Ripple

make waves

A Tutorial Introduction

Team 5

Project Manager:	Amar Dhingra	**asd2157**
System Architect:	Alexander Roth	**air2112**
System Integrator:	Artur Renault	**aur2103**
Language Guru:	Spencer Brown	**stb2132**
Tester & Verifier:	Thomas Segarra	**tas2172**

2015 – 03 – 25

Contents

1 Introduction

Our introduction to Ripple is focused on getting you, the developer, quickly up to speed with the structure and usage of Ripple. While this introduction is not as thorough as the reference manual, we hope that after reading this you will be prepared to write clean reactive programs.

Before we begin, we need to answer the question: What is Ripple? Ripple is an imperative language that implements the reactive programming paradigm. A discussion on reactive programming, along with example use cases, can be found later in this tutorial.

Ripple closely resembles code from languages in the C family, but is more focused on one specific problem than its peers; Ripple focuses on the problem of creating programs that intuitively interact with dynamic data. As such, the most important aspect of Ripple to note from this tutorial is that:

1. Ripple includes `link` statements for linking variables, or "streams of variables" together.

Please consult the language reference manual for a more in-depth explanation of Ripple and an exhaustive discussion of its syntax.

2 Getting Started

The first program we build in Ripple will be a simple one: it prints

```
hello, world
```

and then terminates.

So let's get started! In broad pattern, you must first write the program in text (using exclusively ASCII characters), compile it successfully, and run it.

In Ripple, the program to print "hello, world" is:

```
#* A simple program to print hello world *#
void main(string[] args) {
    output(``hello, world\n''); # prints 'hello, world'
}
```

hello.rpl

A Ripple program must have a ".rpl" file extension (for example, `hello.rpl`) so that it can then be compiled with the command

```
rpl hello.rpl
```

As long as there are no compilation errors, the compiler will produce an executable file with the same name as the file, minus ".rpl". If you run `hello` by typing the command

```
./hello
```

This will print

```
hello, world
```

Now for some explanations about the program itself. A Ripple program, whatever its size, consists of *functions*, *variables*, and *whitespace*. A function contains *statements* that specify the computational operations to be done, and *variables* that store the values used during those computations. Whitespace consists of spaces, tabs, newlines, and comments. Note the unique commenting style in Ripple: a block comment begins with "#*" and ends with "*#", while single line comments begin with "##" and continue until the end of the line. Comments allow your program to be read by humans, but do not affect the execution of the program. In our example, we have a function named *main*, which does not return a value, as specified by the use of the `void` type in the function declaration. As in C, the "`main`" function is a unique function. Every Ripple program will begin execution with the `main` function. Thus, every program must include a `main`.

`main` often calls functions written elsewhere, either within the same file or in external files. One such function is used in our `main` function:

```
output(``hello, world\n'');
```

This `output` function prints the given argument to standard output. A function is called by naming it and giving it a parenthesized list of arguments. Thus, this program calls `output` with the argument ``` ``hello, world\n'' ```. `output` is a built-in function, so it is not necessary to `import` any outside library to use it.

A sequence of characters in double quotes, like ``` ``hello, world\n'' ```, is called a *string*. In our example, the string ``` ``hello, world\n'' ``` is an argument that is passed into the function `output`.

Note the use of the *newline* character (`\n` within the string in the `output` function call). Because a newline character cannot be written inline, it must be escaped using the same sequence as in C: a backslash followed by the character `n`. The characters immediately following a newline will appear at the leftmost position of the next line.

3 Variables and Types

3.1 Variables

If a Ripple program is to perform any amount of substantial work, it is necessary to have a construct that stores and accesses values at different times in the lifetime of the program: these are called variables. Like most languages, all variables are declared before use. Each declaration consists of a type name and variable name.

```
string word;
```

In this example, a variable is being declared as a `string` type with the variable name `word`.

The variable name can be as short as one letter or as long as you wish. Variable names cannot start with a number. Type names are predefined by the compiler (See § 3.2). A variable is *initialized* when you give it a type and give it a value.

Initialization is done with the `=` operator, and occurs after or with declaration.

```
string name = ``PLT'';
```

Here, a variable is being declared as a `string` type with the variable name `name` and it is being assigned (`=`) to the value of the *string literal* `PLT`.

3.2 Types

Types are paramount to the structure of Ripple programs. A type sets the behavior for any given variable within the lifetime of a program. Types are also used to ensure that a function returns the proper value on successful completion. As such, it is important that every variable and function within a program is declared with a type. Any function or variable without a type name in its declaration will throw a compiler error.

There are two distinct classes of types in Ripple:

The Basic (Built-in) Types

1. `void`
2. `bool`

3. `int`
4. `float`
5. `string`

The Derived Types

1. `dataset`
2. `array`

3.2.1 `void`

The `void` type is similar to C's or C++'s `void` type. Any function that is defined as a `void` function does not return any value. Note that `void` uniquely only be used with functions; it cannot be used with a variable name.

```
void func() {...}
```

The above function has the function name `func` and returns nothing, since its type is `void`.

3.2.2 `bool`

The `bool` type is a boolean value that can only take two values: `true` and `false`. A function can return a `bool` value and a variable can be set to a `bool` value. Unlike in some other languages, there are no other "truthy" or "falsy" values. A `bool` is explicitly only the value `true` or `false`, and only `bool`s are true or false.

```
bool alwaysTrue() { bool t = true; return t;}
```

Here the function named `alwaysTrue` states that its return type will be a `bool` value. Within the function body the `bool` value `t` is set to `true`. It is then returned to the function caller with the statement `return t`. `bool`s can be used in evaluations of logical expressions as well, such as equivalency or range checking. There is no good reason to write a function that returns `true` or `false` without some form of evaluation.

3.2.3 `byte`

A `byte` can be used to represent an actual byte (8 bits). The `byte` type is intended for low-level use, such as network programming, or for programs that involves a fine granularity on the data being used throughout the system. A function can return a `byte` and a variable can be declared as a `byte`. There are two ways to write a `byte` variable.

1. As a human-readable decimal value

   ```
   0b42
   ```

2. As a binary value

   ```
   0b00101010
   ```

`byte`s are unsigned, so the values can range from 0 to 255.

3.2.4 `int`

An int in Ripple is effectively C++'s long type, and thus its size is machine-dependent, but maintains the guarantee that it is at least 8 bytes. `int`s can thus represent numbers within the range $-2^{31}+1$ to $+2^{31}-1$, but this range may differ from machine to machine. Despite this, it is important to note that in Ripple, like most other languages, the int type floors. `int`s handle all the standard mathematical operations with the respective operator.

3.2.5 `float`

Floats are effectively equivalent to C++'s double type, as in Ripple float variables generally support a range of numbers from $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$, and are 64 bits in size. However, a `float` is ultimately machine dependent for their actual size and numbers they can represent. `float`s handle all the standard mathematical operations with the respective operator.

3.2.6 `string`

The `string` type is similar to Java's `String` object. It is an array of characters concatenated together, which is delimited by either single quotations (‘ ’) or double quotations (“ ”). A `string` can be of any length, with the smallest length being 0 (whose string is the empty string: ‘’). `string`s that are delimited with quotation marks are known as *string literals*, while variables declared with the type `string` are known as `string`s. Strings can be concatenated together using the `+` operator.

```
string prof = ‘‘Aho’’;
```

The variable `prof` is declared as a `string` with the string literal value of `‘‘Aho’’`.

3.2.7 `dataset`

The Ripple designers realize that it is not possible to provide every imaginable data type that the designer might require. Thus, the `dataset` type allows a Ripple developer to easily define their own data type, which are constructed from the other built-in data types. The `dataset` construct must be declared and implemented outside of any function implementation.

```
dataset team {
    string prof = ‘‘Aho’’;
    string mentor = ‘‘Chae’’;
    int team_sz = 5;
};
```

This `dataset` named `team` which contains two string variables

1. The `string` variable with the string literal value of `Aho`.
2. The `string` variable with the string literal value of `Chae`.

and one `int` variable named `team_sz` with the value of 5.

3.2.8 Arrays

An array is a dynamic data structure that allows you to hold multiple values of the same type. An array can hold any number of the same data types; the smallest array is the empty array – an array initialized with nothing.

Arrays have three built-in operations.

1. Concatenation of two or more variables together, which is supported through the use of the plus (`+`) operator, and
2. Length of an array, which is supported through the the use of the `@` operator.
3. An indexing operation, represented through the bracket (`[]`) operators. This operation retrieves the variable at the given address.

```
String[] mentors = { ``Aho'', ``Chae'' };
```

This statement creates a `string` array of size 2, named `mentors` with the initial values of `Aho` and `Chae`.

The statement

```
@mentors
```

will return the value of 2, which is the size of the `mentors` array.

The statement

```
mentors[0] + ``and '' + mentors[1]
```

returns a concatenation of three strings, which is `Aho and Chae`.

The statement

```
mentors[1]
```

returns the `string` at index 1, which will return `Chae`. Arrays are ordinal when accessing; that is, the first index is represented by 0 and the last index is $n-1$ for an n length array.

3.3 Example Program

Diving in, we will examine at a program that averages a set of grades and prints the result.

```
#* A simple calculation of averages *#
void main(string[] args) {

    float average;
    int floored_average;
    int total;

    int grades[] = {25, 30, 55, 75, 80, 80, 82, 85, 97, 100};
    total = 0;

    for(int i = 0; i < 10; i = i + 1){ # loop to total grades
        total = total + grades[i];
    }

    average = total / 10;
    floored_average = total // 10;

    output("Regular Average:\t", average);
    output("Integer Division Average:\t", floored_average);
}
```

avg.rpl

After compiling *avg.rpl*, we can run the program's executable and see the following result:

$ **rpl avg.rpl**

$ **./avg**

```
Regular Average:    70.9
Integer Division Average:    70
```

4 Control Flow

Similar to its predecessors, Ripple provides the developer with the standard `if`, `else`, `for` and `while` control flow statements. To make the transition to writing Ripple code easier these statements remain relatively unchanged from the predecessor languages.

4.1 `if` and `else` statements

`if` and `else` statements are used for events where the developer wants to test some condition and perform one stream of execution if the condition is true and some other stream of execution if the condition is false. The condition to be tested must be a boolean or an expression that evaluates to a boolean. The `else` statement after each `if` statement is optional.

An example is

```
if (x AND y) {
    # do something if both x and y are true
    } else {
    # do something if either x or y is false
}
```

4.2 `while` loop

In the event that a developer wants to execute a block of code repeatedly until some condition is met they would use a `while` loop. Developers provide the `while` loop with a boolean variable or statement to be evaluated. The boolean or statement is evaluated once at the start and once at the end of every loop and the loop is executed if it are true.

An example is

```
while ( x ) {
    # do something while x remains true
}
```

4.3 `for` loop

Equivalent in power to `while` loops, `for` loops provide a concise syntax to initialize a variable, test a condition and execute an expression in the statement declaration itself. The initialization is performed once, before the start of the loop. Next, the conditional expression is tested at the start of the loop and for every subsequent iteration. Finally, after each iteration, the end expression is executed.

An example is

```
for (int x = 0; x < 10; x = x + 1 ) {
    # do something while x is less than 10
}
```

4.4 `break` keyword

The `break` statement can be used to terminate a loop at any time during its execution. The program will continue execution at the instruction following the loop body.

An example is

```
while ( true ) {
    # breaking out of the infinite loop
    break;
}
```

4.5 `continue` keyword

The `continue` keyword causes the program to continue on to the next iteration of the loop. It starts by going back to the start of the loop, reevaluating the loop condition.

An example is

```
while ( true ) {
    # goes back to the start of the loop
    continue;
}
```

5 Functions and Arguments

Functions and their arguments are the bread and butter of any programming language, and are key to writing clean, concise code. Modularizing your code into functions is extremely important in Ripple in light of the nature of link statements, and the subsequent need to conceptualize the reactive nature of Ripple programs. In order for you to begin writing clean Ripple code, we will now discuss the nature of functions and their arguments in Ripple.

5.1 Functions

Mentioned at the start of this tutorial, but important to repeat again, is that all Ripple programs must contain a `main()` function of the type void, meaning that it returns nothing. This function will always be executed first within a Ripple program.

Moving on to a more general overview of functions, we begin by pointing out the nice feature in Ripple that allows functions to be used before being declared. Continuing, most functions in Ripple behave the same as C functions; however, several key differences exist. First and foremost, if a function is declared with the type void, it does not mean that it returns the type void, but merely that it returns nothing. Another key difference are those functions that are provided to link statements. These functions need to be declared with the link keyword. Additionally, a link function cannot be used for anything other than link statements. The declaration for this type of function would be `link type func_name(arg_list);`. Variables in link functions are passed by reference, meaning that when they change in the function, they are changed in the main thread of the program as well.

5.2 Arguments

Arguments are used to transfer data between functions. The parentheses after a function name can contain a list of arguments. The list of arguments within a function definition can be a variadic size (i.e. the list of arguments can be of an arbitrary length).

6 Scope

Scoping within Ripple is handled much like C's version of scoping; that is, *automatic* or *local* variables are created within a block and are removed after the block in which they were declared ends. This scoping rule also applies to any variables that are linked within a function.

Similarly, if two variables within a block have the same name a compiler error will be generated.

For example,

```
...
{
    int x = 4;
    int x = 5;
}
```

Within this code snippet, `x` is initialized twice: first to 4, then to 5. This programmer probably meant to change the value of the initial `x` to 5.

This can be accomplished with the following code:

```
...
{
    int x = 4;
    x = 5;
}
```

Here, `x` is only declared once and its value is changed to 5. This does not throw any errors since there is only one variable named `x` in this block.

Likewise, we cannot reference variables created in an inner block from an outer block.

7 Character I/O

Character input and output is an incredibly important piece of any programming language. You have already seen hints at how these concepts work in Ripple, but we will now go into a more in-depth discussion of Ripple's input and output.

We will begin with character output, which has already been covered to some degree. Outtputting strings is handled by the previously seen `output()` function. This function accepts any of the fundamental datatypes, excluding datasets, and will be convert these types to strings and print them. To output a dataset, one can print its constituent parts, or define a function that returns a string describing a given dataset. Additionally, the `output()` function will accept any number of arguments and concatenate them. Thus, the `output` function can either look like `output(arg0, arg1, arg2, ...` . This can also be accomplished by using the `+` operator, which also concatenates two strings: `output(arg0 + arg1 + arg2)`.

The `output()` function will return void. This is a distinct difference from the `input()` function which will return a string. `input()` takes newline delimited lines from standard input; if a file has been piped into a Ripple program, the `input()` function will return newline delimited strings from this file. Another aspect of the input function is that it accepts a string argument that acts as a prompt and will be printed to Standard Out.

8 File I/O

Ripple provides an easy to use File I/O library. The functions used for File I/O are `open()`, `close()`, `read()`, `readline()` and `write()`. Unlike FileStreamReaders which iterate through a file in a separate thread from that in which they are created, these functions run on the same thread in which they are called. All open files are represented and stored as integers. The functions are described underneath and explained with the help of a simple program that copies the contents of one file into another.

```
#**
 * Reads in a file line by line and writes the output to a second file and
 * prints out  the line with the line number.
 **#
void main(String[] args) {

    string input_filename = args[1];
    string output_filename = args[2];

    # opens the two files
    int infile = open(input_filename, 'r');
    int outfile = open(output_filename, 'w');

    # Iterate through the input file line by line
    for (String line = readline(infile); line != ""; line = readline(infile))
        # write each line to the output file
        write(outfile, line);

    # closing the open files
    close(infile);
    close(outfile);
}
```

count.rpl

The program begins by assigning the input and output filename strings which were provided as command line arguments to two variables. It then calls the `open()` function which takes as arguments the name of the file and the mode in which it is to be opened. The supported modes are

1. '`r`' - provides read-only access to the file
2. '`w`' - provides write-only access to the file, creating the file if it doesn't exist or scraps the file and creates a blank file if it exists
3. '`a`' - provides write only access to writing at the end of the file, creating the file if it doesn't already exist

Next the program calls the `readline()` function on the open input file in a loop ending when the file returns an empty line which it returns on `EOF`. The `readline()` function takes a single argument; the integer representing the open file. The `read()` function takes the same argument as `readline` but returns the entire file as a single string. Within the loop body the program writes the string read, calling the `write()` function which takes as its two arguments the integer representing an open file and a string to be written to the file.

Finally the program closes both files calling the `close()` function whose only argument is the file integer to close.

9 Reactive Programming

The defining feature of Ripple is its implementation of the reactive programming paradigm. **Note:** it must be emphasized that Ripple's reactive programming is *not* functional. Reactive programming, in its most abstract definition, is programming with asynchronous data streams. Another explanation could also be, it is a paradigm that relies on the propagation of change throughout variables of a program. These, however, are somewhat abstruse statements, especially for those who have never programmed before. The simplest way to express this paradigm is to think of an excel spreadsheet – a relatively painless visualization for propagation of changes exhibited in reactive programming.

In other words, within a spreadsheet you might have a cell, `A1`, set to the sum of the values of cells `B1` and `B2`, such as

```
A1 = B1 + B2
```

Furthermore, `B1` could be set to the values of cells `C1` and `C2`,

```
B1 = C1 + C2
```

In a spreadsheet, changing the value of `C2` would propagate through the rest of the sheet, changing the value of `B1`, and subsequently `A1`.

In a typical imperative language might have the statement `x = y + z` which sets `x` to the summation of the current values of `y` and `z`; if `y` or `z` change value after this assignment, `x` is not subsequently updated unless explicitly specified by the programmer.

Combining both the imperative and reactive programming paradigms, then, is where Ripple comes in. In Ripple, you are able to use the imperative paradigm to explicitly *link* variables that will then exhibit the reactive paradigm by propagating changes. The link statement, and all of its intricacies, is explored in depth in the following section, but hopefully this gives you a small idea of reactive programming.

10 The `link` Statement

A defining feature in Ripple that is not found in most programming languages is the functionality provided by the `link` keyword. The `link` keyword is what implements the reactive programming paradigm in Ripple. Let's look at the this keyword in action.

```
final float TEMP_CONV = 5//9;

#* prints Fahrenheit-Celsius temperature *#
void main(string[] args) {

    int deg_f = (int) args[1];
    link( int deg_c <- (deg_f - 32) * TEMP_CONV)
        output(‘‘Temp in F: ’’, deg_f, ‘‘\n’’,
               ‘‘Temp in C: ’’, deg_c, ‘‘\n’’);

    deg_f = 32;
}
```

temp1.rpl

After compiling *temp1.rpl*, we can run the program's executable and see the following results:

```
$ rpl temp1.rpl
$ ./temp1 50
Temp in F: 50
Temp in C: 10
Temp in F: 32
Temp in C: 0
```

On line 6, the program uses the `link` statement to connect the variable `deg_c` to the expression `(deg_f - 32) * TEMP_CONV`. By *linking* a variable to another variable, the program creates a dependency between the two variables, which is added to the program's `dependency tree`. This dependency tree is used by the compiler to update variables connected along the tree based upon the time they were linked to the previous variable in the tree. The dependency tree (Figure 1) depicts the flow of data from `deg_f` to `deg_c`.

Note how the arrow is unidirectional; that is, the updates do not flow in both directions. This decision was made intentionally to avoid the issue of cyclical dependencies. Dependency cycles should be avoided in Ripple at all costs, meaning the ordering of link statements is important.

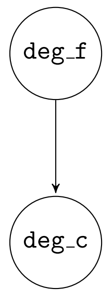

Figure 1: The dependency tree for *temp.rpl*

Suppose we have the following piece of code:

```
1 ...
2 int x = 10;
3 link( int y <- x + 2)
4 link ( int z <- y - 1)
5
6 link ( int q <- x - 3)
7 y = x + 7;
8 ...
```

link.rpl

The dependency tree for *link.rpl* is shown in Figure 2. As shown on line 3 through 4, there is a nested `link` statement within the initial `link` statement. This nesting is illustrated in the dependency tree by the addition of a new layer of nodes.

Let's walk through the program, starting on line 2, to see what's happening here.

Line 2 The variable `x` is initialized with the value of 10.

Line 3 The variable `y` is initialized and linked to `x` through the statement `x + 2`; the value of `y` becomes 12. This creates the root node `x` and the child node `y` and connects them on the dependency tree.

Line 4 The nested link statement initializes the integer variable `z` to the variable `y`; the value of `z` is 11. The node `z` is added to the dependency tree as a child of `y` and a link is made between them.

Line 7 The variable `q` is initialized as an integer linked to `x`; the value of `q` is 7. The dependency graph creates the node `q` as a child of `x` and creates a link between them.

Line 8 `y`'s value is changed to 17. This change will cause a cascade down to `z`, registering `z`'s new value as 16. However, the values of `x` and

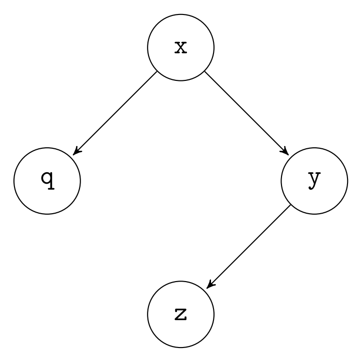

Figure 2: The dependency tree for *link.rpl*

`q` remains at 10 and 7 respectively because both `variables` are not linked to `y`.

Moving back to the `temp.rpl` example; on line 11, when `deg_f` is updated to the new value of 32, `deg_c` is notified of the change and updates its value accordingly. This update causes the change in value seen on the fourth line of output.

`link`s have the interesting property of being many-to-one; that is, we can have a variable link to a myriad of previously declared variables. When one of those parent variables is updated, the child will be updated accordingly. Again, the programmer must be careful in these cases not to create cycles in the dependency tree.

11 Stream Readers

`StreamReader`s are interesting consequences related to the `link` keyword within the Ripple language. The intuition behind their genesis is as follows: If the language is to be truly reactive, it would be necessary to have a construct that could handle live data coming from a variety of diverse sources. Thus, the `StreamReader` was born.

A `StreamReader` is fairly simple to implement within code itself, and has the ability to be extended for other types of media. Before we delve further into the implementation of `StreamReader`, let's look at a sample program.

A simple implementation of a `StreamReader` is a line counting program.

```
1  import ‘‘FileStreamReader’’;
2
3  #**
4  * Reads in a file and prints out the line with the line number.
5  **#
6
7  link void printLineNumber(int count, string line){
8      count = count + 1;
9      output(count, line, "\n");
10 }
11
12 void main(String[] args) {
13     String filename = args[1];
14
15     int count = 0;
16     link (String line <- FileStreamReader(filename))
17         printLineNumber(count, line);
18 }
```

count.rpl

Let's walk through the program to see how the `FileStreamReader` works.

Line 1 The `import` keyword tells the compiler that the program is using an external `StreamReader`. In this program, we are using the `FileStreamReader`, which provides a way to read through a file asynchronously line by line. This tool is an extension of of the generic `StreamReader` construct.

Line 7 The `link` keyword before the function indicates that `printLineNumber()` will be provided to a link statement.

Line 16 The `line` variable is initialized and linked to an `FileStreamReader` which is created by providing it with a filename to open and iterate through.

Line 17 The `printLineNumber()` function is called and passed the count function and line that was obtained from the `FileStreamReader`. All variables passed into functions provided to `link` statements are passed by value allowing them to be updated every time the `link` statement updates.

```
1  import ``XMLStreamReader'';
2
3  #**
4   * Prints Fahrenheit-Celsius temperature based on stream
5   * from www.weather.gov
6   **#
7  FINAL float TEMP_CONV = 9 // 5;
8  void main(String[] args) {
9
10     link(int deg_f <- (int) XMLStreamReader(``www.weather.gov'',
11                                             ``temp_farenheit'', 5))
12         output(``Temperature in F: '', deg_f, ``\n'');
13
14     link(int deg_c <- (deg_f - 32) * TEMP_CONV)
15         output(``Temp in C: '', deg_c, ``\n'');
16
17 }
```

temp3.rpl

For simplicity's sake, we shall assume that the stream coming from `http://www.weather.gov` is of a well-formatted XML file, with the tag "`farenheit`". If we run the program after compilation, we have the following output.

```
$ rpl temp2.rpl
  ./temp2
Temperature in F: 27
Temperature in C: -3
```

The dependency tree for temp3 is shown in Figure 3.

Let's walk through the program to see how `XMLStreamReader` is implemented.

Line 1 The `import` keyword tells the compiler that the program is using an external library. In this program, we are importing `XMLStreamReader`, which contains a `StreamReader` to read streams of XML files. This construct is an extension of the generic `StreamReader` construct.

Line 10 The `deg_f` variable is initialized and linked to an `XMLStreamReader` which is created by providing it with a URL, an XML tag to look for and a refresh time interval in seconds.

Line 12 The `output` statement outputs the value of `deg_f` to standard output. Since this function is provided to the link statement it is executed ever time the `XMLStreamReader` refreshes the feed.

Lines 14 and 15 As above, when the `deg_f` variable changes, the corresponding temperature in Celsius is calculated and printed.

12 Final Statements

Most modern programming languages have the concept of a read-only keyword that defines a value or construct at compile time and does not alter it at run time. Ripple is no exception. The keyword used for this read-only declaration is the `final` keyword followed by a construct declaration.

Similar to C's macros and Java's `final` variables, these `final` variables must be declared and initialized at the top of the file and cannot change throughout the life of the program. Therefore, it is best to use these constructs for readability and maintainability of the code base. For example, in the source code for *temp2.rpl*, the variable `TEMP_CONV` is used to permanently store the conversion rate from Fahrenheit to Celsius.

13 Conclusion

This, hopefully, has given a general overview of the basics of Ripple. Unfortunately, this tutorial cannot give a full and in-depth discussion of the subtler intricacies of Ripple, especially when dealing with `link` statements and `StreamReaders`. However, we believe that we have provided enough tools and a general understanding of the core aspects of Ripple that you may begin experimenting and writing your own code, and we encourage you to go out and do so.