A Swift Tutorial

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Introduction **1**

2 Hello World **1**

3 Language features **2**

3.1 Parameters . . . 2
3.2 Adding another application . . . 3
3.3 Anonymous files . . . 4
3.4 Datatypes . . . 4
3.5 Arrays . . . 5
3.6 Mappers . . . 5
3.6.1 The Regexp Mapper . . . 6
3.6.2 fixed_array_mapper . . . 6
3.7 foreach . . . 7
3.8 If . . . 7
3.9 Sequential iteration . . . 8

4 Runtime features **9**

4.1 Visualizing the workflow as a graph . . . 9
4.2 Running on a remote site . . . 9
4.3 Starting and restarting . . . 9

5 Bits **11**

5.1 Named and optional parameters . . . 11

1 Introduction

This is an introductory tutorial describing the use of Swift and its programming language SwiftScript. It is intended to introduce new users to the basics of Swift. It is structured as a series of simple exercises/examples which you can try for yourself as you read along.

For information on getting an installation of Swift running, consult the Swift Quickstart Guide. We advise you to install the latest stable release of Swift. Return to this document when you have successfully run the test SwiftScript program mentioned there.

There is also a Swift User Guide which contains a more detailed reference material on topics covered in this manual. All of the programs included in this tutorial can be found in your Swift distribution in the examples/tutorial directory.

2 Hello World

The first example program, hello.swift, outputs a hello world message into a file called hello.txt.

hello.swift

```
type messagefile;

app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}

messagefile outfile <"hello.txt">;

outfile = greeting();
```

To run hello.swift, change directories to the location of the script and run the swift command as follows.

> **Tip**
> Make sure the bin directory of your swift installation is in your PATH.

```
$ cd examples/tutorial
$ swift hello.swift
Swift svn swift-r3334 (swift modified locally) cog-r2752

RunID: 20100526-1925-8zjupq1b
Progress:
Final status:  Finished successfully:1
$ cat hello.txt
Hello, world!
```

The basic structure of this program is a type definition, an application procedure definition, a variable definition and then a call to the procedure.

First we define a new type, called messagefile. In this example, we will use this messagefile type for our output message.

```
type messagefile;
```

All data in SwiftScript must be typed, whether it is stored in memory or on disk. This example defines a very simple type. Later on we will see more complex type examples.

```
app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}
```

Next we define a procedure called greeting. This procedure will write out the "hello world" message to a file. To achieve this, it executes the unix utility *echo* with a parameter "Hello, world!" and directs the standard output into the output file.

The actual file to use is specified by the return parameter, t.

```
messagefile outfile <"hello.txt">;
```

Here we define a variable called outfile. The type of this variable is messagefile, and we specify that the contents of this variable will be stored on disk in a file called hello.txt

```
app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}
```

Now we call the greeting procedure, with its output going to the outfile variable and therefore to hello.txt on disk.

Over the following exercises, we'll extend this simple hello world program to demonstrate various features of Swift.

3 Language features

3.1 Parameters

Procedures can have parameters. Input parameters specify inputs to the procedure and output parameters specify outputs. Our hello world greeting procedure already uses an output parameter, t, which indicates where the greeting output will go. In this section, we will modify the previous script to add an input parameter to the greeting function.

parameter.swift

```
type messagefile;

app (messagefile t) greeting (string s) {
    echo s stdout=@filename(t);
}

messagefile outfile <"parameter.hello.txt">;
outfile = greeting("hello world");
```

We have modified the signature of the greeting procedure to indicate that it takes a single parameter, s, of type *string*.

We have modified the invocation of the *echo* utility so that it takes the value of s as a parameter, instead of the string literal "Hello, world!".

We have modified the output file definition to point to a different file on disk.

We have modified the invocation of greeting so that a greeting string is supplied.

The code for this section can be found in parameter.swift. It can be invoked using the swift command, with output appearing in parameter.hello.txt:

```
$ swift parameter.swift
```

Now that we can choose our greeting text, we can call the same procedure with different parameters to generate several output files with different greetings. The code is in manyparam.swift and can be run as before using the swift command.

manyparam.swift

```
type messagefile;

app (messagefile t) greeting (string s) {
    echo s stdout=@filename(t);
}

messagefile english <"manyparam.english.txt">;
messagefile french <"manyparam.french.txt">;
messagefile japanese <"manyparam.japanese.txt">;

english = greeting("hello");
french = greeting("bonjour");
japanese = greeting("konnichiwa");
```

Note that we can intermingle definitions of variables with invocations of procedures.

When this program runs, there should be three new files in the working directory (manyparam.english.txt, manyparam.francais.txt and manyparam.nihongo.txt) each containing a greeting in a different language.

In addition to specifying parameters positionally, parameters can be named, and if desired a default value can be specified.

3.2 Adding another application

Now we'll define a new application procedure. The procedure we define will capitalise all the words in the input file.

To do this, we'll use the unix tr (translate) utility. Here is an example of using tr on the unix command line, not using Swift:

```
$ echo hello | tr '[a-z]' '[A-Z]'
HELLO
```

There are two main steps - updating the transformation catalog, and updating the application block.

The transformation catalog lists where application executables are located on remote sites. We need to modify the transformation catalog to define a logical transformation for the tr utility. The transformation catalog can be found in etc/tc.data. There are already several entries specifying where executables can be found. Add a new line to the file, specifying where tr can be found (usually in /usr/bin/tr but it may differ on your system), like this:

```
localhost       tr      /usr/bin/tr     INSTALLED       INTEL32::LINUX  null
```

For now, ignore all of the fields except the second and the third. The second field tr specifies a logical application name and the third specifies the location of the application executable.

Now that we have defined where to find tr, we can use it in SwiftScript.

We can define a new procedure, capitalise, which calls tr.

```
app (messagefile o) capitalise(messagefile i) {
    tr "[a-z]" "[A-Z]" stdin=@filename(i) stdout=@filename(o);
}
```

We can call capitalise like this:

```
messagefile final <"capitalise.2.txt">;
hellofile = greeting("hello from Swift");
final = capitalise(hellofile);
```

Here is the full program based on this exercise:

capitalise.swift

```
type messagefile;

app (messagefile t) greeting (string s) {
    echo s stdout=@filename(t);
}

app (messagefile o) capitalise(messagefile i) {
    tr "[a-z]" "[A-Z]" stdin=@filename(i) stdout=@filename(o);
}

messagefile hellofile <"capitalise.1.txt">;
messagefile final <"capitalise.2.txt">;
hellofile = greeting("hello from Swift");
final = capitalise(hellofile);
```

Next, run swift and verify the output is correct.

```
$ swift capitalise.swift
...
$ cat capitalise.2.txt
HELLO FROM SWIFT
```

3.3 Anonymous files

In the previous section, the file hello.txt is used only to store an intermediate result. We don't really care about which name is used for the file, and we can let Swift choose the name.

To do that, omit the mapping entirely when declaring hellofile:

```
messagefile hellofile;
```

Swift will choose a filename, which in the present version will be in a subdirectory called _concurrent.

3.4 Datatypes

All data in variables and files has a data type. So far, we've seen two types:

- string - this is a built-in type for storing strings of text in memory, much like in other programming languages
- messagefile - this is a user-defined type used to mark disc resident files as containing messages

SwiftScript has the additional built-in types: boolean, integer and float that function much like their counterparts in other programming languages.

It is also possible to create user defined types with more structure, for example:

```
type details {
    string name;
    int pies;
}
```

Each element of the structured type can be accessed using a . like this:

```
person.name = "John";
```

The following complete program, types.swift, outputs a greeting using a user-defined structure type to hold parameters for the message:

types.swift

```
type messagefile;

type details {
    string name;
    int pies;
}

app (messagefile t) greeting (details d) {
    echo "Hello. Your name is" d.name "and you have eaten" d.pies "pies." stdout= ↩
        @filename(t);
}

details person;

person.name = "John";
person.pies = 3;

messagefile outfile <"types.pies.txt">;

outfile = greeting(person);
```

Structured types can be comprised of marker types for files. See the later section on mappers for more information about this.

3.5 Arrays

We can define arrays using the [] suffix in a variable declaration:

```
string words[] = ["how","are","you"];
```

This program, arrays.swift, will declare an array of message files.

arrays.swift

```
type messagefile;

app (messagefile t) greeting (string s[]) {
     echo s[0] s[1] s[2] stdout=@filename(t);
}

messagefile outfile <"arrays.txt">;

string words[] = ["how","are","you"];

outfile = greeting(words);
```

Observe that the type of the parameter to greeting is now an array of strings, string s[], instead of a single string, string s, that elements of the array can be referenced numerically, for example s[0], and that the array is initialised using an array literal, ["how","are","you"].

3.6 Mappers

A significant difference between SwiftScript and other languages is that data can be referred to on disk through variables in a very similar fashion to data in memory. For example, in the above examples we have seen a variable definition like this:

```
messagefile outfile <"arrays.txt">;
```

This means that outfile is a dataset variable, which is mapped to a file on disk called arrays.txt. This variable can be assigned to using = in a similar fashion to an in-memory variable. We can say that outfile is mapped onto the disk file arrays.txt by a mapper.

There are various ways of mapping in SwiftScript. Two forms of mapping, simple named mapping and anonymous mapping, have already been seen in this tutorial. Later exercises will introduce more forms.

In simple named mapping, the name of the file that a variable is mapped to is explictly listed.

```
messagefile outfile <"hello.txt">;
```

This is useful when you want to explicitly name input and output files for your program. An example of this can be seen with *outfile* in the hello world exercise.

With anonymous mapping no name is specified in the source code. A name is automatically generated for the file. This is useful for intermediate files that are only referenced through SwiftScript. A variable declaration is mapped anonymously by ommitting any mapper definition.

```
messagefile hellofile;
```

Later exercises will introduce other ways of mapping from disk files to SwiftScript variables.

3.6.1 The Regexp Mapper

In this exercise, we introduce the regexp mapper. This mapper transforms a string expression using a regular expression, and uses the result of that transformation as the filename to map.

regexp.swift demonstrates the use of this by placing output into a file that is based on the name of the input file. Our input file is mapped to the inputfile variable using the simple named mapper, then we use the regular expression mapper to map the output file. We then use the countwords() procedure to count the words in the input file and store the result in the output file. In order for the countwords() procedure to work correctly, add the wc utility (usually found in /usr/bin/wc) to tc.data.

The following program replaces the suffix of the input file (regexp_mapper.words.txt) with a new suffix (.count) to create regexp_mapper.words.count.

regexp_mapper.swift

```
type messagefile;
type countfile;

app (countfile t) countwords (messagefile f) {
        wc "-w" @filename(f) stdout=@filename(t);
}

messagefile inputfile <"regexp_mapper.words.txt">;

countfile c <regexp_mapper;
             source=@inputfile,
             match="(.*)txt",
             transform="\\1count">;

c = countwords(inputfile);
```

3.6.2 fixed_array_mapper

The fixed array mapper maps a list of files into an array. Each element of the array is mapped into one file in the specified directory. See fixed_array_mapper.swift below.

fixed_array_mapper.swift

```
type messagefile;
type countfile;

app (countfile t) countwords (messagefile f) {
   wc "-w" @filename(f) stdout=@filename(t);
}

string inputNames = "fixed_array_mapper.1.txt fixed_array_mapper.2.txt  ↩
    fixed_array_mapper.3.txt";
string outputNames = "fixed_array_mapper.1.count fixed_array_mapper.2.count  ↩
    fixed_array_mapper.3.count";

messagefile inputfiles[] <fixed_array_mapper;files=inputNames>;
countfile outputfiles[] <fixed_array_mapper;files=outputNames>;

outputfiles[0] = countwords(inputfiles[0]);
outputfiles[1] = countwords(inputfiles[1]);
outputfiles[2] = countwords(inputfiles[2]);
```

3.7 foreach

SwiftScript provides a control structure, foreach, to operate on each element of an array in parallel.

In this example, we will run the previous word counting example over each file in an array without having to explicitly list the array elements. The source code for this example is in foreach.swift. This program uses three input files: foreach.1.txt, foreach.2.txt, and foreach.3.txt. After you have run the workflow, you should see that there are three output files: foreach.1.count, foreach.2.count and foreach.3.count, each containing the word count for the corresponding input file. We combine the use of the fixed_array_mapper and the regexp_mapper.

foreach.swift

```
type messagefile;
type countfile;

app (countfile t) countwords (messagefile f) {
     wc "-w" @filename(f) stdout=@filename(t);
}

string inputNames = "foreach.1.txt foreach.2.txt foreach.3.txt";

messagefile inputfiles[] <fixed_array_mapper;files=inputNames>;

foreach f in inputfiles {
  countfile c<regexp_mapper;
            source=@f,
            match="(.*)txt",
            transform="\\1count">;
  c = countwords(f);
}
```

3.8 If

Decisions can be made using *if*, like this:

```
if(morning) {
  outfile = greeting("good morning");
} else {
  outfile = greeting("good afternoon");
}
```

if.swift contains a simple example of this. Compile and run if.swift and see that it outputs "good morning". Changing the morning variable from true to false will cause the program to output "good afternoon". Here is the contents of the full script:

if.swift

```
type messagefile;

app (messagefile t) greeting (string s) {
   echo s stdout=@filename(t);
}

messagefile outfile <"if.txt">;

boolean morning = true;

if(morning) {
  outfile = greeting("good morning");
} else {
  outfile = greeting("good afternoon");
}
```

3.9 Sequential iteration

A serial execution of instructions can be carried out using the sequential iteration construct. Iterate expressions allow a block of code to be evaluated repeatedly, with an integer parameter sweeping upwards from 0 until a termination condition holds.

The general form is:

```
iterate var {
    statements;
} until (terminationExpression);
```

The following example demonstrates one simple application. We will use iterate to set the value of i from 0 to 4. We will then use i as an index to sequentially print the values of an array.

Here's the program:

sequential_iteration.swift

```
string alphabet[];
alphabet[0] = "a";
alphabet[1] = "b";
alphabet[2] = "c";
alphabet[3] = "d";
alphabet[4] = "e";

iterate i
{
   tracef("Letter %i is: %s\n", i, alphabet[i]);
}  until(i == 5);
```

You should see a result similar to this:

```
Swift trunk swift-r5746 cog-r3370

RunID: 20120417-0820-g1q1m8b3
Progress:  time: Tue, 17 Apr 2012 08:20:22 -0500
Letter 0 is: a
Letter 1 is: b
Letter 2 is: c
Letter 3 is: d
Letter 4 is: e
Final status: Tue, 17 Apr 2012 08:20:22 -0500
```

4 Runtime features

4.1 Visualizing the workflow as a graph

When running a workflow, its possible to generate a provenance graph at the same time:

```
$ swift -pgraph graph.dot first.swift
$ dot -ograph.png -Tpng graph.dot
```

graph.png can then be viewed using your favourite image viewer. The dot application is part of the graphViz project. More information can be found at http://www.graphviz.org.

4.2 Running on a remote site

As configured by default, all jobs are run locally. In the previous examples, we've invoked echo and tr executables from our SwiftScript program. These have been run on the local system (the same computer on which you ran swift). We can also make our computations run on a remote resource. For more information on running Swift on a remote site please see the Site Configuration Guide.

4.3 Starting and restarting

Now we're going to try out the restart capabilities of Swift. We will make a workflow that will deliberately fail, and then we will fix the problem so that Swift can continue with the workflow.

First we have the program in working form, restart.swift.

restart.swift

```
type file;

app (file f) touch() {
    touch @f;
}

app (file f) processL(file inp) {
    echo "processL" stdout=@f;
}

app (file f) processR(file inp) {
    broken "process" stdout=@f;
}

app (file f) join(file left, file right) {
    echo "join" @left @right stdout=@f;
}

file f = touch();

file g = processL(f);
file h = processR(f);

file i = join(g,h);
```

We must define some transformation catalog entries:

```
localhost   touch   /usr/bin/touch  INSTALLED   INTEL32::LINUX  null
localhost   broken  /bin/true   INSTALLED   INTEL32::LINUX  null
```

Now we can run the program:

```
$ swift restart.swift
Swift 0.9 swift-r2860 cog-r2388

RunID: 20100526-1119-3kgzzi15
Progress:
Final status:  Finished successfully:4
```

Four jobs run - touch, echo, broken and a final echo. (note that broken isn't actually broken yet).

Now we will break the broken job and see what happens. Replace the definition in tc.data for broken with this:

```
localhost    broken     /bin/false   INSTALLED       INTEL32::LINUX  null
```

Now when we run the workflow, the broken task fails:

```
$ swift restart.swift

Swift 0.9 swift-r2860 cog-r2388

RunID: 20100526-1121-tssdcljg
Progress:
Progress:  Stage in:1  Finished successfully:2
Execution failed:
    Exception in broken:
Arguments: [process]
Host: localhost
Directory: restart-20100526-1121-tssdcljg/jobs/1/broken-1i6ufisj
stderr.txt:
stdout.txt:
```

From the output we can see that touch and the first echo completed, but then broken failed and so swift did not attempt to execute the final echo.

There will be a restart log with the same name as the RunID:

```
$ ls *20100526-1121-tssdcljg*rlog
restart-20100526-1121-tssdcljg.0.rlog
```

This restart log contains enough information for swift to know which parts of the workflow were executed successfully.

We can try to rerun it immediately, like this:

```
$ swift -resume restart-20100526-1121-tssdcljg.0.rlog restart.swift

Swift 0.9 swift-r2860 cog-r2388

RunID: 20100526-1125-7yx0zi6d
Progress:
Execution failed:
    Exception in broken:
Arguments: [process]
Host: localhost
Directory: restart-20100526-1125-7yx0zi6d/jobs/m/broken-msn1gisj
stderr.txt:
stdout.txt:

Caused by:
    Exit code 1
```

Swift tried to resume the workflow by executing "broken" again. It did not try to run the touch or first echo jobs, because the restart log says that they do not need to be executed again.

Broken failed again, leaving the original restart log in place.

Now we will fix the problem with "broken" by restoring the original tc.data line that works.

Remove the existing "broken" line and replace it with the successful tc.data entry above:

```
localhost       broken          /bin/true   INSTALLED       INTEL32::LINUX  null
```

Now run again:

```
$ swift -resume restart-20100526-1121-tssdcljg.0.rlog restart.swift

Swift 0.9 swift-r2860 cog-r2388

RunID: 20100526-1128-a2gfuxhg
Progress:
Final status:  Initializing:2  Finished successfully:2
```

Swift tries to run "broken" again. This time it works, and so Swift continues on to execute the final piece of the workflow as if nothing had ever gone wrong.

5 Bits

5.1 Named and optional parameters

In addition to specifying parameters positionally, parameters can be named, and if desired a default value can be specified:

default.swift

```
type file;

// s has a default value
app (file t) echo (string s="hello world") {
        echo s stdout=@filename(t);
}

file hw1<"default.1.txt">;
file hw2<"default.2.txt">;

// procedure call using the default value
hw1 = echo();

// using a different value
hw2 = echo(s="hello again");
```