Session

```
o.setSlotValue("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.

urbiscript
Session

```
// This is equivalent to o.setSlotValue("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 `getSlotValue` method, which takes the slot name as argument. The following listing illustrates the use of `getSlotValue` and `setSlotValue` to read and write slots whose names are unknown at code-writing time.

urbiscript
Session

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

function get(object, name)
{
    // We have to use getSlotValue here, since we don't
```

```
// know the actual name of the slot.
object.getSlotValue("x_" + name)
};

var x = Object.clone();
[00000000] Object_0x42342448
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 `getSlotValue` and `setSlotValue`. 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
```

urbiscript
Session

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"
```

urbiscript
Session

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 `setSlotValue` or `var`, is an error. The slot must be defined once with `setSlotValue`, 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
```

urbiscript
Session

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

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

urbiscript
Session

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.

urbiscript
Session

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

5.2 Methods

Methods in urbiscript are simply object slots containing functions. Using `obj.slot` if slot contains a function will return the function itself. Use `obj.slot()` to evaluate the function instead.

Inside a method, `this` gives access to the target — as in C++. It can be omitted if there is no ambiguity with local variables.

urbiscript
Session

```
var o = Object.clone();
[00000000] Object_0x00000001
// 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.getSlotValue("f");
[00000001] function () {
    echo("This is f with target ".+'(this));
    return 42;
}
// Huho, no parenthesis, no call.
o.f;
[00000001] function () {
    echo("This is f with target ".+'(this));
    return 42;
}
o.f();
[00000000] *** This is f with target Object_0x00000001
[00000000] 42
```

Let's use `getSlotValue` to introspect some functions:

urbiscript
Session

```
// The 'asList' method of strings returns the characters in a list.
"foo".asList();
[00003447] ["f", "o", "o"]
// Using getSlot, we can fetch the function without calling it.
"".getSlotValue("asList");
[00000000] function () { split("") }
```

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

urbiscript
Session

```
"foo".isBlank();
[00000000] false
"foo".getSlotValue("isBlank");
[00000000] Primitive_0x422f0908
```

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

5.3 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
```

urbiscript
Session

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

5.4 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"
```

urbiscript
Session

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
```

urbiscript
Session

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).

urbiscript
Session

```
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 a non-trivial fact: any modification on **a** will affect **c** as well, as shown below.

urbiscript
Session

```
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.

urbiscript
Session

```
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.

urbiscript
Session

```
function test(arg)
{
  var arg.flag; // add a slot in arg
  echo(arg.uid); // print its uid
}|;
var x = Object.clone();
[00000000] Object_0x00000001
```

```
x.uid;
[00000000] "0x00000001"
test(x);
[00000000] *** 0x00000001
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
```

urbiscript
Session

5.5 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 6

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++.

6.1 if

The `if` construct is the same has 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
```

urbiscript
Session

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

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

urbiscript
Session

6.2 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
```

urbiscript
Session

6.3 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.

urbiscript
Session

```
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.

urbiscript
Session

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

6.4 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`.

urbiscript
Session

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

6.5 do

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

urbiscript
Session

```
var o1 = Object.clone();
[00000000] Object_0x423a0708
var o1.one = 1;
[00000000] 1
var o1.two = 2;
[00000000] 2
echo(o1.uid);
[00000000] *** 0x423a0708
```

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.

urbiscript
Session

```
var o2 = Object.clone();
[00000000] Object_0x42339e08
// 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] *** 0x42339e08  
[00000000] Object_0x42339e08
```


Chapter 7

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.

7.1 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. The last semicolon inside a scope is optional.

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

urbiscript
Session

7.2 Advanced scoping

Scopes can be nested. Variables can be redefined in nested 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);  
};  
[00000000] *** 3  
x;  
[00000000] 3
```

urbiscript
Session

7.3 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.

urbiscript Session

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

7.4 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.

urbiscript Session

```
function printSalaries(var rate)
{
    var charges = 100;
    function computeSalary(var hours)
    {
        // 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);
[00000000] *** Alice's salary is 425
[00000000] *** Bob's salary is 350
```

Closures can also change captured variables, as shown below.

urbiscript Session

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

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

Chapter 8

Objective Programming, urbiscript Object Model

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

8.1 Prototype-Based Programming in urbiscript

You're probably already familiar with class-based object programming, since this is the C++, Java, C# model. Classes and objects are very different entities. Classes and 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 programming 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](#)). 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)
```

urbiscript
Session

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)
```

urbiscript
Session

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.

urbiscript
Session

```
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);
```

8.2 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.

urbiscript
Session

```
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.

urbiscript
Session

```
// 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.

urbiscript
Session

```
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`.

urbiscript
Session

```
// We create a fresh object.
var c = Object.clone();
[00000000] Object_0x00000001
```

```
// As expected, it has no 'slot' slot.
c.slot;
[00000000:error] !!! lookup failed: slot
var p = Object.clone();
[00000000] Object_0x00000002
var p.slot = 0;
[00000000] 0
c.addProto(p);
[00000000] Object_0x00000001
// Now, 'slot' is found in c, because it is inherited from p.
c.slot;
[00000000] 0
c.removeProto(p);
[00000000] Object_0x00000001
// Back to our good old lookup error.
c.slot;
[00000000:error] !!! lookup failed: slot
```

The slot lookup algorithm in urbiscript in 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
```

urbiscript
Session

You can check where in the prototype hierarchy a slot is found with the `locateSlot` method. This is a very handful 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
```

urbiscript
Session

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)`.

8.3 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.

urbiscript
Session

```
var proto = Object.clone();
[00000000] Object_0x00000001
var proto.x = 0;
[00000000] 0
var o1 = proto.clone();
[00000000] Object_0x00000002
var o2 = proto.clone();
[00000000] Object_0x00000003
// 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.

urbiscript
Session

```
// 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.

8.4 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++, i.e., 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 6.5](#). 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_0x00000001
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
```

urbiscript
Session

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 + ")"; }
};
```

urbiscript
Session

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

It is also possible to specify a proto for the newly created “class”, using the same syntax as Java and C++:

```
class Top
{
    var top = "top";
};
[00000000] Top

class Bottom : Top
{
    var bottom = "bottom";
};
[00000000] Bottom

Bottom.new().top;
[00000000] "top"
```

urbiscript
Session

For more details, see [Section 21.1.6.8](#).

8.5 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.

urbiscript Session

```
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.

urbiscript Session

```
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)
```

8.6 Operators

In urbiscript, operators such as `+`, `&&` and others, are regular functions 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.

urbiscript Session

```
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)
```

8.7 Properties

Sometimes one needs to attach attributes to a variable, and not the value it contains, for instance to store whether the variable is constant or not. In urbiscript we use objects named `Slots` that stands before the actual value, and persist when a new value is assigned to the variable. Variables of the `Slot` are called `properties`.

8.7.1 Features of Values

In following example, we attach some random slot `foo` to the value pointed to by the slot `x`.

```
var x = 123;
[00000000] 123
var x.foo = 42;
[00000000] 42
```

urbiscript
Session

If `y` is another slot to the value of `x`, then it provides the same `foo` feature:

```
var y = x;
[00000000] 123
y.foo;
[00000000] 42
// The value in the slots x and y are the same object
x==y;
[00000000] true
x.foo = 43 | y.foo;
[00000001] 43
```

urbiscript
Session

If `x` is bound to a new object (e.g., 456), then the feature `foo` is no longer present, since it's a feature of the *value* (i.e., 123), and not one of the slot (i.e., `x`).

```
x = 456;
[00000000] 456
x.foo;
[00000000:error] !!! lookup failed: foo
```

urbiscript
Session

Of course, `y`, which is still linked to the original value (123), answers to queries to `foo`.

```
y.foo;
[00000000] 43
```

urbiscript
Session

8.7.2 Features of Slots

If, on the contrary you want to attach a feature to the slot-as-a-name, rather than to the value it contains, use the *properties*. The syntax is `slotName->propertyName`.

```
x = 123;
[00000000] 123
x->foo = 42;
[00000000] 42
x->foo;
[00000000] 42
```

urbiscript
Session

Copying the value contained by a slot does *not* propagate the properties of the slot:

```
y = x;
[00000000] 123
y->foo;
[00000000:error] !!! property lookup failed: y->foo
```

urbiscript
Session

And if you assign a new value to a slot, the properties of the slot are preserved:

```
x = 456;
[00000000] 456
x->foo = 42;
[00000000] 42
```

urbiscript
Session

8.8 Getters and setters

All the properties with a special meaning are described in [Slot](#). Two of particular interest are `oget` and `oset`. They can be used to define a getter and a setter function, which are called each time the Slot is read or written to, respectively, thus implementing the `property` semantics as defined in JavaScript. urbiscript also borrowed the JavaScript shortcut syntax `get foo` and `set foo` to define setter and getter properties.

Consider this example:

```
urbiscript
Session
class Vector
{
    var x=0;
    var y=0;
    function init(x=0, y=0)
    {
        this.x = x; // Here we use copy on write to create our own slot x.
        var this.y = y; // Here we force local slot creation, both are valid.
    };
    // Return the L2 norm of the vector
    get norm() // sugar for norm->oget = function()
    {
        (x*x+y*y).sqrt()
    };
    // Scale the vector so that its norm becomes newNorm
    set norm(newNorm) // sugar for norm->oset = function(newNorm)
    {
        var oldNorm = norm;
        x *= newNorm/oldNorm;
        y *= newNorm/oldNorm;
    };
    function asString()
    {
        "<" + x + "," + y + ">"
    };
}
[00000000] <0,0>
var v = Vector.new(1, 1);
[00016149] <1,1>
v.norm;
[00018036] 1.41421
v.norm = 2.sqrt() | v;
[00177142] <1,1>
```

If you use a setter without a getter, the value returned by your function will be stored if it is not void, as demonstrated in [Slot.set](#).

8.8.1 When to use properties?

The c# documentation provides a very good explanation of when to use properties that we took the liberty to reproduce verbatim here:

In most cases, properties represent data, and methods perform actions. Properties are accessed like fields, which makes them easier to use. If a method takes no arguments and returns an object's state information, or accepts a single argument to set some part of an object's state, it is a good candidate for becoming a property.

Properties should behave as if they are fields; if the method cannot, it should not be changed to a property. Methods are preferable to properties in the following situations:

- The method performs a time-consuming operation. The method is perceivably slower than the time it takes to set or get a field's value.
- The method performs a conversion. Accessing a field does not return a converted version of the data it stores.

- The "Get" method has an observable side effect. Retrieving a field's value does not produce any side effects.
- The order of execution is important. Setting the value of a field does not rely on other operations having occurred.
- Calling the method twice in succession creates different results. The method is static but returns an object that can be changed by the caller. Retrieving a field's value does not allow the caller to change the data stored by the field.

Chapter 9

Organizing your code using package and import

In addition to the lookup mechanism described in the previous chapter, urbiscript provides a secondary lookup mechanism similar to the one found in most languages, through the `import` and `package` keywords.

9.1 Why object lookup is not enough

Consider the example below, where one would try to emulate a C++ namespace or Java package using only object lookup in urbiscript.

```
// Make a globally-accessible 'namespace' Shapes.
class Global.Shapes
{
    // Create an object representing a point
    class Point { var x=0; var y=0};

    // And one representing a color
    class Color { var r=0; var g=0; var b=0};

    // Now try a colored point using the objects below.
    class ColoredPoint
    {
        var point;
        var color;
        function init()
        {
            point = Point.new();
            color = Color.new();
        }
    };
}
```

urbiscript
Session

This looks fine, except that `Point` and `Color` are not visible from within `ColoredPoint.init`:

```
var cp = Global.Shapes.ColoredPoint.new();
[00000001:error] !!! lookup failed: Point
[01234567:error] !!!     called from: new
```

urbiscript
Session

One would need to make `ColoredPoint` *inherit* from `Shapes`, instead of simply having it as one of its variables, or make all classes in `Shapes` globally accessible. Those are not satisfying solutions.

9.2 Import and package

`package` works in the way one would expect:

```

package Shapes
{
    // Create an object representing a point
    class Point { var x=0; var y=0};
    // And one representing a color
    class Color { var r=0; var g=0; var b=0};
    // Now try a colored point using the objects below.
    class ColoredPoint
    {
        function init()
        {
            var this.point = Point.new();
            var this.color = Color.new();
        };
    };
    var p = ColoredPoint.new();
    p.print();
}
[00000001] ColoredPoint_0x00000000

```

Do not nest package declarations, use `package` and then `class` as in the example above.

`import` has two syntax: `import foo.bar` to make `foo.bar` visible as `bar` in the current scope, and `import foo.*` to make everything in the package `foo` visible in the local scope.

urbscript
Session

```

{
    import Shapes.*;
    Point;
    Color;
};

[00000001] Color
// It only applies to the scope.
Point;
[01234567:error] !!! lookup failed: Point
{
    // Import only Point from Shapes
    import Shapes.Point;
    Point;
    Color;
};

[01234567:error] !!! lookup failed: Color
// Packages have visibility over themselves as one might expect,
// like an implicit 'import this.*'.
{
    import Shapes.ColoredPoint;
    ColoredPoint.new();
};

[00000002] ColoredPoint_0x00000001

```

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.setSlotValue("m1", function () { echo("Hello") })|;
// This is strictly equivalent/
var o.m2 = function () { echo("Hello") }|;
o.m1();
[00000000] *** Hello
o.m2();
[00000000] *** Hello
```

urbiscript
Session

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

urbiscript
Session

It turns out that `all` already exists: instead of `all(list, predicate)`, use `list.all(predicate)`, see [RangeIterable.all](#).

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.

urbiscript
Session

```
// 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.

urbiscript
Session

```
// This ...
function obj.f /*...*/ {/*...*/};
// ... is actually a shorthand for:
const 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.

urbiscript
Session

```
// 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.

urbiscript
Session

```
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
```

urbiscript
Session

```
[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.

urbiscript
Session

```
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.

urbiscript
Session

```
function test()
{
    // Wait for one second, and echo "foo".
    detach({sleep(1s); echo("foo")});
}
test();
echo("Not blocked");
[00000000] Job<shell_15>
[00000000] *** Not blocked
sleep(2s);
echo("End of sleep");
[00001000] *** foo
[00002000] *** End of sleep
```

11.3 Tags for parallel control flows

A [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
```

urbiscript
Session

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
```

urbiscript
Session

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](#)):

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. (if you are familiar with exceptions, imagine that `tag.stop()` throws an exception in all tasks tagged by tag, that is caught by outermost occurrence of `tag:`).

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
```

urbiscript
Session

```
// Suspend execution
mytag.freeze();
// No printing anymore
sleep(1s);
// Resume execution
mytag.unfreeze();
sleep(1s);
[00007000] *** ping
```

urbiscript
Session

```
// 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();
```

urbiscript
Session

```
// 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 timeout, freezeif and stopif

`timeout`, `freezeif` and `stopif` are three keywords that take advantage of the primitive parallelism tools we've seen above to provide useful constructs.

11.4.1 timeout

```
var countdown = 3;
timeout(3.5s) every(1s) { echo(countdown); countdown --}; echo("stopped");
[00000001] *** 3
[00000001] *** 2
[00000001] *** 1
[00000001] *** 0
[00000001] *** stopped
```

urbiscript
Session

As you can see from the above example, timeout stops (in a similar fashion to `Tag.stop()`) the given code after given duration.

The timeout itself does not detach, so the timeout command lasts for the given duration, or until the command finishes, whichever comes first.

11.4.2 stopif and freezeif

`freezeif` and `stopif` use similar construct, and freeze/stop (respectively) the given code when the condition is true.

```
var b = false;
timeout(3.2s) detach({
    freezeif(b) every(500ms) echo("tick"),
    freezeif(!b) every(500ms) echo("tack")
})|;
sleep(1.2s); b = true;
[00000001] *** tick
[00000001] *** tick
[00000001] *** tick
[00000001] true
sleep(1s); b = false;
[00000001] *** tack
[00000001] *** tack
[00000001] *** tack
[00000001] false
sleep(1s); echo("done");
[00000001] *** tick
[00000001] *** tick
[00000001] *** done
```

urbiscript
Session

11.5 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.
```

urbiscript
Session

```
{  
    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 Watchdog constructs

The first construct we will study uses the `at` keyword. Given a condition and a statement, `at` will evaluate the statement each 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
```

urbiscript
Session

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
```

urbiscript
Session

See [Section 21.11.1](#) for more details on `at` statements.

12.2 Events

In addition to monitoring an expression with a watchdog, urbiscript enables you to define events that can be caught with the `at` construct we saw earlier. You can create events by instantiating the `Event` prototype. They can then be emitted with the `!` keyword.

12.2.1 Emitting Events

```
urbiscript
Session
var myEvent = Event.new();
[00000000] Event_Oxb5579008
at (myEvent?)
  echo("ping");
myEvent!;
[00000000] *** ping
// events work well with parallelism
myEvent! & myEvent!;
[00000000] *** ping
[00000000] *** ping
```

`myEvent!` triggers an instantaneous event, which has no effect on `at` registered after it:

```
urbiscript
Session
var ev = Event.new();
[00000000] Event_Oxb5579008
ev!;
at(ev?) echo("ping");
// Nothing happens unless ev is emitted again.
```

Events can be emitted for a given duration using the `~(duration)` construct after the `!`:

```
urbiscript
Session
var event = Event.new();
[00000000] Event_Oxb5579008
// emit event for 1 second in parallel with the scope
event!~(1s),
// Use an at 100ms later, the event is still in triggered state.
sleep(100ms) | at(event?) echo("ping");
sleep(1s);
[00000000] *** ping
```

Note in the example above that `at` only triggers once for the whole duration of the emit.

12.2.2 Emitting events with a payload

Events behave very much like “channels”: listeners use `at`, and producers use `!`. Fortunately, the messages can include a *payload*, i.e., something sent in the “message”. The Event then behaves very much like an identifier of the message type. To send/catch the payload, just pass arguments to `!` and `?`:

```
urbiscript
Session
var event = Event.new();
[00000000] Event_Ox00000001

at (event?(var payload))
  echo("received: " + payload)
onleave
  echo("had received: " + payload);

event!(1);
[00000008] *** received: 1
[00000009] *** had received: 1

event!([{"string": 124}]);
```

```
[00000010] *** received: ["string", 124]
[00000011] *** had received: ["string", 124]
```

Like functions, events have an *arity*, i.e., they depend on the number of arguments: `at (event?(arg))` will only match emissions whose payload contain exactly one argument, i.e., `event!(arg)`.

```
// Too many arguments.
event!(1, 2);

// Not enough arguments.
event!;
event!();
```

urbiscript
Session

Event handlers that do not specify their arity (i.e., without parentheses) match event emissions of any arity.

```
at (event?)
  echo("received an event")
onleave
  echo("had received an event");

event!;
[00000014] *** received an event
[00000015] *** had received an event

event!(1);
[00000016] *** received: 1
[00000017] *** had received: 1
[00000018] *** received an event
[00000019] *** had received an event

event!(1, 2);
[00000020] *** received an event
[00000021] *** had received an event
```

urbiscript
Session

Actually, the feature is much more powerful than this: full pattern matching applies, as with the `switch/case` construct.

```
var e = Event.new()|;

at (e?)
  echo("e");

at (e?(var x))
  echo("e(x)");

at (e?(1))
  echo("e(1)");

at (e?(var x) if x.isA(Float) && x % 2)
  echo("e(odd)");

// Payload must be a list of three members, the first two being 1 and 2, and
// the third one being greater than 2, when converted as a Float.
at (e?([1, 2, var x]) if 2 < x.asFloat())
  echo("e([1, 2, x = %s])" % x);

e!;
[00000845] *** e

e!(0);
[00011902] *** e
[00011902] *** e(x)

e!(1);
```

urbiscript
Session

```
[00023327] *** e
[00023327] *** e(x)
[00023327] *** e(1)
[00023327] *** e(odd)

e!([1, 2, 1]);
[00024327] *** e
[00024327] *** e(x)

e!([1, 2, 3]);
[00025327] *** e
[00025327] *** e(x)
[00025327] *** e([1, 2, x = 3])

e!([1, 2, "4"]);
[00026327] *** e
[00026327] *** e(x)
[00026327] *** e([1, 2, x = 4])
```

Remember [stopif11.4.2](#) and [freezeif11.4.2](#) from the previous section? They also accept an event in their condition:

urbiscript
Session

```
var e = Event.new();
var counter = 3;
stopif(e?(0)) while(true) { echo(counter); e!(counter); counter--;};
[00026327] *** 3
[00026327] *** 2
[00026327] *** 1
[00026327] *** 0
```

12.2.3 waituntil

Sometimes it is useful to wait until a condition becomes true, or an event triggers. This task can be accomplished with the [waituntil21.11.2](#) construct:

urbiscript
Session

```
counter = 0;
waituntil(counter == 2) | echo("Now!"), // in parallel
for (var i: 4) { echo(i); counter = i};
[00000001] *** 0
[00000001] *** 1
[00000001] *** 2
[00000001] *** Now!
[00000001] *** 3
```

Chapter 13

Urbi for ROS Users

This chapter extends the [ROS official tutorials](#)¹. Be sure to complete this tutorial before reading this document.

13.1 Communication on topics

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

```
$ rosmake beginner_tutorial
```

Shell
Session

13.1.1 Starting a process from Urbi

To communicate with ROS components, you need to launch them. You can do it by hand, or ask Urbi to do it for you. To launch new processes through Urbi, we will use the class [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", []);
[00000001] Process roscore
var talker = Process.new("rosrun", ["beginner_tutorial", "talker"]);
[00000002] Process rosrun
core.run;
talker.run;
```

urbiscript
Session

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](#)).

13.1.2 Listening to Topics

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

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

urbiscript
Session

Then we can get the list of launched nodes:

```
Ros.nodes;
```

urbiscript
Session

This returns a [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:

```
urbiscript
Session
// Get the structure.
// ";" is an idiom to discard the display of the return value.
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':

```
urbiscript
Session
// Initialize the subscription object.
var chatter = Ros.Topic.new("/chatter")|;
// Subscribe.
chatter.subscribe;
// This is the way we are called on new message.
var chatTag = Tag.new|;
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:

```
urbiscript
Session
[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:

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

To stop temporarily the `Global.echo`, we take advantages of tags (Section 11.3), 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:

```
urbiscript
Session
// 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 an convenient alias for `Ros.Topic.publish`.

¹<http://www.ros.org/wiki/ROS/Tutorials>

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("rosrun", ["turtlesim", "turtlesim_node"]);|
turtle.run;
```

urbiscript
Session

`Ros.topics` shows 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]
```

urbiscript
Session

Now we want to have it moving in circle with a small sinusoid wave. This goes in two step. First, we set up the infrastructure so that changes in Urbi are seamlessly published in ROS.

```
// 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.
at (angular->changed?)
{
    m["angular"] = angular;
    velocity << m
};
```

urbiscript
Session

In the future Urbi will provide helping functions to spare the user from the need to perform this “binding”. But once this binding done, all the features of urbiscript can be used transparently.

For instance we can assign a sinusoidal trajectory to 'angular', which results in the screen-shot on the right-hand side.

```
// For 20 seconds, bind "angular" to a sine.
timeout (20s)
angular = 0.3 sin: 2s ampli: 2;
```



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 command is stopped.

Alternatively, `Tags` could have been used to get more control over the trajectory:

```
// A Tag to control the following endless statement.
var angTag = Tag.new|;

angTag:
    // Bind "angular" to a trajectory.
    // Put in background thanks to ",", since this statement is never ending.
    angular = 0.3 sin: 2s ampli: 2,
    // Leave 20 seconds to the turtle...
```

urbiscript
Session

```
sleep(20s);

// before freezing it.
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](#) for more information. 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 available services, and filter out loggers:

```
urbiscript
Session
var logger = Regexp.new("(get|set)_logger") |;
var services = Ros.services.keys |;
for (var s in services)
  if (s not in logger)
    echo(s);
[00000001] *** "/clear"
[00000001] *** "/kill"
[00000001] *** "/turtle1/teleport_absolute"
[00000001] *** "/turtle1/teleport_relative"
[00000001] *** "/turtle1/set_pen"
[00000001] *** "/reset"
[00000001] *** "/spawn"
```

The `closure` construct allows us to keep access to the local variables, here `logger`.

Now there is a service called '`/spawn`'; to initialize it:

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

The `new` function takes the service name as first argument, and as second argument whether the connection should be kept alive.

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 with `spawn.reqStruct`, and the structure of the response with `spawn.resStruct`.

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

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

13.3 Image Publisher from ROS to Urbi

This section will use topics manipulation with advertising and subscription. Be sure to understand these topics before doing this tutorial.

Requirements You have to finish the image Publisher/Subscriber tutorial (http://www.ros.org/wiki/image_transport/Tutorials) before doing this tutorial.

First, we will make a ROS Publisher and subscribe to it with Urbi. Make sure that Publisher 'learning_image_transport' package is compiled:

```
$ rosmake learning_image_transport
```

We will also run `urbi` with a network connection opened (e.g., on port 54000) to allow `urbi-image` (Section 20.4) to connect to it.

```
$ urbi --host=127.0.0.1 --port=54000 -- -f
```

Shell Session

Also, you have to run `roscore` to communicate with ROS.

```
var core = Process.new("roscore", []);
[00000001] Process roscore
core.run;
```

urbiscript Session

Run the Publisher The Publisher is a process that will send a image and wait for a Subscriber to get it.

```
// In this example the image is in the current directory.
var publisher =
  Process.new("rosrun",
    ["learning_image_transport", "my_publisher", "test.jpg"]);
[00000002] Process rosrun
publisher.run;
```

urbiscript Session

Using a camera to display By default, `urbi-image` displays the images that are available via the `camera` device (see Section 20.4). To simplify the setup, let's define a pseudo `camera` which will store the data received:

```
class Global.camera: Loadable
{
  // A variable to store image data.
  UVar.new(this, "val");
  val = 0;
};
```

urbiscript Session

Subscribe to the topic Now, our Publisher is running and we have a camera waiting for data. All we need to do is connecting to the Publisher with a topic, the Subscriber.

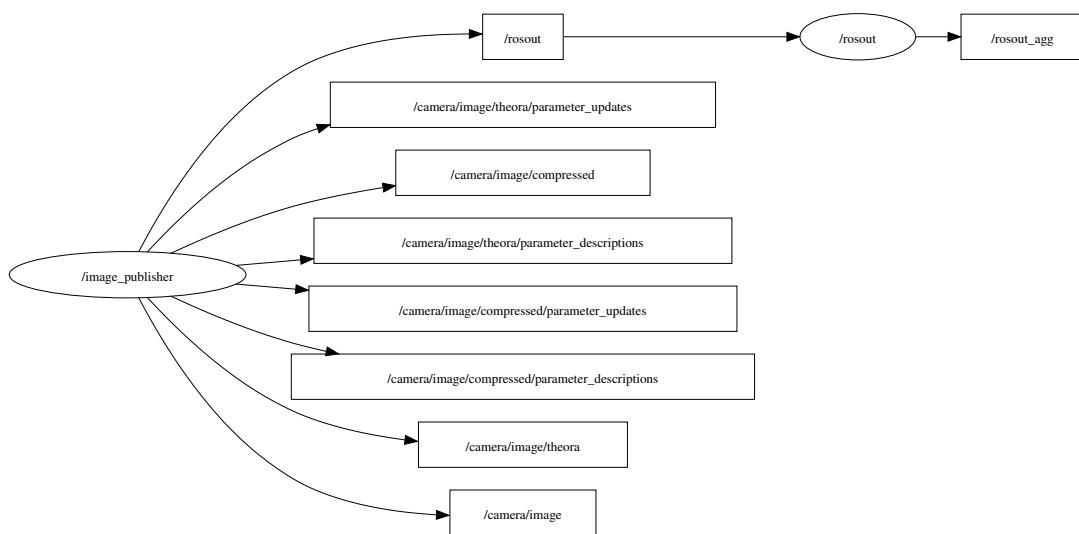


Figure 13.1: Output from `rxgraph`

Have a look at the different topics created by the Publisher, for instance by running `rxgraph`, which generates the graph in Figure 13.1. As you can see, seven topics are available for the

camera. We will use the ‘/camera/image/compressed’ topic for this example. For further information about the image format in ROS see http://www.ros.org/doc/api/sensor_msgs/html/msg/CompressedImage.html.

```
urbiscript
Session
var cameraTopic = Ros.Topic.new("/camera/image/compressed")|;
at (cameraTopic.onMessage?(var imgMsg))
{
    // Converting the ROS image to Urbi format.
    imgMsg["data"].keywords = imgMsg["format"]|
    // We can now store the data into camera.
    if (!camera.val)
        echo("Image well received. Store the image into the camera") |
        camera.val = imgMsg["data"];
    },
    // Waiting for the "publisher" Process to be set up.
    sleep(2s);
    cameraTopic.subscribe;
```

We are now connected and ready to display.

```
urbiscript
Session
[00000003] *** Image well received. Store the image into the camera
```

In a new terminal run `urbi-image`:

```
Shell
Session
$ urbi-image
Monitor created window 62914561
***Frame rate: 5.000000 fps***
```

You have now your image displayed in a window.

13.4 Image Subscriber from Urbi to ROS

Now, we want to send images to ROS using a Urbi Publisher. Make sure `roscore` is running and ‘`learning_image_transport`’ package is compiled.

Run the Subscriber The basic Subscriber in the ‘`learning_image_transport`’ package is expecting a ‘/camera/image’ topic. To avoid modifying the Subscriber code in ROS, we will simply ask to the Subscriber topic to accept ‘/camera/image/compressed’ topics.

```
urbiscript
Session
var subscriber =
    Process.new("rosrun",
                ["learning_image_transport", "my_subscriber",
                 "_image_transport:=compressed"]);
[00037651] Process rosrun
subscriber.run;
```

Publishing images with Urbi The ‘`sensor_msgs/CompressedImage`’ message format provides a structure that requires a few changes.

```
urbiscript
Session
// File.new("...").content returns a Binary.
var urbiImage = File.new("test.jpg").content|;
urbiImage.keywords = "jpeg"|

var publisher = Ros.Topic.new("/camera/image/compressed")|;
// Advertising the type of message used.
publisher.advertise("sensor_msgs/CompressedImage");

var rosImg = publisher.structure.new|;
// The rosImg is a dictionary containing a Binary and a String.
rosImg["data"] = urbiImage|;
rosImg["format"] = "jpeg"|;
```

This message contains more fields but you need only these two to send an image.
Now, you just have to publish the image.

```
// Publishing at regular intervals.  
every (500ms)  
{  
    publisher << img;  
},
```

urbscript
Session

Communication is done, the image should be displayed.

13.5 Remote communication

We have worked with a `roscore` running on the machine as the ROS processes but the purpose of using ROS with Urbi is to communicate with a remote machine. All you need is to setup your network configuration to avoid unexpected behaviors (see [NetworkSetup²](#)).

Make sure the ROS environment variables are well set, especially `ROS_URI`, `ROS_HOSTNAME`, `ROS_IP`.

See [Tutorials/MultipleMachines³](#) for additional information.

Try our tutorials remotely to check if the connection is set correctly.

To go further... Please see the Urbi/ROS Reference Manual, [Chapter 23](#).

²<http://www.ros.org/wiki/ROS/NetworkSetup>

³<http://www.ros.org/wiki/ROS/Tutorials/MultipleMachines>

Part II

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 binary packages.

[Chapter 15 — Frequently Asked Questions](#)

Some answers to common questions.

[Chapter 16 — Urbi Guideline](#)

Based on our own experience, and code that users have submitted to us, we suggest a programming guideline for Urbi SDK.

[Chapter 17 — 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 19 — Building Urbi SDK](#)

Building Urbi SDK from the sources. How to install it, how to check it and so forth. This chapter is meant only for people who want to *build* Urbi SDK, which is orders of magnitude more delicate than simply installing the binary packages from Gostai.

Chapter 14

Installation

All instructions to install and compile this version of Urbi can be found on the github repository:

<https://github.com/urbiforge/urbi>.

Alternatively, you may want to run the following Urbi docker container:

<https://hub.docker.com/r/urbiforge/urbi>

Cygwin Issues

Inputs and outputs of windows native applications 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 ‘+=’

Although we tried to avoid it, there might still be shell scripts where we use ‘+=’, 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 19.1](#).

15.1.4 configure: error: The Java VM java failed

If you experience the following failure:

```
checking if java works...
configure: error: The Java VM java failed
          (see config.log, check the CLASSPATH?)
```

and if you looked at ‘`config.log`’, you should find something like:

```

Exception in thread "main" java.lang.NoClassDefFoundError: Test
Caused by: java.lang.ClassNotFoundException: Test
    at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
Could not find the main class: Test. Program will exit.

```

You might be trying to compile Urbi SDK from a directory with non-ASCII characters in its full name (for instance ‘/home/jessy/Téléchargements/Sources’). In that case the JVM fails to decode properly the path, and `configure` fails with the above message.

Move your sources elsewhere, with only plain ASCII characters, such as ‘/home/jessy/Sources’.

15.1.5 ‘make check’ fails

Be sure to read [Section 19.3](#). In particular, run ‘`make check`’ several times (see [Section 19.3](#) to know why). If the failures remain, please submit the ‘`test-suite.log`’ file(s) (see [Section 15.5.3](#)).

15.1.6 check: error: Unable to load native library: libjava.jnilib

If you experience the following failure on Mac OS X:

```

$ urbi-launch-java --port-file server.port tests/all/All.jar
Error occurred during initialization of VM
Unable to load native library: libjava.jnilib

```

then you have something in the `DYLD_LIBRARY_PATH` that annoys the Java VM. Although we have not pinpointed it exactly, it seems to be a problem in our `libjpeg`: first run ‘`make -C sdk-remote/jpeg clean`’ then ‘`make -S sdk-remote/jpeg`’.

15.2 Troubleshooting

15.2.1 error while loading shared libraries: libport.so

If on GNU/Linux you get an error such as:

Shell Session	<pre> urbi-sdk/2.7.1 \$./bin/urbi ./bin/urbi: error while loading shared libraries: libport.so: \ cannot open shared object file: No such file or directory </pre>
---------------	---

then check that ‘/proc’ is properly mounted. To make Urbi SDK relocatable, executables and libraries use a relative path to their peers. To resolve these paths into absolute paths, the loader needs to know where the program is located, a feature provided by ‘/proc’. If for instance you run Urbi SDK in a chrooted environment, then it is possible that you forgot to mount ‘/proc’. The traditional `ps` utility also needs ‘/proc’ to be mounted, so running it would also help checking if the setup is complete.

15.2.2 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 uses 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.

15.2.3 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.

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

Same answer as [Section 15.2.3](#).

15.2.5 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 20.3](#).

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);
```

urbiscript
Session

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);
```

urbiscript
Session

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}).

15.2.6 ‘myuobject: file not found’. What can I do?

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

15.2.6.1 Getting a better diagnostic

First, set the `GD_LEVEL` environment variable (see [Section 20.1.2](#)) 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.

15.2.6.2 GNU/Linux

A libltdl 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 ‘dlsym’ 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).

Shell Session

```
$ 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.

Shell Session

```
# 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.

Shell Session

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

15.2.6.3 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.

Shell Session

```
$ 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.

Shell Session

```
# 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.

Shell Session

```
$ 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
$
```

15.2.6.4 Windows

If you are running Cygwin, then have a look at the following section, which uses some of its tools.

A specific constraint, for which currently we do not have nice solutions, is that when Windows loads a DLL, it looks for all its dependencies (i.e., other DLL that are needed) in the directory from which the program was run, or in the PATH. There is no way, that we are aware of, to embed in a DLL the information about where the dependencies are. When trying to load a DLL with missing dependencies, say ‘foo.dll’, the error message will be something like “can’t open the module”, and worse yet, if you read the detailed log messages (by setting GD_LEVEL to DUMP for instance) it will report “failed with error 126: The specified module could not be found” although the file *is* there.

So first try to understand what are the missing dependencies. Under Windows, use DependencyWalker (see <http://dependencywalker.com>) to check that a given DLL finds all its dependencies. If some dependencies are not found either:

- change your PATH so that it goes via the directories that contain the dependencies of your DLLs;
- or copy these dependencies in the ‘bin/’ directory of Urbi SDK, since that’s the directory of the program, ‘urbi-launch’.

The first approach is more tractable. Beware that dependencies may also have dependencies...

Cygwin Use the `cygcheck.exe` program to check dependencies. Beware that you must provide a qualified path to the file. Chances are that if

Shell Session

```
$ cygcheck foo.dll
```

does not work and will pretend that ‘`foo.dll`’ does not exist (although it’s *right there*), then this will work:

Shell Session

```
$ cygcheck ./foo.dll
```

In this output, look for lines like these:

Shell Session

```
cygcheck: track_down: could not find OgreMain.dll
cygcheck: track_down: could not find OIS.dll
cygcheck: track_down: could not find libuobject-vc90.dll
```

and make sure that ‘`OgreMain.dll`’, ‘`OIS.dll`’ and so forth are visible in the PATH (don’t be worry about ‘`libuobject-vc90.dll`’, `urbi-launch` will take care of it). Note that when they are finally visible from the PATH, then you can run

Shell Session

```
$ cygcheck OgreMain.dll
```

without having to specify the path.

15.3 urbscript

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:

urbscript Session

```
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 17.2](#)). 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 5.1](#). Users who want to clear an object can empty it — see [Section 17.2](#).

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:

```
urbiscript
Session
var mind = 42;
[00000002] 42
function get()
{
    echo(mind);
}|;
get();
[00000003] *** 42
function Object.get()
{
    echo(mind)
}|;
// Where is my mind?
Object.get();
[00000004:error] !!! lookup failed: mind
[00000004:error] !!!      called from: get
```

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.

```
urbiscript
Session
// 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.

```
urbiscript
Session
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`.

{

urbiscript
Session

```
// 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
```

urbiscript
Session

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 an unique `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;
```

urbiscript
Session

```
[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
```

urbiscript
Session

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
[00000000:error] !!!      called from: getHis
```

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:

urbiscript Session

```
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 always accessible no matter where you are. So creating a genuine global variable is as simple as storing it in `Global`:

urbiscript Session

```
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:

urbiscript Session

```
// Store all constants here
package lang.Constants
{
    var Pi = 3.14;
    var Euler = 2.17;
    var One = 1;
    // ...
}

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

urbiscript Session

```
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:

urbiscript Session

```
{
  // 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 `Object.updateSlot` method.

urbiscript
Session

```
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 `=`.

urbiscript
Session

```
// 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:

urbiscript
Session

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

15.3.2.6 How do I call a constructor of a parent class?

The `init` function of parent classes is not called automatically when calling `new` on a child class. It is the responsibility of the child to invoke `init` itself, using `Code.apply`:

urbiscript
Session

```

package MyModule
{
    class Parent
    {
        function init(x)
        {
            var this.x = x;
        };
   };

    class Child: Parent
    {
        function init(x, y)
        {
            Parent.init.apply([this, x]);
            var this.y = y;
        }
   };
};

import MyModule;
var c = MyModule.Child.new(1, 2);
[00028287] Child_0xb48e0e48
c.x;
[00029646] 1

```

15.3.2.7 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]

```

urbiscript
Session

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

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

15.3.2.9 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) };

```

urbiscript
Session

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

```

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

```

urbiscript
Session

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) { /* ... */ };
}

```

urbiscript
Session

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

```
urbiscript
Session
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](#).

15.3.3.1 How do I create a tag?

See [Section 22.70.4](#).

15.3.3.2 How do I stop a tag?

Use the `stop` method (see [Section 22.70.1.1](#)).

```
urbiscript
Session
myTag.stop();
```

15.3.3.3 Can tagged statements return a value?

Yes, by giving it as a parameter to `stop`. See [Section 22.70.1.1](#).

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](#).

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.

```
urbiscript
Session
var ev = Event.new();
```

See [Section 22.15.5](#).

15.3.4.2 How do I emit an event?

Use the `!` operator.

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

urbiscript
Session

15.3.4.3 How do I catch an event?

Use the `at(event?args)` construct (see [Section 21.11.1](#)).

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

urbiscript
Session

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 21.7.6), or the `List.each` method:

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

urbiscript
Session

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 another 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 UObject 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;
}
```

C++

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

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

C++

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";
```

urbiscript
Session

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:

```
C++
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:

```
C++
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 31](#).

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 <https://github.com/urbiforge/urbi/urbi/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 32.10](#). 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 32.3](#)), or under the modified BSD license (see [Section 32.2](#)).

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, specify the Urbi version you are using (run ‘`urbi --version`’) and whether this bug is blocking you or not. Please keep kernel-bugs@lists.gostai.com in copy of all your correspondence: 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 19.3](#).

Chapter 16

Urbi Guideline

16.1 urbiscript Programming Guideline

16.1.1 Prefer Expressions to Statements

Code like this:

```
if (delta)
    cmd = "go";
else if (alpha)
    cmd = "turn"
else
    cmd = "stop";
```

urbiscript
Session

is more legible as follows:

```
cmd =
{
    if      (delta) "go"
    else if (alpha) "turn"
    else          "stop"
};
```

urbiscript
Session

16.1.2 Avoid `return`

The `return` statement (Section 21.3.3) is actually costly, because it is also in charge of stopping all the event-handlers, detached code, and code sent into background (via ‘,’ or ‘&’).

In a large number of cases, `return` is actually useless. For instance instead of:

```
function inBounds1(var x, var low, var high)
{
    if (x < low)
        return false;
    if (high < x)
        return false;
    return true;
}|;
assert
{
    inBounds1(1, 0, 2);    inBounds1(0, 0, 2);  inBounds1(2, 0, 2);
    !inBounds1(0, 1, 2);  !inBounds1(3, 0, 2);
};
```

urbiscript
Session

write

```
function inBounds2(var x, var low, var high)
{
    if (x < low)
        false
```

urbiscript
Session

```
    else if (high < x)
        false
    else
        true;
}|;
assert
{
    inBounds2(1, 0, 2);    inBounds2(0, 0, 2);  inBounds2(2, 0, 2);
    !inBounds2(0, 1, 2);  !inBounds2(3, 0, 2);
};
```

or better yet, simply evaluate the Boolean expression. As a matter of fact, returning a Boolean is often the sign that you ought to “return” a Boolean expression, rather than evaluating that Boolean value using a conditional, and return either true or false.

```
function inBounds3(var x, var low, var high)
{
    low <= x <= high
}|;
assert
{
    inBounds3(1, 0, 2);    inBounds3(0, 0, 2);  inBounds3(2, 0, 2);
    !inBounds3(0, 1, 2);  !inBounds3(3, 0, 2);
};
```

urbiscript
Session

Chapter 17

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](#) object contains functions that support some urbiscript 1 features.

17.1 \$(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](#).

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>

17.2 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 `Object.removeLocalSlot` 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;
// ...
```

urbiscript
Session

```
delete this.a;
// ...
cout << this.a;
```

For these reasons, and others, `delete` Foo was removed. To remove the *name* Foo, run `removeLocalSlot("Foo")` (Section 5.1) — 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:

urbiscript
Session

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

17.3 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>

17.4 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 17.1 for means of getting rid of them.

17.5 foreach

The same feature with a slightly different syntax is now provided by `for`. See Section 21.7.6.

17.6 group

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

urbiscript
Session

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

write

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

urbiscript
Session

17.7 loopn

The same feature and syntax is now provided by [for](#). See [Section 21.7.7](#).

17.8 new Foo

See [Section 8.5](#) 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)`

17.9 self

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

17.10 stop Foo

Use `Foo.stop` instead, see [Tag](#).

17.11 # line

Use `//#line` instead, see [Section 21.1.3](#).

17.12 tag+end

To detect the end of a statement, instead of

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

use the `leave?` method of the [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"
```

urbiscript
Session

urbiscript
Session

Chapter 18

Migration from urbiscript 2 to urbiscript 3

This chapter describes the changes needed to make urbiscript code written for urbiscript 2 work in the urbiscript 3 interpreter.

18.1 Automatic function evaluation without parenthesis

The most visible change from urbiscript 2 to urbiscript 3 is that functions are no longer called when parenthesis are omitted. The function itself is returned instead.

So one must add missing parenthesis around all function calls, or convert functions with no arguments to properties (8.8) where appropriate.

To help you track function calls missing parenthesis, you can define the environment variable `URBI_REPORT_MISSING_PAREN`.

```
function f() {1};  
[00004397] function () { 1 }  
f; // In urbiscript 2 this would have called f  
[ Urbi.Compatibility ] Maybe missing parens at 2.1  
[00005092] function () { 1 }
```

urbiscript
Session

This will produce false positives in the urbiscript standard library that you can safely ignore, and false positives in urbiscript 3 code.

18.2 getSlot and setSlot

Since `Slots` are now exposed in urbiscript, the urbiscript 2 `Slot.getSlot` and `Slot.setSlot` have been renamed to `Slot.getValue` and `Slot.setValue`.

18.3 Packages

Since packages (9) were introduced, some functions are no longer directly accessible without the proper import or package prefix. One notable instance is the content of `System`:

```
hostName;  
[02623134:error] !!! 29.1-8: lookup failed: hostName  
System.hostName;  
[02627733] "redlance"  
import System.*;  
hostName;  
[02635245] "redlance"
```

urbiscript
Session

Chapter 19

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](#).

A foreword that applies to any package, not just Urbi SDK: **building or checking as root is a bad idea.** Build as a regular user, and run ‘`sudo make install`’ just for the install time *if you need privileges to install to the chosen destination.*

19.1 Building

Urbi SDK uses qibuild as its buildsystem.

19.1.1 Getting and installing qibuild

The simplest and recommended way if you have python and pip is:

```
$ pip install qibuild
```

Shell
Session

Otherwise you can build it from sources:

```
$ git clone https://github.com/aldebaran/qibuild
$ cd qibuild
$ python setup.py install # without pip
```

Shell
Session

19.1.2 Setting up an urbi worktree

```
$ mkdir work
$ cd work
$ qisrc init
$ qisrc add https://github.com/aldebaran/libport.git
$ qisrc add https://github.com/aldebaran/urbi.git
$ qisrc add https://github.com/aldebaran/libjpeg.git
```

Shell
Session

19.1.3 Building

```
$ qibuild configure urbi
$ qibuild make urbi
```

Shell
Session

Add `--release` to make a release build.

19.1.4 Modifying the grammar or the AST

The build procedure above will use pregenerated files for the grammar and the AST files.

If you wish to modify ugrammar.y, utoken.l, or ast.yml, you must enable grammar/ast processing by passing `-DHAVE_CUSTOM_BISON=On`. This currently requires custom patches into bison that you can find in urbi-sdk 2.7.5 source tarball.

19.1.5 Installing

```
$ qibuild install urbi <INSTALL\_DIR> # add --release for release build
```

Shell Session

19.2 Run

You can now run the urbi-launch binary, either from the build directory or the install directory. See [Section 20.5](#) for informations on its command line.

In addition to the “public” environment variables ([Section 20.1.2](#)), some other, reserved for developers, alter the behavior of the programs.

URBI_ACCEPT_BINARY_MISMATCH As a safety net, Urbi checks that loaded modules were compiled with exactly the same version of Urbi SDK. Define this variable to skip this check, at your own risks.

URBI_CHECK_MODE Skip lines in input that look like Urbi output. A way to accept test files (*.chk) as input.

URBI_DESUGAR Display the desugared ASTs instead of the original one.

URBI_DOC Where to find the filedoc directory, which contains ‘THANKS.txt’ and so forth.

URBI_IGNORE_URBI_U Ignore failures (such as differences between kernel revision and ‘urbi.u’ revision) during the initialization.

URBI_INTERACTIVE Force the interactive mode, as if ‘`--interactive`’ was passed.

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. Obsolete, use the `Urbi.Parser` category and `GD_CATEGORY` instead ([Section 20.1.2](#)).

URBI_REPORT Display statistics 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. Obsolete, use the `Urbi.Scanner` category and `GD_CATEGORY` instead ([Section 20.1.2](#)).

URBI_SHARE Where to find the fileshare directory, which contains ‘images/gostai-logo’, ‘urbi/urbi.u’ and so forth.

URBI_SHARE Where to find the fileshare directory, which contains ‘images/gostai-logo’, ‘urbi/urbi.u’ and so forth.

URBI_TEXT_MODE Forbid binary communications with UObjects.

URBI_TOPOLEVEL Force the display the result of the top-level evaluation into the lobby.

19.3 Check

Root Running the test-suite as a super-user (root) is a bad idea ([Chapter 19](#)): some tests check that Urbi SDK respects file permissions, which of course cannot work if you are omnipotent.

Parallel Tests There are several test suites that will be run if you run ‘`qibuild test`’ (‘`-j4`’ works on most machines).

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 ‘`qibuild test`’ will give an opportunity for the test to pass (remind that the tests that passed will not be run again). Also, using ‘`qibuild test -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 ‘`qibuild test -j1`’ invocations.

19.4 Debug

Urbi can be debugged with `gdb`. It is highly recommended to use ‘`--enable-compilation-mode=debug`’ to configure if you intend to debug (see [Chapter 19](#)).

When compiled in debug mode, `gdb` extensions would be installed in the library directory and inside the share directory. These extensions are also available if you run `gdb` inside the root of the source directory.

These extensions provide pretty printing of some C++ objects, such as AST nodes, Urbi objects, intrusive pointers. Additional commands are provided for printing an equivalent of the urbiscript backtrace and for adding and removing breakpoints set on urbiscript.

To benefit from these extensions, install a recent `gdb` (higher than 7.2). Older versions may not be able to load the extensions, or may not provide the features on which the extensions rely. To avoid disturbance in case the extension cause much trouble than benefits, `gdb` can be run with extra arguments to disable these extensions.

```
$ gdb -xe 'set auto-load-scripts no' urbi
```

Shell Session

The following sections assume that `gdb` is recent enough.

19.4.1 GDB commands

urbi stack Print the backtrace of the current coroutine. The backtrace is indexed by the C++ frame numbers and includes frame which are manipulating the sources. Each primitive object is pretty printed inside this backtrace.

```
(gdb) b urbi::object::system_backtrace
Breakpoint 1 at 0x7ffff4be229d: file system.cc, line 305.
(gdb) c
Continuing.

//#push 1 "input.u"
function foo (x) { backtrace }|;
for(var i: [1]) foo([i.asString => i]);


Breakpoint 1, urbi::object::system_backtrace () at system.cc:305
305          runner::Job& r = runner();
(gdb) urbi stack
#12 [input.u:1.20-29] Lobby_0x7ffff7f03208.backtrace()
#16 [input.u:1.20-29] Call backtrace
#19 [input.u:1.20-29] Scope
#22 [input.u:2.17-39] Lobby_0x7ffff7f03208.foo(["1" => 1])
#26 [input.u:2.17-39] Call foo
```

```
#29 [input.u:2.17-39] Scope
#32 [flower/flower.cc:132.19-25] Scope
#35 [??] Code_0x7ffff7fbdfb8.each(1)
#48 [input.u:2.12-39] [1].each(Code_0x7ffff7fbdfb8)
#52 [input.u:2.12-39] Call each
#55 [input.u:2.1-40] Stmt
```

urbi break Set a breakpoint on an urbiscript function. A breakpoint is defined with at least a message and additional optional information such as a file, a line, a target and arguments can be used as condition for the breakpoint. Be aware that the target and the arguments are lexically compared with the pretty-printed versions.

```
(gdb) urbi break input.u:2 Lobby_0x7ffff7f03208.foo(["2" => 2])
UBreakpoint 1:
    Location: input.u:2
    Lobby_0x7ffff7f03208.foo(["2" => 2])
(gdb) c
Continuing.

//#push 1 "input.u"
function foo (x) { backtrace }|;
for(var i: [1, 2, 3]) foo([i.asString => i]);

[00219506:backtrace] foo (input.u:2.23-44)
[00219506:backtrace] each (input.u:2.12-44)
UBreakpoint 1: [input.u:2.23-45] Lobby_0x7ffff7f03208.foo(["2" => 2])
Inside Job 0x6a2f70
(gdb)
```

urbi delete Remove a breakpoint set by `urbi-break`. It expects one argument: the Urbi breakpoint number.

```
(gdb) urbi break foo
UBreakpoint 2:
<any>.foo(<any>)
(gdb) urbi delete 2
UBreakpoint 2:
Removed
```

urbi call Evaluate its arguments as an urbiscript call to eval. Live objects can be manipulated easily by using the object name followed by its address.

```
(gdb) urbi call Lobby_0x7ffff7f03208.echo(42)
[00219506] *** 42
```

urbi print Do the same as `urbi call` except that the result of the expression is printed at the end of the evaluation. Objects printed like that are likely to be destroyed at the end of the command unless the value has been stored in a global variable.

```
(gdb) urbi print { var a = "1"; var b = [1]; [a => b] }
["1" => [1]]
```

urbi continue Alias to `continue`.

urbi finish Execute until the end of the current function.

urbi next Execute until next Urbi function call.

urbi next job Execute until a function call is executed in another job.

urbi step Execute next Urbi function call until it return to the current function scope or leave the current scope.

```
(gdb) urbi break foo
UBreakpoint 1:
<any>.foo(<any>)
(gdb) urbi continue

//#push 1 "input.u"
function foo () {
    [1].map(closure (x) { x }); // yield
    [2].map(closure (x) { x }); // yield
    [3].map(closure (x) { x })
}
/* first */ foo & foo /* second */;

UBreakpoint 1: [input.u:6.1-4] Lobby_0xfffff35137a8.foo()
Inside Job 0x6a2f70
(gdb) # Go deeper inside foo
(gdb) urbi next
[input.u:2.3-29] [1].map(Code_0x7ffff3659338)
(gdb) urbi next
[urbi/list.u:129.18-130.18] [1].each|(Code_0x7ffff3658960)
(gdb) # The current coroutine will hold on the next function call after map
(gdb) urbi finish
UBreakpoint 1: [input.u:6.7-10] Lobby_0xfffff35137a8.foo()
Inside Job 0x9fab50
(gdb) # The breakpoint has hit on the second job calling foo.
(gdb) # Ignore this job, with "urbi continue"
(gdb) urbi continue
[input.u:3.3-29] [2].map(Code_0x7ffff365ca10)
Inside Job 0x6a2f70
(gdb) # Back into the first job where the finish has ended.
(gdb) # Step over the map call.
(gdb) urbi step
[input.u:4.3-29] [3].map(Code_0x7ffff36600e8)
(gdb) # Switch to the next job to be executed.
(gdb) urbi next job
[input.u:4.3-29] [3].map(Code_0x7ffff365d6b8)
Inside Job 0x9fab50
(gdb)
```


Part III

Urbi SDK Reference Manual

About This Part

This part defines the specifications of the urbiscript language. 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. For an urbiscript Tutorial, see [Part I](#).

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

[Chapter 20 — 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 21 — urbiscript Language Reference Manual](#)

Core constructs of the language and their semantics.

[Chapter 22 — urbiscript Standard Library](#)

The classes and methods provided in the standard library.

[Chapter 23 — 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>.

[Chapter 24 — Urbi 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 20

Programs

20.1 Environment Variables

There is a number of environment variables that alter the behavior of the Urbi tools.

20.1.1 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
```

Shell
Session

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
```

Shell
Session

20.1.2 Environment Variables

The following variables control the way the log messages are displayed. They are meant for developers.

GD_COLOR If set, force the use of colors in the logs, even if the output device does not seem to support them. See also **GD_NO_COLOR**.

GD_CATEGORY If set, a comma-separated list of category patterns with optional modifiers ‘+’ or ‘-’; for instance, ‘-Urbi.*’, Urbi.UObject’. See [Logger.set](#) for a more precise description of syntax, and [Section 22.36.2](#) for a description of the categories used by Urbi SDK.

GD_ENABLE_CATEGORY If set, a comma-separated list of categories or globs (e.g., ‘UValue’, ‘Urbi.*’) to display. All the others will be discarded. See also **GD_DISABLE_CATEGORY**.

GD_DISABLE_CATEGORY If set, a comma-separated list of categories or globs (e.g., ‘UValue’, ‘Urbi.*’) *not* to display. See also **GD_ENABLE_CATEGORY**.

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.

GD_LOC If set, display the location (file, line, function name) from which the log message was issued.

GD_NO_COLOR If set, forbid the use of colors in the logs, even if the output device seems to support them, or if **GD_COLOR** is set.

GD_PID If set, display the PID (Process Identity).

GD_THREAD If set, display the thread identity.

GD_TIME If set, display timestamps.

GD_TIMESTAMP_US If set, display timestamps in microseconds.

The following variables control more high-level features, typically to override the default behavior.

URBI_PATH The search-path for urbiscript source files (i.e., ‘*.u’ files).

URBI_ROOT 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 all the files. This variable permits to override that guess. Do not use it unless you know exactly what you are doing.

URBI_TEXT_MODE If set in the environment of a remote urbi-launch, disable binary protocol and force using urbiscript messages.

URBI_UOBJECT_PATH The search-path for UObject files. This is used by `urbi-launch`, by [System.loadModule](#) and [System.loadLibrary](#).

There are also very special environment variables, meant to be used only by Urbi developers, see [Section 19.2](#).

20.2 Special Files

‘CLIENT.INI’ This is the obsolete name for ‘global.u’.

‘global.u’ If found in the URBI_PATH (see [Section 20.1](#)), 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 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](#).

‘local.u’ If found in the URBI_PATH (see [Section 20.1](#)), 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’.

20.3 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 20.5](#)).

20.3.1 Options

General Options

`-h`, `--help`

Output the help message and exit successfully.

`--version`

Output version information and exit successfully.

`--print-root`

Output the `urbi-root` and exit successfully.

Tuning

`-d`, `--debug=level`

Set the verbosity level of traces. See the `GD_LEVEL` documentation ([Section 20.1.2](#)).

`-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 [System.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") };
[00000463] *** before
[00001463] *** after
```

urbiscript
Session

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.

Enabled if no input was provided (i.e., none of ‘**-e**’/‘**--expression**’, ‘**-f**’/‘**--file**’, ‘**-P**’/‘**-port**’ was given).

'-m', '--module=*module*'

Load the UObject*module*.

The options ‘**-e**’, ‘**-f**’ accumulate, and are run in the same **Lobby** as ‘**-i**’ if used. In other words, the following session is valid:

Shell Session

```
# 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
```

20.3.2 Quitting

To exit `urbi`, use the command `System.shutdown`.

If you are in interactive mode, you can also use the shortcut sequence C-d in the server lobby. The command `Lobby.quit` does not shutdown the server, it only closes the current lobby which closes the connection while in remote mode (using `telnet` for example), but only closes the interactive mode if performed on the server lobby.

On Unix systems, `urbi` handles the SIGINT signal (action performed when C-c is pressed). If you are in interactive mode, a first C-c kills the foreground job (or does nothing if no job is running on foreground).

For example consider the following piece of code:

```
every (4s)
  echo("Hello world!");
```

urbiscript
Session

If you try to execute this code, you will notice that you can no longer execute commands, since the `every` job is foreground (because of the semicolon). Now if you press C-c, the `every` job is killed, all the pending commands you may have typed are cleared from the execution queue, and you get back with a working interactive mode.

```
every (1s) echo("Hello world!");
[00010181] *** Hello world!
[00011181] *** Hello world!
// This command is entered while the foreground job blocks the evaluation.
// It waits to be executed.
echo("done");
[00012181] *** Hello world!
[00013181] *** Hello world!

// Pressing Control-C here:
[00014100:error] !!! received interruption, killing foreground job.
// Note that the "echo" is not run, the command queue is flushed.

// New commands are honored.
echo("interrupted");
[00019709] *** interrupted
```

urbiscript
Session

A second C-c (or SIGINT) within the 1.5s after the first one tries to execute `System.shutdown`. This can take a while, in particular if you have remote UObjects, since a clean shutdown will first take care of disconnecting them cleanly.

urbiscript
Session

```
// Two Control-C in a row.
[00493865:error] !!! received interruption, killing foreground job.
[00494672:error] !!! received interruption, shutting down.
```

Pressing C-c a third time triggers the default behavior: killing the program in emergency, by-passing all the cleanups.

In non-interactive mode (after a `Lobby.quit` on the server lobby for example), the first C-c executes `System.shutdown`, and the second one triggers the default behavior.

20.4 urbi-image — Querying Images from a Server

```
urbi-image option...
```

Shell
Session

Connect to an Urbi server, and fetch images from it, for instance from its camera.

20.4.1 Options

General Options

‘-h’, ‘--help’
 Output the help message and exit successfully.

‘--version’
 Output 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=bigest).

‘-s’, ‘--scale=*factor*’
 Rescale image with given *factor* (display only).

20.5 urbi-launch — Running a UObject

The `urbi-launch` program launches an Urbi system. It is more general than `urbi` (Section 20.3): everything `urbi` can do, `urbi-launch` can do it too.

20.5.1 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 ‘`--`’:

<code>urbi-launch [option...] module... [-- module-option...]</code>	Shell Session
--	------------------

The *modules* are looked for in the `URBI_UOBJECT_PATH`. The *module* extension (‘`.so`’, or ‘`.dll`’) does not need to be specified.

Options

'-h', '--help'

Output the help message and exit successfully.

'--version'

Output version information and exit successfully.

'--print-root'

Output the *urbi-root* 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 20.1.2](#)).

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 20.3](#).

20.5.2 Examples

To launch a fresh server in an interactive session with the `UMachine` UObject compiled as the file '`test/machine.so`' (or '`test/machine.dll`' plugged in, run:

```
urbi-launch --start test/machine -- --interactive
```

Shell
Session

To start an Urbi server accepting connections on the local port 54000 from any remote host, with `UMachine` plugged in, run:

```
urbi-launch --start --host 0.0.0.0 --port 54000 test/machine
```

Shell
Session

Since `urbi-launch` in server mode is basically the same as running the `urbi` program, both programs are quit the same way (see [Section 20.3.2](#)).

20.5.3 UObject Module Options

Being a program that runs programs `urbi-launch` is also providing access to the options supported by the UObject middleware: pass them after '--' in plugin mode. For instance:

```
$ urbi-launch test/all -- --version
Urbi version 3.x.y
```

Shell
Session

The options supporting by the UObject middleware are described below.

UObject options

`'-h', '--help'`

Output the help message and exit successfully.

`--version`

Output version information and exit successfully.

`--describe`

Describe the loaded modules on the standard output and exit.

`--describe-file=file`

Describe the loaded modules in the *file* exit.

`--no-sync-client`

Use an asynchronous connection. This mode is extremely dangerous, any attempt to use synchronous operation will crash.

`'-s', '--stay-alive'`

Do not disconnect at the end of the program.

`'-d', '--disconnect'`

Disconnect at the end of the program. This is the default.

`--server`

Start the remote in server mode.

Networking Beside the usual options (`--host`, `--port`, `--port-file`, see [Section 20.3](#)), the following options are supported.

`'-b', '--buffer=size'`

Specify the buffer size.

Input

`'-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.

`'-m', '--module=module'`

Load the UObject*module*.

20.6 urbi-launch-java — Running a Java UObject

Java UObject can only be run remotely, they cannot be “plugged”. So, while `urbi-launch-java` is really alike `urbi-launch`, it is not the same either. See also [Section 27.1.2](#) for a more tutorial-like introduction to `urbi-launch-java`.

20.6.1 Invoking `urbi-launch-java`

`urbi-launch-java` launches Java UObject, either in plugged-in mode, or in remote mode. Since UObject can also accept options, the command line features two parts, separated by `--`:

<code>urbi-launch-java [option...] module... [-- module-option...]</code>	<small>Shell Session</small>
---	------------------------------

The *modules* are looked for in the `URBI_UOBJECT_PATH`. The *module* extension (`'.so'`, or `'.dll'`) does not need to be specified.

Options

-h, **--help**

Output the help message and exit successfully.

--version

Output version information and exit successfully.

--print-root

Output the *urbi-root* and exit successfully.

-C, **--check**

Exit successfully if and only if Java UObject is usable (it might not be compiled or installed).

-d, **--debug=level**

Set the verbosity level of traces. See the GD_LEVEL documentation ([Section 20.1.2](#)).

Java

-c, **--classpath=path**

Pass a colon-separated list of directories, JAR archives, and ZIP archives to search for class files.

Networking `urbi-launch-java` supports the same networking options ('**--host**', '**--port**', '**--port-file**') as `urbi`, see [Section 20.3](#).

20.7 urbi-ping — Checking the Delays with a Server

<code>urbi-ping option... [host] [interval] [count]</code>	Shell Session
--	------------------

Send “ping”s to an Urbi server, and report how long it took for “pong”s to come back.

20.7.1 Options

General Options

-h, **--help**

Output the help message and exit successfully.

--version

Output 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

‘-c’, ‘--count=count’

Number of pings to send. Pass 0 to send an unlimited number of pings. Defaults to 0.

‘-i’, ‘--interval=interval’

Set the delays between pings, in milliseconds. Pass 0 to flood the connection. Defaults to 1000.

20.8 urbi-send — Sending urbiscript Commands to a Server

Shell
Session

```
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.

‘-m’, ‘--module=module’

Load the UObject*module*. This is equivalent to ‘-e ’loadModule("module");’’.

‘-h’, ‘--help’

Output 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;’’ at the end of the options. 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’

Output version information and exit successfully.

20.9 urbi-sound — Querying Sounds from a Server

Shell
Session

```
urbi-sound option...
```

Connect to an Urbi server, and record sounds from it, and play them on ‘/dev/dsp’. This is only supported on GNU/Linux. If ‘[o]output is specified, the recording is saved instead of being played.

20.9.1 Options

General Options

'-h', '--help'

Output the help message and exit successfully.

'--version'

Output 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

'-d', '--device=*device*'

Query image on *device.val* (default: `camera`).

'-D', '--duration=*duration*'

Specify, in seconds, how long the recording is performed.

'-n', '--no-header'

When saving the sound, do not include the WAV headers in the file.

'-o', '--output=*file*'

Save the sound into *file*.

20.10 umake — Compiling UObject Components

The `umake` programs builds loadable modules, UObject, to be later run using `urbi-launch` ([Section 20.5](#)). 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.

20.10.1 Invoking `umake`

Usage:

<code>umake option... file...</code>	<small>Shell Session</small>
--------------------------------------	------------------------------

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.

There are several environment variables that `umake` reads:

`CPPFLAGS` Options passed to the preprocessor.

`CXXFLAGS` Options passed to the C++ compiler.

`LDLFLAGS` Options passed to the linker.

The arguments of `umake` may also include assignments, i.e., arguments of the type '*var=val*'. The behavior depends on *var*, the name of the variable:

`EXTRA_CPPFLAGS` Appended to the options passed to the preprocessor.

`EXTRA_CXXFLAGS` Appended to the options passed to the C++ compiler.

`EXTRA_LDLFLAGS` Appended to the options passed to the linker.

`VPATH` Appended to the `VPATH`, i.e., the list of directories in which the sources are looked for.

Otherwise, the argument is passed as is to `make`.

General options

`--debug`

Turn on shell debugging (`set -x`) to track `umake` problems.

`-h`, `--help`

Output the help message and exit successfully.

`-q`, `--quiet`

Produce no output except errors.

`-v`, `--version`

Output 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.

--package=pkg'
Use `pkg-config` to retrieve compilation flags (`cflags` and `libs`). Fails if `pkg-config` does not know about the package `pkg`.

Compiler and linker options

'-Dsymbol'
Forwarded to the preprocessor (i.e., added to the `EXTRA_CPPFLAGS`).

'-Ipath'
Forwarded to the preprocessor (i.e., added to the `EXTRA_CPPFLAGS`).

'-Lpath'
Forwarded to the linker (i.e., added to the `EXTRA_LDFLAGS`).

'-l lib'
Forwarded to the linker (i.e., added to the `EXTRA_LDFLAGS`).

Developer options

'-p', '--prefix=dir'
Set library files location.

'-P', '--param-mk=file'
Set '`param.mk`' location.

'-k', '--kernel=dir'
Set the kernel location.

20.10.2 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 20.5). Same as '`umake --shared-library`'.

Chapter 21

urbiscript Language Reference Manual

21.1 Syntax

21.1.1 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 21.1.6.6](#)), 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'
```

urbiscript
Session

21.1.2 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  
/*
```

urbiscript
Session

```

/*
 * Multi-line.
 */
;

[00000006] 7

```

21.1.3 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 previous saved location. `//#push` and `//#pop` must match.

21.1.4 Identifiers

Identifiers in urbiscript are composed of one or more alphanumeric or underscore (`_`) characters, not starting by a digit. Additionally, identifiers must not match any of the urbiscript reserved words¹ documented in Section 21.1.5. Identifiers can also be written between simple quotes ('), in which case they may contain any character.

urbiscript
Session

```

var x;
var foobar51;
var this.a_name_with_underscores;

// Invalid because "if" is a keyword.
var if;
[00000498:error] !!! syntax error: unexpected if
obj.if();
[00013826:error] !!! syntax error: unexpected if

// However, keywords can be escaped with simple quotes.
var 'if';
var this.'else';

// Invalid identifiers: cannot start with a digit.
var 3x;
[00000009:error] !!! syntax error: invalid token: '3x'
obj.3x();
[00000009:error] !!! syntax error: invalid token: '3x'

// Identifiers can be escaped with simple quotes.
var '3x';

```

¹ The only exception to this rule is `new`, which can be used as the method identifier in a method call.

```
var '%x';
var '1 2 3';
var this.'[]';
```

21.1.5 Keywords

Keywords are reserved words that cannot be used as identifiers, for instance. They are listed in [Table 21.1](#).

21.1.6 Literals

21.1.6.1 Angles

Angles are floats (see [Section 21.1.6.4](#)) 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 21.2](#).

```
Math.pi == 180deg;
Math.pi == 200grad;
```

Assertion
Block

21.1.6.2 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](#) 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] [ => ]
["a" => 1, "b" => 2, "c" => 3];
[00000000] ["a" => 1, "b" => 2, "c" => 3]
["one" => 1, "one" => 2];
[00000001:error] !!! duplicate dictionary key: "one"
```

urbiscript
Session

21.1.6.3 Durations

Durations are floats (see [Section 21.1.6.4](#)) 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 21.3](#).

Assertion
Block

```
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;
```

21.1.6.4 Floats

urbiscript supports the *scientific notation* for floating-point literals. See [Float](#) for more details. Examples include:

```
1 == 1;
1 == 1.0;
1.2 == 1.2000;
1.234e6 == 1234000;
1e+11 == 1E+11;
1e10 == 10000000000;
1e30 == 1e10 * 1e10 * 1e10;
```

Assertion Block

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

0.0000000001;
[00000013] 1e-11

1e+3;
[00000014] 1000

1E-5;
[00000014] 1e-05
```

urbiscript Session

In order to tell apart 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 ;
```

urbiscript Session

Hexadecimal notation is supported for integers: 0x followed by one or more hexadecimal digits, whose case is irrelevant.

```
0x2a == 42;
0x2A == 42;
0xabcdef == 11259375;
0xABCD E == 11259375;
0xFFFFFFFF == 4294967295;
```

Assertion Block

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'
```

urbiscript Session

In order to augment readability, you may separate digits with underscores.

```
123_456_789 == 123456789;
12_34_56_78_90 == 1234567890;
1_2__3___45 == 12345;
1_2.3_4 == 12.34;
0xFFFF_FFFF == 0xFFFFFFFF;
1e1_0 == 1e10;
```

Assertion Block

The underscores must always be *between* digits.

urbiscript
Session

```
// This is actually an identifier.
_0__2;
[00000112:error] !!! lookup failed: _0__2

20_;
[00029616:error] !!! syntax error: invalid token: '20_'

// This is actually a call to the method '_2' in 1.
1._2;
[00029653:error] !!! lookup failed: _2

1.2_;
[00029654:error] !!! syntax error: invalid token: '1.2_'

0x_12;
[00116580:error] !!! syntax error: invalid token: '0x_12'
0_x12;
[00116582:error] !!! syntax error: invalid token: '0_x12'

1e_10;
[00253832:error] !!! syntax error: invalid token: '1e_10'
1e10_;
[00257727:error] !!! syntax error: invalid token: '1e10_'
```

21.1.6.5 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](#) for more details.

urbiscript
Session

```
[]; // The empty list
[00000000] []
[1, 2, 3];
[00000000] [1, 2, 3]
```

21.1.6.6 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 21.4](#).

Assertion
Block

```
// 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)(\"")" == "\"\\\"";
```

Invalid escapes are errors.

```
"\h\777\ ";
[00000002:error] !!! syntax error: invalid character after \-escape: 'h'
[00000005:error] !!! syntax error: invalid number after \-escape: '777'
[00000004:error] !!! syntax error: invalid character after \-escape: ' '
```

urbiscript
Session

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";
```

Assertion
Block

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
```

urbiscript
Session

See [String](#) for more details.

21.1.6.7 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](#) for more details.

```
();;
[00000000] ()
(1,);
[00000000] (1,)
(1, 2);
[00000000] (1, 2)
(1, 2, 3, 4,);
[00000000] (1, 2, 3, 4)
```

urbiscript
Session

21.1.6.8 Pseudo classes

Objects meant to serve as prototypes are best defined using the [class](#) construct. See also the tutorial, [Section 8.4](#).

This results in the (constant) definition of the name *lvalue* in the current context ([class](#) construct can be used inside a scope or in an object) with:

- a slot named `type` which is the trailing component of *lvalue* as a [String](#);
- a slot named `as type` that returns `this`.

- the list of prototypes is equal to the list *prototypes* that served as parent, defaulting to `Object`;
- the *block* is evaluated in the context of this object, as with a `do`-block;
- the value of the whole statement is the newly defined object.

```
urbiscript
Session

class Base
{
    var slot = 12;
}|;

assert
{
    hasLocalSlot("Base");
    Base.type == "Base";
    Base.protos == [Object];
    Base.slot == 12;
    Base.asBase() === Base;
};

class Global.Derive : Base
{
    var slot = 34;
}|;

assert
{
    Global.hasLocalSlot("Derive");
    Global.Derive.type == "Derive";
    Global.Derive.protos == [Base];
    Global.Derive.slot == 34;
    Global.Derive.asDerive() === Global.Derive;
    Global.Derive.asBase() === Global.Derive;
};

class Base2 {}|;

class Derive2 : Base, Base2 {}|;

assert
{
    Derive2.type == "Derive2";
    Derive2.protos == [Base, Base2];
    Derive2.slot == 12;
    Derive2.asDerive2() === Derive2;
    Derive2.asBase() === Derive2;
    Derive2.asBase2() === Derive2;
};
```

It is guaranteed that the expressions that define the class name and its parents are evaluated only once.

```
urbiscript
Session

function verboseId(var x)
{
    echo(x) | x
}|;
class verboseId(Global).math : verboseId(Math)
{
};
[00000686] *** Global
[00000686] *** Math
[00000686] math
```

21.1.7 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 fine tuned sequentiality, urbiscript features four different statement-separators (or connectors):

‘;’ sequentiality

‘|’ tight sequentiality

‘,’ background concurrency

‘&’ fair-start concurrency

21.1.7.1 ‘;’

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.

urbiscript
Session

```
1; 2; 3;
[00000000] 1
[00000000] 2
[00000000] 3
```

21.1.7.2 ‘,’

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](#) or [Events](#) 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
```

urbiscript
Session

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
```

urbiscript
Session

```
urbiscript
Session
{
    sleep(100ms) | echo("1"),
    sleep(400ms) | echo("2"),
};

echo("done");
[00000316] *** 1
[00000316] *** 2
[00000316] *** done
```

21.1.7.3 ‘|’

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.

```
urbiscript
Session
{
    { echo("11") ; sleep(100ms) ; echo("12") },
    { echo("21") ; sleep(400ms) ; echo("22") },
};

[00000316] *** 11
[00000316] *** 21
[00000316] *** 12
[00000316] *** 22
```

```
urbiscript
Session
{
    { 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.

21.1.7.4 ‘&’

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.

21.1.8 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 used for any other purpose — for instance, + is also used to concatenate lists and strings. Their semantics is thus not limited to what is presented here.

Tables in this section sort operators by decreasing 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 call. In this case, the equivalent desugared expression is shown in the “Equivalence” column. To define an operator for an object, override the corresponding method (see [Section 8.6](#)).

This section defines the syntax (what they look like), precedence and associativity of the operators. Their semantics (what they actually do) is described in [Chapter 22](#) in the documentation of the classes that provide them.

21.1.8.1 Arithmetic operators

urbiscript supports classic *arithmetic operators*, with the classic semantics on numeric values. See [Table 21.5](#) and the listing below.

```
1 + 1 ==      2;
1 - 2 ==      -1; 1 - 2 - 3 == (1 - 2) - 3 != 1 - (2 - 3);
2 * 3 ==      6;
10 / 2 ==      5; 6 / 3 / 2 == (6 / 3) / 2 != 6 / (3 / 2);
2 ** 10 ==    1024; 2 ** 3 ** 2 == 2 ** (3 ** 2) != (2 ** 3) ** 2;
-(1 + 2) ==    -3;
1 + 2 * 3 ==    7;
(1 + 2) * 3 ==    9;
-2 ** 2 ==     -4 == -(2 ** 2) != (-2) ** 2;
----- 1 ==      1;
```

Assertion Block

21.1.8.2 Assignment operators

Assignment in urbiscript can be performed with the `=` operator. Assignment operators, such as `+=`, are supported too, see [Table 21.6](#) and the examples below.

The following example demonstrates that `a += b` behaves as `a = a + b` for Floats.

```
var y = 0;
[00000000] 0
y = 10;
[00000000] 10
y += 10;
[00000000] 20
y /= 5;
[00000000] 4
```

urbiscript Session

These operators are redefinable. Indeed, `a += b` is actually processed as `a = a.'+='(b)`. This definition, which is neither that of C (`a = a.'+'(b)`) nor that of C++ (`a.'+='(b)`), provides support for both *immutable* and *mutable* values.

Immutable Values Small objects such as Floats should typically be immutable, i.e., the value of a Float cannot change:

```
var value = 0|;
var valueAlias = value|;
value += 10;
[00002275] 10
valueAlias;
[00002301] 0
```

urbiscript Session

It would be traitorous for most users that `valueAlias` be equal to 10 too. That's why `Float.'+='` (which is actually `Object.'+='`) simply bounces to `Float.'+'`. The "net result" of `value += 10` is therefore `value = value.'+'(10)`, i.e., a new Float is computed from `0.'+'(10)`, and `value` is rebound to it. The binding from `valueAlias` to 0 is left as is.

Mutable Values On the contrary, large, "updatable" objects should provide an implementation of `'+='` that mutates them. For instance, implementing `a.'+='(b)` as `a.'+'(b)` would be too costly for [Lists](#). Each time `+=` is used, we need to create a new List (whose content is that of `a`), then to append the contents of `b`, and finally throw away the former value of `a`.

Not only is this inefficient, this is also wrong (at least from a certain point of view). Indeed, since we no longer update the List pointed to by `a`, but rather store a new List, everything that was special to the original List (its uid or whatever special slot the user may have defined) is lost. The proper implementation of `List.'+='` is therefore to *modify* `this` by appending the added members.

```
var myList = []!;
var myList.specialFeature = 42!;
myList += [1, 2, 3];
[00848865] [1, 2, 3]
myList.specialFeature;
[00848869] 42
var myOtherList = myList + [4, 5];
[00848873] [1, 2, 3, 4, 5]
myOtherList.specialFeature;
[00848926:error] !!! lookup failed: specialFeature
```

Note however that this means that because $a += b$ is *not* processed as $a = a + b$, aliases to a are possibly modified.

urbiscript
Session

```
var something = []!;
var somethingElse = something!;
something += [1, 2];
[00008557] [1, 2]
somethingElse += [3, 4];
[00008562] [1, 2, 3, 4]
something;
[00008566] [1, 2, 3, 4]
```

Example So basically, the rules to redefine these operators are:

Immutable (small) objects `'+='` should redirect to `'+'` (which of course should *not* modify its target).

Mutable (large) objects `'+='` should update `this` and return it.

The following examples contrasts both approaches.

urbiscript
Session

```
class Counter
{
    var count = 0;
    function init (n) { var this.count = n };
    // Display the value, and the identity.
    function asString() { "%s @ %s" % [count, uid] };
    function '+'(var n) { new(count + n) };
    function '-'(var n) { new(count - n) };
}
```

```
class ImmutableCounter : Counter
{
    function '+='(var n) { this + n };
    function '-='(var n) { this - n };
}

var ic1 = ImmutableCounter.new(0);
[00010566] 0 @ 0x100354b70
var ic2 = ic1;
[00010574] 0 @ 0x100354b70

ic1 += 1;
[00010588] 1 @ 0x10875bee0

// ic1 points to a new object.
ic1;
[00010592] 1 @ 0x10875bee0
// ic2 still points to its original value.
```

```
ic2;
[00010594] 0 @ 0x100354b70
```

```

class MutableCounter : Counter
{
    function '+='(var n) { count += n | this };
    function '-='(var n) { count -= n | this };
};

var mc1 = MutableCounter.new(0);
[00029902] 0 @ 0x100364e00
var mc2 = mc1;
[00029911] 0 @ 0x100364e00

```

```

mc1 += 1;
[00029925] 1 @ 0x100364e00

// mc1 points to the same, updated, object.
mc1;
[00029930] 1 @ 0x100364e00
// mc2 too.
mc2;
[00029936] 1 @ 0x100364e00

```

21.1.8.3 Increment/decrement Operators

In the tradition of C, urbiscript provides pre- and postfix increment and decrement operators (Table 21.7): `++a` and `a++`.

Prefix operators modify their operand and evaluate to the new value, whereas postfix operators evaluate to the *former* value.

These operators *modify* the variable/slot they are applied to

Assertion Block

```

var count = 5;
var alias = count;

alias = count; count++ == 5; count == 6; alias == 5; count !== alias;
alias = count; count++ == 6; count == 7; alias == 6; count !== alias;
alias = count; count-- == 7; count == 6; alias == 7; count !== alias;
alias = count; --count == 5; count == 5; alias == 6; count !== alias;
alias = count; ++count == 6; count == 6; alias == 5; count !== alias;

```

Similarly to assignment operators, these operators are redefinable. Indeed, `++a` is actually processed like `a = a.'++'`, and `a++` is actually processed like `{ var '$save' = a | ++a | '$save' }` (which, in turn, is like `{ var '$save' = a | a = a.'++' | '$save' }`). In other words, you are entitled to redefine the operator `'++'` whose semantics is “return the successor of `this`”.

Beware that the function `'++'` should *not* modify its target, but rather return a fresh value. Indeed, if it alters `this`, the copy made in `'\$/save'` will also have its value updated. In other words, the value of `a++` would be its new one, not its former one.

21.1.8.4 Bitwise operators

urbiscript features *bitwise operators*. They are also used for other purpose than bit-related operations. See Table 21.8 and the listing below.

Assertion Block

```

4 << 2 == 16;
4 >> 2 == 1;

compl 0x0FOF_FOF0 == 0xFOFO_OF0F;

0x0000 bitand 0xFOFO == 0x0000;
0x0FOF bitand 0xFOFO == 0x0000;
0xFOFO bitand 0xFOFO == 0xFOFO;
0xFFFF bitand 0xFOFO == 0xFOFO;

0x0000 bitor 0xFOFO == 0xFOFO;
0x0FOF bitor 0xFOFO == 0xFFFF;
0xFOFO bitor 0xFOFO == 0xFOFO;
0xFFFF bitor 0xFOFO == 0xFFFF;

0x0000 ^ 0xFOFO == 0xFOFO;
0x0FOF ^ 0xFOFO == 0xFFFF;
0xFOFO ^ 0xFOFO == 0x0000;
0xFFFF ^ 0xFOFO == 0x0FOF;

```

21.1.8.5 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.

The operator `!` returns the Boolean that is the negation of the value of its operand. See `Object.'!'`.

```
!true    === false; !false === true;
!42     === false; !0     === true;
!"42"   === false; !"  === true;
![42]   === false; ![]   === true;
![42=>2] === false; ![=>] === true;
```

Assertion Block

The operator `&&`, the short-circuiting logical and, behaves as follows. If the left-hand side operand evaluates to a “true” value, return the evaluation of the right-hand side operand; otherwise return the value of the left-hand side operand (not necessarily `false`).

Assertion Block

```
true && true; !(true && false); !(false && true); !(false && false);

(0 && "foo") == 0;
(2 && "foo") == "foo";

("") && "foo" == "";
("foo" && "bar") == "bar";
```

Its arguments are evaluated at most once.

urbiscript Session

```
var zero = 0|;
var one = 1|;
var two = 2|;

// First argument evaluated once, second is not needed.
({ echo("lhs") | zero } && { echo("rhs") | one }) === zero;
[00029936] *** lhs
[00029936] true

({ echo("lhs") | one } && { echo("rhs") | two }) === two;
[00029966] *** lhs
[00029966] *** rhs
[00029966] true
```

The operator `||`, the short-circuiting logical or, behaves as follows. If the left-hand side operand evaluates to a “false” value, return the evaluation of the right-hand side operand; otherwise return the value of the left-hand side argument (not necessarily `true`).

```
true || false; true || true; false || true; !(false || false);

(0 || "foo") == "foo";
(2 || 1/0) == 2;

("") || "foo" == "foo";
("foo" || 1/0) == "foo";
```

Assertion Block

Its arguments are evaluated at most once.

```
var zero = 0|;
var one = 1|;
var two = 2|;

// First argument evaluated once, second is not needed.
({ echo("lhs") | one } || { echo("rhs") | two }) === one;
[00029936] *** lhs
[00029936] true

({ echo("lhs") | zero } || { echo("rhs") | one }) === one;
```

urbiscript Session

```
[00029966] *** lhs
[00029966] *** rhs
[00029966] true
```

See [Section 22.3.1](#) for more information about “true” and “false” values.

21.1.8.6 Comparison operators

urbiscript supports classical *comparison operators*, plus a few of its own. See [Table 21.10](#) and the listing below.

For a description of the semantics of these operators, see their definition as slots of [Object](#).

```
! (0 < 0);
0 <= 0;
0 == 0;
0 != 0;

var z = 0;
z === z;
! (z != z);
```

Assertion Block

The comparisons can be chained, e.g., `var{a} < b <= c`, in which case:

- the sub-expression are evaluated left-to-right (i.e., the order of evaluation is *a*, then *b*, then *c*);
- the sub-evaluations are evaluated at most once;
- expressions that are not needed are not evaluated (i.e., if `!(var{a} < b)`, then *c* is not evaluated).

In other words, `var{a} < b <= c` is equivalent to `var{a} < b && b <= c` with the guaranty that *b* is not evaluated twice. Any sequence of comparison is valid, even those that mathematicians would not write (e.g., *a* < *b* > *c* == *d*).

```
function v(x) { echo(x) | x }|;

v(1) < v(2) <= v(3) != v(4) == v(4) > v(0) >= v(0) !== v(Global) === v(Global);
[00000010] *** 1
[00000010] *** 2
[00000010] *** 3
[00000010] *** 4
[00000010] *** 4
[00000010] *** 0
[00000010] *** 0
[00000010] *** Global
[00000010] *** Global
[00000010] true

v(1) == v(2) < v(3) < v(4);
[00033927] *** 1
[00033927] *** 2
[00033927] false

v(1) == v(2) < v(3) < v(4) || v(10) < v(11) < v(12);
[00033933] *** 1
[00033933] *** 2
[00033933] *** 10
[00033933] *** 11
[00033933] *** 12
[00033933] true
```

urbiscript Session

21.1.8.7 Container operators

These operators work on containers and their members. See [Table 21.11](#).

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](#)). 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];
```

Assertion Block

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.

Assertion Block

```
// On lists.
var l = [1, 2, 3, 4, 5];
l[0] == 1;
l[-1] == 5;
(l[0] = 10) == 10;
l == [10, 2, 3, 4, 5];

// On strings.
var s = "abcdef";
s[0] == "a";
s[1,3] == "bc";
(s[1,3] = "foo") == "foo";
s == "afoodef";
```

Assertion Block

21.1.8.8 Object operators

These core operators provide access to slots and their properties. See [Table 21.12](#).

Assertion Block

```
var obj = Object.new();
var obj.f = function() { 24 };

obj.f() == 24;
obj.&f != 24;
obj.&f.isA(Slot);
obj.&f === obj.getSlot("f");
```

Assertion Block

21.1.8.9 All operators summary

[Table 21.13](#) 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.

21.2 Scopes and local variables

21.2.1 Scopes

Scopes are sequences of statements, enclosed in curly brackets (`{}`). Statements are separated with the four statements separators (see [Section 21.1.7](#)). 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 | {};
```

Urbiscript Session

```
// 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;
    2;
    3; // This last separator is optional.
};

[00000000] 3
// Scopes can be used as expressions
{1; 2; 3} + 1;
[00000000] 4
```

urbiscript
Session

21.2.2 Local variables

Local variables are introduced with the `var` keyword, followed by an identifier (see [Section 21.1.4](#)) and an optional initialization value assignment. If the initial value is omitted, it defaults to `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
```

urbiscript
Session

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);
    };
    // This refers to the outer y.
    echo(y);
    // This would be invalid: z does not exist anymore.
```

urbiscript
Session

³Local variables can actually escape their scope with lexical closures, see [Section 21.3.6](#).

```
// 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
```

21.3 Functions

21.3.1 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. The syntax to declare named or *anonymous functions* is as follows:

In words: One can create an anonymous function thanks to the `function` keyword (or `closure`, see [Section 21.3.6](#)), followed by the list of formal arguments and a block 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.

urbiscript
Session

```
function () { echo(0) }|;

function (arg1, arg2) { echo(0) }|;

function (
    arg1, // Ignored argument.
    arg2, // Also ignored.
)
{
    echo(0)
}|;
```

Declaring a named function is actually strictly equivalent to binding a variable to the corresponding anonymous function:

urbiscript
Session

```
// Functions are often stored in variables to call them later.
const var f1 = function () { "hello" }|

// This form is strictly equivalent, yet simpler.
function f2() { "hello" }|

assert (f1() == f2());
```

Therefore, like regular values, functions can either be plain local variables or slots of objects. In the following example, initially the object `Foo` features neither a `foo` nor a `bar` slot, but its `init` function declares a *local* `foo` function, and a *slot* `bar`. The whole difference is the initial `this` in the definition of `bar` which makes it a slot, not a variable.

urbiscript
Session

```
class Foo
{
    function init()
    {
        // This is a function local to init().
        function foo() { 42 };
        function this.bar() { 51 };
        foo() + bar();
    };
}|;
```

```

Foo.foo();
[00001720:error] !!! lookup failed: foo
Foo.bar();
[00001750:error] !!! lookup failed: bar

[00001787] 93
Foo.init();
Foo.foo();
[00001787:error] !!! lookup failed: foo
Foo.bar();
[00001818] 51

```

21.3.2 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

```

urbiscript
Session

Non-empty lists of effective arguments may end with an optional comma.

```

f(
    "bar",
    "foo",
);
[00000000] *** foobar

```

urbiscript
Session

21.3.2.1 Default value

Arguments may have a default value specified at the declaration point, and are thus optional when calling the function:

```

f = function(x, y=10, z=100) { x+y+z}|;
f();
[00000001:error] !!! f: expected between 1 and 3 arguments, given 0
f(2);
[00000002] 112
f(2, 30);
[00000003] 132
f(2, 30, 400);
[00000004] 432

```

urbiscript
Session

21.3.2.2 Argument typing

Arguments can be constrained to a given type using the `argname: type` syntax. Urbi will check that the argument inherits from `type` and throw an error if it is not the case.

```

f = function(x: Float) {x}|;
f(1);
[00000001] 1

```

urbiscript
Session

```
f("foo");
[00000002:error] !!! f: argument 1: unexpected "foo", expected a Float

f = function(x: 1) {x}|;
f(1);
[00000002:error] !!! f: argument 1: unexpected 1, expected a Float
// It did not work because 1 does not inherit from "1",
// they are two different objects both inheriting from Float
```

Argument typing can be combined with default values, but the default value must come before the type:

```
urbiscript
Session
f = function(x=1 : Float) {x}|;
f(2);
[00000001] 2
f();
[00000002] 1
f("foo");
[00000002:error] !!! f: argument 1: unexpected "foo", expected a Float

// This will not do what you expect...
f = function(x: Float = 1) {x}|;
// ... but you will find out fast:
f();
[00232426:error] !!! f: expected 1 argument, given 0
f(1);
[00235466:error] !!! f: argument 1: unexpected 1, expected a Float
```

21.3.3 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.

The execution of the function can be interrupted by a `return` statement. The control flow then resumes to the caller. The value of the function call is the argument provided to the `return` statement, or `void` if no argument was provided.

```
urbiscript
Session
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.
```

Avoid useless `returns`, see [Section 16.1.2](#).

21.3.4 Call messages

Functions can access meta-information about how they were called, via a `CallMessage` object. The *call message* associated with a function can be accessed with the `call` keyword. It contains information such as not-yet evaluated arguments, the name of the function, the target...

21.3.5 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.

urbiscript
Session

```
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 arguments *not* evaluated. The listing below gives an example. More information about this can be found in the `CallMessage` class documentation.

urbiscript
Session

```
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.

urbiscript
Session

```
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
```

21.3.6 Closures

urbiscript features *closures* (sometimes referred to as *lexical closures*, or *function closures*): function bodies can use non-local variables whose scope might even have been closed.

There is nothing surprising at first sight in the following piece of code:

```
urbiscript
Session
var n = 0|
function cl()
{
    // n refers to a variable outside the function.
    ++n;
}|;
assert
{
    cl() == 1;
    n == 1;
    ++n == 2;
    cl() == 3;
};
```

Actually, even C supports this, with `n` being a simple global variable. But actually the `cl` function has *captured* an access to `n`, which will even survive the destruction of its enclosing scope:

```
urbiscript
Session
{
    var local = 0;
    function lobby.getLocal() { local };
    function lobby.setLocal(var x) { local = x };
}|;
assert
{
    getLocal() == 0;
    setLocal(42);
    getLocal() == 42;
};
```

As demonstrated, urbiscript supports read/write closures, which provides true sharing between the function and the outer environment.

It is sometimes needed to also capture the current value of `this` (and actually, of the current `Lobby`). In some languages, this is called a *delegate*, in urbiscript use `closure` instead of `function`. Contrast the following two runs which differ only by `function` vs. `closure`.

```
urbiscript
Session
{
    var x = 0;
    class lobby.Foo
    {
        function fun () { (++x, Lobby.lobby, type) };
        closure clo () { (++x, Lobby.lobby, type) };
    };
}|;
var aLobby = Lobby.create();
var aLobby.name = "A Fresh Lobby";
lobby.name = "The main Lobby";

var Global.fun = Foo.getSlotValue("fun");
var Global.clo = Foo.getSlotValue("clo");

// A "function" invoked in the context of a different lobby and a different
// this refers to the "new" lobby and "this":
aLobby.receive("Global.fun();");
[00000012] (1, Lobby<A Fresh Lobby>, "Global")

// A "closure" invoked in the context of a different lobby and a different
// this refers to the "original" lobby and "this":
```

```
aLobby.receive("Global.clo();");
[00000014] (2, Lobby<The main Lobby>, "Foo")
```

21.3.7 Variadic functions

Variadic functions are functions that take a variable number of arguments. They are created by appending `[]` to a formal argument: the function will accept any number of arguments, and they will be assigned to the variadic formal argument as a list.

```
function variadic(var args[])
{
    echo(args)
} |
```

```
variadic();
[00000000] *** []
variadic(1, 2, 3);
[00000000] *** [1, 2, 3]
```

urbiscript
Session

There can be other formal arguments, as long as the variadic argument is at the last position. If `n` is the number of non variadic arguments, the function will request as least `n` effective arguments, which will be assigned to the non variadic arguments in order like a classical function call. All remaining arguments will be passed in list as the variadic argument.

```
function invalid(var args[], var last)
{} |;
[00000000:error] !!! syntax error: argument after list-argument
```

```
function variadic(var a1, var a2, var a3, var args[])
{
    echo(a1);
    echo(a2);
    echo(a3);
    echo(args)
} |
```

```
// Not enough arguments.
variadic();
[00000000:error] !!! variadic: expected at least 3 arguments, given 0
```

```
// No variadic arguments.
variadic(1, 2, 3);
[00000000] *** 1
[00000000] *** 2
[00000000] *** 3
[00000000] *** []
```

```
// Two variadic arguments.
variadic(1, 2, 3, 4, 5);
[00000000] *** 1
[00000000] *** 2
[00000000] *** 3
[00000000] *** [4, 5]
```

urbiscript
Session

21.4 Objects

Any urbiscript value is an object. Objects contain:

- A set of slots, which associate an object to a name.
- A list of prototypes, which are also objects.

21.4.1 Slots

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.

In urbiscript, a slot is itself an object, and stores meta-information about the value it contains, such as its constness, setter and getter function...

21.4.1.1 Manipulation

The `Object.createSlot` function adds a slot to an object with the `void` value. The `Object.updateSlot` function changes the value of a slot; `Object.getSlot` reads it. The `Object.setSlot` method creates a slot with a given value. Finally, the `Object.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 22.41](#).

```
var o = Object.new() |
assert (o.localSlotNames() == []);

o.createSlot("test");
assert
{
    o.localSlotNames() == ["test"];
    o.getSlotValue("test").isVoid;
};

o.updateSlot("test", 42);
[00000000] 42
assert
{
    o.getSlotValue("test") == 42;
};
```

urbiscript
Session

21.4.1.2 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.setSlotValue("name", value)`.
- `o.name = value` is equivalent to `o.updateSlot("name", value)`.
- `o.&name` is equivalent to `o.getSlotValue("name")` (`o` can be omitted, like for regular method invocations: `&name` is equivalent to `getSlot("name")`).

21.4.2 Properties

Slots can have properties, see [Section 8.7](#) for an introduction to properties.

21.4.2.1 Manipulation

There is a number of functions to manipulate properties:

- `Object.setProperty`, to define/set a property.
- `Object.getProperty`, to get a property.
- `Object.removeProperty`, to delete a property.
- `Object.hasProperty`, to test for the existence of a property.

- `Object.properties`, to get all the properties of a slot.

There is also syntactic sugar for some of them:

- `slot->name` is equivalent to `getProperty("slot", "name")`.
- `slot->name = value` is equivalent to `setProperty("slot", "name", value)`.

21.4.3 Direct slot access

21.4.3.1 Getting and setting slots

Slots are themselves objects that can be accessed using `Object.getSlot`. Slots of a known name can be accessed by the `&` alias.

urbiscript
Session

```
class a {
    var c = 0;
}|;
var s = a.getSlot("c");
[00000001] Slot_0x42339508
a.c = 1;
[00000002] 1
assert
{
    /// The slot content has changed, but the slot itself remains the same
    a.getSlot("c") === s;
    /// Alias for getSlot.
    a.&c === s;
};
```

`Object.setSlot` can be used to set the Slot in its owning object. One possible use is to share a Slot between multiple objects.

urbiscript
Session

```
var c = 1;
[00000000] 1
setSlot("d", getSlot("c"));
[00000001] Slot_0x42340648
c=2;
[00000002] 2
assert
{
    // The same slot is reached through 'c' and 'd'
    d === c
};
```

21.4.3.2 Properties

Properties are stored as slots of slots, so `obj->prop` is equivalent to `getSlot(obj).prop`.

21.4.4 Other features

All the features offered by slot are listed in the documentation section dedicated to the [Section 22.61 Object](#).

21.4.5 Prototypes

21.4.5.1 Manipulation

urbiscript is a prototype-based language, unlike most classical object oriented languages, 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 `Object.protos` method; they can be added or removed with `Object.addProto` and `Object.removeProto`. See [Section 22.41](#) for more documentation.

```
var ob = Object.new()|
assert (ob.protos == [Object]);| urbiscript
| Session
| ob.addProto(Pair);| [00000000] (nil, nil)
| assert (ob.protos == [(nil, nil), Object]);| 
| ob.removeProto(Object);| [00000000] (nil, nil)
| assert (ob.protos == [(nil, nil)]);| 
```

21.4.5.2 Inheritance

Objects inherit their prototypes' slots: `Object.getSlot` will also look in an object prototypes' slots. `Object.getSlot` performs a depth-first traversal of the prototypes hierarchy to find slots. That is, when looking for a slot in an object:

- `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 `Object.addProto` inserts in the front of the prototype list, the last prototype added has priority). This search is recursive: `Object.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.

The following example shows how slots are inherited.

```
urbiscript
Session
var a = Object.new()|
var b = Object.new()|
var c = Object.new()|
a.setSlotValue("x", "slot in a")|
b.setSlotValue("x", "slot in b")|
// c has no "x" slot
c.getSlotValue("x");
[00000000:error] !!! lookup failed: x
// c can inherit the "x" slot from a.
c.addProto(a)|
c.getSlotValue("x");
[00000000] "slot in a"
// b is prepended to the prototype list, and has thus priority.
c.addProto(b)|
c.getSlotValue("x");
[00000000] "slot in b"
// A local slot in c has priority over prototypes.
c.setSlotValue("x", "slot in c")|
c.getSlotValue("x");
[00000000] "slot in c"
```

21.4.5.3 Copy on write

The `Object.updateSlot` method has a particular behavior with respect to prototypes. Although performing an `Object.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.

urbiscript
Session

```
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
```

This behavior can be inhibited by setting the `copyOnWrite` property of a slot to false.

21.4.6 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. 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 does not send the message, it just returns the content of 'msg'.
obj.msg;
```

urbiscript
Session

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, a 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
```

urbiscript
Session

21.5 Enumeration types

Enumeration types enable to create types represented by a finite set of values, like the `enum` declaration in C.

urbiscript
Session

```
enum Suit
{
    hearts,
    diamonds,
    clubs,
    spades, // Last comma is optional
};

[00000001] Suit
```

Since everything is an object in urbiscript, enums are too, with [Enumeration](#) as prototype.

urbiscript
Session

```
Suit;
[00000001] Suit
Suit.protos;
[00000002] [Enumeration]
```

The possible enum values are stored inside the enum object. They inherit the enum object, so you can easily test whether an object is a Suit or not.

urbiscript
Session

```
Suit.hearts;
[00000001] hearts
Suit.diamonds;
[00000002] diamonds
Suit.clubs.isA(Suit);
[00000003] true
42.isA(Suit);
[00000003] false
```

Enumeration values support comparison and pattern matching. You can iterate on the enum object to cycle through all possible values.

urbiscript
Session

```
function find_ace(var suit)
{
    switch (suit)
    {
        case Suit.spades: "The only card I need is";
        default:           "I have";
    }
};

for (var suit in Suit)
    echo("%s the ace of %s." % [find_ace(suit), suit]);
[00000001] *** I have the ace of hearts.
[00000002] *** I have the ace of diamonds.
[00000003] *** I have the ace of clubs.
[00000004] *** The only card I need is the ace of spades.
```

21.6 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)
```

urbiscript
Session

21.6.1 Basic Pattern Matching

Matching is used in many locations and allows to match literal values (e.g., `List`, `Tuple`, `Dictionary`, `Float`, `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"]
```

urbiscript
Session

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"
```

urbiscript
Session

21.6.2 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]);
};
```

urbiscript
Session

```
[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
```

21.6.3 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` where a `Dictionary` is used to set the trajectory type.

```
urbiscript
Session
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.

```
urbiscript
Session
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.

```
urbiscript
Session
{
  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
```

21.7 Imperative flow control

This section specifies the traditional constructs that control the control flow for typical imperative features (loops, scopes etc.). More specific constructs are described elsewhere:

- [Section 21.10](#) defines the control flow constructs for concurrency, including the concurrency-based alternates of loop constructs;
- [Section 21.11](#) specifies the event-handling constructs.

See [Table 21.14](#).

21.7.1 `break`

When encountered within a loop (`every`, `for`, `loop`, `while`), `break` makes the execution jump after the loop.

```
var i = 1;
for (; true; echo(i))
{
    if (i > 8)
        break;
    ++i;
};
[00000000] *** 6
[00000000] *** 7
[00000000] *** 8
[00000000] *** 9
```

urbiscript
Session

```
var i = 0;
while (i < 10)
{
    echo(i);
    if (i == 2)
        break;
    ++i;
};
echo(i);
[00000004] *** 0
[00000005] *** 1
[00000006] *** 2
[00000007] *** 2
```

urbiscript
Session

Occurrences of `break` outside any loop is a syntax error.

```
break;
[00000011:error] !!! syntax error: 'break' not within a loop
```

urbiscript
Session

It is also invalid within function bodies inside loops.

```
for (var i : 10)
{
    function f() { break };
    f;
};
[00000011:error] !!! syntax error: 'break' not within a loop
```

21.7.2 `continue`

When encountered in a loop (`every`, `for`, `loop`, `while`), `continue` short-circuits the remainder of the body of the loop, and runs the next iteration (if there remains one).

urbiscript
Session

```
for (var i = 0; i < 8; i++)
{
  if (i % 2 != 0)
    continue;
  echo(i);
};

[00000000] *** 0
[00000002] *** 2
[00000004] *** 4
[00000006] *** 6
```

urbiscript
Session

```
var i = 0;
while (i < 6)
{
  ++i;
  if (i % 2 != 0)
    continue;
  echo(i);
};

echo(i);
[00000002] *** 2
[00000004] *** 4
[00000006] *** 6
[00000008] *** 6
```

Occurrences of `continue` outside any loop is a syntax error.

urbiscript
Session

```
continue;
[00000011:error] !!! syntax error: 'continue' not within a loop
```

It is also invalid within function bodies inside loops.

urbiscript
Session

```
for (var i : 10)
{
  function f() { continue };
  f;
};

[00000011:error] !!! syntax error: 'continue' not within a loop
```

21.7.3 do

The `do` construct changes the target (`this`) when evaluating an expression. In some programming languages it is also named `with`. It is a convenient means to avoid repeating the same target several times.

It evaluates the `block`, with `this` denoting the value of the `expression`, 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
```

urbiscript
Session

21.7.4 for

There are several kinds of `for` loops:

C-like for such as `for (var i = 0; i < n; ++i) echo(i);`, see [Section 21.7.5](#).

Range-for also known as *foreach*-loops, `for (var i : [0, 1, 2, 4]) echo(i);`, see Section 21.7.6.

Anonymous range-for for instance `for (10) echo(10);`, see Section 21.7.7.

Besides each kind of `for` support several flavors; see Table 21.14.

21.7.5 C-like `for`

urbiscript support the classical C-like `for` construct.

It has the exact same behavior as C's `for`. Using the following names:

```
for (initialization; condition; increment)
  body
```

urbiscript
Session

1. The *initialization* is evaluated.
2. *condition* is evaluated. If the result is false, execution jumps 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`.

There are two (sequential) flavors: `for;` (the default) and `for|`.

21.7.5.1 `for;`

This is the default flavor. Basically its semantics is that of:

```
initialization|
condition|      // break if done
body;

increment|
condition|      // break if done
body;

increment|
// etc.
```

urbiscript
Session

```
var i = 42|;
for; (i = 0; i < 3; i += 1)
  echo(i);
echo(i);
[00000001] *** 0
[00000002] *** 1
[00000003] *** 2
[00000004] *** 3
```

urbiscript
Session

If the *initialization* declares a variable, then this variable is scoped to the loop itself.

```
var i = 42|;
for; (var i = 0; i < 3; i += 1)
  echo(i);
assert (i == 42);
[00000005] *** 0
[00000006] *** 1
[00000007] *** 2
```

urbiscript
Session

21.7.5.2 `for|`

Compared to the ‘;’ flavor, the statements are executed as if they were separated with ‘|’. This flavor is therefore more efficient, at the expense of being selfish: no other piece of code may interleave.

urbiscript
Session

```
initialization|
condition|      // break if done
body|
```



```
increment|
condition|      // break if done
body|
```



```
increment|
// etc.
```

21.7.6 Range-`for`

urbiscript supports iteration over a collection with another form of the `for` loop. This is called *foreach loops* in some programming languages.

There are two (sequential) flavors: `for;` (the default) and `for|`.

21.7.6.1 Range-`for;`

urbiscript
Session

```
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:

urbiscript
Session

```
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.
```

urbiscript
Session

21.7.6.2 Range-for|

Using this flavor, no other part of the system is allowed to run while the loop is executed. It is more efficient, but selfish.

21.7.7 Anonymous range-for

Iterations over a range can be performed without having to provide a variable name to iterate over the collection. 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;` (default), `for|`, and `for&`.

The loop evaluates to `void`.

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; r};
3 == { var r = 0; for ("123")      ++r; r};
```

Assertion Block

21.7.7.1 Anonymous Range-for;

```
{
  var res = [];
  for; (3) res << 1, for; (3) res << 2 ;
  res
}
== [1, 2, 1, 2, 1, 2];
```

Assertion Block

21.7.7.2 Anonymous Range-for|;

```
{
  var res = [];
  for| (3) res << 1, for| (3) res << 2 ;
  res
}
== [1, 1, 1, 2, 2, 2];
```

Assertion Block

21.7.8 if

As in most programming languages, conditionals are expressed with `if`.

First the condition (*statements*) is evaluated; if it evaluates to a value which is true (Section 22.3.1), evaluate the *then-clause* (the first *statements*), otherwise, if applicable, evaluate *else-clause* (the optional statement introduced by `else`).

urbiscript
Session

```
if (true) assert(true) else assert(false);
if (false) assert(false) else assert(true);
if (true) assert(true);
```

Assertion Block

Beware that contrary to most programming languages *there must not be a terminator after the then-clause*:

urbiscript
Session

```
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:

urbiscript
Session

```
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.

Assertion
Block

```
{if (false) 10 else 20} == 20;
{if (true) 10 else 20} == 10;

{if (true) 10 } == 10;

{if (var x = 10) x + 2 else x - 2} == 12;
{if (var x = 0) x + 2 else x - 2} == -2;

{if (var xx = 123) xx | xx};
[00000005:error] !!! lookup failed: xx
```

21.7.9 loop

Endless loops can be created with `loop`, which is equivalent to `while (true)`. The loop evaluates to `void`.

There are two flavors: `loop;` (default) and `loop|`.

21.7.9.1 loop;

Assertion
Block

```
{
  var n = 10|;
  var res = []|;
  loop;
  {
    n--;
    res << n;
    if (n == 0)
      break
  };
  res
}
==
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0];
```

21.7.9.2 loop|

```
{
  var n = 10|;
  var res = []|;
  loop|
  {
    n--;
    res << n;
    if (n == 0)
      break
  };
  res
}
```

Assertion
Block

```
==  
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0];
```

21.7.10 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;  
};
```

urbiscript
Session

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 is 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. Actually, the comparison process is richer than simply using `==`: it support pattern-matching (Section 21.6).

```
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"
```

urbiscript
Session

21.7.11 while

The `while` loop is similar to C's one.

```
while (condition)  
    body;
```

urbiscript
Session

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;`.

21.7.11.1 `while;`

The semantics of:

urbiscript
Session

<code>while; (condition)</code>
<code>body;</code>

is the same as

urbiscript
Session

<code>condition body ; condition body ; ...</code>
--

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.

urbiscript
Session

<code>var i = 4 ;</code>
<code>while (true)</code>
<code>{</code>
<code> --i;</code>
<code> echo("in: " + i);</code>
<code> if (i == 1)</code>
<code> break</code>
<code> else if (i == 2)</code>
<code> continue;</code>
<code> echo("out: " + i);</code>
<code>};</code>
<code>[00000000] *** in: 3</code>
<code>[00000000] *** out: 3</code>
<code>[00000000] *** in: 2</code>
<code>[00000000] *** in: 1</code>

21.7.11.2 `while|`

The semantics of:

<code>while (condition)</code>
<code>body;</code>

urbiscript
Session

is the same as

<code>condition body condition body ...</code>
--

urbiscript
Session

The execution is can be controlled by `break` and `continue`.

<code>var i = 4 ;</code>
<code>while (true)</code>
<code>{</code>
<code> --i;</code>
<code> echo("in: " + i);</code>
<code> if (i == 1)</code>
<code> break</code>
<code> else if (i == 2)</code>
<code> continue;</code>

urbiscript
Session

```

    echo("out: " + i);
};

[00000000] *** in: 3
[00000000] *** out: 3
[00000000] *** in: 2
[00000000] *** in: 1

```

21.8 Exceptions

21.8.1 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: inner
[00000000:error] !!!      called from: outer

```

urbiscript
Session

21.8.2 Catching exceptions

Exceptions are *caught* with the `try/catch` construct. Its syntax is as follows:

It consists of a first block of statements (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++). It is a syntax error if this catch-all clause is followed by a catch-clause with a pattern:

```

try {} catch {} catch (var e) {};
[00000701:error] !!! syntax error: catch: exception already caught by a previous clause

```

urbiscript
Session

The catch-all clause, if present, must be last:

```

try {} catch (var e) {} catch {};

```

urbiscript
Session

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 exception 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

```

urbiscript
Session

```
[00002143:error] !!!    called from: test
```

If an `else`-clause is specified, it is executed if the `try` block did not raise an exception.

urbiscript
Session

```
try { echo("try") }
catch { echo("catch") }
else { echo("else") };
[00002855] *** try
[00002855] *** else

try { echo("try"); echo("throw"); throw 0 }
catch { echo("catch") }
else { echo("else") };
[00002855] *** try
[00002855] *** throw
[00002855] *** catch
```

The value of the whole construct is:

- if the `try` block raised an exception
 - if the exception is not caught (or an exception is thrown from the catch-clause), then there is no value, as the control flow is broken;
 - if the exception is caught; then it's the value of the corresponding catch clause.

urbiscript
Session

```
try { throw 0; "try" } catch (0) { "catch" } else { "else" };
[00467080] "catch"
```

- if the `try` block finish properly, then

- if there is an `else`-clause, its value.

urbiscript
Session

```
try { "try" } catch (0) { "catch" } else { "else" };
[00467080] "else"
```

- otherwise the value of the `try`-block.

```
try { "try" } catch (0) { "catch" };
[00467080] "try"
```

urbiscript
Session

21.8.3 Inspecting exceptions

An `Exception` is a regular object, on which introspection can be performed.

urbiscript
Session

```
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
```

21.8.4 Finally

Using the finally-clause construct, you can ensure some code is executed upon exiting a try-clause, be it naturally or through an exception, `return`, `continue`, ...

21.8.4.1 Regular execution

The finally-clause is executed when the try-clause exits normally.

```
try
{
    echo("inside");
}
finally
{
    echo("finally");
};
[00000001] *** inside
[00000002] *** finally
```

urbiscript
Session

Because it is meant to be used to reclaim resources (“clean up”, the finally-clause never contributes to the value of the whole statement.

```
try { "try" } catch { "catch" } else { "else" } finally { "finally" };
[00000001] "else"
try { throw "throw" } catch { "catch" } else { "else" } finally { "finally" };
[00000001] "catch"
```

urbiscript
Session

21.8.4.2 Control-flow

The finally clause is executed even if `return` is run.

```
function with_return(var enable)
{
    try
    {
        echo("before return");
        if (enable)
            return;
        echo("after return");
    }
    finally
    {
        echo("finally");
    };
    echo("after try-block")
}

with_return(false);
[00001983] *** before return
[00001985] *** after return
[00001985] *** finally
[00001986] *** after try-block

with_return(true);
[00001991] *** before return
[00001992] *** finally
```

urbiscript
Session

It is also the case when the control flow is disrupted by `continue` or `break`.

```
for (var i : ["1", "continue", "2", "break", "3"])
    try
    {
        echo("before: " + i);
```

urbiscript
Session

```

switch (i)
{
    case "break": break;
    case "continue": continue;
};
echo("after: " + i);
}
finally
{
    echo("finally: " + i);
};
[00000663] *** before: 1
[00000671] *** after: 1
[00000671] *** finally: 1
[00000673] *** before: continue
[00000675] *** finally: continue
[00000682] *** before: 2
[00000703] *** after: 2
[00000703] *** finally: 2
[00000704] *** before: break
[00000705] *** finally: break

```

21.8.4.3 Tags

The finally close cannot be interrupted from outside by [Tags](#):

urbiscript
Session

```

var tag = Tag.new()|;
detach({
    tag: {
        try {
            echo("starting"); sleep(1s); echo("ending")
        }
        finally {
            echo("finally one");
            sleep(1s);
            echo("finally two");
        };
        echo("out of finally");
    }
})|;
[00000001] *** starting
tag.stop();
[00000001] *** finally one
tag.stop();
sleep(500ms);
tag.stop();
[00000001] *** finally two
sleep(800ms);

```

No matter how many time you stop `tag`, the finally-clause will not be interrupted. The stop request is still recorded though, so stopping occurs right at the end of the finally-close, which is what 'out of finally' is never displayed.

21.8.4.4 Exceptions

Exceptions caught in the try-catch clause are much like a regular execution flow. In particular, the value of the construct is that of the try-catch clause regardless of the execution of the `finally` clause.

urbiscript
Session

```

try      { echo("try");      "try" }
catch (var e) { echo("catch");  "catch" }
finally   { echo("finally"); "finally" };

```

```
[00000614] *** try
[00000615] *** finally
[00000616] "try"

try { echo("try");      "try" }
catch (var e) { echo("catch");   "catch" }
else { echo("else");     "else" }
finally { echo("finally"); "finally" };

[00000614] *** try
[00000615] *** else
[00000615] *** finally
[00000616] "else"

try { echo("throw 42"); throw 42; "try" }
catch (var e if e == 42) { echo("caught " + e);      "catch" }
finally { echo("finally");      "finally" };

[00000626] *** throw 42
[00000626] *** caught 42
[00000631] *** finally
[00000631] "catch"
```

Uncaught exceptions (i.e., exceptions for which there were no handlers) are propagated after the exception of the finally-clause.

urbiscript
Session

```
try { echo("throw"); throw 51; "try" }
catch (var e if e == 42) { echo("caught " + e);      "catch" }
finally { echo("finally");      "finally" };

[00000616] *** throw
[00000617] *** finally
[00000625:error] !!! 51
```

Exceptions launched in the finally-clause override previous exceptions.

```
try { throw "throw" }
catch { throw "catch" }
finally { throw "finally" };
[00005200:error] !!! finally
```

urbiscript
Session

21.9 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](#).

The `assert` features two constructs: with a function-like syntax, which is adequate for single claims, and a block-like syntax, to group claims together.

21.9.1 Asserting an Expression

```
assert (true);
assert (42);
```

urbiscript
Session

Failed assertions are displayed in a user friendly fashion: first the assertion is displayed before evaluation, then the effective values are reported.

urbiscript
Session

```
function fail () { false }|;
assert (fail);
[00010239:error] !!! failed assertion: fail (fail == false)

function lazyFail { call.evalArgAt(0); false }|;
```

```
assert (lazyFail(1+2, "+* 2));
[00010241:error] !!! failed assertion: lazyFail(1.'+'(2), "+'.'*(2))\
(lazyFail(3, ?) == false)
```

The following example is more realistic.

```
urbiscript
Session
function areEqual(var args[])
{
    var res = true;
    if (!args.empty)
    {
        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) == false)
assert (areEqual(1*2, 1+2));
[00001393:error] !!! failed assertion: areEqual(1.'*'2, 1.'+'(2))\
(areEqual(2, 3) == false)
```

Comparison operators are recognized, and displayed specially:

```
urbiscript
Session
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), them the message is unlikely to make sense.

21.9.2 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.

```
urbiscript
Session
assert
{
    true;
    42;
    1 == 1 + 1;
};
[00000002:error] !!! failed assertion: 1 == 1.'+'(1) (1 != 2)
```

Variable declarations are valid in assertion blocks, in which case their scope is, of course, limited to the block. If the variable is initialized, the initial value is *not* asserted. Contrast the following two similar examples.

```
assert
{
    var x = 0;
    ! x;
};

assert
{
    var x;
```

urbiscript
Session

```

x = 0;
! x;
};

[00000576:error] !!! failed assertion: x = 0

```

For sake of readability and compactness, this documentation shows assertion blocks as follows (see [Chapter 29](#) for a description of the notations).

```

true;
42;
1 == 1 + 1;
[00000002:error] !!! failed assertion: 1 == 1.'+'(1) (1 != 2)

```

Assertion Block

21.10 Parallel Flow Control

This section defines the syntactic constructs that allows concurrent executions. Some of these constructs, loop-like, are concurrent alternative of traditional imperative constructs.

21.10.1 Tagging

To provide control over concurrent jobs, urbiscript uses [Tags](#). To put some statement under the control of a tag, the syntax is as follows:

The *expression* that controls the *statement* must evaluate to a [Tag](#).

```

function id(var i) { echo(i) | i }|;
id(1):id(2);
[00002029] *** 1
[00002029:error] !!! unexpected 1, expected a Tag

```

urbiscript Session

It is always evaluated before the *statement* itself. The value of a tagged statement is that of the *statement*.

```

id(Tag.new()):id(2);
[00015631] *** Tag<tag_25>
[00015631] *** 2
[00015631] 2

```

urbiscript Session

Since job control is especially useful in interactive sessions, urbiscript provides *implicit tags*: unknown single identifiers auto-instantiate a fresh [Tag](#) that will be stored in package.lang.

urbiscript Session

```

// t does not exist.
t.stop();
[00000001:error] !!! lookup failed: t
t: 1;
[00015631] 1
// t is created. It is part of Tag.tags.
assert
{
  t === 'package'.lang.t
};

// Composite Tags cannot be created this way.
t.t2: echo(2);
[00015631:error] !!! lookup failed: t2

// If "this" is not a Lobby, then implicit tags are not supported.
class Foo { function f() { t1: echo(12) } }|;
Foo.f();
[00015631:error] !!! lookup failed: t1
[00015631:error] !!!      called from: f

```

See the documentation of [Tag](#) for more information about the job control. See [Section 11.3](#) for a tutorial on tags.

21.10.2 every

The `every` statement enables to execute a block of code repeatedly, with the given period.

urbiscript
Session

```
// 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?
```

The whole `every` statement itself remains in foreground: following statements separated with ; or | will not be reached unless you `break` out of it. You may use `continue` to finish one iteration; the following iteration is then launched as expected by the given period, *not* immediately.

urbiscript
Session

```
var count = 4|;
var start = time|;
echo("before");
every (1s)
{
  --count;
  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
```

Exceptions thrown in the body interrupt the loop:

urbiscript
Session

```
var t = 2|;
every (1s) { echo (2 / t); --t };
[00000001] *** 1
[00000002] *** 2
[00000003:error] !!! /: division by 0
[01234567:error] !!!      called from: []
```

It exists in several flavors, with the same syntax:

21.10.2.1 every;

In this flavor, several instances of the body may overlap if the duration of their execution is longer than the period:

```
var x = 0|;
timeout (990ms)
  every (125ms)
  {
    x++;
    sleep(250ms);
  };
assert (x == 8);
```

urbiscript
Session

Contrast with `every|`, Section 21.10.2.2.

21.10.2.2 every|

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. Contrast the following example with that of `every`, Section 21.10.2.1.

```
var x = 0|;
timeout (990ms)
every| (125ms)
{
    x++;
    sleep(250ms);
};
assert (x == 4);
```

urbiscript
Session

The flow-control constructs `break` and `continue` are supported.

```
var count = 0|;
every| (250ms)
{
    ++count;
    if (count == 2)
        continue;
    if (count == 4)
        break;
    echo(count);
};
[00000000] *** 1
[00001500] *** 3
```

urbiscript
Session

21.10.2.3 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
```

urbiscript
Session

21.10.3 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 21.10.10.

```
for, (var i = 3; 0 < i; --i)
{
    var j = i |
    echo("in: i = %s, j = %s" % [i, j]);
```

urbiscript
Session

```

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

```

urbiscript
Session

```

for, (var i = 9; 0 < i; --i)
{
    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

```

21.10.4 Range-for&

One can iterate concurrently over the members of a collection.

urbiscript
Session

```

for& (var i: [0, 1, 2])
{
    echo(i * i);
    echo(i * i);
};

[00000000] *** 0
[00000000] *** 1
[00000000] *** 4
[00000000] *** 0
[00000000] *** 1
[00000000] *** 4

```

As all the loops, the loop's value is `void`.

If an iteration executes `continue`, it is stopped; the other iterations are not affected.

urbiscript
Session

```

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)
    {

```

urbiscript
Session

```

    echo("break");
    break;
};

sleep(1s);
echo(j);
};

[00000001] *** 0
[00000001] *** 1
[00000001] *** 2
[00000002] *** break

```

21.10.5 Anonymous range-for&

Since `for& (n) body` is processed as `for& (var tmp: n) body`, with `tmp` being a hidden variable, see [Section 21.10.4](#) for details.

21.10.6 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];

```

Assertion Block

21.10.7 timeout

The `timeout` construct allows to interrupt a piece of code that took too long to run. Its syntax is based on the `try/catch` construct ([Section 21.8.2](#)).

If the statement finishes its expiration before `duration` expires, then the optional `else`-clause is run. Otherwise, if `duration` expires, the optional `catch`-clause is run. In both cases, then the optional `finally`-clause is executed.

The value of the `timeout`-statement is that of the last completed non-`finally` block that was run; `void` if no block was completed. The value of the `finally` is always ignored, because this clause is meant to release resources, and in typical uses it does not contribute to the computation.

Because there are too many cases to demonstrate, only some of them are exemplified below.

```

// timeout alone.
timeout (1s) { echo("body"); sleep(0.5s); "body-value" };
[00000002] *** body
[00000004] "body-value"

timeout (1s) { echo("body"); sleep(2s); "body-value" };

```

urbiscript Session

```
[00000002] *** body

// timeout and catch.
timeout (1s) { echo("body"); sleep(0.5s); "body-value" }
catch       { echo("catch"); "catch-value" };
[00000002] *** body
[00000002] "body-value"

timeout (1s) { echo("body"); sleep(2s); "body-value" }
catch       { echo("catch"); "catch-value" };
[00000002] *** body
[00000002] *** catch
[00000002] "catch-value"

// timeout and else.
timeout (1s) { echo("body"); sleep(0.5s); "body-value" }
else        { echo("else"); "else-value" };
[00000002] *** body
[00000002] *** else
[00000002] "else-value"

timeout (1s) { echo("body"); sleep(2s); "body-value" }
else        { echo("else"); "else-value" };
[00000002] *** body

// timeout, catch, and else.
timeout (1s) { echo("body"); sleep(0.5s); "body-value" }
catch       { echo("catch"); "catch-value" }
else        { echo("else"); "else-value" };
[00000002] *** body
[00000002] *** else
[00000002] "else-value"

timeout (1s) { echo("body"); sleep(2s); "body-value" }
catch       { echo("catch"); "catch-value" }
else        { echo("else"); "else-value" };
[00000002] *** body
[00000002] *** catch
[00000002] "catch-value"

// timeout, catch, else and finally.
timeout (1s) { echo("body"); sleep(0.5s); "body-value" }
catch       { echo("catch"); "catch-value" }
else        { echo("else"); "else-value" }
finally     { echo("finally"); "finally-value" };
[00000002] *** body
[00000002] *** else
[00000002] *** finally
[00000002] "else-value"

timeout (1s) { echo("body"); sleep(2s); "body-value" }
catch       { echo("catch"); "catch-value" }
else        { echo("else"); "else-value" }
finally     { echo("finally"); "finally-value" };
[00000002] *** body
[00000002] *** catch
[00000002] *** finally
[00000002] "catch-value"
```

If an exception is thrown from the `timeout`-clause, it is *not* caught by the `catch`-clause (nor the `else`-clause), rather it continues its propagation to the outer statements.

```
timeout (1s) { echo("body"); throw Exception.new("Ouch") }
catch       { echo("catch"); "catch-value" }
else        { echo("else"); "else-value" }
```

urbscript
Session

```
finally      { echo("finally"); "finally-value" };
[00000566] *** body
[00000569] *** finally
[00000585:error] !!! Ouch
```

Alternatively, you may use the [Timeout](#) object.

21.10.8 stopif

urbiscript
Session

```
stopif(condition) { code};
```

The [stopif](#) constructs interrupts its code block argument when the condition becomes true. It is syntactic sugar for:

urbiscript
Session

```
var t = Tag.new();
detach({ waituntil(condition) | t.stop()});
t: { code};
```

The condition can be an arbitrary expression, or an event match.

urbiscript
Session

```
var e = Event.new();
stopif(e?) every(500ms) echo("running"),
sleep(1.2s); e!; echo("stopped");
[00000001] *** running
[00000002] *** running
[00000002] *** running
[00000003] *** stopped
```

21.10.9 freezeif

The [freezeif](#) construct is similar to [stopif](#), but freeze/unfreeze the code argument when the condition is true/false.

urbiscript
Session

```
var b = false;
timeout(3.2s) detach({
  freezeif(b) every(500ms) echo("tick"),
  freezeif(!b) every(500ms) echo("tack")
})|;
sleep(1.2s); b = true;
[00000001] *** tick
[00000001] *** tick
[00000001] *** tick
[00000001] true
sleep(1s); b = false;
[00000001] *** tack
[00000001] *** tack
[00000001] *** tack
[00000001] false
sleep(1s); echo("done");
[00000001] *** tick
[00000001] *** tick
[00000001] *** done
```

The condition can be an arbitrary expression, or an event match.

21.10.10 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

urbiscript
Session

```
while, (condition)
  body;
```

is the same as

urbiscript
Session

```
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:

urbiscript
Session

```
var i = 4!;
while, (i)
{
  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 = 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.

urbiscript
Session

```
var i = 10!;
while, (i)
{
  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
```

21.11 Event Handling

21.11.1 at

Using the `at` construct, one can arm code that will be triggered each time some condition is true.

There are two different kinds of `at` statements, depending on the type of the “condition”:

- `e?(args)` to catch when events are sent (Section 21.11.1.1),

- `exp` to catch each time a Boolean `exp` becomes true ([Section 21.11.1.2](#)).

The `onleave`-clause 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.

Event-programming constructs actually set up monitors that become “back-ground jobs”. As a result:

- they return void;
- they escape the traditional flow-control evaluation scheme. Use `Tags` to control them ([Section 21.11.1.5](#)).

21.11.1.1 at on Events

See [Section 12.2](#) for an example of using `at` statements to watch events.

Be cautious not to forget the question-mark, as when it is forgotten it means that you mean to monitor changes to `e`, not its activations. In the following example `e` is “true”, so the `at` statement fires immediately, but remains still on later activations of the event.

```
var e = Event.new()!;
at (e) echo("w00t");
[00000002:warning] !!! at (<event>) without a '?', probably not what you mean
[00000003] *** w00t

e!;
e!;
```

urbiscript
Session

Durations Since events may last for a given duration (`e! ~ duration`), event handlers may also require an event to be sustained for a given amount of time before being “accepted” (`at (e? ~ duration)`).

```
var e = Event.new()!;
at (e?(var start) ~ 1s)
  echo("in : %s" % (time - start).round())
onleave
  echo("out: %s" % (time - start).round());

// This emission is too short to trigger the at.
e!(time);

// This one is long enough.
// The body triggers is after the emission started.
e!(time) ~ 2s;
[00001000] *** in : 1
[00002000] *** out: 2
```

urbiscript
Session

21.11.1.2 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);
```

urbiscript
Session

It can also wait for some condition to hold long enough: $\text{exp} \sim \text{duration}$, as a condition, denotes the fact that exp was true for duration seconds.

```
urbiscript
Session
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);
```

Your expression inside the `at` should not have any side effect, or use any code that will delay execution (for instance `sleep`, `waituntil`...).

Furthermore, `at` will not work properly on expressions invoking user-defined C++ functions, since `at` has no way to track when those change.

```
urbiscript
Session
// Do not do this, when and how many times the at condition will be evaluated
// is not specified.
at( x++ | y==2) {};
// Avoid this also, result is unspecified.
at (sleep(1s) | y==2) {};
```

21.11.1.3 Synchronous and asynchronous `at`

By default, `at` is asynchronous: the enter and leave actions are executed in detached jobs and won't interfere with the execution flow of the job that triggered it.

```
urbiscript
Session
var e = Event.new();
[00000001] Event_0x42389008

at (e?)
{
  sleep(1s);
  echo("in");
}
onleave
{
  sleep(2s);
  echo("out");
};

e!;
// Actions are triggered in the background and won't block
// the execution flow.
sleep(500ms);
echo("Not blocked");
[00000002] *** Not blocked
sleep(1s);
[00000003] *** in
echo("Not blocked");
[00000004] *** Not blocked
sleep(500ms);
[00000003] *** out
```

When using the `sync` keyword after `at`, it becomes synchronous: when a job triggers it, all enter and leave actions are executed synchronously before the triggering statement returns.

```
var e = Event.new();
[00000001] Event_0x42389008

at sync (e?)
{
    sleep(1s);
    echo("in");
}
onleave
{
    sleep(1s);
    echo("out");
};

e!;
// Actions are triggered synchronously, the next line will be executed
// when they're done.
echo("Blocked");
[00000002] *** in
[00000003] *** out
[00000004] *** Blocked
```

urbiscript
Session

Alternatively, the event emission may request a synchronous handling, see [Section 22.15.2](#).

21.11.1.4 Execution Context

When a body of an `at`-construct is executed (be it the enter- or the leave-clause), its context is that of the whole construct.

Context: `this` The current object, `this`, when executing a sub-clause, is the current object when the `at`-construct was evaluated.

```
class Global.Foo
{
    function event()
    {
        var res = Event.new();
        at sync (res?) { assert(this === Global.Foo) | echo(1) }
        onleave       { assert(this === Global.Foo) | echo(2) };

        at (res?)     { assert(this === Global.Foo) | echo(3) }
        onleave       { assert(this === Global.Foo) | echo(4) };
        res;
    };
}!;

class Global.Bar
{
    var event = Global.Foo.event();
}!;

Global.Bar.event!;
```

urbiscript
Session

Context: Tags The tags that apply to the execution of enter- and leave-clauses include that of the whole `at`-construct:

urbiscript
Session

```

var t = Tag.new("t")|;
var e = Event.new()|;
t: at sync (e?) { assert(t in Job.current.tags) | echo(1) }
    onleave { assert(t in Job.current.tags) | echo(2) };

t: at (e?) { assert(t in Job.current.tags) | echo(3) }
    onleave { assert(t in Job.current.tags) | echo(4) };
e!;

[00004898] *** 1
[00004898] *** 2
[00004898] *** 3
[00004898] *** 4

```

21.11.1.5 Tags and `at`

Event programming constructs such as `at` install background jobs that monitor the events. They are not submitted to the usual rules of control-flow. To control them, use `Tags`.

Tags allow to disable, temporarily or momentarily, event monitors.

urbiscript
Session

```

var t = Tag.new()|;
var e = Event.new()|;
t: at (e?(var x)) echo("enter" + x) onleave echo("leave" + x);
t: at sync (e?(var x)) echo("syncEnter" + x) onleave echo("syncLeave" + x);
e!(1);

[00001822] *** syncEnter1
[00001822] *** syncLeave1
[00001822] *** enter1
[00001822] *** leave1

// Nothing happens when the code is frozen.
t.freeze();
e!(2);
e!(3);
t.unfreeze();

// Back on line!
e!(4);
[00001843] *** syncEnter4
[00001843] *** syncLeave4
[00001843] *** enter4
[00001843] *** leave4

// Dead, for ever.
t.stop();
e!(5);

```

Tags not only control the monitor part (the part checking whether the event is triggered), but also the execution of the body. And Tags can be used from the event-handler itself.

urbiscript
Session

```

var t = Tag.new()|;
var e = Event.new()|;
t: at (e?(var x))
{
    echo(x+1) | t.stop() | echo(x+2)
}
onleave
{
    echo(x+3)
};
e!("a");
[00001843] *** a1
e!("b");

```

The same applies when the monitor is interrupted from the `onleave` clause.

urbiscript
Session

```
var t = Tag.new()!;
var e = Event.new()!;
t: at (e?(var x))
{
    echo(x+1)
}
onleave
{
    echo(x+2) | t.stop() | echo(x+3)
};
e!("a");
[00001843] *** a1
[00001843] *** a2
e!("b");
```

21.11.1.6 Scoped `at`

`at` statements are not “scoped”: they install event-handlers that escape the scope in which they are defined:

urbiscript
Session

```
function Event.newVerbose()
{
    var res = Event.new();
    at (res?(var x))
        echo(x);
    res
}|;
Event.newVerbose()!(12);
[00001202] *** 12
```

Using a `Tag`, one can control them (Section 21.11.1.5). In the following example, `Tag.scope` is used to label the `at` statement. When the function ends, the `at` is no longer active.

urbiscript
Session

```
function Event.newScoped()
{
    var res = Event.new();
    Tag.scope:
    at (res?(var x))
        echo(x);
    res
}|;
Event.newScoped()!(23);
// Nothing happens.
```

21.11.2 `waituntil`

The `waituntil` construct is used to hold the execution until some condition is verified. Similarly to `at` (Section 21.11.1) and the other event-based constructs, `waituntil` may work on events, or on Boolean expressions.

21.11.2.1 `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.

urbiscript
Session

```
{|
    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”.

urbiscript
Session

```

{
  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.

urbiscript
Session

```

{
  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.

urbiscript
Session

```

{
  var e = Event.new();
  e!;
  {
    waituntil (e?);
    echo("caught e");
  },
  e!;
[00021054] *** caught e
};

```

21.11.2.2 [waituntil](#) on Boolean Expressions

You may use any expression that evaluates to a truth value as argument to [waituntil](#).

```

{
  var foo = Object.new();
  {

```

urbiscript
Session

```

    waituntil (foo.hasLocalSlot("bar"));
    echo(foo.getLocalSlotValue("bar"));
},
var foo.bar = 123|;
};

[00021054] *** 123

```

21.11.3 watch

The `watch` construct is similar in spirit to using the `at` construct to monitor expressions, except it enables you to be notified when an arbitrary expression changed, not only when it becomes true or false. This makes `watch` a more primitive tool than `at` on expressions. Actually, `at` on expressions uses `watch` to determine when to reevaluate its condition.

`watch(expression)` evaluates to an `Event` that triggers every time `expression` changes, with its new value as payload.

```

var x = 0;
[00000000] 0
var y = 0;
[00000000] 0
var e = watch(x + y);
[00000000] Event_0x103a1e978
at (e?(var value))
  echo("x + y = %s" % value);
x = 1;
[00000000] 1
[00000000] *** x + y = 1
y = 2;
[00000000] 2
[00000000] *** x + y = 3

```

urbiscript
Session

Note that “the expression changed” might be ambiguous: Urbi considers the expression to have changed when any component involved in its evaluation changed. If a `Float` is replaced with another `Float` of the same value, the expression has changed, since the new `Float` may have different slots.

```

var x = 0;
[00000000] 0
at (watch(x)?(var value))
  echo("x = %s" % value);
// This is considered as a change, although the new float value is also 0.
x = 0;
[00000000] 0
[00000000] *** x = 0

```

urbiscript
Session

Also, some modification may modify the evaluation, but still yield the same result.

```

var x = 1;
[00000000] 1
at (watch(x % 2)?(var value))
  echo("x %% 2 = %s" % value);

// This is considered as a change, although the computation yields the
// same result.
x = 3;
[00000000] 3
[00000000] *** x % 2 = 1

```

urbiscript
Session

21.11.4 whenever

The `whenever` construct really behaves like a never-ending `loop if` construct. It also works on events and Boolean expressions. `whenever` will execute its statement in a loop while the

condition stays true.

urbiscript
Session

```
whenever (condition)
  statement1
```

It supports an optional `else` clause, which is run in a loop while the condition is false.

urbiscript
Session

```
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.

21.11.4.1 whenever on Events

A `whenever` clause can be used to catch events with or without payloads.

urbiscript
Session

```
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] *** ...
```

urbiscript
Session

If the body of the `whenever` lasts for a long time, it is possible that two executions be run concurrently.

urbiscript
Session

```
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

```

21.11.4.2 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();

```

urbiscript
Session

At all times, the `whenever` is evaluating either the main clause or the else clause, no parallel evaluation of the clauses will occur. The clauses will not be interrupted if the condition switches state. The `whenever` will instead wait for the end of the evaluation:

```

var y = 0|;
var startTime = time()|;
function roundTime(t) { (t*2).round() / 2}|; // Round at 500ms precision
function deltaTime() {roundTime(time()-startTime)}|;
y=1|;
t:whenever(y==1)
{
    echo("y=1 at %s s" % deltaTime());
}

```

urbiscript
Session

```

    sleep(1s);
}
else
{
    echo("y=0 at %s s" % deltaTime());
    sleep(1s);
};

[00000001] *** y=1 at 0 s
[00000001] *** y=1 at 1 s
sleep(1500ms);
y=0|;
[00000001] *** y=0 at 2 s
[00000001] *** y=0 at 3 s
sleep(2000ms);
t.stop();

```

Two special variables are available inside a `whenever`: `\$wheneverOn` and `\$wheneverOff`. They are `Tags` that are stopped when the condition triggers from true to false and from false to true respectively. They can be used to interrupt your clauses:

```

startTime = time()|;
y=1|;
t:whenever(y==1)
{
    echo("y=1 at %s s" % deltaTime());
    '$wheneverOn':sleep(1s);
}
else
{
    echo("y=0 at %s s" % deltaTime());
    '$wheneverOff': sleep(1s);
};
[00000001] *** y=1 at 0 s
[00000001] *** y=1 at 1 s
sleep(1500ms);
y=0|;
[00000001] *** y=0 at 1.5 s
[00000001] *** y=0 at 2.5 s
sleep(1500ms);
t.stop();

```

urbiscript
Session

21.12 Trajectories

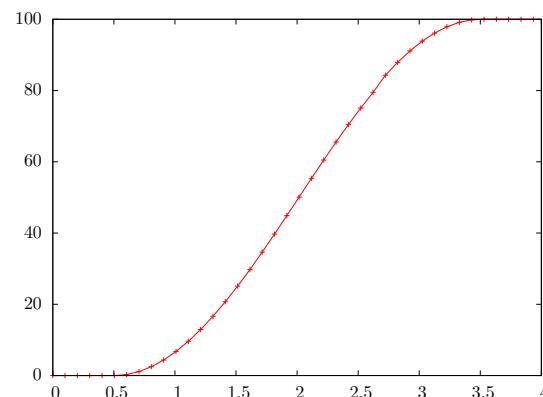
In robotics, *trajectories* are often used: they are a means to change the value of a variable (actually, a slot) over time. This can be done using detached executions, for instance using a combination of `every` and `detach`, but urbiscript provides syntactic sugar to this end.

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).

```

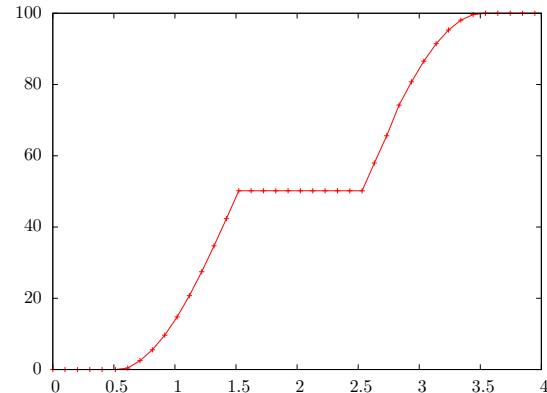
var y = 0;
{
    sleep(0.5s);
    y = 100 smooth:3s,
},

```



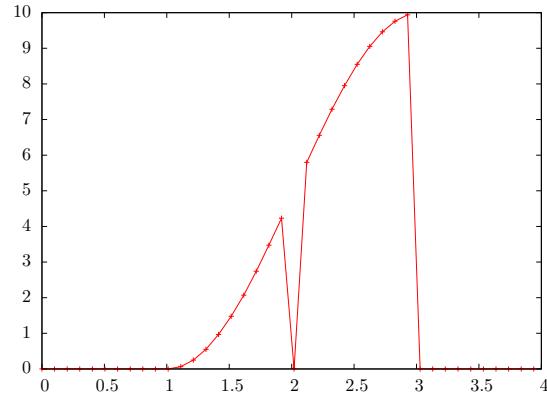
Trajectories can be frozen and unfrozen, using tags (Section 11.3). In that case, “time is suspended”, and the trajectory resumes as if the trajectory was never interrupted.

```
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](#) for the list of supported trajectories.

21.13 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.

urbiscript
Session

```
{
    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.

urbiscript
Session

```
// 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

removeLocalSlot("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.removeLocalSlot("A"); // Break the cycle
removeLocalSlot("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.
```

urbiscript
Session

Keyword	Remark	Keyword	Remark
<code>and</code>	Synonym for <code>&&</code>	<code>and_eq</code>	Synonym for <code>&=</code>
<code>asm</code>	Reserved	<code>assert</code>	Section 21.9
<code>at</code>	Section 21.11.1	<code>auto</code>	Reserved
<code>bitand</code>	Section 21.1.8.4	<code>bitor</code>	Section 21.1.8.4
<code>bool</code>	Reserved	<code>break</code>	Section 21.7.1
<code>call</code>	Section 21.3.3	<code>case</code>	Section 21.7.10
<code>catch</code>	Section 21.8.2, Section 21.10.7	<code>char</code>	Reserved
<code>class</code>	Section 21.1.6.8	<code>closure</code>	Section 21.3.6
<code>compl</code>	Section 21.1.8.4	<code>const</code>	Section 21.4.2
<code>const_cast</code>	Reserved	<code>continue</code>	Section 21.7.2
<code>default</code>	Section 21.7.10	<code>delete</code>	Reserved
<code>detach</code>		<code>disown</code>	
<code>do</code>	Section 21.7.3	<code>double</code>	Reserved
<code>dynamic_cast</code>	Reserved	<code>else</code>	Section 21.7.8
<code>enum</code>	Section 21.7.2	<code>every</code>	Section 21.10.2
<code>explicit</code>	Reserved	<code>export</code>	Reserved
<code>extern</code>	Reserved	<code>external</code>	Reserved
<code>float</code>	Reserved	<code>for</code>	<code>for&</code> and <code>for </code> flavors
<code>foreach</code>	Deprecated, use <code>for</code>	<code>freezeif</code>	
<code>friend</code>	Reserved	<code>function</code>	Section 21.3.1, Section 21.3.6
<code>goto</code>	Reserved	<code>if</code>	Section 21.7.8
<code>in</code>	Section 21.7.6	<code>inline</code>	Reserved
<code>int</code>	Reserved	<code>internal</code>	Deprecated
<code>long</code>	Reserved	<code>loop</code>	<code>loop&, loop </code> , Section 21.7.9
<code>mutable</code>	Reserved	<code>namespace</code>	Reserved
<code>new</code>	Section 8.5	<code>not</code>	Synonym for <code>!</code>
<code>not_eq</code>	Synonym for <code>!=</code>	<code>onleave</code>	Section 21.11.1
<code>or</code>	Synonym for <code> </code>	<code>or_eq</code>	Synonym for <code> =</code>
<code>private</code>	Ignored	<code>protected</code>	Ignored
<code>public</code>	Ignored	<code>register</code>	Reserved
<code>reinterpret_cast</code>	Reserved	<code>return</code>	Section 21.3.3
<code>short</code>	Reserved	<code>signed</code>	Reserved
<code>sizeof</code>	Reserved	<code>static</code>	Deprecated
<code>static_cast</code>	Reserved	<code>stopif</code>	
<code>struct</code>	Reserved	<code>switch</code>	Section 21.7.10
<code>template</code>	Reserved	<code>this</code>	
<code>throw</code>	Section 21.8.1	<code>timeout</code>	Section 21.10.7
<code>try</code>	Section 21.8.2	<code>typedef</code>	Reserved
<code>typeid</code>	Reserved	<code>typename</code>	Reserved
<code>union</code>	Reserved	<code>unsigned</code>	Reserved
<code>using</code>	Reserved	<code>var</code>	Section 21.2.2, Section 21.4.1
<code>virtual</code>	Reserved	<code>volatile</code>	Reserved
<code>waituntil</code>	Section 21.11.2	<code>watch</code>	Section 21.11.3
<code>wchar_t</code>	Reserved	<code>whenever</code>	Section 21.11.4
<code>while</code>	<code>while&</code> and <code>while </code> flavors	<code>xor</code>	Synonym for <code>^</code>
<code>xor_eq</code>	Synonym <code>^=</code>		

Table 21.1: Keywords

unit	abbreviation	equivalence for n
radian	rad	n
degree	deg	$n/180 * \pi$
grad	grad	$n/200 * \pi$

Table 21.2: Angle units

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 21.3: Duration units

\\"	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 21.4: String escapes

Oper.	Syntax	Assoc.	Semantics	Equivalence
**	a ** b	Right	Exponentiation	a.'**'(b)
+	+a	-	Identity	a.'+'()
-	-a	-	Opposite	a.'-'()
*	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 21.5: Arithmetic operators

Oper.	Syntax	Assoc.	Semantics	Equivalence
=	a = b	Right	Assignment	updateSlot("a", b) ²
+=	a += b	Right	In place +	a = a.'+='(b)
-=	a -= b	Right	In place -	a = a.'-='(b)
*=	a *= b	Right	In place *	a = a.'*='(b)
/=	a /= b	Right	In place /	a = a.'/='(b)
%=	a %= b	Right	In place %	a = a.'%='(b)
^=	a ^= b	Right	In place ^	a = a.'^='(b)

Table 21.6: Assignment operators

Oper.	Syntax	Assoc.	Semantics	Equivalence
++	++a	-	Pre-incrementation	a = a.'++'
--	--a	-	Pre-decrementation	a = a.'--'
++	a++	-	Post-incrementation	{var '\$a' = a ++a '\$a'}
--	a--	-	Post-decrementation	{var '\$a' = a --a '\$a'}

Table 21.7: Pre-/postfix operators

Oper.	Syntax	Assoc.	Semantics	Equivalence
compl	compl a	-	Bitwise complement	a.'compl'()
<<	a << b	Left	Left bit shift	a.'<<'(b)
>>	a >> b	Left	Right bit shift	a.'>>'(b)
bitand	a bitand b	Left	Bitwise and	a.'bitand'(b)
^	a ^ b	Left	Bitwise exclusive or	a.'^'(b)
bitor	a bitor b	Left	Bitwise or	a.'bitor'(b)

Table 21.8: Bitwise operators

Oper.	Syntax	Assoc.	Semantics	Equivalence
!	!a	Left	Logical negation	a.'!'()
&&	a&&b	Left	Logical and	if (a) b else a
	a b	Left	Logical or	if (a) a else b

Table 21.9: Boolean operators

Oper.	Syntax	Assoc.	Semantics	Equivalence
==	a == b	Left	Equality	a.'=='(b)
!=	a != b	Left	Inequality	a.'!='](b)
====	a === b	Left	Physical equality	a.'===='](b)
!==	a !== b	Left	Physical inequality	a.'!=='](b)
~=	a ~= b	Left	Relative approximate equality	a.'~='](b)
=~	a =~ b	Left	Absolute approximate equality	a.'=~'](b)
<	a < b	Left	Less than	a.'<'(b)
<=	a <= b	Left	Less than or equal to	a.'<='(b)
>	a > b	Left	Greater than	a.'>'(b)
>=	a >= b	Left	Greater than or equal to	a.'>'](b)

Table 21.10: Comparison operators

Oper.	Syntax	Assoc.	Semantics	Equivalence
[]	a[args]	Left	Subscript	a.'[]'](args)
[] =	a[args] = v	Right	Subscript assignment	a.'[]'='](args, v)
in	a in b	None	Membership	b.has(a)
not in	a not in b	None	Non-membership	b.hasNot(a)

Table 21.11: Container operators

Oper.	Syntax	Assoc.	Semantics	Equivalence
.	a.b	Left	Message sending	Not redefinable
.	a.b(args)	Left	Message sending	Not redefinable
->	a->b	Left	Property access	getProperty("a", "b")
->	a->b = v	Right	Property assignment	setProperty("a", "b", v)
&	&a	-	Slot access	getSlot("a")
.&	a.&b	Left	Slot access	a.getSlot("b")

Table 21.12: Object operators

Oper.	Syntax	Assoc.	Semantics	Equivalence
.	a.b	Left	Message sending	Not redefinable
.	a.b(args)	Left	Message sending	Not redefinable
->	a->b	Left	Property access	getProperty("a", "b")
->	a->b = v	Right	Property assignment	setProperty("a", "b", v)
&	&a	-	Slot access	getSlot("a")
.&	a.&b	Left	Slot access	a.getSlot("b")
[]	a[args]	Left	Subscript	a.'[]'(args)
[] =	a[args] = v	Right	Subscript assignment	a.'[]='(args, v)
**	a ** b	Right	Exponentiation	a.'**'(b)
compl	compl a	-	Bitwise complement	a.'compl'()
+	+a	-	Identity	a.'+'()
-	-a	-	Opposite	a.'-'()
!	!a	Left	Logical negation	a.'!'()
*	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)
bitand	a bitand b	Left	Bitwise and	a.'bitand'(b)
^	a ^ b	Left	Bitwise exclusive or	a.'^'(b)
bitor	a bitor b	Left	Bitwise or	a.'bitor'(b)
==	a == b	Left	Equality	a.'=='(b)
!=	a != b	Left	Inequality	a.'!='(b)
====	a === b	Left	Physical equality	a.'===='(b)
!==	a !== b	Left	Physical inequality	a.'!=='(b)
=~	a =~ b	Left	Absolute approximate equality	a.'=~='(b)
~=	a ~= b	Left	Relative approximate equality	a.'~='(b)
<	a < b	Left	Less than	a.'<'(b)
<=	a <= b	Left	Less than or equal to	a.'<='(b)
>	a > b	Left	Greater than	a.'>'(b)
>=	a >= b	Left	Greater than or equal to	a.'>='(b)
in	a in b	None	Membership	b.has(a)
not in	a not in b	None	Non-membership	b.hasNot(a)
&&	a&&b	Left	Logical and	if (a) b else a
	a b	Left	Logical or	if (a) a else b
=	a = b	Right	Assignment	updateSlot("a", b)
+=	a += b	Right	In place +	a = a.'+='(b)
-=	a -= b	Right	In place -	a = a.'--'(b)
*=	a *= b	Right	In place *	a = a.'*='(b)
/=	a /= b	Right	In place /	a = a.'/='(b)
%=	a %= b	Right	In place %	a = a.'%='(b)
^=	a ^= b	Right	In place ^	a = a.'^='(b)
++	++a	-	Pre-incrementation	a = a.'++'
--	--a	-	Pre-decrementation	a = a.'--'
++	a++	-	Post-incrementation	{var '\$a' = a ++a '\$a'}
--	a--	-	Post-decrementation	{var '\$a' = a --a '\$a'}

Table 21.13: Operators sorted by decreasing precedence groups.

	;		,	&
<code>every</code>	Section 21.10.2.1	Section 21.10.2.2	Section 21.10.2.3	
<code>for (i; t; c)</code>	Section 21.7.5.1	Section 21.7.5.2	Section 21.10.3	
<code>for (var i: c)</code>	Section 21.7.6.1	Section 21.7.6.2		Section 21.7.6.2
<code>for (c)</code>	Section 21.7.7.1	Section 21.7.7.2		Section 21.7.7.2
<code>loop</code>	Section 21.7.9.1	Section 21.7.9.2	Section 21.10.6	
<code>while</code>	Section 21.7.11.1	Section 21.7.11.2	Section 21.10.10	

Table 21.14: Keywords with flavors

Chapter 22

urbiscript Standard Library

This chapter details the predefined objects.

22.1 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.

22.1.1 Prototypes

- **Object**

22.1.2 Construction

A **Barrier** can be created with no argument. Calls to **signal** and **wait** done on this instance are restricted to this instance.

urbiscript
Session

```
Barrier.new();
[00000000] Barrier_0x25d2280
```

22.1.3 Slots

- **signal(payload)**

Wake up one of the job waiting for a signal. The *payload* is sent to the **wait** method. Return the number of jobs woken up.

urbiscript
Session

```
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.

urbiscript
Session

```
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.

urbiscript
Session

```
do (Barrier.new())
{
    echo(wait()) &
```

```
    signal(1)
}|;
[00000000] *** 1
```

22.2 Binary

A Binary object, sometimes called a *blob*, is raw memory, decorated with a user defined header.

22.2.1 Prototypes

- `Object`

22.2.2 Construction

Binaries are usually not made by users, but they are heavily used by the internal machinery when exchanging Binary UVValues. A binary features some `content` and some `keywords`, both simple `Strings`.

urbiscript
Session

```
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.

22.2.3 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.

Assertion
Block

```
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.

Assertion
Block

```
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/UVValue protocol. Incoming binaries must use a semicolon to separate the header part from the content, while outgoing binaries use a carriage-return.

urbiscript
Session

```
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 21.1.6.6](#)):

Assertion
Block

```
Binary.new("head", "\B(7)(content)").asString()
== "BIN 7 head\ncontent";
```

- `data`

The data carried by the Binary.

Assertion
Block

```
Binary.new("head", "content").data == "content";
```

- **empty**

Whether the data is empty.

Assertion
Block

```
Binary.new("head", "").empty;  
!Binary.new("head", "content").empty;
```

- **keywords**

The headers carried by the Binary.

Assertion
Block

```
Binary.new("head", "content").keywords == "head";
```

22.3 Boolean

There is no object `Boolean` in urbiscript, but two specific objects `true` and `false`. They are the result of all the comparison statements.

22.3.1 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).`

- `Dictionary`, `List`, `String`
`false iff empty (Dictionary, List, String).`
- otherwise
`true.`

The method `Object.asBool` is in charge of converting some arbitrary value into a Boolean.

urbiscript
Session

```
assert(Global.asBool() == true);
assert(nil.asBool() == false);
void.asBool();
[00000421:error] !!! unexpected void
```

22.3.2 Prototypes

The objects `true` and `false` have the following prototype.

- `Singleton`

22.3.3 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;
```

Assertion
Block

22.3.4 Slots

- `'!'`

Logical negation. If `this` is `false` return `true` and vice-versa.

```
!true === false;
!false === true;
```

Assertion
Block

- `asBool`

Identity.

Assertion
Block

```
true.asBool === true;  
false.asBool === false;
```

22.4 CallMessage

Capturing a method invocation: its target and arguments.

22.4.1 Examples

22.4.1.1 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`.

urbiscript
Session

```
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
```

22.4.1.2 Strict Functions

Strict functions do support `call`.

urbiscript
Session

```
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
```

22.4.2 Prototypes

- `Object`

22.4.3 Slots

- `args`

The list of not yet evaluated arguments.

urbiscript
Session

```
function args { call.args }|
assert
{
    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.

urbiscript
Session

```
function argsCount { call.argsCount }|;
assert
{
    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**.

urbiscript
Session

```
function code { call.getSlotValue("code") }|
assert (code == getSlotValue("code"));
```

- **eval**

Evaluate **this**, and return the result.

urbiscript
Session

```
var c1 = do (CallMessage.new())
{
    var this.target = 1;
    var this.message = "+";
    var this.args = [Lazy.new(function () {2})];
}|
assert { c1.eval() == 3 };

// A lazy function that returns the sum of this and the second argument,
// regardless of the first argument.
function Float.addSecond
{
    this + call.evalArgAt(1);
}|
var c2 = do (CallMessage.new())
{
    var this.target = 2;
    var this.message = "addSecond";
    var this.args = [Lazy.new(function () { assert (false) }),
                    Lazy.new(function () { echo (5); 5 })];
}|
assert { c2.eval() == 7 };
[00000454] *** 5
```

- **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 22.4.1.1](#).

urbiscript
Session

```
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] !!! sumTwice: invalid index: 3
[01234567:error] !!!      called from: evalArgAt
[00000014:error] !!!      called from: sumTwice
sumTwice(3.14, one(), one());
[00000015:error] !!! sumTwice: invalid index: 3.14
[01234567:error] !!!      called from: evalArgAt
[00000015:error] !!!      called from: sumTwice
```

- **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]
```

urbiscript
Session

- **message**

The name under which the function was called.

```
function myself { call.message }|
assert(myself() == "myself");
```

urbiscript
Session

- **sender**

The object *from which* the invocation was made (the *caller* in other languages). Not to be confused with `target`.

```
function Object.getSender { call.sender } |
function Object.callGetSender { getSender() } |

assert
{
    // Call from the current Lobby, with the Lobby as target.
    getSender() === lobby;
    // Call from the current Lobby, with Object as the target.
    Object.getSender() === lobby;
    // Ask Lobby to call getSender.
    callGetSender() === lobby;
    // Ask Global to call getSender.
    Object.callGetSender() === Object;
};
```

urbiscript
Session

- **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 Object.getTarget { call.target } |
function Object.callGetTarget { getTarget() } |
```

urbiscript
Session

```
assert
{
    // Call from the current Lobby, with the Lobby as target.
    getTarget() === lobby;
    // Call from the current Lobby, with Global as the target.
    Object.getTarget() === Object;
    // Ask Lobby to call getTarget.
    callGetTarget() === lobby;
    // Ask Global to call getTarget.
    Object.callGetTarget() === Object;
};
```

22.5 Channel

Returning data, typically asynchronous, with a label so that the “caller” can find it in the flow.

22.5.1 Prototypes

- [Object](#)

22.5.2 Construction

Channels are created like any other object. The constructor must be called with a string which will be the label.

urbiscript
Session

```
var ch1 = Channel.new("my_label");
[00000201] Channel<my_label>

ch1 << 1;
[00000201:my_label] 1

var ch2 = ch1;
[00000201] Channel<my_label>

ch2 << 1/2;
[00000201:my_label] 0.5
```

22.5.3 Slots

- `'<<'(value)`

Send *value* to `this` tagged by its label if non-empty.

urbiscript
Session

```
Channel.new("label") << 42;
[00000000:label] 42

Channel.new("") << 51;
[00000000] 51
```

- `echo(value)`

Same as `lobby.echo(value, name)`, see [Lobby.echo](#).

urbiscript
Session

```
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.

urbiscript
Session

```
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`.

urbiscript
Session

```
// 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_0x100b93460
// 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 () {}]
```

- **name**

The name of the Channel, used to label the output.

urbiscript
Session

```
assert
{
    Channel.new("").name == "";
    Channel.new("foo").name == "foo";
};
```

- **null**

A predefined stream whose `enabled` is `false`.

urbiscript
Session

```
Channel.null << "Message";
```

urbiscript
Session

- **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"
```

- **topLevel**

A predefined stream for regular output. Strings are output escaped.

```
Channel.topLevel << "Message";
[00015895] "Message"
Channel.topLevel << "\"quote\"";
[00015895] "\"quote\""
```

urbiscript
Session

- **warning**

A predefined stream for warning messages. Strings sent to it are not escaped.

urbiscript
Session

```
Channel.warning << "Message";
[00015895:warning] Message
Channel.warning << "\"quote\"";
[00015895:warning] "quote"
```

22.6 Code

Functions written in urbiscript.

22.6.1 Prototypes

- Comparable
- Executable

22.6.2 Construction

The keywords `function` and `closure` build Code instances.

```
function(){}.protos[0] === 'package'.lang.getSlotValue("Code");
closure (){}.protos[0] === 'package'.lang.getSlotValue("Code");
```

Assertion Block

22.6.3 Slots

- `'==' (that)`

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 };
```

Assertion Block

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 };
```

Assertion Block

Arguments do matter, even if in practice the functions are the same.

```
function (var ignored) {} != function () {};
function (var x) { x } != function (y) { y };
```

Assertion Block

A lazy function cannot be equal to a strict one.

```
function () { 1 } != function { 1 };
```

Assertion Block

If the functions capture different variables, they are different.

```
{
  var x;
  function Object.capture_x() { x };
  function Object.capture_x_again () { x };
  {
    var x;
    function Object.capture_another_x() { x };
  }
}|;
assert
{
  getSlotValue("capture_x") == getSlotValue("capture_x_again");
  getSlotValue("capture_x") != getSlotValue("capture_another_x");
};
```

urbiscript Session

If the functions capture different targets, they are different.

urbiscript Session

```

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;

```

Assertion Block

```

function () {}.apply([]);
[00000001:error] !!! apply: argument list must begin with 'this'

function () {}.apply([1, 2]);
[00000002:error] !!! apply: expected 0 argument, given 1

```

urbiscript Session

- **asString**

Conversion to `String`.

```

closure () { 1 }.asString() == "closure () { 1 }";
function () { 1 }.asString() == "function () { 1 }";

```

Assertion Block

- **bodyString**

Conversion to `String` of the routine body.

```

closure () { 1 }.bodyString() == "1";
function () { 1 }.bodyString() == "1";

```

Assertion Block

- **spawn(*clear*)**

Run `this`, with fresh tags if `clear` is true, otherwise under the control of the current tags. Return the spawn `Job`. This is an internal function, instead, use `detach` and `disown`.

urbiscript Session

```

var jobs = []|;
var res = []|;
for (var i : [0, 1, 2])
{
    jobs << closure () { res << i; res << i }.spawn(true) |
    if (i == 2)
        break
}|
jobs;
[00009120] [Job<shell_7>, Job<shell_8>, Job<shell_9>]

```

urbiscript
Session

```
// 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 << closure () { res << i; res << i }.spawn(false) |
    if (i == 2)
        break
}
jobs;
[00009120] [Job<shell_10>, Job<shell_11>, Job<shell_12>]
// Give some time to get the output of the detached expressions.
sleep(100ms);
assert (res == [0, 1, 0]);
```

22.7 Comparable

Objects that can be compared for equality and inequality. See also [Orderable](#).

22.7.1 Example

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 `==`.

urbiscript
Session

```
class Foo : Comparable
{
    var value = 0;
    function init (v) { value = v; };
    function '==' (that) { value == that.value; };
}|;
assert
{
    Foo.new(1) == Foo.new(1);
    Foo.new(1) != Foo.new(2);
};
```

22.7.2 Prototypes

- `Object`

22.7.3 Slots

- `'!=' (that)`

Whether `! (this == that)`.

urbiscript
Session

```
class FiftyOne : Comparable
{
    function '==' (that) { 51 == that };
}|;
assert
{
    FiftyOne == 51;
    FiftyOne != 42;
};
```

- `'==' (that)`

Whether `! (this != that)`.

```
class FortyTwo : Comparable
{
    function '!=' (that) { 42 != that };
}|;
assert
{
    FortyTwo != 51;
    FortyTwo == 42;
};
```

urbiscript
Session

22.8 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.

22.8.1 Prototypes

- `Object`

22.8.2 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.

urbiscript
Session

```
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`”.

urbiscript
Session

```
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;
};
```

22.9 Control

Control is a namespace for control sequences used by the Urbi engine to execute some urbiscipt features. It is internal; in other words, users are not expected to use it, much less change it.

22.9.1 Prototypes

- **Object**

22.9.2 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**. Same as calling `Code.spawn: closure () { exp }.spawn(true)`. See also [disown](#).

urbiscipt
Session

```
{
    var jobs = [];
    var res = [];
    for (var i : [1, 2, 3])
    {
        jobs << detach({ res << i; sleep(i * 100ms); res << i }) |
        if (i == 3)
            // break interrupts all the jobs that are launched by the for.
            break
    };
    // The third job did not even have enough time to start.
    assert (res == [1, 2]);
    jobs
};
[00009120] [Job<shell_7>, Job<shell_8>, Job<shell_9>]
```

- **'disown'(*exp*)**

Same as [detach](#) except that tags used to tag the [disown](#) call are not inherited inside the expression. Return the spawned **Job**. Same as calling `Code.spawn: closure () { exp }.spawn(true)`.

urbiscipt
Session

```
{
    var jobs = [];
    var res = [];
    for (var i : [1, 2, 3])
    {
        jobs << disown({ res << i; sleep(i * 100ms); res << i }) |
        if (i == 3)
            // Contrary to the previous case, the jobs are not controlled
            // by this break.
            break
    };
    jobs.each (function (var j) { j.waitForTermination() });
    assert (res == [1, 2, 3, 1, 2, 3]);
    jobs
};
[00009120] [Job<shell_10>, Job<shell_11>, Job<shell_12>]
```

- **persistent(*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
```

urbiscipt
Session

as

urbiscript
Session

```
var u = persist (condition, delay);
at (u.val)
action
```

The `persist` action will be controlled by the same tags as the initial `at` block.

22.10 Date

This class is meant to record dates in time, with microsecond resolution. See also [System.time](#).

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

22.10.1 Prototypes

- [Orderable](#)
- [Comparable](#)

22.10.2 Construction

Without argument, newly constructed Dates refer to the current date.

urbiscript
Session

```
Date.new;
[00000001] 2010-08-17 14:40:52.549726
```

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.

urbiscript
Session

```
Date.new("2003-10-10 20:10:50:637");
[00000001] 2003-10-10 20:10:50.637000

Date.new("2003-10-10 20:10:50");
[00000001] 2003-10-10 20:10:50.000000

Date.new("2003-Oct-10 20:10");
[00000002] 2003-10-10 20:10:00.000000

Date.new("2003-10-10 20");
[00000003] 2003-10-10 20:00:00.000000
```

Pay attention that the format is rather strict; for instance too many spaces between day and time result in an error.

```
Date.new("2003-10-10 20:10:50");
[00001968:error] !!! new: cannot convert to date: 2003-10-10 20:10:50
```

urbiscript
Session

Pay attention that the format is not strict enough either; for instance, below, the ‘.’ separator seem to prefix microseconds, but actually merely denotes the minutes. Seconds must be spelled out in order to introduce microseconds.

urbiscript
Session

```
Date.new("2003-10-10 00.12");
[00000003] 2003-10-10 00:12:00.000000

Date.new("2003-10-10 00:00.12");
[00000003] 2003-10-10 00:00:12.000000
```

22.10.3 Slots

- ‘+’ (*that*)

The date which corresponds to waiting [Duration](#) *that* after [this](#).

Assertion
Block

```
Date.new("2010-08-17 12:00:00.2") + 63.2s == Date.new("2010-08-17 12:01:03.4");
```

- ‘-’ (*that*)

If *that* is a Date, the difference between [this](#) and *that* as a [Duration](#).

Assertion
Block

```
Date.new("2010-08-17 12:01:00.50") - Date.new("2010-08-17 12:00") == 60.5s;
Date.new("2010-08-17 12:00") - Date.new("2010-08-17 12:01") == -60s;
```

If *that* is a Duration or a Float, the corresponding Date.

Assertion Block

```
Date.new("2010-08-17 12:01") - 60s == Date.new("2010-08-17 12:00");
Date.new("2010-08-17 12:01") - 60s
== Date.new("2010-08-17 12:01") - Duration.new(60s);
```

- '*<*' (*that*)

Order comparison.

Assertion Block

```
Date.new("2010-08-17 12:00") < Date.new("2010-08-17 12:01");
!(Date.new("2010-08-17 12:01") < Date.new("2010-08-17 12:00"));
```

- '*==*' (*that*)

Equality test.

Assertion Block

```
Date.new("2010-08-17 12:00:00.123") == Date.new("2010-08-17 12:00:00.123");
Date.new("2010-08-17 12:00") != Date.new("2010-08-17 12:01");
```

- *asFloat*

The duration since the [epoch](#), as a Float.

```
var d = Date.new("2002-01-20 23:59:59");

d.asFloat() == d - d.epoch;
d.asFloat().isA(Float);
```

Assertion Block

- *asString*

Present as '*yyyy-mm-dd hh:mm:ss.us*' where:

- *yyyy* is the four-digit year,
- *mm* the three letters name of the month (Jan, Feb, ...),
- *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),
- *ss* the two-digit number of seconds in the minute (from 0 to 59), and
- *iiiiii* the six-digit number of microseconds.

```
Date.new("2009-02-14 00:31:30").asString() == "2009-02-14 00:31:30.000000";
```

Assertion Block

- *day*

The day as a [Float](#).

```
var d = Date.new("2010-09-29 17:32:53");
d.day == 29;
d.day = 1;
d == Date.new("2010-09-01 17:32:53");
```

Assertion Block

```
Date.new("2010-02-01 17:32:53").day = 29;
[00000001:error] !!! updateSlot: Day of month is not valid for year
```

urbiscript Session

- *epoch*

A fixed value, the “origin of times”: January 1st 1970, at midnight.

urbiscript Session

```
Date.epoch == Date.new("1970-01-01 00:00:00.00");
```

- **hour**

The hour as a [Float](#). Always less than 24.

```
var d = Date.new("2010-09-29 17:32:53");
d.hour == 17;
d.hour = 8;
d == Date.new("2010-09-29 08:32:53");
```

Assertion Block

- **microsecond**

The number of microseconds in the current second, as a [Float](#). See also [us](#). Always less than 1000000.

```
{
  var d = Date.new("2010-09-29 17:32:53.123456");
  assert(d.microsecond == 123456);
  d.microsecond = 654321;
  assert(d == Date.new("2010-09-29 17:32:53.654321"));
};
```

urbiscript Session

- **minute**

The minute as a [Float](#). Always less than 60.

```
var d = Date.new("2010-09-29 17:32:53");
d.minute == 32;
d.minute = 12;
d == Date.new("2010-09-29 17:12:53");
```

Assertion Block

- **month**

The month as a [Float](#). Always less or equal to 12.

Assertion Block

```
var d = Date.new("2010-09-29 17:32:53");
d.month == 9;
d.month = 3;
d == Date.new("2010-03-29 17:32:53");
```

Assertion Block

- **now**

The current date. Equivalent to `Date.new`.

urbiscript Session

```
Date.now;
[00000000] 2012-03-02 15:31:42
```

- **second**

The second as a [Float](#).

Assertion Block

```
var d = Date.new("2010-09-29 17:32:53");
d.second == 53;
d.second = 37;
d == Date.new("2010-09-29 17:32:37");
```

Assertion Block

- **timestamp**

Synonym for [asFloat](#).

- **us**

The *total* number of microseconds since midnight, as a [Float](#). See also [microsecond](#).

Assertion Block

```
Date.new("2010-08-17 00:00:00.0") .us == 0;
Date.new("2010-08-17 00:00:00.123456").us == 123456;
Date.new("2010-08-17 00:00:01.234567").us == 1234567;
Date.new("2010-08-17 01:02:03.456789").us
  == (1 * 3600 + 2 * 60 + 3) * 1000000 + 456789;
```

urbscript
Session

```
{
  var d = Date.new("2010-09-29 17:32:53.123456");
  assert(d.us == 63173123456);
  d.us = 123456;
  assert(d == Date.new("2010-09-29 00:00:00.123456"));
};
```

- **year**

The year as a [Float](#).

Assertion
Block

```
var d = Date.new("2010-09-29 17:32:53");
d.year == 2010;
d.year = 2000;
d == Date.new("2000-09-29 17:32:53");
```

22.11 Dictionary

A *dictionary* is an *associative array*, also known as a *hash* in some programming languages. They are arrays whose indexes are arbitrary objects.

22.11.1 Example

The following session demonstrates the features of the Dictionary objects.

urbiscript
Session

```
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



```

22.11.2 Hash values

Arbitrary objects can be used as dictionary keys. To map to the same cell, two objects used as keys must have equal hashes (retrieved with the `Object.hash` method) and be equal to each other (in the `Object.'=='` sense).

This means that two different objects may have the same hash: the equality operator (`Object.'=='`) is checked in addition to the hash, to handle such collision. However a good hash algorithm should avoid this case, since it hinders performances.

See `Object.hash` for more detail on how to override hash values. Most standard value-based classes implement a reasonable hash function: see `Float.hash`, `String.hash`, `List.hash`, ...

22.11.3 Prototypes

- [Comparable](#)
- [Container](#)
- [Object](#)
- [RangeIterable](#)

22.11.4 Construction

The Dictionary constructor takes arguments by pair (key, value).

urbiscript
Session

```
Dictionary.new("one", 1, "two", 2);
[00000000] ["one" => 1, "two" => 2]
Dictionary.new();
[00000000] [ => ]
```

There must be an even number of arguments.

urbiscript
Session

```
Dictionary.new("1", 2, "3");
[00000001:error] !!! new: odd number of arguments
```

You are encouraged to use the specific syntax for Dictionary literals:

urbiscript
Session

```
[ "one" => 1, "two" => 2];
[00000000] ["one" => 1, "two" => 2]
[=>];
[00000000] [ => ]
```

An extra comma can be added at the end of the list.

urbiscript
Session

```
[ "one" => 1,
  "two" => 2,
];
[00000000] ["one" => 1, "two" => 2]
```

It is guaranteed that the pairs to insert are evaluated left-to-write, key first, the value.

Assertion
Block

```
["a".fresh() => "b".fresh(), "c".fresh() => "d".fresh()]
== ["a_5"      => "b_6",      "c_7"      => "d_8"];
```

Duplicate keys in Dictionary literal are an error. On this regards, urbiscript departs from choices made in JavaScript, Perl, Python, Ruby, and probably many other languages.

```
[ "one" => 1, "one" => 2];
[00000001:error] !!! duplicate dictionary key: "one"
```

urbiscript
Session

22.11.5 Slots

- `'==' (that)`

Whether `this` equals `that`. Expects members to be [Comparable](#).

```
[ => ] == [ => ];
[ "a" => 1, "b" => 2] == [ "b" => 2, "a" => 1];
```

Assertion
Block

- `'[]' (key)`

Syntactic sugar for `get(key)`.

```
assert ([ "one" => 1][ "one"] == 1);
[ "one" => 1][ "two"];
[00000012:error] !!! missing key: two
```

urbiscript
Session

- `'[]=' (key, value)`

Syntactic sugar for `set(key, value)`, but returns `value`.

```
var d = [ "one" => "2"];
(d[ "one" ] = 1) == 1;
d[ "one" ] == 1;
```

Assertion
Block

- `asBool`

Negation of `empty`.

```
[=>].asBool() == false;
[ "key" => "value" ].asBool() == true;
```

Assertion
Block

- `asList`

The contents of the dictionary as a [Pair](#) list (`key, value`).

```
[ "one" => 1, "two" => 2].asList() == [ ("one", 1), ("two", 2)];
```

Assertion
Block

Since Dictionary derives from [RangeIterable](#), it is easy to iterate over a Dictionary using a range-for ([Section 21.7.6](#)). No particular order is ensured.

urbiscript
Session

```
{
    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]";
```

Assertion Block

- **asTree**

Display the content of the List as a tree representation.

```
echo("simple dictionary:" + [{"key1" => "elt1", "key2" => [{"key3" => "elt3"}]].asTree());
[00000001] *** simple dictionary:
[
    key1 => elt1,
    key2 =>
    [
        key3 => elt3,
    ]
]
echo("dictionary with list:" +
    [{"key1" => "elt1", "key2" => [{"key3" => [{"key4", "key5"}]}]].asTree());
[00000002] *** dictionary with list:
[
    key1 => elt1,
    key2 =>
    [
        key3 =>
        [
            key4,
            key5,
        ]
    ]
]
```

urbiscript Session

- **clear**

Empty the dictionary.

```
["one" => 1].clear().empty;
```

Assertion Block

- **elementAdded**

An event emitted each time a new element is added to the Dictionary.

- **elementChanged**

An event emitted each time the value associated to a key of the Dictionary is changed.

- **elementRemoved**

An event emitted each time an element is removed from the Dictionary.

```
d = [ => ] |;
at(d.elementAdded?) echo ("added");
at(d.elementChanged?) echo ("changed");
at(d.elementRemoved?) echo ("removed");

d["key1"] = "value1";
[00000001] "value1"
```

urbiscript Session

```
[00000001] *** added
d["key2"] = "value2";
[00000001] "value2"
[00000001] *** added

d["key2"] = "value3";
[00000001] "value3"
[00000001] *** changed

d.erase("key2");
[00000002] ["key1" => "value1"]
[00000001] *** removed

d.clear();
[00000003] [ => ]
[00000001] *** removed

d.clear();
[00000003] [ => ]
```

- **empty**

Whether the dictionary is empty.

Assertion Block

```
[=>].empty == true;
["key" => "value"].empty == false;
```

- **erase(key)**

Remove the mapping for *key*.

urbiscript Session

```
{
  var d = ["one" => 1, "two" => 2];
  assert
  {
    d.erase("two") === d;
    d == ["one" => 1];
  };

  try
  {
    ["one" => 1, "two" => 2].erase("three");
    echo("never reached");
  }
  catch (var e if e.isA(Dictionary.KeyError))
  {
    assert(e.key == "three")
  };
};
```

- **get(key)**

The value associated to *key*. A `Dictionary.KeyError` exception is thrown if the key is missing.

urbiscript Session

```
var d = ["one" => 1, "two" => 2] |;

assert(d.get("one") == 1);
["one" => 1, "two" => 2].get("three");
[00000010:error] !!! missing key: three

try
{
  d.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.

```

var d = ["one" => 1, "two" => 2];
d.getWithDefault("one", -1) == 1;
d.getWithDefault("three", 3) == 3;

```

Assertion Block

- `has(key)`

Whether the dictionary has a mapping for `key`.

```

var d = ["one" => 1];
d.has("one");
!d.has("zero");

```

Assertion Block

The infix operators `in` and `not in` use `has` (see Section 21.1.8.7).

```

"one" in      ["one" => 1];
"two" not in ["one" => 1];

```

Assertion Block

- `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]

```

urbiscript Session

Assertion Block

- `keys`

The list of all the keys. No particular order is ensured. Since [List](#) features the same function, uniform iteration over a List or a Dictionary is possible.

```

var d = ["one" => 1, "two" => 2];
d.keys == ["one", "two"];

```

urbiscript Session

- `matchAgainst(handler, pattern)`

Pattern matching on members. See [Pattern](#).

```

{
    // Match a subset of the dictionary.
    ["a" => var a] = ["a" => 1, "b" => 2];
    // get the matched value.
    assert(a == 1);
}

```

urbiscript Session

- `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("two", 2)
    .set("one", 1);
[00000000] ["one" => 1, "two" => 2]

```

Assertion Block

- `size`

Number of element in the dictionary.

```
var d = [=>];  d.size == 0;
d["a"] = 10;   d.size == 1;
d["b"] = 20;   d.size == 2;
d["a"] = 30;   d.size == 2;
```

22.12 Directory

A *Directory* represents a directory of the file system.

22.12.1 Prototypes

- `Object`

22.12.2 Construction

A *Directory* can be constructed with one argument: the path of the directory using a `String` or a `Path`. It can also be constructed by the method `open` of `Path`.

urbiscript
Session

```
Directory.new(".");
[00000001] Directory(".");
Directory.new(Path.new("."));
[00000002] Directory(".")
```

22.12.3 Slots

- `'/'(str)`

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

The `str` `String` is concatenated with the directory path. If the resulting path is either a directory or a file, `'/'` will return either a `Directory` or a `File` object.

urbiscript
Session

```
var dir1 = Directory.create("dir1");
var dir2 = Directory.create("dir1/dir2");
var file = File.create("dir1/file");
dir1 / "dir2";
[00000001] Directory("dir1/dir2")
dir1 / "file";
[00000002] File("dir1/file")
```

- `'<<'(entity)`

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

If `entity` is a `Directory` or a `File`, `'<<'` copies `entity` in the `this` directory. Return `this` to allow chained operations.

urbiscript
Session

```
var dir1 = Directory.create("dir1");
var dir2 = Directory.create("dir2");
var file = File.create("file");
dir1 << file << dir2;
[00000001] Directory("dir1")
dir1.content;
[00000003] ["dir2", "file"]
dir2;
[00000004] Directory("dir2")
file;
[00000005] File("file")
```

- **asList**

The contents of the directory as a `Path` list. The various paths include the name of the directory `this`.

- **asPath**

A `Path` being the path of the directory.

- **asString**

A `String` containing the path of the directory.

Assertion Block

```
Directory.new(".").asString() == ".";
```

- **basename**

Return a `String` containing the path of the directory without its dirname. Deprecated, use `asPath.basename` instead.

Assertion Block

```
var dir1 = Directory.create("dir1");
var dir2 = Directory.create("dir1/dir2");

dir1.basename == dir1.asPath().basename == "dir1";
[00000808:warning] !!! 'basename' is deprecated, use 'asPath.basename'
dir2.basename == dir2.asPath().basename == "dir2";
[00000811:warning] !!! 'basename' is deprecated, use 'asPath.basename'
```

- **clear**

Remove all children recursively but not the directory itself. After a call to `clear`, a call to `empty` should return `true`.

Assertion Block

```
var dir1 = Directory.create("dir1");
var dir2 = Directory.create("dir1/dir2");
var file1 = File.create("dir1/file1");
var file2 = File.create("dir1/dir2/file2");

dir1.content == ["dir2", "file1"];
dir2.content == ["file2"];
dir1.clear().isVoid;
dir1.empty;
```

- **content**

The contents of the directory as a `String` list. The strings include only the last component name; they do not contain the directory name of `this`.

- **copy(*dirname*)**

Copy recursively all items of the `this` directory into the directory `dirname` after creating it.

urbiscript
Session

```
var dir1 = Directory.create("dir1")|;
var dir2 = Directory.create("dir1/dir2")|;
var file = File.create("dir1/file")|;
var directory1 = dir1.copy("directory1")|;
dir1;
[00000002] Directory("dir1")
directory1.content;
[00000003] ["dir2", "file"]
```

- **copyInto(*dirname*)**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Copy `this` into `dirname` without creating it.

```
var dir = Directory.create("dir")|;
var dir1 = Directory.create("dir1")|;
var dir2 = Directory.create("dir1/dir2")|;
var file = File.create("dir1/file")|;
dir1.copyInto(dir);
[00000001] Directory("dir/dir1")
dir1;
[00000002] Directory("dir1")
dir1.content;
[00000003] ["dir2", "file"]
dir.content;
[00000004] ["dir1"]
Directory.new("dir/dir1").content;
[00000005] ["dir2", "file"]
```

urbiscript
Session

- `create(name)`

Create the directory `name` where `name` is either a [String](#) or a [Path](#). In addition to system errors that can occur, errors are raised if directory or file `name` already exists.

```
Directory.new("dir");
[00000001:error] !!! new: no such file or directory: dir
var dir = Directory.create("dir");
[00000002] Directory("dir")
dir = Directory.create("dir");
[00000001:error] !!! create: directory already exists: dir
dir.content;
[00000003] []
```

urbiscript
Session

- `createAll(name)`

Create the directory `name` where `name` is either a [String](#) or a [Path](#). If `name` is a path (or a [String](#) describing a path) no errors are raised if one directory doesn't exist or already exists. Instead `createAll` creates them all as in the Unix 'make -p' command.

```
Directory.create("dir1/dir2/dir3");
[00000001:error] !!! create: no such file or directory: "dir1/dir2/dir3"
var dir1 = Directory.create("dir1");
[00000002] Directory("dir1")
Directory.createAll("dir1/dir2/dir3");
[00000002] Directory("dir1/dir2/dir3")
```

urbiscript
Session

- `empty`

Whether the directory is empty.

Assertion
Block

```
var dir = Directory.create("dir");
dir.empty;
File.create("dir/file");
!dir.empty;
```

urbiscript
Session

- `exists`

Whether the directory still exists.

Assertion
Block

```
var dir = Directory.create("dir");
dir.exists;
dir.remove().isVoid;
!dir.exists;
```

urbiscript
Session

- `fileCreated(name)`

Event launched when a file is created inside the directory. May not exist if not supported by your architecture.

urbiscript
Session

```

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.

urbiscript
Session

```

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();
}|;
```

- **lastModifiedDate**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Return a **Date** object stating when the directory was last modified. Deprecated, use `asPath.lastModifiedDate` instead.

- **moveInto(*dirname*)**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Move `this` into `dirname` without creating it.

urbiscript
Session

```

var dir1 = Directory.create("dir1")|;
var dir2 = Directory.create("dir1/dir2")|;
var file = File.create("dir1/file")|;
var dir = Directory.create("dir")|;
dir1.moveInto(dir);
```

```
[00000001:warning] !!! 'rename(name)' is deprecated, use 'asPath.rename(name)'
dir1;
[00000002] Directory("dir/dir1")
assert
{
    dir1.content == ["dir2", "file"];
    dir.content == ["dir1"]
};
```

- **parent**

The Directory parent of `this`.

urbiscript
Session

```
Directory.create("dir")|;
var dir = Directory.create("dir/dir")|;
dir.parent;
[00000001] Directory("dir")
assert(dir.parent.parent.asString() == Directory.current.asString());
```

- **remove**

Remove `this` directory if it is empty.

urbiscript
Session

```
var dir = Directory.create("dir")|;
File.create("dir/file")|;
dir.remove();
[00000001:error] !!! remove: directory not empty: dir
dir.clear();
dir.remove();
assert(!dir.exists);
```

- **removeAll**

Remove all children recursively including the directory itself.

urbiscript
Session

```
var dir1 = Directory.create("dir1")|;
var dir2 = Directory.create("dir1/dir2")|;
var file1 = File.create("dir1/file1")|;
var file2 = File.create("dir1/dir2/file2")|;
dir1.removeAll();
assert(!dir1.exists);
```

- **rename**

Rename or move the directory. Deprecated, use `asPath.rename` instead.

urbiscript
Session

```
var dir = Directory.create("dir")|;
File.create("dir/file")|;
dir.rename("other");
[00000001:warning] !!! 'rename(name)' is deprecated, use 'asPath.rename(name)'
dir;
[00000002] Directory("other")
dir.content;
[00000003] ["file"]
var dir2 = Directory.create("dir2")|;
dir.rename("dir2/other2");
[00000002:warning] !!! 'rename(name)' is deprecated, use 'asPath.rename(name)'
dir;
[00000005] Directory("dir2/other2")
dir.content;
[00000006] ["file"]
```

- **size**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

The size of all the directory content computed recursively, in number of bytes.

urbiscript
Session

```
var dir = Directory.create("dir")|;  
Directory.create("dir/dir")|;  
File.save("dir/file", "content");  
var file1 = File.create("dir/file")|;  
File.save("dir/dir/file", "content");  
var file2 = File.create("dir/dir/file")|;  
assert(dir.size() == file1.size() + file2.size());
```

22.13 Duration

This class records differences between [Dates](#).

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

22.13.1 Prototypes

- [Float](#)

22.13.2 Construction

Without argument, a null duration.

urbiscript
Session

```
Duration.new();
[00000001] Duration(0s)
Duration.new(1h);
[00023593] Duration(3600s)
```

Durations can be negative.

urbiscript
Session

```
Duration.new(-1);
[00000001] Duration(-1s)
```

22.13.3 Slots

- [asFloat](#)

Return the duration as a [Float](#).

```
Duration.new(1000).asFloat() == 1000;
Duration.new(1000.1234).asFloat() == 1000.1234;
```

Assertion
Block

- [asString](#)

Return the duration as a [String](#).

Assertion
Block

```
Duration.new(1000).asString() == "1000s";
```

Assertion
Block

- [seconds](#)

Return the duration as a [Float](#).

Assertion
Block

```
Duration.new(1000) .seconds() == 1000;
Duration.new(1000.52).seconds() == 1000.52;
```

Assertion
Block

22.14 Enumeration

Prototype of enumeration types.

22.14.1 Examples

See [Section 21.5](#).

22.14.2 Prototypes

- [RangeIterable](#)
- [Container](#)

22.14.3 Construction

An `Enumeration` is created with two arguments: the name of the enumeration type, and the list of possible values. Most of the time, it is a good idea to store it in a variable with the same name.

urbiscript
Session

```
var Direction = Enumeration.new("Direction", ["up", "down", "left", "right"]);
[00000001] Direction
Direction.up;
[00000002] up
```

The following syntax is equivalent.

urbiscript
Session

```
enum Direction
{
    up,
    down,
    left,
    right
};
[00000001] Direction
```

The created values are derive from the created enumeration type.

Assertion
Block

```
Direction.isA(Enumeration);
Direction.up.isA(Direction);
```

22.14.4 Slots

- [asList](#)

Synonym for [values](#).

```
Direction.asList()
== [Direction.up, Direction.down, Direction.left, Direction.right];
```

Assertion
Block

Since it also derives from [RangeIterable](#), this enables all its features. For instance:

```
Direction.each(function (var d) { echo(d) });
[00000001] *** up
[00000001] *** down
[00000001] *** left
[00000001] *** right

for (var d in Direction)
    echo(d);
[00000001] *** up
[00000001] *** down
[00000001] *** left
```

urbiscript
Session

```
[00000001] *** right

for| (var d in Direction)
    echo(d);
[00000001] *** up
[00000001] *** down
[00000001] *** left
[00000001] *** right

assert
{
    Direction.any(closure (var v) { v == Direction.up });
};
```

- **asString**

The name of the enumeration, or the name of the member if applied to a member.

```
Direction.asString() == "Direction";
Direction.up.asString() == "up";
```

Assertion Block

- **has(*v*)**

Whether *v* is a member of `this`. Since `Enumeration` derives from `Container`, it provides all its features.

```
enum CardinalDirection { north, east, south, west }!;

assert
{
    Direction.has(Direction.up);
    Direction.up in Direction;
    12 not in Direction;
    CardinalDirection.south not in Direction;
};
```

urbiscript Session

- **name**

The name of the enumeration. See also `asString`.

```
Direction.name == "Direction";
Direction.up.name == "Direction";
```

Assertion Block

- **size**

Number of possible values in the enumeration type. Equivalent to `values.size`.

```
Direction.size == 4;
```

Assertion Block

- **values**

The list of values.

```
Direction.values
== [Direction.up, Direction.down, Direction.left, Direction.right];
```

Assertion Block

22.15 Event

An *event* can be “emitted” and “caught”, or “sent” and “received”. See also [Section 12.2](#).

22.15.1 Examples

There are several examples of uses of events in the documentation of event-based constructs. See `at` ([Section 21.11.1](#)), `waituntil` ([Section 21.11.2](#)), `whenever` ([Section 21.11.4](#)), and so forth. The tutorial chapter about event-based programming contains other examples, see [Chapter 12](#).

22.15.2 Synchronicity of Event Handling

A particular emphasis should be put on the *synchronicity* of the event handling, i.e., whether the bodies of the event handlers are run before the control flow returns from the event emission. By default, (i.e., `at (e?...)` and `e!(...)/e.emit(...)`) the execution is *asynchronous*, but if either the emitted or the handler is marked asynchronous (i.e., `at sync (e?...)` or `e.syncEmit(...)`), then the execution is *synchronous*.

Contrast the following examples:

Asynchronous handlers

```
var e = Event.new() |;

at (e?)
  { echo("a"); sleep(20ms); echo("b") }
onleave
  { echo("c"); sleep(20ms); echo("d") };

e! | echo("done");
[00000001] *** done
sleep(25ms);
[00000002] *** a
[00000003] *** c
[00000101] *** b
[00000102] *** d

e.syncEmit() | echo("done");
[00000001] *** a
[00000101] *** b
[00000102] *** c
[00000202] *** d
[00000203] *** done
```

Synchronous handlers

```
var e = Event.new() |;

at sync (e?)
  { echo("a"); sleep(20ms); echo("b") }
onleave
  { echo("c"); sleep(20ms); echo("d") };

e! | echo("done");
// No need to sleep.
[00000011] *** a
[00000031] *** b
[00000031] *** c
[00000052] *** d
[00000052] *** done

e.syncEmit() | echo("done");
[00000052] *** a
[00000073] *** b
[00000073] *** c
[00000094] *** d
[00000094] *** done
```

For more information about the synchronicity of event handlers, see [Section 21.11.1.3](#).

22.15.3 Sustained Events

Events can also be sustained during a time span starting at `trigger` and ending at `handler.stop`. Note that the `onleave`-clauses of the event handlers is not executed right after the event was first triggered, but rather when it is stopped.

Synchronicity for sustained events is more complex: the `at`-clause is handled asynchronously iff *both* the emission and the handler are asynchronous, whereas the `onleave`-clause is handled asynchronously iff the emission was synchronous. Be warned, but do not depend on this, as in the future we might change this.

Asynchronous Trigger

```
var e = Event.new()|;
at (e?(var v))
{ echo("a"+v); sleep(20ms); echo("b"+v) }
onleave
{ echo("c"+v); sleep(20ms); echo("d"+v) };

var handler = e.trigger(1) | echo("triggered");
[00000001] *** triggered
[00000002] *** a1
[00000102] *** b1
sleep(200ms);
handler.stop() | echo("stopped");
[00000301] *** stopped
sleep(25ms);
[00000302] *** c1
[00000402] *** d1

// at and onleave clauses may overlap.
handler = e.trigger(2) | handler.stop();
sleep(25ms);
[00000001] *** a2
[00000002] *** c2
sleep(25ms);
[00000201] *** b2
[00000202] *** d2

handler = e.syncTrigger(3) | echo("triggered");
[00000002] *** a3
[00000102] *** b3
[00000001] *** triggered
handler.stop() | echo("stopped");
[00000302] *** c3
[00000402] *** d3
[00000301] *** stopped
```

Synchronous Trigger

```
var e = Event.new()|;
at sync (e?(var v))
{ echo("a"+v); sleep(20ms); echo("b"+v) }
onleave
{ echo("c"+v); sleep(20ms); echo("d"+v) };

var handler = e.trigger(1) | echo("triggered");
// No need to sleep.
[00000002] *** a1
[00000102] *** b1
[00000001] *** triggered
handler.stop() | echo("stopped");
[00000301] *** stopped
sleep(25ms);
[00000302] *** c1
[00000402] *** d1

// at and onleave clauses don't overlap.
handler = e.trigger(2) | handler.stop();
sleep(25ms);
[00000001] *** a2
[00000201] *** b2
[00000002] *** c2
[00000202] *** d2

handler = e.syncTrigger(3) | echo("triggered");
[00000002] *** a3
[00000102] *** b3
[00000001] *** triggered
handler.stop() | echo("stopped");
[00000302] *** c3
[00000402] *** d3
[00000301] *** stopped
```

22.15.4 Prototypes

- `Object`

22.15.5 Construction

An `Event` is created like any other object. The constructor takes no argument.

urbiscript
Session

```
var e = Event.new();
[00000001] Event_Ox9ad8118
```

22.15.6 Slots

- `'<<' (that)`

Watch a `that` 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.

urbiscript
Session

```
var e3 = Event.new()|;
var e4 = Event.new()|;
var e_watch = Event.new() << e3 << e4 |;
at (e_watch?)
  echo("!");
e3!;
```

```
[00000006] *** !
e4!;
[00000007] *** !
```

- `'||'(that)` Logical “or” on events: a new Event that triggers whenever `this` or `that` triggers.

```
var e1 = Event.new()|;
var e2 = Event.new()|;
var either = e1.'||'(e2)|;
at (either?)
  echo("!");
e1|;
[00000004] *** !
e2|;
[00000005] *** !
```

urbiscript
Session

- `asEvent`

Return `this`.

- `emit(args[])`

Fire an “instantaneous” and “asynchronous” `Event`. This function is called by the `!` operator. It takes any number of arguments, passed to the receiver when the event is caught.

```
var e = Event.new()|;
// No handler, lost message.
e.emit();
at (e?)           echo("e");
at (e?())         echo("e()");
at (e?(var x))   echo("e(%s)" % [x]);
at (e?(var x, var y)) echo("e(%s, %s)" % [x, y]);

// This is what e! does.
e.emit();
[00000135] *** e
[00000135] *** e()

// This is what e!() does: same as e!.
e.emit();
[00000138] *** e
[00000138] *** e()

// This is what e!(1, [2]) does.
e.emit(1, [2]);
[00000141] *** e
[00000141] *** e(1, [2])

// This is what e!(1, [2], "three") does.
e.emit(1, [2], "three");
[00000146] *** e
```

To sustain an event, see `trigger`. See Section 22.15.2 and `syncEmit` for details about the synchronicity of the handling.

- `hasSubscribers`

Return true if at least one subscriber is registered to this event.

```
var ev = Event.new()|;
ev.hasSubscribers;
[00000000] false
tag: at(ev?) echo(1);
```

urbiscript
Session

```
ev.hasSubscribers;
[00000000] true
tag.stop();
ev.hasSubscribers;
[00000000] false
```

- **onEvent(*guard*, *enter*, *leave*, *sync*)**

This is the low-level routine used to implement the `|at—` construct. Indeed, `at (e? if cond) enter onl` is (roughly) translated into

```
e
.onEvent(
  closure (var '$event', var '$payload') { cond },
  closure (var '$event', var '$payload', var '$pattern') { enter },
  closure (var '$event', var '$payload', var '$pattern') { leave },
  false)
```

urbiscript
Session

where the `false` would be `true` in case of an `at sync` construct. The `cond` discards the event iff it returns `void`.

```
var e = Event.new()!;
e.onEvent(
  function (var args[]) { echo("cond 1") | true },
  function (var args[]) { echo("enter 1") },
  function (var args[]) { echo("leave 1") },
  true);

e.onEvent(
  function (var args[]) { echo("cond 2") },
  function (var args[]) { echo("enter 2") },
  function (var args[]) { echo("leave 2") },
  true);

e.emit(12);
[00001619] *** cond 1
[00001619] *** enter 1
[00001619] *** leave 1
[00001619] *** cond 2

var h = e.trigger()!;
[00001620] *** cond 1
[00001620] *** enter 1
[00001620] *** cond 2

h.stop();
[00001621] *** leave 1
```

urbiscript
Session

This function is internal and it might change in the future.

- **onSubscribe**

You can optionally assign an `Event` to this slot. In this case, it is triggered each time some code starts watching this event (by setting up an `at` or a `waituntil` on it for instance). This slot cannot be inherited; it defaults to `void`.

```
var e = Event.new()!;
assert (e.onSubscribe.isVoid());
e.onSubscribe = Event.new()!;
at (e.onSubscribe?)
  echo("new subscriber");
at (e?(12)) {};
[00000001] *** new subscriber
waituntil(e?(15)),
[00000002] *** new subscriber
```

urbiscript
Session

The following example shows how to set up a `clock` event that is triggered only on the first “use” (subscription).

urbiscript
Session

```
var clock = Event.new();
clock.onSubscribe = Event.new();
at (clock.onSubscribe? if clock.subscribers.size == 1)
{
    echo("arming clock event");
    disown({ every| (1s) clock! });
};
echo("ready");
[00000000] *** ready
at (clock?)
    echo("tick");
sleep(1.5s);
[00000001] *** arming clock event
[00000002] *** tick
[00000003] *** tick
```

- **`subscribe(subscription)`**

Register a [Subscription](#) to be called each time the event triggers. Using a subscription object gives you more control on on often and how your code will be evaluated.

- **`subscribers`**

The list of subscriptions attached to this event.

- **`syncEmit(args[])`**

Same as `emit` but require a synchronous handling. See [Section 22.15.2](#) for details.

- **`syncTrigger(args[])`**

Same as `trigger` but the call will be synchronous (see [Section 22.15.2](#)). The `stop` method of the handler object will be synchronous as well. See [Section 22.15.3](#) for examples.

- **`trigger(args[])`**

Fire a sustained event (for an unknown amount of time) and return a handler object whose `stop` method stops the event. This method is asynchronous and the `stop` call will be asynchronous as well. See [Section 22.15.3](#) for examples.

22.16 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 21.8](#)) work perfectly well with any kind of object, yet it is a good idea to throw only objects that derive from `Exception`.

22.16.1 Prototypes

- `Object`
- `Traceable`

22.16.2 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.

urbiscript
Session

```
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'
```

22.16.3 Slots

`Exception` has many slots which are specific exceptions. See [Section 22.16.3.2](#) for their documentation.

22.16.3.1 General Features

- `backtrace`

The call stack at the moment the exception was thrown (not created), as a `List` of `StackFrames`, from the innermost to the outermost call. Uses `Traceable.backtrace`.

urbiscript
Session

```
//#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).

```

System.eval("1/0");
[00090441:error] !!! 1.1-3: /: division by 0
[00090441:error] !!!      called from: eval
try
{
  System.eval("1/0");
}
catch (var e)
{
  assert (e.location().asString() == "1.1-3");
}

```

urbiscript
Session

- **message**

The error message provided at construction.

Assertion
Block

```
Exception.new("Ouch").message == "Ouch";
```

22.16.3.2 Specific Exceptions

In the following, since these slots are actually Objects, what is presented as arguments to the slots are actually arguments to pass to the constructor of the corresponding exception type.

- **Argument(*routine*, *index*, *exception*)**

During the call of *routine*, the instantiation of the *index*-th argument has thrown an *exception*.

Assertion
Block

```

Exception.Argument
  .new("myRoutine", 3, Exception.Type.new("19/11/2010", Date))
  .asString()
== "myRoutine: argument 3: unexpected \"19/11/2010\", expected a Date";

```

Assertion
Block

- **ArgumentType(*routine*, *index*, *effective*, *expected*)**

Deprecated exception that derives from [Type](#). The *routine* was called with a *index*-th argument of type *effective* instead of *expected*.

```

Exception.ArgumentType
  .new("myRoutine", 1, "hisResult", "Expectation")
  .asString()
== "myRoutine: argument 1: unexpected \"hisResult\", expected a String";
[00000003:warning] !!! 'Exception.ArgumentType' is deprecated

```

Assertion
Block

- **Arity(*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).asString()
== "myRoutine: expected between 10 and 23 arguments, given 1";

```

Assertion
Block

- **BadInteger(*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](#).

Assertion
Block

```
Exception.BadInteger.new("myRoutine", "bad integer: %s", 12).asString()
== "myRoutine: bad integer: 12";
```

- **BadNumber(*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).asString()
== "myRoutine: bad number: 12.34";
```

Assertion Block

- **Constness**

An attempt was made to change a constant value.

```
Exception.Constness.new().asString()
== "cannot modify const slot";
```

Assertion Block

- **Duplicate(*fmt*, *name*)**

A duplication was made. Use the format *fmt* to create an error message from *name*.

```
Exception.Duplicate.new("duplicate dictionary key", "key").asString()
== "duplicate dictionary key: \\"key\\\"";
```

Assertion Block

- **FileNotFoundException(*name*)**

The file named *name* cannot be found.

Assertion Block

```
Exception.FileNotFound.new("foo").asString()
== "file not found: foo";
```

Assertion Block

- **Lookup(*object*, *name*)**

A failed name lookup was performed on *object* to find a slot named *name*. Suggest what the user might have meant if `Exception.Lookup.fixSpelling` is true (which is the default).

Assertion Block

```
Exception.Lookup.new(Object, "GetSlot").asString()
== "lookup failed: Object";
```

Assertion Block

- **MatchFailure**

A pattern matching failed.

Assertion Block

```
Exception.MatchFailure.new().asString()
== "pattern did not match";
```

Assertion Block

- **NegativeNumber(*routine*, *effective*)**

The *routine* was called with a negative number (*effective*). Derives from [BadNumber](#).

Assertion Block

```
Exception.NegativeNumber.new("myRoutine", -12).asString()
== "myRoutine: unexpected -12, expected non-negative number";
```

Assertion Block

- **NonPositiveNumber(*routine*, *effective*)**

The *routine* was called with a non-positive number (*effective*). Derives from [BadNumber](#).

Assertion Block

```
Exception.NonPositiveNumber.new("myRoutine", -12).asString()
== "myRoutine: unexpected -12, expected positive number";
```

Assertion Block

- **Primitive(*routine*, *msg*)**

The built-in *routine* encountered an error described by *msg*.

Assertion Block

```
Exception.Primitive.new("myRoutine", "cannot do that").asString()
== "myRoutine: cannot do that";
```

- **Redefinition(*name*)**

An attempt was made to refine a slot named *name*.

Assertion Block

```
Exception.Redefinition.new("foo").asString()
== "slot redefinition: foo";
```

- **Scheduling(*msg*)**

Something really bad has happened with the Urbi task scheduler.

Assertion Block

```
Exception.Scheduling.new("cannot schedule").asString()
== "cannot schedule";
```

- **Syntax(*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.

Assertion Block

```
Exception.Syntax
.new(Location.new(Position.new("file.u", 14, 25)),
    "unexpected pouCharque", "file.u")
.asString()
== "file.u:14.25: syntax error: unexpected pouCharque";
```

```
try
{
    System.eval("1 / / 0");
}
catch (var e)
{
    assert
    {
        e.isA(Exception.Syntax);
        e.loc.asString() == "1.5";
        e.input == "1 / / 0";
        e.message == "unexpected /";
    }
};
```

urbiscript Session

- **TimeOut**

Used internally to implement the `timeout` construct (Section 21.10.7).

Assertion Block

```
Exception.TimeOut.new().asString()
== "timeout expired";
```

- **Type(*effective*, *expected*)**

A value of type *effective* was received, while a value of type *expected* was expected.

Assertion Block

```
Exception.Type.new("hisResult", "Expectation").asString()
== "unexpected \"hisResult\"", expected a String";
```

- **UnexpectedVoid**

An attempt was made to read the value of `void`.

Assertion Block

```
Exception.UnexpectedVoid.new().asString()  
== "unexpected void";
```

```
var a = void;  
a;  
[00000016:error] !!! unexpected void  
[00000017:error] !!! lookup failed: a
```

urbiscript
Session

22.17 Executable

This class is used only as a common ancestor to [Primitive](#) and [Code](#).

22.17.1 Prototypes

- [Object](#)

22.17.2 Construction

There is no point in constructing an Executable.

22.17.3 Slots

- [asExecutable](#)

Return [this](#).

22.18 File

22.18.1 Prototypes

- [Object](#)

22.18.2 Construction

Files may be created from a [String](#), or from a [Path](#). Using `new`, 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).

```
urbiscript
Session
File.new("file.txt");
[00000001:error] !!! new: no such file or directory: file.txt

File.create("file.txt");
[00000002] File("file.txt")

File.new(Path.new("file.txt"));
[00000003] File("file.txt")
```

You may use [InputStream](#) and [OutputStream](#) to read or write to Files.

22.18.3 Slots

- [asList](#)

Read the file, and return its content as a list of its lines. Lines can be Unix-style (ended with `\n`) or DOS-style (ended with `\r\n`).

```
urbiscript
Session
File.save("file.txt", "1\n2\n"  "3\r\n4\r\n");
assert(File.new("file.txt").asList() == ["1", "2", "3", "4"]);
```

An error is throw if the file cannot be opened, for instance if it is unreadable ([Path.readable](#)).

```
urbiscript
Session
File.save("file.txt", "1\n2\n");
System.system("chmod a-r file.txt");
File.new("file.txt").asList();
[00000003:error] !!! asList: file not readable: file.txt
```

- [asPath](#)

A [Path](#) being the path of the file.

```
Assertion
Block
var file1 = File.create("file");
file1.asPath() == Path.new("file");

var dir = Directory.create("dir");
var file2 = File.create("dir/file");
file2.asPath() == Path.new("dir/file");
```

- [asPrintable](#)

A readable description of [this](#).

```
urbiscript
Session
File.save("file.txt", "1\n2\n");
assert(File.new("file.txt").asPrintable() == "File(\"file.txt\")");
```

- [asString](#)

The name of the opened file.

```
urbiscript
Session
File.save("file.txt", "1\n2\n");
assert(File.new("file.txt").asString() == "file.txt");
```

- **basename**

A [String](#) containing the path of the file without its dirname. Deprecated, use `asPath.basename` instead.

urbiscript
Session

```
var dir = Directory.create("dir")|;
var file1 = File.create("dir/file")|;
var file2 = File.create("file")|;
assert
{
    file1.basename == file1.asPath().basename == "file";
    file2.basename == file2.asPath().basename == "file";
};
[00000808:warning] !!! 'basename' is deprecated, use 'asPath().basename'
[00000811:warning] !!! 'basename' is deprecated, use 'asPath().basename'
```

- **content**

The content of the file as a [Binary](#) object.

urbiscript
Session

```
File.save("file.txt", "1\n2\n");
assert
{
    File.new("file.txt").content == Binary.new("", "1\n2\n");
};
```

- **copy(*filename*)**

Copy the file to a new file named *filename*.

urbiscript
Session

```
File.save("file", "content");
var file = File.new("file");
[00000001] File("file")
var file2 = file.copy("file2");
[00000002] File("file2")
assert(file2.content == file.content);
```

- **copyInto(*dirname*)**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Copy file into *dirname* directory.

```
var dir = Directory.create("dir")|;
var file = File.create("file")|;
file.copyInto(dir);
[00000001] File("dir/file")
file;
[00000002] File("file")
dir.content;
[00000003] ["file"]
```

urbiscript
Session

- **create(*name*)**

If the file *name* does not exist, create it and return a File to it. Otherwise, first empty it. See [OutputStream](#) for methods to add content to the file.

```
var p = Path.new("create.txt") |
assert (!p.exists);

// Create the file, and put something in it.
var f = File.create(p)|;
var o = OutputStream.new(f)|;
```

urbiscript
Session

```

o << "Hello, World!"|;
o.close();

assert
{
    // The file exists, with the expect contents.
    p.exists;
    f.content.data == "Hello, World!";

    // If we create is again, it is empty.
    File.create(p).isA(File);
    f.content.data == "";
};

```

- **lastModifiedDate**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Return a [Date](#) object stating when the file was last modified. Deprecated, use `asPath.lastModifiedDate` instead.

- **moveInto(*dirname*)**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Move file into *dirname* directory.

```

var dir = Directory.create("dir")|;
var file = File.create("file")|;
file.moveInto(dir);
[00000001] File("dir/file")
file;
[00000001] File("dir/file")
dir.content;
[00000001] ["file"]

```

urbiscript
Session

- **remove**

Remove the current file. Returns void.

```

var p = Path.new("file.txt");
!p.exists;

var f = File.create(p);
p.exists;

f.remove().isVoid;
!p.exists;

```

Assertion
Block

- **rename(*name*)**

Rename the file to *name*. If the target exists, it is replaced by the opened file. Return the file renamed. Deprecated, use `asPath.rename` instead.

```

File.save("file.txt", "1\n2\n");
File.new("file.txt").rename("bar.txt");
[00000001:warning] !!! 'rename(name)' is deprecated, use 'asPath().rename(name)'
assert
{

```

urbiscript
Session

```
!Path.new("file.txt").exists;
File.new("bar.txt").content.data == "1\n2\n";
};
```

- **save(*name*, *content*)**

Use `create` to create the File named *name*, store the *content* in it, and close the file.
Return void.

Assertion
Block

```
File.save("file.txt", "1\n2\n").isVoid;
File.new("file.txt").content.data == "1\n2\n";
```

- **size**

The size of the file, in number of bytes.

Assertion
Block

```
File.save("file.txt", "1234").isVoid;
File.new("file.txt").size == 4;
```

22.19 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*.

22.19.1 Example

The following object is set up to die verbosely.

```
urbiscript
Session
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|;
```

urbiscript
Session

Here, the object is still alive, since `alias` references it. Once it no longer does, the object dies.

```
alias = nil|;
[00000004] *** Ouch
```

urbiscript
Session

22.19.2 Prototypes

- `Object`

22.19.3 Construction

The constructor takes no argument.

```
urbiscript
Session
Finalizable.new();
[00000527] Finalizable_Ox135360
```

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 this invalid constructs,

```
urbiscript
Session
class o2
{
  protos = [Finalizable];
  function finalize()
  {
    echo("Ouch");
  }
};
```

[00000010:error] !!! apply: cannot inherit from a Finalizable without being one

write:

```
urbiscript
Session
class o1 : Finalizable
{
  function finalize()
  {
    echo("Ouch");
  }
};
```

// Or

```
var o3 =
  do (Finalizable.new())
{
  function finalize()
  {
    echo("Ouch");
  }
}|;
```

If you need multiple prototypes, do as follows.

urbiscript
Session

```
import Global.*;
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
```

22.19.4 Slots

- **finalize**

a simple function that takes no argument that will be evaluated when the object is reclaimed. Its return value is ignored.

urbiscript
Session

```
Finalizable.new().setSlot("finalize", function() { echo("Ouch") })|;
[00033240] *** Ouch
```

22.20 Float

A Float is a floating point number. It is also used, in the current version of urbiscript, to represent integers.

22.20.1 Prototypes

- Comparable

Assertion Block

```
!(0 != 0) ; 0 != 1; 1 != 0;
```

- Orderable

Assertion Block

```
0 <= 0;      0 <= 1;    !(1 <= 0);
!(0 > 0);   !(0 > 1);  1 > 0;
0 >= 0;     !(0 >= 1); 1 >= 0;
```

- RangeIterable

22.20.2 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:

Assertion Block

```
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;
```

Assertion Block

Actually, underscores can be inserted between digits to improve legibility.

Assertion Block

```
123_456.78_90 == 1234567890 / 10_000;
```

Assertion Block

There are also some special numbers, `nan`, `inf`.

```
Math.log(0) == -inf;
Math.exp(-inf) == 0;
(inf/inf).isNaN;
```

Assertion Block

A null float can also be obtained with `new`.

```
Float.new() == 0;
```

22.20.3 Slots

- `'%'(that)`
`this` modulo `that`.

Assertion Block

```
50 % 11 == 6;
```

- `'*(that)`
Product of `this` by `that`.

Assertion Block

```
2 * 3 == 6;
```

- `'**'(that)`
`this` to the `that` power ($this^{that}$). The infix exponential operator is right-associative (Section 21.1.8.1).

Assertion Block

```
2 ** 3 ** 2 == 2 ** (3 ** 2) != (2 ** 3) ** 2;
2 ** 10 == 1024;
2 ** 31 == 2_147_483_648;
-2 ** 31 == -2_147_483_648 == -(2**31);
2 ** 32 == 4_294_967_296;
-2 ** 32 == -4_294_967_296 == -(2**32);
```

- `'+'(that)`
The sum of `this` and `that`.

Assertion Block

```
1 + 1 == 2;
```

- `'-'(that)`
`this` subtracted by `that`.

Assertion Block

```
6 - 3 == 3;
```

- `'/'(that)`
The quotient of `this` divided by `that`.

Assertion Block

```
50 / 10 == 5; 10 / 50 == 0.2;
```

- `'<'(that)`
Whether `this` is less than `that`. The other comparison operators (`<=`, `>`, ...) can thus also be applied on floats since Float inherits `Orderable`.

Assertion Block

```
0 < 1; !(1 < 0); !(0 < 0);
```

- `'<<'(that)`
`this` shifted by `that` bits towards the left.

Assertion Block

```
4 << 2 == 16;
```

- `'=='(that)`
Whether `this` equals `that`.

Assertion Block

```
1 == 1;
!(1 == 2); !(2 == 1);
```

- `'>>'(that)`
`this` shifted by `that` bits towards the right.

Assertion Block

```
4 >> 2 == 1;
```

- `'^' (that)`

Bitwise *exclusive or* between `this` and `that`.

```
3 ^ 6 == 5;
```

Assertion Block

- `abs`

Absolute value of the target.

```
(-5).abs() == 5 == 5.abs(); 0.abs() == 0;
```

Assertion Block

- `acos`

Arccosine of the target.

```
0.acos() == Float.pi/2;
1.acos() == 0;
```

Assertion Block

- `asBool`

Whether non null.

Assertion Block

```
0.asBool() === false;
0.1.asBool() === true;
(-0.1).asBool() === true;
Math.inf.asBool() === true;
Math.nan.asBool() === true;
```

- `asFloat`

`this`.

Assertion Block

```
var x = 51;
x.asFloat() === x;
```

- `asin`

Arcsine of the target.

Assertion Block

```
0.asin() == 0;
```

- `asList`

Bounce to `seq`.

Assertion Block

```
0.asList() == [] ; 3.asList() == [0, 1, 2];
```

- `asString`

A `String` denoting `this`.

Assertion Block

```
42.asString() == "42";
42.51.asString() == "42.51";
21_474_836_470.asList() == "21474836470";
4_611_686_018_427_387_904.asList() == "4611686018427387904";
(-4_611_686_018_427_387_904).asString() == "-4611686018427387904";
```

- `atan`

Arctangent of `this`.

Assertion Block

```
0.atan() == 0;
1.atan() == Float.pi/4;
```

- **'bitand'(that)**

The bitwise-and between `this` and `that`.

Assertion Block

```
3 bitand 6 == 2;
```

- **'bitor'(that)**

Bitwise-or between `this` and `that`.

Assertion Block

```
3 bitor 6 == 7;
```

- **ceil**

The smallest integral value greater than or equal to `this`. See also `floor`, `round`, and `trunc`.

Assertion Block

```
0.ceil() == 0;
1.4.ceil() == 2;    1.5.ceil() == 2;    1.6.ceil() == 2;
(-1.4).ceil() == -1; (-1.5).ceil() == -1; (-1.6).ceil() == -1;
inf.ceil() == inf; (-inf).ceil() == -inf;
nan.ceil().isNaN();
```

- **clone**

Return a fresh `Float` with the same value as the target.

```
var x = 0;
var y = x.clone();
x == y;  x !== y;
```

Assertion Block

- **'compl'**

The complement to 1 of the target interpreted as a 32-bit integer.

```
compl 0 == 0xFFFF_FFFF;    compl 0xFFFF_FFFF == 0;
```

Assertion Block

- **cos**

Cosine of `this`.

```
0.cos() == 1;    Float.pi.cos() == -1;
```

Assertion Block

- **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
}
== [];
```

Assertion Block

- **'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];
```

Assertion Block

- **exp**

Exponential of the target.

```
0.exp() == 1;
1.exp() ~= 2.71828;
```

Assertion Block

- **floor**

The largest integral value less than or equal to **this**. See also **ceil**, **round**, and **trunc**.

```
0.floor() == 0;
1.4.floor() == 1;      1.5.floor() == 1;      1.6.floor() == 1;
(-1.4).floor() == -2; (-1.5).floor() == -2; (-1.6).floor() == -2;
Math.inf.floor() == Math.inf; (-Math.inf).floor() == -inf;
Math.nan.floor().isNaN;
```

Assertion Block

- **format (*finfo*)**

Format according to the **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.

Assertion Block

```
"%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";
```

Conversion to hexadecimal requires **this** to be integral.

Assertion Block

```
"%x" % 42 == "2a";
"%x" % 0xFFFF == "ffff";

"%x" % 0.5;
[00000005:error] !!! %: expected integer: 0.5
```

- **fresh**

Return a new integer at each call.

urbiscript Session

```
{
    var res = [];
    for (var i: 10)
        res << Float.fresh();
    assert (res == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
    res = [];
    for (var i: 10)
```

```

    res << Float.fresh();
    assert (res == [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]);
};

```

- **hash**

A [Hash](#) object corresponding to this float value. Equal floats (in the sense of `'=='`) have equal hashes. See [Object.hash](#).

Assertion Block

```

0.hash().isA(Hash);
0.hash() == 0.hash() != 1.hash();

```

- **hex**

A [String](#) with the conversion of `this` in hexadecimal. Requires `this` to be integral.

Assertion Block

```

0.hex() == "0";
0xFF.hex() == "ff";
0xFFFF.hex() == "ffff";
65535.hex() == "ffff";
0xffff_hex().hex() == "fffffff";

0.5.hex();
[00000005:error] !!! format: expected integer: 0.5

```

- **inf**

The infinity.

Assertion Block

```

2**2**2**2**2 == inf;
inf != -inf;
inf == - -inf;
inf * inf == inf;
inf * -inf == -inf;
inf * -2 == -inf;

```

- **isInf**

Whether `this` is infinite.

Assertion Block

```

!0.isInf(); !1.isInf(); !(-1).isInf();
!Math.nan.isInf();
inf.isInf(); (-inf).isInf();

```

- **isNaN**

Whether is NaN.

Assertion Block

```

!0.isnan(); !1.isnan(); !(-1).isnan();
!inf.isnan(); !(-inf).isnan();
nan.isnan();

```

- **limits**

See [Float.limits](#).

- **log**

The logarithm of the target.

Assertion Block

```

0.log() == -inf;
1.log() == 0;
1.exp().log() == 1;

```

- **max(arg1, ...)**

Bounces to [List.max](#) on `[this, arg1, ...]`.

Assertion Block

```
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;
```

Assertion Block

- **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
inf / inf;
[00000000] nan
```

urbiscript Session

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.

```
!(nan === nan); !(nan == nan); nan != nan;
var n = Float.nan;
n === n; n != nan;
```

Assertion Block

Use `isNaN` to check if a Float is NaN.

Assertion Block

```
(inf/inf).isNaN(); (inf/-inf).isNaN();
(nan / nan).isNaN();
(0 * nan).isNaN(); (0 * nan).isNaN();
```

- **pi**

π .

Assertion Block

```
Float.pi.cos() ** 2 + Float.pi.sin() ** 2 == 1;
```

- **random**

A random integer between 0 (included) and the target (excluded).

urbiscript Session

```
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 integral value nearest to `this` rounding half-way cases away from zero. See also `ceil`, `floor`, and `trunc`.

Assertion Block

```
0.round() == 0;
1.4.round() == 1; 1.5.round() == 2; 1.6.round() == 2;
(-1.4).round() == -1; (-1.5).round() == -2; (-1.6).round() == -2;
inf.round() == inf; (-inf).round() == -inf;
nan.round().isNaN();
```

- **seq**

The sequence of integers from 0 to `this`-1 as a list. `this` must be non-negative.

Assertion Block

```
0.seq() == [];
3.seq() == [0, 1, 2];

(-1).seq();
[00004586:error] !!! seq: expected non-negative integer: -1
```

- **sign**

Return 1 if `this` is positive, 0 if it is null, -1 otherwise.

Assertion Block

```
(-1664).sign() == -1;    0.sign() == 0;    1664.sign() == 1;
```

- **sin**

The sine of the target.

Assertion Block

```
0.sin() == 0;
```

- **sqr**

Square of the target.

Assertion Block

```
1.5.sqr() == 2.25 == 1.5 ** 2;
32.sqr() == 1024 == 32 ** 2;
```

- **sqrt**

The square root of the target.

```
1024.sqrt() == 32 == 1024 ** 0.5;
```

Assertion Block

- **random**

Initialized the seed used by the random function. As opposed to common usage, you should not use

```
{
  var now = Date.now().timestamp;
  now().random();
  var list1 = 20.map(function (dummy) { 5.random() });
  now().random();
  var list2 = 20.map(function (dummy) { 5.random() });
  assert
  {
    list1 == list2;
  }
};
```

urbiscript Session

- **tan**

Tangent of `this`.

```
0.tan() == 0;
(Float.pi/4).tan() ~= 1;
```

Assertion Block

- **times(*fun*)**

Call the functional argument `fun` `this` times.

```
3.times(function () { echo("ping") });
[00000000] *** ping
[00000000] *** ping
[00000000] *** ping
```

urbiscript Session

- **trunc**

The integral value nearest to but no larger in magnitude than `this`. See also `ceil`, `floor`, `round`.

Assertion Block

```

0.trunc() == 0;
1.4.trunc() == 1;    1.5.trunc() == 1;    1.6.trunc() == 1;
(-1.4).trunc() == -1; (-1.5).trunc() == -1; (-1.6).trunc() == -1;
inf.trunc() == inf; (-inf).trunc() == -inf;
nan.trunc().isNaN;

```

- `'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];

```

Assertion Block

22.21 Float.limits

This singleton handles various limits related to the `Float` objects.

22.21.1 Prototypes

- `Singleton`

22.21.2 Slots

- `digits`

Number of digits (in `radix` base) in the mantissa.

```
Float.limits.digits;
```

Assertion Block

- `digits10`

Number of digits (in decimal base) that can be represented without change.

```
Float.limits.digits10;
```

Assertion Block

- `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;
```

Assertion Block

- `max`

Maximum finite value.

```
Float.limits.max != Float.inf;
Float.limits.max * 2 == Float.inf;
```

Assertion Block

- `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;
```

Assertion Block

- `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);
```

Assertion Block

- `min`

Minimum positive normalized value.

```
0 != Float.limits.min;
```

Assertion Block

- `minExponent`

Minimum negative integer value for the exponent that generates a normalized floating-point number.

```
0 != Float.limits.radix ** Float.limits.minExponent;
```

Assertion Block

- **minExponent10**

Minimum negative integer value such that 10 raised to that power generates a normalized floating-point number.

```
0 != 10 ** Float.limits.minExponent10;
```

Assertion
Block

- **radix**

Base of the exponent of the representation.

```
Float.limits.radix == 2;
```

Assertion
Block

22.22 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.

22.22.1 Prototypes

- `Object`

22.22.2 Construction

The constructor expects a string as argument, whose syntax is similar to `printf`'s. It is detailed below.

```
FormatInfo.new("%+2.3d");
[00000001] %+2.3d
```

urbiscript
Session

A formatting pattern must one of the following (brackets denote optional arguments):

- `%rank%`
- `%[rank$] options spec`
- `%[rank$] options[spec] |`

where:

- *rank* is a non-null integer which denotes a positional argument: a means to output arguments in a different order.
- *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

When accepted, the format string is decoded, and its features are made available as separate slots of the `FormatInfo` object.

```
do (FormatInfo.new("%5$+#=#06.12X"))
{
    assert
    {
        rank      == 5;      // 5$
        prefix   == "+";    // +
        group    == " ";    // '
        alignment == 0;    // =
        alt      == true;   // #
        pad      == "0";    // 0
        width    == 6;     // 6
        precision == 12;   // .12
        uppercase == 1;    // X
        spec     == "x";    // X
    };
}!;
```

Formats that do not conform raise errors.

urbiscript
Session

```
FormatInfo.new("foo");
[00000001:error] !!! new: format: pattern does not begin with %: foo

FormatInfo.new("%20m");
[00000002:error] !!! new: format: invalid conversion type character: m

FormatInfo.new("%");
[00000003:error] !!! new: format: trailing '%'

FormatInfo.new("%ss");
[00062475:error] !!! new: format: spurious characters after format: s

FormatInfo.new("%.ss");
[00071153:error] !!! new: format: invalid width after '.': s

FormatInfo.new("%|-8.2f|%%");
[00034983:error] !!! new: format: spurious characters after format: %%
```

22.22.3 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;

"%5s" % 1 == "    1";
"%=5s" % 2 == " 2 ";
"%-5s" % 3 == "3  ";
```

Assertion
Block

- **alt**

Whether the “alternative” display is requested (#).

```
FormatInfo.new("%s") .alt == false;
FormatInfo.new("%#s").alt == true;

"%s" % 12.3 == "12.3";
"%#s" % 12.3 == "12.3000";
"%x" % 12 == "c";
"%#x" % 12 == "0xc";
```

Assertion
Block

- **group**

Separator to use for thousands as a `String`. Corresponds to the “`,` option. Currently produces no effect at all.

Assertion Block

```
FormatInfo.new("%s") .group == "";
FormatInfo.new("%'s").group == " ";
"%d" % 123456 == "%'d" % 123456 == "123456";
```

- **pad**

The padding character to use for alignment requests. Defaults to space.

Assertion Block

```
FormatInfo.new("%s") .pad == " ";
FormatInfo.new("%0s").pad == "0";

"%5s" % 1 == "    1";
"%05s" % 1 == "00001";
```

- **pattern**

The pattern given to the constructor.

Assertion Block

```
FormatInfo.new("%#'12.8s").pattern == "%#'12.8s";
```

- **precision**

When formatting a `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.

Assertion Block

```
FormatInfo.new("%s") .precision == 6;
FormatInfo.new("%23.3s").precision == 3;

"%f" % 12.3 == "12.300000";
"%.0f" % 12.3 == "12";
"%.2f" % 12.3 == "12.30";
"%.8f" % 12.3 == "12.30000000";
```

- **prefix**

The string to display before positive numbers. Defaults to empty.

Assertion Block

```
FormatInfo.new("%s") .prefix == "";
FormatInfo.new("% s").prefix == " ";
FormatInfo.new("%+s").prefix == "+";
```

- **rank**

In the case of a positional argument, its number, otherwise 0.

Assertion Block

```
FormatInfo.new("%s") .rank == 0;
FormatInfo.new("%2$s") .rank == 2;
FormatInfo.new("%03$s").rank == 3;
FormatInfo.new("%4%") .rank == 4;

"%3$s%2$s%2$s%1$s" % ["bar", "o", "f"]
== "%3%2%2%1%" % ["bar", "o", "f"] == "foobar";
```

Cannot be null.

urbscript Session

```
FormatInfo.new("%00$s").rank;
[00001243:error] !!! new: format: invalid positional argument: 00
```

- **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";
```

Assertion Block

- **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;
FormatInfo.new("%|D|") .uppercase == 1;
FormatInfo.new("%|d|") .uppercase == -1;
```

Assertion Block

- **width**

Width requested for alignment.

```
FormatInfo.new("%s") .width == 0;
FormatInfo.new("%10s") .width == 10;
FormatInfo.new("%-10s").width == 10;
FormatInfo.new("%8.2f").width == 8;
```

Assertion Block

22.23 Formatter

A *formatter* stores information of a format string like used in `printf` in the C library or in `boost::format`.

22.23.1 Prototypes

- `Object`

22.23.2 Construction

Formatters are created from format strings: they are split into regular strings and formatting patterns (`FormatInfo`), and stores them.

urbiscript
Session

```
Formatter.new("Name:%s, Surname:%s;");
[00000001] Formatter ["Name:", %s, "Surname:", %s, ";"]
```

All the patterns are introduced with the percent character (%), and they must conform to a specific syntax, detailed in the section on the construction of the `FormatInfo`. To denote the percent character instead of introducing a formatting-specification, use two percent characters.

urbiscript
Session

```
var f = Formatter.new("%10s level: %-4.1f%");
[00039525] Formatter [%10s, "level: ", %-4.1f, "%"]

for (var d: ["Battery" => 83.3, "Sound" => 60])
    echo (f % d.asList());
[00041133] ***      Battery level: 83.3%
[00041138] ***      Sound level: 60 %
```

Patterns can either all be non-positional (e.g., `%s%s`), or all positional (e.g., `%1$s%2$s`).

urbiscript
Session

```
Formatter.new("%s%s");
[00371506] Formatter [%s, %s]
Formatter.new("%1$s%2$s");
[00385602] Formatter [%1$s, %2$s]

Formatter.new("%1$s%$s");
[00409657:error] !!! new: format: cannot mix positional and non-positional\
arguments: %1$s vs. %s
```

22.23.3 Slots

- `asList`

Return the content of the *formatter* as a list of strings and `FormatInfo`.

```
Formatter.new("Name:%s, Surname:%s;").asList().asString()
== "[\"Name:\", %s, \", Surname:\", %s, \";\"]";
```

Assertion
Block

- `'%'(args)`

Use `this` as format string and `args` as the list of arguments, and return the result (a `String`).

This operator concatenates regular strings and the strings that are result of `asString` called on members of `args` with the appropriate `FormatInfo`.

```
Formatter.new("=>") % [] == ">";
Formatter.new("=> %s") % [1] == "> 1";
Formatter.new("Name:%s, Surname:%s;") % ["Foo", "Bar"]
== "Name:Foo, Surname:Bar;";
```

Assertion
Block

The arity of the `Formatter` (i.e., the number of expected arguments) and the size of `args` must match exactly.

urbiscript
Session

```
var f = Formatter.new("%s")|;
f % [];
[00000002:error] !!! %: format: too few arguments

f% ["foo", "bar"];
[00000004:error] !!! %: format: too many arguments
```

If *args* is not a [List](#), then the call is equivalent to calling `'%'([args])`.

```
Formatter.new("%06.3f") % Math.pi
== "03.142";
```

Assertion Block

Note that `String.'%'` provides a nicer interface to this operator:

```
"%06.3f" % Math.pi == "03.142";
```

Assertion Block

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;
```

urbiscript Session

22.24 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.

22.24.1 Prototypes

- [uobjects](#)
- [Tag.tags](#) (see [Tag](#))
- [Math](#)
- [System](#)
- [Object](#)

22.24.2 Slots

- [Barrier](#)
See [Barrier](#).
- [Binary](#)
See [Binary](#).
- [CallMessage](#)
See [CallMessage](#).
- [cerr](#)
A predefined stream for error messages. Strings sent to it are not escaped, contrary to regular streams (see [output](#) for instance).

urbiscript
Session

```
cerr << "Message";
[00015895:error] Message
cerr << "\"quote\"";
[00015895:error] "quote"
```

- [Channel](#)
See [Channel](#).
- [clog](#)
A predefined stream for log messages. Strings are output escaped.

urbiscript
Session

```
clog << "Message";
[00015895:clog] "Message"
```

- [Code](#)
See [Code](#).
- [Comparable](#)
See [Comparable](#).
- [cout](#)
A predefined stream for output messages. Strings are output escaped.

urbiscript
Session

```
cout << "Message";
[00015895:output] "Message"
cout << "\"quote\"";
[00015895:output] "\"quote\""
```

- **Date**
See [Date](#).
- **detach(*exp*)**
Bounce to [Control.detach](#), see [Control](#).
- **Dictionary**
See [Dictionary](#).
- **Directory**
See [Directory](#).
- **disown(*exp*)**
Bounce to [Control.disown](#), see [Control](#).
- **Duration**
See [Duration](#).
- **echo(*value*, *channel* = "")**
Bounce to [lobby.echo](#), see [Lobby.echo](#).

```
echo("111", "foo");
[00015895:foo] *** 111
echo(222, "");
[00051909] *** 222
echo(333);
[00055205] *** 333
```

urbiscript
Session

- **evaluate**

This [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 17.1](#)). 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).

```
import Global.*;
```

urbiscript
Session

```
(evaluate = "1+2;") == "1+2;";
evaluate == 3;
{ evaluate = "1+2;" | evaluate } == 3;
{ evaluate = "var x = 1;" | x } == 1;
```

Assertion
Block

Errors raise an exception.

```
Global.evaluate = "1/0;";
[00087671:error] !!! 1.1-3: /: division by 0
[00087671:error] !!!      called from: updateSlot
```

urbiscript
Session

- **Event**
See [Event](#).
- **Exception**
See [Exception](#).
- **Executable**
See [Executable](#).

- [external](#)

An system object used to implement UObject support in urbiscript.

- [false](#)

See [Section 22.3.1](#).

- [File](#)

See [File](#).

- [Finalizable](#)

See [Finalizable](#).

- [Float](#)

See [Float](#).

- [FormatInfo](#)

See [FormatInfo](#).

- [Formatter](#)

See [Formatter](#).

- [getProperty\(slotName, propName\)](#)

This wrapper around [Object.getProperty](#) is actually a by-product of the existence of the [evaluate UVar](#).

- [Global](#)

See [Global](#).

- [Group](#)

See [Group](#).

- [InputStream](#)

See [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.

urbiscript
Session

```
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](#).
- **Kernel1**
See [Kernel1](#).
- **Lazy**
See [Lazy](#).
- **List**
See [List](#).
- **Loadable**
See [Loadable](#).
- **Lobby**
See [Lobby](#).
- **Math**
See [Math](#).
- **methodToFunction(*name*)**
Create a function from the method *name* so that calling the function which arguments (*a*, *b*, ...) is that same as calling *a.name(b, ...)*.

urbiscript
Session

```

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](#).
- **nil**
See [nil](#).
- **Object**
See [Object](#).
- **Orderable**
See [Orderable](#).

- **OutputStream**
See [OutputStream](#).
- **Pair**
See [Pair](#).
- **Path**
See [Path](#).
- **Pattern**
See [Pattern](#).
- **persist(*exp*)**
Bounce to [Control.persist](#), see [Control](#).
- **Position**
See [Position](#).
- **Primitive**
See [Primitive](#).
- **Process**
See [Process](#).
- **Profile**
See [Profile](#).
- **PseudoLazy**
See [PseudoLazy](#).
- **PubSub**
See [PubSub](#).
- **RangeIterable**
See [RangeIterable](#).
- **Regexp**
See [Regexp](#).
- **Semaphore**
See [Semaphore](#).
- **Server**
See [Server](#).
- **Singleton**
See [Singleton](#).
- **Socket**
See [Socket](#).
- **String**
See [String](#).
- **System**
See [System](#).
- **Tag**
See [Tag](#).

- **Timeout**

See [Timeout](#).

- **TrajectoryGenerator**

See [TrajectoryGenerator](#).

- **Triplet**

See [Triplet](#).

- **true**

See [Section 22.3.1](#).

- **Tuple**

See [Tuple](#).

- **UObject**

See [UObject](#).

- **uobjects**

An object whose slots are all the [UObject](#) bound into the system. See [uobjects](#).

- **UValue**

See [UValue](#).

- **UVar**

See [UVar](#).

- **void**

See [void](#).

- **wall(*value*, *channel* = "")**

Bounce to [lobby.wall](#), see [Lobby](#).

urbiscript
Session

```
wall("111", "foo");
[00015895:foo] *** 111
wall(222, "");
[00051909] *** 222
wall(333);
[00055205] *** 333
```

- **warn(*message*)**

Issue *message* on [Channel.warning](#).

urbiscript
Session

```
warn("cave canem");
[00015895:warning] !!! cave canem
```

22.25 Group

A transparent means to send messages to several objects as if they were one.

22.25.1 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
```

urbiscript
Session

22.25.2 Prototypes

- [RangeIterable](#)
- [Comparable](#)

22.25.3 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"]
```

22.25.4 Slots

- '`<<` (*member*)
- Syntactic sugar for [add](#).

- `'==' (that)`

Whether `this.members == that.members`.

Assertion Block

```
Group.new() == Group.new();
Group.new(1, [2], "foo") == Group.new(1, [2], "foo");
Group.new(1, 2) != Group.new(2, 1);
Group.new(1) != Group.new(2);
```

- `add(member, ...)`

Add members to `this` group, and return `this`.

```
var g = Group.new(1, 2);
g.add(3, 4) === g;
g.members == [1, 2, 3, 4];
```

Assertion Block

- `asString`

Report the `asString` of the members.

```
Group.new(1, 2).asString() == "Group [1, 2]";
```

Assertion Block

- `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. See [Object.getProperty](#).

urbiscript Session

```
class C
{
  var val = 0;
};

var a = C.new(); var b = C.new();
var g = Group.new() << a << b;

g.val->prop;
[00010640:error] !!! property lookup failed: val->prop

a.val->prop = 42;
g.val->prop;
[00010640:error] !!! property lookup failed: val->prop

b.val->prop = 51;
assert
{
  g.val->prop == Group.new(42, 51);
};
```

- **hasProperty(*slot*, *prop*)**

Bounce to the members and return a group for the results. See [Object.hasProperty](#).

urbiscript
Session

```
class C
{
    var val = 0;
};

var a = C.new(); var b = C.new();
var g = Group.new() << a << b;

assert
{
    g.hasProperty("val", "prop") == Group.new(false, false);
    a.val->prop = 21;
    g.hasProperty("val", "prop") == Group.new(true, false);
    b.val->prop = 42;
    g.hasProperty("val", "prop") == Group.new(true, true);
};
```

- **hasSlot(*name*)**

True if and only if all the members have the slot.

Assertion
Block

```
var g = Group.new(1, 2);

!g.hasSlot("foo");
g.hasSlot("+");
g + 1 == Group.new(2, 3);
```

- **remove(*member*, ...)**

Remove members from `this` group, and return `this`. Non-existing members are silently ignored.

Assertion
Block

```
var g = Group.new(1, 2, 1);
g.remove(1, 3) === g == Group.new(2);
g.remove(2) === g == Group.new();
```

- **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, and return a group for the collected result. See [Object.setProperty](#).

urbiscript
Session

```
class C
{
    var val = 0;
};

var g = Group.new() << C.new() << C.new();

assert
{
    (g.val->prop = 31) == Group.new(31, 31);
};
```

- **updateSlot(*name*, *value*)**

Bounced to the members so that `this.name = value` actually updates the value of the slot `name` in the group members.

22.26 Hash

A *hash* is a condensed, easily comparable representation of another value. They are mainly used to map [Dictionary](#) keys to values.

Equal objects must always have the same hash. Different objects should, as much as possible, have different hashes.

22.26.1 Prototypes

- [Object](#)

22.26.2 Construction

Objects can be hashed with [Object.hash](#).

```
Object.new().hash().isA(Hash);
```

Assertion Block

22.26.3 Slots

- [asFloat](#)

A Float value equivalent to the Hash object. Two hashes have the same Float representation if and only if they are equal.

```
var h1 = Object.new().hash();
var h2 = Object.new().hash();
assert
{
    h1.asFloat() == h1.asFloat();
    h1.asFloat() != h2.asFloat();
};
```

urbiscript Session

- [combine\(that\)](#)

Combine *that*'s hash with *this*, and return *this*. This is used to hash composite objects based on more primitive object hashes. For instance, an object with two slots could be hashed by hashing its first one, and combining the second one.

```
class C
{
    function init(var a, var b)
    {
        var this.a = a;
        var this.b = b;
    };

    function hash()
    {
        this.a.hash().combine(b)
    };
}

assert
{
    C.new(0, 0).hash() == C.new(0, 0).hash();
    C.new(0, 0).hash() != C.new(0, 1).hash();
};
```

urbiscript Session

22.27 InputStream

InputStreams are used to read (possibly binary) files by hand. [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.

22.27.1 Prototypes

- [Stream](#)

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.

22.27.2 Construction

An InputStream is a reading-interface to a file, so its constructor requires a [File](#).

urbiscript
Session

```
InputStream.new(0);
[00000919:error] !!! new: argument 1: unexpected 0, expected a File

File.save("file.txt", "1\n2\n");
var i1 = InputStream.new(File.new("file.txt"));
[00001208] InputStream_0x1046d16e0
```

Cloning a closed [Stream](#) is valid, but it is forbidden to clone an opened one.

urbiscript
Session

```
var i2 = InputStream.clone().new(File.new("file.txt"));
[00001211] InputStream_0x1045f6760

i1.clone();
[00001288:error] !!! clone: cannot clone opened Streams
```

Do not forget to close the streams you opened ([Section 22.64.2](#)).

urbiscript
Session

```
i1.close();
i2.close();
```

22.27.3 Slots

- [asList](#)

Get the remainder as a [List](#), and an empty [List](#) if there is nothing left in the stream. Raise an error if the file is closed.

urbiscript
Session

```
var f = "file.txt" |
File.save(f, "1\n2\n") |
var i = InputStream.new(File.new(f)) |
assert (["1", "2"] == i.asList());
assert ([] == i.asList());
i.close();
i.asList();
[00000001:error] !!! asList: stream is closed
```

- [content](#)

Get the remainder as a [String](#), or an empty [String](#) if there is nothing left in the stream. Raise an error if the file is closed.

urbiscript
Session

```
var f = "file.txt" |
File.save(f, "1\000\n2\n") |
var i = InputStream.new(File.new(f)) |
assert (File.new(f).content.data == i.content());
assert ("" == i.content());
i.close();
i.content();
[00000001:error] !!! content: stream is closed
```

- **get**

Get the next available byte as a `Float`, or `nil` if the end of file was reached. Raise an error if the file is closed.

```
File.save("file.txt", "1\n2\n") |
var i = InputStream.new(File.new("file.txt")) |
var x;
while (!(x = i.get()).isNil)
  cout << x;
[00000001:output] 49
[00000002:output] 10
[00000003:output] 50
[00000004:output] 10

i.close();

i.get();
[00000005:error] !!! get: stream is closed
```

urbiscript
Session

- **getChar**

Get the next available byte as a `String`, or `nil` if the end of file was reached. Raise an error if the file is closed.

```
File.save("file.txt", "1\n2\n") |
var i = InputStream.new(File.new("file.txt")) |
var x;
while (!(x = i.getChar()).isNil)
  cout << x;
[00000001:output] "1"
[00000002:output] "\n"
[00000003:output] "2"
[00000004:output] "\n"

i.close();

i.getChar();
[00000005:error] !!! getChar: stream is closed
```

urbiscript
Session

- **getLine**

Get the next available line as a `String`, or `nil` if the end of file was reached. The end-of-line characters are trimmed. Raise an error if the file is closed.

```
File.save("file.txt", "1\n2\n") |
var i = InputStream.new(File.new("file.txt")) |
var x;
while (!(x = i.getLine()).isNil)
  cout << x;
[00000001:output] "1"
[00000002:output] "2"

i.close();

i.getLine();
```

urbiscript
Session

```
[00000005:error] !!! getLine: stream is closed
```

22.28 IoService

An `IoService` is used to manage the various operations of a set of `Socket`.

All `Socket` and `Server` are by default using the default `IoService` which is polled regularly by the system.

22.28.1 Example

Using a different `IoService` is required if you need to perform synchronous read operations.

The `Socket` must be created by the `IoService` that will handle it using its `makeSocket` function.

urbiscript
Session

```
var io = IoService.new()|;
var s = io.makeSocket()|;
```

You can then use this socket like any other.

urbiscript
Session

```
// 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/pollFor/pollOneFor` functions of `io`.

urbiscript
Session

```
sleep(200ms);
s.isConnected(); // Nothing was received yet
[00000001] true
io.poll();
[00000002] *** received something
sleep(200ms);
```

22.28.2 Prototypes

- `Object`

22.28.3 Construction

A `IoService` is constructed with no argument.

22.28.4 Slots

- `makeServer`
Create and return a new `Server` using this `IoService`.
- `makeSocket`
Create and return a new `Socket` using this `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.

22.29 Job

Jobs are independent threads of executions. Jobs can run concurrently. They can also be managed using [Tags](#).

22.29.1 Prototypes

- [Object](#)
- [Traceable](#)

22.29.2 Construction

A Job is typically constructed via [Control.detach](#), [Control.disown](#), or [Code.spawn](#).

urbiscript
Session

```
detach(sleep(10));
[00202654] Job<shell_7>

disown(sleep(10));
[00204195] Job<shell_8>

function () { sleep(10) }.spawn(false);
[00274160] Job<shell_9>
```

22.29.3 Slots

- [asJob](#)
Return `this`.

Assertion
Block

```
Job.current.asJob() === Job.current;
```

- [asString](#)

The string `Job<name>` where `name` is the name of the job.

Assertion
Block

```
Job.current.asList() == "Job<shell>";
var a = function () { sleep(1) }.spawn(false);
a.asList() == "Job<" + a.name + ">";
```

- [backtrace](#)

The current backtrace of the job as a [List](#) of [StackFrames](#) starting from innermost, to outermost. Uses [Traceable.backtrace](#).

urbiscript
Session

```
//#push 100 "foo.u"
function innermost () { Job.current.backtrace ||;
function inner () { innermost() }||;
function outer () { inner() }||;
function outermost () { outer() }||;
echoEach(outermost());
[01234567] *** foo.u:100.25-45: backtrace
[00001732] *** foo.u:101.25-35: innermost
[00001732] *** foo.u:102.25-31: inner
[00001732] *** foo.u:103.25-31: outer
[00001732] *** foo.u:104.10-20: outermost
//#pop

//#push 1 "file.u"
var s = detach(sleep(1))|;
// Leave some time for s to be started.
sleep(100ms);
assert
{
```

```

    s.backtrace[0].asString() == "file.u:1.16-23: sleep";
    s.backtrace[1].asString() == "file.u:1.9-24: detach";
};

// Leave some time for sleep to return.
sleep(1);
assert
{
    s.backtrace.size == 0;
};
//#pop

```

In the case of events, the backtrace is composite: the bottom part corresponds to the location of the event handler, and the top part is the location of the event emission. The event emission is denoted by `Event.emit` rather than `!`, and its reception by `Event.onEvent` rather than `?` or `at`.

urbiscript
Session

```

//#push 1 "file.u"
var e = Event.new();
function showBacktrace(var where)
{
    echo("===== " + where)|
    echoEach(Job.current.backtrace);
}|;
at (e?)
    showBacktrace("at-enter")
onleave
    showBacktrace("at-leave");
e!;
[00000647] *** ===== at-enter
[01234567] *** file.u:5.12-32: backtrace
[00000647] *** file.u:8.3-27: showBacktrace
[00000647] *** file.u:11.1: emit
[00000647] *** ---- event handler backtrace:
[00000647] *** file.u:7.1-10.27: onEvent
[00000647] *** ===== at-leave
[01234567] *** file.u:5.12-32: backtrace
[00000647] *** file.u:10.3-27: showBacktrace
[00000647] *** file.u:11.1: emit
[00000647] *** ---- event handler backtrace:
[00000647] *** file.u:7.1-10.27: onEvent


//#push 1 "file.u"
var f = Event.new();
function watchEvent()
{
    at (f?)
        showBacktrace("at-enter")
    onleave
        showBacktrace("at-leave");
}|;
function sendEvent()
{
    f!;
}|;
watchEvent();
sendEvent();
[00000654] *** ===== at-enter
[01234567] *** file.u:5.12-32: backtrace
[00000654] *** file.u:5.5-29: showBacktrace
[00000654] *** file.u:11.3: emit
[00000654] *** file.u:14.1-11: sendEvent
[00000654] *** ---- event handler backtrace:
[00000654] *** file.u:4.3-7.29: onEvent

```

```
[00000654] *** file.u:13.1-12: watchEvent
[00000654] *** ===== at-leave
[01234567] *** file.u:5.12-32: backtrace
[00000654] *** file.u:7.5-29: showBacktrace
[00000654] *** file.u:11.3: emit
[00000654] *** file.u:14.1-11: sendEvent
[00000654] *** ---- event handler backtrace:
[00000654] *** file.u:4.3-7.29: onEvent
[00000654] *** file.u:13.1-12: watchEvent
```

- **clone**

Cloning a job is forbidden.

- **current**

The [Job](#) in charge of executing this “thread” of code.

urbiscript
Session

```
var j = Job.current!;
assert { j.isA(Job) };

var j1; var j2;

// Both sequential compositions use the same job: the current one.
{ j1 = Job.current; j2 = Job.current }|;
assert { j1 == j2; j1 == j };

{ j1 = Job.current | j2 = Job.current }|;
assert { j1 == j2; j1 == j };

// Concurrency requires several jobs: j1 and j2 are different.
// 
// As an optimization, the current job is used for one of these
// jobs. This is an implementation detail, do not rely on it.
{ j1 = Job.current & j2 = Job.current }|;
assert { j1 != j2; j1 == j; j2 != j };

{ j1 = Job.current, j2 = Job.current }|;
assert { j1 != j2; j1 == j; j2 == j };
```

- **dumpState**

Pretty-print the state of the job.

urbiscript
Session

```
//#push 1 "file.u"
var t = detach(sleep(1))|;
// Leave some time for s to be started.
sleep(100ms);
t.dumpState();
[00004295] *** Job: shell_14
[00004295] *** State: sleeping
[00004295] *** Time Shift: 0.003ms
[00004295] *** Tags:
[00004295] *** Tag<Lobby_3>
[00004297] *** Backtrace:
[00004297] ***     file.u:1.16-23: sleep
[00004297] ***     file.u:1.9-24: detach
//#pop
```

- **jobs**

All the existing jobs as a [List](#).

```
Job.jobs.isA(List);
Job.current in Job.jobs;
```

Assertion
Block

- **name**

The name of the job as a [String](#).

urbiscript
Session

```
Job.current.name;
[00004293] "shell"
detach(sleep(1)).name;
[00004297] "shell_15"
```

- **interruptible**

Return a boolean telling if the job is interruptible (see [System.nonInterruptible](#))

- **frozen**

Return true if the job is frozen.

- **resetStats**

Reinitialize the [stats](#) computation.

- **stats**

Return a [Dictionary](#) containing information about the execution cycles of Urbi. This is an internal feature made for developers, it might be changed without notice. See also [resetStats](#).

urbiscript
Session

```
var j = detach({
    // Each ';' increments the Cycles with their execution time.
    var i = 0;
    {
        // Increment the Waiting time.
        waituntil(i == 1);
        // Increment the Sleeping time.
        sleep(100ms);
        i = 2;
    }, // Will fork and join a child job.

    sleep(200ms);
    i = 1;
    waituntil(i != 1);

    // Stop breaks the workflow of each job tagged with the tag.
    var t = Tag.new();
    t: t.stop(21);
});|;
j.waitForTermination();
var stats = j.stats|;

Float.epsilonTilde = 0.01 |;
assert
{
    0 < stats["Cycles"];
    stats["CyclesMin"] <= stats["CyclesMean"] <= stats["CyclesMax"];
    stats["WaitingMin"] <= stats["WaitingMean"] <= stats["WaitingMax"] ~= 100ms;
    stats["SleepingMin"] <= stats["SleepingMean"] <= stats["SleepingMax"] ~= 200ms;
    stats["Fork"] == stats["Join"] == 1;
    stats["WorkflowBreak"] == 1;

    stats["TerminatedChildrenWaitingMean"] ~= 200ms;
    stats["TerminatedChildrenSleepingMean"] ~= 100ms;
};
```

- **status**

A [String](#) that describes the current status of the job (starting, running, ...), and its properties (frozen, ...).

Assertion
Block

```
var t = detach(sleep(10));
Job.current.status == "running (current job)";
t.status == "sleeping" ;
```

- **tags**

The list of [Tags](#) that manage this job.

- **terminate**

Kill this job.

```
var r = detach({ sleep(1s); echo("done") })|;
assert (r in Job.jobs);
r.terminate();
assert (r not in Job.jobs);
sleep(2s);
```

urbiscript
Session

- **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
```

urbiscript
Session

- **waitForTermination**

Wait for the job to terminate before resuming execution of the current one. If the job has already terminated, return immediately.

22.30 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.

22.30.1 Prototypes

- [Singleton](#)

22.30.2 Construction

Since it is a [Singleton](#), you are not expected to build other instances.

22.30.3 Slots

- [commands](#)
Ignored for backward compatibility.
- [connections](#)
Ignored for backward compatibility.
- [copy\(*binary*\)](#)
Obsolete syntax for *binary*.[copy](#), see [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
```

urbiscript
Session

- [devices](#)
Ignored for backward compatibility.
- [events](#)
Ignored for backward compatibility.
- [functions](#)
Ignored for backward compatibility.
- [isvoid\(*obj*\)](#)
Obsolete syntax for *obj*.[isVoid](#), see [Object](#).
- [noop](#)
Do nothing. Use {} instead.
- [ping](#)
Return time verbosely, see [System](#).

```
Kernel1.ping;
[00000421] *** pong time=0.12s
```

urbiscript
Session

- **reset**
Ignored for backward compatibility.
- **runningcommands**
Ignored for backward compatibility.
- **seq(*number*)**
Obsolete syntax for *number*.**seq**, see [Float](#).
- **size(*list*)**
Obsolete syntax for *list*.**size**, see [List](#).

Assertion Block

```
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](#).

Assertion Block

```
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.

22.31 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 22.4](#).

22.31.1 Examples

22.31.1.1 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
```

urbiscript
Session

22.31.1.2 Evaluating several times

Evaluating a lazy several times only makes sense with lazy arguments and call messages. See example with call messages in [Section 22.4.1.1](#).

22.31.2 Caching

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](#). 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. UnaryLazy simply takes the function as argument.
var p = UnaryLazy.new(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

```

22.31.3 Prototypes

- Comparable

22.31.4 Construction

Lazies are seldom instantiated manually. They are mainly created automatically when a lazy function call is made (see Section 21.3.4). 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.

urbiscript
Session

```

Lazy.new(closure () { /* Value to make lazy */ 0 });
[00000000] 0

```

22.31.5 Slots

- `'=='` (*that*)

Whether `this` and `that` are the same source code and value (an not yet evaluated 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 });

```

Assertion
Block

urbiscript
Session

```
{  
    var l1 = Lazy.new(closure () { 1 + 1 });  
    var l2 = Lazy.new(closure () { 1 + 1 });  
    assert (l1 == l2);  
    l1.eval();  
    assert (l1 != l2);  
    l2.eval();  
    assert (l1 == l2);  
};
```

- **asString**

The conversion to `String` of the body of a non-evaluated argument.

Assertion Block

```
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.

22.32 List

Lists implement possibly-empty ordered (heterogeneous) collections of objects.

22.32.1 Prototypes

- [Container](#)

- [RangeIterable](#)

Therefore lists also support [RangeIterable.all](#) and [RangeIterable.any](#).

Assertion Block

```
// Are all elements positive?
! [-2, 0, 2, 4].all(function (e) { 0 < e });
// Are all elements even?
[-2, 0, 2, 4].all(function (e) { e % 2 == 0 });

// 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) { 0 < e });
```

- [Orderable](#)

22.32.2 Construction

Lists can be created with their literal syntax: a possibly empty sequence of expressions in square brackets, separated by commas. Non-empty lists may end with a comma ([Section 21.1.6.5](#)).

urbiscript Session

```
[]; // The empty list
[00000000] []
[1, "2", [3,],];
[00000000] [1, "2", [3]]
```

However, `new` can be used as expected.

urbiscript Session

```
List.new();
[00000001] []
[1, 2, 3].new();
[00000002] [1, 2, 3]
```

22.32.3 Slots

- `'*' (n)`

Return the target, concatenated n times to itself.

Assertion Block

```
[0, 1] * 0 == [];
[0, 1] * 3 == [0, 1, 0, 1, 0, 1];
```

n must be a non-negative integer.

urbiscript Session

```
[0, 1] * -2;
[00000063:error] !!! *: argument 1: expected non-negative integer: -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”.

urbiscript Session

```
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.

Assertion Block

```
[0, 1] + [2, 3] == [0, 1, 2, 3];
[] + [2, 3] == [2, 3];
[0, 1] + [] == [0, 1];
[] + [] == [];
```

The target is left unmodified (contrary to **' $+ =$ '**).

Assertion Block

```
var l = [1, 2, 3];
l + l == [1, 2, 3, 1, 2, 3];
l == [1, 2, 3];
```

- **' $+ =$ '** (*that*)

Concatenate the contents of the List *that* to **this**, and return **this**. This function modifies its target, contrary to **'+'**. See also **' $<<$ '**.

Assertion Block

```
var l = [];
var alias = l;

(l += [1, 2]) == l;
l == [1, 2];
(l += [3, 4]) == l;
l == [1, 2, 3, 4];
alias == [1, 2, 3, 4];
```

- **'-** (*other*)

Return the target without the elements that are equal to any element in the *other* list.

Assertion Block

```
[0, 1, 0, 2, 3] - [1, 2] == [0, 0, 3];
[0, 1, 0, 1, 0] - [1, 2] == [0, 0, 0];
```

- **'<'** (*that*)

Whether **this** is less than the *that* List. This is the lexicographic comparison: **this** is “less than” *that* if, from left to right, one of its member is “less than” the corresponding member of *that*:

Assertion Block

```
[0, 0, 0] < [0, 0, 1];    !([0, 0, 1] < [0, 0, 0]);
[0, 1, 2] < [0, 2, 1];    !([0, 2, 1] < [0, 1, 2]);

[1, 1, 1] < [2];          !([2] < [1, 1, 1]);

!([0, 1, 2] < [0, 1, 2]);
```

or *that* is a prefix (strict) of **this**:

Assertion Block

```
[] < [0];           !(      [0] < []);
[0, 1] < [0, 1, 2]; !([0, 1, 2] < [0, 1]);
!([0, 1, 2] < [0, 1, 2]);
```

Since List derives from **Orderable**, the other order-based operators are defined.

Assertion Block

```
[] <= [];
[] <= [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, 1, 2] > [];
    !(0, 1, 2) > [0, 1, 2];
    [0, 1, 2] > [0, 0, 2];

```

- '`<<`' (*that*)

A synonym for `insertBack`.

- '`==`' (*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]);

```

Assertion Block

- '`[]' (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";
    ["0", "1", "2"][2] == "2";
};

["0", "1", "2"][3];
[00007061:error] !!! []: invalid index: 3

assert
{
    ["0", "1", "2"][-1] == "2";
    ["0", "1", "2"][-3] == "0";
};
["0", "1", "2"][-4];
[00007061:error] !!! []: invalid index: -4

```

urbiscript Session

- '`[]=' (index, value)`

Assign *value* to the element of the target at the given *index*.

```

var f = [0, 1, 2];
assert
{
    (f[1] = 42) == 42;
    f == [0, 42, 2];
};

for (var i: [0, 1, 2])
    f[i] = 10 * f[i];
assert (f == [0, 420, 20]);

```

urbiscript Session

- `asTree`

Display the content of the List as a tree representation.

```

echo("simple list:" + ["a", "b", ["d", "e", "f", "g"]].asTree());
[00000004] *** simple list:
[
  a,
  b,
  [
    d,
    e,
    f,
    g,
  ]
]

```

urbiscript Session

```
]
echo("list with dictionary:" +
  ["a", "b", {"c" => ["d", "e"], "f" => "g"}].asTree());
[00000005] *** list with dictionary:
[
  a,
  b,
  [
    c =>
    [
      d,
      e,
    ]
    f => g,
  ]
]
```

- **append(*that*)**

Deprecated alias for '`+ =`'.

urbiscript
Session

```
var one = [1];
one.append(["one", [1]]);
[00000005:warning] !!! 'list.append(that)' is deprecated, use 'list += that'
[00000005] [1, "one", [1]]
```

- **argMax(*fun* = `function(a, b) { a < b }`)**

The index of the (leftmost) “largest” member based on the comparison function *fun*.

Assertion
Block

```
[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.

urbiscript
Session

```
[] .argMax();
[00000007:error] !!! argMax: list cannot be empty
```

- **argMin(*fun* = `function(a, b) { a < b }`)**

The index of the (leftmost) “smallest” member based on the comparison function *fun*.

Assertion
Block

```
[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.

urbiscript
Session

```
[] .argMin();
[00000011:error] !!! argMin: list cannot be empty
```

- **asBool**

Whether not empty.

Assertion
Block

```
[] .asBool() == false;
[1] .asBool() == true;
```

- **asList**

Return the target.

```
var l = [0, 1, 2];
l.asList() === l;
```

Assertion Block

- **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\"]";
```

Assertion Block

- **back**

The last element of the target. An error if the target is empty.

```
assert([0, 1, 2].back() == 2);
[] .back();
[00000017:error] !!! back: cannot be applied onto empty list
```

urbiscript Session

- **clear**

Empty the target, return it.

```
var l = [0, 1, 2];
l.clear() === l == [];
l.clear() === l == [];
```

Assertion Block

- **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
```

urbiscript Session

- **'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
```

urbiscript Session

- **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
```

urbiscript Session

- **empty**

Whether the target is empty.

Assertion Block

```
[] .empty;
! [1] .empty;
```

- **filter(*fun*)**

The list of all the members of the target that verify the predicate *fun*.

urbiscript
Session

```
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.

Assertion
Block

```
[] .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.

urbiscript
Session

```
assert([0, 1, 2].front() == 0);
[] .front();
[00000000:error] !!! front: cannot be applied onto empty list
```

- **has(*that*)**

Whether *that* equals one of the members.

```
[0, 1, 2] .has(1);
! [0, 1, 2] .has(5);
```

Assertion
Block

The infix operators **in** and **not in** use **has** (see Section 21.1.8.7).

```
1 in [0, 1];
2 not in [0, 1];
!(2 in [0, 1]);
!(1 not in [0, 1]);
```

Assertion
Block

- **hash**

A **Hash** object corresponding to this list value. Equal lists (in the sense of '`==`') have equal hashes, see **Object.hash**.

```
[] .hash().isA(Hash);

[] .hash() == [] .hash();
[1, "foo"] .hash() == [1, "foo"] .hash();
[0, 1] .hash() != [1, 0] .hash();
```

Assertion
Block

- **hasSame(*that*)**

Whether *that* is physically equal to one of the members.

```
var x = 1;

[0, x, 2].hasSame(x);
! [0, x, 2].hasSame(1);
```

Assertion Block

- **head**

Synonym for **front**.

```
assert([0, 1, 2].head() == 0);
[] .head();
[00000000:error] !!! head: cannot be applied onto empty list
```

urbiscript Session

- **insert(*where*, *what*)**

Insert *what* before the value at index *where*, return **this**.

```
var l = [0, 1];

l.insert(0, 10) === l == [10, 0, 1];
l.insert(2, 20) === l == [10, 0, 20, 1];
```

Assertion Block

The index must be valid, to insert past the end, use **insertBack**.

```
[] .insert(0, "foo");
[00044239:error] !!! insert: invalid index: 0
[1, 2, 3].insert(4, 30);
[00044339:error] !!! insert: invalid index: 4
```

urbiscript Session

Assertion Block

- **insertBack(*that*)**

Insert *that* at the end of the target, return **this**.

```
var l = [0, 1];

l.insertBack(2) === l;
l == [0, 1, 2];
```

Assertion Block

- **insertFront(*that*)**

Insert the given element at the beginning of the target. Return **this**.

```
var l = [0, 1];
l.insertFront(0) === l;
l == [0, 0, 1];
```

Assertion Block

- **insertUnique(*that*)**

If *that* is not in **this**, append it. Return **this**.

```
var l = [0, 1];

l.insertUnique(0) === l == [0, 1];
l.insertUnique(2) === l == [0, 1, 2];
```

Assertion Block

- **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.

Assertion Block

```
[] .keys == [];
["a", "b", "c"].keys == [0, 1, 2];
```

This allows uniform iteration over a [Dictionary](#) or a [List](#).

urbiscript Session

```
var l = ["a", "b", "c"]|;
var res = []|;
for (var k: l.keys)
    res << l[k];
assert (res == l);
```

- **map(*fun*)**

Apply the given functional value on every member, and return the list of results.

Assertion Block

```
[0, 1, 2, 3].map(function (v) { v % 2 == 0})
== [true, false, true, false];

[1, 2, 3].map (function (x) { x*2 })
== [2, 4, 6];
```

- **matchAgainst(*handler*, *pattern*)**

If *pattern* is a List of same size, use *handler* to match each member of **this** against the corresponding *pattern*. Return true if the match succeeded, false in other cases.

```
assert
{
    ([1, 2] = [1, 2]) == [1, 2];
};

([1, var a] = [1, 2]) == [1, 2];
[00004360] true
assert
{
    a == 2;
};

([var u, var v, var w] = [1, 2, 3]) == [1, 2, 3];
[00004376] true
assert
{
    [u, v, w] == [1, 2, 3];
};

[1, 2] = [2, 1];
[00005863:error] !!! pattern did not match

[1, var a] = [2, 1];
[00005864:error] !!! pattern did not match
[1, var a] = [1];
[00005865:error] !!! pattern did not match
[1, var a] = [1, 2, 3];
[00005865:error] !!! pattern did not match
```

urbiscript Session

- **max(*comp* = `function(a, b) { a < b }`)**

Return the “largest” member based on the comparison function *comp*.

Assertion Block

```
[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] !!! max: list cannot be empty
```

urbiscript
Session

The members must be comparable.

```
[0, 2, "a", 1].max();
[00000002:error] !!! max: argument 2: unexpected "a", expected a Float
```

urbiscript
Session

- **min(*co*; *p* = `function(a, b) { a < b }`)**

Return the “smallest” member based on the comparison function *comp*.

```
[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;
```

Assertion
Block

The list cannot be empty.

```
[] .min();
[00000001:error] !!! min: list cannot be empty
```

urbiscript
Session

- **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
```

urbiscript
Session

- **remove(*val*)**

Remove all elements from the target that are equal to *val*, return `this`.

```
var c = [0, 1, 0, 2, 0, 3];
c.remove(0) === c == [1, 2, 3];
c.remove(42) === c == [1, 2, 3];
```

Assertion
Block

- **removeBack**

Remove and return the last element of the target. An error if the target is empty.

Assertion
Block

```
var t = [0, 1, 2];
t.removeBack() == 2;
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*.

Assertion Block

```
var d = 1;
var e = [0, 1, d, 1, 2];
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.

Assertion Block

```
var g = [0, 1, 2];
g.removeFront() == 0;
g == [1, 2];

[] .removeFront();
[00000000:error] !!! removeFront: cannot be applied onto empty list
```

- **reverse**

The target with the order of elements inverted.

Assertion Block

```
[0, 1, 2].reverse() == [2, 1, 0];
```

- **size**

The number of elements in *this*.

Assertion Block

```
[0, 1, 2].size == 3;
[] .size == 0;
```

- **sort(*comp* = **function**(*a*, *b*) { *a* < *b* })**

A new List with the contents of *this*, sorted with respect to the *comp* comparison function.

Assertion Block

```
var l = [3, 0, -2, 1];
l.sort() == [-2, 0, 1, 3];
l      == [3, 0, -2, 1];

l.sort(function(a, b) {a.abs() < b.abs()})
      == [0, 1, -2, 3];
```

urbiscript Session

```
[2, 1].sort(1);
[00000001:error] !!! unexpected 1, expected a Executable
```

Following the **Garbage In, Garbage Out** principle, if *comp* is not a strict weak ordering (e.g., if *comp(a, b)* && *comp(b, a)* holds for some *a* and *b*), the result is meaningless.

urbiscript Session

```
[1, 2, 3].sort(function(a, b) { true });
[00011293] [2, 3, 1]
```

- **subset(*that*)**

Whether the members of *this* are members of *that*.

Assertion Block

```
[] .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**

`this` 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
```

urbiscript
Session

- **unique**

A new List composed of a single (based on `==` comparison) copy of all the members of `this` in no particular order.

```
[] .unique() == [];
[1] .unique() == [1];
[1, 1] .unique() == [1];
[1, 2, 3, 2, 1] .unique() == [1, 2, 3];
```

Assertion
Block

- **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];
```

Assertion
Block

22.33 Loadable

Loadable objects can be switched on and off — typically physical devices.

22.33.1 Example

The intended use is rather as follows:

```
urbiscript
Session
import gsrapি.*;
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_0x42364388

m.load;
[00000004] false

m.go(1);
[00000006] *** cannot advance, the motor is off

m.on();
[00000007] Motor_0x42364388

m.go(123);
m.val;
[00000009] 123
```

22.33.2 Prototypes

- [Object](#)

22.33.3 Construction

`Loadable` can be constructed, but it hardly makes sense. This object should serve as a prototype.

22.33.4 Slots

- `load`

The current status.

- `off(val)`

Set `load` to `false` and return `this`.

Assertion
Block

```
do (Loadable.new())
{
    assert
    {
        !load;
        off() === this;
        !load;
        on() === this;
    }
}
```

```

    load;
    off() === this;
    !load;
};

};


```

- **on(*val*)**

Set `load` to `true` and return `this`.

Assertion Block

```

do (Loadable.new())
{
    assert
    {
        !load;
        on() === this;
        load;
        on() === this;
        load;
    };
};


```

- **toggle**

Set `load` from `true` to `false`, and vice-versa. Return `val`.

Assertion Block

```

do (Loadable.new())
{
    assert
    {
        !load;
        toggle() === this;
        load;
        toggle() === this;
        !load;
    };
};


```

22.34 Lobby

A *lobby* is the local environment for each (remote or local) connection to an Urbi server.

22.34.1 Examples

Since every lobby is-a [Channel](#), one can use the methods of Channel.

```
lobby << 123;
[00478679] 123
lobby << "foo";
[00478679] "foo"
```

urbiscript
Session

22.34.2 Prototypes

- [Channel topLevel](#), an instance of [Channel](#) with an empty Channel name.

22.34.3 Construction

A lobby is implicitly created at each connection. At the top level, [this](#) is a *Lobby*.

urbiscript
Session

```
this.protos;
[00000001] [Lobby]
this.protos[0].protos;
[00000003] [Channel_0x4233b248]
```

Lobbies cannot be cloned, they must be created using [create](#).

urbiscript
Session

```
Lobby.new();
[00000177:error] !!! new: 'Lobby' objects cannot be cloned
Lobby.create();
[00000174] Lobby_0x126450
```

22.34.4 Slots

- [authors](#)

Credit the authors of Urbi SDK. See also [thanks](#) and [Section 1.4](#).

- [banner](#)

Internal. Display Urbi SDK banner.

- [bytesReceived](#)

The number of bytes that were “input” to [this](#). See also [receive](#).

```
var l = Lobby.create();
assert (l.bytesReceived == 0);

l.receive("123456789;");
[00000022] 123456789
assert (l.bytesReceived == 10);

l.receive("1234;");
[00000023] 1234
assert (l.bytesReceived == 15);
```

urbiscript
Session

- [bytesSent](#)

The number of bytes that were “output” by [this](#). See also [send](#) and [write](#).

urbiscript
Session

```
var l = Lobby.create();
assert (l.bytesSent == 0);

l.send("0123456789");
[00011988] 0123456789
// 22 = "[00011988] 0123456789\n".size.
assert (l.bytesSent == 22);

l.send("xx", "label");
[00061783:label] xx
// 20 = "[00061783:label] xx\n".size.
assert (l.bytesSent == 42);
```

- **connected**

Whether `this` is connected.

```
connected;
```

Assertion Block

- **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!"
```

urbiscript Session

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!"
```

urbiscript Session

- **copyright(*deep* = true)**

Display the copyright of Urbi SDK. Include copyright information about sub-components if `deep`.

- **create**

Instantiate a new Lobby.

```
Lobby.create().isA(Lobby);
```

- **echo(*value*, *channel* = "")**

Send *value*.asString to **this**, prefixed by the **String** *channel* name if specified. This is the preferred way to send informative messages (prefixed with ‘***’).

urbiscript
Session

```
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*.

urbiscript
Session

```
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**.

Assertion
Block

```
lobby in Lobby.instances;
Lobby in Lobby.instances;
```

- **license**

Display the end user license agreement of the Urbi SDK.

- **lobby**

The lobby of the current connection. This is typically **this**.

```
Lobby.lobby === this;
```

Assertion
Block

But when several connections are active (e.g., when there are remote connections), it can be different from the target of the call.

```
Lobby.create() | Lobby.create() |
for (var l : lobbies)
    assert (l.lobby == Lobby.lobby);
```

urbiscript
Session

- **onDisconnect(*lobby*)**

Event launched when **this** is disconnected. There is a single event instance for all the lobby, **Lobby.onDisconnect**, the disconnected lobby being passed as argument.

- **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** *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.

urbiscript
Session

```
Lobby.create().receive("12;");
[00478679] 12
```

- **remoteIP**

When `this` is connected to a remote server, it's Internet address.

- **send(*value*, *channel* = "")**

This is low-level routine. Send the `String value` to `this`, prefixed by the `String channel` name if specified.

```
lobby.send("111", "foo");
[00015895:foo] 111
lobby.send("222", "");
[00051909] 222
lobby.send("333");
[00055205] 333
```

urbiscript
Session

- **thanks**

Credit the contributors of Urbi SDK. See also [authors](#) and [Section 1.4](#).

- **wall(*value*, *channel* = "")**

Perform `echo(value, channel)` on all the existing lobbies (except Lobby itself).

```
Lobby.wall("111", "foo");
[00015895:foo] *** 111
```

urbiscript
Session

- **write(*value*)**

This is low-level routine. Send the `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!
```

urbiscript
Session

22.35 Location

This class aggregates two Positions and provides a way to print them as done in error messages.

22.35.1 Prototypes

- [Object](#)

22.35.2 Construction

Without argument, a newly constructed Location has its Positions initialized to the first line and the first column.

urbiscript
Session

```
Location.new();
[00000001] 1.1
```

With a Position argument *p*, the Location will clone the Position into the begin and end Positions.

urbiscript
Session

```
Location.new(Position.new("file.u",14,25));
[00000001] file.u:14.25
```

With two Positions arguments *begin* and *end*, the Location will clone both Positions into its own fields.

urbiscript
Session

```
Location.new(Position.new("file.u",14,25), Position.new("file.u",14,35));
[00000001] file.u:14.25-34
```

22.35.3 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);
  };
}
```

urbiscript
Session

- `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";
```

Assertion
Block

```

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);

```

Assertion Block

- **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);

```

Assertion Block

22.36 Logger

`Logger` is used to report information to the final user or to the developer. It allows to pretty print warnings, errors, debug messages or simple logs. `Logger` can also be used as `Tag` objects for it to handle nested calls indentation. A log message is assigned a category which is shown between brackets at beginning of lines, and a level which defines the context in which it has to be shown (see [Section 20.1.2](#)). Log level definition and categories filtering can be changed using environment variables defined in [Section 20.1.2](#).

22.36.1 Examples

The proper use of Loggers is to instantiate your own category, and then to use the operator `<<` for log messages, possibly qualified by the proper level (in increase order of importance: `dump`, `debug`, `trace`, `log`, `warn`, `err`):

```
var logger = Logger.new("Category")|;

logger.dump() << "Low level debug message"|;
// Nothing displayed, unless the debug level is set to DUMP.

logger.warn() << "something wrong happened, proceeding"|;
[ Category ] something wrong happened, proceeding

logger.err() << "something really bad happened!"|;
[ Category ] something really bad happened!
```

urbiscript
Session

You may also directly use the logger and passing arguments to these slots.

```
Logger.log("message", "Category") |;
[ Category ] message

Logger.log("message", "Category") :
{
    Logger.log("indented message", "SubCategory")
}|;
[ Category ] message
[ SubCategory ] indented message
```

22.36.2 Existing Categories

Urbi SDK comes with a number of built-in categories that all belong to one of the four following category families. Precise categories are provided for information, but there is no guarantee that these categories will be maintained, that their semantics will not change, or that they are all listed here.

Libport The Libport library.

Sched The coroutine library.

Serialize The serialization library.

Urbi Everything about Urbi SDK.

Urbi.Parser Bison parser traces.

Urbi.Scanner Flex scanner traces.

22.36.3 Prototypes

- **Tag**

22.36.4 Construction

`Logger` can be used as is, without being cloned. It is possible to clone `Logger` defining a category and/or a level of debug.

urbiscript
Session

```
Logger.log("foo");
[      Logger      ] foo
[00004702] Logger<Logger>

Logger.log("bar", "Category") |;
[      Category     ] bar

var l = Logger.new("Category2");
[00004703] Logger<Category2>

l.log("foo") |;
[      Category2    ] foo

l.log("foo", "ForcedCategory") |;
[      ForcedCategory ] foo
```

22.36.5 Slots

- `'<<'(object)`

Allow to use the `Logger` object as a [Channel](#). This slot can only be used if both category and level have been defined when cloning.

urbiscript
Session

```
l = Logger.new("Category", Logger.Levels.Log);
[00090939] Logger<Category>
l << "abc";
[      Category      ] abc
[00091939] Logger<Category>
```

- `categories`

A [Dictionary](#) of known categories, mapping their name to their activation status as a Boolean. Note that changing this dictionary has no influence over the set of active categories, see `enable` and `disable`.

Assertion
Block

```
var c = Logger.categories;
c.isA(Dictionary);
c["Urbi.Call"];
// The four category families used by Urbi SDK.
c.keys.map(function (var s) { s.split(".")[0] }).unique()
  == ["Libport", "Logger", "Runner", "Sched", "Serialize", "Urbi", "test"];
```

- `debug(message = "", category = category)`

Report a debug `message` of `category` to the user. It will be shown if the debug level is `Debug` or `Dump`. Return `this` to allow chained operations.

urbiscript
Session

```
// None of these are displayed unless the current level is at least DEBUG.
Logger.debug() << "debug 1"|;
Logger.debug("debug 2")|;
Logger.debug("debug 3", "Category2")|;
```

Assertion
Block

- `disable(pattern)`

Disable the categories that match the `pattern`, i.e., suppress logs about these categories independently of the current `level`. Affects existing and future categories. See also `enable`.

```

var c = Logger.new("Logger.Foo");
c << "Enabled";
// Enabled by default.
Logger.categories["Logger.Foo"];

// Disable any categories that starts with Logger.Foo, existing or not.
Logger.disable("Logger.Foo*").isVoid;
c << "Disabled";
!Logger.categories["Logger.Foo"];

// A new category, disabled at creation.
var c2 = Logger.new("Logger.Foo.Bar");
c2 << "Disabled";
!Logger.categories["Logger.Foo.Bar"];

```

- **dump(*message* = "", *category* = *category*)**

Report a debug *message* of *category* to the user. It will be shown if the debug level is **Dump**. Return **this** to allow chained operations.

urbiscript
Session

```

// None of these are displayed unless the current level is at least DEBUG.
Logger.dump() << "dump 1"|;
Logger.dump("dump 2")|;
Logger.dump("dump 3", "Category2")|;

```

- **enable(*pattern*)**

Enable the categories that match the *pattern*, i.e., suppress logs about these categories independently of the current **level**. Affects existing and future categories. See also **disable**.

urbiscript
Session

```

// Disable any categories that starts with Logger.Foo, existing or not.
Logger.disable("Logger.Foo*");
Logger.log("disabled", "Logger.Foo.Bar");
Logger.log("disabled", "Logger.Foo.Baz");

// Enable those that start with Logger.Foo.Bar.
Logger.enable("Logger.Foo.Bar*");
Logger.log("enabled", "Logger.Foo.Bar");
[   Logger.Foo.Bar      ] enabled
Logger.log("enabled", "Logger.Foo.Bar.Qux");
[   Logger.Foo.Bar.Qux  ] enabled
Logger.log("disabled", "Logger.Foo.Baz");

```

- **err(*message* = "", *category* = *category*)**

Report an error *message* of *category* to the user. Return **this** to allow chained operations.

- **init(*category*)**

Define the *category* of the new **Logger** object. If no category is given the new **Logger** will inherit the category of its prototype.

- **level**

The current level of verbosity (see **Levels**). Can be read and assigned to.

urbiscript
Session

```

// Logger is enabled by default.
Logger.categories["Logger"];
[00000001] true
// Log level is enabled by default.
Logger.level;
[00000002] Log
Logger.log("Logger enabled");
[   Logger          ] Logger enabled

```

```
// Disable it.
Logger.level = Logger.Levels.None;
Logger.level;
[00000003] None
Logger.log("Logger is disabled");

// Enable it back.
Logger.level = Logger.Levels.Log;
[00000004] Log
Logger.log("Logger enable again");
[      Logger      ] Logger enable again
```

- **Levels**

An [Enumeration](#) of the log levels defined in [Section 20.1.2](#).

```
Logger.Levels.values;
[00000001] [None, Log, Trace, Debug, Dump]
```

urbiscript
Session

- **log(*message* = "", *category* = category)**

Report a debug *message* of *category* to the user. It will be shown if debug is not disabled. Return [this](#) to allow chained operations.

```
Logger.log() << "log 1" |;
[      Logger      ] log 1

Logger.log("log 2") |;
[      Logger      ] log 2

Logger.log("log 3", "Category") |;
[      Category     ] log 3
```

urbiscript
Session

- **onEnter**

The primitive called when `Logger` is used as a `Tag` and is entered. This primitive only increments the indentation level.

- **onLeave**

The primitive called when `Logger` is used as a `Tag` and is left. This primitive only decrements the indentation level.

- **set(*specs*)**

Enable/disable the categories according to the *specs*. The *specs* is a comma-separated list of categories or globs (e.g., ‘`Libport`’, ‘`Urbi.*`’) with optional modifiers ‘+’ or ‘-’. If the modifier is a ‘+’ or if there is no modifier the category will be displayed. If the modifier is a ‘-’ it will be hidden.

Modifiers can be chained, accumulated from left to right: ‘`-*,+Urbi.*,-Urbi.At`’ will only display categories beginning with ‘`Urbi.`’ except ‘`Urbi.At`’, ‘`-Urbi.*`’ will display all categories except those beginning with ‘`Urbi.`’.

Actually, the first character defines the state for unspecified categories: ‘`Urbi`’ (or ‘`+Urbi`’) enables only the ‘`Urbi`’ category, while ‘`-Urbi`’ enables everything but ‘`Urbi`’. Therefore, ‘`+Urbi.*,-Urbi.At`’ and ‘`Urbi.*,-Urbi.At`’ are equivalent to ‘`-*,+Urbi.*,-Urbi.At`’.

```
var l1 = Logger.new("Logger") |;
var l2 = Logger.new("Logger.Sub") |;
var l3 = Logger.new("Logger.Sub.Sub") |;
function check(specs, s1, s2, s3)
{
    Logger.set(specs);
    assert
    {
```

urbiscript
Session

```

    Logger.categories["Logger"]      == s1;
    Logger.categories["Logger.Sub"]  == s2;
    Logger.categories["Logger.Sub.Sub"] == s3;
};

} |;

check("-L*",      false, false, false);
check("L*",       true,  true,  true);
check("-L*,+Lo*", true,  true,  true);
check("-L*,+Logger", true, false, false);

check("+Logger,-Logger.*", true, false, false);
check("-*,+Logger.*,-Logger.Sub.*", false, true, false);
check("+Logger*,-Logger.*,+Logger.Sub.*", true, false, true);

```

- `trace(message = "", category = category)`

Report a debug *message* of *category* to the user. It will be shown if the debug level is Trace, Debug or Dump. Return `this` to allow chained operations.

urbiscript
Session

```
// None of these are displayed unless the current level is at least TRACE.
Logger.trace() << "trace 1"|;
Logger.trace("trace 2")|;
Logger.trace("trace 3", "Category2")|;
```

- `warn(message = "", category = category)`

Report a warning *message* of *category* to the user. Return `this` to allow chained operations.

urbiscript
Session

```

Logger.warn() << "warn 1"|;
[     Logger      ] warn 1

Logger.warn("warn 2")|;
[     Logger      ] warn 2

Logger.warn("warn 3", "Category")|;
[     Category    ] warn 3

```

22.37 Math

This object is 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.

Math is a package, so you can use

urbiscript
Session

```
import Math.*;
```

to make its Slots visible.

22.37.1 Prototypes

- [Singleton](#)

22.37.2 Construction

Since it is a [Singleton](#), you are not expected to build other instances.

22.37.3 Slots

- `abs(float)`

Bounce to [Float.abs](#).

```
Math.abs(1) == Math.abs(-1) == 1;
Math.abs(0) == 0;
Math.abs(3.5) == Math.abs(-3.5) == 3.5;
```

Assertion
Block

- `acos(float)`

Bounce to [Float.acos](#).

```
Math.acos(0) == Float.pi/2;
Math.acos(1) == 0;
```

Assertion
Block

- `asin(float)`

Bounce to [Float.asin](#).

```
Math.asin(0) == 0;
```

Assertion
Block

Assertion
Block

- `atan(float)`

Bounce to [Float.atan](#).

```
Math.atan(1) ~= pi/4;
```

Assertion
Block

Assertion
Block

- `atan2(x, y)`

Bounce to `x.atan2(y)`.

```
Math.atan2(2, 2) ~= pi/4;
Math.atan2(-2, 2) ~= -pi/4;
```

Assertion
Block

Assertion
Block

- `ceil`

Bounce to [Float.ceil](#).

```
Math.ceil(1.4) == Math.ceil(1.5) == Math.ceil(1.6) == Math.ceil(2) == 2;
```

Assertion
Block

Assertion
Block

- `cos(float)`

Bounce to [Float.cos](#).

Assertion
Block

```
Math.cos(0) == 1;
Math.cos(pi) ~= -1;
```

- `exp(float)`

Bounce to [Float.exp](#).

Assertion Block

```
Math.exp(1) ~= 2.71828;
```

- `floor`

Bounce to [Float.floor](#).

Assertion Block

```
Math.floor(1) == Math.floor(1.4) == Math.floor(1.5) == Math.floor(1.6) == 1;
```

- `inf`

Bounce to [Float.inf](#).

Assertion Block

- `log(float)`

Bounce to [Float.log](#).

```
Math.log(1) == 0;
```

- `max(arg1, ...)`

Bounce to `[arg1, ...].max`, see [List.max](#).

Assertion Block

```
Math.max( 100, 20, 3 ) == 100;
Math.max("100", "20", "3") == "3";
```

- `min(arg1, ...)`

Bounce to `[arg1, ...].min`, see [List.min](#).

Assertion Block

```
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](#).

urbiscript Session

```
20.map(function (dummy) { Math.random(5) });
[00000000] [1, 2, 1, 3, 2, 3, 2, 2, 4, 4, 4, 1, 0, 0, 0, 3, 2, 4, 3, 2]
```

- `round(float)`

Bounce to [Float.round](#).

Assertion Block

```
Math.round(1) == Math.round(1.1) == Math.round(1.49) == 1;
Math.round(1.5) == Math.round(1.51) == 2;
```

- `sign(float)`

Bounce to [Float.sign](#).

Assertion Block

```
Math.sign(-2) == -1; Math.sign(0) == 0; Math.sign(2) == 1;
```

- `sin(float)`

Bounce to [Float.sin](#).

```
Math.sin(0) == 0;
Math.sin(pi) ~= 0;
```

Assertion
Block

- `sqr(float)`

Bounce to [Float.sqr](#).

```
Math.sqr(1.5) == Math.sqr(-1.5) == 2.25;
Math.sqr(16) == Math.sqr(-16) == 256;
```

Assertion
Block

- `sqrt(float)`

Bounce to [Float.sqrt](#).

```
Math.sqrt(4) == 2;
```

Assertion
Block

- `random(float)`

Bounce to [Float.random](#).

Assertion
Block

- `tan(float)`

Bounce to [Float.tan](#).

```
Math.tan(pi/4) ~= 1;
```

Assertion
Block

- `trunc(float)`

Bounce to [Float.trunc](#).

```
Math.trunc(1) == Math.trunc(1.1) == Math.trunc(1.49) == Math.trunc(1.5)
== Math.trunc(1.51)
== 1;
```

22.38 Matrix

22.38.1 Prototypes

- Object

22.38.2 Construction

The `init` function is overloaded, and its behavior depends on the number of arguments and their types.

When there is a single argument, it can either be a List, or another Matrix.

If it's a List of “Vectors and/or Lists of Floats”, then they must have the same sizes and constitute the rows.

```
urbiscript
Session
var listList      = Matrix.new([          [0, 1],          [0, 1] ]);
[00071383] Matrix([
  [0, 1],
  [0, 1]])
var listVector   = Matrix.new([          [0, 1],  Vector.new([0, 1])]);
var vectorList   = Matrix.new([Vector.new([0, 1]),          [0, 1]]);
var vectorVector = Matrix.new([Vector.new([0, 1]), Vector.new([0, 1])]);

assert
{
  listList == listVector;
  listList == vectorList;
  listList == vectorVector;
};
Matrix.new([          [0],          [1, 2]]);
[00000030:error] !!! new: expecting rows of size 1, got size 2 for row 2
Matrix.new([Vector.new([0]),          [1, 2]]);
[00000056:error] !!! new: expecting rows of size 1, got size 2 for row 2
Matrix.new([          [0], Vector.new([1, 2])]);
[00000071:error] !!! new: expecting rows of size 1, got size 2 for row 2
Matrix.new([Vector.new([0]), Vector.new([1, 2])]);
[00052403:error] !!! new: expecting rows of size 1, got size 2 for row 2
```

If it's a Matrix, then there is a deep-copy: they are not aliases.

```
urbiscript
Session
var m1 = Matrix.new([[1, 1], [1, 1]])|;
var m2 = Matrix.new(m1)|;
m2[0, 0] = 0|;
assert
{
  m1 == Matrix.new([[1, 1], [1, 1]]);
  m2 == Matrix.new([[0, 1], [1, 1]]);
};
```

When given two Float arguments, they must be the two integers, defining the size of the null Matrix.

```
urbiscript
Session
Matrix.new(2, 3);
[00051329] Matrix([
  [0, 0, 0],
  [0, 0, 0]])
```

In other cases, the arguments are expected to be Lists of Floats and/or Vectors.

```
Matrix.new([1, 2])          == Matrix.new([[1, 2]]);
Matrix.new([1, 2],           [3, 4]) == Matrix.new([[1, 2], [3, 4]]);
Matrix.new([1, 2], Vector.new(3, 4)) == Matrix.new([[1, 2], [3, 4]]);
```

Assertion Block

These rows must have equal sizes.

urbiscript
Session

```
// Lists and Lists.
Matrix.new([0], [1, 2]);
[00000160:error] !!! new: expecting rows of size 1, got size 2 for row 2
Matrix.new([0, 1], [2]);
[00000169:error] !!! new: expecting rows of size 2, got size 1 for row 2

// Lists and Vectors.
Matrix.new([0], Vector.new(1, 2));
[00000178:error] !!! new: expecting rows of size 1, got size 2 for row 2
Matrix.new(Vector.new(0, 1), [2]);
[00000186:error] !!! new: expecting rows of size 2, got size 1 for row 2

// Vectors and Vectors.
Matrix.new(Vector.new(0), Vector.new(1, 2));
[00000195:error] !!! new: expecting rows of size 1, got size 2 for row 2
Matrix.new(Vector.new(0, 1), Vector.new(2));
[00000204:error] !!! new: expecting rows of size 2, got size 1 for row 2
```

22.38.3 Slots

- `'*' (that)`

If *that* is a Matrix, matrix product between `this` and *that*.

```
Matrix.new([1, 2]) * Matrix.new([10], [20])
  == Matrix.new([50]);
Matrix.new([3, 4]) * Matrix.new([10], [20])
  == Matrix.new([110]);
Matrix.new([1, 2], [3, 4]) * Matrix.new([10], [20])
  == Matrix.new([50], [110]);
```

Assertion Block

The sizes must be compatible (`this.size.second = that.size.first`).

```
Matrix.new([1, 2]) * Matrix.new([3, 4]);
[00081168:error] !!! *: incompatible sizes: 1x2, 1x2
```

urbiscript Session

If *that* is a Float, the scalar product.

```
Matrix.new([1, 2], [3, 4]) * 3 == Matrix.new([3, 6], [9, 12]);
```

Assertion Block

- `'*=' (that)`

In place (Section 21.1.8.2) product (`'*'`). The same constraints apply.

```
var lhs1 = Matrix.new([1, 2], [3, 4]);
var lhs2 = Matrix.new(lhs1);
var rhs = Matrix.new([10], [20]);
var res = lhs1 * rhs;

(lhs1.'*='(rhs)) === lhs1; lhs1 == res;
(lhs2 *= rhs) === lhs2; lhs2 == res;

rhs *= rhs;
[00272182:error] !!! *=: incompatible sizes: 2x1, 2x1
```

Assertion Block

```
var v = Matrix.new([1, 2], [3, 4]);
var res = v * 3;
(v *= 3) === v; v == res;
```

Assertion Block

- `'+' (that)`

The sum of `this` and *that*. Their sizes must be equal.

Assertion Block

```
Matrix.new([1, 2]) + Matrix.new([10, 20])
  == Matrix.new([11, 22]);

Matrix.new([1, 2], [3, 4]) + Matrix.new([10, 20], [30, 40])
  == Matrix.new([11, 22], [33, 44]);

Matrix.new([1, 2]) + Matrix.new([10, 20], [30, 40]);
[00002056:error] !!! +: incompatible sizes: 1x2, 2x2
```

If *that* is a Float, the scalar addition.

Assertion Block

```
Matrix.new([1, 2], [3, 4]) + 3 == Matrix.new([4, 5], [6, 7]);
```

- `'+='(that)`

In place ([Section 21.1.8.2](#)) sum (`'+'`). The same constraints apply.

Assertion Block

```
var lhs1 = Matrix.new([ 1 , 2], [ 3 , 4]);
var lhs2 = Matrix.new(lhs1);
var rhs = Matrix.new([10, 20], [30, 40]);
var res = lhs1 + rhs;

(lhs1.'+'(rhs)) === lhs1; lhs1 == res;
(lhs2 += rhs) === lhs2; lhs2 == res;

lhs1 += Matrix.new([1, 2]);
[00338194:error] !!! +=: incompatible sizes: 2x2, 1x2
```

Assertion Block

```
var v = Matrix.new([3, 6], [9, 12]);
var res = v + 3;
(v += 3) === v; v == res;
```

- `'-'(that)`

The difference of `this` and *that*. Their sizes must be equal.

Assertion Block

```
Matrix.new([1, 2]) - Matrix.new([10, 20])
  == Matrix.new([-9, -18]);

Matrix.new([1, 2], [3, 4]) - Matrix.new([10, 20], [30, 40])
  == Matrix.new([-9, -18], [-27, -36]);

Matrix.new([1, 2]) - Matrix.new([10, 20], [30, 40]);
[00002056:error] !!! -: incompatible sizes: 1x2, 2x2
```

If *that* is a Float, the scalar difference.

Assertion Block

```
Matrix.new([1, 2], [3, 4]) - 3 == Matrix.new([-2, -1], [0, 1]);
```

- `'-='(that)`

In place ([Section 21.1.8.2](#)) difference (`'-'`). The same constraints apply.

Assertion Block

```
var lhs1 = Matrix.new([ 1 , 2], [ 3 , 4]);
var lhs2 = Matrix.new(lhs1);
var rhs = Matrix.new([10, 20], [30, 40]);
var res = lhs1 - rhs;

(lhs1.'-'(rhs)) === lhs1; lhs1 == res;
(lhs2 -= rhs) === lhs2; lhs2 == res;

lhs1 -= Matrix.new([1, 2]);
[00362383:error] !!! -=: incompatible sizes: 2x2, 1x2
```

Assertion Block

```
var v = Matrix.new([3, 6], [9, 12]);
var res = v - 3;
(v == 3) === v; v == res;
```

- `'/'(that)`

Same as `this * that.inverse`. `that` must be invertible.

```
var lhs = Matrix.new([20, 0], [0, 200]);
var rhs = Matrix.new([10, 0], [0, 100]);
var res = Matrix.new([2, 0], [0, 2]);

lhs / rhs == res;
(lhs / rhs) * rhs == lhs;
rhs * (lhs / rhs) == lhs;

lhs / Matrix.createZeros(2, 2);
[00160168:error] !!! /: non-invertible matrix: <>0, 0>, <0, 0>>
```

Assertion Block

If `that` is a Float, the scalar division.

```
Matrix.new([3, 6], [9, 12]) / 3 == Matrix.new([1, 2], [3, 4]);
```

Assertion Block

- `'/='(that)`

In place ([Section 21.1.8.2](#)) division (`'/'`). The same constraints apply.

```
var lhs1 = Matrix.new([20, 0], [0, 200]);
var lhs2 = Matrix.new(lhs1);
var rhs = Matrix.new([10, 0], [0, 100]);
var res = Matrix.new([2, 0], [0, 2]);

(lhs1.'/='(rhs)) === lhs1; lhs1 == res;
(lhs2 /= rhs) === lhs2; lhs2 == res;

lhs1 /= Matrix.createZeros(2, 2);
[00207285:error] !!! /=: non-invertible matrix: <>0, 0>, <0, 0>>
```

Assertion Block

Assertion Block

```
var v = Matrix.new([3, 6], [9, 12]);
var res = v / 3;
(v /= 3) === v; v == res;
```

- `'==='(that)`

Whether `this` and `that` have the same dimensions and members.

urbiscript Session

```
do (Matrix)
{
    assert
    {
        new([[1], [2], [3]]) == new([[1], [2], [3]]);
        !(new([[1], [2], [3]]) == new([[1], [2]]));
        !(new([[1], [2], [3]]) == new([[3], [2], [1]]));
    };
}|;
```

- `'[]'(row, col)`

The element at `row, col`. The index `row` must verify $0 \leq row < \text{this.size.first}$, or $-\text{this.size.first} \leq row < 0$, in which case it is equivalent to using index `row + this.size.first`: it counts “backward”. Similarly for `col`.

urbiscript Session

```
var m = Matrix.new([1, 2, 3], [10, 20, 30])|;
assert
```

```
{
    m[0, 0] == 1;    m[0, -3] == 1;
    m[0, 1] == 2;    m[0, -2] == 2;
    m[0, 2] == 3;    m[0, -1] == 3;

    m[1, 2] == 30;   m[-1, -1] == 30;
};

m[2, 0];
[00127812:error] !!! []: invalid row: 2

m[-3, 0];
[00127824:error] !!! []: invalid row: -3

m[0, 3];
[00127836:error] !!! []: invalid column: 3

m[0, -4];
[00127850:error] !!! []: invalid column: -4
```

- `'[]='(row, col, val)`

Set the element at `row`, `col` to `val`, and return `val`. The index `row` must verify $0 \leq \text{row} < \text{this.size.first}$, or $-\text{this.size.first} \leq \text{row} < 0$, in which case it is equivalent to using index `row + this.size.first`: it counts “backward”. Similarly for `col`.

urbiscript
Session

```
var m = Matrix.new([1, 2], [10, 20])!;
assert
{
    (m[0, 0] == -1) == -1;    m[0, 0] == -1;
    (m[-1, -1] == -2) == -2;  m[1, 1] == -2;
};

m[2, 0] = -1;
[00127812:error] !!! []=: invalid row: 2

m[-3, 0] = -1;
[00127824:error] !!! []=: invalid row: -3

m[0, 2] = -1;
[00127836:error] !!! []=: invalid column: 2

m[0, -3] = -1;
[00127850:error] !!! []=: invalid column: -3
```

- `appendRow(vector)`

Append `vector` to `this` and return `this`.

Assertion
Block

```
var m2x1 = Matrix.new([0], [1]);
m2x1.appendRow(Vector.new(2)) === m2x1;
m2x1 == Matrix.new([0], [1], [2]);

var m2x2 = Matrix.new([0, 1], [10, 11]);
m2x2.appendRow(Vector.new(20, 21)) == m2x2;
m2x2 == Matrix.new([0, 1], [10, 11], [20, 21]);
```

Sizes must match.

urbiscript
Session

```
Matrix.new([0], [1]).appendRow(Vector.new(10, 11));
[00017936:error] !!! appendRow: incompatible sizes: 2x1, 2

Matrix.new([0, 1]).appendRow(Vector.new(10));
[00050922:error] !!! appendRow: incompatible sizes: 1x2, 1
```

- **asMatrix**
`this.`

```
Matrix.asMatrix() === Matrix;
var m = Matrix.new([1], [2]);
m.asMatrix() === m;
```

Assertion Block

- **asPrintable**

A String that denotes `this`.

```
Matrix           .asPrintable() == "Matrix([])";
Matrix.new([1, 2], [3, 4]).asPrintable() == "Matrix([[1, 2], [3, 4]])";
```

Assertion Block

- **asString**

A String that denotes `this`.

```
Matrix           .asString() == "<>";
Matrix.new([1, 2], [3, 4]).asString() == "<<1, 2>, <3, 4>>";
```

Assertion Block

- **asTopLevelPrintable**

A String that denotes `this`.

```
Matrix           .asTopLevelPrintable()
== "Matrix([])";
Matrix.new([1, 2], [3, 4]).asTopLevelPrintable()
== "Matrix(\n [1, 2],\n [3, 4])";
```

Assertion Block

- **column(*i*)**

The *i*th column as a Vector. See also `row`.

Assertion Block

```
var m = Matrix.new([1, 2, 3], [4, 5, 6]);
m.column(0) == Vector.new(1, 4); m.column(-3) == m.column(0);
m.column(1) == Vector.new(2, 5); m.column(-2) == m.column(1);
m.column(2) == Vector.new(3, 6); m.column(-1) == m.column(2);

m.column(3);
[00000232:error] !!! column: invalid column: 3
```

- **createIdentity(*size*)**

The unit Matrix of dimensions *size*.

Assertion Block

```
Matrix.createIdentity(0) == Matrix;
Matrix.createIdentity(3) == Matrix.new([1, 0, 0], [0, 1, 0], [0, 0, 1]);

Matrix.createIdentity(-2);
[00000328:error] !!! createIdentity: argument 1: expected non-negative integer: -2
```

- **createOnes(*row*, *col*)**

Same as `createScalar(row, col, 1)`.

Assertion Block

```
Matrix.createOnes(0, 0) == Matrix;
Matrix.createOnes(2, 3) == Matrix.new([1, 1, 1], [1, 1, 1]);

Matrix.createOnes(-2, 2);
[00000328:error] !!! createOnes: argument 1: expected non-negative integer: -2
```

- **createScalars(*row*, *col*, *scalar*)**

A Matrix of size *row* * *col* filled with *scalar*.

Assertion Block

```
Matrix.createScalars(0, 0, 99) == Matrix;
Matrix.createScalars(2, 3, 99) == Matrix.new([99, 99, 99], [99, 99, 99]);

Matrix.createScalars(-2, 2, 99);
[00000328:error] !!! createScalars: argument 1: expected non-negative integer: -2
```

- **createZeros(*row*, *col*)**

Same as `createScalar(row, col, 0)`.

Assertion Block

```
Matrix.createZeros(0, 0) == Matrix;
Matrix.createZeros(2, 3) == Matrix.new([0, 0, 0], [0, 0, 0]);

Matrix.createZeros(-2, 2);
[00000328:error] !!! createZeros: argument 1: expected non-negative integer: -2
```

- **distanceMatrix(*that*)**

Considering that `this` and `that` are collections of Vectors that denote positions in an Euclidean space, produce a Matrix whose value at (*i*, *j*) is the distance between points `this[i]` and `that[j]`.

Assertion Block

```
// Left-hand side matrix.
var l0 = Vector.new([0, 0]); var l1 = Vector.new([2, 1]);
var lhs = Matrix.new([l0, l1]);
// Right-hand side matrix.
var r0 = Vector.new([0, 1]); var r1 = Vector.new([2, 0]);
var rhs = Matrix.new([r0, r1]);

lhs.distanceMatrix(rhs)
== Matrix.new([l0.distance(r0), l0.distance(r1)],
[l1.distance(r0), l1.distance(r1)]);
```

- **init**

See [Section 22.38.2](#).

- **inverse**

The inverse of `this` if it exists, raise an error otherwise.

Assertion Block

```
var m = Matrix.new(
  [1, 3, 1],
  [1, 1, 2],
  [2, 3, 4]);

m * m.inverse() == Matrix.createIdentity(3);
m.inverse() * m == Matrix.createIdentity(3);

m.inverse() == Matrix.new(
  [ 2,  9, -5],
  [ 0, -2,  1],
  [-1, -3,  2]);

Matrix.createZeros(2, 2).inverse();
[00000534:error] !!! inverse: non-invertible matrix: <>0, 0>, <0, 0>
```

- **resize(*row*, *col*)**

Change the dimensions of `this`, using 0 for new members. Return `this`.

```
// Check that <>1, 2><3, 4> is equal to the Matrix composed of rows,
// when resized to the dimensions of rows.
function resized(var rows[])
{
  var m = Matrix.new([1, 2], [3, 4]);
```

urbiscript
Session

```

var res = Matrix.new(rows);
// Resize returns this...
m.resize(res.size.first, res.size.second) === m;
// ...and does resize.
m == res;
}|;
assert
{
    // Fewer rows/cols.
resized([1], [3]);
resized([1, 2]);
resized([1]);
resized([]);

    // As many rows and cols.
resized([1, 2], [3, 4]);

    // More rows/cols.
resized([1, 2, 0], [3, 4, 0]);
resized([1, 2], [3, 4], [0, 0]);
resized([1, 2, 0], [3, 4, 0], [0, 0, 0]);

    // More rows, less cols, and conversely.
resized([1], [3], [0]);
resized([1, 2, 0]);
};

```

- **row(*i*)**

The *i*th row as a Vector. See also [column](#).

```

var m = Matrix.new([1, 2, 3], [4, 5, 6]);
m.row(0) == Vector.new(1, 2, 3); m.row(-2) == m.row(0);
m.row(1) == Vector.new(4, 5, 6); m.row(-1) == m.row(1);

m.row(2);
[00195645:error] !!! row: invalid row: 2

```

Assertion Block

- **rowAdd(*vector*)**

A Matrix whose rows (vectors) are the sum of each row of **this** with *vector*.

```

do (Matrix)
{
    assert
    {
        rowAdd(Vector) == Matrix;
        new([[1, 2]]).rowAdd(Vector.new(10)) == new([[11, 12]]);
        new([1], [2]).rowAdd(Vector.new(10, 20)) == new([11], [22]);
        new([1, 2], [3, 4]).rowAdd(Vector.new(10, 20)) == new([11, 12], [23, 24]);
    }
}|;

```

urbiscript Session

The dimensions must be compatible: `size.first == vector.size`.

```

Matrix.new([1], [2]).rowAdd(Vector.new(10));
[00000415:error] !!! rowAdd: incompatible sizes: 2x1, 1

Matrix.new([1, 2]).rowAdd(Vector.new(10, 20));
[00000425:error] !!! rowAdd: incompatible sizes: 1x2, 2

```

urbiscript Session

- **rowDiv(*vector*)**

A Matrix whose rows (vectors) are the member-wise division of each row of **this** with *vector*.

```
do (Matrix)
{
    assert
    {
        rowDiv(Vector) == Matrix;
        new([[10, 20]]).rowDiv(Vector.new(10)) == new([[1, 2]]);
        new([10], [20]).rowDiv(Vector.new(10, 20)) == new([1], [1]);
        new([10, 30], [20, 40]).rowDiv(Vector.new(10, 20)) == new([1, 3], [1, 2]);
    }
};
```

The dimensions must be compatible: `size.first == vector.size`.

urbiscript
Session

```
Matrix.new([1], [2]).rowDiv(Vector.new(10));
[00000415:error] !!! rowDiv: incompatible sizes: 2x1, 1

Matrix.new([1, 2]).rowDiv(Vector.new(10, 20));
[00000425:error] !!! rowDiv: incompatible sizes: 1x2, 2
```

- **rowMul(*vector*)**

A Matrix whose rows (vectors) are the member-wise product of each row of `this` with `vector`.

urbiscript
Session

```
do (Matrix)
{
    assert
    {
        rowMul(Vector) == Matrix;
        new([[10, 20]]).rowMul(Vector.new(10)) == new([[100, 200]]);
        new([10], [20]).rowMul(Vector.new(10, 20)) == new([100], [400]);
        new([1, 2], [3, 4]).rowMul(Vector.new(10, 20)) == new([10, 20], [60, 80]);
    }
};
```

The dimensions must be compatible: `size.first == vector.size`.

urbiscript
Session

```
Matrix.new([1], [2]).rowMul(Vector.new(10));
[00000415:error] !!! rowMul: incompatible sizes: 2x1, 1

Matrix.new([1, 2]).rowMul(Vector.new(10, 20));
[00000425:error] !!! rowMul: incompatible sizes: 1x2, 2
```

- **rowNorm**

A Vector whose values are the (Euclidean) norms of the rows of `this`.

```
var m = Matrix.new([1, 2], [3, 3]);
m.rowNorm()[0] == m.row(0).norm();
m.rowNorm()[-1] == m.row(-1).norm();
```

Assertion
Block

- **rowSub(*vector*)**

A Matrix whose rows (vectors) are the difference of each row of `this` with `vector`.

```
do (Matrix)
{
    assert
    {
        rowSub(Vector) == Matrix;
        new([[10, 20]]).rowSub(Vector.new(1)) == new([[9, 19]]);
        new([11], [22]).rowSub(Vector.new(10, 20)) == new([1], [2]);
        new([1, 2], [3, 4]).rowSub(Vector.new(1, 2)) == new([0, 1], [1, 2]);
    }
};
```

urbiscript
Session

The dimensions must be compatible: `size.first == vector.size`.

```
Matrix.new([1], [2]).rowSub(Vector.new(10));
[00000415:error] !!! rowSub: incompatible sizes: 2x1, 1

Matrix.new([1, 2]).rowSub(Vector.new(10, 20));
[00000425:error] !!! rowSub: incompatible sizes: 1x2, 2
```

urbiscript
Session

- `set(vectors)`

Change `this` to be equal to the Matrix defined by the list of vectors given as argument, and return `this`.

```
var m = Matrix.new([]);

var m1 = Matrix.new([0, 1], [0, 1]);
m === m.set([[0, 1], [0, 1]]);
m == m1;

var m2 = Matrix.new([2, 3]);
m === m.set([[2, 3]]);
m == m2;
```

Assertion
Block

- `setRow(index, vector)`

Set the `index`th row of `this` to `vector` and return `this`.

```
var m2x1 = Matrix.new([0], [1]);
m2x1.setRow(0, Vector.new(2)) === m2x1;
m2x1 == Matrix.new([2], [1]);

var m2x2 = Matrix.new([0, 1], [10, 11]);
m2x2.setRow(0, Vector.new(20, 21)) == m2x2;
m2x2 == Matrix.new([20, 21], [10, 11]);
```

Assertion
Block

Sizes and index must match.

urbiscript
Session

```
Matrix.new([0], [1]).setRow(0, Vector.new(10, 11));
[00017936:error] !!! setRow: incompatible sizes: 2x1, 2

Matrix.new([0, 1]).setRow(0, Vector.new(10));
[00050922:error] !!! setRow: incompatible sizes: 1x2, 1

Matrix.new([0], [1]).setRow(2, Vector.new(10));
[00017936:error] !!! setRow: invalid row: 2
```

Assertion
Block

- `size(arg)`

The dimensions of the `this`, as a Pair of Floats.

Assertion
Block

```
Matrix.size == Pair.new(0, 0);
Matrix.new([1, 2], [3, 4], [5, 6]).size == Pair.new(3, 2);
Matrix.new([1, 2, 3], [4, 5, 6]) .size == Pair.new(2, 3);
```

Assertion
Block

- `transpose(arg)`

The transposed of `this`.

Assertion
Block

```
Matrix           .transpose() == Matrix;
Matrix.new([1]) .transpose() == Matrix.new([1]);
Matrix.new([1], [2]) .transpose() == Matrix.new([1, 2]);
Matrix.new([1, 2], [3, 4]).transpose() == Matrix.new([1, 3], [2, 4]);
```

Assertion
Block

- `type`

The String Matrix.

Assertion
Block

```
Matrix.type      == "Matrix";
Matrix.new([]).type == "Matrix";
```

22.39 Mutex

Mutex allow to define critical sections.

22.39.1 Prototypes

- [Tag](#)

22.39.2 Construction

A Mutex can be constructed like any other Tag but without name.

urbiscript
Session

```
var m = Mutex.new();
[00000000] Mutex_0x964ed40
```

You can define critical sections by tagging your code using the Mutex.

urbiscript
Session

```
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.

Mutexes must be used when manipulating data structures in a non atomic way to avoid inconsistent states.

Consider this apparently simple code:

urbiscript
Session

```
function appendAndTellIfFirst(list, val)
{
    var res = list.empty;
    list << val;
    res
}|;
var l = [];
[00000001] []
appendAndTellIfFirst(l, 1);
[00000002] true
appendAndTellIfFirst(l, 2);
[00000002] false
```

Now look what happens if called twice in parallel:

urbiscript
Session

```
l = [];
[00000001] []
var res1; var res2;
res1 = appendAndTellIfFirst(l, 1) & res2 = appendAndTellIfFirst(l, 2)|;
res1;
[00000002] true
res2;
[00000003] true
l.sort(); // order is unspecified
[00000004] [1, 2]
```

Both tasks checked if the list was empty at the same time, and then appened the element. A mutex will solve this problem:

urbiscript
Session

```
l = [];
[00000001] []
var m = Mutex.new();
[00000000] Mutex_0x964ed40
// redefine the function using the mutex
appendAndTellIfFirst = function (list, val)
{m:{|
    var res = list.empty;
```

```

list << val;
res
}};

// check again
res1 = appendAndTellIfFirst(l, 1) & res2 = appendAndTellIfFirst(l, 2);
// we do not know which one was first, but only one was
[res1, res2].sort();
[00000001] [false, true]
l.sort();
[00000004] [1, 2]

```

Mutex constructor accepts an optional maximum queue size: code blocks trying to wait when maximum queue size is reached will not be executed:

urbiscript
Session

```

var m = Mutex.new(1);
var e = Event.new();
at(e?) {
  m: { echo("executing at"); sleep(200ms);};
e!;e!;e!;
sleep(600ms);
[00000001] *** executing at
[00000001] *** executing at

```

As you can see above the message is only displayed twice: First at got executed right away, the second was queued and executed when the first one finished, and the third one got stopped.

22.39.3 Slots

- **asMutex**

Return **this**.

urbiscript
Session

```

var m1 = Mutex.new();
assert
{
  m1.asMutex() === m1;
};

```

22.40 nil

The special entity `nil` is an object used to denote an empty value. Contrary to `void`, it is a regular value which can be read.

22.40.1 Prototypes

- `Singleton`

22.40.2 Construction

Being a singleton, `nil` is not to be constructed, just used.

Assertion Block

```
nil == nil;
```

22.40.3 Slots

- `isNil`

Whether `this` is `nil`. I.e., true. See also `Object.isNil`.

```
nil.isNil;  
!Object.isNil; !42.isNil; !(function () { nil }).isNil;
```

Assertion Block

- `isVoid`

In order to facilitate the transition from older code to newer, return true. In the future, false will be returned. Therefore, if you really need to check whether `foo` is `void` but not `nil`, use `!foo.acceptVoid().isNil && foo.isVoid`.

```
nil.isVoid;  
[ Logger      ] nil.isVoid will return false eventually, adjust your code.  
[ Logger      ] For instance replace InputStream loops from  
[ Logger      ]   while (!(x = i.get()).acceptVoid()).isVoid()  
[ Logger      ]       cout << x;  
[ Logger      ]       to  
[ Logger      ]   while (!(x = i.get()).isNil())  
[ Logger      ]       cout << x;
```

Assertion Block

22.41 Object

`Object` includes the mandatory primitives for all objects in urbiscript. All objects in urbiscript must inherit (directly or indirectly) from it.

22.41.1 Prototypes

- `Comparable`
- `Global`

22.41.2 Construction

A fresh object can be instantiated by cloning `Object` itself.

urbiscript
Session

```
Object.new();
[00000421] Object_0x00000000
```

The keyword `class` also allows to define objects intended to serve as prototype of a family of objects, similarly to classes in traditional object-oriented programming languages (see Section 8.4).

urbiscript
Session

```
{
  class Foo
  {
    var attr = 23;
  };
  assert
  {
    Foo.localSlotNames() == ["asFoo", "attr", "type"];
    Foo.asFoo() === Foo;
    Foo.attr == 23;
    Foo.type == "Foo";
  };
}
```

22.41.3 Slots

- `'!'`

Logical negation. If `this` evaluates to false return `true` and vice-versa.

Assertion
Block

```
!1 == false;
!0 == true;

!"foo" == false;
!"'" == true;
```

- `'!== (that)`

The negation of `\this === \that`, see `'==='`.

```
var o1 = Object.new();
var o2 = Object.new();

o1 !== o2;
!(o1 !== o1);

1 !== 1;
"1" !== "1";
[1] !== [1];
```

Assertion
Block

- `'&&'(that)`

Short-circuiting logical (Boolean) *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";
```

Assertion Block

- `'*='(that)`

Bounce to `this * that` ([Section 21.1.8.2](#)).

```
var x = 2;
(x *= 5) == 10; x == 10;
x.'*='(2) == 20; x == 10;
```

Assertion Block

- `'+='(that)`

Bounce to `this + that`. Be sure to understand how in-place operators are handled in urbiscript: [Section 21.1.8.2](#).

```
var x = 1;
(x += 1) == 2; x == 2;
x.'+='(1) == 3; x == 2;
```

Assertion Block

- `'-='(that)`

Bounce to `this - that` ([Section 21.1.8.2](#)).

Assertion Block

```
var x = 10;
(x -= 3) == 7; x == 7;
x.'-='(3) == 4; x == 7;
```

1

Assertion Block

- `'/='(that)`

Bounce to `this / that` ([Section 21.1.8.2](#)).

Assertion Block

```
var x = 200;
(x /= 10) == 20; x == 20;
x.'/='(2) == 10; x == 20;
```

1

Assertion Block

- `'=='(that)`

Whether `this` and `that` are equal. See also [Comparable](#) and [Section 21.1.8.6](#). By default, bounces to `'==='`. This operator *must* be redefined for objects that have a value-semantics; for instance two `String` objects that denotes the same string should be equal according to `==`, although physically different (i.e., not equal according to `===`).

Assertion Block

```
var o1 = Object.new();
var o2 = Object.new();

    o1 == o1;
!(o1 == o2);
    o1 != o2;
!(o1 != o1);

    1 == 1;
"1" == "1";
[1] == [1];
```

1

Assertion Block

- `'=='(that)`

Whether `this` and `that` are exactly the same object (i.e., `this` and `that` are two different means to denote the very same location in memory). To denote equivalence, use `'=='`; for instance two `Float` objects that denote 42 can be different objects (in the sense of `==`), but will be considered equal by `==`. See also `'=='`, and [Section 21.1.8.6](#).

Assertion Block

```
var o1 = Object.new();
var o2 = Object.new();

o1 === o1;
!(o1 === o2);

!(1 === 1);
!("1" === "1");
!([1] === [1]);
```

- `'^(that)`

Logical *exclusive or*. If `this` evaluates to false evaluate and return `that`, otherwise return `!that`. Beware that the semantics of `Float.'^'` (*bitwise exclusive or*, not *logical logical or*) is different, so in case of doubt use `a.asBool ^ b` instead of `a ^ b`.

Assertion Block

```
"foo" ^ "" === true;
"" ^ "foo" == "foo";
"" ^ 1 == 1;

"" ^ "" == "";
"a" ^ "b" === false;

// Beware of bitwise operations.
1 ^ 2 == 3; // As a Boolean, 3 is "true".
1.asBool ^ 2 === false;
```

- `'^=(that)`

Bounce to `this ^ that` ([Section 21.1.8.2](#)).

Assertion Block

```
var x = 0xff00;
(x ^= 0xffff) == 0x00ff; x == 0x00ff;
x.^=(0xffff) == 0xff00; x == 0x00ff;
```

- `'||(that)` Short-circuiting logical (Boolean) *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";
```

Assertion Block

- `'$id'`

- `acceptVoid`

Return `this`. See `void` to know why.

```
var o = Object.new();
o.acceptVoid() === o;
```

Assertion Block

- `addProto(proto)`

Add `proto` into the list of prototypes of `this`. Return `this`.

```
do (Object.new())
{
    assert
    {
        addProto(Orderable) === this;
        protos == [Orderable, Object];
    };
}|;
```

- **allProto**

A list with `this`, its parents, their parents,...

```
123.allProtos().size == 6;
```

Assertion Block

- **allSlotNames**

Deprecated alias for `slotNames`.

```
Object.allSlotNames() == Object.slotNames();
```

Assertion Block

- **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;
```

Assertion Block

- **as(type)**

Convert `this` to `type`. This is syntactic sugar for `asType` when `Type` is the type of `type`.

Assertion Block

```
12.as(Float) == 12;
"12".as(Float) == 12;
12.as(String) == "12";
Object.as(Object) === Object;
```

- **asBool**

Whether `this` is “true”, see [Section 22.3.1](#).

Assertion Block

```
Global.asBool() == true;
nil.asBool() == false;
```

urbiscript Session

```
void.asBool();
[00000421:error] !!! unexpected void
```

- **asPrintable**

A `String` that can be used to display faithfully `this` using `Lobby.echo`. Defaults to `asString`.

For instance, `String.asList` returns the string itself, but `"foo".asPrintable` is the string `"\"foo\""`, so that using `echo`, you would see `"foo"`.

Assertion Block

```
Object.asList().isA(String);
Object.asList() == Object.asList();

"foo".asString() == "foo";
"foo".asPrintable() == "\"foo\"";
```

See also `asString` and `asTopLevelPrintable`.

- **asString**

A conversion of `this` into a `String`. In general:

- if `this` is a value for which there is a literal syntax (e.g., `Float`, `String`, `List`, etc.), then use that syntax:

Assertion Block

```
3.1415.asString() == "3.1415";
"Hello, World!".asString() == "Hello, World!";
[1, "2", [3]].asString() == "[1, \"2\", [3]]";
```

- if `this` is a “class”, then the name of the class, otherwise a `String` composed of the name of the “class” the object is an instance of, and some integral value that depends on `this`.

urbiscript Session

```
class Class {};
Class.asString();
[00002106] "Class"
var c1 = Class.new(); var c2 = Class.new();
c1.asString();
[00002112] "Class_0x1040e0108"
c2.asString();
[00002115] "Class_0x1040c17a8"
assert
{
    c1.asString() == c1.asString() != c2.asString() == c2.asString();
    c1.asString() == c1.type + "_" + c1.uid();
    c2.asString() == c2.type + "_" + c2.uid();
};
```

See also `asTopLevelPrintable` and `asPrintable`.

- **asTopLevelPrintable**

A `String` used to display `this` in interactive sessions, or `nil` if `this` should not be printed.

urbiscript Session

```
class Test
{
    function init(v) { var this.value = v };
    function asTopLevelPrintable() { value };
};

Test.new("12");
[00004345] 12

// Nothing to display here.
Test.new(nil);

// This is an error, asTopLevelPrintable must return a string or nil.
Test.new(1);
[00004370:error] !!! send: argument 1: unexpected 1, expected a String
```

Defaults to `asPrintable`. For instance, since `nil` and `void` are not reported in interactive sessions, their `asTopLevelPrintable` is `nil`, but not their `asPrintable`.

Assertion Block

```
Object.asTopLevelPrintable().isA(String);
Object.asTopLevelPrintable() == Object.asPrintable();

// A version of void on which we can call methods.
var Void = void.acceptVoid();
nil.asPrintable() == "nil"; nil.asTopLevelPrintable() == nil;
Void.asPrintable() == "void"; Void.asTopLevelPrintable() == nil;
```

See also `asString` and `asPrintable`.

- **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";
```

Assertion Block

- **callMessage(*msg*)**

Invoke the `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() == [];
```

Assertion Block

- **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));
```

urbiscript Session

- **copySlot(*from*, *to*)**

Same as `cloneSlot`, but the slot aren't cloned, so the two slot are the same.

```
var moo = Object.new() |;
copySlot("moo", "loo") |;
assert(&moo === &loo);
```

urbiscript Session

- **createSlot(*name*)**

Create an empty slot (which actually means it is bound to `void`) named `name`. Raise an error if `name` was already defined.

```
var o = Object.new();

!o.hasLocalSlot("foo");
o.createSlot("foo").isVoid;
o.hasLocalSlot("foo");
```

Assertion Block

- **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] ***     value = String_0x23a750 <...>
[00015137] ***
do (2) { var this.attr = "foo"; this.attr->prop = "bar" }.dump(1);
[00020505] *** Float_0x42389f08 {
[00020505] ***   /* Special slots */
```

urbiscript Session

```
[00020505] ***  protos = Float
[00020505] ***  value = 2
[00020505] ***  /* Slots */
[00020505] ***  attr = String_0x42392b48 {
[00020505] ***    /* Special slots */
[00020505] ***    protos = String
[00020505] ***    /* Slots */
[00020505] ***    }
[00020505] ***    /* Properties */
[00020505] ***    prop = String_0x239330 {
[00020505] ***      /* Special slots */
[00020505] ***      protos = String
[00020505] ***      /* Slots */
[00020505] ***      }
[00020505] ***      value = String_0x42392b48 {
[00020505] ***        /* Special slots */
[00020505] ***        protos = String
[00020505] ***        /* Slots */
[00020505] ***        }
[00020505] ***      }
```

urbiscript
Session

```
Object.dump(1);
[00020555] *** Object {
[00020555] ***   /* Special slots */
[00020555] ***   protos = Comparable, Global
[00020555] ***   /* Slots */
[...]
[00020555] *** }
```

0.dump(1.5);

```
[00020605:error] !!! dump: expected integer: 1.5
```

- **getLocalSlot(*name*)**

The value associated to *name* in **this**, excluding its ancestors (contrary to **getSlot**).

urbiscript
Session

```
var a = Object.new();
// Local slot.
var a.slot = 21;
assert
{
  a.locateSlot("slot") === a;
  a.getLocalSlotValue("slot") == 21;
};

// Inherited slot are not looked-up.
assert { a.locateSlot("init") == Object };
a.getLocalSlotValue("init");
[00041066:error] !!! lookup failed: init
```

- **getPeriod**

Deprecated. Use **System.period** instead.

- **getProperty(*slotName*, *propName*)**

The value of the *propName* property associated to the slot *slotName* if defined. Raise an error otherwise.

urbiscript
Session

```
const var myPi = 3.14;
assert (getProperty("myPi", "constant"));

getProperty("myPi", "foobar");
```

```
[00000045:error] !!! property lookup failed: myPi->foobar
```

- **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;

// Local slot.
b.locateSlot("slot") === b;
b.getSlotValue("slot") == 21;

// Inherited slot.
b.locateSlot("init") === Object;
b.getSlotValue("init") == Object.getSlotValue("init");

// Unknown slot.
b.locateSlot("ENOENT") == nil;
b.getSlotValue("ENOENT");
[00041066:error] !!! lookup failed: ENOENT
```

Assertion Block

- **hash**

A `Hash` object for `this`. This default implementation returns a different hash for every object, so every key maps to a different cells. Classes that have value semantic should override the hash method so as objects that are equal (in the `Object.'=='` sense) have the same hash. `String.hash` does so for instance; as a consequence different String objects with the same value map to the same cell.

A hash only makes sense as long as the hashed object exists.

```
var o1 = Object.new();
var o2 = Object.new();

o1.hash() == o1.hash();
o1.hash() != o2.hash();
```

Assertion Block

- **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"));
```

urbiscript Session

- **hasProperty(*slotName*, *propName*)**

Whether the slot *slotName* of `this` has a property *propName*.

```
var o = Object.new();
const var o.halfPi = Math.pi / 2;

o.hasProperty("halfPi", "constant");
!o.hasProperty("halfPi", "foobar");
```

Assertion Block

- **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");
```

Assertion Block

- **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`.

urbiscript
Session

```
do (2) { var this.attr = "foo"; this.attr->prop = "bar"}.inspect();
[00001227] *** Inspecting 2
[00001227] *** ** Prototypes:
[00001227] ***     0
[00001227] *** ** Local Slots:
[00001228] ***     attr : gettable
[00001228] ***     Properties:
[00001228] ***         prop : String = "bar"
```

- **isA(*obj*)**

Whether `this` has `obj` in his parents.

Assertion
Block

```
Float.isA(Orderable);
! String.isA(Float);
```

- **isNil**

Whether `this` is `nil`.

Assertion
Block

```
nil.isNil;
! O.isNil;
```

- **isProto**

Whether `this` is a prototype.

```
Float.isProto;
! 42.isProto;
```

Assertion
Block

- **isVoid**

Whether `this` is `void`. See `void`.

```
void.isVoid;
! 42.isVoid;
```

Assertion
Block

- **localSlotNames**

A list with the names of the local (i.e., not including those of its ancestors) slots of `this`.

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;

top.localSlotNames() == ["top1", "top2"];
bot.localSlotNames() == ["bot1", "bot2"];
```

Assertion
Block

- **locateSlot(*slot*)**

The `Object` that provides `slot` to `this`, or `nil` if `this` does not feature `slot`.

```
locateSlot("locateSlot") == Object;
locateSlot("doesNotExist").isNil;
```

Assertion
Block

- **print**

Send `this` to the `Channel topLevel` channel.

urbiscript
Session

```
1.print();
[00001228] 1
[1, "12"].print();
[00001228] [1, "12"]
```

- **properties(*slotName*)**

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]
```

urbiscript
Session

- **protos**

The list of prototypes of `this`.

```
12.protos == [Float];
```

Assertion
Block

- **removeLocalSlot(*slot*)**

Remove *slot* from the (local) list of slots of `this`, and return `this`. Raise an error if *slot* does not exist. See also `removeSlot`.

```
var base = Object.new();
var base.slot = "base";

var derive = Base.new();
var derive.slot = "derive";

derive.removeLocalSlot("foo");
[00000080:error] !!! lookup failed: foo

assert
{
  derive.removeLocalSlot("slot") === derive;
  derive.localSlotNames() == [];
  base.slot == "base";
};

derive.removeLocalSlot("slot");
[00000090:error] !!! lookup failed: slot

assert
{
  base.slot == "base";
};
```

urbiscript
Session

- **removeProperty(*slotName*, *propName*)**

Remove the property *propName* from the slot *slotName*. Raise an error if the slot does not exist. Warn if *propName* does not exist; in a future release this will be an error.

```
var r = Object.new();

// Non-existing slot.
r.removeProperty("slot", "property");
[00000072:error] !!! lookup failed: slot

var r.slot = "slot value";
// Non-existing property.
r.removeProperty("slot", "property");
```

urbiscript
Session

```
[00000081:warning] !!! no such property: slot->property
[00000081:warning] !!!      called from: removeProperty

r.slot->property = "property value";
assert
{
    r.hasProperty("slot", "property");
    // Existing property.
    r.removeProperty("slot", "property") == "property value";
    ! r.hasProperty("slot", "property");
};


```

- **removeProto(proto)**

Remove *proto* from the list of prototypes of *this*, and return *this*. Do nothing if *proto* is not a prototype of *this*.

Assertion Block

```
var o = Object.new();

o.addProto(Orderable);
o.removeProto(123) === o;
o.protos == [Orderable, Object];
o.removeProto(Orderable) === o;
o.protos == [Object];
```

- **removeSlot(slot)**

Remove *slot* from the (local) list of slots of *this*, and return *this*. Warn if *slot* does not exist; in a future release this will be an error. See also [removeLocalSlot](#).

Assertion Block

```
var base = Object.new();
var base.slot = "base";

var derive = Base.new();
var derive.slot = "derive";

derive.removeSlot("foo") === derive;
[00000080:warning] !!! no such local slot: foo
[00000080:warning] !!!      called from: removeSlot
[00000080:warning] !!!      called from: code
[00000080:warning] !!!      called from: eval
[00000080:warning] !!!      called from: value
[00000080:warning] !!!      called from: assertCall

derive.removeSlot("slot") === derive;
derive.localslotNames() == [];
base.slot == "base";
derive.removeSlot("slot") === derive;
[00000099:warning] !!! no such local slot: slot
[00000099:warning] !!!      called from: removeSlot
[00000099:warning] !!!      called from: code
[00000099:warning] !!!      called from: eval
[00000099:warning] !!!      called from: value
[00000099:warning] !!!      called from: assertCall

base.slot == "base";
```

- **setConstSlot**

Like [setSlotValue](#) but the created slot is const.

urbiscript Session

```
assert(setConstSlotValue("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.

```
var o = Object.new();
var o.slot = "slot";
var value = "value";

o.setProperty("slot", "prop", value) === value;
"prop" in o.properties("slot");
o.getProperty("slot", "prop") === value;
o.slot->prop === value;
o.setProperty("slot", "noSuchProperty", value) === value;

o.setProperty("noSuchSlot", "prop", "12");
[00000081:error] !!! lookup failed: noSuchSlot
```

Assertion Block

In order to respect copy-on-write semantics, when a property is added to an inherited slot, the slot value is first copied.

```
var top = Object.new();
var top.x = "top";
var bot = top.new();

// bot.x is inherited from top.
bot.locateSlot("x") === top;

// Setting the property from bot's point of view detaches bot.x from
// top.x.
bot.x->prop = "irrelevant";
bot.locateSlot("x") === bot;
bot.x == top.x;
bot.x === top.x;
// top.x and bot.x are detached.
top.x = "new top";
bot.x == "top";
```

Assertion Block

- **setProtos(protos)**

Set the list of prototypes of *this* to *protos*. Return void.

```
var o = Object.new();

o.protos == [Object];
o.setProtos([Orderable, Object]).isVoid;
o.protos == [Orderable, Object];
```

Assertion Block

- **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;
```

Assertion Block

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 [System.redefinitionMode](#).

- **slotNames**

A list with the slot names of *this* and its ancestors.

Assertion Block

```
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 8.4). This is used to display the name of “instances”.

urbiscript
Session

```
class Example {};
[00000081] Example
assert
{
  Example.type == "Example";
};
Example.new();
[00000081] Example_0x423740c8
```

- **uid**

The unique id of `this`.

Assertion
Block

```
var foo = Object.new();
var bar = Object.new();

foo.uid == foo.uid;
foo.uid != bar.uid;
```

- **unacceptVoid**

Return `this`. See `void` to know why.

Assertion
Block

```
var o = Object.new();
o.unacceptVoid() === o;
```

- **updateSlot(*name*, *value*)**

Map the existing slot named *name* to *value*. Raise an error if *name* was not defined.

Assertion
Block

```
Object.setSlot("one", 1) == 1;
Object.updateSlot("one", 2) == 2;
Object.one == 2;
```

Implements *copy-on-write*, i.e., when updating an inherited slot, first copy this slot in `this`.

Assertion
Block

```
var top = Object.new();
var top.x = 123;
var bot = top.new();
// x is inherited.
bot.locateSlot("x") === top;
bot.updateSlot("x", 456) == 456;
// x is local.
bot.locateSlot("x") === bot;
```

If the slot is declared constant (Section 21.4.2), updating the slot is invalid, but copy-on-write will work.

urbiscript
Session

```
class Top
{
  const var x = [1];
};

// Cannot update local const slot.
```

```
Top.updateSlot("x", [2]);
[00007858:error] !!! cannot modify const slot

// Updating inherited const slot is possible.
var bot = Top.new();
bot.updateSlot("x", [3]);
[00007872] [3]
```

The constant property prevents the slot from being assigned a new value, it does not forbid mutable values from being modified.

```
Top.x << 2|;
assert
{
    // The associated value is updated.
    Top.x == [1, 2];
};
```

urbiscript
Session

22.42 Orderable

Objects that have a concept of “less than”. See also [Comparable](#).

22.42.1 Example

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 '<' (that) { value < that.value; };
    function asString() { "<" + value.asString + ">"; };
}|;
var one = Foo.new(1)|;
var two = Foo.new(2)|;

assert
{
    one <= one ; one <= two ; !(two <= one);
    !(one > one) ; !(one > two) ; two > one;
    (one >= one) ; !(one >= two) ; two >= one;
};
```

urbiscript
Session

22.42.2 Prototypes

- [Object](#)

22.42.3 Slots

- `'<'(that)`
Whether `this <= that && this != that`.
- `'<='(that)`
Whether `that > this || this == that`.
- `'>'(that)`
Whether `this >= that && this != that`.
- `'>='(that)`
Whether `that < this || this != that`.

22.43 OutputStream

OutputStreams are used to write (possibly binary) files by hand.

22.43.1 Prototypes

- [Stream](#)

22.43.2 Construction

An OutputStream is a writing-interface to a file; its constructor requires a [File](#). If the file already exists, content is *appended* to it. Remove the file beforehand if you want to override its content.

```
urbiscript
Session
var o1 = OutputStream.new(File.create("file.txt"));
[00001379] OutputStream_Ox10808a300

var o2 = OutputStream.new(File.new("file.txt"));
[00001396] OutputStream_Ox1080872e0
```

Cloning a closed [Stream](#) is valid, but it is forbidden to clone an opened one.

```
urbiscript
Session
var o3 = OutputStream.clone().new(File.new("file.txt"));
[00001399] OutputStream_Ox10803e7a0

o1.clone();
[00001401:error] !!! clone: cannot clone opened Streams
```

Do not forget to close the streams you opened ([Section 22.64.2](#)).

```
o1.close();
o2.close();
o3.close();
```

urbiscript
Session

22.43.3 Slots

- `'<<'(that)`

Output `this.asString`. Return `this` to enable chains of calls. Raise an error if the file is closed.

```
urbiscript
Session
var o = OutputStream.new(File.create("fresh.txt"))|;
o << 1 << "2" << [3, [4]]|;
o.close();
assert (File.new("fresh.txt").content.data == "12[3, [4]]");
o << 1;
[00000005:error] !!! <<: stream is closed
```

- `flush`

To provide 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. Raise an error if the file is closed.

```
urbiscript
Session
var s = OutputStream.new(File.create("file.txt"))|
s.flush();
s.close();
s.flush();
[00039175:error] !!! flush: stream is closed
```

- `put(byte)`

Output the character corresponding to the numeric code `byte` in `this`, and return `this`. Raise an error if the file is closed.

urbiscript
Session

```
var f = File.create("put.txt") |
var os = OutputStream.new(f) |

assert
{
    os.put(0)
        .put(255)
        .put(72).put(101).put(108).put(108).put(111)
    === os;
    f.content.data == "\0\xffHello";
};

os.put(12.5);
[00029816:error] !!! put: argument 1: expected integer: 12.5
os.put(-1);
[00034840:error] !!! put: argument 1: expected non-negative integer: -1
os.put(256);
[00039175:error] !!! put: argument 1: expected non-positive integer: 256
os.close();
os.put(0);
[00039179:error] !!! put: stream is closed
```

22.44 Pair

A *pair* is a container storing two objects, similar in spirit to `std::pair` in C++.

22.44.1 Prototypes

- [Tuple](#)

22.44.2 Construction

A *Pair* is constructed with two arguments.

urbiscript
Session

```
Pair.new(1, 2);
[00000001] (1, 2)

Pair.new();
[00000003:error] !!! new: expected 2 arguments, given 0

Pair.new(1, 2, 3, 4);
[00000003:error] !!! new: expected 2 arguments, given 4
```

22.44.3 Slots

- [first](#)

Return the first member of the pair.

Assertion
Block

```
Pair.new(1, 2).first == 1;
Pair[0] === Pair.first;
```

- [second](#)

Return the second member of the pair.

Assertion
Block

```
Pair.new(1, 2).second == 2;
Pair[1] === Pair.second;
```

22.45 Path

A *Path* points to a file system entity (directory, file and so forth).

22.45.1 Prototypes

- Comparable
- Orderable

22.45.2 Construction

Path itself is the root of the file system: `/` on Unix, and `C:\` on Windows.

urbiscript
Session

```
Path;
[00000001] Path("/")
```

A *Path* is constructed with the string that points to the file system entity. This path can be relative or absolute.

urbiscript
Session

```
Path.new("foo");
[00000002] Path("foo")

Path.new("/path/file.u");
[00000001] Path("/path/file.u")
```

Some minor simplifications are made, such as stripping useless `'./'` occurrences.

urbiscript
Session

```
Path.new("./../../../../foo/");
[00000002] Path("./foo")
```

22.45.3 Slots

- `'/'(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
```

urbiscript
Session

- `'<'(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"));
```

Assertion
Block

- `'=='(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"));
```

Assertion
Block

- `absolute`

Whether `this` is absolute.

```
Path.new("/abs/path").absolute;
!Path.new("rel/path").absolute;
```

Assertion
Block

- **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();
```

Assertion Block

- **asPrintable**

```
Path.new("file.txt").asPrintable() == "Path(\"file.txt\");"
```

Assertion Block

- **asString**

The name of the file.

```
Path.new("file.txt").asString() == "file.txt";
```

Assertion Block

- **basename**

Base name of the path. See also [dirname](#).

```
Path.new("/absolute/path/file.u").basename == "file.u";
Path.new("relative/path/file.u").basename == "file.u";
```

Assertion Block

- **cd**

Change the current working directory to [this](#). Return the new current working directory as a Path.

```
var a = Directory.create("a").asPath();
var b = Directory.create("a/b").asPath();
var cwd = Path.cwd; // Current location.
cwd.isA(Path);
// cd returns the new current working directory.
b.cd() == cwd / b == cwd / "a" / "b";
Path.cwd == cwd / b;
// Go back to the original location.
Path.new("../..").cd() == cwd;
Path.cwd == cwd;
```

Assertion Block

Exceptions are thrown on cases of error.

urbiscript Session

```
Path.new("does/not/exist").cd();
[00003991:error] !!! cd: no such file or directory: does/not/exist

var f = File.create("file.txt");
f.asPath().cd();
[00099415:error] !!! cd: not a directory: file.txt
```

Permissions are not properly handled on Windows, so the following example would actually fail.

urbiscript Session

```
var d = Directory.create("forbidden");
System.system("chmod 444 %s" % d);
d.asPath().cd();
[00140753:error] !!! cd: Permission denied: forbidden
```

- **cwd**

The current working directory.

Assertion Block

```
// Save current directory.
var pwd = Path.cwd;
// Go into "/".
var root = Path.new("/").cd();
// Current working directory is "/".
Path.cwd == root;
// Go back to the directory we were in.
pwd.cd() == pwd;
```

- **dirname**

Directory name of the path. See also [basename](#).

Assertion Block

```
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.

Assertion Block

```
Path.cwd.exists;
Path.new("/").exists;
var p = Path.new("file.txt");
!p.exists;
File.create(p);
p.exists;
```

- **isDir**

Whether `this` is a directory.

Assertion Block

```
Path.cwd.isDir;
var f = File.create("file.txt");
!f.asPath().isDir;
!Path.new("does/not/exist").isDir;
```

- **isReg**

Whether `this` is a regular file.

Assertion Block

```
var f = File.create("file.txt");
f.asPath().isReg;
!Path.cwd.isReg;
!Path.new("does/not/exist").isReg;
```

- **lastModifiedDate**

Last modified date of the path.

Assertion Block

```
var p = Path.new("test");
File.create(p);
0 <= Date.now - p.lastModifiedDate <= 5s;
```

- **open**

Open `this`. Return either a [Directory](#) or a [File](#) according the type of `this`. See [File](#) and [Directory](#).

Assertion Block

```
Path.new("/").open().isA(Directory);
```

- **readable**

Whether `this` is readable. Throw if does not even exist. On Windows, always returns true.

Assertion Block

```
Path.new(".").readable;
var p = Path.new("file.txt");
File.create(p);
p.readable;
System.system("chmod a-r %s" % p) == 0;
!p.readable;
System.system("chmod a+r %s" % p) == 0;
p.readable;
```

- **rename(*name*)**

Rename the file-system object (directory, file, etc.) pointed to by `this`, as *name*. Return `void`.

```
var dir1 = Directory.create("dir1");
var p = dir1.asPath();
p.rename("dir2").isVoid;
p.basename == "dir2";
```

Assertion Block

- **writable**

Whether `this` is writable. Throw if does not even exist. On Windows, always returns true.

```
Path.new(".").writable;
var p = Path.new("file.txt");
File.create(p);
p.writable;
System.system("chmod a-w %s" % p) == 0;
!p.writable;
System.system("chmod a+w %s" % p) == 0;
p.writable;
```

Assertion Block

22.46 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
```

urbiscript
Session

22.46.1 Prototypes

- **Object**

22.46.2 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
```

urbiscript
Session

22.46.3 Slots

- **Binding**

A class used to create pattern variables.

```
Pattern.Binding.new("a");
[00000000] var a
```

urbiscript
Session

- **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]
```

urbiscript
Session

- **match(*value*)**

Use *value* to unify the current pattern with this value. Return the status of the match.

- **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
};
```

Assertion Block

- 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);
```

Assertion Block

22.47 Position

This class is used to handle file locations with a line, column and file name.

22.47.1 Prototypes

- `Object`

22.47.2 Construction

Without argument, a newly constructed Position has its fields initialized to the first line and the first column.

```
Position.new();
[00000001] 1.1
```

urbiscript
Session

With a position argument *p*, the newly constructed Position is a clone of *p*.

```
Position.new(Position.new(2, 3));
[00000001] 2.3
```

urbiscript
Session

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
```

urbiscript
Session

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
```

urbiscript
Session

22.47.3 Slots

- `'+' (n)`

A new Position shifted from *n* columns to the right of `this`. 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);
```

Assertion
Block

- `'-' (n)`

A new Position shifted from *n* columns to the left of `this`. 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);
```

Assertion
Block

- `'<' (that)`

Order comparison of lines and columns.

```
Position.new(2, 3) < Position.new(2, 4);
Position.new(2, 3) < Position.new(3, 1);
```

Assertion
Block

- `'==' (that)`

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);
```

Assertion
Block

- **asString**

Present as ‘*file*:*line*.*column*’, the file name is omitted if it is not defined.

```
Position.new("file.u", 2, 3).asString() == "file.u:2.3";
```

Assertion Block

- **column**

The column number of the Position.

```
Position.new(2, 3).column == 3;
```

Assertion Block

- **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);
```

Assertion Block

- **file**

The [Path](#) of the Position file.

Assertion Block

```
Position.new("file.u", 2, 3).file == Path.new("file.u");
Position.new(2, 3).file == nil;
```

Assertion Block

- **line**

The line number of the Position.

```
Position.new(2, 3).line == 2;
```

Assertion Block

- **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);
```

22.48 Primitive

C++ routine callable from urbiscript.

22.48.1 Prototypes

- [Executable](#)

22.48.2 Construction

It is not possible to construct a Primitive.

22.48.3 Slots

- [apply\(*args*\)](#)

Invoke a primitive. The argument list, *args*, must start with the target.

Assertion Block

```
Float.getSlotValue("+").apply([1, 2]) == 3;  
String.getSlotValue("+").apply(["1", "2"]);
```

- [asPrimitive](#)

Return [this](#).

Assertion Block

```
Float.getSlotValue("+").asPrimitive() === Float.getSlotValue("+");
```

22.49 Process

A Process is a separated task handled by the underneath operating system.

Windows Issues

Process is not yet supported under Windows.

22.49.1 Example

The following examples runs the `cat` program, a Unix standard command that simply copies on its (standard) output its (standard) input.

```
urbiscript
Session
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
```

urbiscript
Session

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();
```

urbiscript
Session

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”.

```
urbiscript
Session
p.join();

p.status;
[00000008] exited with status 0
```

Finally we can check its output.

```
urbiscript
Session
p.stdout.asList();
[00000009] ["1", "2", "3"]
```

22.49.2 Prototypes

- [Object](#)

22.49.3 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.

```
urbiscript
Session
Process.new("cat", []);
[00000004] Process cat

Process.new("cat", ["--version"]);
[00000004] Process cat
```

22.49.4 Slots

- **asProcess**

Return `this`.

urbiscript
Session

```
do (Process.new("cat", []))
{
    assert (asProcess() === this);
}|;
```

- **asString**

`Process` and the name of the program.

Assertion
Block

```
Process.new("cat", ["--version"]).asString()
== "Process cat";
```

- **done**

Whether the process has completed its execution.

urbiscript
Session

```
do (Process.new("sleep", ["1"]))
{
    assert (!done);
    run();
    assert (!done);
    join();
    assert (done);
}|;
```

- **join**

Wait for the process to finish. Changes its status.

urbiscript
Session

```
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`.

urbiscript
Session

```
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.

Assertion
Block

```
Process.new("cat", ["--version"]).name == "cat";
```

- **run**

Launch the process. Changes its 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
```

urbiscript
Session

- **runTo**

- **status**

An object whose slots describe the status of the process.

- **stderr**

An [InputStream](#) (the output of the Process is an input for Urbi) to the standard error stream of the process.

```
do (Process.new("urbi-send", ["--no-such-option"]))
{
    run();
    join();
    assert
    {
        stderr.asList() ==
        ["urbi-send: invalid option: --no-such-option",
         "Try 'urbi-send --help' for more information."];
    };
} |;
```

urbiscript
Session

- **stdin**

An [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) 2004-2012 Gostai S.A.S.";
    };
} |;
```

urbiscript
Session

- **stdout**

An [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!"]);
} |;
```

urbiscript
Session

22.50 Profile

A `Profile` object contains information about the efficiency of a piece of code.

22.50.1 Example

22.50.1.1 Basic profiling

One can profile a piece of code with the `System.profile` function.

urbiscript
Session

```
var profile = System.profile(function() { echo("foo") });
[00000001] *** foo
[00001672] Profile(
  Yields:          0
  Total time (us):    1085
  Wall-clock time (us): 1085
  Function calls:    12
  Max depth:        5

  -----
  |   function   |   %   | cumulative (us) | self (us) | calls | self (us/call) |
  |             |       |               |           |       |           |
  +-----+-----+-----+-----+-----+-----+
  |     apply    | 26.91 |      292 |     292 |     1 |      292 |
  |     echo     | 25.35 |      567 |     275 |     1 |      275 |
  | <profiled> | 20.18 |      786 |     219 |     1 |      219 |
  |     send     |  6.36 |      855 |      69 |     1 |      69 |
  |     apply    |  4.61 |      905 |      50 |     1 |      50 |
  |     oget     |  4.24 |      951 |      46 |     2 |      23 |
  |     +        |  4.06 |      995 |      44 |     2 |      22 |
  | getSlotValue|  3.32 |     1031 |      36 |     1 |      36 |
  |     +        |  2.76 |     1061 |      30 |     1 |      30 |
  | asString   |  2.21 |     1085 |      24 |     1 |      24 |
,-----,-----,-----,-----,-----,-----,
)
```

The result is a `Profile` object that contains information about which functions where used when evaluating the given code, how many time they were called, how much time was spent in them, ... Lines are sorted by decreasing “self time”. Note that the `<profiled>` special function stands for the function given in parameter. Every line is represented by a `Profile.Function` object, see its documentation for the meaning of every column.

22.50.1.2 Asynchronous profiling

If the profiled code spawns asynchronous tasks via `detach` or `at` for instance, additional statistics will be included in the resulting `Profile` every time the detached code is executed. This is extremely useful to profile asynchronous code based on `at` for instance.

urbiscript
Session

```
var x = false;
// Make sure 'x' is visible whoever the caller of 'profiled' is.
import this.*;
function profiled()
{
  at (x)
    echo("true")
  onleave
    echo("false")
};

// This is the profiling for the creation of the 'at'. Note that the
// condition was already evaluated once, to see whether it should trigger
// immediately.
var profile_async = System.profile(getSlotValue("profiled"));
```

```
[00000000] Profile(
  Yields: 0
  Total time (us): 73
  Wall-clock time (us): 73
  Function calls: 9
  Max depth: 4

  -----
  | function | % | cumulative | self | calls | self | (us) | (us) | (us/call) |
  |         |   | (us)       | (us) |       |       |       |       |
  +-----+-----+-----+-----+-----+-----+
  | <profiled> | 46.58 | 34 | 34 | 1 | 34 | 34.58 | 34 | 34.58 |
  | at: { x } | 21.92 | 50 | 16 | 1 | 16 | 21.92 | 16 | 21.92 |
  | onEvent | 15.07 | 61 | 11 | 1 | 11 | 15.07 | 11 | 15.07 |
  | clone | 9.59 | 68 | 7 | 2 | 3.500 | 9.59 | 7 | 9.59 |
  | new | 2.74 | 72 | 2 | 2 | 1 | 2.74 | 2 | 2.74 |
  | init | 1.37 | 73 | 1 | 2 | 0.500 | 1.37 | 1 | 1.37 |
  +-----+-----+-----+-----+-----+-----+
)

// Trigger the at twice.
x = true|;
[00106213] *** true
x = false|;
[00172119] *** false

// The profile now includes additional statistic about the evaluations of
// the condition and the bodies of the at.
profile_async;
[00178623] Profile(
  Yields: 2
  Total time (us): 251
  Wall-clock time (us): 251
  Function calls: 29
  Max depth: 4

  -----
  | function | % | cumulative | self | calls | self | (us) | (us) | (us/call) |
  |         |   | (us)       | (us) |       |       |       |       |
  +-----+-----+-----+-----+-----+-----+
  | <profiled> | 13.55 | 34 | 34 | 1 | 34 | 13.55 | 34 | 13.55 |
  | event | 11.55 | 63 | 29 | 1 | 29 | 11.55 | 29 | 11.55 |
  | send | 11.16 | 91 | 28 | 2 | 14 | 11.16 | 14 | 11.16 |
  | event | 10.76 | 118 | 27 | 1 | 27 | 10.76 | 27 | 10.76 |
  | at: { x } | 10.76 | 145 | 27 | 3 | 9 | 10.76 | 27 | 10.76 |
  | clone | 5.98 | 183 | 15 | 4 | 3.750 | 5.98 | 15 | 3.750 |
  | echo | 5.58 | 212 | 14 | 2 | 7 | 5.58 | 14 | 7 |
  | onEvent | 4.38 | 223 | 11 | 1 | 11 | 4.38 | 11 | 11 |
  | + | 1.99 | 228 | 5 | 4 | 1.250 | 1.99 | 5 | 1.250 |
  | new | 1.99 | 233 | 5 | 4 | 1.250 | 1.99 | 5 | 1.250 |
  | asString | 0.80 | 242 | 2 | 2 | 1 | 0.80 | 2 | 1 |
  | init | 0.80 | 248 | 2 | 4 | 0.500 | 0.80 | 2 | 0.500 |
  +-----+-----+-----+-----+-----+-----+
)
```

Note that part of the internal machinery shows in these figures (and left visible on purpose). For instance the three additional calls to `new` correspond to the creation of the `changed` event.

22.50.2 Prototypes

- [Object](#)

22.50.3 Construction

Profile objects are not meant to be cloned as they are created by `System.profile` internal machinery.

22.50.4 Slots

- `calls`

Return a `List` of `Profile.Function` objects. Each element of this list describes, for a given function, statistics about how many times it is called and how much time is spent in it.

- `Function`

See `Profile.Function`.

- `maxFunctionCallDepth`

The maximum function call depth reached.

- `totalCalls`

The total number of function calls made.

- `totalTime`

The total CPU time. It can be higher than the wall clock time on multi-core processors for instance.

- `wallClockTime`

The time spent between the beginning and the end as if measured on a wall clock.

- `yields`

The scheduler has to execute many coroutines in parallel. A coroutine yields when it gives the opportunity to another to be executed until this one yields and so on... This slot contains the number of scheduler yields.

22.51 Profile.Function

A Function object contains information about calls of a given function during a profiling operation.

22.51.1 Prototypes

- `Object`

22.51.2 Construction

Function objects are not meant to be cloned as they are created by `System.profile` internal machinery. As an example, let us profile the traditional factorial function.

urbiscript
Session

```
function Float.fact()
{
    if (this <= 1)
        this
    else
        this * (this - 1).fact();
};
```

To improve the consistency of the results, you are advised to run the profiling system (and function to profile) once before the real measure. This ensures that all the code is loaded and ready to be run: profiling will be about computations, not about initializations.

urbiscript
Session

```
System.profile(function() { 20.fact() });

var profile = System.profile(function() { 20.fact() });
[00009050] Profile(
  Yields: 0
  Total time (us): 171
  Wall-clock time (us): 171
  Function calls: 79
  Max depth: 22

  -----
  |   function   |   %   | cumulative | self | calls | self | 
  |             |   (us) |           | (us) |       | (us/call) | 
  |-----+-----+-----+-----+-----+-----+
  |   fact     | 70.18 |      120 |    120 |     20 |      6 |
  |   -         | 10.53 |      138 |     18 |    19 |  0.947 |
  |   <=        |  8.77 |      153 |     15 |    20 |  0.750 |
  |   *         |  8.19 |      167 |     14 |    19 |  0.737 |
  | <profiled> |  2.34 |      171 |      4 |     1 |      4 |
  ,-----,-----,-----,-----,-----,-----,
)

profile.calls[0];
[00123833] Function('fact', 20, 0.000120, 0.000006)
profile.calls[0].isA(Profile.Function);
[00123933] true
```

22.51.3 Slots

- `calls`

The number of times this function was called during the profiling.

Assertion
Block

```
// Example continued from Construction section.
profile.calls[0].calls == 20;
```

- `name`

The name of the function called.

Assertion
Block

```
// Example continued from Construction section.  
profile.calls[0].name == "fact";
```

- **selfTime**

Total CPU time spent in all calls of the function.

Assertion Block

```
// Example continued from Construction section.  
profile.calls[0].selfTime.isA(Float);
```

- **selfTimePer**

Average CPU time spent in one function call. It is computed as the ratio of **selfTime** divided by **calls**.

Assertion Block

```
// Example continued from Construction section.  
do (profile.calls[0])  
{  
    selfTimePer == selfTime / calls;  
}
```

22.52 PseudoLazy

22.52.1 Prototypes

- [Lazy](#)

22.52.2 Slots

- [eval](#)

22.53 PubSub

PubSub provides an abstraction over `Barrier` `Barrier` to queue signals for each subscriber.

22.53.1 Prototypes

- `Object`

22.53.2 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
```

urbiscript
Session

22.53.3 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)
}|;
```

urbiscript
Session

- `subscribe`

Create a `Subscriber` and insert it inside the list of subscribers.

```
var sub = ps.subscribe() |
ps.subscribers == [sub];
[00000000] true
```

urbiscript
Session

- `Subscriber`

See `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] []
```

urbiscript
Session

22.54 PubSub.Subscriber

Subscriber is created by [PubSub.subscribe](#). It provides methods to access to the list of values published by PubSub instances.

22.54.1 Prototypes

- [Object](#)

22.54.2 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
```

urbiscript
Session

22.54.3 Slots

- [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]
```

urbiscript
Session

- [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
```

urbiscript
Session

22.55 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 21.7.6).

22.55.1 Prototypes

- `Object`

22.55.2 Slots

- `all(fun)`

Return whether all the members of the target verify the predicate `fun`.

Assertion Block

```
// 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`.

Assertion Block

```
// 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.

urbiscript Session

```
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.

urbiscript Session

```
{  
    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.

urbiscript Session

```
{  
    var res = [];  
    for| (var i : range)  
        res << i;
```

```
    assert(res == [10, 20, 30]);  
};
```

22.56 Regexp

A `Regexp` is an object which allow you to match strings with a regular expression.

22.56.1 Prototypes

- `Container`
- `Object`

22.56.2 Construction

A `Regexp` is created from a regular expression once and for all; it can be used several times to match with other strings.

```
Regexp.new(".");
[00000001] Regexp(".")

var num = Regexp.new("\d+\.\d+");
[00000004] Regexp("\d+\.\d+")
"1.3" in num;
[00019618] true
"1." in num;
[00023113] false
```

urbiscript
Session

urbiscript supports Perl regular expressions, see [the perlre man page](#).

Expressions cannot be empty, and must be syntactically correct.

```
Regexp.new("");
[00000001:error] !!! new: invalid regular expression: empty expression: ''

Regexp.new("(");
[00003237:error] !!! new: invalid regular expression:\n\nunmatched marking parenthesis ( or \(: '(>>HERE>>''

Regexp.new("*");
[00004372:error] !!! new: invalid regular expression:\n\nthe repeat operator "*" cannot start a regular expression: '>>HERE>>*'
```

urbiscript
Session

22.56.3 Slots

- `[]` `(n)`

Same as `this.matches[n]`.

```
var d = Regexp.new("(1+)(2+)(3+)")|;
assert
{
    "01223334" in d;
    d[0] == "122333";
    d[1] == "1";
    d[2] == "22";
    d[3] == "333";
};
d[4];
[00000009:error] !!! []: out of bound index: 4
```

urbiscript
Session

- `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+)?\")";
```

Assertion
Block

- **asString**

The regular expression that was compiled.

```
Regexp.new("abc").asString() == "abc";
Regexp.new("\d+(\.\d+)?").asString() == "\d+(\.\d+)?";
```

Assertion Block

- **has(str)**

An experimental alias to `match`, so that the infix operators `in` and `not in` can be used (see [Section 21.1.8.7](#)).

```
"23.03"    in Regexp.new("^\\d+\\.\\d+$");
"-3.14" not in Regexp.new("^\\d+\\.\\d+$");
```

Assertion Block

- **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("foooooobar");
  !r.match("far");
};

// (), grouping.
r = Regexp.new("f(oo)*bar")|
assert
{
  r.match("fooobar");
  !r.match("fooobar");
};
```

urbiscript Session

- **matches**

If the latest `match` was successful, the matched groups, as delimited by parentheses in the regular expression; the first element being the whole match. Otherwise, the empty list. See also `[]`.

```
var re = RegExp.new("[a-zA-Z0-9._]+@[a-zA-Z0-9._]+")|;
assert
{
    re.match("Someone <someone@somewhere.com>");
    re.matches == ["someone@somewhere.com", "someone", "somewhere.com"];

    "does not match" not in re;
    re.matches == [];
};
```

22.57 Semaphore

Semaphore are useful to limit the number of access to a limited number of resources.

22.57.1 Prototypes

- [Object](#)

22.57.2 Construction

A **Semaphore** can be created with as argument the number of processes allowed to enter critical sections at the same time.

urbiscript
Session

```
Semaphore.new(1);
[00000000] Semaphore_0x8c1e80
```

22.57.3 Slots

- [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.

- [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.

urbiscript
Session

```
{
  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
```

- **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
```

urbiscript
Session

- **v**

Historical synonym for [release](#).

22.58 Serializables

This object is used to store the set of prototypes that support exchange data between C++ and urbiscipt. See also [UValueSerializable](#) and [Section 26.18.2](#).

22.58.1 Prototypes

- [Object](#)

22.58.2 Slots

Nothing specific. Slots should be mapping from “class” name to “class” implementation. See [Section 26.18.2](#).

22.59 Server

A *Server* can listen to incoming connections. See [Socket](#) for an example.

22.59.1 Prototypes

- [Object](#)

22.59.2 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"]
```

urbiscript
Session

22.59.3 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](#) as the server.

```
at (s.connection?(var socket))
{
    // This code is run at each connection. 'socket' is the incoming
    // connection.
};
```

urbiscript
Session

- [getIoService](#)

Return the [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
```

urbiscript
Session

- [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
```

urbiscript
Session

- [sockets](#)

The list of the sockets created at each incoming connection.

22.60 Singleton

A *singleton* is a prototype that cannot be cloned. All prototypes derived of `Singleton` are also singletons.

22.60.1 Prototypes

- `Object`

22.60.2 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);
```

urbiscript
Session

22.60.3 Slots

- `clone`
Return `this`.
- `'new'`
Return `this`.

22.61 Slot

A slot is an intermediate object that embodies the concept of “variable” or “field” in urbiscript. It contains an underlying value, meta-information about this value, and slots to alter the behavior of read and write operations.

22.61.1 Accessing the slot object

Section [Section 21.4.1](#) describes how to access a slot object.

22.61.2 Key features

The contained value returned by default when `Object.getSlotValue` is called is stored in the `value` slot.

The `changed` slot is an `Event` that is triggered each time the slot is written to.

Setters and getters to modify the slot behaviors can be installed by writing to `set`, `get`, `oset` and `oget`.

Two slots can be linked together to build dataflows using operator `>>`

22.61.3 Split mode

It can sometimes be convenient to store two values in one slot, one which is read, and the other written. For instance, the `val` slot of a rotational motor Object can be the current motor position when reading, and a target position to reach when writing.

This behavior is controlled by the `split` slot.

22.61.4 Construction

Slots are automatically created when `Object.setSlot` is called.

22.61.5 Prototypes

- `Object`

22.61.6 Slots

- `'<<'(slot)`

Bounces to `>>` reversing the two Slots.

- `'>>'(slot)`

The `>>` operator connects two `Slot` together through a `Subscription`. Each time the source Slot is updated, its new value is written to the target Slot. This function should be used to bridge a component producing an output value to a component expecting an input value. It returns a `Subscription` that can be used to configure the link, and gather statistics.

```
var tick = 0;
var tack = 0;
var sub = &tick >> &tack;
[00000001] Slot_0x42389d88 >> Slot_0x42387c88
tack->set = function(v) { echo("tack " + v)}|;
timeout(10.5s) every|(1s) tick++,
sleep(1.5s) | sub.callCount;
[00000002] *** tack 1
[00000003] *** tack 2
[00000004] 2
sub.enabled = false| sleep(1s)| sub.enabled = true|;
sleep(2s);
```

urbiscript
Session

```
[00000002] *** tack 4
[00000003] *** tack 5
sub.disconnect();
sleep(2s);
```

- **changed**

Contains an [Event](#) which is emitted each time the slot value is written to. It is used by the system to implement the [watch](#)-expression ([Section 21.11.3](#)).

urbiscript
Session

```
var x = [] |;

at (x->changed?) // Same thing as &x.changed
echo("x->changed");

x = [1] |;
[00092656] *** x->changed
x = [1, 2] |;
[00092756] *** x->changed
```

Even if the slot is assigned to the very same value, the `x->changed` event is emitted.

urbiscript
Session

```
x = x |;
[00092856] *** x->changed
```

This is different from checking value *updates*. In the following example, `x` is not rebound to another list, it is the content of the list that changes, therefore the `changed` event is *not* fired:

```
x << 3;
[00092866] [1, 2, 3]
```

urbiscript
Session

To monitor changes of value, use the [watch](#)-construct ([Section 21.11.3](#)).

- **constant**

Defines whether a slot can be assigned a new value.

urbiscript
Session

```
var c = 0;
[00000000] 0
c = 1;
[00000000] 1

c->constant = true;
[00000000] true
c = 2;
[00000000:error] !!! cannot modify const slot

c->constant = false;
[00000000] false
c = 3;
[00000000] 3
```

urbiscript
Session

A new slot can be declared constant when first defined, in which case the initial value is required.

urbiscript
Session

```
const var two;
[00000030:error] !!! syntax error: const declaration without a value
const var two = 2;
[00000036] 2
two = 3;
[00000037:error] !!! cannot modify const slot
two->constant;
[00000038] true
```

- **copy(*targetObject*, *targetName*)**

Duplicate the Slot to the slot *targetName* of object *targetObject*.

- **copyOnWrite**

If set to false, disables copy on write behavior for this slot (see [Section 21.4.5.3](#)).

urbiscript
Session

```
class a {
    var x = 0;
    var y = 0;
}|;
var b = a.new();
a.&x.copyOnWrite = false|;
b.x = 1| b.y = 1|;
assert
{
    a.x == 1;
    a.y == 0;
};
```

- **get**

Together with **oget**, this slot can be set with a function that will be called each time the slot is accessed. If one exists, the **value** slot is ignored, and the value returned from **get** or **oget** is used instead. Only one of them can be set. **get** is called on the slot itself with no other argument, whereas **oget** is called on the object who first owned the slot, with the slot as optional argument.

Use **get** when all the information needed to compute the value are in the slot itself.

urbiscript
Session

```
var counter = 0|;
var &counter.increment = 2|;
counter->get = function()
{ value += increment | value}|;
counter;
[00000001] 2
counter;
[00000002] 4
counter->increment = 3|;
counter;
[00000003] 7
```

Use **oget** when the computation needs information in the object owning the slot. Using **oget** is better than using **get** with a closure.

urbiscript
Session

```
class Vector
{
    var x;
    var y;
    function init(x_, y_)
    {
        x = x_;
        y = y_;
    };
    var length;
    length->oget = function() // or function(slot)
    {
        ((x*x)+(y*y)).sqrt()
    };
}|;
var v = Vector.new(2, 0);
v.length;
[00000001] 2
```

- **oget**

Similar to `get`, but with a different signature: the callback function is called on the object owning the slot, instead of the slot itself.

The `oget` slot can be changed using `get x` syntax described in [Section 8.8](#).

- **set**

Together with `oset`, this slot can be set with a function that will be called each time the slot is written to. If one exists, the default behavior that simply writes the value to `value` is disabled. `set` is called on the slot itself, with the value as its sole argument. `oset` is called on the object owning the slot, with the value and optionally the slot as arguments.

Use `set` when the operation performed by your function needs no information outside the slot.

`set` and `oset` can either return the value or write it to the `value` slot.

```
var positiveInt = 0|;
positiveInt->set = function(v)
{
    if (v >= 0)
        v // return the value
    else
        {} // return void: ignore the value
}|;
positiveInt = 5 | positiveInt;
[00000001] 5
positiveInt = -1 | positiveInt;
[00000001] 5
```

urbiscript
Session

Only the object owning the slot should use `set` and `oset`. Other objects should use `changed`, watch constructs (([Section 21.11.3](#))) or `>>`.

```
var integer = 0|;
integer->set = function(v) { v.round() }|;
integer = 1.6 | integer;
[00000001] 2
```

urbiscript
Session

Use `oset` when you need access to the object from your function.

```
class Vector
{
    var x;
    var y;
    function init(x_, y_)
    {
        x = x_;
        y = y_;
    };
    var length;
    length->oget = function(slot) { ((x*x)+(y*y)).sqrt() };
    // Change the vector length, keeping the same direction
    length->oset = function(value) // or function(value, slotName)
    {
        var ratio = value / length;
        x *= ratio;
        y *= ratio;
        {}; // return no value
    };
}|;
var v = Vector.new(2, 0)|;
v.length *= 2|;
v.x;
[00000001] 4
```

urbiscript
Session

```
v.y;
[00000002] 0
```

- **oset**

Similar to `set` but with a different signature: the function is called on the object owning the slot instead of the slot itself. The function can take 1 or two arguments: the input value, and optionally the name of the slot.

The `oset` slot can be changed using `set x` syntax described in [Section 8.8](#).

- **notifyAccess(*onAccess*)**

Deprecated, use `get` or `oget`. Similar to the C++ `UNotifyAccess`, calls `onAccess` each time the Slot is accessed (read).

urbiscript
Session

```
var Global.counter = 0|
var Global.access = 0|
var accessHandle = Global.&access.notifyAccess(closure() {
    Global.access = ++Global.counter
})|
import Global.*;
assert
{
    access == 1;
    access == 2;
    access == 3;
};
Global.&access.removeNotifyAccess(accessHandle)|;
assert
{
    access == 3;
    access == 3;
};
```

- **notifyChange(*onChange*)**

Deprecated, use `>>` or `changed`. Similar to the C++ `UNotifyChange` (see [Section 26.5](#)), register `onChange` and call it each time this Slot is written to. Return an identifier that can be passed to `removeNotifyChange` to unregister the callback. Subscribing to the `changed` event has a similar effect.

urbiscript
Session

```
var Global.y = 0|
var handle = Global.&y.notifyChange(closure() {
    echo("The value is now " + Global.y)
})|
Global.y = 12;
[00000001] *** The value is now 12
[00000002] 12
Global.&y.removeNotifyChange(handle)|;
Global.y = 13;
[00000003] 13
```

- **notifyChangeOwned(*onChangeOwned*)**

Deprecated, this call now just set the `set` slot. Similar to the C++ `UNotifyChange` for a split Slot (see [Section 26.5](#)), register `onChange` and call it each time this `UVar` is written to. Return an identifier that can be passed to `removeNotifyChangeOwned` to unregister the callback.

- **outputValue**

The value that is returned when a read occurs, if in `split` mode.

- **owned**

True if the `Slot` is in *split mode*, that is if it contains both a sensor and a command value. This name is for backward compatibility.

- **removeNotifyAccess(*id*)**

Disable the notification installed as *id* by `notifyAccess`.

- **removeNotifyChange(*id*)**

Disable the notification installed as *id* by `notifyChange`.

- **removeNotifyChangeOwned(*id*)**

Disable the notification installed as *id* by `notifyChangeOwned`.

- **setOutputValue(*val*)**

In `split` mode, update output value slot `outputValue` and trigger `changed`.

- **split**

Indicates that the slot has both an input value and an output value. The input value is written to by external code, and the slot will act on it. The output value is updated by the slot itself and made visible to external code.

This feature's intended use is to have both a sensor value and an actuator command accessible on the same slot.

More formally, once activated by setting the `split` slot to true:

- Writing to the slot no longer triggers `changed`, but still updates `value` and calls `set`.
- Reading the slot returns `outputValue` instead of `value`

Practically, code external to the slot owner continue to use the slot as usual. The object owning the slot must:

- Directly access `value`, or use a setter to read the value written by external code.
- Call `setOutputValue` to update the value visible to external call. This call will trigger the `changed` event.

urbiscript
Session

```
class Motor
{
    function init()
    {
        // our val is both motor command, and motor current position
        var this.val = 0;
        var this.running = false;
        var this.runTag = Tag.new();
        &val.owned = true;
        // Initialize current position
        &val.outputValue = 0;
        // Install a setter, that will be called when 'val' is written to.
        // Use oset, so that the 'this' passed to oset is the motor.
        set val(command)
        {
            setCommand(command);
        };
    };

    function setCommand(command)
    { // This function is called when someone writes 'command' to 'val'.
        var same = (command == &val.outputValue);
        echo("Motor command is now " + command);
        if (same && running)
        { // Target reached, stop motor control loop
    }
```

```

        runTag.stop();
        running = false;
    }
    else if (!same && !running)
    { // Start motor control loop
        detach({runMotor()});
    };
    // Return command so that the value gets written to the slot
    command
};

function runMotor()
{ // Move current motor position toward target, one unit per second.
    running = true;
    runTag: while(&val.outputValue != &val.value)
    {
        &val.setOutputValue(
            &val.outputValue + 1 * (&val.value -&val.outputValue).sign());
        sleep(1s);
    };
    running = false;
}
};

var m = Motor.new();
[00000001] Motor_0x42347788
// Send a command to motor by writing to val
m.val = 5!;
[00000002] *** Motor command is now 5
sleep(1.5s);
// Read motor current position by reading val.
m.val;
[00000003] 2
// Changed triggers when current position is updated by the Motor object.
at(m.val->changed?) echo("Motor position is " + m.val);
sleep(10s);
[00000004] *** Motor position is 3
[00000005] *** Motor position is 4
[00000006] *** Motor position is 5

```

- **type**

If set, only values inheriting from **type** will be accepted into the slot.

```

var h = 0;
[00000001] 0
h->type = Float|;
h = 1;
[00000001] 1
h = "hello";
[00000002:error] !!! unexpected "hello", expected a Float
h;
[00000001] 1

```

urbiscript
Session

- **value**

The underlying value.

```

var z = 2;
[00000001] 2
&z.value;
[00000001] 2
&z.value = 3;
[00000002] 3
z;
[00000002] 3

```

urbiscript
Session

22.62 Socket

A *Socket* can manage asynchronous input/output network connections.

22.62.1 Example

The following example demonstrates how both the *Server* and *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*.

```
urbiscript
Session
class 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";
}
};

// Import lobby so that Dialog is visible from everywhere.
import this.*;
```

The server, an instance of *Server*, expects incoming connections, notified by *Server.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);
        echo("server: " + reply);
        socket.write(reply);
        if (reply == "off")
            socket.disconnect();
    };
};
```

urbiscript
Session

The client, an instance of *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);
        echo("client: " + reply);
        write(reply);
    };
};
```

urbiscript
Session

```
 }!;
```

The server is then activated, listening to incoming connections on a port that will be chosen by the system among the free ones.

urbiscript
Session

```
server.listen("localhost", "0");
clog << "connecting to %s:%s" % [server.host, server.port];
```

The client connects to the server, and initiates the dialog.

urbiscript
Session

```
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.

urbiscript
Session

```
waituntil(client.disconnected?) | echo("done");
[00000004] *** server: Hey!
[00000005] *** client: Hey you doin'?
[00000006] *** server: Whazaaa!
[00000007] *** client: See ya.
[00000008] *** server: off
[00000008] *** done
```

There is a catch though: the last message was still being processed by the system, and arrived *after* we waited for the `client` is to be `disconnected`:

```
sleep(100ms);
[00000009] *** client: off
```

urbiscript
Session

This is because both the last message sent from the server, and the disconnection request have arrived at the same “instant” to the client. Both are processed asynchronously, in particular the message reception since the code used an *asynchronous at* ([Section 21.11.1.3](#)). In the case of asynchronous event handling, this is no guarantee on the order of event processing. This can be addressed by *synchronous* event processing on the client side; pay extra attention to the `sync` qualifier passed to `at`:

```
var syncClient =
  do (Socket.new())
{
  at sync (received?(var data))
  {
    var reply = Dialog.reply(data);
    write(reply);
    echo("syncClient: " + reply);
  };
}!;

syncClient.connect(server.host, server.port);
echo("syncClient: " + Dialog.lines[0]);
syncClient.write(Dialog.lines[0]);
[00000003] *** syncClient: Hi!
waituntil(syncClient.disconnected?) | echo("done");
[00000004] *** server: Hey!
[00000005] *** syncClient: Hey you doin'?
[00000006] *** server: Whazaaa!
[00000007] *** syncClient: See ya.
[00000008] *** server: off
[00000008] *** syncClient: off
[00000008] *** done
```

urbiscript
Session

This time, as one would expect at first, the `*** done` appears after the full dialog was performed.

22.62.2 Prototypes

- `Object`

22.62.3 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();
```

urbiscript
Session

22.62.4 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` used by this socket. Only the default `IoService` is automatically polled.

- `host`

The remote host of the connection.

- `isConnected`

Whether `this` is connected.

```
! Socket.new().isConnected;
```

Assertion
Block

- `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`.

- `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.
- **syncWrite(*data*)**
Similar to `write`, but forces the operation to complete synchronously. Synchronous and asynchronous write operations cannot be mixed.
- **write(*data*)**
Sends *data* through the connection.

22.63 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` are filled with `StackFrames` when the is thrown.

```
urbiscript
Session
//#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.

22.63.1 Prototypes

- [Object](#)

22.63.2 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](#) from which it has been called.

```
urbiscript
Session
StackFrame.new("inner",
  Location.new(
    Position.new("foo.u", 7, 5),
    Position.new("foo.u", 7, 10)
  )
);
[00000001] foo.u:7.5-9: inner
```

22.63.3 Slots

- [asString](#)

Clean display of the call location.

```
urbiscript
Session
getStackFrame();
[00000004] foo.u:7.5-11: inner
```

- [location](#)

[Location](#) of the function call.

```
urbiscript
Session
getStackFrame().location;
[00000003] foo.u:7.5-11
```

- **name**

String, representing the name of the called function.

urbiscript
Session

```
getStackFrame().name;  
[00000002] "inner"
```

22.64 Stream

This is used to factor code between `InputStream` and `OutputStream`.

22.64.1 Prototypes

- `Object`

22.64.2 Construction

Streams are not meant to be built, rather, use `InputStream` or `OutputStream`.

When a stream (`OutputStream` or `InputStream`) is opened on a `File`, that File cannot be removed. On Unix systems, this is handled gracefully (the references to the file are removed, but the content is still there for the streams that were already bound to this file); so in practice, the File appears to be removable. On Windows, the File cannot be removed at all. Therefore, do not forget to close the streams you opened.

22.64.3 Slots

- `close`

Flush the buffers, close `this`, return `void`. Raise an error `this` is closed.

urbiscript
Session

```
var i = InputStream.new(File.create("file.txt"))|;
assert(i.close().isVoid);
i.close();
[00000001:error] !!! close: stream is closed

var o = OutputStream.new(File.create("file.txt"))|;
assert(o.close().isVoid);
o.close();
[00000002:error] !!! close: stream is closed
```

The file is actually closed with `this` is destroyed, if it was not already closed.

- `closed`

Whether `this` is closed.

Assertion
Block

```
var i = InputStream.new(File.create("file.txt"));

!i.closed;
i.close().isVoid;
i.closed;
```

22.65 String

A *string* is a sequence of characters.

22.65.1 Prototypes

- Comparable
- Orderable
- RangeIterable

22.65.2 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 21.1.6.6](#) for details and examples.

A null String can also be obtained with `String.new()`.

```
String.new() == "";
String == "";
"123".new() == "123";
```

Assertion Block

22.65.3 Slots

- `'%'(args)`

It is an equivalent of `Formatter.new(this) % args`. See [Formatter](#) and [FormatInfo](#), which provide more examples.

Assertion Block

```
"%s + %s = %s" % [1, 2, 3] == "1 + 2 = 3";

var f = "%10s level: %-4.1f%";
f % ["Battery", 83.3] == "    Battery level: 83.3%";
f % [ "Sound", 60 ] == "      Sound level: 60 %";
```

Assertion Block

- `'*(n)`

Concatenate `this`*n* times.

Assertion Block

```
"foo" * 0 == "";
"foo" * 1 == "foo";
"foo" * 3 == "foofoofoo";
```

Assertion Block

- `'+(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"));
```

Assertion Block

- `'=='` (*that*)

Whether `this` and `that` are the same string.

```
"" == "";           !("" != "");
!("" == "\0");     "" != "\0";

"0" == "0";        !"0" != "0";
!"0" == "1");     "0" != "1";
!"1" == "0");     "1" != "0";
```

Assertion Block

- `'[]'` (*from*, *to* = *from* + 1)

The sub-string starting at *from*, up to and not including *to*.

```
"foobar"[0, 3] == "foo";
"foobar"[0] == "f";
```

Assertion Block

The indexes must be integers from `-size` up to `size` + 1 (inclusive). Independently of their order as integers, *from* must be *before* or equal to *to*.

```
"foobar"[1.1];
[00051825:error] !!! []: invalid index: 1.1
"foobar"[-7];
[00051841:error] !!! []: invalid index: -7
"foobar"[6];
[00051853:error] !!! []: invalid index: 6

"foobar"[3, 1];
[00051953:error] !!! []: range starting after its end: 3, 1
```

urbiscript Session

Negative indexes means “from the end”.

Assertion Block

```
"foobar"[-1] == "r";
"foobar"[-6] == "f";
"foobar"[-1, 0] == "r";
"foobar"[-6, -3] == "foo";
// Valid since position 2 is before position -1.
"foobar"[2, -1] == "oba"
```

- `'[]='` (*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*. The condition on *from* and *to* are the same of for `'[]'`. Return *other*.

Beware that this routine is imperative: it changes the value of `this`.

urbiscript Session

```
var s1 = "foobar" | var s2 = s1 |
assert
{
  (s1[0, 3] = "quux") == "quux";
  s1 == "quuxbar";
  s2 == "quuxbar";
  (s1[4, 7] = "") == "";
  s2 == "quux";

  (s1[-3, -1] = "UU") == "UU";
  s1 == "qUUx";
  (s1[-1] = "X") == "X";
  s1 == "qUX";
```

- `asFloat`

The value of `this` as a `Float`. See [Section 21.1.6.4](#) for their syntax.

Assertion Block

```
"23".asFloat() == 23;
"23.03".asFloat() == 23.03;
"123_456_789".asFloat() == 123_456_789;
"12_34_56_78_90".asFloat() == 12_34_56_78_90;
"1_2__3___45".asFloat() == 1_2__3___45;
"1_2.3_4".asFloat() == 1_2.3_4;
"0xFFFF_FFFF".asFloat() == 0xFFFF_FFFF;
"1e1_0".asFloat() == 1e1_0;
```

Raise an error on invalid numbers.

urbscript
Session

```
"123abc".asFloat();
[00000001:error] !!! asFloat: invalid number: "123abc"
"OxabcdEfg".asFloat();
[00061848:error] !!! asFloat: invalid number: "OxabcdEfg"
"1.2_".asFloat();
[00048342:error] !!! asFloat: invalid number: "1.2_"
```

- **asList**

A List of one-letter Strings that, concatenated, equal `this`. This allows to use `for` to iterate over the string.

urbscript
Session

```
assert("123".asList() == ["1", "2", "3"]);
for (var v : "123")
    echo(v);
[00000001] *** 1
[00000001] *** 2
[00000001] *** 3
```

- **asPrintable**

`this` as a literal (escaped) string.

Assertion
Block

```
"foo".asPrintable() == "\"foo\"";
"foo".asPrintable().asPrintable() == "\"\\\\\"foo\\\\\"\"";
```

- **asString**

`this`.

```
var s = "\"foo\"";
s.asString() === s;
```

Assertion
Block

- **closest(*set*)**

The closest (in the sense of `distance`) string in `set` to `this`. If there is no convincing match, return `nil`.

```
"foo".closest(["foo", "baz", "qux", "quux"]) == "foo";
"bar".closest(["foo", "baz", "qux", "quux"]) == "baz";
"FOO".closest(["foo", "bar", "baz"]) == "foo";
"qux".closest(["foo", "bar", "baz"]) == nil;
```

Assertion
Block

- **distance(*other*)**

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;

"foo".distance("fozo") == 1; // Deletion.
"fozo".distance("foo") == 1; // Insertion.
```

Assertion
Block

```
"ofzo".distance("fozo") == 1; // Transposition.  
"fpzo".distance("fozo") == 1; // Substitution.  
  
"fpzzo".distance("fozo") == 2; // Substitution and insertion.
```

- **empty**

Whether this is the empty string.

```
"".empty;  
!"x".empty;
```

Assertion Block

- **find(*pattern*, *position* = 0)**

Search for the *pattern* string in **this** starting from the *position*. When the *pattern* is not found -1 is returned.

```
"Hello, World!".find("o") == 4;  
"Hello, World!".find("o", 4) == 4;  
"Hello, World!".find("o", 5) == 8;  
"Hello, World!".find("o", 9) == -1;
```

Assertion Block

- **fresh**

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";
```

Assertion Block

- **fromAscii(*v*)**

The character corresponding to the integer *v* according to the ASCII coding. See also **toAscii**.

```
String.fromAscii(97) == "a";  
String.fromAscii(98) == "b";  
String.fromAscii(0xFF) == "\xff";  
[0, 1, 2, 254, 255]  
  .map(function (var v) { String.fromAscii(v) })  
  .map(function (var v) { v.toAscii() })  
  == [0, 1, 2, 254, 255];
```

Assertion Block

- **hash**

A **Hash** object corresponding to this string value. Equal strings (in the sense of '`==`') have equal hashes. See **Object.hash**.

```
"".hash().isA(Hash);  
"foo".hash() == "foo".hash();  
"foo".hash() != "bar".hash();
```

Assertion Block

- **isAlnum**

Whether all the characters of **this** are digits or/and characters (see [Table 22.1](#)).

```
"".isAlnum();  
"123".isAlnum();  
"abc".isAlnum(); "1b".isAlnum();
```

Assertion Block

- **isAlpha**

Whether all the characters of **this** are letters, upper or lower case (see [Table 22.1](#)).

```
"".isAlpha();  
"abcABC".isAlpha();  
!"123".isAlpha(); "b".isAlpha();
```

Assertion Block

ASCII values	Characters	isCntrl	isSpace	isBlank	isUpper	isLower	isAlpha	isDigit	isXdigit	isAlnum	isPunct	isGraph	isPrint
0x00 .. 0x08		•											
0x09	\t	•	•	•									
0x0A .. 0x0D	\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)	•											

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).

Note the following equivalences:

```
isAlnum ≡ isAlpha || isDigit
isAlpha ≡ isLower || isUpper
isGraph ≡ isAlpha || isDigit || isPunct
```

Table 22.1: Character handling functions

- `isAscii`

Whether all the characters of `this` are ascii characters (see Table 22.1).

```
"".isAscii();
"abc123".isAscii();
!"é".isAscii(); !"è".isAscii();
```

Assertion Block

- `isBlank`

Whether all the characters of `this` are spaces or tabulations (see Table 22.1).

```
"".isBlank();
"\t \t ".isBlank();
!"123".isBlank(); !"\\v".isBlank();
```

Assertion Block

- `isCntrl`

Whether all the characters of `this` are control characters (non-printable) (see Table 22.1).

```
"".isCntrl();
"\t\n\f".isCntrl();
"\10".isCntrl(); !"abc".isCntrl();
```

Assertion Block

- `isDigit`

Whether all the characters of `this` are decimal digits (see Table 22.1).

```
"".isDigit();
"0123456789".isDigit();
!"a".isDigit(); !"0x10".isDigit();
```

Assertion Block

- **isGraph**

Whether all the characters of `this` are printable characters (not a space) (see [Table 22.1](#)).

```
"".isGraph();
"abc123".isGraph();
"{}[]".isGraph();
!"\t\n\r".isGraph(); !" ".isGraph();
```

Assertion Block

- **isLower**

Whether all the characters of `this` are lower characters (see [Table 22.1](#)).

```
"".isLower();
"abc".isLower();
!"123".isLower(); !"A".isLower();
```

Assertion Block

- **isPrint**

Whether all the characters of `this` are printable characters (see [Table 22.1](#)).

```
"".isPrint();
"abcd1234".isPrint();
"{}[]".isPrint(); !"\\r".isPrint();
```

Assertion Block

- **isPunct**

Whether all the characters of `this` are punctuation marks (see [Table 22.1](#)).

```
"".isPunct();
"!_[]".isPunct();
!"abc".isPunct(); !"a".isPunct();
```

Assertion Block

- **isSpace**

Whether all the characters of `this` are spaces (see [Table 22.1](#)).

```
"".isSpace();
" ".isSpace();
!" a ".isSpace();
```

Assertion Block

- **isUpper**

Whether all the characters of `this` are upper characters (see [Table 22.1](#)).

```
"".isUpper();
"ABC".isUpper();
!"123".isUpper(); !"a".isUpper();
```

Assertion Block

- **isXdigit**

Whether all the characters of `this` are hexadecimal digits (see [Table 22.1](#)).

```
"".isXdigit();
"abcdef".isXdigit();
"0123456789".isXdigit();
!"g".isXdigit(); "123abc".isXdigit();
```

Assertion Block

- **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]";
```

Assertion Block

- **length**

The number of characters in the string. Currently, this is a synonym of `size`.

Assertion Block

```
"foo".length == 3;
"".length == 0;
```

- **replace(*from*, *to*)**

Replace every occurrence of the string `from` in `this` by `to`, and return the result. `this` is not modified.

Assertion Block

```
"Hello, World!".replace("Hello", "Bonjour")
    .replace("World!", "Monde !") ==
"Bonjour, Monde !";
```

- **rfind(*pattern*, *position* = -1)**

Search backward for the `pattern` string in `this` starting from the `position`. To denote the end of the string, use -1 as `position`. When the `pattern` is not found -1 is returned.

Assertion Block

```
"Hello, World!".rfind("o")      == 8;
"Hello, World!".rfind("o", 8)   == 8;
"Hello, World!".rfind("o", 7)   == 4;
"Hello, World!".rfind("o", 3)   == -1;
"Hello, World!".rfind("o", -1) == 8;
```

- **size**

The size of the string.

Assertion Block

```
"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.

Assertion Block

```
"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.

Assertion Block

```
"a,b;c".split([" ", ";"]) == ["a", "b", "c"];
```

By default splitting is performed on white-spaces:

Assertion Block

```
" abc def\thgi\n".split() == ["abc", "def", "ghi"];
```

Splitting on the empty string stands for splitting between each character:

Assertion Block

```
"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:

Assertion Block

```
"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:

Assertion Block

```
"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"];
```

Assertion Block

- **toAscii**

Convert the first character of `this` to its integer value in the ASCII coding. See also [fromAscii](#).

```
"a".toAscii() == 97;
"b".toAscii() == 98;
"\xff".toAscii() == 0xff;
>Hello, World!\n"
.asList()
.map(function (var v) { v.toAscii() })
.map(function (var v) { String.fromCharCode(v) })
.join()
== "Hello, World!\n";
```

Assertion Block

- **toLowerCase**

A String which is `this` with upper case letters converted to lower case. See also [toLower](#).

```
var hello = "Hello, World!";
hello.toLowerCase() == "hello, world!";
hello == "Hello, World!";
```

Assertion Block

- **toUpperCase**

A String which is `this` with lower case letters converted to upper case. See also [toLower](#).

Assertion Block

```
var hello = "Hello, World!";
hello.toUpperCase() == "HELLO, WORLD!";
hello == "Hello, World!";
```

22.66 Subscription

Connection between InputPorts, see [Section 26.7](#).

22.66.1 Prototypes

- [Object](#)

22.66.2 Construction

22.66.3 Slots

- **asynchronous**

If true, notifies on the target InputPort will trigger asynchronously. The system will also prevent two instances from running in parallel by dropping updates until the callback functions terminate.

- **callCount**

Number of times the link was reset.

- **disconnect**

Disconnect the link.

- **enabled**

Set to false to disable the link.

- **fireRate**

Rate in Hertz at which the link triggers.

- **getAll**

The list of all the connections in the system.

- **maxCallTime**

Maximum call time.

- **meanCallTime**

Average time taken by the callback function on the target InputPort.

- **minCallTime**

Minimum call time.

- **minInterval**

Minimal interval in seconds at which the link can activate. Changes of the source at a higher rate will be ignored.

- **reconnect(*src*)**

Reconnect the link by changing the source to *src*.

- **resetStats**

Reset all statistics.

22.67 System

Details on the architecture the Urbi server runs on.

22.67.1 Prototypes

- `Object`

22.67.2 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.

- `arguments`

The list of the command line arguments passed to the user script. This is especially useful in scripts.

Shell Session

```
$ 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 21.9](#).

urbiscript Session

```
'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`.

urbiscript Session

```
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 21.9](#).

- `currentRunner`

An obsolete alias for `Job.current`.

- `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.

urbiscript Session

```
{
    var first = cycle ; var second = cycle ;
    assert(first + 1 == second);
    first = cycle | second = cycle ;
    assert(first == second);
};
```

- **env**

A [Dictionary](#) containing the current environment of Urbi. See also [env.init](#).

Assertion Block

```
(env["MyVar"] == 12) == "12";
env["MyVar"] == "12";

// A child process that uses the environment variable.
System.system("exit $MyVar") >> 8 ==
    {if (Platform.isWindows) 0 else 12};
(env["MyVar"] == 23) == "23";
System.system("exit $MyVar") >> 8 ==
    {if (Platform.isWindows) 0 else 23};

// Defining to empty is defining, unless you are on Windows.
(env["MyVar"] == "") == "";
env["MyVar"].isNil == Platform.isWindows;

env["UndefinedEnvironmentVariable"].isNil;
!env["PATH"].isNil;

(env["MyVar"] == 12) == "12";
!env["MyVar"].isNil;
env.erase("MyVar") == "12";
env["MyVar"].isNil;
```

- [env.init](#)

Refresh the Urbi environment by fetching all the environment variables. Beware that [env](#) is not updated when calling [getenv](#), [setenv](#) or [unsetenv](#) from the C library. Initialize it first and then manipulate your environment as a simple [Dictionary](#).

Assertion Block

```
env.init() == env;
!env["USER"].isNil;
```

- [eval\(source, target = this\)](#)

Evaluate the urbiscript *source*, and return its result. See also [loadFile](#). The *source* must be complete, yet the terminator (e.g., ‘;’) is not required.

Assertion Block

```
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](#)) or *target* if specified. In particular, to create local variables, create scopes.

```
// Create a slot in the current object.
eval("var a = 23;") == 23;
this.a == 23;

eval("var a = 3", Global) == 3;
Global.a == 3;
```

Assertion Block

Exceptions are thrown on error (including syntax errors).

```
// Scanner errors.
eval("#");
[00000004:error] !!! 1.1: syntax error: invalid character: '#'
[00000005:error] !!!      called from: eval

// Syntax errors.
eval("3; 1 * * 2");
[00000002:error] !!! 1.8: syntax error: unexpected *
[00000003:error] !!!      called from: eval

// Exceptions.
eval("1/0");
[00008316:error] !!! 1.1-3: /: division by 0
[00008316:error] !!!      called from: eval
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";
    }
};

});
```

Warnings are reported.

```
eval("new Object");
[00001388:warning] !!! 1.1-10: 'new Obj(x)' is deprecated, use 'Obj.new(x)'
[00001388:warning] !!!      called from: eval
[00001388] Object_0x1001b2320
```

urbiscript
Session

Nested calls to `eval` behave as expected. The locations in the inner calls refer to the position inside the evaluated string.

urbiscript Session

- `getenv(name)`

Deprecated function use `env[name]` instead. The value of the environment variable `name` as a [String](#) if set, [nil](#) otherwise. See also [env](#), [setenv](#) and [unsetenv](#).

```
getenv("UndefinedEnvironmentVariable").isNil;  
[01234567:warning] !!! 'System.getenv(that)' is deprecated, use 'System.env[that]'  
!getenv("PATH").isNil;  
[01234567:warning] !!! 'System.getenv(that)' is deprecated, use 'System.env[that]'
```

Assertion Block

- `getLocale(category)`

A `String` denoting the locale set for `category`, or raise an error. See `setLocale` for more details.

urbiscript
Session

```
getLocale("LC_IMAGINARY");
[00006328:error] !!! getLocale: invalid category: LC_IMAGINARY
```

- `load(file, target = this)`

Look for `file` in the Urbi path (Section 20.1), and load it in the context of `target`. See also `loadFile`. Throw a `Exception.FileNotFound` error if the file cannot be found. Return the last value of the file.

urbiscript
Session

```
// Create the file ``123.u'' that contains exactly ``var t = 123;''.
File.save("123.u", "var this.t = 123;");
assert
{
    load("123.u") == 123;
    this.t == 123;

    load("123.u", Global) == 123;
    Global.t == 123;
};
```

- `loadFile(file, target = this)`

Load the urbiscript file `file` in the context of `target`. Behaves like `eval` applied to the content of `file`. Throw a `Exception.FileNotFound` error if the file cannot be found. Return the last value of the file.

urbiscript
Session

```
// Create the file ``123.u'' that contains exactly ``var y = 123;''.
File.save("123.u", "var y = 123;");
assert
{
    loadFile("123.u") == 123;
    this.y == 123;

    loadFile("123.u", Global) == 123;
    Global.y == 123;
};
```

- `loadLibrary(library)`

Load the library `library`, to be found in `UObject.searchPath`. The `library` may be a `String` or a `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 20.1). If the file is found announce on `Channel` that `file` is about to be loaded, and load it.

urbiscript
Session

```
// Create the file "123.u" that contains exactly "123;".
File.save("123.u", "123;");
assert
{
    maybeLoad("123.u") == 123;
    maybeLoad("u.123").isVoid;
};
```

- **ndebug**

If true, do not evaluate the assertions. See [Section 21.9](#).

```
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!
```

urbiscript
Session

- **PackageInfo**

See [System.PackageInfo](#).

- **period**

The *period* of the Urbi kernel. Influences the trajectories ([TrajectoryGenerator](#)), and the **UObject** monitoring. Defaults to 20ms.

```
System.period == 20ms;
```

Assertion
Block

- **Platform**

See [System.Platform](#).

- **profile(*function*)**

Compute some measures during the execution of *function* and return the results as a **Profile** object. A **Profile** details information about time, function calls and scheduling.

- **programName**

The path under which the Urbi process was called. This is typically ‘.../urbi’ ([Section 20.3](#)) or ‘.../urbi-launch’ ([Section 20.5](#)).

```
Path.new(System.programName).basename
in ["urbi", "urbi.exe", "urbi-launch", "urbi-launch.exe"];
```

Assertion
Block

- **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;
```

urbiscript
Session

```
// 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
```

- **requireFile(*file*, *target*)**

Load *file* in the context of *target* if it was not loaded before (with `load` or `requireFile`). Unlike `load`, `requireFile` always returns `void`. If *file* is being loaded concurrently `requireFile` waits until the loading is finished.

urbiscript
Session

```
// Create the file "test.u" that echoes a string.
File.save("test1.u", "echo(\"test 1\"); 1;");
requireFile("test1.u");
[00000001] *** test 1
requireFile("test1.u");
// File is not re-loaded

File.save("test2.u", "echo(\"test 2\"); 2;");
load("test2.u");
[00000004] *** test 2
[00000004] 2
requireFile("test2.u");
load("test2.u");
[00000006] *** test 2
[00000006] 2
```

The *target* is not taken into account to check whether the file has already been loaded: if you require twice the same file with two different targets, it will be loaded only for the first.

urbiscript
Session

```
requireFile("test2.u", Global);
```

- **resetStats**

Reinitialize the `stats` computation.

Assertion
Block

```
0 < System.stats()["cycles"];
System.resetStats().isVoid;
1 == System.stats()["cycles"];
```

- **scopeTag**

Bounce to `Tag.scope`.

- **searchFile(*file*)**

Look for *file* in the `searchPath` and return its `Path`. Throw a `Exception.FileNotFound` error if the file cannot be found.

```
// Create the file "123.u" that contains exactly "123;".
File.save("123.u", "123;");
assert
{
    searchFile("123.u") == Path.cwd / Path.new("123.u");
};
```

urbiscript
Session

- **searchPath**

The Urbi path (i.e., the directories where the urbiscript files are looked for, see [Section 20.1](#)) as a `List of Paths`.

Assertion
Block

```
System.searchPath.isA(List);
System.searchPath[0].isA(Path);
```

- `setenv(name, value)`

Deprecated, use `env[name] = value` instead. Set the environment variable `name` to `value.asString`, and return this value. See also [env](#), [getenv](#) and [unsetenv](#).

Windows Issues

Under Windows, setting to an empty value is equivalent to making undefined.

```
setenv("MyVar", 12) == "12";
[00000001:warning] !!! 'System.setenv(var, value)' is deprecated, \
use 'System.env[var] = value'
env["MyVar"] == "12";
```

Assertion Block

- `setLocale(category, locale = "")`

Change the system's `locale` for the `category` to `locale` and return void. If `locale` is empty, then use the locale specified by the user's environment (e.g., the environment variables).

The `category` can be:

`LC_ALL` Overrides all the following categories.

`LC_COLLATE` Controls how string sorting is performed.

`LC_CTYPE` Change what characters are considered as letters and so on.

`LC_MESSAGES` The catalog of translated messages. This category is not supported by Microsoft Windows.

`LC_MONETARY` How to format monetary values.

`LC_NUMERIC` Set a locale for formatting numbers.

`LC_TIME` Set a locale for formatting dates and times.

With `urbi` is run, it does *not* change its locales: it defaults to the “good old C mode”, which corresponds to the ‘C’ (or ‘POSIX’) locale. See also [getLocale](#).

```
// Initially they are all set to "C".
getLocale("LC_ALL") == "C";
getLocale("LC_CTYPE") == "C";
getLocale("LC_NUMERIC") == "C";

// Windows does not understand the "fr_FR" locale, it supports "French"
// which actually denotes "French_France.1252".
var fr_FR =
  { if (System.Platform.isWindows) "French_France.1252" else "fr_FR.utf8" };
// Changing one via the environment does not affect the others.
(env["LC_CTYPE"] = fr_FR) == fr_FR;
getLocale("LC_CTYPE") == "C";
setLocale("LC_CTYPE").isVoid;
getLocale("LC_CTYPE") == fr_FR;
getLocale("LC_NUMERIC") == "C";

// Changing one via setLocale does not change the others either.
setLocale("LC_CTYPE", "C").isVoid;
getLocale("LC_CTYPE") == "C";
getLocale("LC_NUMERIC") == "C";

// The environment variable LC_ALL overrides all the others.
env["LC_ALL"] = fr_FR;
setLocale("LC_ALL").isVoid;
```

Assertion Block

```

getLocale("LC_ALL") == fr_FR;
getLocale("LC_CTYPE") == fr_FR;
getLocale("LC_NUMERIC") == fr_FR;

// Explicit changes of LC_ALL overrides all the others.
setLocale("LC_ALL", "C").isVoid;
getLocale("LC_ALL") == "C";
getLocale("LC_CTYPE") == "C";
getLocale("LC_NUMERIC") == "C";

```

On invalid requests, raise an error.

```

setLocale("LC_IMAGINARY");
[00006328:error] !!! setLocale: invalid category: LC_IMAGINARY

env["LC_ALL"] = "elfic";
setLocale("LC_ALL");
[00024950:error] !!! setLocale: cannot set locale LC_ALL to elfic

setLocale("LC_ALL", "klingon");
[00074958:error] !!! setLocale: cannot set locale LC_ALL to klingon

```

urbiscript
Session

- **shiftedTime**

The number of seconds elapsed since the Urbi server was launched. Contrary to `time`, time spent in frozen code is not counted.

Assertion Block

```

{ 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(*exit_status* = 0)**

Have the Urbi server shut down, with exit status `exit_status`. All the connections are closed, the resources are released. Architecture dependent.

- **sleep(*duration* = inf)**

Suspend the execution for `duration` seconds. No CPU cycle is wasted during this wait. If no `duration` is given the execution is suspended indefinitely.

Assertion Block

```
(time - {sleep(1s); time}).round() == -1;
```

- **spawn(*function*, *clear*)**

Deprecated internal function. Bounces to `function.spawn(clear)`, see [Code.spawn](#).

```

System.spawn(closure () { echo(1) }, true).isA(Job);
[00016657:warning] !!! 'System.spawn' is deprecated, use 'Code.spawn'
[00016659] *** 1

```

Assertion Block

- **stats**

A [Dictionary](#) containing information about the execution cycles of Urbi. This is an internal feature made for developers, it might be changed without notice. See also `resetStats`. These statistics make no sense in ‘`--fast`’ mode ([Section 20.3.1](#)).

Assertion Block

```
var stats = System.stats();

stats.isA(Dictionary);
stats.keys.sort() == ["cycles",
    "cyclesMin", "cyclesMean", "cyclesMax",
    "cyclesVariance", "cyclesStdDev"].sort();

// Number of cycles.
0 < stats["cycles"];
// Cycles duration.
0 <= stats["cyclesMin"] <= stats["cyclesMean"] <= stats["cyclesMax"];

stats["cyclesVariance"].isA(Float);
stats["cyclesStdDev"].isA(Float);
```

- **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 };
```

Assertion Block

- **time**

The number of seconds elapsed since the Urbi server was launched. See also [Date](#). In presence of a frozen [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();
};
```

Assertion Block

- **timeReference**

The “origin of time” of this run of Urbi, as a [Date](#). It is a constant during the run. Basically, `System.time` is about `Date.now - System.timeReference`. See also [time](#) and [Date.now](#).

```
var t1 = System.timeReference();
sleep(1s);
var t2 = System.timeReference();
assert
{
    t1 == t2;
    t1.isA(Date);
```

urbiscript Session

```
(Date.now - (System.timeReference + System.time)) < 0.5s;  
};
```

- **unsetenv(*name*)**

Deprecated use `env.erase (name)` instead. Undefine the environment variable *name*, return its previous value. See also `env`, `getenv` and `setenv`.

Assertion
Block

```
(env["MyVar"] = 12) == "12";  
!env["MyVar"].isNil;  
unsetenv("MyVar") == "12";  
[01234567:warning] !!! 'System.unsetenv(var)' is deprecated, use 'System.env.erase(var)'  
env["MyVar"].isNil;
```

- **version**

The version of Urbi SDK. A string composed of two or more numbers separated by periods: `"3.0.0"`.

Assertion
Block

```
System.version in Regexp.new("\d+(\.\d+)+");
```

22.68 System.PackageInfo

Information about Urbi SDK and its components.

22.68.1 Prototypes

- `Object`

22.68.2 Slots

- `bugReport`

The address where to send bug reports.

Assertion Block

```
System.PackageInfo.components["Urbi"].bugReport
  == "kernel-bugs@lists.gostai.com";
```

Assertion Block

- `components`

A Dictionary of the components loaded in the package. The component `"Urbi SDK"` duplicates the component `"Urbi"` for backward compatibility: prefer the latter.

```
System.PackageInfo.components.keys
  == ["Libport", "Urbi", "Urbi SDK"];
System.PackageInfo.components["Urbi SDK"]
  == System.PackageInfo.components["Urbi"];
```

Assertion Block

- `copyrightHolder`

Who owns the copyright of the package.

```
System.PackageInfo.components["Urbi"].copyrightHolder
  == "Gostai S.A.S.>";
```

Assertion Block

- `copyrightYears`

The years that the copyright covers.

```
System.PackageInfo.components["Urbi"].copyrightYears
  == "2004-2012";
```

Assertion Block

- `date`

A string corresponding to the source date (day and time).

Assertion Block

```
System.PackageInfo.components["Urbi"].date.isA(String);
```

Assertion Block

- `day`

The day part of the date string.

```
System.PackageInfo.components["Urbi"].day.isA(String);
```

Assertion Block

- `description`

The complete description string, with slashes.

```
System.PackageInfo.components["Urbi"].description.isA(String);
```

Assertion Block

- `id`

The identification string.

```
System.PackageInfo.components["Urbi"].id.isA(String);
```

- **major**

The major version component (see [version](#)), for instance 2.

Assertion Block

```
System.PackageInfo.components["Urbi"].major.isA(Float);
```

- **minor**

The minor version component (see [version](#)), for instance 7.

Assertion Block

```
System.PackageInfo.components["Urbi"].minor.isA(Float);
```

- **name**

The name of the package as a String.

Assertion Block

```
System.PackageInfo.components["Urbi"].name
== "Urbi";
```

- **patch**

The number of changes since the [version](#).

```
System.PackageInfo.components["Urbi"].patch.isA(Float);
```

Assertion Block

- **revision**

The revision string.

```
System.PackageInfo.components["Urbi"].revision.isA(String);
```

Assertion Block

- **string**

Name and version concatenated, for instance "Urbi 3.0.0".

```
System.PackageInfo.components["Urbi"].string.isA(String);
```

Assertion Block

- **subMinor**

The sub-minor version component (see [version](#)), for instance 4 for Urbi SDK 2.7.4.

```
System.PackageInfo.components["Urbi"].subMinor.isA(Float);
```

Assertion Block

- **tarballVersion**

The complete version string, with dashes.

```
System.PackageInfo.components["Urbi"].tarballVersion.isA(String);
```

Assertion Block

- **tarname**

Name of the tarball.

```
System.PackageInfo.components["Urbi"].tarname
== "urbi-sdk";
```

Assertion Block

- **version**

The version string, such as "2.7" or "2.7.4".

```
do (System.PackageInfo.components["Urbi"])
{
    assert
    {
        version ==
        {
            if (subMinor)
                "%s.%s.%s" % [major, minor, subMinor]
            else
                "%s.%s" % [major, minor]
        }
    }
}
```

urbscript Session

```

        "%s.%s" % [major, minor]
    };
};

}|;

```

- **versionRev**

Version and revision together.

```

do (System.PackageInfo.components["Urbi"])
{
    assert
    {
        versionRev
        == "version %s patch %s revision %s"
            % [version, patch, revision];
    };
}|;

```

urbiscript
Session

- **versionValue**

The version as an integer, for instance 2007004 for 2.7.4.

```

do (System.PackageInfo.components["Urbi"])
{
    assert
    {
        versionValue == (major * 1e6 + minor * 1e3 + subMinor);
    };
}|;

```

urbiscript
Session

22.69 System.Platform

A description of the platform (the computer) the server is running on.

22.69.1 Prototypes

- [Object](#)

22.69.2 Slots

- **host**

The type of system Urbi SDK runs on.

- **hostAlias**

The name of the system Urbi SDK runs on as the person who compiled it decided to name it. Typically empty, it is fragile to depend on it.

```
System.Platform.hostAlias.isA(String);
```

Assertion Block

- **hostName**

The name of the machine, as given by the `uname -n` command. Unix only.

urbiscript
Session

```
if (!System.Platform.isWindows)
    assert(System.Platform.hostName == System.system("uname -n"));
```

Assertion Block

- **hostOs**

The OS type of system Urbi SDK runs on. For instance `darwin9.8.0` or `linux-gnu` or `mingw32`.

- **isWindows**

Whether running under Windows.

Assertion
Block

```
System.Platform.isWindows in [true, false];
```

Assertion Block

- **kind**

Either "POSIX" or "WIN32".

```
System.Platform.kind in ["POSIX", "WIN32"];
```

22.70 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](#).

22.70.1 Examples

22.70.1.1 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();
```

urbiscript
Session

`Tag.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");
```

urbiscript
Session

22.70.1.2 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
```

urbiscript
Session

```
[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.

Assertion Block

```
{
var t = Tag.new();
var res = [];
for (3)
    detach(res << {t: sleep()});
t.block("foo");
res;
}
===[["foo", "foo", "foo"]];
```

22.70.1.3 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.

urbiscript Session

```
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
```

22.70.1.4 Scope tags

Scopes feature a `scopeTag`, i.e., a tag which will be stopped when the execution reaches the end of the current scope. This is handy to implement cleanups, however the scope was exited from.

urbiscript
Session

```
{
  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);
```

22.70.1.5 Enter/leave events

Tags provide two events, `enter` and `leave`, that trigger whenever flow control enters or leaves tagged statements.

urbiscript
Session

```
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 provides 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:

urbiscript
Session

```
{
  function cook()
  {
    turnGasOn();
    // Cooking code ...
}
```

```

        turnGasOff();
    }|
    enterTheKitchen();
    cook();
    leaveTheKitchen();
};
```

This `cook` function is wrong because there are several situations where we could leave the kitchen with gas still turned on. Consider the following cooking code:

```

urbiscript
Session

{
    function cook()
    {
        turnGasOn();

        if (mealReady)
        {
            echo("The meal is already there, nothing to do!");
            // Oops ...
            return;
        };

        for (var i in recipe)
            if (i in kitchen)
                putIngredient(i)
            else
                // Oops ...
                throw Exception("missing ingredient: %s" % i);

        // ...

        turnGasOff();
    }|
};
```

Here, if the meal was already prepared, or if an ingredient is missing, we will leave the `cook` function without executing the `turnGasOff` statement, through the `return` statement or the exception. One correct way to ensure gas is necessarily turned off is:

```

urbiscript
Session

{
    function cook()
    {
        var withGas = Tag.new("withGas");

        at (withGas.enter?)
            turnGasOn();
        // Even if exceptions are thrown or return is called,
        // the gas will be turned off.
        at (withGas.leave?)
            turnGasOff();

        withGas: {
            // Cooking code...
        }
    }|
};
```

If you need your enter/leave functions to be called synchronously and very efficiently, you can as an alternative to enter/leave define the ilocal slots `onEnter` and `onLeave`.

```

urbiscript
Session

{
    function cook()
    {
        var withGas = Tag.new("withGas");
```

```

var withGas.onEnter = turnGasOn; // copy the function
var withGas.onLeave = turnGasOff;
withGas : {
    // Cooking code...
}
};


```

The `onEnter` and `onLeave` must be local slots, inheritance will not work for them.

Alternatively, the `try/finally` construct provides an elegant means to achieve the same result ([Section 21.8.4](#)).

```

{
    function cook()
    {
        try
        {
            turnGasOn();
            // Cooking code...
        }
        finally
        {
            // Even if exceptions are thrown or return is called,
            // the gas will be turned off.
            turnGasOff();
        }
    }
};


```

urbiscript
Session

22.70.1.6 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 22.70.1.5](#)), 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


```

urbiscript
Session

22.70.2 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.


```

urbiscript
Session

```
{
  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.

urbiscript
Session

```
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
```

22.70.3 Prototypes

- [Object](#)

22.70.4 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.

urbiscript
Session

```
// Anonymous tag.
var t1 = Tag.new();
[00000001] Tag<tag_77>

// Named tag.
var t2 = Tag.new("cool name");
[00000001] Tag<cool name>
```

22.70.5 Slots

- [begin](#)

A sub-tag that prints out "tag_name: begin" each time flow control enters the tagged code. See [Section 22.70.1.6](#).

- [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 22.70.1.2](#).

- [blocked](#)

Whether code tagged by `this` is blocked. See [Section 22.70.1.2](#).

- **end**
A sub-tag that prints out "tag_name: end" each time flow control leaves the tagged code. See [Section 22.70.1.6](#).
- **enter**
An event triggered each time the flow control enters the tagged code. See [Section 22.70.1.5](#).
- **freeze**
Suspend code tagged by `this`, already running or forthcoming. Frozen code can be later unfrozen using `unfreeze`. See [Section 22.70.1.3](#).
- **frozen**
Whether the tag is frozen. See [Section 22.70.1.3](#).
- **leave**
An event triggered each time flow control leaves the tagged code. See [Section 22.70.1.5](#).
- **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 22.70.1.4](#).
- **stop(*result* = void)**
Stop any code tagged by `this`. If some `result` was specified, let stopped code return `result` as value. See [Section 22.70.1.1](#).
- **unblock**
Unblock `this`. See [Section 22.70.1.2](#).
- **unfreeze**
Unfreeze code tagged by `this`. See [Section 22.70.1.3](#).

22.71 Timeout

Timeout objects can be used as [Tags](#) to execute some code in limited time. See also the [timeout](#) construct ([Section 21.10.7](#)).

22.71.1 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] !!! new: Timeout_0x133ec0 has timed out.
[00000007:error] !!!     called from: ---- event handler backtrace:
[00000007:error] !!!     called from: new
```

urbiscript
Session

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] !!! new: Timeout_0x133ec0 has timed out.
[00000007:error] !!!     called from: ---- event handler backtrace:
[00000007:error] !!!     called from: new

t: { echo("Open"); sleep(1s); echo("Close"); };
[00000007] *** Open
[00000007:error] !!! new: Timeout_0x133ec0 has timed out.
[00000007:error] !!!     called from: ---- event handler backtrace:
[00000007:error] !!!     called from: new
```

urbiscript
Session

Even if exceptions have been disabled, you can check whether the count-down expired with `timedOut`.

```
t:sleep(500ms);
[00000007:error] !!! new: Timeout_0x133ec0 has timed out.
[00000007:error] !!!     called from: ---- event handler backtrace:
[00000007:error] !!!     called from: new

if (t.timedOut)
    echo("The Timeout expired.");
[00000000] *** The Timeout expired.
```

urbiscript
Session

22.71.2 Prototypes

- [Tag](#)

22.71.3 Construction

At construction, a Timeout takes a duration, and a [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_0x953c1e8
```

urbiscript
Session

22.71.4 Slots

- **asTimeout**

Return `this`.

```
var t = Timeout.new(10);

Timeout.asTimeout() === Timeout;
t.asTimeout() === t;
```

Assertion Block

- **end**

Stop `this`, return `void`. Can be called several times. See `launch`.

```
var t = Timeout.new(10);
!t.running;
t.end().isVoid;
!t.running;
t.launch().isVoid;
t.running;
t.end().isVoid;
t.end().isVoid;
```

Assertion Block

- **launch**

Fire `this`, return `void`. See `end`.

```
var t = Timeout.new(10);
t.launch().isVoid;
```

Assertion Block

- **running**

Whether is currently running.

Assertion Block

```
var t = Timeout.new(10);

!t.running;
t.launch().isVoid;
t.running;
t.end().isVoid;
!t.running;
```

Assertion Block

- **timedOut**

Whether `this` has timed out.

```
var t = Timeout.new(100ms);
t.launch().isVoid;
!t.timedOut;
sleep(200ms).isVoid;
t.timedOut;
```

22.72 Traceable

Objects that have a concept of backtrace.

This object, made to serve as prototype, provides a definition of backtrace which can be filtered based on the desired level of verbosity.

This prototype is not made to be constructed.

22.72.1 Prototypes

- [Object](#)

22.72.2 Slots

- [backtrace](#)

A call stack as a [List of StackFrames](#). Used by [Exception.backtrace](#) and [Job.backtrace](#).

urbiscript
Session

```
try
{
  [1].map(closure (v) { throw Exception.new("Ouch") })
}
catch (var e)
{
  for| (var sf: e.backtrace)
    echo(sf.name)
};
[00000001] *** map
```

- [hideSystemFiles](#)

Remove system files from the backtrace if this value equals `true`. Defaults to `true`.

urbiscript
Session

```
Traceable.hideSystemFiles = false |

try
{
  [1].map(closure (v) { throw Exception.new("Ouch") })
}
catch (var e)
{
  for| (var sf: e.backtrace)
    echo(sf.name)
};
[00000002] *** f
[00000003] *** each|
[00000003] *** []
[00000004] *** map
```

22.73 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.

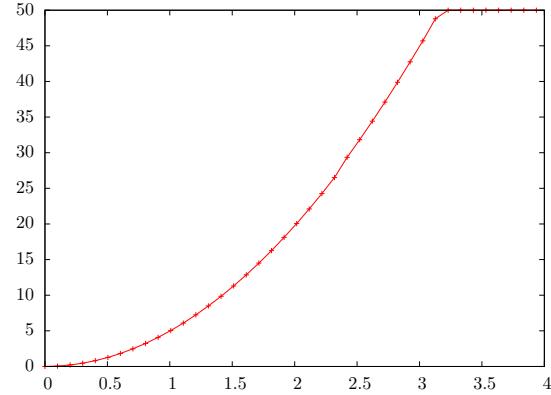
Trajectory generators are not made to be used directly, rather use the “continuous assignment” syntax ([Section 21.12](#)).

22.73.1 Examples

22.73.1.1 Accel

The `Accel` trajectory reaches a target value at a fixed acceleration (`accel` attribute).

```
var y = 0;
y = 50 accel:10,
```

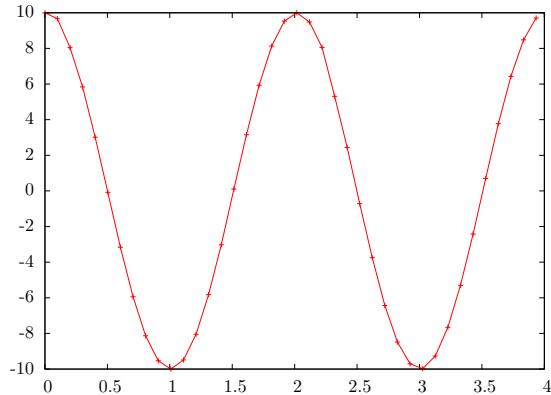


22.73.1.2 Cos

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,
```

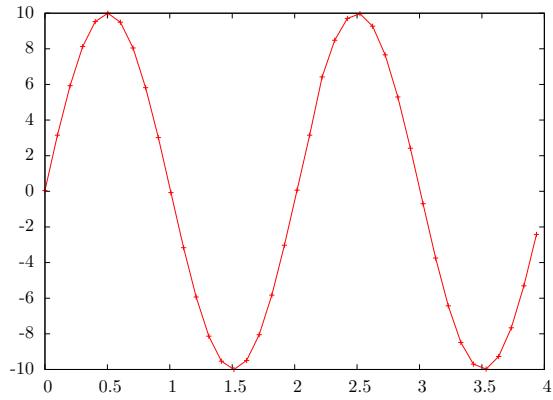


22.73.1.3 Sin

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.

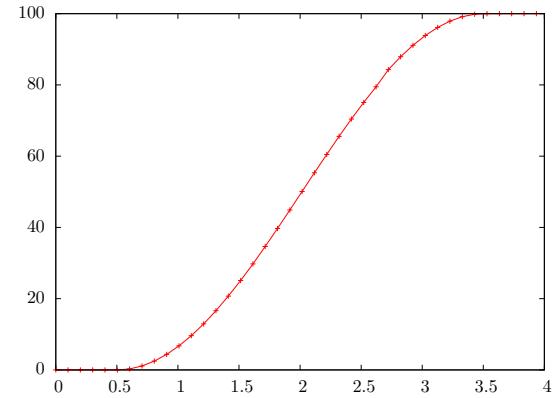
```
var y = 0;
y = 0 sin:2s ampli:10,
```



22.73.1.4 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).

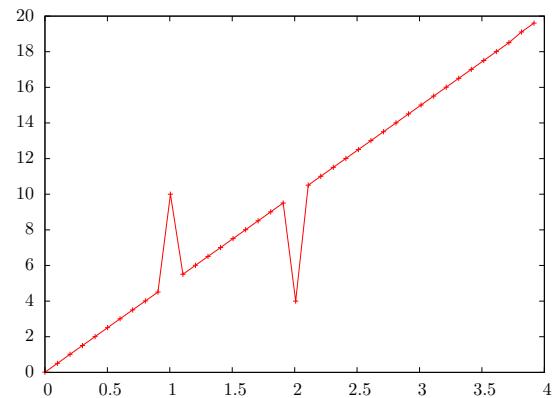
```
var y = 0;
{
    sleep(0.5s);
    y = 100 smooth:3s,
},
```



22.73.1.5 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;
},
```

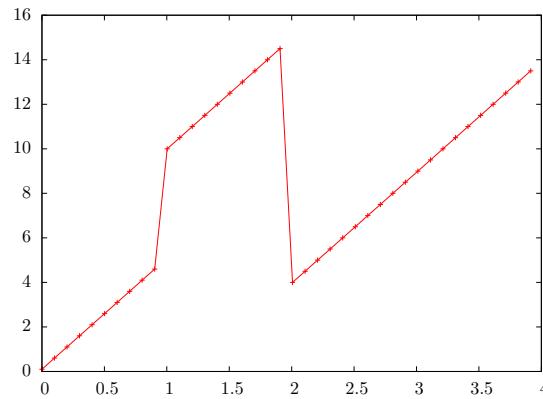


If the **adaptive** attribute is set to true, then the duration of the trajectory is constantly reevaluated.

```
var y = 0;
assign: y = 20 speed: 5 adaptive: true,
```

```
{
    sleep(1s);
    y = 10;
```

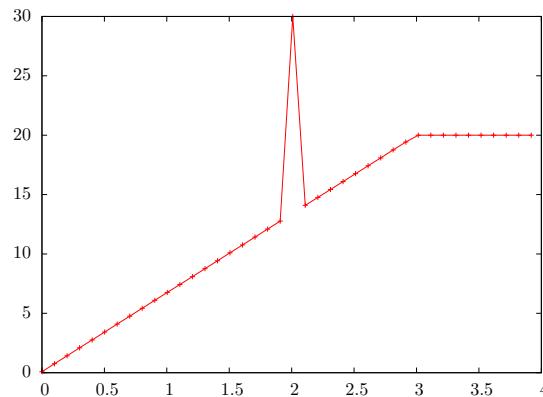
```
sleep(1s);
y = 4;
},
```



22.73.1.6 Time

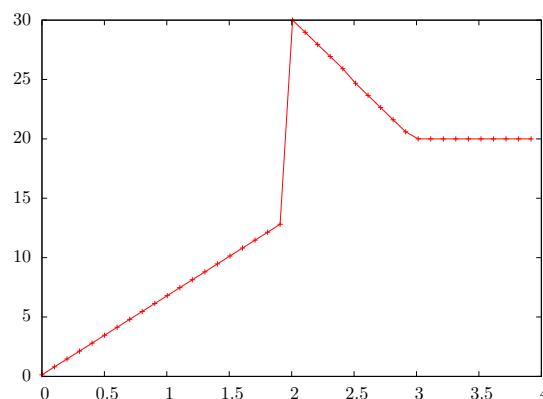
The `Time` trajectory changes the value of the variable from its current value to the target value within a given duration (the `time` attribute).

```
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;
},
```



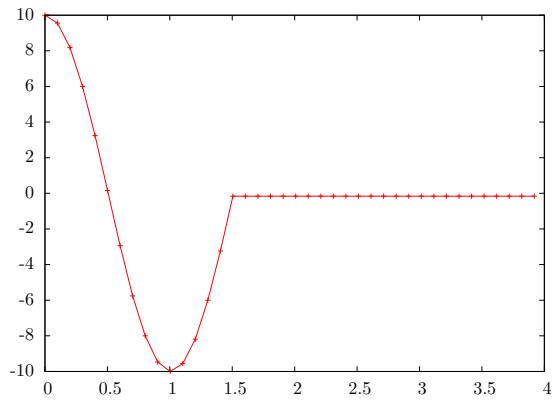
22.73.1.7 Trajectories and Tags

Trajectories can be managed using `Tags`. Stopping or blocking a tag that manages a trajectory kill the trajectory.

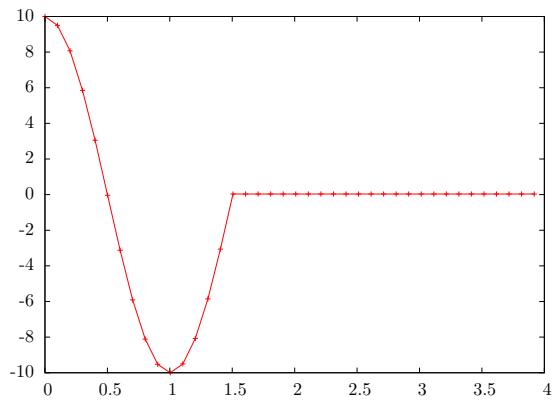
```
var y = 0;
assign: y = 0 cos:2s ampli:10,
```

```
{
  sleep(1.5s);
  assign.stop;
```

```
},
```

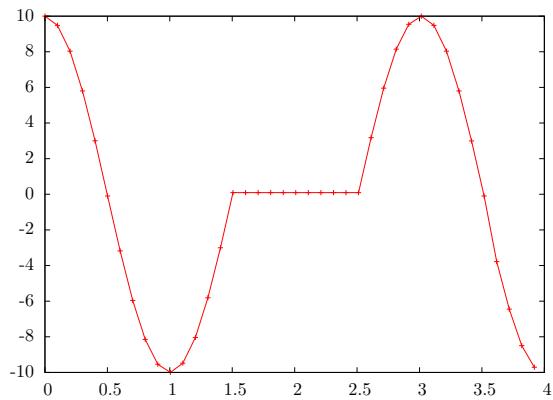


```
var y = 0;
assign: y = 0 cos:2s ampli:10,
{
  sleep(1.5s);
  assign.block;
  sleep(1s);
  assign.unblock;
},
```



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 had it been not frozen.

```
var y = 0;
assign: y = 0 cos:2s ampli:10,
{
  sleep(1.5s);
  assign.freeze;
  sleep(1s);
  assign.unfreeze;
},
```



22.73.2 Prototypes

- [Object](#)

22.73.3 Construction

You are not expected to construct trajectory generators by hand, using modifiers is the recommended way to construct trajectories. See [Section 21.12](#) for details about trajectories, and see [Section 22.73.1](#) for an extensive set of examples.

22.73.4 Slots

- [Accel](#)

This class implements the `Accel` trajectory ([Section 22.73.1.1](#)). 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 22.73.1.2](#), [Section 22.73.1.3](#)).

It derives from `OpenLoop`.

- **Smooth**

This class implements the `Smooth` trajectory ([Section 22.73.1.4](#)). It derives from `OpenLoop`.

- **SpeedAdaptive**

This class implements the `Speed` trajectory when the `adaptive` attribute is given ([Section 22.73.1.5](#)). It derives from `ClosedLoop`.

- **Time**

This class implements the non-adaptive `Speed` and `Time` trajectories ([Section 22.73.1.5](#), [Section 22.73.1.6](#)). It derives from `OpenLoop`.

- **TimeAdaptive**

This class implements the `Time` trajectory when the `adaptive` attribute is given ([Section 22.73.1.6](#)). It derives from `ClosedLoop`.

22.74 Triplet

A *triplet* (or *triple*) is a container storing three objects.

22.74.1 Prototypes

- [Tuple](#)

22.74.2 Construction

A Triplet is constructed with three arguments.

```
Triplet.new(1, 2, 3);
[00000001] (1, 2, 3)

Triplet.new(1, 2);
[00000003:error] !!! new: expected 3 arguments, given 2

Triplet.new(1, 2, 3, 4);
[00000003:error] !!! new: expected 3 arguments, given 4
```

urbiscript
Session

22.74.3 Slots

- **first**

Return the first member of the pair.

```
Triplet.new(1, 2, 3).first == 1;
Triplet[0] === Triplet.first;
```

Assertion
Block

- **second**

Return the second member of the triplet.

```
Triplet.new(1, 2, 3).second == 2;
Triplet[1] === Triplet.second;
```

Assertion
Block

- **third**

Return the third member of the triplet.

```
Triplet.new(1, 2, 3).third == 3;
Triplet[2] === Triplet.third;
```

Assertion
Block

22.75 Tuple

A `tuple` is a container storing a fixed number of objects. Examples include `Pair` and `Triplet`.

22.75.1 Prototypes

- `Object`

22.75.2 Construction

The `Tuple` object is not meant to be instantiated, its main purpose is to share code for its descendants, such as `Pair`. Yet it accepts its members as a list.

```
var t = Tuple.new([1, 2, 3]);
[00000000] (1, 2, 3)
```

urbiscript
Session

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);
[00000000] 1

// Tuples
();
[00000000] ()
(1,);
[00000000] (1,)
(1, 2);
[00000000] (1, 2)
(1, 2, 3, 4,);
[00000000] (1, 2, 3, 4)
```

urbiscript
Session

22.75.3 Slots

- `'*' (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");
```

Assertion
Block

- `'+' (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");
```

Assertion
Block

Assertion
Block

- `'<' (other)`

Lexicographic comparison between two tuples.

```
(0, 0) < (0, 1);
(0, 0) < (1, 0);
(0, 1) < (1, 0);

(1, 2, 3) < (2,);
```

Assertion
Block

- `'==' (other)`

Whether `this` and `other` have the same contents (equality-wise).

Assertion Block

```
(1, 2) == (1, 2);
!((1, 1) == (2, 2));
```

- `[] (index)`

Return the `index`-th element. `index` must be in bounds.

Assertion Block

```
(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.

urbscript Session

```
{
    var t = (1, 2, 3);
    assert
    {
        (t[0] = 2) == 2;
        t == (2, 2, 3);
    };
}
```

- `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);"
```

Assertion Block

- `hash`

A `Hash` object corresponding to this tuple value. Equal tuples (in the sense of `'=='`) have equal hashes, see [Object.hash](#).

```
(().hash().isA(Hash);
().hash() == ().hash();
(1, "foo").hash() == (1, "foo").hash();
// Tuple and List are hashed differently.
(0, 1).hash != [0, 1].hash();
// Tuple uses a List to store elements.
(0, 1).members.hash() == [0, 1].hash();
(0, 1).hash() != (1, 0).hash();
```

Assertion Block

- `matchAgainst(handler, pattern)`

Pattern matching on members. See [Pattern](#).

```
{
    // Match a tuple.
    (var first, var second) = (1, 2);
    assert { first == 1; second == 2 };
};
```

urbscript Session

- `size`

Number of members.

```
(().size == 0;
(1,).size == 1;
(1, 2, 3, 4).size == 4;
(1, 2, 3, 4,).size == 4;
```

Assertion Block

22.76 UObject

UObject is used by the [UObject API](#) (see [Part IV](#)) to represent a bound C++ instance.

All the UObject instances are copied under a unique name as slots of [Global.uobjects](#).

22.76.1 Prototypes

- [Object](#)

22.76.2 Slots

- `uobjectName` Unique name assigned to this object. This is also the slot name of [Global.uobjects](#) containing this [UObject](#).

- `searchPath`

The search-path for UObject files (see [Section 20.1](#)) as a [List of Paths](#). See also [System.loadLibrary](#) and [System.loadModule](#).

```
UObject.searchPath.isA(List);  
UObject.searchPath[0].isA(Path);
```

Assertion
Block

22.77 uobjects

This object serves only to store the UObjects that are bound into the system (plug or remote).

22.77.1 Prototypes

- [Object](#)

22.77.2 Slots

- `asuobjects`

Return `this`.

- `clearStats`

Reset counters.

- `connectionStats`

- `enableStats(bool)`

Depending on `bool`, enable or disable the statistics gathering.

- `getStats`

Return a dictionary of all bound C++ functions called, including timer callbacks, along with the average, min, max call durations, and the number of calls.

- `resetConnectionStats`

- `searchPath`

- `setTrace(bool)`

22.78 UValue

The UValue object is used internally by the UObject API and is mostly hidden from the user. Do not depend on it.

22.78.1 Prototypes

- [Object](#)

22.78.2 Slots

- [asPrintable](#)
- [asString](#)
- [asTopLevelPrintable](#)
- [asUValue](#)
- [extract](#)
- [extractAsToplevelPrintable](#)
- [invalidate](#)
- [put](#)
- [transparent](#)

22.79 UValueSerializable

This pseudo-class is made to be derived from. It provides support to exchange data between C++ and urbscript via `uvalueSerialize`. See [Section 26.18.2](#).

22.79.1 Example

For the conversion to/from C++ to work properly, your class must be globally accessible and inherit from `UValueSerializable`.

```
class Point: UValueSerializable
{
    var x = 0;
    var y = 0;

    function init(var xx = 0, var yy = 0)
    {
        x = xx;
        y = yy
    };

    function asString()
    {
        "<%s, %s>" % [x, y]
    };
}
```

urbscript
Session

22.79.2 Prototypes

- `Object`

22.79.3 Slots

- `uvalueSerialize`

Return a `String` that corresponds to a serialization of `this`.

```
123.uvalueSerialize() == 123;
"ab".uvalueSerialize() == "ab";
[1, 2].uvalueSerialize() == [1, 2];
["b" => 2, "a" => 1].uvalueSerialize() == ["a" => 1, "b" => 2];

Point.uvalueSerialize()
    == ["$sn" => "Point", "type" => "Point", "x" => 0, "y" => 0];
Point.new(12, 34).uvalueSerialize()
    == ["$sn" => "Point", "x" => 12, "y" => 34];
```

Assertion
Block

22.80 Vector

22.80.1 Prototypes

- `Object`

22.80.2 Construction

The Vector constructor can be either be given a single List (of Floats), or any number of Floats.

Vector can be constructed by using either `Vector.new`, or the literal syntax `< >`. However only a subset of the possible expressions are acceted inside this syntax: boolean operators are not allowed.

urbiscript
Session

```
Vector;
[00000140] <>

Vector.new();
[00000146] <>

Vector.new(1.1);
[00000147] <1.1>

Vector.new(1.1, 2.2, 3.3);
[00000155] <1.1, 2.2, 3.3>

Vector.new("123");
[00000174:error] !!! unexpected "123", expected a Float

Vector.new([]);
[00000187] <>

Vector.new([1, 2, 3, 4, 5]);
[00000189] <1, 2, 3, 4, 5>

Vector.new(["123"]);
[00000193:error] !!! unexpected "123", expected a Float

<1, 1>;
[00000147] <1, 1>

var x = 1|;
<1+1, x+2>;
[00000000] <2, 3>
```

22.80.3 Slots

- `'*' (that)`

A Vector whose members are member-wise products of `this` and `that`. The argument `that` must either be a Vector of same dimension, or a Float.

<pre>Vector * Vector == Vector; Vector.new(1, 2, 3) * Vector.new(5, 6, 7) == Vector.new(5, 12, 21); Vector.new(1, 2, 3) * 3 == Vector.new(3, 6, 9); Vector.new(1, 2) * Vector.new(0); [00000601:error] !!! *: incompatible vector sizes: 2, 1</pre>	Assertion Block
---	--------------------

- `'+' (that)`

A Vector whose members are member-wise sums of `this` and `that`. The argument `that` must either be a Vector of same dimension, or a Float.

Assertion
Block

```
Assertion Block
Vector + Vector == Vector;
<1, 2, 3> + <5, 6, 7> == <6, 8, 10>;
<1, 2, 3> + 3 == <4, 5, 6>;
<1, 2> + <0>;
[00000601:error] !!! +: incompatible vector sizes: 2, 1
```

- `'-'(that)`

A Vector whose members are member-wise subtractions of `this` and `that`. The argument `that` must either be a Vector of same dimension, or a Float.

```
Assertion Block
Vector - Vector == Vector;
<1, 2, 3> - <5, 6, 7> == <-4, -4, -4>;
<1, 2, 3> - 3 == <-2, -1, 0>;
Vector.new(1, 2) - Vector.new(0);
[00000601:error] !!! -: incompatible vector sizes: 2, 1
```

- `'/'(that)`

A Vector whose members are member-wise divisions of `this` and `that`. The argument `that` must either be a Vector of same dimension, or a Float.

```
Assertion Block
Vector / Vector == Vector;
<6, 4, 1> / <3, 2, 1> == <2, 2, 1>;
<9, 6, 3> / 3 == <3, 2, 1>;
<1, 2> / <1>;
[00000601:error] !!! /: incompatible vector sizes: 2, 1
```

- `'<'(that)`

Whether `this` is less than the `that` Vector. This is the lexicographic comparison: `this` is “less than” `that` if, from left to right, one of its member is “less than” the corresponding member of `that`:

```
Assertion Block
<0, 0, 0> < <0, 0, 1>;
!(<0, 0, 1> < <0, 0, 0>);

<0, 1, 2> < <0, 2, 1>;
!(<0, 2, 1> < <0, 1, 2>);

<1, 1, 1> < <2>;
!(<2> < <1, 1, 1>);

!(<0, 1, 2> < <0, 1, 2>);
```

or `that` is a prefix (strict) of `this`:

```
Assertion Block
<> < <0>;
!(<0> < <>);

<0, 1> < <0, 1, 2>;
!(<0, 1, 2> < <0, 1>);

!(<0, 1, 2> < <0, 1, 2>);
```

Since Vector derives from `Orderable`, the other order-based operators are defined.

```
Assertion Block
<> <= <>;
<> <= <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, 1, 2> > <>;
!(<0, 1, 2> > <0, 1, 2>);
<0, 1, 2> > <0, 0, 2>;
```

- `'=='(that)`

Whether `this` and `that` have the same size, and members.

```
<> == <>;
<1, 2, 3> == <1, 2, 3>;
!(<1, 2, 3> == <1, 2>);
!(<1, 2, 3> == <3, 2, 1>);
```

Assertion Block

- `'[]'(index)`

The value stored at `index`.

```
var v = <1, 2, 3>;

v[ 0] == 1; v[ 1] == 2; v[ 2] == 3;
v[-3] == 1; v[-2] == 2; v[-1] == 3;

v[4];
[00000100:error] !!! []: invalid index: 4
```

Assertion Block

- `'[]='(index, value)`

Set the value stored at `index` to `value` and return it.

```
var v = <0, 0, 0>;

(v[ 0] = 1) == 1; (v[ 1] = 2) == 2; (v[ 2] = 3) == 3;
v[ 0] == 1; v[ 1] == 2; v[ 2] == 3;

(v[-3] = 4) == 4; (v[-2] = 5) == 5; (v[-1] = 6) == 6;
v[-3] == 4; v[-2] == 5; v[-1] == 6;

v[4] = 7;
[00000100:error] !!! []=: invalid index: 4
```

Assertion Block

- `asString`

Assertion Block

```
<>.asString() == "<>";
<1, 2>.asString() == "<1, 2>";
<1, 2, 3.4>.asString() == "<1, 2, 3.4>";
```

- `asList`

Return this as a list.

Assertion Block

```
<>.asList() == [];
<1,-1>.asList() == [1, -1];
```

- `asVector`

Return `this`.

Assertion Block

```
var v = Vector.new();
v.asVector() === v;
```

- `combAdd(that)`

A `Matrix res` such that $res[i, j] = \text{this}[i] + that[j]$, for all i in the range of `this`, and j in the range of `that`.

Assertion Block

```
<0, 1, 2>.combAdd(<0, 10, 20>
    == Matrix.new([ 0, 10, 20],
                 [ 1, 11, 21],
                 [ 2, 12, 22]);
```

- `combDiv(that)`

A `Matrix res` such that $res[i, j] = \text{this}[i] / that[j]$, for all i in the range of `this`, and j in the range of `that`.

Assertion Block

```
<10, 20>.combDiv(<2, 5, 10>
    == Matrix.new([ 5, 2, 1],
                  [10, 4, 2]);
```

- `combMul(that)`

A `Matrix res` such that $res[i, j] = \text{this}[i] * that[j]$, for all i in the range of `this`, and j in the range of `that`.

Assertion Block

```
<1, 2, 3, 4>.combMul(<5, 6, 7>
    == Matrix.new([ 5, 6, 7],
                  [10, 12, 14],
                  [15, 18, 21],
                  [20, 24, 28]);
```

- `combSub(that)`

A `Matrix res` such that $res[i, j] = \text{this}[i] - that[j]$, for all i in the range of `this`, and j in the range of `that`.

Assertion Block

```
<-1, 0, +1>.combSub(<-2, 0, +2>
    == Matrix.new([1, -1, -3],
                  [2, 0, -2],
                  [3, 1, -1]);
```

- `distance(that)`

The (Euclidean) distance between `this` and `that`: (Euclidean) norm of `this - that`. They must have equal dimensions.

Assertion Block

```
<>.distance(<>) == 0;
<0, 0>.distance(<3, 4>) == 5;
<3, 4>.distance(<0, 0>) == 5;
```

urbscript Session

```
<0>.distance(<1, 2>);
[00000319:error] !!! distance: incompatible vector sizes: 1, 2

<1, 2>.distance(<0>);
[00000334:error] !!! distance: incompatible vector sizes: 2, 1
```

- `norm`

The (Euclidean) norm of `this`: square root of the sum of the square of the members.

Assertion Block

```
<>.norm == 0;
<0>.norm == 0;
<1>.norm == 1;
<1, 1>.norm == 2.sqrt();
<0, 0, 0, 0>.norm == 0;
<1, 1, 1, 1>.norm == 2;
```

- **resize(*dim*)**

Change the dimensions of `this`, using 0 for possibly new members. Return `this`.

```
// Check that <1, 2> is equal to the Vector composed of values,
// when resized the number of values.
function resized(var values[])
{
    var m = <1, 2>;
    var res = Vector.new(values);
    // Resize returns this...
    m.resize(res.size) === m;
    // ...and does resize.
    m == res;
}

assert
{
    resized();
    resized(1);
    resized(1, 2);
    resized(1, 2, 0);
};
```

urbiscript
Session

- **scalarEQ(*that*)**

A Vector whose members are 0 or 1 depending on the equality of the corresponding members of `this` and `that`.

```
<1, 2, 3>.scalarEQ(<3, 2, 1>) == <0, 1, 0>;
```

Assertion
Block

`this` and `that` must have equal sizes.

```
<1, 2>.scalarEQ(<0>);
[00000601:error] !!! scalarEQ: incompatible vector sizes: 2, 1
```

urbiscript
Session

- **scalarGE(*that*)**

A Vector whose members are 0 or 1 if the corresponding member of `this` is greater than or equal to the one of `that`.

```
<1, 2, 3>.scalarGE(<3, 2, 1>) == <0, 1, 1>;
```

Assertion
Block

`this` and `that` must have equal sizes.

```
<1, 2>.scalarGE(<0>);
[00000601:error] !!! scalarGE: incompatible vector sizes: 2, 1
```

urbiscript
Session

- **scalarGT(*that*)**

A Vector whose members are 0 or 1 if the corresponding member of `this` is greater than to the one of `that`.

```
<1, 2, 3>.scalarGT(<3, 2, 1>) == <0, 0, 1>;
```

Assertion
Block

`this` and `that` must have equal sizes.

```
<1, 2>.scalarGT(<0>);
[00000601:error] !!! scalarGT: incompatible vector sizes: 2, 1
```

urbiscript
Session

- **scalarLE(*that*)**

A Vector whose members are 0 or 1 if the corresponding member of `this` is less than or equal to the one of `that`.

```
<1, 2, 3>.scalarLE(<3, 2, 1>) == <1, 1, 0>;
```

Assertion
Block

`this` and `that` must have equal sizes.

urbscript
Session

```
<1, 2>.scalarLE(<0>);
[00000601:error] !!! scalarLE: incompatible vector sizes: 2, 1
```

- **scalarLT(*that*)**

A Vector whose members are 0 or 1 if the corresponding member of `this` is less than to the one of `that`.

Assertion
Block

```
<1, 2, 3>.scalarLT(<3, 2, 1>) == <1, 0, 0>;
```

`this` and `that` must have equal sizes.

urbscript
Session

```
<1, 2>.scalarLT(<0>);
[00000601:error] !!! scalarLT: incompatible vector sizes: 2, 1
```

- **set(*list*)**

Change the members of `this`, and return `this`.

```
var v = <1, 2, 3>;
v.set([10, 20]) === v;
v == <10, 20>;
```

Assertion
Block

- **size**

The size of the vector.

```
<>.size == 0;
<0>.size == 1;
Vector.new(0, 1, 2, 3) .size == 4;
Vector.new([0, 1, 2, 3]).size == 4;
```

Assertion
Block

- **sum**

The sum of the values stored in `this`.

```
<> .sum() == 0;
<1> .sum() == 1;
<0, 1, 2, 3> .sum() == 6;
```

Assertion
Block

- **trueIndexes**

A Vector whose values are the indexes of `this` whose value is non-null.

```
<>.trueIndexes() == <>;
<1, 0, 1, 0, 0, 1>.trueIndexes() == <0, 2, 5>;
```

Assertion
Block

- **zip(*others*)**

Zip this vector with the given list of other vectors, which can be of different sizes.

```
<0,3>.zip([<1, 4>, <2, 5>]) == <0, 1, 2, 3, 4, 5>
```

Assertion
Block

- **serialize(*wordSize*, *littleEndian*)**

Serialize to binary integral data.

```
<1, 2>.serialize(2, false) == "\x01\x00\x02\x00";
<1, 2>.serialize(2, true) == "\x00\x01\x00\x02";
```

Assertion
Block

- **type**

"`Vector`".

```
Vector.type == "Vector";
<>.type == "Vector";
```

Assertion
Block

22.81 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`, which is a valid object, `void` denotes a value one is not allowed to read.

22.81.1 Prototypes

None.

22.81.2 Construction

`void` is the value returned by constructs that return no value.

```
void.isVoid;
{} .isVoid;
{if (false) 123}.isVoid;
```

Assertion Block

22.81.3 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
```

urbiscript Session

- `isVoid`

Whether `this` is `void`. Therefore, return `true`. Actually there is a temporary exception: `nil.isVoid`.

```
void.isVoid;
void.acceptVoid().isVoid;
! 123.isVoid;

nil.isVoid;
[   Logger    ] nil.isVoid will return false eventually, adjust your code.
[   Logger    ]     For instance replace InputStream loops from
[   Logger    ]         while (!(x = i.get().acceptVoid()).isVoid())
[   Logger    ]             cout << x;
[   Logger    ]             to
[   Logger    ]         while (!(x = i.get()).isNil())
[   Logger    ]             cout << x;
```

Assertion Block

- `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
```

urbiscript Session

Chapter 23

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 IV](#)): `Ros`, `Ros.Topic`, and `Ros.Service`.

This module is loaded automatically if `ROS_ROOT` is set in your environment. If `roscore` is not launched, you will be warned and Urbi will check regularly for `roscore`.

If for any reason you need to load this module manually, use:

urbscript
Session

```
loadModule("urbi/ros");
```

23.1 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` and so forth.

23.1.1 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).

23.1.2 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 is a node name. Its associated value is another [Dictionary](#), with the following keys: `publish`, `subscribe`, `services`. Each of these keys is associated with a list of topics or services.

Assertion
Block

```
"/rosout" in Ros.nodes;
Ros.name in Ros.nodes;
```

- Service

The `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.

urbiscript
Session

```
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` class.

- topics

A `Dictionary` containing all the topics advertised to `roscore`. Each key is the name of a topic, and the associated value is another `Dictionary`, with the following keys: `subscribers`, `publishers`, `type`.

urbiscript
Session

```
var topics = Ros.topics|;
topics.keys;
[03316144] ["/rosout_agg"]

// The actual value of the "type" field depends on the ROS version
// (the location changed, but the type is the same):
// - "roslib/Log" for CTurtle.
// - "rosgraph_msgs/Log" for Diamondback and later.
topics["/rosout_agg"];
[03325634] ["publishers" => ["/rosout"], \
"subscribers" => [], \
"type" => "rosgraph_msgs/Log"]
```

23.2 Ros.Topic

This `UObject` provides a handy way to communicate through ROS topics, by subscribing to existent topics or advertising to them.

23.2.1 Construction

To create a new topic, call `Ros.topic.new` with a string (the name of the topic you want to subscribe to / advertise on).

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.

23.2.2 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.

23.2.2.1 Common

- `name`

The name of the topic provided to `init`.

```
Ros.Topic.new("/test").name == "/test";
```

Assertion Block

- `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 a [Dictionary](#).

urbiscript Session

```
var logTopic = Ros.Topic.new("/rosout_agg")|;
logTopic.subscribe|;
assert
{
    logTopic.structure.keys
    == ["file", "function", "header", "level", "line", "msg", "name", "topics"];
};
```

- `subscriberCount`

The number of subscribers for the topic given in `init`; 0 if the topic is not registered.

23.2.2.2 Subscription

- `onMessage`

Event triggered when a new message is received from a subscribed channel, the payload of this event contains the message.

urbiscript Session

```
var t = Ros.Topic.new("/test")|;
at (t.onMessage?(var m))
    echo(m);
t.subscribe;
```

- `subscribe`

Subscribe to the provided topic. Throw an exception if 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 deserializing the received messages.

- `unsubscribe`

Unsubscribe from a previously subscribed channel, and set the state of the instance as if `init` was called but not `subscribe`.

23.2.2.3 Advertising

- '`<<`' (*message*)

An alias for `publish`.

```
var stringPub = Ros.Topic.new("/mytest")|;
stringPub.advertise("std_msgs/String");
stringPub << ["data" => "Hello world!"];
```

urbiscript Session

- **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 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" => ""]
```

urbiscript
Session

- **onConnect(*name*)**

Event triggered when the current instance is advertising on a topic and a node subscribes to this topic. The payload *name* is the name of the node that subscribed. See also `onDisconnect`.

```
var p = Ros.Topic.new("/test/publisher")|;
at sync (p.onConnect?(var n))
  echo("%s has subscribed to %s" % [n, p.name]);
// The structure is not defined, yet.
assert (p.structure.isVoid);
p.advertise("std_msgs/String");
// The structure is ready to be used.
assert (p.structure == ["data" => ""]);

var s = Ros.Topic.new("/test/publisher")|;
```

urbiscript
Session

Subscription and unsubscription are asynchronous. To make them synchronous, we `waituntil` the connection event is sent. The first idea:

```
s.subscribe;
waituntil (p.onConnect?);
```

urbiscript
Session

is wrong: the event (fired in the first line) might be accomplished before the second line is even started. In that case, Urbi will remain suspended, waiting for an already passed event. Rather, we will “`waituntil`” in background *before* subscribing, and rely on the fact that scopes “`wait`” for all the background statements to be finished ([Section 21.1.7.2](#)):

```
{ 
  waituntil (p.onConnect?),
  s.subscribe;
};
```

[00077308] *** /urbi_1317980847216045000 has subscribed to /test/publisher

urbiscript
Session

- **onDisconnect(*name*)**

Event triggered when the current instance is advertising on a topic, and a node unsubscribes from this topic. The payload *name* is the name of the node that unsubscribed. See also `onConnect`. The following example is a continuation of the previous one.

```
at (p.onDisconnect?(var n))
  echo("%s has unsubscribed to %s" % [n, p.name]);

{
  waituntil (p.onDisconnect?),
  s.unsubscribe;
};
```

[00077308] *** /urbi_1317980847216045000 has unsubscribed to /test/publisher

urbiscript
Session

- **publish(*message*)**

Publish *message* to the current topic. This method is usable only if `advertise` was called.

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.

urbscript
Session

```
var stringPub = Ros.Topic.new("/mytest")|;
stringPub.advertise("std_msgs/String");
stringPub.publish(["data" => "Hello world!"]);
```

- `unadvertise`

Tells ROS that we stop publishing on this topic. The `Object` then gets back to a state where it could call `init`, `subscribe` or `advertise`.

23.2.3 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.

urbscript
Session

```
var publisher = Ros.Topic.new("/example")|
at (publisher.onConnect?(var name))
echo(name[0,5] + " is now listening on " + publisher.name);
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). Thanks to the previous `at` above, a message is displayed at subscription time.

urbscript
Session

```
var subscriber = Ros.Topic.new("/example")|
at (subscriber.onMessage?(var m))
echo(m["data"]);
subscriber.subscribe;
// Let the "is now listening" message arrive.
sleep(200ms);
[00026580] *** /urbi is now listening on /example
```

The structure for the messages are, of course, equal between the subscriber and the publisher.

```
subscriber.structure == publisher.structure;
```

Assertion
Block

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;
// Let the "is no longer" message arrive.
sleep(200ms);
[00252566] *** /urbi is no longer listening on /example
```

urbscript
Session

23.3 Ros.Service

This `UObject` provides a handy way to call services provided by other ROS nodes.

23.3.1 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 `service.initialized` to be true to be able to call `request`.

23.3.2 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);
```

urbiscript
Session

- `name`

The name of the current service.

```
logService.name;
[00036689] "/rosout/get_loggers"
```

urbiscript
Session

- `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] [ => ]
```

urbiscript
Session

- `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);
[00034567] *** ["level" => "INFO", "name" => "ros"]
[00034571] *** ["level" => "INFO", "name" => "ros.roscpp"]
[00034575] *** ["level" => "INFO", "name" => "ros.roscpp.roscpp_internal"]
[00034586] *** ["level" => "WARN", "name" => "ros.roscpp.superdebug"]
```

urbiscript
Session

- `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" => []]
```

urbiscript
Session

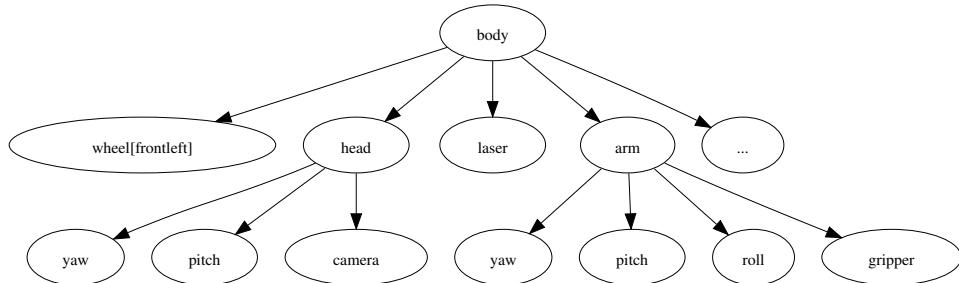
Chapter 24

Urbi Standard Robotics API

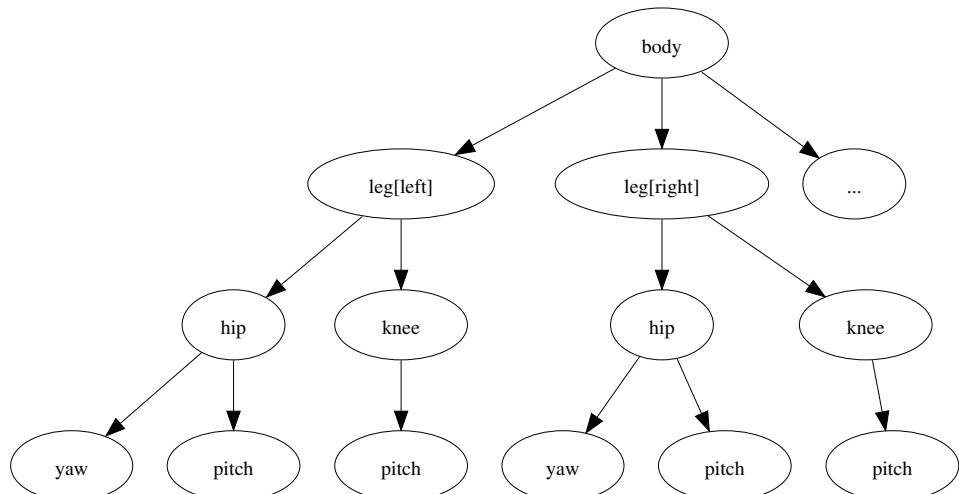
This section aims at clarifying the naming conventions in Urbi Engines for standard hardware/-software devices and components implemented as UObject and the corresponding methods/attributes/events to access them. The list of available hardware types and software component is increasing and this document will be updated accordingly. Please contact the maintainers, should you be working on a component not described or closely related to one described here.

24.1 The Structure Tree

The hardware 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 hardware: 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):

urbiscript
Session

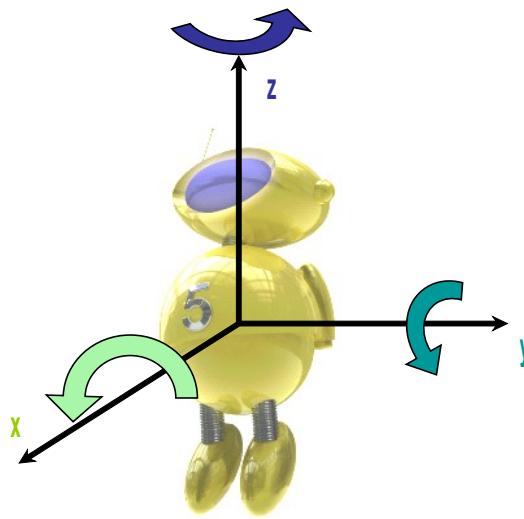
```
body.leg[right].hip.tilt;
body.arm.grip;
body.laser;
// ...
```

The structure tree should not be mistaken for a representation of the kinematics chain. 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.

24.2 Frame of Reference

In many cases, it will be necessary to refer to an absolute frame of reference attached to the hardware. To avoid ambiguities, the standard frame of reference will have the following definition:



Origin the center of mass

X axis oriented towards the front of the machine. 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, 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 machine's 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.

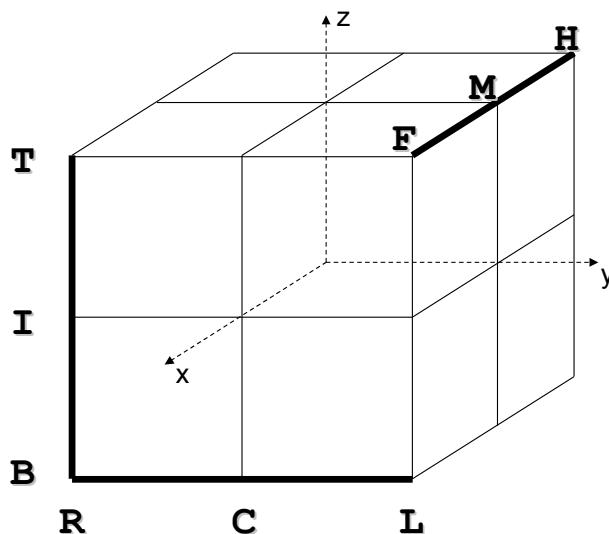
24.3 Component naming

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 hardware configuration. However, this document includes a table covering many different possible cases. We recommend that manufacturers pick from this table the name that fits the most the description of their component.

24.4 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 machine, and *up/pown* matches the Z-axis, as depicted in the figure below:



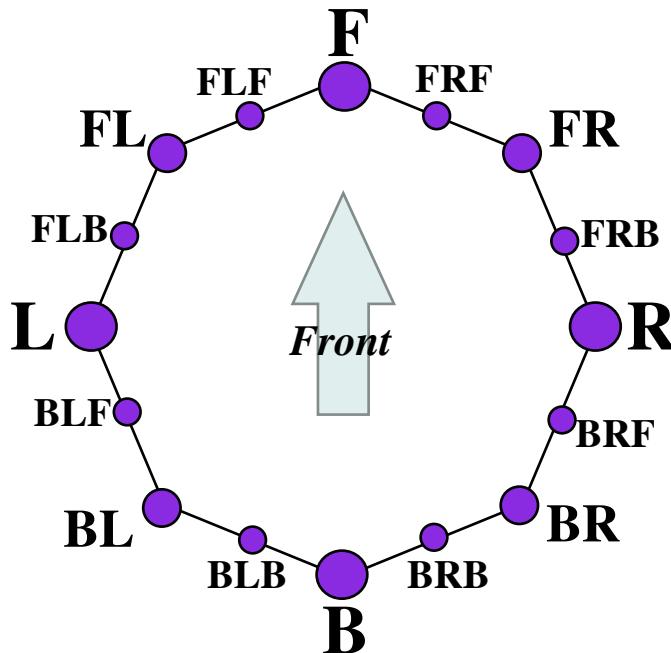
Several geometric qualifiers can be used at the same time to further refine the position. In this case, multiple Localizer nodes are used. As a convention, height information (U/I/D) comes first, followed by depth information (F/M/H), and then side information (R/C/L).

```
// 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];
```

urbiscript
Session

```
// 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.

urbscript
Session

```
robot.body.accel[x]; // accelerometer in the x direction
```

Examples of component names including localization:

urbscript
Session

```
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
```

24.5 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 [Interface.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.

24.5.1 AudioIn

The AudioIn interface groups every information relative to microphones.

- **duration**

Amount of sound in the val attribute, expressed in *ms*.

- **gain**

(Optional.) Microphone gain amplification (expressed between 0 and 1).

- **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.

24.5.2 AudioOut

The AudioOut interface groups every information relative to speakers.

- **playing**

This is a Boolean value which is true when there is a sound currently playing (the buffer is not empty)

- **remain**

The amount of time remaining to play in the speaker sound buffer (expressed in *ms* as a default unit).

- **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.

- **volume**

(Optional.) Volume of the play back, in decibels.

24.5.3 Battery

Power source of any kind.

- **capacity**
Storage capacity in Amp.Hour.
- **current**
Current current consumption in Amp.
- **remain**
Estimation of the remaining energy between 0 and 1.
- **voltage**
Current voltage in Volt.

24.5.4 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:

- **elongation**
(Optional.) Ratio between the main and the second diameter of the blob enveloping ellipse.
- **orientation**
(Optional.) Angle of the main ellipsoid axis of the blob (0 = horizontal), expressed in radians.
- **ratio**
The size of the blob expressed as a normalized image size: 1 = full image, 0 = nothing.
- **threshold**
The minimum value of ratio to decide that the blob is visible.
- **visible**
A Boolean expressing whether there is a blob in the image or not (see [threshold](#)).
- **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

24.5.5 Identity

Information about the hardware.

- **kind**
This describes the category among: humanoid, four-legged, wheeled, industrial arm, etc. It gives a general idea of the hardware family, but does not replace a more systematic probe of available services by investigating the list of attributes of the object.
- **model**
Model of the hardware.
- **name**
Name of the hardware.
- **serial**
Serial number (if available).

24.5.6 Led

Simple uni-color Led.

- **val**
Led intensity between 0 and 1.

24.5.6.1 RGLed

Subclass of [Interface.Led](#). Tri-color led.

- **b**
Intensity of the blue component, between 0 and 1.
- **g**
Intensity of the green component, between 0 and 1.
- **r**
Intensity of the red component, between 0 and 1.

24.5.7 Mobile

Mobile robots all share this generic interface to provide high order level motion control capabilities.

24.5.7.1 Blocking API

The following set of functions will block until associated movement is finished. If a function of this set is called while an other is already running, the first call will be canceled: the movement will be interrupted and the call will return.

Many of the functions below take a position argument. This position is expressed as a subset of the six values x, y, z, rho, theta, phi.

- **go(*x*)**
Move approximately *x* meters forward (along the X axis) if *x* is positive, backward otherwise.
- **goTo(*position*)**
Try to reach the given coordinates in the robot frame of reference.
- **goToAbsolute(*position*)**
Try to reach given coordinates in an absolute frame of reference. This frame of reference is maintained using the robot odometry, or any other localization system available.
- **goToChargingStation**
Try to reach the charging station. If the feature is not available, return immediately.
- **position**
The current robot position in the absolute frame of reference.
- **setAbsolutePosition(*position*)**
Force absolute position to the given value.
- **stop**
Stop current movement.
- **turn(*theta*)**
Turn left (in the direction going from the positive X axis to the positive Y axis) approximately *theta* radians. *theta* can be a positive or negative value.

24.5.7.2 Speed-control API

In addition to the previous functions, one can directly set the target movement speed in all linear/angular directions using the 6 scalar variables `xSpeed`, `ySpeed`, `zSpeed`, `yawSpeed`, `pitchSpeed`, `rollSpeed`, or the variable `speed` which contains a list. Writing to one of those slots will abort any function in the blocking API.

Implementations should provide default reasonable values for speed in the slots `defaultXSpeed`, `defaultYSpeed`, `defaultZSpeed`, `defaultYawSpeed`, `defaultPitchSpeed` and `defaultRollSpeed`.

24.5.7.3 Safety

- **`watchdog`**

Writing any value to `watchdog` will reset the watchdog timer.

- **`watchdogInterval`**

If non-zero, duration after which the robot will stop if no order was received. Once activated, one has to write to `watchdog`, to one of the speed variables or call one of the blocking API functions every `watchdogInterval` at most, or the `stop` function will be called.

24.5.7.4 State

- **`aborted`**

(Optional.) Emitted each time a function in the blocking API is aborted because it was preempted by an other movement.

- **`finished`**

(Optional.) Emitted each time a function in the blocking API finished normally.

- **`moving`**

True if the robot is currently moving for any reason, false otherwise.

- **`unreachable`**

(Optional.) Emitted when a function in the blocking API is aborted because the target cannot be reached.

24.5.8 Motor

This interface is used to describe a generic motor controller.

- **`DGain`**

(Optional.) Controls the D gain of the PID controller.

- **`IGain`**

(Optional.) Controls the I gain of the PID controller.

- **`PGain`**

(Optional.) Controls the P gain of the PID 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`.

24.5.8.1 LinearMotor

Subclass of [Interface.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.

- **force**

Intensity of the measured or estimated force applied on a linear motor.

- **position**

Position of the motor in meters. Pointed to by the **val** slot.

24.5.8.2 LinearSpeedMotor

Subclass of [Interface.Motor](#). Motor similar to [Interface.LinearMotor](#), but controlled by its translation speed.

- **speed**

Translation speed in meters per second. Pointed to by the **val** slot.

24.5.8.3 RotationalMotor

Subclass of [Interface.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.

- **torque**

Intensity of the measured or estimated torque applied on the motor.

- **turn**

Absolute angular position of the motor, expressed in number of turns.

24.5.8.4 RotationalSpeedMotor

Subclass of [Interface.Motor](#). Interface describing a motor similar to [RotationalMotor](#) controlled by its rotation speed.

- **speed**

Rotation speed in radians per second.

24.5.9 Network

Contains information about the network identification of the hardware.

- **IP**

IP address of the hardware.

24.5.10 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 [Interface.Sensor](#).

24.5.10.1 AccelerationSensor

Subclass of [Interface.Sensor](#). This interface is used to describe an accelerometer.

- **acceleration**

Acceleration expressed in m/s^2 . Pointed to by the **val** slot.

24.5.10.2 DistanceSensor

Subclass of [Interface.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.

24.5.10.3 GyroSensor

Subclass of [Interface.Sensor](#). This interface is used to describe an gyrometer.

- **speed**

Rotational speed in rad/s. Pointed to by the **val** slot.

24.5.10.4 Laser

Subclass of [Interface.Sensor](#). Interface for a scanning laser rangefinder, or other similar technologies.

- **angleMax**

End scan angle in radians, relative to the front of the device.

- **angleMin**

Start scan angle in radians, relative to the front of the device.

- **distanceMax**

Maximum measurable distance.

- **distanceMin**

Minimum measurable distance.

- **rate**

(Optional.) Number of scans per second.

- **resolution**

Angular resolution of the scan, in radians.

- **val**

Last scan result. Can be either a list of ufloat, or a binary containing a packed array of doubles.

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.

24.5.10.5 TemperatureSensor

Subclass of [Interface.Sensor](#). This interface is used to describe a temperature sensor.

- **temperature**

Measured temperature in Celsius degrees. Pointed to by the **val** slot.

24.5.10.6 TouchSensor

Subclass of [Interface.Sensor](#). This interface is used to describe a touch pressure sensor (contact, induction, etc.).

- **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.

24.5.11 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.

- **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).

- **lang**

(Optional.) The language used, in international notation (fr, en, it...): ISO 639.

24.5.12 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.

- **age**

(Optional.) Age of the speaker, if applicable.

- **gender**

(Optional.) Gender of the speaker (0:male/1:female).

- **lang**

(Optional.) The language used, in international notation ISO 639 (fr, en, it...).

- **pitch**

(Optional.) Voice pitch. A positive number, with 1 standing the regular pitch.

- **say(*s*)**

Speak the sentence given in parameter *s*.

- **script(*s*)**

(Optional.) Speak the text *s* augmented by script markups to generate Urbi events.

- **speed**

(Optional.) How fast the voice should go. A positive number, with 1 standing for “regular speed”.

- **voice**

(Optional.) Most TTS engines propose several voices, this attribute allows picking one. It's a string identifier specific to the TTS developer.

- **voicexml(*s*)**

(Optional.) Speak the text *s* expressed as a VoiceXML string.

24.5.13 Tracker

Camera-equipped machines 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.

- **pitch**
Rotational articulation around the Y axis, expressed in radians.
- **yaw**
Rotational articulation around the Z axis, expressed in radians.

24.5.14 VideoIn

The VideoIn interface groups every information relative to cameras or any image sensor.

- **exposure**
(Optional.) Exposure duration, expressed in seconds. 0 if non applicable.
- **format**
(Optional.) Format of the image, expressed as an integer in the enum `urbi::UIImageFormat`. See below for more information.
- **gain**
(Optional.) Camera gain amplification (expressed as a coefficient between 0 and infinity). 1 if non applicable.
- **height**
Height of the image in the current resolution, expressed in pixels.
- **quality**
(Optional.) 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).
- **resolution**
(Optional.) 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.
- **val**
Image represented as a [Binary](#) value.
- **wb**
(Optional.) White balance (expressed with an integer value depending on the camera documentation). 0 if non applicable.
- **width**
Width of the image in the current resolution, expressed in pixels
- **xfov**
The x field of view of the camera expressed in radians.
- **yfov**
The y field of view of the camera expressed in radians.

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;
}
```

urbiscript
Session

24.6 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.

24.6.1 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.

24.6.2 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	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	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 ...	
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	

continued on next page

Name	Description	Sub. of	Contains	Facets
finger	A series of articulated motors at the extremity of the arm, and connected to the hand. Fingers are usually localized with arrays and/or lateral localization respective to the hand.	hand	touch	Motor
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 a 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 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 24.6.1 .	body neck knee ankle shoulder elbow wrist torso		Rotational-Motor Rotational-Speed-Motor
pitch	Rotational articulation around the Y axis in the robot. See Section 24.6.1 .	body neck knee ankle shoulder elbow wrist torso		Rotational-Motor Rotational-Speed-Motor

continued on next page

Name	Description	Sub. of	Contains	Facets
roll	Rotational articulation around the X axis in the robot. See Section 24.6.1 .	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	
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 <code>camera</code> to one of the effective cameras (like <code>cameraR</code> or <code>cameraL</code>).	head body		VideoIn

continued on next page

Name	Description	Sub. of	Contains	Facets
<code>speaker</code>	Speaker device. If several speakers are available, localization shall apply; however there must always be an alias from <code>speaker</code> to one of the effective speakers (like <code>speakerR</code> or <code>speakerL</code>).	<code>head body</code>		<code>AudioOut</code>
<code>micro</code>	Microphone devices. If several microphones are available, localization shall apply; however there must always be an alias from <code>micro</code> to one of the effective microphones (like <code>microR</code> or <code>microL</code>).	<code>head body</code>		<code>AudioIn</code>
<code>speech</code>	Speech recognition component.	<code>robot</code>		<code>Speech-Recognizer</code>
<code>voice</code>	Voice synthesis component.	<code>robot</code>		<code>TextTo-Speech</code>

24.7 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
<code>robot.body.armR.elbow</code>	<code>elbowR</code>
<code>robot.body.head.yaw</code>	<code>headYaw</code>
<code>robot.body.legL.knee.pitch</code>	<code>kneeL</code>
<code>robot.body.armR.hand.finger[3] [2]</code>	<code>fingerR[3] [2]</code>
<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.

Some detailed examples:

- `robot.body.armL.hand.fingerR` gives `fingerLR`.
 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`
- `robot.body.head.yaw` gives `headYaw`.
 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`
- `robot.body.legL.knee.pitch` gives `kneeL`.
 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`

24.8 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’.

24.8.1 Interface

The `Interface` class contains urbiscript objects for all the interfaces defined in this document. Implementations must inherit from the correct interface.

urbiscript
Session

```
// 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.

24.8.2 Component

This 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`.
- **`dump`**
Display a hierarchical view of the component hierarchy.
- **`flatDump`**
Display all the devices in the hierarchy, sorted by the `Interface` they implement.
- **`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.

24.8.3 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 mechanism 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.
function robot.go (d)
{
    robot.wheel.val = robot.wheel.val + d / wheelRadius adaptive:1
};
function robot.turn(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
```

urbiscript
Session

Part IV

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 I](#)), or the reference manual ([Chapter 21](#)).

[Chapter 25 — Quick Start](#)

This chapter, self-contained, shows the potential of Urbi used as a middleware.

[Chapter 26 — The UObject API](#)

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

[Chapter 27 — The UObject Java API](#)

UObjects can also be written in Java. This section demonstrates it all.

[Chapter 28 — Use Cases](#)

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

Chapter 25

Quick Start

This chapter presents Urbi SDK with a specific focus on its middleware features. It is self-contained in order to help readers quickly grasp the potential of Urbi used as a middleware. References to other sections of this document are liberally provided to point the reader to the more complete documentation; they should be ignored during the first reading.

25.1 UObject Basics

As a simple running example, consider a (very) basic factory. Raw material delivered to the factory is pushed into some assembly machines, which takes some time.

25.1.1 The Objects to Bind into Urbi

As a first component of this factory, the core machine of the factory is implemented as follows. This class is pure regular C++, it uses no Urbi feature at all.

The header ‘`machine.hh`’, which declares `Machine`, is traditional. The documentation uses the Doxygen syntax.

```
#ifndef MACHINE_MACHINE_HH
#define MACHINE_MACHINE_HH

#include <urbi/uobject.hh>

class Machine
{
public:
    /// Construction.
    /// \param duration how long the assembly process takes.
    ///                 In seconds.
    Machine(float duration);

    /// Lists of strings.
    typedef std::list<std::string> strings;

    /// Assemble the raw components into a product.
    std::string operator()(const strings& components) const;

    /// The duration of the assembly process, in seconds.
    /// Must be positive.
    float duration;
};

#endif // ! MACHINE_MACHINE_HH
```

C++

The implementation file, ‘`machine.cc`’, is equally straightforward.

```
// A wrapper around Boost.Foreach.
```

C++

```
#include <libport/foreach.hh>
#include "machine.hh"

Machine::Machine(float d)
    : duration(d)
{
    assert(0 <= d);
}

std::string
Machine::operator()(const std::vector<std::string>& components) const
{
    // Waiting for duration seconds.
    useconds_t one_second = 1000 * 1000;
    usleep(useconds_t(duration * one_second));

    // Iterate over the list of strings (using Boost.Foreach), and
    // concatenate them in res.
    std::string res;
    foreach (const std::string& s, components)
        res += s;
    return res;
}
```

25.1.2 Wrapping into an UObject

By *binding* a UObject, we mean using the UObject API to declare objects to the Urbi world. These objects have member variables (also known as *attributes*) and/or member functions (also known as *methods*) all of them or some of them being declared to Urbi.

One could modify the Machine class to make it a UObject, yet we recommend wrapping pure C++ classes into a different, wrapping, UObject. It is strongly suggested to aggregate the native C++ objects in the UObject — rather than trying to derive from it. By convention, we prepend a “U” to the name of the base class, hence the UMachine class. This class provides a simplified interface, basically restricted to what will be exposed to Urbi. It must derive from urbi::UObject.

```
C++
#ifndef MACHINE_UMACHINE_HH
#define MACHINE_UMACHINE_HH

// Include the UObject declarations.
#include <urbi/uobject.hh>

// We wrap factories.
#include "machine.hh"

/// A UObject wrapping a machine object.
class UMachine
    : public urbi::UObject
{
public:
    /// C++ constructor.
    /// \param name name given to the instance.
    UMachine(const std::string& name);

    /// Urbi constructor.
    /// \param d the duration of the assembly process.
    /// Must be positive.
    /// \return 0 on success.
    int init(urbi::ufloat d);

    /// Wrapper around Machine::operator().
}
```

```

    std::string assemble(std::list<std::string> components);

    /// Function notified when the duration is changed.
    /// \param v the UVar being modified (i.e., UMachine::duration).
    /// \return 0 on success.
    int duration_set(urbi::UVar& v);

private:
    /// The duration of the assembly process.
    urbi::UVar duration;

    /// The actual machine, wrapped in this UObject.
    Machine* machine;
};

#endif // ! MACHINE_UMACHINE_HH

```

The implementation of `UMachine` is simple. It uses some of the primitives used in the binding process ([Section 26.2](#)):

- `UStart(class)` declares classes that are `UObjects`; eventually such classes will appear in `urbiscript` as `uobjects.class` (but since `Global` derives from `uobjects` you may just use the simpler name `class`). Use it once.
- `UBindFunction(class, function)` declares a *function*. Eventually bound in the `urbiscript` world as `uobjects.class.function`.
- Similarly, `UBindVar(class, variable)` declares a *variable*.

Urbi relies on the prototype model for object-oriented programming, which is somewhat different from the traditional C++ class-based model ([Chapter 5](#)). This is reflected by the presence of *two* different constructors:

- `UMachine::UMachine`, the C++ constructor invoked for every single instance of the `UObject`. It is always invoked by the Urbi system when instantiating a `UObject`, *including* the prototype itself. Its sole argument is its name (an internal detail you need not be aware of). The C++ constructor must register the `UMachine::init` function. It may also bind functions and variables.
- `UMachine::init`, the Urbi constructor invoked each time a new clone of `UMachine` is made, i.e., for every instance except the first one.

Functions and variables that do not make sense for the initial prototype (which might not be fully functional) should be bound here, rather than in the C++ constructor.

The following listing is abundantly commented, and is easy to grasp.

```

#include "umachine.hh"                                     C++

// Register the UMachine UObject in the Urbi world.
UStart(UMachine);

// Bouncing the name to the UObject constructor is mandatory.
UMachine::UMachine(const std::string& name)
    : urbi::UObject(name)
    , machine(0)
{
    // Register the Urbi constructor. This is the only mandatory
    // part of the C++ constructor.
    UBindFunction(UMachine, init);
}

int
UMachine::init(urbi::ufloat d)

```

```

{
    // Failure on invalid arguments.
    if (d < 0)
        return 1;

    // Bind the functions, i.e., declare them to the Urbi world.
    UBindFunction(UMachine, assemble);
    UBindThreadedFunctionRename
        (UMachine, assemble, "threadedAssemble", urbi::LOCK_FUNCTION);
    // Bind the UVars before using them.
    UBindVar(UMachine, duration);

    // Set the duration.
    duration = d;
    // Build the machine.
    machine = new Machine(d);

    // Request that duration_set be invoked each time duration is
    // changed. Declared after the above "duration = d" since we don't
    // want it to be triggered for this first assignment.
    UNotifyChange(duration, &UMachine::duration_set);

    // Success.
    return 0;
}

int
UMachine::duration_set(urbi::UVar& v)
{
    assert(machine);
    urbi::ufloat d = static_cast<urbi::ufloat>(v);
    if (0 <= d)
    {
        // Valid value.
        machine->duration = d;
        return 0;
    }
    else
        // Report failure.
        return 1;
}

std::string
UMachine::assemble(std::list<std::string> components)
{
    assert(machine);

    // Bounce to Machine::operator().
    return (*machine)(components);
}

```

25.1.3 Running Components

As a first benefit from using the Urbi environment, this source code is already runnable! No `main` function is needed, the Urbi system provides one.

25.1.3.1 Compiling

Of course, beforehand, we need to compile this UObject into some loadable module. The Urbi modules are *shared objects*, i.e., libraries that can be loaded on demand (and unloaded) during the execution of the program. Their typical file names depend on the architecture: ‘`machine.so`’

on most Unix platforms (including Mac OS X), and ‘machine.dll’ on Windows. To abstract away from these differences, we will simply use the base name, ‘machine’ with the Urbi tool chain.

There are several options to compile our machine as a UObject. If you are using Microsoft Visual Studio, the Urbi SDK installer created a “UObject” project template accessible through the “New project” wizard. Otherwise, you can use directly your regular compiler tool chain. You may also use `umake-shared` from the ‘`umake-*`’ family of programs ([Section 20.10.2](#)):

```
$ ls machine.uob
machine.cc machine.hh umachine.cc umachine.hh
$ umake-shared machine.uob -o machine
# ... Lots of compilation log messages ...
$ ls
_ubuild-machine.so machine.la machine.so machine.uob
```

Shell Session

The various files are:

‘`machine.uob`’ Merely by convention, the sources of our UObject are in a ‘`*.uob`’ directory.

‘`machine.so`’ A shared dlopen-module. This is the “true” product of the `umake-shared` invocation. Its default name can be quite complex (‘`uobject-i386-apple-darwin9.7.0.so`’ on my machine), as it will encode information about the architecture of your machine; if you don’t need such accuracy, use the option ‘`--output`/‘`-o`’ to specify the output file name.

`umake-shared` traversed ‘`machine.uob`’ to gather and process relevant files (source files, headers, libraries and object files) in order to produce this output file.

‘`_ubuild-machine.so`’ this is a temporary directory in which the compilation takes place. It can be safely removed by hand, or using `umake-deepclean` ([Section 20.10.2](#)).

‘`machine.la`’ a [GNU Libtool](#) file that contains information such as dependencies on other libraries. While this file should be useless most of the time, we recommend against removing it: it may help understand some problems.

25.1.3.2 Running UObject

There are several means to toy with this simple UObject. You can use `urbi-launch` ([Section 20.5](#)) to plug the UMachine in an Urbi server and enter an interactive session.

```
# Launch an Urbi server with UMachine plugged in.
$ urbi-launch --start machine -- --interactive
[00000000] *** Urbi version 3.x.y
var f = UMachine.new(1s);
[00020853] UMachine_0x1899c90
f.assemble(["Hello, ", "World!"]);
[00038705] "Hello, World!"
shutdown;
```

Shell Session

You may also launch the machine UObject in the background, as a network component:

```
$ urbi-launch --start machine --host 0.0.0.0 --port 54000 &
```

Shell Session

and interact with it using your favorite client (`telnet`, `netcat`, `socat`, …), or using the `urbi-send` tool ([Section 20.8](#)).

```
$ urbi-send --port 54000 \
-e 'UMachine.assemble([12, 34]);' \
--quit
[00126148] "1234"
[00000000:client_error] End of file
$ urbi-send --port 54000 \

```

Shell Session

```

-e 'var f = UMachine.new(1s)|' \
-e 'f.assemble(["ab", "cd"]);' \
--quit
[00146159] "abcd"
[00000000:client_error] End of file

```

25.2 Using urbiscript

urbiscript is a programming language primarily designed for robotics. Its syntax is inspired by that of C++: if you know C, C++, Java or C#, writing urbiscript programs is easy. It's a dynamic object-oriented ([Chapter 8](#)) scripting language, which makes it well suited for high-level application. It supports and emphasizes parallel ([Chapter 11](#)) and event-based programming ([Chapter 12](#)), which are very popular paradigms in robotics, by providing core primitives and language constructs.

Thanks to its client/server approach, one can easily interact with a robot, to monitor it, or to experiment live changes in the urbiscript programs.

Courtesy of the UObject architecture, urbiscript is fully integrated with C++. As already seen in the above examples ([Section 25.1.3](#)), you can bind C++ classes in urbiscript seamlessly. urbiscript is also integrated with many other languages such as Java ([Chapter 27](#)), MatLab or Python. The UObject framework also naturally provides urbiscript with support for distributed architectures: objects can run in different processes, possibly on remote computers.

25.2.1 The urbiscript Scripting Language

The following example shows how one can easily interface UObject into the urbiscript language. The following simple class (actually, a genuine object, in urbiscript “classes are objects”, see [Chapter 8](#)) aggregates two assembly machines, a fast one, and a slow one. This class demonstrates usual object-oriented, sequential, features.

```

class TwoMachineFactory
{
    // A shorthand common to all the Two Machine factories.
    var UMachine = uobjects.UMachine;

    // Default machines.
    var fastMachine = UMachine.new(10ms);
    var slowMachine = UMachine.new(100ms);

    // The urbiscript constructor.
    // Build two machines, a fast one, and a slow one.
    function init(fast = 10ms, slow = 100ms)
    {
        // Make sure fast <= slow.
        if (slow < fast)
            [fast, slow] = [slow, fast];
        // Two machines for each instance of TwoMachineFactory.
        fastMachine = UMachine.new(fast);
        slowMachine = UMachine.new(slow);
    };

    // Wrappers to make invocation of the machine simpler.
    function fast(input) { fastMachine.assemble(input) };
    function slow(input) { slowMachine.assemble(input) };

    // Use the slow machine for large jobs.
    function assemble(input)
    {
        var res|
        var machine|

```

urbiscript
Session

```

if (5 < input.size)
{ machine = "slow" | res = slow(input); }
else
{ machine = "fast" | res = fast(input); } |
echo ("Used the %s machine (%s => %s)" % [machine, input, res]) |
res
};

};

[00000001] TwoMachineFactory

```

Using this class is straightforward.

```

var f = TwoMachineFactory.new();
f.assemble([1, 2, 3, 4, 5, 6]);
[00000002] *** Used the slow machine ([1, 2, 3, 4, 5, 6] => 123456)
[00000003] "123456"
f.assemble([1]);
[00000004] *** Used the fast machine ([1] => 1)
[00000005] "1"

```

urbiscript
Session

The genuine power of urbiscript is when concurrency comes into play.

25.2.2 Concurrency

Why should we wait for the slow job to finish if we have a fast machine available? To do so, we must stop requesting a *sequential* composition between both calls. We did that by using the sequential operator, ‘;’. In urbiscript, there exists its concurrent counter-part: ‘,’. Indeed, running *a*, *b* means “launch the program *a* and then launch the program *b*”. The urbiscript Manual contains a whole section devoted to explaining these operators ([Chapter 11](#)).

25.2.2.1 First Attempt

Let's try it:

```

f.assemble([1, 2, 3, 4, 5, 6]),
f.assemble([1]),
[00000002] *** Used the slow machine ([1, 2, 3, 4, 5, 6] => 123456)
[00000004] *** Used the fast machine ([1] => 1)

```

urbiscript
Session

This is a complete failure.

Why?

Since Urbi cannot expect that your code is thread-safe, by default all calls to UObject features are *synchronous*, or *blocking*: *the whole Urbi system is suspended until the function returns*. There is a single thread of execution for Urbi, and when calling a function from a (plugged) UObject, that thread of execution is devoted to evaluated the code.

See for instance below that, in even though `f.assemble` is slow and launched in background, the almost instantaneous display of `ping` is not performed immediately.

urbiscript
Session

```

echo(f.slow([1, 2, 3, 4, 5, 6])),
echo("ping");
[00000002] *** 123456
[00000002] *** ping

```

There are several means to address this unintended behavior. If the base library provides a threaded API (in our example, the `Machine` class, not the `UMachine` UObject wrapper), then you could use it. Yet we don't recommend it, as it takes away from Urbi the possibility to really control concurrency issues (for instance it cannot turn non-blocking functions into blocking functions).

A better option is to ask Urbi to turn blocking function calls into non-blocking ones.

25.2.2.2 Second Attempt: Threaded Functions

If you read carefully the body of the `UMachine::init` function, you will find the following piece of code:

```
UBindFunction(UMachine, assemble);
UBindThreadedFunctionRename
    (UMachine, assemble, "threadedAssemble", urbi::LOCK_FUNCTION);
```

C++

Both calls bind the function `UMachine::assemble`, but the second one will run the call in a separate thread. Since multiple calls to a single function (or different functions of a single object etc.) are likely to fail, a locking model must be provided. Here, `urbi::LOCK_FUNCTION` means that concurrent calls to `UMachine::assemble` must be serialized: one at a time.

To use this threaded version of `assemble`, we can simply patch our `TwoMachineFactory` class:

```
do (TwoMachineFactory)
{
    fast = function (input) { fastMachine.threadedAssemble(input) };
    slow = function (input) { slowMachine.threadedAssemble(input) };
};
```

urbiscript
Session

Let's try again where we failed previously ([Section 25.2.2.1](#)):

```
f.assemble([1, 2, 3, 4, 5, 6]),
f.assemble([1]),
[00000004] *** Used the fast machine ([1] => 1)
[00000002] *** Used the slow machine ([1, 2, 3, 4, 5, 6] => 123456)
sleep(200ms);
```

urbiscript
Session

Victory! The fast machine answered first.

You may have noticed that the result is no longer reported. Indeed, the urbiscript interactive shell only displays the result of synchronous expressions (i.e., those ending with a ';'): asynchronous answers are confusing (see the inversion here).

[Channels](#) are useful to “send” asynchronous answers.

urbiscript
Session

```
var c1 = Channel.new("c1");
var c2 = Channel.new("c2");
c1 << f.assemble([10, 20, 30, 40, 50, 60]),
c2 << f.assemble([100]),
sleep(200ms);
[00000535] *** Used the fast machine ([100] => 100)
[00000535:c2] "100"
[00000625] *** Used the slow machine ([10, 20, 30, 40, 50, 60] => 102030405060)
[00000625:c1] "102030405060"
```

25.3 Conclusion

This section gave only a quick glance to all the power that Urbi provides to support concurrency. Actually, it makes plenty of sense to embed an Urbi engine in a native C++ program, and to delegate the concurrency issues to it. Thanks to its middleware and client/server architecture, it is then possible to connect it to remote components of different kinds, such as using ROS for instance.

Then, a host of features are at hand, ready to be used when you need them: event-driven programming, automatic monitoring of expressions, interactive sessions, ... and, last but not least, the urbiscript programming language.

Chapter 26

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-lever 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 `UIImage`. 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`.

26.1 Compiling UObject

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

The following sections work again on the running example of [Chapter 25](#). We will try to compile a shared library named ‘`machine.so`’ (or ‘`machine.dll`’ on Windows platforms) from a set of four files: ‘`machine.hh`’, ‘`machine.cc`’, ‘`umachine.hh`’, ‘`umachine.cc`’ (see their listings in [Section 25.1](#)). These files are stored in a ‘`machine.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](#).

26.1.1 Compiling with qibuild

`qibuild` is a set of tools on top of cmake to help manage dependencies, among other things. The link above contains instructions to install qibuild and to get you started.

Once qibuild is installed, add `urbi-sdk` to a new toolchain using the `toolchain.xml` file provided in the root directory of `urbi`:

Shell Session

```
$ qitoolchain create urbi-sdk toolchain.xml
```

Then setup a new work directory for your uobject:

Shell Session

```
$ mkdir work
$ cd work
$ qibuild init
```

In this new working directory, create a new project for your uobject:

Shell Session

```
$ qisrc create uobject
$ cd uobject
```

This will create the uobject directory, with a skeleton CMakeLists.txt in it. Modify it to add your uobject, assuming you have one source file named uobject.cc :

```
cmake_minimum_required(VERSION 2.8)
project(uobject)

find_package(qibuild)

find_package(urbi REQUIRED)
# Workaround a long-standing glitch
add_definitions(-D_USE_MATH_DEFINES)
qi_create_lib(uobject MODULE uobject.cc DEPENDS uobject)
```

Configure and build using:

Shell Session

```
$ qibuild configure -c urbi-sdk
$ qibuild make -c urbi-sdk
```

This will produce the file build-urbi-sdk/lib/uobject.dll, or uobject.so depending on your architecture.

You can then load this uobject using urbi-launch([20.5.3](#)).

26.1.2 Compiling by hand

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

Shell Session

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

On Mac OS X the flag ‘-Wl,-undefined,dynamic_lookup’ is needed:

Shell Session

Shell Session

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

26.1.3 The umake-* family of tools

umake can be used to compile UObjects. See [Section 20.10](#) for its documentation.

You can give it a list of files to compile:

Shell Session

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

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

Shell Session

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

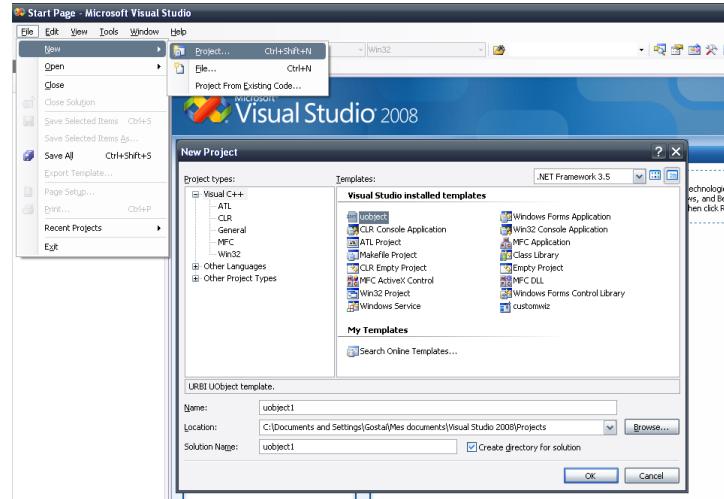
or finally, if you give no argument at all, the sources in the current directory:

Shell Session

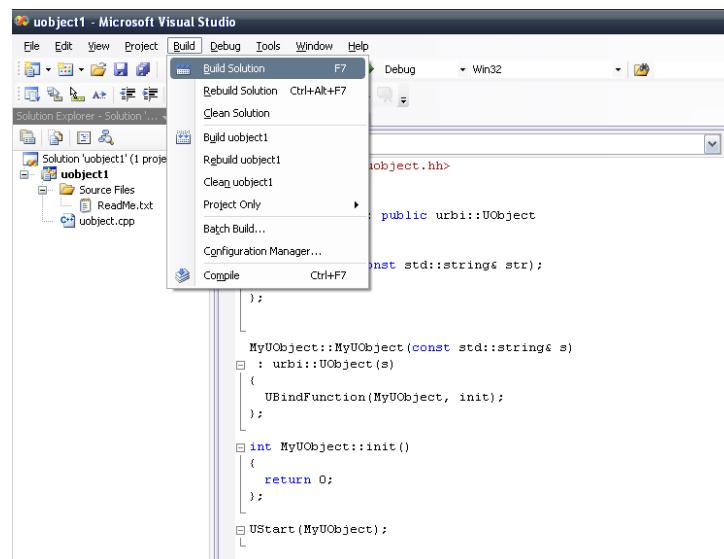
```
$ cd machine.uob
$ umake -q --shared-library -o machine.so
umake: running to build library.
```

26.1.4 Using the Visual C++ Wizard

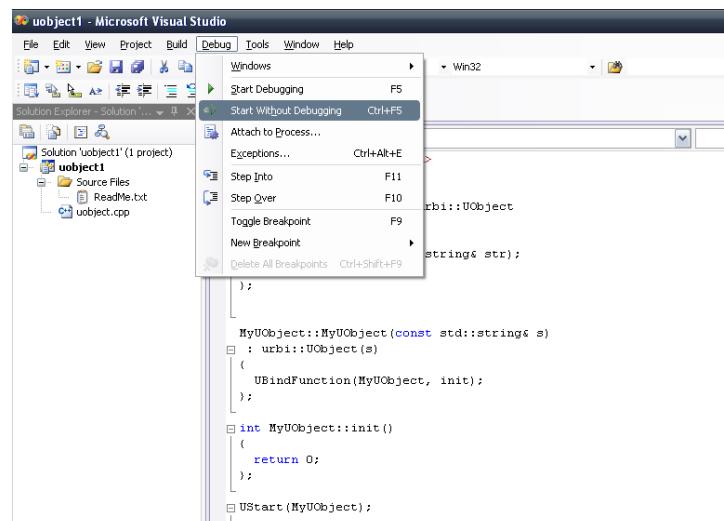
If you installed Urbi SDK using its installer, and if you had Visual C++ installed, then the UObject wizard was installed. Use it to create your UObject code:



Then, compile your UObject.



And run it.



26.2 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:

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

- Then we declare our `adder` class:

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

    // Our variable.
    urbi::UVar v;

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

- The implementation of the constructor.

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

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

- The implementation of our `add` method.

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

- And register this class:

```
C++ // 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.

26.3 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.

26.4 Binding functions

26.4.1 Simple binding

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

`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...`).
- `boost::numeric::ublas::vector<ufloat>` and `boost::numeric::ublas::matrix<ufloat>`
- `UObject*`. Just pass the `UObject` from urbiscript.
- `UVar&`. Pass `object.&uvar` from urbiscript.
- `std::list`, `std::vector` or `boost::unordered_map` of the above types.

The procedure to register new types to this system is explained in [Section 26.18](#).

26.4.2 Multiple bindings

If you have multiple functions to bind, you can use the `UBindFunctions` macro to bind multiple functions at once:

`UBindFunctions(class-name, function1, function2...).`

26.4.3 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`:

`UBindThreadedFunction(class-name, function-name, lock-mode).`

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

The `lock-mode` argument can be used to prevent parallel execution of multiple bound functions if your code is not thread-safe. It can be any of the following values.

- `LOCK_NONE`
No locking is performed.
- `LOCK_FUNCTION`
Parallel execution is limited to one instance of the bound function.

- **LOCK_FUNCTION_DROP**
Same as `LOCK_FUNCTION`, but operations are dropped instead of being queued if one is already running.
- **LOCK_FUNCTION_KEEP_ONE**
Same as `LOCK_FUNCTION`, but the queue is limited to one, and subsequent calls are dropped.
- **LOCK_INSTANCE**
Parallel execution is limited to one bound function for each object instance.
- **LOCK_CLASS**
Parallel execution is limited to one bound function for the class.
- **LOCK_MODULE**
Parallel execution is limited to one bound function for the whole module (shared object).

Other queue sizes can be used by passing `LockSpec(LOCK_FUNCTION, my-queue-size)` as *lock-mode*.

There is a restriction to the locking mechanism: *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.

26.5 Notification of a variable change or access

You can register a function that will be called each time a variable is modified by calling `UNotifyChange`: `UNotifyChange(var, func)`.

- *var* must be either the name of an `UVar`, or an `UVar` itself.
- *func* must be a member function of your `UObject`. This function will be called each time the `UVar` receives a new value.

The function can take 0 or 1 argument. If the argument is of type `UVar&`, then the function will receive the `UVar` that was passed to `UNotifyChange`. If it is of any other type, then the new value in the `UVar` will be converted to this type and passed to the function.

In plugin mode, there is a similar mechanism to create a *getter function* that will be called each time an `UVar` is accessed: the `UNotifyAccess` function. It has the same signature as `UNotifyChange`, and calls the given function each time someone tries to access the `UVar`. The function can update the value in the `UVar` before the access takes place. Usage of `UNotifyAccess` should be reserved to infrequently used `UVar` that take a long time to update, as it disrupts the data flow between `UObject`.

You can remove all notifies associated to any given `UVar` by calling its `unnotify` function.

26.5.1 Threaded notification

In a manner similar to `UBindThreadedFunction`, you can request your callback function to be called in a separate thread by using `UNotifyThreadedChange(var, func, lock-mode)`.

The *lock-mode* argument has the same semantic as for bound functions.

There is one restriction: the callback function must not take a `UVar&` as argument. This restriction is here to ensure that each invocation of your callback will receive the correct value that the source `UVar` had at call time.

26.6 Data-flow based programming: exchanging UVars

The `UNotifyChange` and `UNotifyAccess` features can be used to link multiple `UObjects` together, and perform data-flow based programming: the `UNotifyChange` can be called to monitor UVars from other `UObjects`. 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
{
public:
    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);
```

C++

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

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

urbiscript
Session

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.

26.7 Data-flow based programming: InputPort

Urbi provides a second and more standard way to perform data-flow programming. In this approach, inputs of a component are declared as local `InputPort`, and the binding between this `InputPort` and the output of another component is done in urbiscript using the `>>` operator between two `UVar`:

```
class ObjectTracker: public urbi::UObject
{
    ObjectTracker(const std::string& n)
        : urbi::UObject(n)
    {
        // Bind our constructor.
        UBindFunction(ObjectTracker, init);
        // Bind our input port.
        UBindVar(ObjectTracker, input);
        // NotifyChange on our own input port
        UNotifyChange(input, &ObjectTracker::onImage);
    }
};
```

C++

```

}

// Init is empty.
void init()
{
}

// onImage is unchanged.
void onImage(UVar& src)
{
    UBinary b = src;
    // Processing here.
    val = processing_result;
}

UVar val;

// Declare our input port.
InputPort input;
};

UStart(ObjectTracker);

```

In this model, linking the components is done in urbiscript:

```
var tracker = ObjectTracker.new();
camera.&val >> tracker.&input;
```

urbiscript
Session

26.7.1 Customizing data-flow links

The `>>` operator to establish a data-flow link between two `UVar` returns an object of type `UConnection` that can be used to customize the link.

This object is also present in the slot `changeConnections` of the source `UVar`.

The function `uobjects.connectionStats` displays the statistics of all the connections, and `uobjects.resetConnectionStats` resets all the statistics.

26.8 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.

26.9 The special case of sensor/actuator 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 actuator 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.

26.10 Using Urbi variables

The C++ class `UVar` is used to represent any Urbi slot in C++. To bind the `UVar` to a specific slot, pass its name to the `UVar` constructor, or its `init` method. Once the `UVar` is bound, you can write any compatible type to it, and the new value will be visible in urbiscript. Similarly, you can cast the `UVar` (or use the `as()` method) to convert the current urbiscript value held to any compatible type.

Compatible types are the same as for bound functions (see [Section 26.4.1](#) and [Section 26.18](#)).

```
// Set the camera format to 0 if it is 1.
UVar v;
v.init("camera", "format");
if (v == 1)
    v = 0;
```

C++

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`. By default, all changes to the value will be sent to the remote `UObject`. 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 `UVar::syncValue()`.

```
UVar v("Global", "x");
send("every(100ms) Global.x = time,");
// At this point, v is updated approximately every 100 milliseconds.

v.unnotify();
// At this point v is no longer updated. If v was the only UVar pointing to
// 'Global.x', the value is no longer transmitted.

v.syncValue();
// The previous call will ask for the value of Global.x once, and block until
// the value is written to v.
```

C++

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

26.11 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.

26.12 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.

26.13 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.

26.13.1 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.

By default, reading an `UBinary` from a `UVar`, and writing a `UBinary`, `USound` or `UIImage` to an `UVar` performs a deep-copy of the data. See [Section 26.13.3](#) to avoid this deep-copy in plugin-mode.

Reading a `USound` or `UIImage` from an `UVar` directly will perform a shallow copy from the internal data. The structure content is only guaranteed to be valid until the function returns, and should not be modified.

26.13.2 Binary conversion

To convert between various sound and image formats, two functions are provided in the header ‘`urbi/uconversion.hh`’:

```
C++ void urbi::convert(UIImage& source, UIImage& destination);
void urbi::convert(USound& source, USound& destination);
```

For those functions to work, destination must be filled correctly:

- data and size can be both 0, in which case data will be allocated for you using `malloc`. If data is set but size is too small to fit the value, data will be reallocated using `realloc`.
- all the description fields must be set. It is possible to set any field to 0, in which case the value from `source` will be used.

Consider this example of a sound algorithm requiring 8-bit mono input:

```
C++ class SoundAlgorithm: public UObject
{
public:
    <...>
    void init();
    void onData(UVar& v);
    // We reuse the same USound for converted data to avoid reallocation.
    USound convertedData;
}

void SoundAlgorithm::init(UVar& dataSource)
{
    // initialize convertedData
    convertedData.data = 0;
    convertedData.size = 0; // Let convert allocate for us
    convertedData.soundFormat = SOUND_RAW;
    convertedData.channels = 1;
    convertedData.rate = 0; // Use sample rate of the source
    convertedData.sampleSize = 8;
    convertedData.sampleFormat = SAMPLE_UNSIGNED;
    UNotifyChange(dataSource, &SoundAlgorithm::onData);
}
```

```
void SoundAlgorithm::onData(UVar& v)
{
    USound src = v;
    convert(src, convertedData);
    // Work on convertedData.
}
```

26.13.3 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 block until the `UVar` is updated again, and copy the value at this time.

An example will certainly clarify: Let us first declare an `UObject` that will generate binary data using 0-copy mode.

```
// Declare an UObject producing images in 0-copy optimized mode.
class OptimizedImageSource: public UObject
{
    <...>
public:
    UVar val;
    UBinary imageData;
};

void OptimizedImageSource::init()
{
    // Bind val
    UBindVar(OptimizedImageSource, val);
    // Mark it as bypass mode
    val.setBypass(true);
    // Start a timer.
    USetUpdate(10);
}

int OptimizedImageSource::update()
{
    <Update imageData here>
    // Notify all notifyChange callbacks without copying the data.
    val = imageData;
}
```

C++

Let us then declare another component that will access this binary data without any copy:

```
C++ class BinaryProcessor: public UObject
{
public:
    void init();
    void onData(UVar& v);
    InputPort binaryIn;
};

void BinaryProcessor::init()
{
    UBindVar(BinaryProcessor, binaryIn);
    UNotifyChange(binaryIn, &BinaryProcessor::onData);
}

void BinaryProcessor::onData(UVar& v)
{
    const UBinary& b = v;
```

```
// If in urbiscript you connect the two components using:  
// OptimizedImageSource.&val >> BinaryProcessor.&binaryIn  
// then b will be OptimizedImageSource.binaryData, not a copy.  
}
```

Typing `OptimizedImageSource.val` in urbiscript will wait for the next update from `OptimizedImageSource` and copy the data at this point.

26.14 Direct communication between UObjects

For modularity reasons, all interactions between `UObjects` should go through the various middleware communication mechanisms, mainly `InputPort` and `UNotifyChange`. But it is possible to access directly the C++ instance of an `UObject`:

- by binding a function taking a `UObject*` as argument, and calling it from urbiscript, passing the `UObject` itself.
- by calling the C++ function `getUObject(name)`. `name` must be the canonical `UObject` name, passed to your constructor, and stored in the `_name` member.

26.15 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 26.8](#)). A hub instance can be retrieved by calling `getUObjectHub(string class-name)`. The hub also holds the list of registered `UObject` in its `members` attribute.

26.16 Sending urbiscript code

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

urbiscript
Session

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

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

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

urbiscript
Session

26.17 Using RTP transport in remote mode

By default, Urbi uses TCP connections for all communications between the engine and remote `UObjects`. Urbi also supports the UDP-based *RTP* protocol for more efficient transmission of updated variable values. RTP will provide a lower latency at the cost of possible packet loss, especially in bad wireless network conditions.

26.17.1 Enabling RTP

To enable RTP connections, both the engine and the remote-mode urbi-launch containing your remote `UObject` must load the RTP `UObject`. This can be achieved by passing `urbi/rtp` as an extra argument to both urbi-launch command lines (one for the engine, the other for your remote `UObject`).

Once done, all binary data transfer (like sound and image) in both directions will by default use a RTP connection.

26.17.2 Per-UVar control of RTP mode

You can control whether a specific `UVar` uses RTP mode by calling its `useRTP(bool)` function. Each binary-type `UVar` will have its own RTP connection, and all non-binary `UVar` will share one.

From urbiscript, you can also write to the `rtp` slot of each `UVar`. Existing notifies will be modified to use `rtp` if you set it to true.

26.18 Extending the cast system

26.18.1 Principle

The same cast system is used both for bound function's arguments and return values, and for reading/writing `UVar`.

Should you want to add new type `MyType` to the system you must define two functions:

```
namespace urbi
{
    void operator, (UValue& v, const MyType& t)
    {
        // Here you must fill v with the serialized representation of t.
    }

    template<> struct uvalue_caster<MyType>
    {
        MyType operator()(UValue& v)
        {
            // Here you must return a MyType made with the information in v.
        }
    }
}
```

C++

Once done, you will be able without any other change to

- Take `MyType` as argument to a bound function.
- Return a `MyType` from a bound function.
- Write a `MyType` to an `UVar`.
- Convert an `UVar` to `MyType` using the `UVar::as` function.

26.18.2 Casting simple structures

The system provides facilities to serialize simple structures by value between C++ and urbiscript. This system uses two declarations of each structure, one in C++ and the other in Urbi, and maps between the two.

Here is a complete commented example to map a simple Point structure between urbiscript and C++.

C++

```

struct Point
{
    // Your struct must have a default constructor.
    Point()
        : x(0), y(0)
    {}
    double x, y;
};

// Declare the structure to the cast system. First argument is the struct,
// following arguments are the field names.
URBI_REGISTER_STRUCT(Point, x, y);

```

Declare the urbiscript structure. It must be globally accessible, and inheriting from [UValueSerializable](#).

urbiscript
Session

```

class Global.Point: UValueSerializable
{
    var x = 0;
    var y = 0;

    function init(var xx = 0, var yy = 0)
    {
        x = xx|
        y = yy
    };

    function asString()
    {
        "<%s, %s>" % [x, y]
    };
}
|;
```

Add the class to [Serializables](#) to register it.

```
var Serializables.Point = Global.Point|;
```

urbiscript
Session

Once done, you can call bound functions taking a C++ Point by passing them an urbiscript Point and exchange Point between both worlds through UVar read/write:

```

// This function can be bound using UBindFunction.
Point MyObject::opposite(Point p)
{
    return Point(-p.x, -p.y);
}

// Writing a Point to an UVar is OK.
void MyObject::writePoint(Point p)
{
    UVar v(this, "val");
    v = p;
}

// Converting an UVar to a Point is easy.
ufloat MyObject::xCoord()
{
    UVar v(this, "val");
    Point p;
    // Fill p with content of v.
    v.fill(p);
    // Alternate for the above.
    p = v.as<Point>();
    return v.x;
}
|;
```

C++

Chapter 27

The UObject Java API

The UObject Java API can be used to add new remote objects written in Java to the urbiscript language, and to interact from Java with the objects that are already defined. We cover the use cases of interfacing higher-level components (voice recognition, object detection...) with Urbi using Java.

The Java API defines the UObject class. To each instance of a Java 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 setter member functions for all types known to Urbi: `double`, `java.lang.String`, the binary-holding structures `urbi.UBinary`, `urbi.USound` and `urbi.UIImage`, list types `urbi.UList` and dictionaries `urbi.Dictionary`. This type can also read from and write to the `urbi.UValue` class. The API provides methods to set up callbacks functions that will be notified when a variable is modified or read from urbiscript code. Instance methods of any prototype can be made accessible from urbiscript, providing all the argument types and the return type can be converted to/from `urbi.UValue`.

The UObject Java API has the following limitations:

- it is available only to create remote UObject, i.e., these objects run as separate processes.
- the Java library is generated from the C++ SDK implementation, and rely on compiled C++ code. Thus, remote Java UObject can only run on computers having the full Urbi native SDK installed.

27.1 Compiling and running UObject

UObjects can be compiled easily directly with the `javac` compiler, then you can create JAR archives using the `jar` tool.

In the following sections, we will try to create an uobject jar archive named ‘`machine.jar`’ from a set of two files (‘`Machine.java`’, ‘`UMachine.java`’).

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

27.1.1 Compiling and running by hand

To compile your UObject you need to include in the classpath ‘`liburbijava.jar`’:

Shell
Session

```
$ javac -cp urbi-root/share/sdk-remote/java/lib/liburbijava.jar:. \
  Machine.java UMachine.java
$ jar -cvf machine.jar UMachine.class Machine.class
added manifest
adding: UMachine.class
adding: Machine.class
```

Then to run your uobject, you need to call `java`. We provide a main class called `urbi.UMain` in the ‘`liburbijava.jar`’ archive. You can use this class to start your UObjects. This class takes the names of your uobjects jar files as argument. You also need to specify the lib directory of the Urbi SDK into `java.library.path`:

```
Shell Session
$ java -Djava.library.path=urbi-root/lib \
    -cp urbi-root/share/sdk-remote/java/lib/liburbijava.jar \
    urbi.UMain ./machine.jar
urbi-launch: obeying to URBI_ROOT = /usr/local/gostai
UObject: Urbi version 3.x.y
UObject: Remote Component Running on 127.0.0.1 54000
Kernel Version: 0
[LibUObject] Registering function UMachine.init 1 into UMachine.init from UMachine
[LibUObject] Pushing UMachine.init in function
```

27.1.2 The `umake-java` and `urbi-launch-java` tools

`umake-java` can be used to compile Java UObject. It will produce a JAR archive that you can use with `urbi-launch-java`.

You can give it a list of files to compile:

```
Shell Session
$ umake-java -q machine.uob/*.java -o machine.jar
```

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

```
$ umake-java -q machine.uob -o machine.jar
```

Shell Session

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

```
Shell Session
$ cd machine.uob
$ umake-java -q -o machine.jar
```

To run your UObject then use `urbi-launch-java` (see [Section 20.6](#)):

```
Shell Session
$ urbi-launch-java machine.jar
urbi-launch: obeying to URBI_ROOT = /usr/local/gostai
UObject: Urbi version 3.x.y
UObject: Remote Component Running on 127.0.0.1 54000
Kernel Version: 0
[LibUObject] Registering function UMachine.init 1 into UMachine.init from UMachine
[LibUObject] Pushing UMachine.init in function
```

27.2 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 you need some imports:

```
Java
import urbi.UObject;
import urbi.UVar;
import urbi.UValue;
```

- Then we declare and implement our `adder` class:

```
Java
public class Adder extends UObject // must extends UObject
{
    // Register the class within urbi
    static { UStart(Adder.class); }

    // Declare a variable v that will be accessible in Urbi
```

```

private UVar v = new UVar ();

/// the class must have a single constructor taking a string
public Adder (String s)
{
    super (s);
    /// Bind the variable v to Urbi
    UBindVar (v, "v");

    /// Initialize our UVar v to some value
    /// (we choose 42 :)
    v.setValue(42);

    /// Bind the function add to Urbi
    UBindFunction ("add");
}

/// Our method.
public double add (double rhs)
{
    /// Return the value of our UVar v (converted to double)
    /// plus the value of the argument of the function.
    return v.doubleValue () + rhs;
}

```

To bind the variables to Urbi, we use the function:

Java

```
void UBindVar (UVar v, String name)
```

This function takes as argument the UVar variables, and the name of the UVar (because Urbi need to know what is the name of your variable). Once your variable is bound with `UBindVar` it will be accessible in Urbi.

- Each UObject needs to be registered within urbi using the code

Java

```
static { UStart(YourUObject.class); }
```

If you run this UObject and test it from Urbi it gives:

urbiscript
Session

```
[00000102] *** Urbi version 3.x.y
Adder;
[00006783] Adder
Adder.v;
[00010871] 42
Adder.add(-26);
[00025795] 16
Adder.add(-2.6);
[00035411] 39.4
```

To summarize:

- Declare your object class as extending UObject.
- Declare a single constructor taking a String, and pass this string to the constructor of UObject.
- Declare the variables you want to share with Urbi with the type urbi.UVar.
- In the constructor, call `UBindVar` for each UVar you want as an instance variable, and `UBindFunction` for each function you want to bind.
- Call the function `UStart` for each UObject class you define.

27.3 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 Java 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.

For example let's add an Urbi constructor to our `Adder` class. We rewrite it as follow:

```
public class Adder extends UObject // must extends UObject
{
    /// Register the class within urbi
    static { UStart(Adder.class); }

    /// Declare a variable v that will be accessible in Urbi
    private UVar v = new UVar();

    /// Constructor
    public Adder (String s)
    {
        super (s);
        UBindFunction ("init");
    }

    /// The init function is the constructor in Urbi. Here it takes
    /// one argument that we use to initialize the 'v' variable.
    /// The init function must return an int of value 0
    /// if all went OK.
    public int init (double v_init)
    {
        /// Bind the variable v to Urbi
        UBindVar (v, "v");

        /// Initialize our UVar v to the value given in the
        /// constructor
        v.setValue(v_init);

        /// Bind the function add to Urbi
        UBindFunction ("add");

        return 0;
    }

    public double add (double rhs)
    {
        /// Return the value of our UVar v (converted to double)
        /// plus the value of the argument of the function.
        return v.doubleValue () + rhs;
    }
}
```

Java

Now `v` and `add` are bound only when instance of the `Adder` object are constructed. We have added an `init` constructor with one argument that we use to initialize the value of `v`. You can run this `UObject` and test it in Urbi to see the difference with the previous example:

```
[00000097] *** Urbi version 3.x.y
Adder;
[00010592] Adder
Adder.v;
[00013094:error] !!! 2.1-7: lookup failed: v
var a = Adder.new(51);
[00041405] object_13
a.v;
[00044742] 51
```

urbscript
Session

```
a.add(10);
[00054783] 61
```

27.4 Binding functions

To bind the functions to Urbi, you can use:

```
void UBindFunction (Object obj, String method_name, String[] parameters_name)
```

Java

or one of the convenient version:

Java

```
void UBindFunction (String method_name)
void UBindFunctions(String ... method_names)
void UBindFunction (Object obj, String method_name)
void UBindFunctions (Object obj, String ... method_names)
```

Java

The first function takes as argument the object containing the function (currently static methods cannot be bound). The second argument is the name of the function you want to bind. The third argument is a list of the names if the types of the arguments. For example for the function `add`, in the previous `Adder` example, we could have used:

Java

```
String[] args = { "java.lang.Double" };
UBindFunction (this, "add", args);
```

Java

provided that, of course, the signature of the function was fixed to use an `java.lang.Double` instead of a simple `double`.

If in your `UObject` you have different names for each of your methods, then you can use the shorter versions of `UBindFunction`.

The functions you can bind must follow these rules:

- They can have between 0 and 16 arguments.
- Their arguments can be of type:
 - Urbi types:
`urbi.UValue`, `urbi.UVar`, `urbi.UList`, `urbi.UBinary`, `urbi.UIImage`, `urbi.USound`, `urbi.UDictionary`
 - Java instances:
`java.lang.String`, `java.lang.Integer`, `java.lang.Boolean`, `java.lang.Double`, `java.lang.Float`, `java.lang.Long`, `java.lang.Short`, `java.lang.Character`, `java.lang.Byte`
 - Java primitive types:
`int`, `boolean`, `byte`, `char`, `short`, `long`, `float`, `double`.
- Their return type can be one of the following type:
 - `void`
 - Urbi types:
`urbi.UValue`, `urbi.UVar`, `urbi.UList`, `urbi.UBinary`, `urbi.UIImage`, `urbi.USound`, `urbi.UDictionary`
 - Java instances:
`java.lang.String`
 - Java primitive types:
`int`, `boolean`, `byte`, `char`, `short`, `long`, `float`, `double`.

27.5 Notification of a variable change or access

You can register a function that will be called each time a variable is modified by calling `UNotifyChange`, passing either an `UVar` or a variable name as first argument, and a member function of your `UObject` as second argument (and optionally a String array containing the name of the types of the arguments). The prototype for `UNotifyChange` is:

```
void UNotifyChange(String var_name, String method_name, String[] args_name);
void UNotifyChange(String var_name, String method_name);
void UNotifyChange(UVar v, String method_name, String[] args_name);
void UNotifyChange(UVar v, String method_name);
```

Java

The callback function can take zero or one argument: an `UVar` pointing to the `UVar` being modified. And the callback function must return an int (the value returned is currently ignored in the actual implementation) or nothing at all (`void`). The `notifyChange` callback function is always called after the variable value is changed.

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

27.6 Timers

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

- `USetUpdate (double period)`
Sets up a timer that calls the virtual `UObject` method `update ()` with the specified `period` (in milliseconds). Disable updates if `period` is -1.
- `USetTimer (double period, Object o, String method_name)`
or
- `USetTimer (double period, Object o, String method_name, String[] args_name)`
Invoke an `UObject` member function `method_name` every `period` milliseconds.

27.7 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.

Note however that changes on the variable coming from Urbi code or an other module can take time to propagate to the `UVar`. You can read and write all the Urbi properties of an `UVar` by reading and writing the appropriate `UProp` object in the `UVar`.

27.8 Sending Urbi code

The `send` function sends Urbi code to the server:

```
send("myTag:1+1;");
```

Java

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

```
// Java equivalent of urbiscript 'System.someFunc(12, "foo");'
call("System", "someFunc", new UValue(12), new UValue("foo"));
```

Java

They are member functions of the `UObject` class.

27.9 Providing a main class or not

We provide a main class, containing a main function, embedded in the ‘liburbijava.jar’ file. This main class, called `urbi.UMain` is responsible for the loading of the liburbijava native library, and also for the registering of your uobjects.

- When launching your UObjects with `java` or `urbi-launch-java`, please do not provide your main class.
- When launching your UObjects with Eclipse:
 - You can use the `UMain` class we provide, but this class require that you pass as argument the path to your UObject jar file, when you run your uobject.
 - Or you can provide your own main file. In this case you have to put the `UStart(YourUObject.class)` call directly in your main file, and not in your uobject classes. Your main should look like:

Java

```
import liburbi.main.*;

public class Main
{
    /// load urbijava library
    static
    {
        System.loadLibrary("urbijava");
    }

    public static void main(String argv[])
    {
        UObject.UStart(MyUObject1.class);
        UObject.UStart(MyUObject2.class);
        // ...

        UObject.main(argv);
    }
}
```

Call `System.loadLibrary("urbijava");` to load the liburbijava native library.

Note: when you call `System.loadLibrary`, java search for the library in the locations given in `java.library.path`. This special Java variable must be correctly set or you will get a loading error when you run your Java program. You can set this options giving: ‘`-Djava.library.path=path to dir containing urbijava lib`’ to the Java VM running your program.

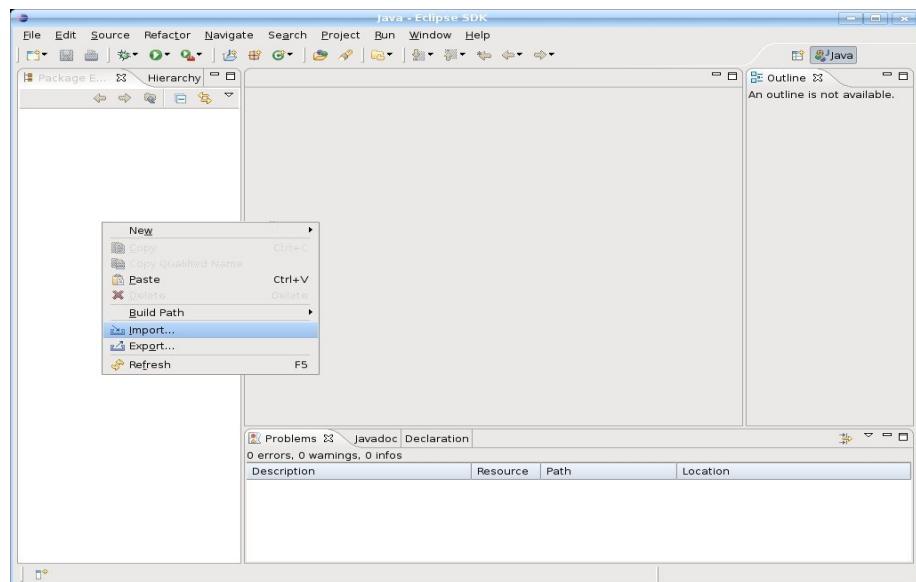
We provide two UObject examples under Eclipse. One uses `urbi.UMain`, the other provides its own main class.

27.10 Import the examples with Eclipse

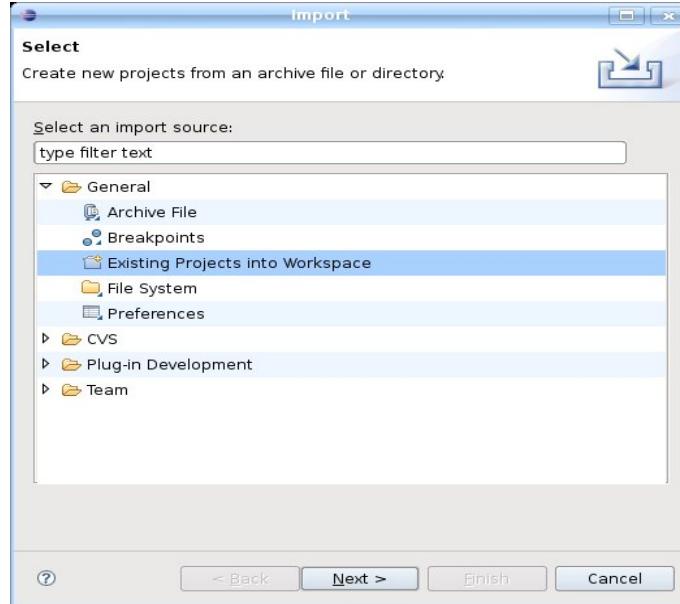
We provide a sample [Eclipse](#) project configuration that you can import in Eclipse and use to create your own UObject Java.

We illustrate here how you can do this:

1. Open Eclipse

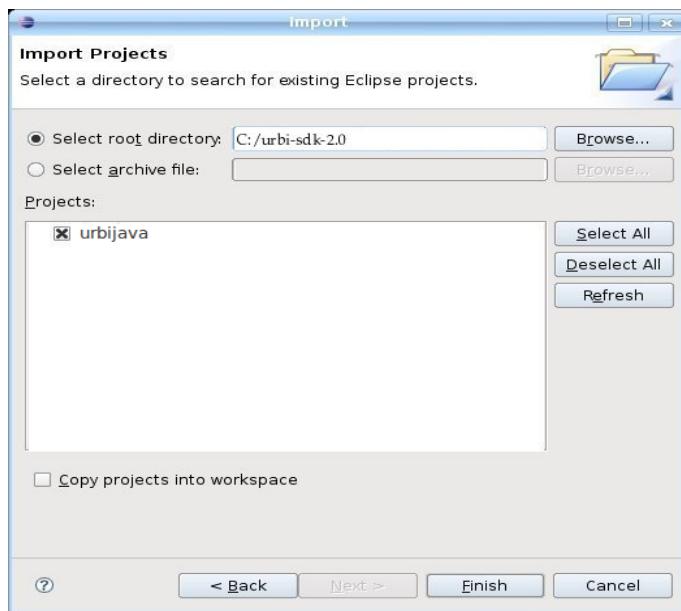


2. Right click in the Package Explorer panel and select “import” (or go in File/import)

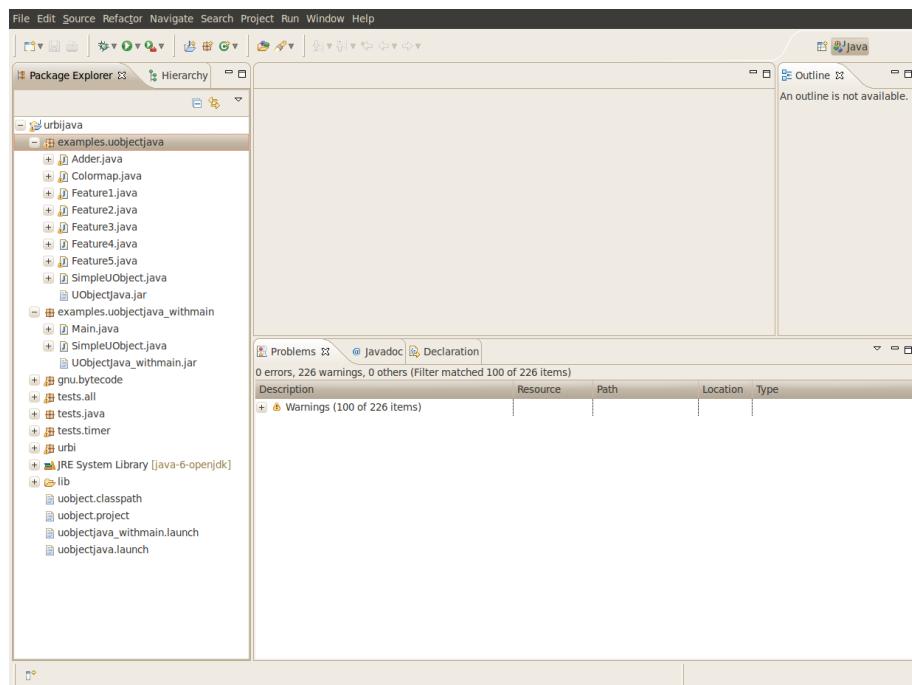


3. Select “Existing Projects into Workspace” in the opened windows

4. Click “Next”

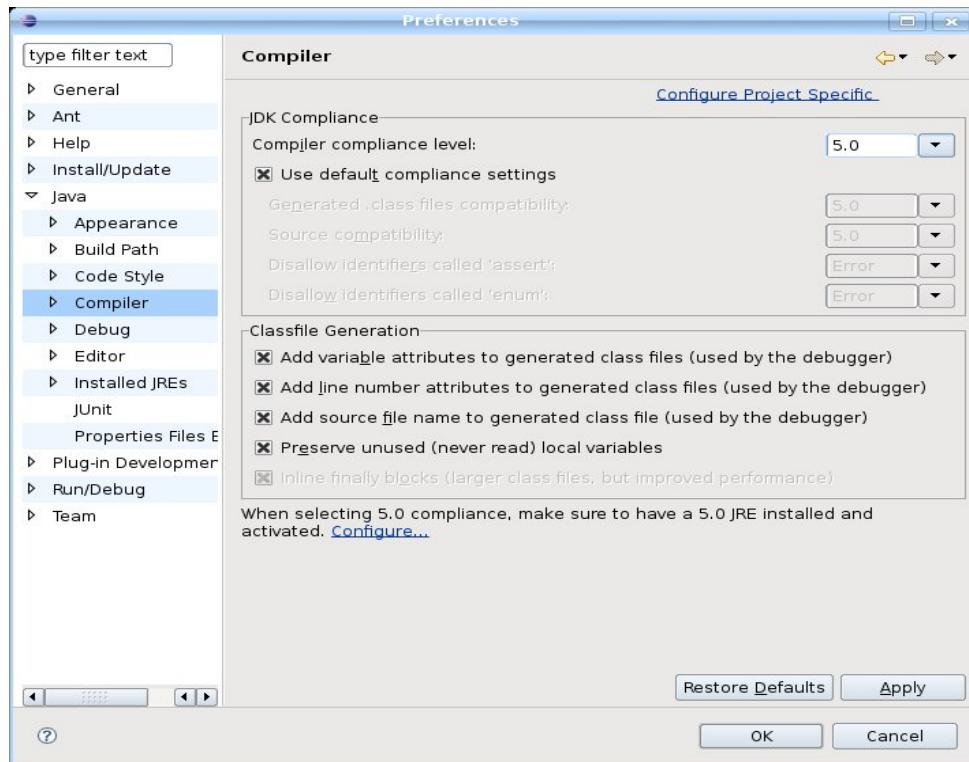


5. Enter the path of the Urbi SDK on your computer
6. Eclipse should find the ‘.project’ file we provide and display the **urbijava** project
7. Select the **urbijava** project and click “Finish”



The Java project is loaded. You can see the jar containing the liburbi ('liburbijava.jar', storing the UObject Java API) which contains the Urbi package, and also see the sources of the example we provide. We put them in the package examples. You can inspire yourself from these examples to make your own UObject. Here, we will see how to compile and run them in eclipse

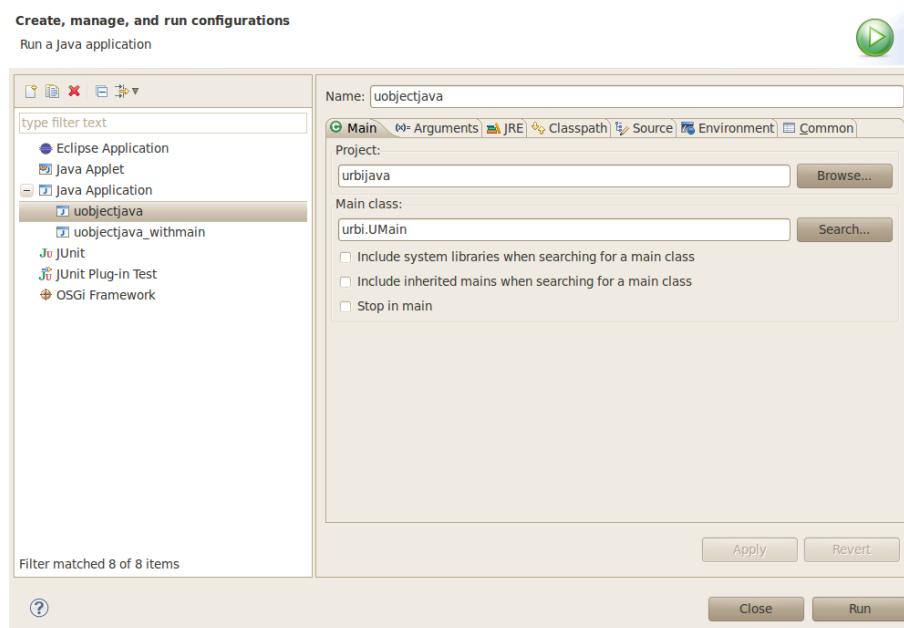
If Eclipse complains about errors in the source code, it can be that your compiler compliance level is two low. You have to set the compiler compliance level to Java 5 at least (Windows/Pref-ferences/Java/Compiler).



27.11 Run the UObject Java examples

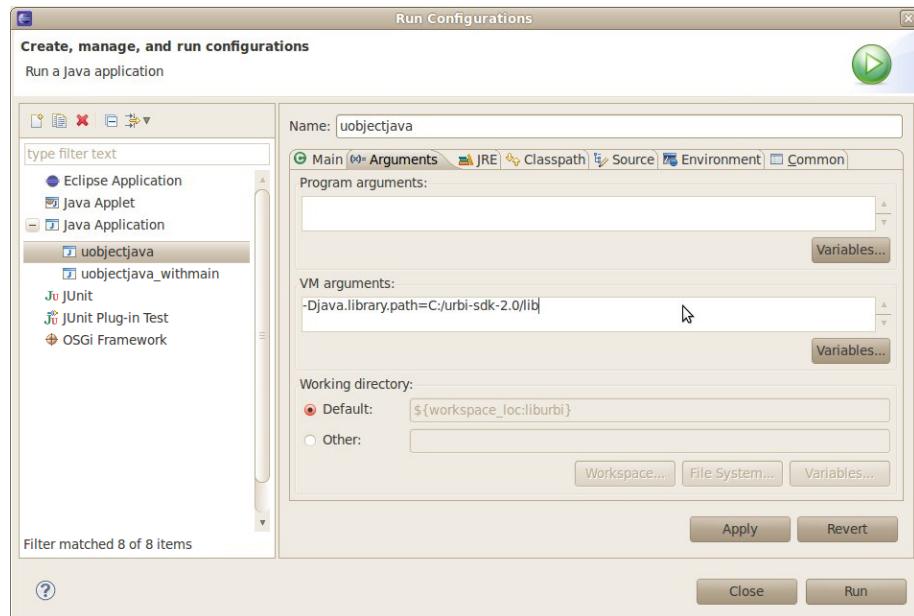
We provide a sample ‘uobjectjava.launch’ files that you can load in Eclipse to run the projects.

- Click on Run/Open Run Dialog (or “Run...” or “Run Configurations” in some versions of Eclipse)



- The launch configurations should be recognized automatically. Choose “uobjectjava”.
- The project needs to load the urbijava native library. To this end, it will search the special `java.library.path` path. If this path is not correctly set, the example will trigger an error. You have to add to `java.library.path` the path to the `lib` folder in the Urbi

SDK. You can do this from the “Run” menu, by selecting the “Arguments” tab, and setting ‘`-Djava.library.path=path-to-lib-directory`’ into the VM arguments. See:



- In order to run your remote UObject, you need also run an Urbi server. Your remote UObject will connect to this Urbi server. By default Urbi servers listen on port 54000, and remote UObjects try to connect to localhost on port 54000. If your urbi server is running on a different port or different address, then you will need to give these port and address as argument to your program, in the “Arguments” tab (something like: ‘`-H address -P port`’). You can also run ‘`--help`’ in this field, and then when you will run the program it will display some help on the arguments available.
- Click “Apply”.
- Click “Run”.

Chapter 28

Use Cases

28.1 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
```

C++

```

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.

```
C++ 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.

```
C++ // 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.
    Owned (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);
}
```

C++

```
// 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.

C++

```
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:

urbiscript
Session

```
// 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.

28.1.1 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.

28.1.2 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:

```
C++ // 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:

```
C++ // 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”.

28.2 Using Hubs to Group Objects

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

C++

```
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:

```
C++ 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:

```
C++ 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:

```
C++ URegister(servohub);
```

Finally, the implementation of our hub methods is:

```
C++ 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)
```

C++

```

{
    // 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.

28.2.1 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;
}

```

C++

28.3 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 `UIImage` 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:

C++

```
// 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);
}
```

C++

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

C++

```
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:

C++

```

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.

C++

```

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`.

28.3.1 Optimization in Plugin Mode

In plugin mode with 0-copy enabled (see [Section 26.13.3](#)), in order to avoid copying images, you can get direct access to the internal buffer by casting the `UVar` to a `const UBinary&`, from which you can get the `image`.

You cannot modify it directly.

28.4 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.

28.5 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:

C++

```

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 UIImage.
    urbi::UIImage 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 V

Tables and Indexes

About This Part

This part contains material about the document itself.

Chapter 29 — Notations

Conventions used in the type-setting of this document.

Chapter 30 — Grammar

Grammar of the urbiscript language.

Chapter 32 — Licenses

Licenses of components used in Urbi SDK.

Chapter 31 — Release Notes

Also known as the Urbi ChangeLog, or Urbi NEWS, this chapter lists the user-visible changes in Urbi SDK releases.

Chapter 33 — Bibliography

References to other documents such as documentation, scientific papers, etc.

Chapter 34 — Glossary

Definition of the terms used in this document.

Chapter 35 — List of Tables

Index of all the tables: list of keywords, operators, etc.

Chapter 36 — List of Figures

Index of all the figures: snapshots, schema, etc.

Chapter 29

Notations

This chapter defines the *notations* used in this document.

29.1 Words

- **code**
A *piece of code* (urbiscript, Java, C++ ...).
- **comment**
A *comment* in some programming language. For instance, `/* hello /* world */ ! */` is a comment in urbiscript.
- **environment-variable**
An *environment variable* name, e.g., PATH.
- **'file-name'**
A *file name*.
- **keyword**
A *keyword* in some programming language. For instance, `watch` is an urbiscript keyword.
- **meta-variable**
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 }.`
- **string**
A *string* in some programming language. For instance, `"Hello, world!"` is a string in urbiscript.

29.2 Frames

29.2.1 C++ Code

C++ source code is presented in frames as follows.

```
class Int
{
public:
    Foo(int v = 0)
        : val_(v)
```

C++

```

    {}

void operator(int v)
{
    std::swap(v, val_);
    return v;
}

int operator() const
{
    return val_;
}

private:
    int val_;
};
```

29.2.2 Grammar Excerpts

The *grammar fragments* are written in *EBNF* (*Extended Backus-Naur Form*). The symbol `::=` separates the left-hand symbol from the right-hand side part of the rule. Infix `|` denotes alternation, postfix-`*` 0-or-more repetition, postfix-`+` 1-or-more repetition, and postfix-`?` denotes optional parts. Terminal symbols are written in double-quotes, and non-terminals in angle-brackets. Parentheses group.

The following frame defines the grammar syntax expressed in the same grammar syntax.

Grammar Excerpt

```

<grammar> ::= <rule>+
<rule> ::= <symbol> ":" <rhs>
<rhs> ::= <rhs>*
        | <rhs> " | " <rhs>
        | <rhs> ("?" | "*" | "+")
        | "(" <rhs> ")"
        | <symbol>
<symbol> ::= "<" <identifier> ">"
            | "<" <escaped-character>* ">"
            | "<" <escaped-character>* ">"
```

[Chapter 30](#) provides the grammar of urbiscript.

29.2.3 Java Code

Java source code is presented in frames as follows.

Java

```

import liburbi.main.*;
public class Main
{
    /// Load urbijava library.
    static
    {
        System.loadLibrary("urbijava");
    }

    public static void main(String argv[])
    {
        // Does nothing for now.
    }
}
```

29.2.4 Shell Sessions

Interactive sessions with a (Unix) shell are represented as follows.

```
$ echo toto
toto
```

Shell
Session

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.

29.2.5 urbiscript Sessions

Interactive sessions with Urbi are represented as follows.

urbiscript
Session

```
echo("toto");
[00000001] *** toto
```

Contrary to shell interaction (see [Section 29.2.4](#)), 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.

urbiscript
Session

```
cout << "toto";
[00000002:output] "toto"
```

29.2.6 urbiscript Assertions

The following *assertion frame*:

```
true;
1 < 2;
1 + 2 * 3 == 7;
"foobar"[0, 3] == "foo";
[1, 2, 3].map (function (a) { a * a }) == [1, 4, 9];
[ => ].empty;
```

Assertion
Block

denotes the following assertion-block (see [Section 21.9](#)) 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;
};
```

urbiscript
Session

Chapter 30

Grammar

This chapter is work-in-progress. We intend to describe the grammar fully in the forthcoming releases. Please, be patient!

Chapter 31

Release Notes

This chapter (also known as the Urbi ChangeLog, or Urbi NEWS) lists the user-visible changes in Urbi SDK releases.

31.1 Urbi SDK 3.0



Released on 2011-XX-YY.

This release includes major changes and incompatibilities with urbi 2. A transition guide 2011-XX-YY can be found in [18](#).

31.1.1 Major changes

- Function evaluation no longer occurs if parenthesis are omitted.
- Proper properties (slot with an associated getter and setter functions) have been implemented: [8.8](#).
- **Slots**, the objects that holds the properties are now accessible in urbiscript through **Slot.getSlot** and **Slot.setSlot**. The old **Slot.getSlot** and **Slot.setSlot** functions are renamed to **Slot.getValue** and **Slot.setValue**. Properties are the slots of the slot object, so `x->foo` is the same thing as `getSlot("x").foo`, or `&x.foo`.
- Code can be organized into packages that can be imported: [9](#).
- All the urbiscript and C++ notification mechanisms (at, UNotifyChange, changed event) were unified in an unique system using **Subscriptions** on **Events**.
- **Vector** and **Matrix** provide support for efficient linear algebra operations.
- Type constraints can be set on function arguments([21.3.2.2](#)) and slots (**Slot.type**).
- **stopif**[21.10.8](#) and **freezeif**[21.10.9](#) now accept event and condition with duration as argument.

31.1.2 Fixes

- The indentation of the Emacs urbiscript mode is fixed.
- **File.asPath** was documented, but not implemented.
- **List.sort** no longer crashes when the predicate is not a strict weak ordering.
- Don't execute the body of a **finally** a second time when an exception is raised inside the **finally** clause.

- Do not emit `Tag.leave` events when a frozen tag waiting to execute statements is stopped.
- Avoid unexpected execution of code caused by `Barrier.signal` (or `Barrier.signalAll`) and a stop of the code waiting on the barrier.
- Be independent of changes of locale settings (such as input/output format for floats) in C++ user code.
- Restore proper behavior on closed standard input.
- When setting a property on an inherited slot, really duplicate the slot instead of making an alias (`Object.setProperty`).
- Declaring constant variables (or slots) without an initial value is a syntax error (see the constant property, [Section 21.4.2](#)).
- Copy-on-write does not apply to slots declared constant (`Object.updateSlot`).

31.1.3 Changes

Please, pay attention to these changes, as your code may have to be adjusted.

31.1.3.1 urbscript

- The class syntax can now be used to inherit from primitives.
- The implicit empty statement, which was warned about:

urbscript
Session

```
if (true) else 12;
[00011780:warning] !!! implicit empty statement. Use '{}' to make it explicit.
```

is now an error:

urbscript
Session

```
if (true) else 12;
[00003258:error] !!! syntax error: unexpected else
```

Write `if (true) {} else 12;`. As a consequence `if (x) &y;` is no longer ambiguous: its unique interpretation is `if (x) { &y };` and no longer `{ if (x) {} } & y;;`

```
{ if (true) &true else &false } === &true;
{ if (false) &true else &false } === &false;
```

urbscript
Session

- Implicit tags are restricted to single identifiers in interactive sessions. In other words, in the Urbi shell you may expect tags to be automatically created:

urbscript
Session

```
assert (!hasSlot("foo"));
foo: 123;
[00003258] 123
assert (hasSlot("foo") && foo.isA(Tag));
```

This is no longer valid for complex tags, nor when used deeper in the code:

urbscript
Session

```
bar.baz: echo(2);
[00015621:error] !!! lookup failed: bar

class Foo { function f() { t1: echo (12) } }|;
Foo.f;
[00015631:error] !!! lookup failed: t1
[00015631:error] !!!      called from: f
```

See [Section 21.10.1](#) for details.

- Drop support for the obsolete Urbi SDK 1 syntax for events: `at (?e)` and `emit e` instead of `at (e?)` and `e!` ([Section 17.3](#)). As a consequence, `emit` is now a regular identifier.
- Drop support for the obsolete `loopn` keyword, replaced by `for` since Urbi SDK 2.0 ([Section 17.7](#)).

31.1.3.2 urbiscript Standard Library

- Features (`basename`, `lastModifiedDate`) of `Directory` and `File` which are actually related to `Path` are deprecated: use `f.asPath.basename` or `f.asPath.lastModifiedDate` instead.
- `FormatInfo` supports positional arguments:

Assertion Block

```
"%2%, %|1$10|!" % ["world", "Hello"] == "Hello,      world!";
```

- Opened Streams (`InputStream` and `OutputStream`) can no longer be cloned.
- `InputStream` and `OutputStream` are no longer pre-opened (to standard input and output).
- `InputStream.get` (and `InputStream.getChar`, `InputStream.getLine`) now returns `nil` instead of `void` when it reaches the end of file. To provide backward compatibility with previous code such as:

urbiscript Session

```
var i = InputStream.new(File.new("file.txt"))|;
var x;
while (!(x = i.get.acceptVoid).isVoid)
    cout << x;
i.close;
```

`nil.isVoid` now returns `true` and issues a warning: eventually it will return again `false`. Rewrite your code as:

urbiscript Session

```
var i = InputStream.new(File.new("file.txt"))|;
var x;
while (!(x = i.get).isNil)
    cout << x;
i.close;
```

- `Object.asToplevelPrintable` was deprecated in favor of `Object.asTopLevelPrintable`, for consistency (with `Channel.topLevel` for instance). Because this backward compatibility function may also hide failures (e.g., if you refine it using its former name), it is removed.
- `Object.'^'` implements the Boolean exclusive or (not to be confused with the bitwise exclusive or, `Float.'^'`).
- When destroyed, `Streams` (including `InputStreams` and `OutputStreams`), are closed if not already closed.
- `Subscription` replaces former `UConnection`.
- The internal functions `System.backtrace`, `System.jobs`, and `System.aliveJobs` are removed in favor of `Job.backtrace`, `Job.jobs` and `Job.jobs.size`.
- `System.currentRunner` is deprecated in favor of `Job.current`.
- `System.getenv`, `System.setenv`, `System.unsetenv` are deprecated in favor of `System.env`.
- `System.shutdown` now accepts the exit status as argument.

- `System.spawn` is deprecated in favor of `Code.spawn`.
- accepts an optional maximum queue size.

31.1.3.3 Miscellaneous

- The undocumented `yield_until_things_changed` was removed from the UObject API, this feature was not safe.

31.1.4 New features

31.1.4.1 urbiscript

- To augment legibility, underscores can be used to separate groups of digits in numbers ([Section 21.1.6.4](#)).

```
123_456_789 == 123456789;
12_34_56_78_90 == 1234567890;
1_2__3___45 == 12345;
1_2.3__4 == 12.34;
0xFFFF_FFFF == 0xFFFFFFFF;
1e1_0 == 1e10;
```

Assertion Block

- Comparisons can be chained, with guarantees over the order and number of evaluations of the operands ([Section 21.1.8.6](#)):

```
function v(x) { echo(x) | x }|;
v(1) == v(2) < v(3) < v(4) || v(10) < v(11) != v(12) <= v(13);
[00033933] *** 1
[00033933] *** 2
[00033933] *** 10
[00033933] *** 11
[00033933] *** 12
[00033933] *** 13
[00033933] true
```

urbiscript Session

- In addition to postfix increment/decrement operators, prefix operators are supported and redefinable ([Section 21.1.8.3](#)).

```
var x = 0;
++x == 1; x++ == 1; x == 2;
--x == 1; x-- == 1; x == 0;
```

Assertion Block

- The `timeout` construct now features optional `catch`, `else`, and `finally` clauses, modeled after the syntax of `try`-blocks. See [Section 21.10.7](#).

```
function testTimeOut(var duration)
{
    timeout (1s) { echo("computation"); sleep(duration); "body-value" }
    catch      { echo("interrupted!");           "catch-value"   }
    else       { echo("completed");            "else-value"   }
    finally    { echo("finally");             "finally-value" }
};

// Run till completion.
testTimeOut(0s);
[00000264] *** computation
[00000265] *** completed
[00000265] *** finally
[00000265] "else-value"
```

urbiscript Session

```
// Interrupted before completion.
testTimeOut(2s);
[00000266] *** computation
[00001267] *** interrupted!
[00001270] *** finally
[00001271] "catch-value"
```

31.1.4.2 urbscript Standard Library

- `Job.backtrace` now includes information about event handling (both sending and receiving parts).

urbscript Session

```
//#push 1 "file.u"
var f = Event.new();
function watchEvent()
{
    at (f?)
        lobby.echoEach(Job.current.backtrace);
}|;
function sendEvent()
{
    f!;
}|;
watchEvent;
sendEvent;
[00008208] *** file.u:5.20-40: backtrace
[00008208] *** file.u:9.3: emit
[00008208] *** file.u:12.1-9: sendEvent
[00008208] *** ---- event handler backtrace:
[00008208] *** file.u:4.3-5.41: onEvent
[00008208] *** file.u:11.1-10: watchEvent
```

On the same example, Urbi SDK 2.7.4 displays only:

urbscript Session

```
[00000030] *** file.u:5.20-42: backtrace
```

- `Group.'=='` checks equality between group members.
- `InputStream.content` returns the content of the stream as a `String`.
- `Tuple.hash`, which enables using tuples as Dictionary keys.
- `List.unique`.
- `List.asTree`, `Dictionary.asTree` return a tree representation for `List` and `Dictionary` as a `String`.
- `Logger` provides runtime control over the debug traces:

- `Logger.level` provides read/write access to the current verbosity level,
- `Logger.categories` gives the set of existing categories,
- categories that match a given pattern can be enabled/disabled (`Logger.enable`, `Logger.disable`),

urbscript Session

```
Logger.enable("Category*");
Logger.disable("Category.Sub");
```

- the status of all the categories can be changed according to a given specification string similar to `GD_CATEGORY` (`Logger.set`).

```
Logger.set("+Category*, -Category.Sub");
```

urbscript Session

- The functions `Math.ceil` and `Math.floor` were added, consistently with `Math.round` and `Math.trunc`. Yet users are encouraged to prefer the “method-like” versions, i.e., `Float.ceil` and so forth.

```
Math.ceil(1.2) == 1.2.ceil();
Math.floor(1.2) == 1.2.floor();
Math.trunc(1.2) == 1.2.trunc();
Math.round(1.2) == 1.2.round();
```

Assertion Block

- `String.'[]'` accepts negative indexes, as `List.'[]'` did.

```
"foobar"[-1] == "r";    "foobar"[-6, -3] == "foo";
```

Assertion Block

- `String.asFloat` is on par with the syntax supported for literal `Floats`, including support for underscores and hexadecimal.

```
"123_456_789".asFloat() == 123_456_789;
"0xFFFF_FFFF".asFloat() == 0xFFFF_FFFF;
```

Assertion Block

- `String.find` and `String.rfind` can search a pattern forward or backward inside a `String`.

Assertion Block

```
"Hello, World!".find("o") == 4;
"Hello, World!".find("o", 5) == 8;
"Hello, World!".find("o", 9) == -1;

"Hello, World!".rfind("o") == 8;
"Hello, World!".rfind("o", 7) == 4;
"Hello, World!".rfind("o", 3) == -1;
```

- `String.isBlank` (spaces and horizontal tabulations) completes the table of character classification functions ([Table 22.1](#)).
- `System.getLocale` and `System.setLocale` allow to change the locale settings.

31.1.4.3 Miscellaneous

- The C++ classes `USound` and `UIImage` now properly initialize their instances to empty sound/image.
- Add `gdb` extensions to improve debug experience of Urbi and urbiscript ([Section 19.4](#)). Urbi objects are pretty printed when accessed and commands are available for printing the backtrace of the current Job and for adding/removing breakpoints on urbiscript.

31.1.5 Documentation

- The various flavors syntactic constructs are better documented, see [Table 21.14](#).
- Extended description of `Group`.
- `String.isAlnum`, `String.isAlpha`, `String.isCntrl`, `String.isdigit`, `String.isGraph`, `String.isLower`, `String.isPrint`, `String.isPunct`, `String.isSpace`, `String.isUpper` and `String.isXdigit`.
- Extended description of bitwise operators ([Section 21.1.8.4](#)).
- Completion and fixes of the description of the operators in general ([Section 21.1.8](#)).
- Better description of `closure` vs. `function` ([Section 21.3.6](#)).

- Formal description of the grammar has started, but is yet to finish ([Chapter 30](#)).
- Extended description of [Event](#).
- The logging categories used by Urbi SDK ([Section 22.36.2](#)).
- The command line interface of UObjects launched by `urbi-launch` ([Section 20.5.3](#)).
- Bibliography, [Chapter 33](#).
- `Object.asPrintable`, `Object.asString`, `Object.asTopLevelPrintable`.
- `System.currentRunner`.



31.2 Urbi SDK 2.7.5

Released on 2012-01-27.

2012-01-27.

This release fixes some packaging-related minor issues.

31.2.1 Fixes

- The source tarballs are significantly smaller.
- Some public C++ headers used to depend on private ones; this is fixed.
- In-place builds from the source tarballs (i.e., when compiling in the same directory as the source) work. Yet, we still discourage them ([Chapter 19](#)).
- The test suite properly skips tests when some preconditions are not met (e.g., Java support not compiled in, or running as root, or `socat` not being available).
- Windows packages with the installer now have some urbscript files (namely ‘`platform.u`’) which properly depend on whether you are using the debug or release flavor.
- GeSHi support is properly installed.
- When defining functions, their qualified name may start with `this`:

urbscript
Session

```
function this.foo() { echo("foo"); }|;
this.foo();
[00016170] *** foo
```

- Timestamps now use a monotonic clock, insensitive to wall clock changes (fired by `ntpdate` for instance).

31.2.2 Changes

- Compatibility with Clang++ 2.1 and GCC 4.6.
- Compatibility with Boost 1.48. Beware that because of bugs in `Boost.Foreach`¹ (see [Ticket 6131](#)²), Urbi SDK now defines a macro `foreach` much more widely than before. Never include ‘`boost/foreach.hpp`’, rather, use ‘`libport/foreach.hpp`’.
- Thanks to Adam Oleksy, we now provide Debian and RedHat packages.
- `urbi-launch-java` supports the new option ‘`[C]check`’ which checks if Java support is available.

¹[Boost.Foreach](http://www.boost.org/doc/libs/release/libs/foreach/), <http://www.boost.org/doc/libs/release/libs/foreach/>.

²[Ticket 6131](https://svn.boost.org/trac/boost/ticket/6131), <https://svn.boost.org/trac/boost/ticket/6131>.

- We now use the version 8 of the Independent JPEG Group's (IJG) ‘libjpeg’.
- To improve performances, the Boolean operators `&&` and `||` can no longer be overridden. In other words, `a && b` no longer maps to `a.'&&'(b)`, but to `if (a) b else a` (with provisions to avoid multiple computations of `a`).
- `System.timeReference` is a `Date`.

31.2.3 New Feature

- `Date` features microsecond support (`Date.microsecond`, `Date.us`).
- `Duration` is accurate at the microsecond.

31.2.4 Documentation

- Extensive overhaul of the HTML rendering of the documentation; compare <https://github.com/urbiforge/urbi/urbi/2.7.5/doc/urbi-sdk.htmldir/> to <https://github.com/urbiforge/urbi/urbi/2.7.4/doc/urbi-sdk.htmldir/>. Please, pay extra attention to the “Index” button.
- The HTML documentation is also available as a single HTML document (<https://github.com/urbiforge/urbi/urbi/2.7.5/doc/urbi-sdk-single.htmldir/>).
- Table of figures (Chapter 36) and table of tables (Chapter 35).
- Bibliography, Chapter 33.
- `urbi-launch-java` is documented, Section 20.6.
- The developer documentation for Urbi SDK Remote Java is included in the binary packages (as ‘`share/doc/urbi-sdk/doc/sdk-remote-java.htmldir`’).

31.3 Urbi SDK 2.7.4

Released on 2011-11-17.

2011-11-17. This release back-ports several fixes from the forthcoming next major release of Urbi, to the 2.7 family.

31.3.1 Fixes

- Freezing an event handler could prevent other event handlers to function properly.
- The indentation of the Emacs urbiscript mode is fixed (contributed by Jeremy W. Sherman).
- Binding a single UVar several times no longer crashes.
- Handling of text and binary files on Windows should no longer be a problem.
- Disconnection of remote UObjects behaves properly.
- In Windows packages, the suffixes of the library (e.g., ‘`-vc90-d`’) are restored.



31.3.2 Changes

- Boost requirement is now 1.40 instead of 1.38.
- The binary packages for Windows, Mac OS X, Debian Etch are built with Boost 1.47 (from Boost Pro, MacPorts, and install by hand).
- The binary packages for GNU/Linux Ubuntu Lucid are built with the packaged version of Boost: 1.40.
- All the binary packages are now built with ROS Diamondback instead of ROS CTurtle.
- We now provide pkg-config files: ‘libport.pc’ and ‘urbi.pc’. Since binary packages are relocatable, it should be noted that the *prefix* is most probably wrong, so it should be defined at runtime as the output of the urbi’s (and urbi-launch’s) new option ‘print-root’:

```
$ pkg-config urbi --cflags
-I/prefix/include
# But urbi was installed in /usr/local, not in /prefix.
# There is nothing there.
$ ls /prefix/include
ls: /prefix/include: No such file or directory

# Let urbi give urbi-root (or urbi-prefix) to pkg-config:
pkg-config --define-variable=prefix=$(urbi --print-root) urbi --cflags
-I/usr/local/bin/../include
# This time, it exists.
$ ls /usr/local/bin/../include
boost jconfig.h jmorecfg.h libport urbi
gostai jerror.h jpeglib.h serialize
```

Shell Session

- GeSHi (Generic Syntax Highlighter)³ support to display colored urbiscript on websites using php.

31.3.3 Documentation

- Instructions to build Urbi SDK are more precise. The requirements should be easier to find ([Section 19.1](#)).
- Formatting of the naming standard is improved ([Chapter 24](#)).
- Instruction for exchanging UObject between UObjects have been clarified ([Section 26.14](#)).
- Various errors in the documentation of UObject were fixed.

31.4 Urbi SDK 2.7.3

Released on 2011-10-07.



2011-10-07

31.4.1 Fixes

- File descriptor leaks when using `Process`.

³GeSHi (Generic Syntax Highlighter), <http://qbnz.com/highlighter/>.

31.4.2 Changes

- Compatibility with Boost 1.46.
- Binary packages now include simple aliases to the Boost libraries (e.g., you may use ‘-lboost_date_time’ instead of ‘-lboost_date_time-gcc44-mt-1_38’).
- Binary packages on Ubuntu Lucid now use its native Boost libraries (1.40) instead of Boost 1.38, and were built with ROS Diamondback.

31.4.3 Documentation

- Support for Gostai Jazz.

31.5 Urbi SDK 2.7.2

Released on 2011-05-13.

There was no public release of Urbi SDK 2.7.2.

31.5.1 Fixes

- Avoid unexpected execution of code caused by [Semaphore](#) release and a stop of the code acquiring the semaphore. This caused pieces of code such as the following not to end.

```
var m = Mutex.new();
for& (10)
{ m.stop | m: { sleep(1) } };
"done";
[00016170] "done"
```

urbscript
Session

Now [Semaphore.acquire](#) either returns when it holds the semaphore or it jumps to the end of the stopped tag.

31.6 Urbi SDK 2.7.1

Released on 2011-03-17.



2011-03-17

31.6.1 Fixes

- Crash when stopping a UObject threaded function via a tag.
- On Mac OS X, `umake` and friends pass the ‘-arch’ option. It is now easier to use on a 64 bit computer an Urbi SDK package built on a 32 bit one.

31.6.2 Changes

- The default activation for `GD_CATEGORY` is computed from the first character: ‘`Libport.Path`’ is equivalent to ‘`*,+Libport.Path`’, and ‘`-Urbi*`’ is equivalent to ‘`*,+Urbi*`’. See [Section 20.1.2](#).
- `System.requireFile` supports the same arguments as `System.load`.

31.6.3 Documentation

- [Chapter 25](#), a quick introduction to some of the basic features of the UObject architecture.
- [uobjects](#).
- gnu.bytecode license, [Section 32.4](#).

31.7 Urbi SDK 2.7

Released on 2011-03-10.

Many optimizations have been implemented, and users should observe a significant speedup. Particularly, the threaded support in UObjects has been modified to perform all operations asynchronously, instead of locking the engine.

31.7.1 Changes

- `WeakDictionary`, `WeakPointer` are removed. `UVar.notifyAccess`, `UVar.notifyChange`, and `UVar.notifyChangeOwned` no longer need a handler as first argument. Instead of:

```
urbiscript
Session
var myHandle = WeakPointer.new();
&sensorsLoad.notifyChange(myHandle,
    closure() { if (sensorsLoad) sensorsOn else sensorsOff; });
```

write:

```
urbiscript
Session
&sensorsLoad.notifyChange(closure()
{ if (sensorsLoad) sensorsOn else sensorsOff; });
```

Backward compatibility is ensured, but a warning will be issued.

- `Dictionary` Duplicate keys in `Dictionary` literals are now an error. Instead of setting the value of the last occurrence, it will throw an error.

```
urbiscript
Session
["one" => 1, "one" => 2];
[00000001:error] !!! duplicate dictionary key: "one"
```

- The functions `urbi::convertRGBtoYCrCb` and `urbi::convertYCrCbtoRGB` have been renamed as `urbi::convertRGBtoYCbCr` and `urbi::convertYCbCrtoRGB` (i.e., a change from ‘YCrCb’ to ‘YCbCr’). Because the previous behavior of these functions was incompatible with their names, after careful evaluation, it was decided not to maintain backward compatibility: it is better to make sure that code that depends on these functions is properly adjusted to their semantics.

31.7.2 New Features

- `Logger` provides a logging service:

```
urbiscript
Session
var logger = Logger.new("Category")|;

logger.dump << "Low level debug message"|;
// Nothing displayed, unless the debug level is set to DUMP.

logger.warn << "something wrong happened, proceeding"|;
[      Category      ] something wrong happened, proceeding

logger.err << "something really bad happened!"|;
[      Category      ] something really bad happened!
```

- `Profile` and `Profile.Function` replace the former `Profiling` object.
- `Stream`, common prototype to `InputStream` and `OutputStream`.
- `System.sleep`’s argument now defaults to `Float.inf`.
- Controlling the maximum queue size for UObject threaded notifies and bound functions is now possible, see [Section 26.4.3](#).

- `at` now comes in synchronous and asynchronous flavors, see [Section 21.11.1.3](#).
- A new construct, `watch`, creates an event that allows to monitor any change of an expression ([Section 21.11.3](#)).

```
var x = 0|;
var y = 0|;
var e = watch(x + y)|;
at (e?(var value))
    echo("x + y = %s" % value);
x = 1|;
[00000000] *** x + y = 1
y = 2|;
[00000000] *** x + y = 3
```

urbiscript
Session

- Gostai Editor for Windows was updated to 2.5. It now includes advanced search and replace features with regular expression support and a “find in all opened documents” option. “Goto line” menu has also been added.
- Gostai Console 2.6 for Windows now offers autocompletion of urbiscript slot names.
- To make simpler to install several versions of Urbi SDK, the Windows installers now include the version number in the destination path.

31.7.3 Documentation

- [Chapter 16](#), A programming guideline in Urbi SDK.
- `UVar.notifyChangeOwned`, `UVar.removeNotifyAccess`, `UVar.removeNotifyChange`, and `UVar.removeNotifyChangeOwned`.
- `urbi-sound` ([Section 20.9](#)).

31.8 Urbi SDK 2.6

Released on 2011-01-06.

This release features several deep changes that are not user visible, but which provide significant optimizations. Several bugs have been fixed too. 2011-01-06.



31.8.1 Fixes

- Improper behavior when there are several concurrent `at` (`exp ~ duration`).
- System interruptions with Control-C sometimes failed.
- Remote UObjects exit properly when the server shuts down.
- Binary packages provide RTP support for all the architectures.

31.8.2 Optimizations

- urbiscript interpretation has been globally sped up by around 30%.
- Event emission routines have been optimized: `Event.emit` by around 25%, `Event.syncEmit` by around 13%, `Event.trigger` by around 25% and `Event.syncTrigger` by around 10%.

31.8.3 New Features

- The usual C++ syntax to declare classes with multiple inheritance is supported, see [Section 21.1.6.8](#).
- A literal syntax for strict variadic functions has been added, see [Section 21.3.7](#). For instance

```
urbiscript
Session

function variadic(var a1, var a2, var a3, var args[])
{
    echo("a1 = %s, a2 = %s, a3 = %s, args = %s"
        % [a1, a2, a3, args]);
}
variadic(1, 2, 3);
[00000002] *** a1 = 1, a2 = 2, a3 = 3, args = []
variadic(1, 2, 3, 4);
[00000002] *** a1 = 1, a2 = 2, a3 = 3, args = [4]
variadic(1, 2, 3, 4, 5);
[00000002] *** a1 = 1, a2 = 2, a3 = 3, args = [4, 5]
```

This is faster than using lazy functions and call messages.

- `Directory` objects have new features for creation, modification and deletion.

```
urbiscript
Session

Directory.createAll("dir1/dir2/dir3")|;
Directory.new("dir1").rename("dir")|;
Directory.new("dir/dir2").copy("dir/dir4")|;
Directory.new("dir").removeAll;
```

- `Directory.size`, `File.size`, `Directory.lastModifiedDate`, `File.lastModifiedDate`.

31.8.4 Documentation

- `Lobby.bytesReceived`, `Lobby.bytesSent`.

31.9 Urbi SDK 2.5

Released on 2010-12-07.

2010-12-07.

31.9.1 Fixes

- Memory consumption at start-up is reduced.

31.9.2 New Features

- `urbi-launch` and `urbi-send` support `'-m'/'--module=file'`, to load a module ([Section 20.5](#), [Section 20.8](#)).
- The search paths for urbiscript files and for UObject files can be changed from urbiscript ([System.searchPath](#), [UObject.searchPath](#)).
- New syntactic sugar for `Object.getSlot`: `o.&name` is equivalent to `o.getSlot("name")` (and `&name` is equivalent to `getSlot("name")`). For instance, instead of

```
urbiscript
Session

function Derive.init(var arg)
{
    Base.getSlot("init").apply([this, arg]);
};

function Foo.'=='(var that)
{
```

```
    getSlot("accessor") == that.getSlot("accessor");
};
```

write

```
function Derive.init(var arg)
{
  Base.&init.apply([this, arg]);
};

function Foo.'=='(var that)
{
  &accessor == that.&accessor;
};
```

urbiscript
Session

- `Dictionary` keys can now be arbitrary objects. Objects hashing can be overridden. See `Object.hash`.
- `Global.warn` sends messages prefixed with `!!!` (as error messages), instead of `***`.
- `Hash`, type for hash codes for `Dictionary`.
- `List.insertUnique` inserts a member if it's not already part of the list.
- `Object.hash`, `Float.hash`, `String.hash`, `List.hash`.
- `Object.removeLocalSlot`. It raises an error when asked to remove of a non-existing slot. Please note that, contrary to what its name suggests, `Object.removeSlot` is only removing *local* slots. Using `Object.removeLocalSlot` is encouraged.
- `System.eval`, `System.load`, and `System.loadFile` accept an optional second argument, the context (`this`) of the evaluation.

31.9.3 Changes

- More types of empty statements are warned about. For instance Urbi used to accept silently `if (foo);`. It now warns about the empty body, and recommends `if (foo) {};`.
- `Dictionary.erase` raises an error if the key does not exist.
- `Exception.ArgumentType` is deprecated, use `Exception.Argument` that wraps around any `Exception` instead.
- `Object.getProperty` raises an error if the property does not exist. It used to return `void`.
- `Object.removeSlot` warns when asked to remove of a non-existing slot, and `Object.removeSlot` about non-existing properties.

```
removeSlot("doesNotExist")|;
[00000002:warning] !!! no such local slot: doesNotExist
[00000002:warning] !!!      called from: removeSlot
```

urbiscript
Session

In the past, it used to accept this silently; in the future, this will be an error, as with `Object.removeLocalSlot`. Use `Object.hasLocalSlot` or `Object.hasProperty` beforehand, if needed.

- A warning is now issued when the evaluation of the condition of an `at` statement yields an Event and no question mark was used, since this is most likely an oversight.

urbiscript
Session

```
var e = Event.new();

// This is not the correct way to match an event.
at (e) echo("Oops.");
[00000002:warning] !!! at (<event>) without a '?', probably not what you mean
[00000003] *** Oops.

// This is.
at (e?) echo("Okay.");
```

- `File.rename` returns `this`.

31.9.4 Documentation

- The `urbi-ping` program, [Section 20.7](#).
- Properties, [Section 8.7](#).
- The `class` statement is better described in [Section 8.4](#) and [Section 21.1.6.8](#).
- The tutorial documents the payloads in event-based constructs ([Section 12.2.2](#)).

31.10 Urbi SDK 2.4

Released on 2010-10-20.

2010-10-20.

31.10.1 Fixes

- Fix transmission of binary data in dictionaries from remote UObjects.
- Fix sound conversion between mono and stereo.
- Fix the `urbi-sendsound` liburbi example.
- Fix liburbi stream formatting that made the code waste a lot of bandwidth.
- Optimize runtime performances when log messages are inhibited.
- Preserve properties when a property assignment triggers a copy-on-write.
- Trigger the `changed` attribute of tags when they are frozen, unfrozen, blocked, ...

This fixes `at (myTag.frozen) ...;`
- Fix `Float.random` on windows, which always returned 41 in some situations.
- Do not scope variables created inside a pipe inside a comma:

```
var a = 0 | var b = 1,
echo(a);
[00000001] *** 0
echo(b);
[00000001] *** 1
```

31.10.2 New Features

- Issue a warning when an UObject is plugged in a different version of the kernel than the SDK that compiled it — which can provoke undefined runtime behavior.
- Java API for UObject, see [Chapter 27](#).
- Enumerations.

Enumeration types can be created with the usual C-like syntax. See [Section 21.5](#) and [Enumeration](#).

urbiscript
Session

```
enum Suit
{
    hearts,
    diamonds,
    clubs,
    spades, // Last comma is optional
};

[00000001] Suit

for (var suit : Suit)
    if (suit in [Suit.spades, Suit.clubs])
        echo("Black: " + suit)
    else
        echo("Red: " + suit);
[00000001] *** Red: hearts
[00000002] *** Red: diamonds
[00000003] *** Black: clubs
[00000004] *** Black: spades
```

- `umake` supports new options: ‘`-I`’, ‘`-L`’, ‘`-l`’, ‘`--package`’ (for `pkg-config`). See [Section 20.10](#). The documentation of `umake` now describes `EXTRA_CPPFLAGS`, `EXTRA_CXXFLAGS`, and `EXTRA_LDFLAGS`.
- RTP mode can now be switched on at any time. Change is applied to existing notifies.
- UObjects can now be instantiated directly from C++, both in plugin and remote mode.
- New mechanism to map simple structures between C++ and urbiscript ([Section 26.18](#)).
- `nil` is now correctly serialized to/from remote UObject.
- The C++ header ‘`urbi/revision.hh`’ contains version information that can be used to set requirements. For instance:

```
#include <urbi/revision.hh>
#if URBI_SDK_VERSION_VALUE < 2003000
# error Urbi SDK 2.3 or better is required.
#endif
```

C++

- `Date.year`, `Date.month`, `Date.day`, `Date.hour`, `Date.minute`, `Date.second`.
- The keys in Dictionary literals are no longer required to be literal strings.

```
["a" + "b" => "ab", 12.asString => "12"];
[00002405] ["12" => "12", "ab" => "ab"]
```

urbiscript
Session

They still need to evaluate into String values (no longer true since Urbi SDK 2.5).

```
[12 => "12"];
[00005064:error] !!! unexpected 12, expected a String
```

urbiscript
Session

31.10.3 Documentation

- `String.empty`, `String.length`.

31.11 Urbi SDK 2.3

Released on 2010-09-28.

2010-09-28.

31.11.1 Fixes

- `Date.asFloat` is restored.
- `File.create` empties existing files first.
- `Lobby.lobby` always returns the current lobby, even if invoked on another lobby.
- `Object.inspect` works properly, even if the target is a remote lobby.
- `Regexp.matchs` renamed as `Regexp.matches`.
- `System.version` Really returns the current version.
- Fix multiple race conditions in RTP handling code preventing proper initialization of remote UObjects.
- Fix Windows deployment to have both debug and release UObjects installed.
- Fix `urbi-sound` in the liburbi examples.
- Fix server mode of ‘`urbi-launch --remote`’.

31.11.2 New Features

- The documentation of Urbi SDK Remote, our middleware layer to communicate with an Urbi server — either by hand or via the UObject —, is included in the binary packages (in ‘`share/doc/urbi-sdk/sdk-remote.htmldir/index.html`’. It is also available on-line at <https://github.com/urbiforge/urbi/urbi/3.0.0/doc/sdk-remote.htmldir>.
- In addition to Gostai Console 2.5, Windows installers now include Gostai Editor 2.4.1.
- By popular demand, all the Boost libraries (1.38) are included in the binary packages. We used to provide only the headers and libraries Urbi SDK depends upon. Boost.Python, because it has too many dependencies, is not included.
- When launched with no input (i.e., none of the options ‘`-e`’/‘`--expression`’, ‘`-f`’/‘`--file`’, ‘`-P`’/‘`--port`’ were given), the interpreter is interactive.
- Assignment operators such as ‘`+=`’ are redefinable. See [Section 21.1.8.2](#).
- `Date.'-'` accepts a `Duration` or a `Float` in addition to accepting a `Date`.
- `Date.year`, `Date.month`, `Date.day`, `Date.hour`, `Date.minute`, `Date.second` slots allow partial modifications of `Date` objects.
- `Float.fresh` generates unique integers.
- `InputStream.close`.
- `List.'+='`.

- Support for `else` in `try` blocks (Section 21.8.2). Run only when the `try` block completed properly.

urbiscript
Session

```
// Can we run "riskyFeature"?
try { riskyFeature } catch { false } else { true };
[00004220] false

function riskyFeature() { throw "die" }|;
try { riskyFeature } catch { false } else { true };
[00004433] false

riskyFeature = function () { 42 }|;
try { riskyFeature } catch { false } else { true };
[00004447] true
```

- Support for `finally` in `try` blocks (Section 21.8.4). Use it for code that must be run whatever the control flow can be. For instance:

urbiscript
Session

```
try { echo(1) } catch { echo(2) } else { echo(3) } finally { echo(4) };
[00002670] *** 1
[00002670] *** 3
[00002670] *** 4

try { throw 1 } catch { echo(2) } else { echo(3) } finally { echo(4) };
[00002671] *** 2
[00002671] *** 4
```

- `System.eval` and `System.load` report syntax warnings.

urbiscript
Session

```
System.eval("new Object");
[00001388:warning] !!! 1.1-10: 'new Obj(x)' is deprecated, use 'Obj.new(x)'
[00001388:warning] !!!      called from: eval
[00001388] Object_0x1001b2320
```

- New functions `as` and `fill` on `UVar` to ease access to the generic cast system.
- Add support to `boost::unordered_map` to `UObject` casting system.
- Optimize remote `UObjects`: notifies between two objects in the same process instance are transmitted locally.
- Provide a `CustomUVar` class to ease encapsulation of custom data in `UVar`.
- Bind the `constant` property on `UVar`.

31.12 Urbi SDK 2.2

Released on 2010-08-23.

2010-08-23.

31.12.1 Fixes

- The main loop optimization triggered several issues with GNU/Linux and Mac OS X in interactive sessions (truncated output, possibly blocked input). These issues are fixed on Snow Leopard and Leopard and GNU/Linux.
- Deep overall of the event handling primitives. Watching an expression will succeed in cases where it used to fail. For instance:

urbiscript
Session

```

at (isdef (myVar))
echo("var myVar = " + myVar),
myVar;
[00000001:error] !!! lookup failed: myVar

var myVar = 42|;
[00000003] *** var myVar = 42

```

or

urbiscript
Session

```

var x = 0|;
function f() { x == 1 }|;
at (f()) echo("OK");
x = 1|;
[00000001] *** OK

```

Support for sustained events is fixed too (see [Section 21.11.1](#)).

- `List.max` and `List.min` will now report the right indexes in error messages.
- `onleave` blocks on lasting events are now run asynchronously, as expected.
- `at (expression ~ duration)` is now supported.
- Fix a bug if emitting an event triggers its unsubscription.
- Fix printing of `Exception.Type` and `Exception.ArgumentType`.
- Fix timestamp overflow on Windows after 40 minutes.
- Fix fatal error when manipulating the first `Job` prototype.

31.12.2 New Features

- Pressing C-c in the urbiscript shell ('`urbi -i`') interrupts the foreground job, and clears the pending commands. A second C-c in a row invokes `System.shutdown`, giving a chance to the system to shut down properly. A third C-c kills `urbi/urbi-launch`. See [Section 20.3.2](#) for more details.
- Closing the standard input (e.g., by pressing C-d) in interactive sessions shuts down the server.
- Remote UObjects now support the RTP protocol to exchange value with the engine ([Section 26.17](#)).
- NotifyChange/access callbacks can now take any type as argument. `UVar&` still has the previous behavior. For any other type, the system will try to convert the value within the `UVar` to this type.
- `CallMessage.eval`.
- `Float.ceil`, `Float.floor`, `Float.isInf`, `Float.isnan`.
- `Traceable`.
- Improved context (the call stacks) when errors are reported. Especially when using `System.eval` or `System.load`.

- `at` (`expression`) — as opposed to `at` (`event?`) — implementation has been improved: the condition will now be reevaluated even if a parameter not directly in the expression (in the body of a called function, for instance) is modified.
- `Regexp.matchs`.
- `Date` objects now have microsecond resolution and have been slightly revamped to not rely on Unix's epoch.
- UVars now have the timestamp of their latest assignment.

31.12.3 Documentation

- `Float.hex`.

31.13 Urbi SDK 2.1

Released on 2010-07-08.

2010-07-08.

31.13.1 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 20.10](#).
- 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.

31.13.2 New Features

- `Regexp.asPrintable`, `Regexp.asString`, `Regexp.has`.
- `System.Platform.host`, `System.Platform.hostAlias`, `System.Platform.hostCpu`, `System.Platform.hostOs`, `System.Platform.hostVendor`.
- UObject init method and methods bound by `notifyChange` no longer need to return an int.
- `Channel.Filter`, a `Channel` that outputs text that can be parsed without error by the liburbi.
- `RangeIterable.all`, `RangeIterable.any`, moved from `List`.
- Support for `ROS`, the Robot Operating System. See [Chapter 13](#) for an introduction, and [Chapter 23](#) for the details.
- `Lobby.lobby` and `Lobby.instances`, bounced to from `System.lobby` and `System.lobbies`.
- `Tag.scope`, bounced to from `System.scopeTag`.

31.13.3 Optimization

- The main loop was reorganized to factor all socket polling in a single place: latency of `Socket` is greatly reduced.

31.13.4 Documentation

- `Lobby.authors`, `Lobby.thanks`.
- `System.PackageInfo`.
- `System.spawn`.
- LEGO Mindstorms NXT support.
- Pioneer 3-DX support.
- Support for Segway RMP.

31.14 Urbi SDK 2.0.3

Released on 2010-05-28.



2010-05-28.

31.14.1 New Features

- `Container`, prototype for `Dictionary`, `List` derive.
- `e not in c` is mapped onto `c.hasNot(e)` instead of `!c.has(e)`.
- `Float.limits`
- `Job.asString`
- `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)
```

urbiscript
Session

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`: `assert_eq`, `assert_ge`, `assert_gt`, `assert_le`, `assert_lt`, `assert_meq`, `assert_mne`, `assert_ne`.

31.14.2 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` events were not launched if a `Directory` had already been created on the same `Path`.
- `waituntil` no longer ignores pattern guards.

31.14.3 Documentation

- Bioloid.
- Garbage collection ([Section 21.13](#)).
- Structural Pattern matching ([Section 21.6](#)).
- `CallMessage.sender` and `CallMessage.target`.
- `Dictionary.asString`.
- `Directory.fileCreated` and `Directory.fileDeleted`.
- `List.max`, `List.min`.
- `Mutex.asMutex`.
- `Object.localSlotNames`.

31.15 Urbi SDK 2.0.2

Released on 2010-05-06.



2010-05-06.

31.15.1 urbascript

- `Control.detach` and `Control.disown` return the `Job`.

31.15.2 Fixes

- ‘`make install`’ failures are addressed.
- `freezeif` can be used more than once inside a scope.

31.15.3 Documentation

- `StackFrame`
- `String.split`

31.16 Urbi SDK 2.0.1

Released on 2010-05-03.



2010-05-03.

31.16.1 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.

31.16.2 Documentation

- How to build Urbi SDK ([Chapter 19](#)).
- Hyperlinks to slots (e.g., `Float.asString`).

31.16.3 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.

31.17 Urbi SDK 2.0

Released on 2010-04-09.



2010-04-09.

31.17.1 urbiscript

31.17.1.1 Changes

- `Global.Tags` is renamed as `Tag.tags`.
- `Global.Task` is renamed as `Global.Job`.
- `Global.topLevel` is renamed as `Channel.topLevel`.
- `Global.output`, `Global.error` 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`.

31.17.1.2 New features

- Dictionary can now be created with literals.

Syntax	Semantics
<code>[=>]</code>	<code>Dictionary.new</code>
<code>["a" => 1, "b" => 2, "c" => 3]</code>	<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'

31.17.2 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.

31.17.3 Documentation

- `Barrier`
- `Date.now`
- `Float.srandom`
- `Pattern`
- `PubSub`
- `PubSub.Subscriber`
- `Profiling`
- `Semaphore`
- Trajectories
- `TrajectoryGenerator`
- `urbi-image` (Section 20.4).
- `waituntil` clauses
- `whenever` clauses

31.18 Urbi SDK 2.0 RC 4

Released on 2010-01-29.



31.18.1 urbiscript

31.18.1.1 Changes

- '\$id' replaces id
- List derives from Orderable.

31.18.1.2 New objects

- [Location](#)
- [Position](#)

31.18.1.3 New features

- [File.remove](#)
- [File.rename](#)

31.18.2 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.

31.19 Urbi SDK 2.0 RC 3

Released on 2010-01-13.



2010-01-13.

31.19.1 urbiscript

31.19.1.1 Fixes

- 'local.u' works as expected.

31.19.1.2 Changes

- [Lobby.quit](#) replaces [System.quit](#).
- [Socket.connect](#) accepts integers.
- UObject remote notifyChange on USensor variable now works as expected.
- UObject timers can now be removed with [UObject::removeTimer\(\)](#).

31.19.2 Documentation

- [Socket](#) provides a complete example.
- The Naming Standard documents the support classes provided to ease creation of the component hierarchy.



31.20 Urbi SDK 2.0 RC 2

Released on 2009-11-30.

This release candidate includes many fixes and improvements that are not reported below. 2009-11-30. The following list is by no means exhaustive.

31.20.1 Optimization

The urbiscript engine was considerably optimized in both space and time.

31.20.2 urbiscript

31.20.2.1 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:

urbiscript
Session

```
function binList(a, b,) { [a, b,] } | binList(1, 2,)
```

is equivalent to

```
function binList(a, b) { [a, b] } | binList(1, 2)
```

urbiscript
Session

- consecutive string literals are joined into a unique string literal, as in C++.

31.20.2.2 New objects

- `Component`, `Localizer`, `Interface`: naming standard infrastructure classes.
- `Date`
- `Directory`
- `File`
- `Finalizable`: objects that call `finalize()` when destroyed.
- `InputStream`
- `Mutex`
- `OutputStream`
- `Process`: Start and monitor child processes.
- `Regexp`
- `Server`: TCP/UDP server socket.
- `Socket`: TCP/UDP client socket.
- `Timeout`
- `WeakDictionary`, `WeakPointer`: Store dictionary of objects without increasing their reference count.

31.20.2.3 New features

- `Object.asBool`.
- `Lobby.wall`.
- `Dictionary.size`.
- `Global.evaluate`.
- `Group.each`, `Group.each&`
- `Lobby.onDisconnect`, `Lobby.remoteIP` `Lobby.create`.
- `Object.inspect`.
- `String.fromAscii`, `String.replace`, `String.toAscii`.
- System: `_exit`, `assert_eq`, `System.system`.

31.20.2.4 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.

31.20.2.5 Deprecations

- `Object.ownsSlot` is deprecated in favor of `Object.hasSlot`/`Object.hasLocalSlot`.
- `Object.slotNames` is deprecated in favor of `Object.allSlotNames`/`Object.localSlotNames`.

31.20.2.6 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.

31.20.3 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`.
- Bind 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");
```

urbiscript
Session

will now work. This hook costs one `at` per `UVar`; set `UVar.hookChanged` to false to disable it.

- Add a statistics-gathering tool. Enable it using `uobjects.enableStats`. 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 UObject.
- Many other bug fixes and performance improvements.
- an exception is now thrown if the C++ init method failed.

31.20.4 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 20](#)).
- environment variables: `URBI_UOBJECT_PATH`, `URBI_PATH`, `URBI_ROOT` ([Section 20.1](#)).
- special files '`global.u`', '`local.u`' ([Section 20.2](#)).
- k1-to-k2: Conversion idioms from urbiscript 1 to urbiscript 2 ([Chapter 17](#)).
- 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`
 - `Channel`
 - `Date`
 - `Dictionary`
 - `Exception`
 - `File`
 - `Kernel1`
 - `InputStream`
 - `Lazy`
 - `Math`
 - `Mutex`
 - `Regexp`
 - `Object`
 - `OutputStream`
 - `Pair`
 - `String`
 - `Tag`
 - `Timeout`
- tutorial:
 - uobjects

31.20.5 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.

31.21 Urbi SDK 2.0 RC 1

Released on 2009-04-03.



2009-04-03

31.21.1 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.

31.21.2 urbiscript

31.21.2.1 Changes

- The keyword `emit` is deprecated in favor of `!: instead of emit e(a);, write e!(a);`. The `? construct is changed for symmetry: instead of at (?e(var a)), write at (e?(var a))`. See [Section 17.3](#) for details. This syntax for sending and receiving is traditional and can be found in various programming languages.
- `System.Platform` enhances former `System.platform`. Use `System.Platform.kind` instead of `System.platform`.

31.21.2.2 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.

31.21.3 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.

31.21.4 Documentation

Extended documentation on: [Comparable](#), [Orderable](#).

31.22 Urbi SDK 2.0 beta 4

Released on 2009-03-03.

2009-03-03.

31.22.1 Documentation

An initial sketch of documentation (a tutorial, and the language and library specifications) is included.

31.22.2 urbiscript

31.22.2.1 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.

31.22.2.2 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: invalid number: "2.0a"
```

urbiscript
Session

31.22.3 Programs

31.22.3.1 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.

31.22.3.2 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"]
```

Shell
Session

31.22.3.3 urbi-console

`urbi-console` is now a simple wrapper around `urbi-launch`. Running

Shell
Session

```
urbi-console arg1 arg2...
```

is equivalent to running

```
urbi-launch --start -- arg1 arg2...
```

Shell
Session

31.22.3.4 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;'
```

urbiscript
Session

31.23 Urbi SDK 2.0 beta 3

Released on 2009-01-05.



31.23.1 Documentation

2009-01-05.

A new document, ‘FAQ.txt’, addresses the questions most frequently asked by our users during the beta-test period.

31.23.2 urbiscript

31.23.2.1 Fixes

- If a file loaded from ‘URBI.INI’ cannot be found, it is now properly reported.

31.23.2.2 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
```

urbiscript
Session

- `catch (type name)` syntax removed.

It was used to catch exceptions if and only if they inherited `type`. This behavior can be obtained with the more general guard system:

```
catch (var e if e.isA(<type>))
{
    ...
}
```

urbiscript
Session

- 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; integer literals up to $2^{63} - 1$ are accepted.

- 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:

urbiscript
Session

```
at (?myEvent(var x, var y, 1))
```

instead of:

urbiscript
Session

```
at (?myEvent((var x), (var y), 1))
```

- Waituntil and bindings.

Bindings performed in `waituntil` constructs are now available in its context:

urbiscript
Session

```
waituntil(?event(var x));
// x is available
echo (x);
```

- `List.insert`.

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"]
```

urbiscript
Session

- `List.sort`.

Now takes an optional argument, which is a function to call instead of the `<` 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.toLowerCase < y.toLowerCase});
[00000002] ["bar", "Baz", "foo"]
```

urbiscript
Session

- `System.searchPath`.

It is now possible to get the search path for files such as ‘urbi.u’ or ‘URBI.INI’ by using `System.searchPath`.

- `System.getenv`.

Now returns `nil` if a variable cannot be found in the environment instead of `void`. This allows you do to things such as:

urbiscript
Session

```
var ne = System.getenv("nonexistent");
[00000001:warning] !!! 'System.getenv(that)' is deprecated, use 'System.env[that]',
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") });
```

urbscript
Session

- **System.stats**.

Its output is now expressed in seconds rather than milliseconds, for consistency with the rest of the kernel.

31.23.3 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}();
```

urbscript
Session

is now safe.

31.23.4 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.

31.24 Urbi SDK 2.0 beta 2

Released on 2008-11-03.



31.24.1 urbscript

2008-11-03.

- `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:

urbscript
Session

```
var object = 1;
var from = 1;
```

- Hexadecimal literals.

It is now possible to enter (integral) hexadecimal numbers by prefixing them with `0x`, as in:

urbscript
Session

```
0x2a;
[00000001] 42
```

Only integral numbers are supported.

31.24.2 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
```

urbscript
Session

- `String.split` method Largely improved.

- `List.min` and `List.max`.

It is now possible to call `min` and `max` on a list. By default, the `<` comparison operator is used, but one explicit “less 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.toLowerCase < r.toLowerCase });
[00000002] "brown"
```

urbscript
Session

`Math.min` and `Math.max` taking an arbitrary number of arguments have also been defined. In this case, the default `<` operator is used for comparison:

```
Math.min(3, 2, 17);
[00000001] 2
```

urbscript
Session

- 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"
```

urbscript
Session

- Tag names.

Tags were displayed as `Tag_0x01234500` 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>
```

urbiscript
Session

- `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.

31.24.3 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).

31.24.4 Run-time

Stack exhaustion check on Windows

As was done on GNU/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
```

urbiscript
Session

31.24.5 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 `slotNames` 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:

urbiscript
Session

```
send(__name + ".val = 1;");
```

This has been fixed. `__name` contains a valid slot name of `uobjects`.

31.24.6 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 32

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.

32.1 Boost Software License 1.0

Urbi SDK uses Boost libraries. Boost libraries and headers are included in binary packages. See <http://www.boost.org/users/license.html>.

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

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, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

32.2 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;
- or the “BSD three-clause License,” see <http://www.opensource.org/licenses/BSD-3-Clause>.

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.

32.3 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.

32.4 gnu.bytecode

The Java API for UObject uses the [gnu.bytecode](#) Java package.

The software (with related files and documentation) in these packages are copyright (C) 1996-2006 Per Bothner.

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.

32.5 ICU License

Urbi SDK uses the Boost which in turn uses ICU. Boost and ICU are included in binary packages.

ICU License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2011 International Business Machines Corporation and others

All rights reserved.

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, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

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 OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

All trademarks and registered trademarks mentioned herein are the

property of their respective owners.

32.6 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.

In plain English:

1. We don't promise that this software works. (But if you find any bugs, please let us know!)
2. You can use this software for whatever you want. You don't have to pay us.
3. You may not pretend that you wrote this software. If you use it in a program, you must acknowledge somewhere in your documentation that you've used the IJG code.

In legalese:

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) 1991-2012, Thomas G. Lane, Guido Vollbeding.
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.

32.7 Libcoroutine License

Urbi uses the libcoroutine from the *Io* language. See <http://www.iolangue.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.

32.8 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.]
*/

```

32.9 ROS

The ROS bridge ([Chapter 13](#)) using ROS libraries, which are included in our binary packages. As of today (October 10th, 2011), the ROS guidelines for developers (<http://www.ros.org/wiki/DevelopersGuide#Licensing>) point to a BSD two-clause license (<http://www.opensource.org/licenses/bsd-license.php>). This is the same license as the three-clause BSD ([Section 32.2](#)), without the third-clause.

```

Copyright (c) 2010, Willow Garage
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.

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 COPYRIGHT
HOLDER 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.

32.10 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 33

Bibliography

- [1] Jean-Christophe Baillie. URBI: A universal language for robotic control. *International journal of Humanoid Robotics*, October 2004.
- [2] Jean-Christophe Baillie. URBI: Towards a universal robotic low-level programming language. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*, pages 820–825, 2005.
- [3] Jean-Christophe Baillie. Design principles for a universal robotic software platform and application to URBI. In Davide Brugali, Christian Schlegel, Issa A. Nesnas, William D. Smart, and Alexander Braendle, editors, *IEEE ICRA 2007 Workshop on Software Development and Integration in Robotics (SDIR-II)*, SDIR-II, Roma, Italy, April 2007. IEEE Robotics and Automation Society.
- [4] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Events! (reactivity in urbiscript). In Serge Stinckwich, editor, *First International Workshop on Domain-Specific Languages and models for ROBotic systems*, October 2010.
- [5] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Tag: Job control in urbiscript. In Noury Bouraqadi, editor, *Fifth National Conference on Control Architecture of Robots*, May 2010.
- [6] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, Matthieu Nottale, and Samuel Tardieu. The Urbi universal platform for robotics. In Itsuki Noda, editor, *First International Workshop on Standards and Common Platform for Robotics*, November 2008.

Chapter 34

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 provides a graphical user interface to a remote Urbi server (see [Figure 34.1](#)). 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 3](#).

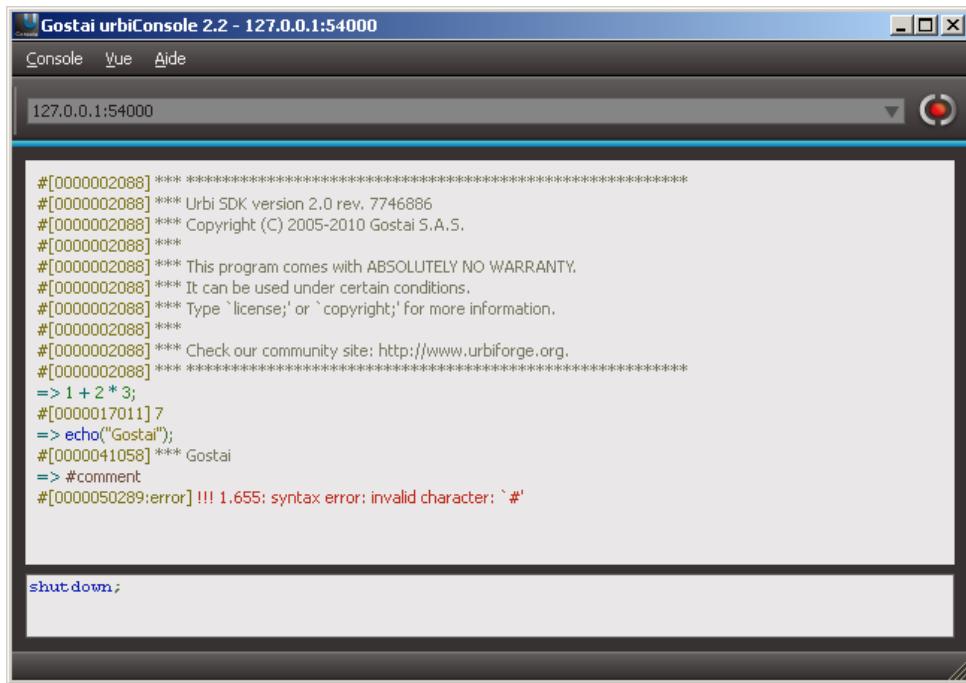


Figure 34.1: Gostai Console

Gostai Editor Also called UEdit, Gostai urbiscript Editor ([Figure 34.2](#)) is a lightweight urbiscript source code editor with semantic highlighting. It is available for Windows and GNU/Linux platforms.

Gostai Lab This tool (see [Figure 34.3](#)), 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.

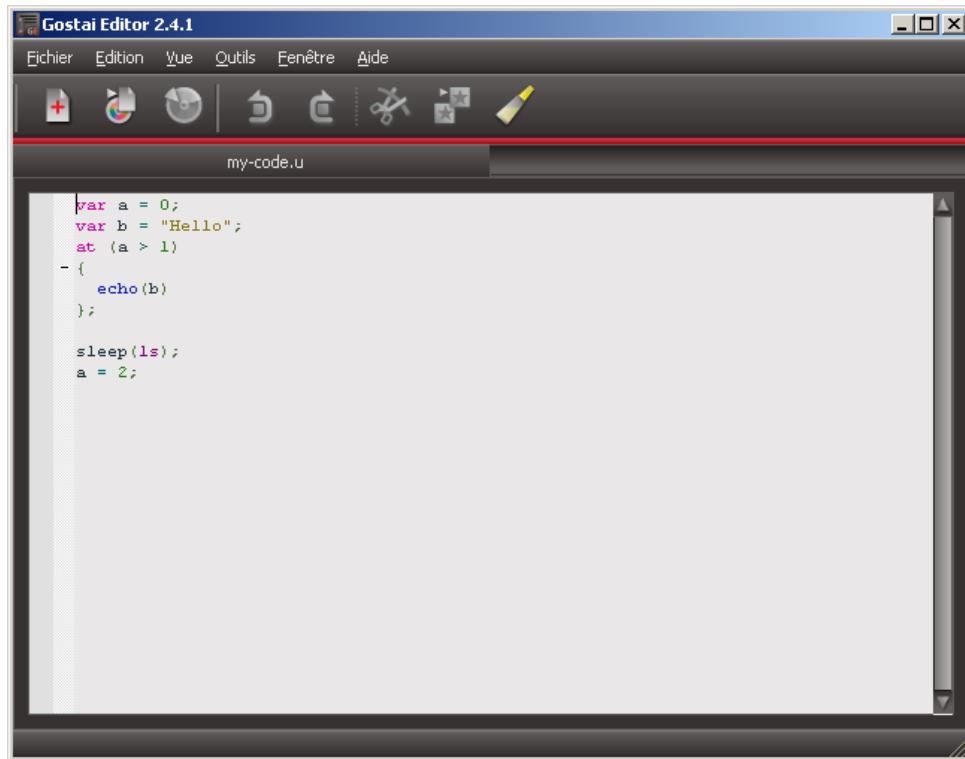


Figure 34.2: Gostai Editor

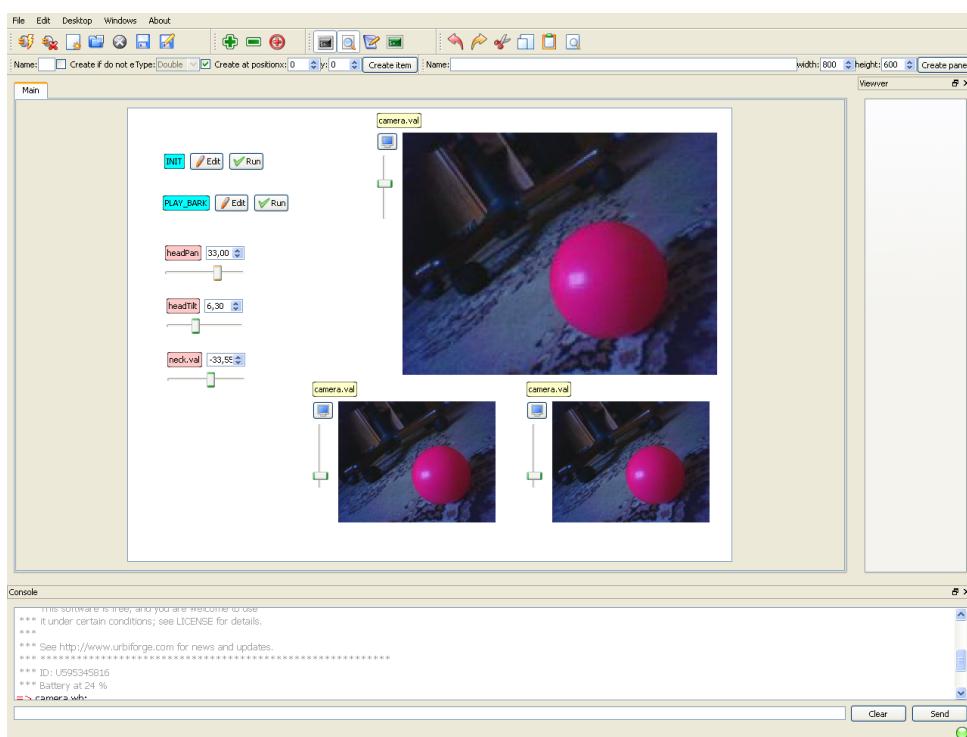


Figure 34.3: Gostai Lab

Gostai Studio This tool (see [Figure 34.4](#)), 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*.

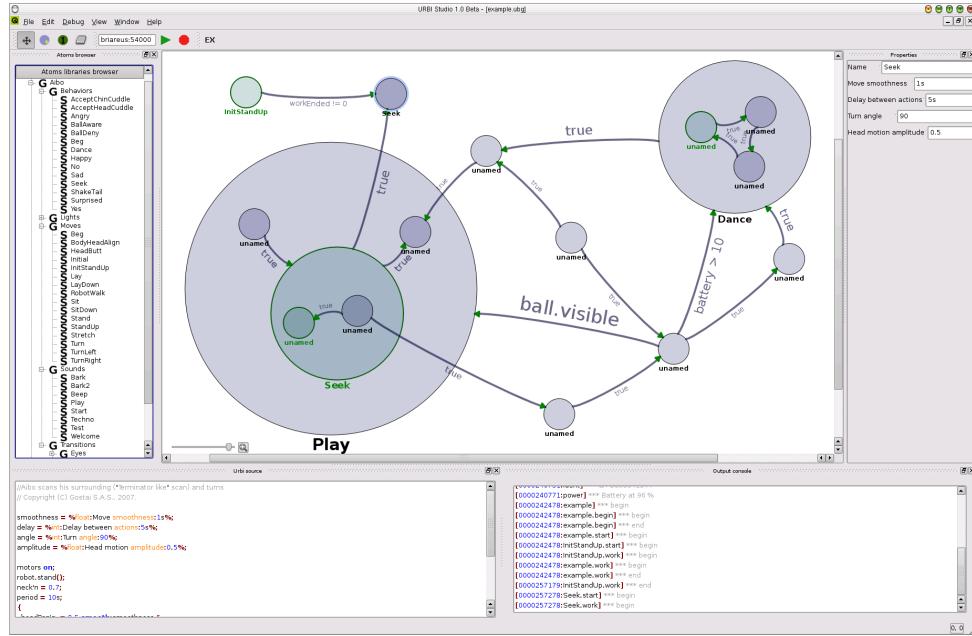


Figure 34.4: Gostai Studio

RMP The Segway Robotic Mobility Platform is a robotic platform based on the Segway Personal Transporter. See <http://rmp.segway.com>.

RTP Real-time Transport Protocol (RFC 3550). A protocol designed for streaming (for instance audio/video).

ROS Robot Operating System, <http://www.ros.org/>, developed by Willow Garage, <http://www.willowgarage.com/>. 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.

Chapter 35

List of Tables

21.1 Keywords	210
21.2 Angle units	210
21.3 Duration units	211
21.4 String escapes	211
21.5 Arithmetic operators	211
21.6 Assignment operators	211
21.7 Pre-/postfix operators	212
21.8 Bitwise operators	212
21.9 Boolean operators	212
21.10 Comparison operators	212
21.11 Container operators	212
21.12 Object operators	212
21.13 Operators sorted by decreasing precedence groups.	213
21.14 Keywords with flavors	214
22.1 Character handling functions	421

Chapter 36

List of Figures

1.1 A Bird-View of the Urbi Architecture	8
13.1 Output from <code>rxgraph</code>	87
34.1 Gostai Console	603
34.2 Gostai Editor	604
34.3 Gostai Lab	604
34.4 Gostai Studio	605

Chapter 37

Index

Symbols

'!',
 Boolean.~, 220
 Object.~, 355
'!=',
 Comparable.~, 232
'!==',
 Object.~, 355
'\$id',
 Object.~, 357
'&&',
 Object.~, 356
'*',
 Float.~, 273
 List.~, 314
 Matrix.~, 342
 String.~, 417
 Tuple.~, 456
 Vector.~, 462
'**',
 Float.~, 273
'*=',
 Matrix.~, 342
 Object.~, 356
'+',
 Binary.~, 218
 Date.~, 236
 Float.~, 273
 List.~, 315
 Matrix.~, 342
 Position.~, 379
 String.~, 417
 Tuple.~, 456
 Vector.~, 462
'+=',
 List.~, 315
 Matrix.~, 343
 Object.~, 356
'-',
 Date.~, 236
Float.~, 273
List.~, 315
Matrix.~, 343
Position.~, 343
Comparable.~, 232
Object.~, 355
Orderable.~, 369
Path.~, 373
Slot.~, 403
Vector.~, 463
Date.~, 237
Float.~, 273
List.~, 315
Matrix.~, 344
Path.~, 373
Vector.~, 463
Matrix.~, 344
Object.~, 356
Directory.~, 246
Float.~, 273
Matrix.~, 344
Path.~, 373
Vector.~, 463
Matrix.~, 344
Object.~, 356
Date.~, 237
Float.~, 273
List.~, 315
Orderable.~, 369
Path.~, 373
Position.~, 379
String.~, 417
Tuple.~, 456
Vector.~, 462
Channel.~, 226
Directory.~, 246
Event.~, 256
Float.~, 273
Group.~, 295
List.~, 316
Logger.~, 334
OutputStream.~, 370
Ros.Topic.~, 471
Slot.~, 403
Orderable.~, 369

```

'=='
    Binary.~, 218
    Code.~, 229
    Comparable.~, 232
    Date.~, 237
    Dictionary.~, 241
    Float.~, 273
    Group.~, 296
    Lazy.~, 312
    List.~, 316
    Location.~, 331
    Matrix.~, 344
    Object.~, 356
    Path.~, 373
    Position.~, 379
    String.~, 418
    Tuple.~, 457
    Vector.~, 464
'==='
    Object.~, 357
'>'
    Orderable.~, 369
'>='
    Orderable.~, 369
'>>'
    Float.~, 273
    Slot.~, 403
'[]'
    Dictionary.~, 241
    List.~, 316
    Matrix.~, 344
    Regexp.~, 395
    String.~, 418
    Tuple.~, 457
    Vector.~, 464
'[]='
    Dictionary.~, 241
    List.~, 316
    Matrix.~, 345
    String.~, 418
    Tuple.~, 457
    Vector.~, 464
',^'
    Float.~, 274
    Object.~, 357
'^='
    Object.~, 357
'assert'
    System.~, 426
'bitand'
    Float.~, 275
'bitor'
    Float.~, 275
'compl'
    Float.~, 275
'detach'
    Control.~, 234
'disown'
    Control.~, 234
'each&'
    Float.~, 276
    Group.~, 296
    List.~, 318
    RangeIterable.~, 393
'new'
    Singleton.~, 402
':=,'
    548
'_exit'
    System.~, 426
A
aborted
    Interface.Mobile.~, 482
abs
    Float.~, 274
    Math.~, 338
absolute
    Path.~, 373
Accel, 450
    TrajectoryGenerator.~, 453
accel, 490
acceleration
    Interface.AccelerationSensor.~, 484
AccelerationSensor, 484
acceptVoid
    Object.~, 357
    void.~, 468
accumulate, 319
acos
    Float.~, 274
    Math.~, 338
acquire
    Semaphore.~, 398
add
    Group.~, 296
addComponent
    Component.~, 492
addDevice
    Component.~, 492
addProto
    Object.~, 357
advertise
    Ros.Topic.~, 472
age
    Interface.TextToSpeech.~, 485
alignment

```

FormatInfo. \sim , 284
 all
 RangeIterable. \sim , 393
 allProto
 Object. \sim , 358
 allSlotNames
 Object. \sim , 358
 alt
 FormatInfo. \sim , 284
 and, 356
 angle, 149
 angle
 Interface.RotationalMotor. \sim , 483
 angleMax
 Interface.Laser. \sim , 484
 angleMin
 Interface.Laser. \sim , 484
 ankle, 489
 anonymous functions, 163
 any
 RangeIterable. \sim , 393
 append
 List. \sim , 317
 appendRow
 Matrix. \sim , 345
 apply
 Code. \sim , 230
 Object. \sim , 358
 Primitive. \sim , 381
 argMax
 List. \sim , 317
 argMin
 List. \sim , 317
 args
 CallMessage. \sim , 222
 argsCount
 CallMessage. \sim , 223
 Argument
 Exception. \sim , 261
 arguments
 System. \sim , 426
 ArgumentType
 Exception. \sim , 261
 Arity
 Exception. \sim , 261
 arity, 81
 arm, 488
 as
 Object. \sim , 358
 asBool
 Boolean. \sim , 220
 Dictionary. \sim , 241
 Float. \sim , 274
 List. \sim , 317
 Object. \sim , 358
 asEvent
 Event. \sim , 257
 asExecutable
 Executable. \sim , 265
 asFloat
 Date. \sim , 237
 Duration. \sim , 252
 Float. \sim , 274
 Hash. \sim , 298
 String. \sim , 418
 asin
 Float. \sim , 274
 Math. \sim , 338
 asJob
 Job. \sim , 304
 asList
 Dictionary. \sim , 241
 Directory. \sim , 247
 Enumeration. \sim , 253
 File. \sim , 266
 Float. \sim , 274
 Formatter. \sim , 287
 InputStream. \sim , 299
 List. \sim , 318
 Path. \sim , 374
 String. \sim , 419
 Vector. \sim , 464
 asMatrix
 Matrix. \sim , 346
 asMutex
 Mutex. \sim , 353
 asPath
 Directory. \sim , 247
 File. \sim , 266
 asPrimitive
 Primitive. \sim , 381
 asPrintable
 File. \sim , 266
 Matrix. \sim , 346
 Object. \sim , 358
 Path. \sim , 374
 Regexp. \sim , 395
 String. \sim , 419
 UValue. \sim , 460
 asProcess
 Process. \sim , 383
 assert, 188
 assert_
 System. \sim , 426
 assert_op

System.~, 426
assertion, 188
assertion frame, 549
assignment, 156
associative array, 240
asString
 Binary.~, 218
 Code.~, 230
 Date.~, 237
 Dictionary.~, 242
 Directory.~, 247
 Duration.~, 252
 Enumeration.~, 254
 File.~, 266
 Float.~, 274
 Group.~, 296
 Job.~, 304
 Lazy.~, 313
 List.~, 318
 Location.~, 331
 Matrix.~, 346
 Object.~, 359
 Path.~, 374
 Position.~, 380
 Process.~, 383
 Regexp.~, 396
 StackFrame.~, 414
 String.~, 419
 Tuple.~, 457
 UValue.~, 460
 Vector.~, 464
asTimeout
 Timeout.~, 448
asTopLevelPrintable
 Matrix.~, 346
 Object.~, 359
 UValue.~, 460
asTree
 Dictionary.~, 242
 List.~, 316
asuobjects
 uobjects.~, 459
asUValue
 UValue.~, 460
asVector
 Vector.~, 464
asynchronous, 255
asynchronous
 Subscription.~, 425
at, 197
atan
 Float.~, 274
 Math.~, 338
atan2
 Math.~, 338
attribute, 207
attributes, 500
AudioIn, 479
AudioOut, 479
authors
 Lobby.~, 327

B

b
 Interface.RGBLed.~, 481
 ‘-b’, 140
back
 List.~, 318
backtrace
 Exception.~, 260
 Job.~, 304
 Traceable.~, 449
BadInteger
 Exception.~, 261
BadNumber
 Exception.~, 262
banner
 Lobby.~, 327
Barrier, 216
 Global.~, 289
basename
 Directory.~, 247
 File.~, 267
 Path.~, 374
Battery, 480
begin
 Location.~, 332
 Tag.~, 445
Binary, 218
 Global.~, 289
Binding
 Pattern.~, 377
binding, 500
bindings
 Pattern.~, 377
blob, 218, 480
BlobDetector, 480
block, 440
 catch-block, 184
 try-block, 184
block
 Tag.~, 445
blocked
 Tag.~, 445
blocking, 505
body, 488

bodyString
 Code.~, 230
Boolean, 220
bounce
 Object.~, 360
break, 176
buffer, 370
 '--buffer=*size*', 140
bugReport
 System.PackageInfo.~, 436
bytesReceived
 Lobby.~, 327
bytesSent
 Lobby.~, 327

C

'-C', 141, 145
 '-c', 139, 141, 142, 144
caching, 311
call, 166
call
 Java, 526
callCount
 Subscription.~, 425
caller, 224
CallMessage, 222
 Global.~, 289
callMessage
 Object.~, 360
calls
 Profile.Function.~, 388
 Profile.~, 387
camera, 538
camera, 490
capacity
 Interface.Battery.~, 480
captured, 167
case
 switch, 182
catch
 timeout, 194
 try, 184
categories
 Logger.~, 334
cd
 Path.~, 374
ceil
 Float.~, 275
 Math.~, 338
cerr
 Global.~, 289
changed
 Slot.~, 404

Channel, 226
 Global.~, 289
 '--check', 141
checkMaster
 Ros.~, 469
class, 152, 355
 '--classpath=*path*', 141
clavicle, 490
 '--clean', 144
clear
 Dictionary.~, 242
 Directory.~, 247
 List.~, 318
clearStats
 uobjects.~, 459
 'CLIENT.INI', 135
clog
 Global.~, 289
clone
 Float.~, 275
 Job.~, 306
 Object.~, 360
 Singleton.~, 402
cloneSlot
 Object.~, 360
close
 Stream.~, 416
closed
 Stream.~, 416
ClosedLoop
 TrajectoryGenerator.~, 454
closest
 String.~, 419
closure, 56
closure, 163, 167
closures, 167
Code, 229
 Global.~, 289
code
 CallMessage.~, 223
column
 Matrix.~, 346
 Position.~, 380
columns
 Position.~, 380
combAdd
 Vector.~, 465
combDiv
 Vector.~, 465
combine
 Hash.~, 298
combMul
 Vector.~, 465

combSub
 Vector. \sim , 465
commands
 Kernel1. \sim , 309
comment, 37, 147, 547
Comparable, 232
 Global. \sim , 289
Component, 492
component, 475
components
 System.PackageInfo. \sim , 436
connect
 Socket. \sim , 412
connected
 Lobby. \sim , 328
 Socket. \sim , 412
connection
 Server. \sim , 401
connections
 Kernel1. \sim , 309
connectionStats
 uobjects. \sim , 459
connectionTag
 Lobby. \sim , 328
connectSerial
 Socket. \sim , 412
constant
 Slot. \sim , 404
Constness
 Exception. \sim , 262
constructor, 109
Container, 233
content
 Directory. \sim , 247
 File. \sim , 267
 InputStream. \sim , 299
continue, 176
Control, 234
Control.detach, 74
copy
 Directory. \sim , 247
 File. \sim , 267
 Kernel1. \sim , 309
 Slot. \sim , 405
copy on write, 172
copy-on-write, 60, 367
copyInto
 Directory. \sim , 247
 File. \sim , 267
copyOnWrite
 Slot. \sim , 405
copyright
 Lobby. \sim , 328

copyrightHolder
 System.PackageInfo. \sim , 436
copyrightYears
 System.PackageInfo. \sim , 436
copySlot
 Object. \sim , 360
‘--core=*core*’, 145
Cos, 450
cos
 Float. \sim , 275
 Math. \sim , 338
‘--count=*count*’, 142
cout
 Global. \sim , 289
CPPFLAGS, 144
create
 Directory. \sim , 248
 File. \sim , 267
 Lobby. \sim , 328
createAll
 Directory. \sim , 248
createIdentity
 Matrix. \sim , 346
createOnes
 Matrix. \sim , 346
createScalars
 Matrix. \sim , 346
createSlot
 Object. \sim , 360
createZeros
 Matrix. \sim , 347
criticalSection
 Semaphore. \sim , 398
current
 Interface.Battery. \sim , 480
 Job. \sim , 306
currentRunner
 System. \sim , 426
‘--customize=*file*’, 139
cwd
 Path. \sim , 374
CXXFLAGS, 144
cycle
 System. \sim , 426

D

‘-D’, 143
 ‘-d’, 135, 138–141, 143
 ‘-Dsymbol’, 145
data
 Binary. \sim , 218
Date, 236
 Global. \sim , 290

date
 System.PackageInfo.~, 436
day
 Date.~, 237
 System.PackageInfo.~, 436
debug
 Logger.~, 334
 '--debug', 144
 '--debug=level', 135, 139, 141
deep copy, 516
 '--deep-clean', 144
default
 switch, 182
delegate, 167
delete, 104
 '--describe', 140
 '--describe-file=file', 140
description
 System.PackageInfo.~, 436
destructor, 270
detach
 Global.~, 290
device, 476
 '--device=device', 138, 143
devices
 Kernel1.~, 309
DGain
 Interface.Motor.~, 482
dictionaries, 149
Dictionary, 240
 Global.~, 290
dictionary, 240
digits
 Float.limits.~, 281
digits10
 Float.limits.~, 281
Directory, 246, 375
Directory, 246
 Global.~, 290
dirname
 Path.~, 375
disable
 Logger.~, 334
 '--disable-automain', 145
disconnect
 Socket.~, 412
 Subscription.~, 425
 '--disconnect', 140
disconnected
 Socket.~, 412
disown
 Global.~, 290
distance
 Interface.DistanceSensor.~, 484
 String.~, 419
 Vector.~, 465
distanceMatrix
 Matrix.~, 347
distanceMax
 Interface.Laser.~, 484
distanceMin
 Interface.Laser.~, 484
DistanceSensor, 484
do, 177
done
 Process.~, 383
dump
 Component.~, 492
 Logger.~, 335
 Object.~, 360
dumpState
 Job.~, 306
Duplicate
 Exception.~, 262
Duration, 252
 Global.~, 290
duration, 149
duration
 Interface.AudioIn.~, 479
 '--duration=duration', 143

E

'-e', 136, 140, 142
each
 Float.~, 275
 Group.~, 296
 List.~, 318
 RangeIterable.~, 393
eachI
 List.~, 318
ear, 489
EBNF, 548
echo, 38
 Channel.~, 226
 Global.~, 290
 Lobby.~, 329
echoEach
 Lobby.~, 329
elbow, 488
elementAdded
 Dictionary.~, 242
elementChanged
 Dictionary.~, 242
elementRemoved
 Dictionary.~, 242
elongation

Interface.BlobDetector.~, 480
else
 if, 180
 timeout, 194
 try, 185
 whenever, 205
else-clause, 180
emit
 Event.~, 257
empty
 Binary.~, 219
 Dictionary.~, 243
 Directory.~, 248
 List.~, 318
 String.~, 420
enable
 Logger.~, 335
enabled
 Channel.~, 226
 Subscription.~, 425
enableStats
 objects.~, 459
encoding, 147
end
 Location.~, 332
 Tag.~, 446
 Timeout.~, 448
engine, 3
enter
 Tag.~, 446
Enumeration, 253
env
 System.~, 427
env.init
 System.~, 427
environment variable, 547
epoch
 Date.~, 237
epsilon
 Float.limits.~, 281
erase
 Dictionary.~, 243
err
 Logger.~, 335
error
 Socket.~, 412
eval
 CallMessage.~, 223
 Lazy.~, 313
 PseudoLazy.~, 390
 System.~, 427
evalArgAt
 CallMessage.~, 223
evalArgs
 CallMessage.~, 224
evaluate
 Global.~, 290
Event, 255
 Global.~, 290
event, 111, 255
events
 Kernel1.~, 309
every, 191
 every,, 192
 every;,, 191
 every|,, 192
Exception, 260
 Global.~, 290
exception
 catching, 184
 throwing, 184
exclusive or, 274, 357
Executable, 265
 Global.~, 290
exists
 Directory.~, 248
 Path.~, 375
exp
 Float.~, 276
 Math.~, 339
exposure
 Interface.VideoIn.~, 486
‘*--expression=exp*’, 136, 140
‘*--expression=script*’, 142
Extended Backus-Naur Form, 548
external
 Global.~, 291
EXTRA_CPPFLAGS, 144
EXTRA_CXXFLAGS, 144
EXTRA_LDFLAGS, 144
extract
 UValue.~, 460
extractAsToplevelPrintable
 UValue.~, 460
eye, 490
eyebrow, 490

F

‘-F’, 135, 138
‘-f’, 136, 140, 142

fallback
 Group.~, 296

false, 220
 Global.~, 291

‘*--fast*’, 135

File, 375

```

File, 266
    Global.~, 291
file
    Position.~, 380
file name, 547
'--file=file', 136, 140, 142
fileCreated
    Directory.~, 248
fileDeleted
    Directory.~, 249
FileNotFoundException
    Exception.~, 262
Filter
    Channel.~, 227
filter
    List.~, 319
Finalizable, 270
    Global.~, 291
finalize
    Finalizable.~, 271
finally
    timeout, 194
    try, 186
find
    String.~, 420
finger, 489
finished
    Interface.Mobile.~, 482
fireRate
    Subscription.~, 425
first
    Pair.~, 372
    Triplet.~, 455
flatDump
    Component.~, 492
Float, 272
    Global.~, 291
Float.limits, 281
floor
    Float.~, 276
    Math.~, 339
flush, 370
flush
    OutputStream.~, 370
foldl
    List.~, 319
foot, 489
for, 177
    for& (c), 194
    for& (var i: c), 193
    for, (;;), 192
    for; (;;), 178
    for; (c), 180
        for; (var i: c), 179
        for| (;;), 179
        for| (c), 180
        for| (var i: c), 180
force
    Interface.LinearMotor.~, 483
foreach, 178
foreach loops, 179
format
    Float.~, 276
    Interface.VideoIn.~, 486
format info, 283
'--format=format', 138
FormatInfo, 283
    Global.~, 291
Formatter, 287
    Global.~, 291
formatter, 287
freeze, 441
freeze
    Tag.~, 446
fresh
    Float.~, 276
    String.~, 420
fromAscii
    String.~, 420
front
    List.~, 319
frozen
    Job.~, 307
    Tag.~, 446
Function
    Profile.~, 387
function, 163
    lazy, 166
    return value, 165
    strict, 166
function, 163
function closures, 167
functions
    Kernel1.~, 309
G
g
    Interface.RGBLed.~, 481
gain
    Interface.AudioIn.~, 479
    Interface.VideoIn.~, 486
garbage collection, 117
GD_CATEGORY, 134
GD_COLOR, 134
GD_DISABLE_CATEGORY, 134
GD_ENABLE_CATEGORY, 134

```

GD_LEVEL, 134
 GD_LOC, 134
 GD_NO_COLOR, 134
 GD_PID, 134
 GD_THREAD, 134
 GD_TIME, 134
 GD_TIMESTAMP_US, 134
 gender
 Interface.TextToSpeech.~, 485
 get
 Dictionary.~, 243
 InputStream.~, 300
 Slot.~, 405
 getAll
 PubSub.Subscriber.~, 392
 Subscription.~, 425
 getChar
 InputStream.~, 300
 getenv
 System.~, 428
 getIoService
 Server.~, 401
 Socket.~, 412
 getLine
 InputStream.~, 300
 getLocale
 System.~, 429
 getLocalSlot
 Object.~, 361
 getOne
 PubSub.Subscriber.~, 392
 getPeriod
 Object.~, 361
 getProperty
 Global.~, 291
 Group.~, 296
 Object.~, 361
 getSlot
 Object.~, 362
 getStats
 uobjects.~, 459
 getter function, 512
 getWithDefault
 Dictionary.~, 244
 Global, 289
 Global, 289
 Global.~, 291
 'global.u', 135
 go
 Interface.Mobile.~, 481
 goTo
 Interface.Mobile.~, 481
 goToAbsolute

Interface.Mobile.~, 481
 goToChargingStation
 Interface.Mobile.~, 481
 grammar fragments, 548
 grip, 489
 Group, 295
 Global.~, 291
 group
 FormatInfo.~, 285
 guard, 175
 gyro, 490
 GyroSensor, 484

H

‘-H’, 136, 138, 141–143, 145
 ‘-h’, 135, 138–144
 hand, 488
 has
 Container.~, 233
 Dictionary.~, 244
 Enumeration.~, 254
 List.~, 319
 Regexp.~, 396
 Hash, 298
 hash, 240, 298
 hash
 Float.~, 277
 List.~, 319
 Object.~, 362
 String.~, 420
 Tuple.~, 457
 hasLocalSlot
 Object.~, 362
 hasNot
 Container.~, 233
 hasProperty
 Group.~, 297
 Object.~, 362
 hasSame
 List.~, 320
 hasSlot
 Group.~, 297
 Object.~, 362
 hasSubscribers
 Event.~, 257
 head, 489
 List.~, 320
 hear
 Interface.SpeechRecognizer.~, 485
 height
 Interface.VideoIn.~, 486
 '--help', 135, 138–144
 hex

Float. \sim , 277
 hideSystemFiles
 Traceable. \sim , 449
 Hierarchical Finite State Machines, 605
 hip, 489
 host
 Server. \sim , 401
 Socket. \sim , 412
 System.Platform. \sim , 439
 '--host=*address*', 136
 '--host=*host*', 138, 141–143, 145
 hostAlias
 System.Platform. \sim , 439
 hostName
 System.Platform. \sim , 439
 hostOs
 System.Platform. \sim , 439
 hour
 Date. \sim , 238

I

'-i', 136, 142
 '-Ipath', 145
 id
 System.PackageInfo. \sim , 436
 identifier, 148
 Identity, 480
 if, 180
 IGain
 Interface.Motor. \sim , 482
 IMAGE, 516
 image, 516
 immutable, 156
 implicit tags, 190
 in, 161
 in
 for, 179
 inf
 Float. \sim , 277
 Math. \sim , 339
 init, 515
 Dictionary. \sim , 244
 Logger. \sim , 335
 Matrix. \sim , 347
 initial value, 207, 450
 initialized
 Ros.Service. \sim , 474
 InputPort, 513
 InputStream, 299
 Global. \sim , 291
 insert
 List. \sim , 320
 insertBack

 List. \sim , 320
 insertFront
 List. \sim , 320
 insertUnique
 List. \sim , 320
 inspect
 Object. \sim , 363
 instances
 Lobby. \sim , 329
 '--interactive', 136
 Interface, 492
 interface, 479
 Interface.AccelerationSensor, 484
 Interface.AudioIn, 479
 Interface.AudioOut, 479
 Interface.Battery, 480
 Interface.BlobDetector, 480
 Interface.DistanceSensor, 484
 Interface.GyroSensor, 484
 Interface.Identity, 480
 Interface.Laser, 484
 Interface.Led, 481
 Interface.LinearMotor, 483
 Interface.LinearSpeedMotor, 483
 Interface.Mobile, 481
 Interface.Motor, 482
 Interface.Network, 483
 Interface.RGBLed, 481
 Interface.RotationalMotor, 483
 Interface.RotationalSpeedMotor, 483
 Interface.Sensor, 483
 Interface.SpeechRecognizer, 485
 Interface.TemperatureSensor, 484
 Interface.TextToSpeech, 485
 Interface.TouchSensor, 485
 Interface.Tracker, 486
 Interface.VideoIn, 486
 interruptible
 Job. \sim , 307
 '--interval=*interval*', 142
 invalidate
 UValue. \sim , 460
 inverse
 Matrix. \sim , 347
 Io, 594
 IoService, 302
 IoService, 302
 IP
 Interface.Network. \sim , 483
 isA
 Object. \sim , 363
 isAlnum
 String. \sim , 420

i
 isAlpha
 String.~, 420
 isAscii
 String.~, 421
 isBlank
 String.~, 421
 isCntrl
 String.~, 421
 isConnected
 Socket.~, 412
 isdef
 Global.~, 291
 isDigit
 String.~, 421
 isDir
 Path.~, 375
 isGraph
 String.~, 422
 isInf
 Float.~, 277
 isLower
 String.~, 422
 isNaN
 Float.~, 277
 isNil
 nil.~, 354
 Object.~, 363
 isPrint
 String.~, 422
 isProto
 Object.~, 363
 isPunct
 String.~, 422
 isReg
 Path.~, 375
 isSpace
 String.~, 422
 isUpper
 String.~, 422
 isVoid
 nil.~, 354
 Object.~, 363
 void.~, 468
 isvoid
 Kernel1.~, 309
 isWindows
 System.Platform.~, 439
 isXdigit
 String.~, 422

J
 '-j', 138, 144
 Job, 304

Global.~, 292
 jobs
 Job.~, 306
 '--jobs=jobs', 144
 join
 List.~, 320
 Process.~, 383
 String.~, 422
 joint, 489
 '--jpeg=factor', 138

K
 '-k', 145
 kernel, 3
 Kernel1, 309
 Global.~, 292
 '--kernel=dir', 145
 keys
 Dictionary.~, 244
 List.~, 321
 keyword, 149, 547
 keywords
 Binary.~, 219
 kill
 Process.~, 383
 kind
 Interface.Identity.~, 480
 System.Platform.~, 439
 knee, 489

L
 '-l', 144
 '-llib', 145
 '-Lpath', 145
 lang
 Interface.SpeechRecognizer.~, 485
 Interface.TextToSpeech.~, 485
 Laser, 484
 lastModifiedDate
 Directory.~, 249
 File.~, 268
 Path.~, 375
 launch
 Timeout.~, 448
 Lazy, 311
 Global.~, 292
 lazy, 311
 LC_ALL, 432
 LC_COLLATE, 432
 LC_CTYPE, 432
 LC_MESSAGES, 432
 LC_MONETARY, 432
 LC_NUMERIC, 432

LC_TIME, 432
 LDFLAGS, 144
 leave
 Tag.~, 446
 Led, 481
 leg, 488
 length
 String.~, 423
 level
 Logger.~, 335
 Levels
 Logger.~, 336
 lexical closures, 167
 '--library', 144
 license
 Lobby.~, 329
 limits
 Float.~, 277
 line
 Position.~, 380
 LinearMotor, 483
 LinearSpeedMotor, 483
 lines
 Position.~, 380
 lip, 490
 List, 314
 Global.~, 292
 list, 112, 151
 listen
 Server.~, 401
 literal, 37
 load
 Loadable.~, 325
 System.~, 429
 Loadable, 325
 Global.~, 292
 loadFile
 System.~, 429
 loadLibrary
 System.~, 429
 loadModule
 System.~, 429
 lobbies
 System.~, 429
 Lobby, 327
 Lobby, 327
 Global.~, 292
 lobby, 106, 327
 lobby
 Lobby.~, 329
 System.~, 429
 'local.u', 135
 locale, 432
 localHost
 Socket.~, 412
 Localizer, 493
 localPort
 Socket.~, 412
 localSlotNames
 Object.~, 363
 locateSlot
 Object.~, 363
 Location, 331
 location
 Exception.~, 261
 StackFrame.~, 414
 LOCK_CLASS, 512
 LOCK_FUNCTION, 511
 LOCK_FUNCTION_DROP, 512
 LOCK_FUNCTION_KEEP_ONE, 512
 LOCK_INSTANCE, 512
 LOCK_MODULE, 512
 LOCK_NONE, 511
 log
 Float.~, 277
 Logger.~, 336
 Math.~, 339
 Logger, 333
 Lookup
 Exception.~, 262
 loop
 closed-loop, 450, 454
 open-loop, 450, 454
 loop, 181
 loop,, 194
 loop;, 181
 loop|, 181
 loopn, 555
M
 '-m', 136, 140, 142, 145
 major
 System.PackageInfo.~, 437
 makeCompactNames
 Component.~, 492
 makeServer
 IoService.~, 302
 makeSocket
 IoService.~, 302
 manifest value, 37
 map
 List.~, 321
 match
 Pattern.~, 377
 Regexp.~, 396
 matchAgainst

Dictionary. \sim , 244
 List. \sim , 321
 Tuple. \sim , 457
 matches
 Regexp. \sim , 396
 MatchFailure
 Exception. \sim , 262
 matchPattern
 Pattern. \sim , 377
 Math, 338
 Global. \sim , 292
 Matrix, 341
 max
 Float.limits. \sim , 281
 Float. \sim , 277
 List. \sim , 321
 Math. \sim , 339
 maxCallTime
 Subscription. \sim , 425
 maxExponent
 Float.limits. \sim , 281
 maxExponent10
 Float.limits. \sim , 281
 maxFunctionCallDepth
 Profile. \sim , 387
 maybeLoad
 System. \sim , 429
 meanCallTime
 Subscription. \sim , 425
 memoization, 311
 message, 40, 172
 call, 166
 message
 CallMessage. \sim , 224
 Exception. \sim , 261
 meta-variable, 547
 methods, 500
 methodToFunction
 Global. \sim , 292
 micro, 491
 microsecond
 Date. \sim , 238
 min
 Float.limits. \sim , 281
 Float. \sim , 278
 List. \sim , 322
 Math. \sim , 339
 minCallTime
 Subscription. \sim , 425
 minExponent
 Float.limits. \sim , 281
 minExponent10
 Float.limits. \sim , 282

minInterval
 Subscription. \sim , 425
 minor
 System.PackageInfo. \sim , 437
 minute
 Date. \sim , 238
 Mobile, 481
 model
 Interface.Identity. \sim , 480
 '--module=module', 136, 140, 142
 month
 Date. \sim , 238
 Motor, 482
 mouth, 489
 moveInto
 Directory. \sim , 249
 File. \sim , 268
 moving
 Interface.Mobile. \sim , 482
 mutable, 156
 Mutex, 352
 Mutex, 352
 Global. \sim , 292

N

'-n', 143
 name
 Channel. \sim , 227
 Enumeration. \sim , 254
 Interface.Identity. \sim , 480
 Job. \sim , 307
 Process. \sim , 383
 Profile.Function. \sim , 388
 Ros.Service. \sim , 474
 Ros.Topic. \sim , 471
 Ros. \sim , 469
 StackFrame. \sim , 415
 System.PackageInfo. \sim , 437

nan
 Float. \sim , 278
 Math. \sim , 339
 ndebug
 System. \sim , 430
 neck, 489
 NegativeNumber
 Exception. \sim , 262
 Network, 483
 nil, 354
 Global. \sim , 292
 '--no-header', 143
 '--no-sync-client', 140
 nodes
 Ros. \sim , 469

NonPositiveNumber
 Exception. \sim , 262
noop
 Kernel1. \sim , 309
norm
 Vector. \sim , 465
not in, 161
notations, 547
notifyAccess
 Slot. \sim , 407
notifyChange
 Slot. \sim , 407
notifyChangeOwned
 Slot. \sim , 407
now
 Date. \sim , 238
null
 Channel. \sim , 227

O

‘-o’, 138, 143, 144
Object, 289
Object, 355
 Global. \sim , 292
object, 43, 169
off
 Loadable. \sim , 325
oget
 Slot. \sim , 406
on
 Loadable. \sim , 326
onConnect
 Ros.Topic. \sim , 472
onDisconnect
 Lobby. \sim , 329
 Ros.Topic. \sim , 472
onEnter
 Logger. \sim , 336
onEvent
 Event. \sim , 258
onLeave
 Logger. \sim , 336
onleave
 at, 197
onMessage
 Ros.Topic. \sim , 471
onSubscribe
 Event. \sim , 258
open
 Path. \sim , 375
OpenLoop
 TrajectoryGenerator. \sim , 454
Operating System, 3

operator, 155
 arithmetics, 156
 bitwise, 158
 Boolean, 159
 comparison, 160
 subscript, 161
or, 357
Orderable, 369
 Global. \sim , 292
orientation
 Interface.BlobDetector. \sim , 480
oset
 Slot. \sim , 407
 ‘--output=*file*’, 138, 143, 144
OutputStream, 370
 Global. \sim , 293
outputValue
 Slot. \sim , 407
owned
 Slot. \sim , 408

P

p
 Semaphore. \sim , 399
 ‘-P’, 136, 138, 141–143, 145
 ‘-p’, 138, 139, 145
 ‘--package=*pkg*’, 145
PackageInfo
 System. \sim , 430
pad
 FormatInfo. \sim , 285
Pair, 372
 Global. \sim , 293
pair, 372
 ‘--param-mk=*file*’, 145
parent
 Directory. \sim , 250
patch
 System.PackageInfo. \sim , 437
Path, 373
Path, 373
 Global. \sim , 293
Pattern, 377
 Global. \sim , 293
pattern
 FormatInfo. \sim , 285
 Pattern. \sim , 378
pattern matching, 173
payload, 80
period, 430
period
 System. \sim , 430
 ‘--period=*period*’, 138

persist
 Control.~, 234
 Global.~, 293
PGain
 Interface.Motor.~, 482
pi
 Float.~, 278
 Math.~, 339
 piece of code, 547
ping
 Kernel1.~, 309
pitch, 489
 Interface.TextToSpeech.~, 485
 Interface.Tracker.~, 486
Platform
 System.~, 430
playing
 Interface.AudioOut.~, 479
 plugin mode, 112
 '--plugin', 139
poll
 IoService.~, 303
 Socket.~, 412
pollFor
 IoService.~, 303
pollInterval
 Socket.~, 412
pollOneFor
 IoService.~, 303
port
 Server.~, 401
 Socket.~, 413
 '--port-file=*file*', 136, 138, 141–143
 '--port=*port*', 136, 138, 141–143
Position, 379
 Global.~, 293
position
 Interface.LinearMotor.~, 483
 Interface.Mobile.~, 481
precision
 FormatInfo.~, 285
prefix
 FormatInfo.~, 285
 '--prefix=*dir*', 145
pressure
 Interface.TouchSensor.~, 485
Primitive, 381
 Exception.~, 262
 Global.~, 293
print
 Object.~, 363
 '--print-root', 135, 139, 141
Process, 382
 Global.~, 293
Profile, 385
 Global.~, 293
profile
 System.~, 430
Profile.Function, 388
programName
 System.~, 430
prompt, 549
properties, 63
properties
 Object.~, 364
protos, 110
 Object.~, 364
prototype, 60, 170
PseudoLazy, 390
 Global.~, 293
publish
 PubSub.~, 391
 Ros.Topic.~, 472
PubSub, 391
 Global.~, 293
PubSub.Subscriber, 392
put
 OutputStream.~, 370
 UValue.~, 460

Q

'-Q', 142
 '-q', 136, 144
quality
 Interface.VideoIn.~, 486
 '--quiet', 136, 144
quit
 Lobby.~, 329
 '--quit', 142
quote
 Channel.~, 227

R

r
 Interface.RGBLed.~, 481
 '-R', 138
 '-r', 138, 139
radix
 Float.limits.~, 282
random
 Float.~, 278
 Math.~, 339
range
 List.~, 322
RangeIterable, 393
 Global.~, 293

rank
 FormatInfo.~, 285
 rate
 Interface.Laser.~, 484
 ratio
 Interface.BlobDetector.~, 480
 readable
 Path.~, 375
 reboot
 System.~, 430
 receive
 Lobby.~, 329
 received
 Socket.~, 413
 reconnect
 Subscription.~, 425
 '--reconstruct', 138
 Redefinition
 Exception.~, 263
 redefinitionMode
 System.~, 430
 reduce, 319
 Regexp, 395
 Global.~, 293
 release
 Semaphore.~, 399
 relocatable, 96
 remain
 Interface.AudioOut.~, 479
 Interface.Battery.~, 480
 remote mode, 112
 '--remote', 139
 remoteIP
 Lobby.~, 330
 remove
 Directory.~, 250
 File.~, 268
 Group.~, 297
 List.~, 322
 removeAll
 Directory.~, 250
 removeBack
 List.~, 322
 removeById
 List.~, 323
 removeFront
 List.~, 323
 removeLocalSlot
 Object.~, 364
 removeNotifyAccess
 Slot.~, 408
 removeNotifyChange
 Slot.~, 408
 removeNotifyChangeOwned
 Slot.~, 408
 removeProperty
 Object.~, 364
 removeProto
 Object.~, 365
 removeSlot
 Object.~, 365
 rename
 Directory.~, 250
 File.~, 268
 Path.~, 376
 replace
 String.~, 423
 reqStruct
 Ros.Service.~, 474
 request
 Ros.Service.~, 474
 requireFile
 System.~, 431
 reset
 Kernel1.~, 310
 resetConnectionStats
 uobjects.~, 459
 resetStats
 Job.~, 307
 Subscription.~, 425
 System.~, 431
 resize
 Matrix.~, 347
 Vector.~, 466
 resolution
 Interface.Laser.~, 484
 Interface.VideoIn.~, 486
 '--resolution=resolution', 138
 resStruct
 Ros.Service.~, 474
 return, 115, 165
 reverse
 List.~, 323
 revision
 System.PackageInfo.~, 437
 rfind
 String.~, 423
 RGBLed, 481
 robot, 488
 roll, 490
 root, 96
 Ros, 469
 Ros.Service, 474
 Ros.Topic, 470
 RotationalMotor, 483
 RotationalSpeedMotor, 483

round
 Float. \sim , 278
 Math. \sim , 339
 routine, 172
 row
 Matrix. \sim , 348
 rowAdd
 Matrix. \sim , 348
 rowDiv
 Matrix. \sim , 348
 rowMul
 Matrix. \sim , 349
 rowNorm
 Matrix. \sim , 349
 rowSub
 Matrix. \sim , 349
 RTP, 518
 run
 Process. \sim , 384
 running
 Timeout. \sim , 448
 runningcommands
 Kernel1. \sim , 310
 runtime path, 102
 runTo
 Process. \sim , 384

S

‘-s’, 136, 138–140, 144
 save
 File. \sim , 269
 say
 Interface.TextToSpeech. \sim , 485
 scalarEQ
 Vector. \sim , 466
 scalarGE
 Vector. \sim , 466
 scalarGT
 Vector. \sim , 466
 scalarLE
 Vector. \sim , 466
 scalarLT
 Vector. \sim , 467
 ‘--scale=*factor*’, 138
 Scheduling
 Exception. \sim , 263
 scientific notation, 150
 scope, 39, 161
 scope
 Tag. \sim , 446
 scopeTag, 442
 System. \sim , 431
 script
 Interface.TextToSpeech. \sim , 485
 search-path, 133
 searchFile
 System. \sim , 431
 searchPath
 System. \sim , 431
 UObject. \sim , 458
 uobjects. \sim , 459
 second
 Date. \sim , 238
 Pair. \sim , 372
 Triplet. \sim , 455
 seconds
 Duration. \sim , 252
 selfTime
 Profile.Function. \sim , 389
 selfTimePer
 Profile.Function. \sim , 389
 Semaphore, 398
 Global. \sim , 293
 send
 Lobby. \sim , 330
 send
 Java, 526
 sender
 CallMessage. \sim , 224
 Sensor, 483
 seq
 Float. \sim , 278
 Kernel1. \sim , 310
 serial
 Interface.Identity. \sim , 480
 Serializables, 400
 serialize
 Vector. \sim , 467
 Server, 401
 Server, 401
 Global. \sim , 293
 ‘--server’, 140
 Service
 Ros. \sim , 470
 services
 Ros. \sim , 470
 set
 Dictionary. \sim , 244
 Logger. \sim , 336
 Matrix. \sim , 350
 Slot. \sim , 406
 Vector. \sim , 467
 setAbsolutePosition
 Interface.Mobile. \sim , 481
 setConstSlot
 Object. \sim , 365

setenv
 System.~, 432
 setLocale
 System.~, 432
 setOutputValue
 Slot.~, 408
 setProperty
 Group.~, 297
 Object.~, 366
 setProtos
 Object.~, 366
 setRow
 Matrix.~, 350
 setSlot
 Object.~, 366
 setTrace
 uobjects.~, 459
 shallow copy, 516
 shared objects, 502
 '--shared', 144
 shiftedTime
 System.~, 433
 shoulder, 488
 shutdown
 System.~, 433
 sign
 Float.~, 279
 Math.~, 339
 signal
 Barrier.~, 216
 signalAll
 Barrier.~, 216
 Sin, 450
 TrajectoryGenerator.~, 454
 sin
 Float.~, 279
 Math.~, 340
 Singleton, 402
 Global.~, 293
 singleton, 402
 size
 Dictionary.~, 244
 Directory.~, 250
 Enumeration.~, 254
 File.~, 269
 Kernel1.~, 310
 List.~, 323
 Matrix.~, 350
 String.~, 423
 Tuple.~, 457
 Vector.~, 467
 sleep
 System.~, 433
 Slot, 403
 slot, 43, 169
 slotNames
 Object.~, 366
 Smooth, 451
 TrajectoryGenerator.~, 454
 Socket, 410
 Socket, 410
 Global.~, 293
 sockets
 Server.~, 401
 sort
 List.~, 323
 SOUND, 516
 sound, 516
 spawn
 Code.~, 230
 System.~, 433
 speaker, 491
 spec
 FormatInfo.~, 286
 speech, 491
 SpeechRecognizer, 485
 Speed, 451
 speed
 Interface.GyroSensor.~, 484
 Interface.LinearSpeedMotor.~, 483
 Interface.RotationalSpeedMotor.~, 483
 Interface.TextToSpeech.~, 485
 SpeedAdaptive
 TrajectoryGenerator.~, 454
 spine, 490
 split
 Slot.~, 408
 String.~, 423
 split mode, 408
 sqr
 Float.~, 279
 Math.~, 340
 sqrt
 Float.~, 279
 Math.~, 340
 srandom
 Float.~, 279
 Math.~, 340
 stack size, 136
 '--stack-size=*size*', 136
 StackFrame, 414
 '--start', 139
 stats
 Job.~, 307
 System.~, 433
 status

Job.~, 307
Process.~, 384
--stay-alive, 140
stderr
 Process.~, 384
stdin
 Process.~, 384
stdout
 Process.~, 384
stop, 440
stop
 Interface.Mobile.~, 481
 Tag.~, 446
Stream, 416
strict
 Kernel1.~, 310
String, 417
 Global.~, 293
string, 151, 373, 417, 547
string
 System.PackageInfo.~, 437
strlen
 Kernel1.~, 310
structural pattern matching, 174
structure
 Ros.Topic.~, 471
structure tree, 475
subMinor
 System.PackageInfo.~, 437
subscribe
 Event.~, 259
 PubSub.~, 391
 Ros.Topic.~, 471
Subscriber
 PubSub.~, 391
subscriberCount
 Ros.Topic.~, 471
subscribers
 Event.~, 259
 PubSub.~, 391
Subscription, 425
subset
 List.~, 323
sum
 Vector.~, 467
switch, 182
syncEmit
 Event.~, 259
synchronicity, 255
synchronous, 255, 505
syncline, 148
syncTrigger
 Event.~, 259
syncWrite
 Socket.~, 413
Syntax
 Exception.~, 263
System, 426
 Global.~, 293
system
 System.~, 434
System.PackageInfo, 436
System.Platform, 439

T

Tag, 440
 Global.~, 293
tag, 111, 440
taglist
 Kernel1.~, 310
tags
 Job.~, 308
tail, 489
 List.~, 324
tan
 Float.~, 279
 Math.~, 340
tarballVersion
 System.PackageInfo.~, 437
target, 40
target
 CallMessage.~, 224
target value, 207, 450
tablename
 System.PackageInfo.~, 437
temperature
 Interface.TemperatureSensor.~, 484
TemperatureSensor, 484
terminate
 Job.~, 308
TextToSpeech, 485
thanks
 Lobby.~, 330
then-clause, 180
third
 Triplet.~, 455
thread-safety, 112
threshold
 Interface.BlobDetector.~, 480
throw, 184
Time, 452
 TrajectoryGenerator.~, 454
time
 System.~, 434
TimeAdaptive
 TrajectoryGenerator.~, 454

timedOut
 Timeout.~, 448
TimeOut
 Exception.~, 263
Timeout, 447
 Global.~, 294
timeReference
 System.~, 434
times
 Float.~, 279
timeShift
 Job.~, 308
timestamp
 Date.~, 238
toAscii
 String.~, 424
toe, 489
toggle
 Loadable.~, 326
toLowerCase
 String.~, 424
Topic
 Ros.~, 470
topics
 Ros.~, 470
topLevel
 Channel.~, 227
torque
 Interface.RotationalMotor.~, 483
torso, 490
totalCalls
 Profile.~, 387
totalTime
 Profile.~, 387
touch, 490
TouchSensor, 485
toUpperCase
 String.~, 424
trace
 Logger.~, 337
Traceable, 449
Tracker, 486
trajectory, 207
TrajectoryGenerator, 450
 Global.~, 294
transparent
 UValue.~, 460
transpose
 Matrix.~, 350
trigger
 Event.~, 259
triple, 455
Triplet, 455
 Global.~, 294
triplet, 455
true, 220
 Global.~, 294
trueIndexes
 Vector.~, 467
trunc
 Float.~, 279
 Math.~, 340
Tuple, 456
 Global.~, 294
tuple, 456
tuples, 152
turn
 Interface.Mobile.~, 481
 Interface.RotationalMotor.~, 483
Type
 Exception.~, 263
type
 Matrix.~, 350
 Object.~, 367
 Slot.~, 409
 Vector.~, 467

U

UBindEvent, 515
UBindFunction, 511
UBindFunction
 Java, 525
UBindFunctions, 511
UBindFunctions
 Java, 525
UBindThreadedFunction, 511
UBindVar, 514
UBindVar
 Java, 523
uid
 Object.~, 367
umake, 143
umake-deepclean, 145
umake-java, 522
umake-shared, 145
unacceptVoid
 Object.~, 367
 void.~, 468
unadvertise
 Ros.Topic.~, 473
unblock, 440
unblock
 Tag.~, 446
undefall
 Kernel1.~, 310
UnexpectedVoid

Exception.~, 263
unfreeze
 Tag.~, 446
unique
 List.~, 324
UNKNOWN, 516
unnotify, 512, 515
unnotify
 Java, 526
UNotifyAccess, 512
UNotifyChange, 512
UNotifyChange
 Java, 526
unreachable
 Interface.Mobile.~, 482
unsetenv
 System.~, 435
unstrict
 Kernel1.~, 310
unsubscribe
 PubSub.~, 391
 Ros.Topic.~, 471
UObject, 3, 112
UObject, 458
 Global.~, 294
uobjects, 459
 Global.~, 294
updateSlot
 Group.~, 297
 Object.~, 367
uppercase
 FormatInfo.~, 286
urbi, 29
urbi, 135
 Urbi Runtime, 3
urbi-image, 137
urbi-launch, 29
urbi-launch, 138
urbi-launch-java, 522
urbi-ping, 141
urbi-root, 96, 134
urbi-send, 142
urbi-sound, 142
‘URBI.INI’, 135
urbi.UMain, 527
URBI_ACCEPT_BINARY_MISMATCH, 124
URBI_CHECK_MODE, 124
URBI_DESUGAR, 124
URBI_DOC, 124
URBI_IGNORE_URBI_U, 124
URBI_INTERACTIVE, 124
URBI_LAUNCH, 124
URBI_NO_ICE_CATCHER, 124
URBI_PARSER, 124
URBI_PATH, 134
URBI_REPORT, 124
URBI_ROOT, 134
URBI_ROOT_LIBname, 124
URBI_SCANNER, 124
URBI_SHARE, 124
URBI_TEXT_MODE, 124, 134
URBI_TOPLEVEL, 124
URBI_UOBJECT_PATH, 134
us
 Date.~, 238
uservars
 Kernel1.~, 310
USetTimer
 C++, 514
 Java, 526
USetUpdate
 C++, 514
 Java, 526
UTF-8, 147
UValue, 460
 Global.~, 294
UValueSerializable, 461
uvalueSerialize
 UValueSerializable.~, 461
UVar, 112
UVar
 Global.~, 294
V
v
 Semaphore.~, 399
‘-V’, 144
‘-v’, 144
val
 Interface.AudioIn.~, 479
 Interface.AudioOut.~, 479
 Interface.Laser.~, 484
 Interface.Led.~, 481
 Interface.Motor.~, 482
 Interface.Sensor.~, 483
 Interface.VideoIn.~, 486
value
 Lazy.~, 313
 Slot.~, 409
values
 Enumeration.~, 254
variable, 39, 547
 local, 162
variadic, 161, 168
vars
 Kernel1.~, 310

Vector, 462
 ‘**--verbose**’, 144
version
 System.PackageInfo.~, 437
 System.~, 435
 ‘**--version**’, 135, 138–144
versionRev
 System.PackageInfo.~, 438
versionValue
 System.PackageInfo.~, 438
VideoIn, 486
visible
 Interface.BlobDetector.~, 480
voice, 491
 Interface.TextToSpeech.~, 485
voicexml
 Interface.TextToSpeech.~, 485
void, 468
 Global.~, 294
voltage
 Interface.Battery.~, 480
volume
 Interface.AudioOut.~, 479
VPATH, 144

W

‘**-w**’, 136
wait
 Barrier.~, 216
waitForTermination
 Job.~, 308
waituntil, 202
wall
 Global.~, 294
 Lobby.~, 330
wallClockTime
 Profile.~, 387
warn
 Global.~, 294
 Logger.~, 337
warning
 Channel.~, 227
watch, 204
watchdog
 Interface.Mobile.~, 482
watchdogInterval
 Interface.Mobile.~, 482
wb
 Interface.VideoIn.~, 486
wheel, 488
whenever, 204, 206
while, 182
 while,, 196

while;, 183
 while|, 183
width
 FormatInfo.~, 286
 Interface.VideoIn.~, 486
wrist, 488
writable
 Path.~, 376
write
 Lobby.~, 330
 Socket.~, 413

X

x, 490
 Interface.BlobDetector.~, 480

xfov
 Interface.VideoIn.~, 486

Y

y, 490
 Interface.BlobDetector.~, 480

yaw, 489
 Interface.Tracker.~, 486

year
 Date.~, 239

yfov
 Interface.VideoIn.~, 486

yields
 Profile.~, 387

Z

z, 490
zip
 List.~, 324
 Vector.~, 467