REFLEX IN ACTION

september 13, 2012

RESEARCH BY

INCAPTURE

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Part I.

Introducing Reflex

1

BACKGROUND

Recently there has been a significant move to create languages that run on top of the Java Virtual Machine. These languages are usually integrated as a fundamental "Java Scripting Language" - such as JRuby (an implementation of Ruby), Jython (the equivalent of Python) and Rhino (a JavaScript implementation) or as a fully defined language that hooks into the JVM at a lower level. Examples of these languages would include Clojure (a functional Lisp dialect), Groovy (a scripting language) and Scala (an object-oriented and functional programming language).

Reflex is a procedural language that hooks into the JVM at a lower level. Its syntax is similar to Python (without the indentation) and there are a number of built-in functions and special operators that are semantic short cuts when interacting with **Rapture** .

This book provides a detailed description of **Reflex** .

why reflex?

The designers of **Rapture** felt that it was very important to provide a non-compiled way of running programs "in the cloud". The idea was that **Rapture** application developers could define a lot of their data manipulations in a scripting language and then have **Rapture** run these "programs" on the servers that make up an installed system. If such a program could be agnostic to the machine (actual server) it was running on such manipulations could be run in a scalable way across all of the servers that make up a **Rapture** system.

Initially the support for such a scripting language was built using JRuby and Jython - two well established languages that have a well educated developer base. Unfortunately this support ended up being impractical - both from a licensing perspective and from the fact that the scripting environment for these languages was very resource (time and memory) intensive. For these reasons it was decided to create a small language (now called **Reflex**) to provide the "glue" for the data interactions in **Rapture** .

a short summary of rapture

Rapture is a system for storing and manipulating data in a cloud (or distributed) server environment. Data is usually stored in "documents" in a json format and **Rapture** isolates a developer from the underlying technology used to store that data. When data is changed in **Rapture** scripts can be run to act upon that change. **Rapture** also defines the concept of a workflow - a set of tasks that are executed serially to perform a formal update to the data (such as an End Of Day process). Each step in a workflow is implemented by the execution of a script. **Rapture** has

a rich API that is exposed through a HTTP based protocol. Client side implementations of this API have been created for Python, Ruby, Java, .NET (and VBA for Excel spreadsheets) and JavaScript. There is also a REST based interface for even easier to access the documents that make up a **Rapture** system in a RESTful way. **Rapture** has a full entitlements system to protect from unauthorized activity and a full audit trail to record authorized activity. **Rapture** is also customized through many hook mechanisms - to provide analytics in a Risk Management System or to provide market data for pricing. The book "**Rapture** in Action" describes this environment in full detail. The important thing to understand is that a scripting language "binds" the **Rapture** system together - it is the glue that runs the workflow system, it is used to propagate events and it can be compared to the "stored procedure language" of a "database management system".

2

SERVER HOSTED SCRIPTING

The initial use for **Reflex** was to provide a scripting language that could be run on a server. However the same language could also be used to help setup a **Rapture** system and hooks were also built in to handle file based io - something that normally would be restricted on a cloud/distributed server environment. In fact a **Reflex** environment now has a number of "hooks" that can be implemented (or wired) differently depending on the context. In general the environment looks like the logical diagram below.

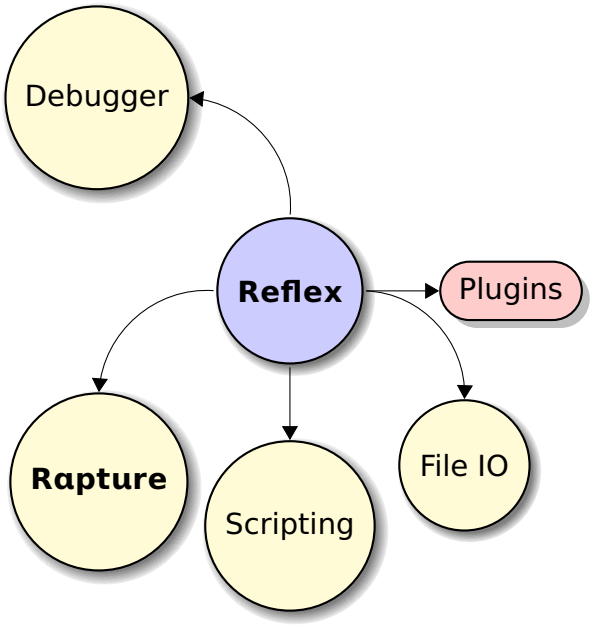

Figure 1.: Logical **Reflex** environment

Here we see that **Reflex** can reach out to a debugger, a **Rapture** environment (for calling its API), a Scripting environment (for loading other scripts) and an IO sub-system for loading and saving data to a file system.

When running within a **Rapture** server, the implementations are frozen to protect the environment:

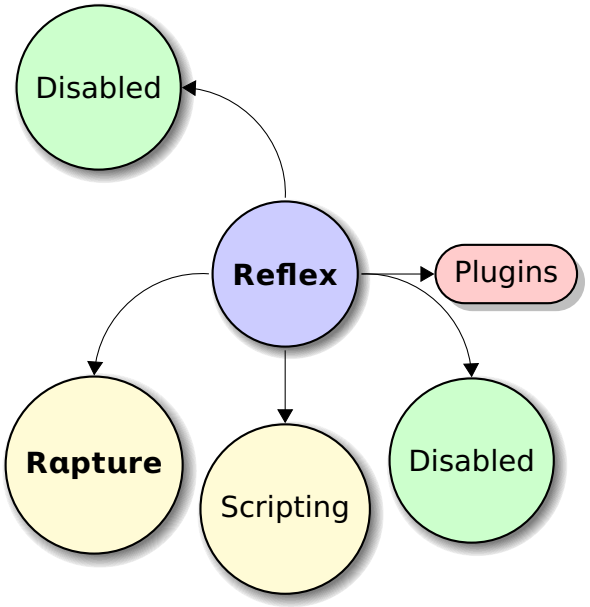

Figure 2.: Server **Reflex** environment

Here the debugger and the file/IO subsystems are disabled.

Reflex can also be run on a local desktop, or on a server outside of a **Rapture** environment. In this case the bindings of the environment are as below:

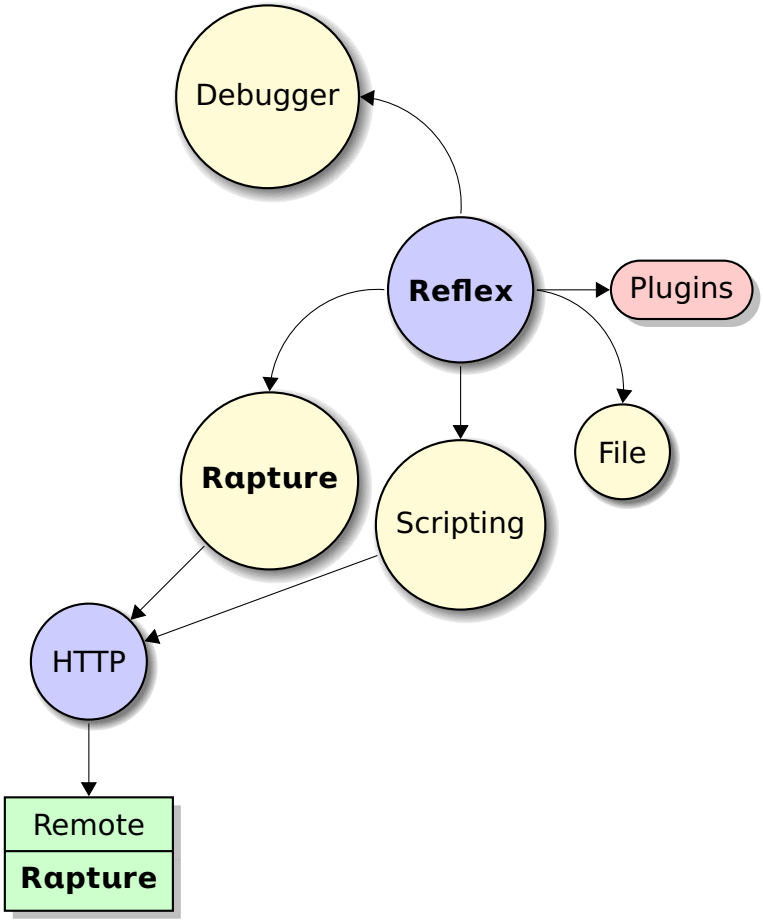

Figure 3.: External **Reflex** environment

In this case the environment has a **Rapture** system wired in via a standard HTTP based API - all **Rapture** commands in **Reflex** will still work through that API. In a server based environment the security context is set by **Rapture** (and is based on the ultimate initiator of the **Reflex** process). In the external approach the user security context is set either by using a **Rapture** "API key" or by logging in manually through the Reflex runner application.

installing reflex

There are three options for using **Reflex** . The most common use for **Reflex** is to run scripts from within a **Rapture** environment - you upload scripts to **Rapture** and then call them through either **Rapture** 's API call `runProgram` or through a **Rapture** workflow, operation or event handling. For testing and debugging it is preferable to install a local environment to play with. This section describes how to do that.

Reflex is bundled into an application called *ReflexRunner* that can be used to run **Reflex** scripts. ReflexRunner is a command line java application that can be downloaded from the **Rapture** Release Site. Once downloaded it can be run using java as follows:

```
java -jar [ReflexRunner.jar]
```

```
        -r [RaptureAPIURL] -f [ReflextScript]
```

Optional parameters are listed below:

```
-u 'user' - User name to login as
-p 'password' - Password to use
-d - Start debugger
```

installing into eclipse

Reflex (and **Rapture**) have an Eclipse plugin that can also be used to assist in the development of **Reflex** scripts. This section describes how to install the plugin.

The first step is to install the latest version of Eclipse (recommended is at least the Java IDE for Eclipse) from http://eclipse.org.

From within Eclipse, navigate to the Help menu and click on *Install New Software...*. You should see a dialog similar to that in Figure 4.

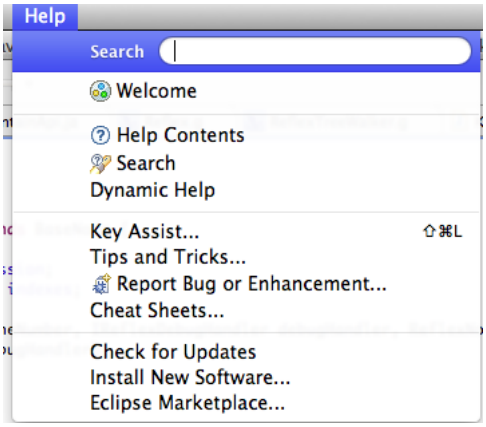

Figure 4.: Help/Install menu in Eclipse

In the Install dialog, click on the Add button and then add the Incapture **Rapture** Eclipse site to the Eclipse sites. If you have already done this simply pick that site from the drop-down instead. This is shown in Figure 5 on the next page.

Clicking on OK will add the site and determine what updates or options are available. In this case pick Rapture and if more than one item is shown pick the item with the greatest release version. The dialog will look similar to that in Figure 6 on page 18.

Clicking Next will then go through a validation process and eventually you will need to accept the license agreement and then restart Eclipse using a dialog similar to that in Figure 7 on page 18.

configuration

The first thing to do post-install is to configure the **Rapture** plugin for your local environment. The plugin will need to be told the location of the **Rapture** environ-

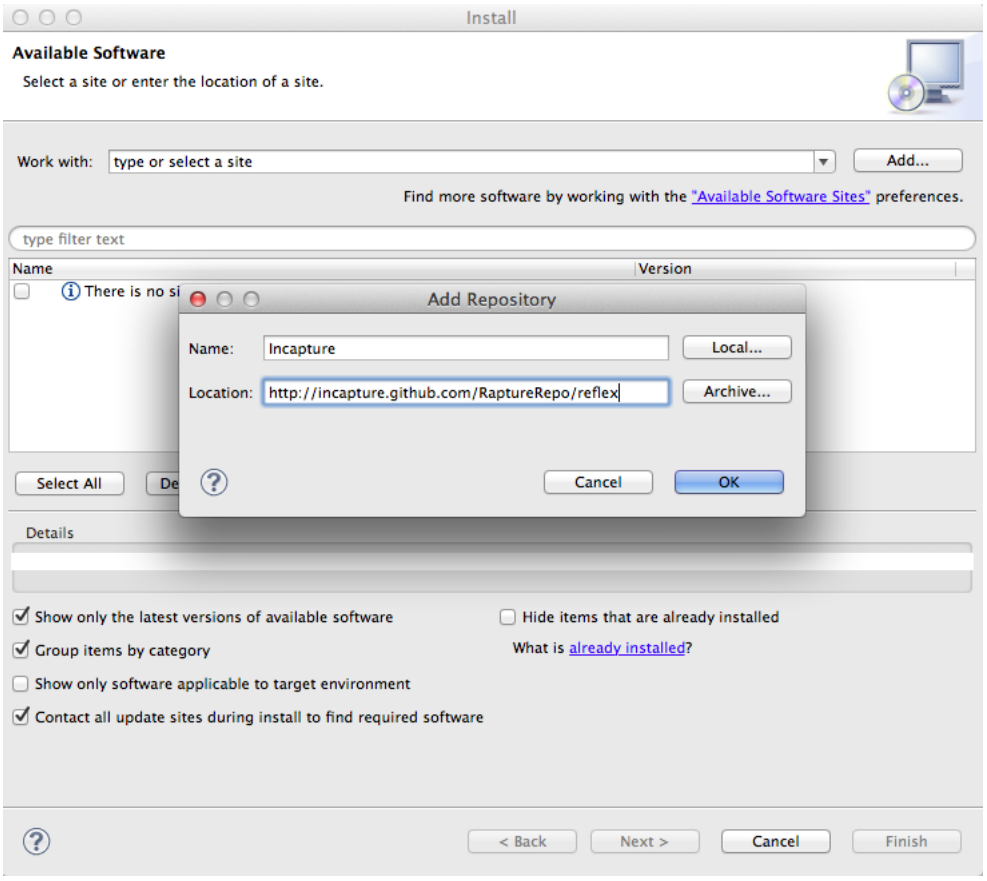

Figure 5.: Install Dialog in Eclipse

ment you will be connecting to. You do this in the Preferences dialog of Eclipse, in the **Rapture** section as shown in Figure 8 on page 19.

reflex scripts

(Script running, Console output, Debug output, Caveats)

reflex fundamentals

Reflex is a basic procedural language with a few special operators and built-in functions to make **Rapture** interactions quicker to write. In this case the meaning of "procedural" is simply that in **Reflex** you can define functions and invoke those functions. This section describes the fundamental characteristics of the language.

hello, world

The simplest program in **Reflex** uses the built-in function `println` to print out the contents to standard out:

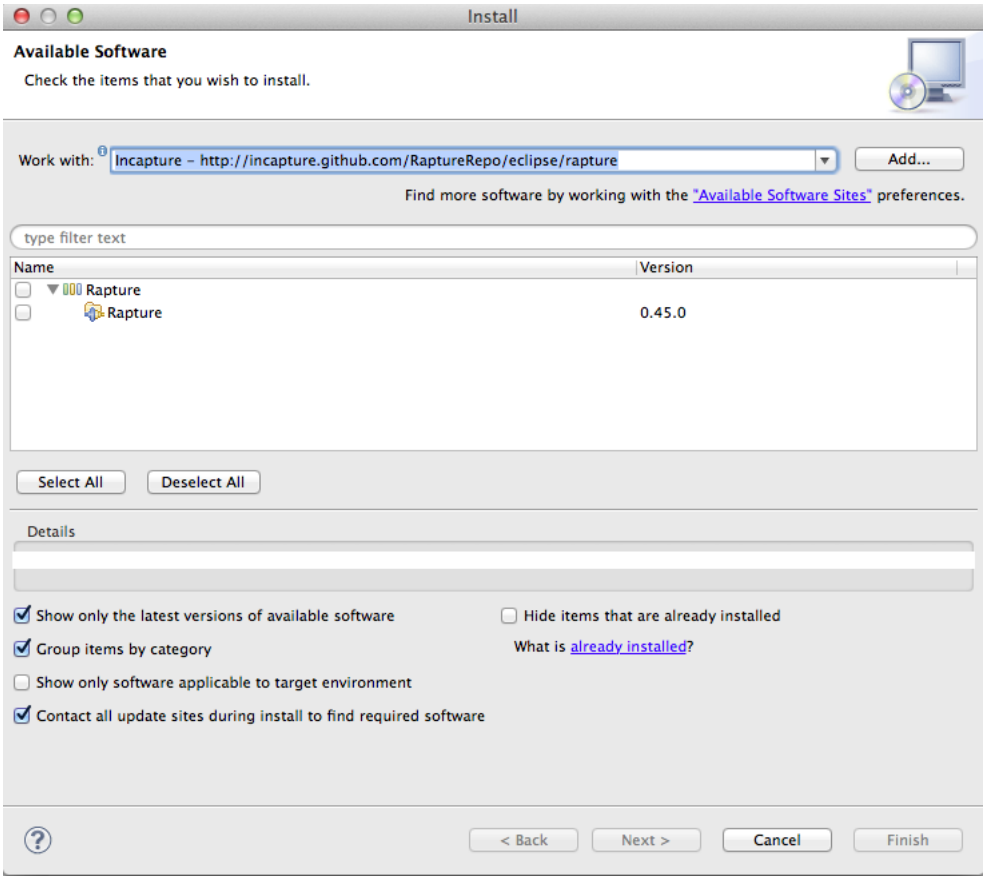

Figure 6.: Available Software

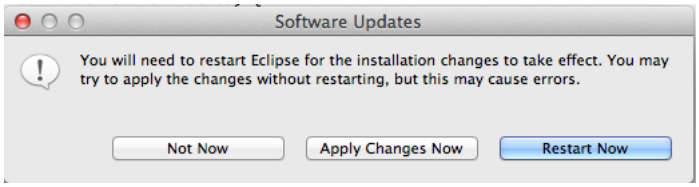

Figure 7.: Accept License

```
1 // This is Hello World in Reflex
2 println('Hello, world');
```

Listing 1: Hello world

Running the above program in Eclipse or through the ReflexRunner will print out the string 'Hello, world' on the console. It introduces two concepts. Line 1 shows how comments are defined in **Reflex** . Comments are prefixed with a double slash and continue to the end of the line. This is the only comment style in **Reflex** . Line 2 shows the built-in function `println` and the fact that strings can be enclosed either in single quotes or double quotes. Finally statements in **Reflex** are terminated by semi-colons.

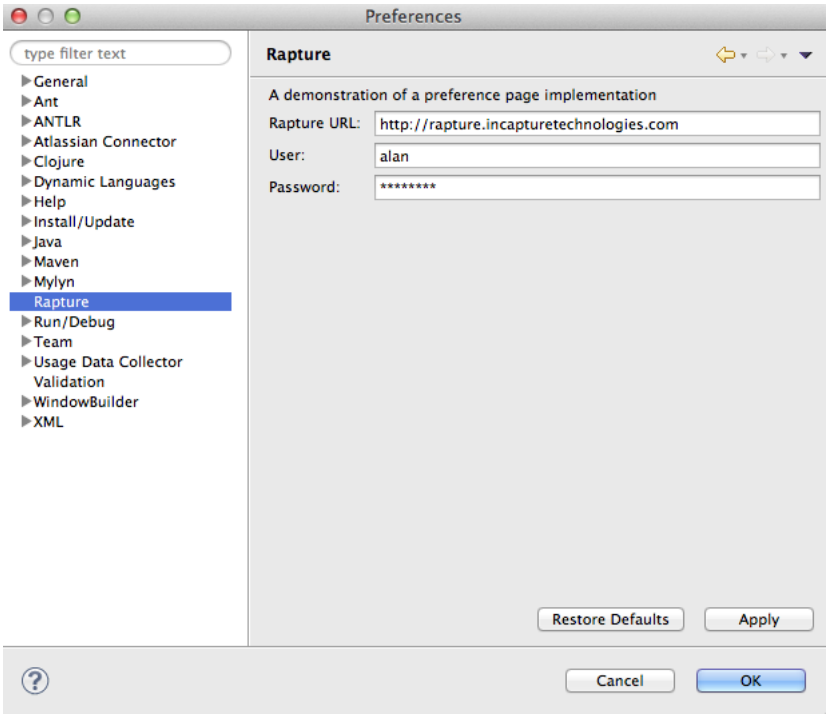

Figure 8.: Configuration Dialog

variables and types

Variables in **Reflex** are created upon first declaration and can be named using the letters of the alphabet and the digits 0 - 9. Some examples of variable names are shown in the table below:

Variable Name	Valid?
abc	Yes
aB0	Yes
a*3	No (contains a *)
reflex	Yes
_var	No (contains a _)

Table 1.: Variable names in **Reflex**

Types in **Reflex** are inferred from the context in which a value is assigned, and wherever possible a type will be coerced into another if the other type is needed for a context. The types are listed in table **??** on the next page.

The more complex types (queue, file) will be introduced in a later section on integration with **Rapture** .

Type	Example	Description
string	'Hello'	A string, an array of characters
number	4.0	A number, either an integer or a float, depending on context
boolean	true	A boolean value, either true or false
list	`[ 1, 2, 3]`	A list of values
map	`{ 'key' : 'value' }`	An associate map, mapping keys (strings) to values.
date	`date()`	A date.
time	`time()`	A time.
file	`file('test.txt')`	A file object, used to read or write data.
queue	`que('test','thequeue')`	A queue object, used to receive and send messages in **Rapture** .
nul	`null`	An object representing a nul value.
void	`void`	An object representing "no" object.

Table 2.: Types in **Reflex**

type conversion

Reflex will attempt to convert from one type to another as needed. The table below shows what conversions are implicit and which ones need a little help from a built in function.

initialization

Reflex types are determined by context - for example the result of calling the built-in `size` function is a number, so assigning a variable to the result of calling that function will result in a numeric variable being created. Another way of defining the type of a variable is to initialize it. Reflex uses the format of the initialization to determine the type.

A variable initialization can also be prefixed with the keyword `const`. This keyword ensures that the value of this variable will be unchanged after first assignment, and that the variable will be accessible globally, including within functions.

	From						
To	String	Number	Boolean	List	Map	Date	Time
String		auto	auto	auto	auto	auto	auto
Number	cast		n/a	n/a	n/a	Julian	Millis
Boolean	n/a	n/a		n/a	n/a	n/a	n/a
List	n/a	n/a	n/a		n/a	n/a	n/a
Map	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Date	YYYYMMDD	Julian	n/a	n/a	n/a		n/a
Time	HH:MM:SS	Millis	n/a	n/a	n/a	n/a	

Table 3.: Conversions in **Reflex**

string

The string type is initialized through a statement enclosed in either single quotes or double quotes. The different quoting options are equivalent in **Reflex** - the choice is a matter of convenience. If your string needs to contain double quotes you can enclose it in single quotes and vice versa.

```
// String initialization
a = 'a string';
b = 'another string';
c = "A string in quotes";
const d = "There's a string in here somewhere";
```

Listing 2: String initialization

number

The number type is initialized through a statement that represents a number - either an integer (a series of digits) or a floating point number through either a series of digits, a decimal point and a further series of digits, or through the definition of a number through scientific notation. If you suffix a number representation with a capital L the number will be locked to an internal integer type, which is useful when using these numbers in native Java calls.

```
// Number initializaion
a = 10;
b = 100.4;
const c = 5L;
d = 4.5E04; // equivalent to 45000
```

Listing 3: Number initialization

boolean

The boolean type is defined using the keywords `true` and `false`, and of course can be initialized through the use of a boolean statement (see boolean operators in a later section).

```
// Boolean initialization
a = true;
const b = false;
c = (1 == 1); // c == true
```

Listing 4: Boolean initialization

list

The list type is defined using square brackets, with elements of the list being separated by commas. The elements can be any expression which includes the name of an existing (pre-defined) variable.

```
// List initialization
a = []; // An empty list
const b = [1, 2, 3, 4, 5];
c = ['A string', 1, []]; // A list of a string, a number, and an empty
    list
```

Listing 5: List initialization

map

A map is initialized using a JSON style format and is best demonstrated through example.

```
// Map initialization
a = {}; // An empty map
b = { 'a' : 4 }; // A map containing a
                 // single entry, with
                 // the key 'a' and the value 4
c = { 'one' : 1, 'two' : 2, 'three' : 3 };
const d = { 'outer' : { 'inner' : true }};
```

Listing 6: Map initialization

simple examples

Based on our understanding of types and variables, and our simple `println` function, we can create some new **Reflex** scripts. Here is a longer script that creates

some variables and introduces some of the operators that will be expanded upon in the next section.

```
1
2 // An example Reflex script showing variables and types
3 x = 10;
4 y = "Hello";
5 z = x + 1;
6 println("Z is " + z);
```

Listing 7: Variables and Types

In this example we have defined three variables, *x*, *y* and *z*. *x* and *z* are ultimately numbers and *y* is a string. We create the variable *x* on line 3, and *y* on line 4. *z* is implicitly created as the value of *x 1* (i.e. 11). Finally we print out the value of *z* on line 6, but note that we have used the operator on a string (`Z is`) and a number (the variable *z*). In this case **Reflex** will convert *z* from a number to a string and then append the strings together.

3 OPERATORS

Operators in **Reflex** will, in the most part, be very familiar to developers of other languages. There are boolean operators (, >, <, <, >, ! , ||, &&), arithmetic operators (,−,/,∗,%) and index operators ([ ]). The ternary operator ? is also supported. The use of these *simple* operators is best illustrated by example. The examples below also introduce the `assert` built-in function - it aborts the **Reflex** script with an error if the result of the boolean expression is `false`.

```
// Examples of use of simple operators
// Boolean operators
assert(true);
assert(true || false);
assert(!false);
assert(true && true);
// Relational
assert(1 < 2);
assert(55 >= 55);
assert('a' < 'b'); // Note that strings can be compared
// Addition
assert(1 + 999 == 1000);
assert([1] + 1 == [1,1]); // Note addition on lists
assert([1,2,3] - 3 == [1,2]); // Note subtraction on lists
// Multiply
assert(3 * 50 == 150);
assert(4 / 2 == 2);
assert(999 % 3 == 0); // % = mod operator
// Power
assert(2 ^ 3 == 8);
```

Listing 8: Simple operators

It is worth calling out explicitly how and − work with lists. If the left hand side of an expression is a list, then adding an element to it results in a new list with that element added to the end. Subtracting from a list removes that element from the list if it is within the list. This also works with strings.

The index [ ] operator is worth its own set of examples.

```
// Examples of the index operator
a = [1, 2, 3, 4, 5];
b = 'abcdefg';
assert(a[0] == 1);
assert(a[1 .. 2] == [2,3]);
assert(b[0] == 'a');
assert(b[1 .. 2] == 'bc');
```

Listing 9: Index operator

There are two forms of the index operator. The first, with one integer parameter, simply returns the element at that position. The second, with the `..` directive is a range operator - it returns the elements between these index points, inclusive of the first parameter and exclusive of the second.

The index operator also applies to map types as well. In this case the parameter is a string and refers to the key to lookup in the associative map.

```
a = { 'one' : 1, 'two' : 2 };
assert(a['one'] == 1);
assert(a['two'] == 2);
```

Listing 10: Index operator on maps

4

FLOW CONTROL

Reflex has the standard flow control statements such as `if ... else`, `while`, `for loops`.

if

The **Reflex** `if` statement has the following form:

```
if booleanExpression do
   block
else do
   block
end
```

For statements without an else block the complete `else do block` can be omitted.

```
1  // An If statement
2  a = 4;
3  b = 2;
4  if a > 3 do
5     println("A is greater than 3");
6  else do
7     println("A is less than or equal to 3");
8  end
9
10 if b == 2 do
11    println("Yes, b is 2");
12 end
```

Listing 11: If statement

Note that there does not need to be a semi-colon after the end keyword here.

while

The **Reflex** `while` statement has the following form:

```
while booleanExpression do
    block
end
```

Reflex currently does not have a `break` keyword.

```
// A while loop
a = true;
b = 0;
while a do
   b = b + 1;
   if b > 5 do
      a = false;
   end
end
```

Listing 12: While statement

for

Reflex has two different `for` loop forms. The first, the counting form, assigns a numeric variable the values from a starting number to an ending number (inclusive) and calls the inner block for each iteration.

```
// A for loop
for a = 0 to 10 do
   println("The value of a is " + a);
end
```

Listing 13: For counting form

The second is known as the iterator form, and it takes as a secondary argument a list expression (which can be a variable or an expression that yields a list). The value of the variable is set to each element in the list and the inner block is called with that element set.

```
// A for loop
a = [1, 2, 3, 4 ];
b = [];
for c in a do
   b = b + ( c * 2 );
end
assert(c == [2, 4, 6, 8 ];
```

Listing 14: For iterator form

pfor

Reflex also has a novel way of running `for` blocks in parallel, through the `pfor` keyword. `Pfor` can replace `for` in most cases and **Reflex** will attempt to run the loop in parallel, with each statement being executed on a pool of threads. Care

must be taken with this approach as sequencing of changes can occur out of a natural order. Both the counting and iterator form are supported.

```
// A pfor loop

res = {};
pfor a = 0 to 10 do
   res['' + a] = a;
end

println("The resultant map is " + res);
```

Listing 15: PFor counting form

5 EXCEPTIONS

Reflex supports exceptions in the form of a `try/catch` construct. An example will best illustrate the approach.

```
// A simple test of exception structure

x = 0;
y = false;

def addIt(var)
   var = var + 1;
   throw "From the function " + var;
   return var;
end

try do
    x = addIt(x);
    println("After function, but not caught");
end
catch e do
    println("Caught exception " + e);
    y = true;
end

assert(x == 0);
assert(y);
```

Listing 16: Exception handling

In this example we calla function that will increment our parameter, but the function throws an exception before the parameter is returned. In the exception handler we set the y variable to true. Both assertions at the end of the script are valid – *x* is still zero because the function threw an exception before it could be updated. And *y* is `true` because we entered the exception handler.

Reflex can also catch general exceptions thrown by internal or addin functions.

6

SPECIAL OPERATORS

There are two special operators in **Reflex** . They are `-->` and `<--`. The former is known as the "push" operator and the latter the "pull" operator.

Reflex scripts are primarily about taking data from **Rapture** , manipulating it, and then putting that data back. The push and pull operators can be used to get data from **Rapture** and to save it back.

As an example, consider a **Rapture** environment with a partition `test` and a type in that partition with the name `config`. The script in the following listing will save some data to **Rapture** in a configuration document and then later on, retrieve that data and use the information within to control the script. The script also shows an example of initializing a map and writing values to it. When used in this form, the push and pull operators assume a map type on the left hand side and a string on the right.

```
1  // Data push and pull
2  config = {};
3  config['option1'] = true;
4  config['level'] = 42;
5
6  displayName = 'test/official/config/main';
7
8  config --> displayName; // Write the map to the document
9
10 // Later on in a different script
11
12 appConfig <-- displayName;
13 if appConfig['option1'] do
14    println("Level is " + appConfig['level']);
15 else do
16    println("Option1 is not set');
17 end
```

Listing 17: Push and Pull

The push and pull operators also work with either a *queue* or a *file* type on the right hand side. For a queue, the operator either puts an entry onto a **Rapture** queue or takes one off. For a file, the pull operator returns the contents of the file (as a string) and the push operator writes the string to the file. These uses are discussed in more detail in the relevant sections below on IO.

7

USER-DEFINED FUNCTIONS

User-defined functions can also be built in **Reflex** . The main structure of a function definition is show below:

```
def functionName ( parameters )
    block
end

functionName(parameters);
```

A simple example of a function being defined and used is in the following listing.

```
1
2 const prefix = "I'll say ";
3
4 def sayWhat(name, what)
5    println(prefix + what + " to " + name);
6 end
7
8 sayWhat('Alan', 'hello');
9 sayWhat('Alan', 42);
```

Listing 18: Function definition

Some important points about function declarations and invocations. When defining a function the parameter types are not defined, just their names. So a developer can be very free with the type of parameters as long as the body of the function can also tolerate the type differences. You can see an example of that in the listing above, where the `what` parameter is also passed as a number as well as a string.

Also variables defined outside the scope of a function are not normally accesible from within the function. You either need to pass the variable as a parameter or declare the variable as `const` to ensure that it can be accessed within a function. The reason for this is to allow future optimizations of invocations of functions in **Reflex** - where the function could actually be executed on a different machine than the one used for the outer script. You can see this in action in the script above with the `prefix` const.

8

BUILT-IN FUNCTIONS

Reflex has a large number of built-in functions that extend the power of the language in a more native way. The `println` function was introduced earlier. This section describes all of the built-in functions in **Reflex** .

println

```
println( expression )
```

The `println` function prints to the registered output handler the single parameter passed, which is coerced to a string type if it is not already. In most implementations of **Reflex** the output handler is wired to be either standard out (the console), the Eclipse console window or the standard log file.

```
1 // Println example
2 println("Hello, world!");
3 println(5);
4 println({}); // Prints an empty map
5 println("one two " + 3);
```

Listing 19: println

print

```
print( expression )
```

The `print` function is identical to `println` except that it does not automatically terminate the output with a carriage return.

```
1 // Print example
2 print("Hello, world!");
3 print(" And this would be on the same line.");
4 println(""); // And now force a carriage return
```

Listing 20: print

typeof

```
typeof( expression )
```

The `typeof` function can be used to determine the type of an expression, which can be a variable identifier as well. The return from the `typeof` function is a string, which can take the values in the table **??**.

Internal Type	Return Value
String	"string"
Number	"number"
Boolean	"bool"
List	"list"
Map	"map"
Date	"date"
Time	"time"
File	"file"
Queue	"queue"
No value	"void"
Null value	"null"
All else	"object"

Table 4.: typeof function return values

An example of the use of `typeof` function is shown below:

```
1 // typeof example
2 a = "This is a string";
3
4 if typeof(a) == "string" do
5    println("Yes, 'a' is a string");
6 end
```

Listing 21: Typeof example

assert

```
assert( boolean-expression )
```

The `assert` function is used to test its single parameter for truth. If the expression does not evaluate to true the **Reflex** script will abort abnormally.

```
1 // assert example
2
3 assert(true);
4 assert(typeof(" ") == "string");
```

Listing 22: Assert example

size

`size( list-expression | string-expression )`

The `size` function returns the size of its single parameter. It is only applicable for strings and lists. For a string the size is the length of the string, for a list it is the size of the list (the number of elements in the list). For convenience, `size(null)` evaluates to zero.

```
// size example
a = [1,2,3,4];

if sizeof(a) == 4 do
   println("Yes, that list has four elements");
end
```

Listing 23: Size example

keys

`keys( map-expression )`

The keys function takes a single map parameter, and returns a list of strings that corresponds to the keys of the associative map. It is useful when you need to iterate over a map.

```
// keys example

a = { 'one' : 1, 'two' : 2 };
b = keys(a);

for k in b do
   println("Key = " + k + ", value is " + b[k]);
end
```

Listing 24: Keys example

debug

`debug( expression )`

The debug function works in a similar way to the `println` function, except that the output is sent to any attached debugger instead of to the console. In some **Reflex** installations this will mean the same thing.

```
// debug example

```

```
println("This will appear in one place");
debug("This will appear in the debugger");
```

Listing 25: Debug example

date

```
date( )
date( string-expression )
```

The date function returns a date object. If called with zero parameters the object will be initialized to the current date. It can also take a single string parameter which must be a date formatted as "yyyyMMdd". The date object will be initialized to the date represented by that string.

```

today = date();
aRealDate = date('20120101');

println("Today is " + today + ", today is fun");
println("The start of the year 2012 is " + aRealDate);
```

Listing 26: Date example

time

```
time( )
time( string-expession )
```

The time function returns a time object. If called with zero parameters the object will be initialized to the current time. It can also take a single string parameter which must be a time formatted as "HH:mm:ss". The time object will be initialized to the time represented by that string.

```
// time example

now = time();
then = time('11:00:01');

println("What time is now? " + now);
```

Listing 27: Time example

readdir

```
readdir( string-expression | file-expression)
```

The `readdir` function returns the contents of a directory as a list of file values, although its behavior is really determined by the IO handler installed in the **Reflex** environment. The function accepts either a string (which corresponds to the name of a folder available to the handler) or a file (returned by the `file` function or a different call to `readdir`).

```
// readdir example
// Recursively look for folders

def readFolder(folder)
    println("Looking at " + folder);
    filesAndFolders = readdir(folder);
    for fAndF in filesAndFolders do
        if isfolder(fAndF) do
           readFolder(fAndF);
        end
    end
end

readFolder('/tmp');
```

Listing 28: readdir example

This example starts with the `/tmp` folder and enumerates all folders below that recursively, printing out the name of each folder found.

isfile

```
isfile( string-expression | file-expression )
```

The `isfile` function evaluates its single argument (which needs to be a file or a string) and returns a boolean indicating whether the argument is actually a file.

```
// isfile example

const name = '/tmp';

if isfile(name) do
   println(name + " is a file!");
else do
   println(name + " is not a file!");
end
```

Listing 29: IsFile example

isfolder

```
isfolder( string-expession | file-expression)
```

The `isfolder` function evaluates its single argument (which needs to be a file or a string) and returns a boolean indicating whether the argument is actually folder.

```
1
2  // isfolder example
3
4  const name = '/tmp/out.log';
5
6  if isfolder(name) do
7     println(name + " is a folder!");
8  else do
9     println(name + " is not a folder!");
10 end
```

Listing 30: IsFolder example

file

`file( string-expression )`

The `file` function creates a **Reflex** file object from a string, where the string is assumed to be an absolute reference to a real file or folder. Files can be read by the `pull` operator (<−−) and written to by the `push` operator (−− >).

```
1  // file example
2
3  a = "/tmp/test.txt";
4  data = "This is some text\n";
5
6  aFile = file(a);
7
8  data −−> aFile;
9
10 b = "/tmp/test.txt";
11 bFile = file(b);
12
13 data2 <−− bFile;
14
15 assert(data == data2);
```

Listing 31: File example

json

`json( map-expression )`

The `json` function converts a map into a JSON formatted string that represents the contents of that map.

```
// json example
a = { 'one' : 1, 'two' : 2 };

a1 = "" + a;
a2 = json(a);

assert(a1 == '{ one=1, two=2 }';
assert(a2 == '{ "one" : 1, "two" : 2 }';
```

Listing 32: Json example

Note that the default "string" representation of a map is not a json document, you must call the `json` function for this.

fromjson

```
map = fromjson( string-expression )
```

The `fromjson` function is the reverse of the `json` function. It takes a JSON formatted string and converts it to an associative map object.

```
// fromjson example
a = '{ "alpha" : 1, "beta", 2 };

b = fromjson(a);

assert(b['alpha'] == 1);
```

Listing 33: FromJson example

uuid

```
string = uuid( )
```

The `uuid` function generates a new unique string that can be used as a unique id.

```
// uuid example
a = uuid();
b = uuid();

assert(a != b);

println(a + " is not the same as " + b);
```

Listing 34: UUID example

que

```
queue = que( partition, name )
```

The que function defines a *queue* object that is bound to the given partition (a string, known to **Rapture**) and a name (the name of the queue in that partition). A queue can be used with the `push` and `pull` operators to send and receive messages to other queue participants. The implementation of a given queue is defined in the **Rapture** system.

It is normal to push a map object onto a queue and to receive a map object from a queue. When pulling from a queue the pull will eventually timeout and a `null` value will be returned which will need to be tested for.

Here is a set of scripts that show both sides of the same queue.

```
// Queue push example
const partition = "test";
const queueName = "thequeue";

q = que(partition, queueName);

message = {};
message['value'] = 42;

message ==> q;
```

Listing 35: Queue push example

```
// Queue pull example, mirroring the queue push example
const partition = "test";
const queueName = "thequeue";

q = que(partition, queueName);

cont = true;

while cont do
   message <== q;
   if message != null do
      println("Message was " + message['value']);
      cont = false;
   end
end
```

Listing 36: Queue pull example

wait

```
wait( string )
```

```
wait( string, int, int )
wait( process )
```

The `wait` function is a convenience function that waits for a document to exist in **Rapture** . The document name is provided in the first parameter and the optional second and third parameters control the retry interval (wait between checks) and retry count (how many times to check). The return value for the function is either the contents of the document (as a map) or null (if the document did not exist after the interval requested).

Finally, `wait` can also be used to wait on a process object returned by the `spawn` command.

```
// wait example

displayName = 'test/official/config/testData';

// Assume the above does not exist at the moment.

result = wait(displayName);

assert(result == null);

value = {};
value —> displayName;

result = wait(displayName);

assert(result == {});
```

Listing 37: Wait example

chain

```
result = chain( string-expression )
result = chain( string-expression, map-expression )
```

The `chain` function is a way of executing a second script in **Reflex** from a first script. The script is provided as the first string argument and can be passed in an optional parameter map as the second argument. The return value from `chain` is the return value from the called script.

```
// chain example
a = "println('The parameter is ' + p); return true;";

res = chain(a, { 'p' : 42 });

println(The result is " + res);
```

Listing 38: Chain example

The output from executing the script above would be:

```
The parameter is 42
The result is true
```

signal

```
signal( string-expression, map-expression )
```

The `signal` function is the mirror of the `wait` function in **Reflex** . The `signal` function creates a document in **Rapture** with the given displayname and value. It's really a synonym for `value --> displayName` .

```
1 // signal example
2
3 signal('test/official/config/doc', { 'hello' : 1 });
4
5 assert(wait('test/official/config/doc') == { 'hello' : 1 });
```

Listing 39: Signal example

sleep

```
sleep( int-expression )
```

The `sleep` function pauses the **Reflex** script for the number of milliseconds specified in the passed parameter.

```
1 // typeof example
2
3 for x = 0 to 10 do
4    sleep(100);
5    if null != wait('test/official/config/doc') do
6         x = 10;
7    end
8 end
```

Listing 40: Sleep example

rand

```
rand( number-expession )
```

The `rand` function returns an integer number between 0 and the passed parameter.

```
values = [];
for i = 1 to 10 do
   values = values + rand(10);
end

println("Here is a list of random numbers - " + values);
```

Listing 41: Rand example

spawn

```
spawn( list-expression )
spawn( list-expression, map-expression, file-expression)
```

The spawn command, where supported, provides a mechansim for spawning a child process. The return value is a special *process* object that can be used in a *pull* context (to retrieve the standard output from the process) and by the `wait` function to wait for it to finish.

The first parameter to the spawn command is a list of parameters to pass to the process. The first member of this list is the process to execute, the rest are parameters to pass to this process.

The second parameter is a map expression that defines the *environment* of the process.

The third parameter is a file object that defines the folder the process should be run in.

```
env = { "PATH" : "/bin" };
folder = file('/tmp');
program = [ '/bin/ls' , '-l' ];

p = spawn(program, env, folder);

wait(p);

out <-- p;

println("output from process is " + out);
```

Listing 42: Spawn example

defined

```
boolean = defined( identifier )
```

The `defined` function returns true if the variable identifier passed in is known to **Reflex** at this point.

```
a = "This is a string";

assert(defined(a) == true);
assert(defined(b) == false);
```

Listing 43: Defined example

round

```
integer = round( number-expression )
```

The round function takes a floating point number argument and returns an integer result that is the closest integer to that value.

```

a = 1.23;
b = 1.56;

assert(round(a) == 1);
assert(round(b) == 2);
```

Listing 44: Round example

lib

```
Library = lib( string-expression )
```

Reflex has the ability to embed *3rd* party code within the language. The definition of how to do this is defined in a later section, but the `lib` command is the way a *3rd* party library is linked in with **Reflex** . The string parameter to the `lib` function is the name of a loadable class that implements the `IReflexLibrary` interface.

The return value from this function is a special *library* object that can be used in the `call` function.

```

mylib = lib('rapture.addins.BloombergData');
```

Listing 45: Lib example

call

```
result = call( library-expression,
               string-expression,
               map-expression )
```

The `call` function takes a library loaded with the `lib` function and calls a function within that library. The function name is passed as the second parameter and any parameters to the internal function are passed in the third parameter. The result of calling the function is implementation specific.

```
1
2 mylib = lib('rapture.test');
3
4 result = call(mylib, 'testFn', { 'param' : 42 } );
```

Listing 46: Call example

template

```
result = template(string-expression, map-expression)
```

The `template` function takes a string "template" and applies parameters to that template to generate a resulting string where the variables in the template have been replaced with the value of the parameters. Internally **Reflex** uses the popular *stringtemplate* library for this task.

```
1
2 tmp = 'Hello <what>';
3 param = { 'what' : 'world' };
4
5 val = template(tmp, param);
6
7 println(val);
8
9 assert(val == 'Hello world');
```

Listing 47: Template example

cast

```
value = cast ( expression, string-expression )
```

Placeholder for cast description.

```
1 // typeof example
2 a = "This is a string";
3
4 if typeof(a) == "string" do
5    println("Yes, 'a' is a string");
6 end
```

Listing 48: Cast example

archive

```
value = archive( string-expression )
```

The `archive` command is used to create a special type of `file` object that tracks a ZIP archive. You can interact with the object in either read mode or write mode.

write mode

In write mode you use the push operator (`-->`) to send either a simple map to an entry in the file or a two element list - the first element being the name of the entry and the second element being the map data.

After all of the data has been "pushed" to the zip archive the file should be closed through the `close` function call.

A typical use of an archive is shown in the listing below:

```
1 arcFile = archive("test.zip");
2
3 dataEntry1 = { "dataField1" : 42, "data2" : "A string" };
4 dataEntry2 = { "dataField1" : 34, "data3" : "A different string"};
5
6 dataEntry1 --> arcFile;
7 ["DataEntryTwo", dataEntry2 ] --> arcFile;
8
9 close(arcFile);
```

Listing 49: Write to Archive example

In this example we create a zip file with two "files" - the first "file" has a default name and the value of the variable `dataEntry1`. The second entry has the name "DataEntryTwo" with the value of the variable `dataEntry2`. The `archive` command is useful for creating backups of large amounts of **Rapture** data.

read mode

In read mode you use the pull operator (`<--`) to retrieve data from the zip file, in the same order you pushed it on. The returned value is a map with two entries - a `data` entry contains the value of this file (its contents as a map) and the `displayName` entry contains the name of the entry. Reading the archive generated in the listing above is show in the example below:

```
1 arcFile = archive("test.zip");
2
3 dataRecord1 <-- arcFile;
4 dataRecord2 <-- arcFile;
5
6 close(arcFile);
7
8 println("First record data is " + dataRecord1['data']);
```

```
println("Second record data is " + dataRecord2['data']);
```

Listing 50: Read from archive example

Part II.

Appendices

9

BLOOMBERG PLUGIN

10

ANALYTICS PLUGIN

DISCLAIMER

INCAPTURE TECHNOLOGIES LLC
183 Madison Avenue, Suite 801
New York, NY 10016

RESEARCH BY
INCAPTURE