
A Swift Tutorial

Abstract

This is an introductory tutorial on the use of Swift and its programming language SwiftScript.
\$LastChangedRevision: 2964 \$

Table of Contents

1. Introduction	1
2. Hello World	1
3. Language features	3
4. Runtime features	10
5. bits	16

1. Introduction

This tutorial is intended to introduce new users to the basics of Swift. It is structured as a series of small exercise/examples which you can try for yourself as you read along. After the first 'hello world' example, there are two tracks - the language track (which introduces features of the SwiftScript language) and the runtime track (which introduces features of the Swift runtime environment, such as running jobs on different sites)

For information on getting an installation of Swift running, consult the [Swift Quickstart Guide](#)¹, and return to this document when you have successfully run the test SwiftScript program mentioned there.

There is also a [Swift User's Guide](#)² which contains more detailed reference material on topics covered in this manual.

2. Hello World

The first example program (found in the file `examples/swift/first.swift`) outputs a hello world message into a file called `hello.txt`.

```
type messagefile;

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

messagefile outfile <"hello.txt">;

outfile = greeting();
```

¹ <http://www.ci.uchicago.edu/swift/guides/quickstartguide.php>
² <http://www.ci.uchicago.edu/swift/guides/userguide.php>

We can run this program as follows:

```
$ cd examples/swift/  
$ swift first.swift  
Swift v0.2  
  
RunID: elbupgygrzn12  
echo started  
echo completed  
  
$ 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:

```
type messagefile;
```

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

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.

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

Next we define a procedure called write. 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

```
outfile = greeting();
```

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 helloworld greeting procedure already uses an output parameter, `t`, which indicates where the greeting output will go. In this section, we will add an input parameter to the greeting function.

The code changes from `first.swift` are highlighted below.

```
type messagefile;

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

messagefile outfile <"hello2.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 `hello2.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.

```
type messagefile;

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

messagefile english <"english.txt">;
messagefile french <"francais.txt">;

english = greeting("hello");
french = greeting("bonjour");

messagefile japanese <"nihongo.txt">;
```

```
japanese = greeting("konnichiwa");
```

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

When this program has been run, there should be three new files in the working directory (english.txt, francais.txt and 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 - see [Named and optional parameters](#).

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 several steps:

- transformation catalog
- application block

First we need to modify the *transformation catalog* to define a logical transformation for the tc utility. The transformation catalog can be found in `etc/tc.data`. There is already one entry specifying where **echo** 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:

```
local      translate  /usr/bin/tr  INSTALLED INTEL32::LINUX null
```

For now, ignore all of the fields except the second and the third. The second field 'translate' 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 transform.

```
(messagefile t) capitalise (messagefile f) {
  app {
    translate "[a-z]" "[A-Z]" stdin=@filename(f) stdout=@filename(t);
  }
}
```

We can call **capitalise** like this:

```
cfile = capitalise(outfile);
```

So a full program based on the first exercise might look like this:

```
type messagefile;

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

(messagefile t) capitalise (messagefile f) {
    app {
        translate "[a-z]" "[A-Z]" stdin=@filename(f) stdout=@filename(t);
    }
}

messagefile outfile <"greeting.txt">;
messagefile cfile <"capitalised.txt">;

outfile = greeting("hello from Swift");
cfile = capitalise(outfile);
```

We can use the swift command to run this:

```
$ swift t.swift
[...]
$ cat capitalised.txt
HELLO FROM SWIFT
```

3.3. Anonymous files

In the previous section, the file `greeting.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 `outfile`:

```
messagefile outfile;
```

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 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;
    string place;
}
```

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

```
person.name = "john";
```

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

```
type messagefile;

type details {
    string name;
    string place;
}

(messagefile t) greeting (details d) {
    app {
        echo "Hello" d.name "You live in" d.place stdout=@filename(t);
    }
}

details person;

person.name = "John";
person.place = "Namibia";

messagefile outfile <"types.txt">;

outfile = greeting(person);
```

Structured types can comprised 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:

```
messagefile m[];
```

This will declare an array of message files.

```
type messagefile;

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

messagefile outfile <"q5out.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 <"q13greeting.txt">;
```

This means that 'outfile' is a dataset variable, which is mapped to a file on disk called 'q13greeting.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 'q13greeting.txt' by a *mapper*.

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

The two forms of mapping seen so far are:

simple named mapping - the name of the file that a variable is mapped to is explicitly listed. Like this:

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

This is useful when you want to explicitly name input and output files for your program. For example, 'outfile' in exercise HELLOWORLD.

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, such as 'outfile' in exercise ANONYMOUSFILE. A variable declaration is mapped anonymously by omitting any mapper definition, like this:

```
messagefile outfile;
```

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

TODO: introduce `@v` syntax.

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, and then we use the regular expression mapper to map the output file. Then we use the `countwords()` procedure that we defined in exercise AN-

OTHERPROCEDURE to count the works in the input file and store the result in the output file.

The important bit of `regexp.swift` is:

```
messagefile inputfile <"q16.txt">;

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

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 `fixedarray.swift`.

```
string inputNames = "one.txt two.txt three.txt";
string outputNames = "one.count two.count three.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 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`. The three input files (`one.txt`, `two.txt` and `three.txt`) are supplied. After you have run the workflow, you should see that there are three output files (`one.count`, `two.count` and `three.count`) each containing the word count for the corresponding input file. We combine the use of the `fixed_array_mapper` and the `regexp_mapper`.

```
string inputNames = "one.txt two.txt three.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'.

3.9. Sequential iteration

A development version of Swift after 0.2 (revision 1230) introduces a sequential iteration construct.

The following example demonstrates a simple application: each step of the iteration is a string representation of the byte count of the previous step's output, with iteration terminating when the byte count reaches zero.

Here's the program:

```
type counterfile;

(counterfile t) echo(string m) {
  app {
    echo m stdout=@filename(t);
  }
}

(counterfile t) countstep(counterfile i) {
  app {
    wcl @filename(i) @filename(t);
  }
}

counterfile a[] <simple_mapper;prefix="foldout">;

a[0] = echo("793578934574893");

iterate v {
  a[v+1] = countstep(a[v]);
  trace("extract int value ",@extractint(a[v+1]));
} until (@extractint(a[v+1]) <= 1);
```

echo is the standard unix echo.

wcl is our application code - it counts the number of bytes in the one file and writes that count out to another, like this:

```
$ cat ../wcl
#!/bin/bash
echo -n $(wc -c < $1) > $2

$ echo -n hello > a
$ wcl a b
$ cat b
5
```

Install the above wcl script somewhere and add a transformation catalog entry for it. Then run the example program like this:

```
$ swift fold9.swift
Swift v0.2-dev r1230

RunID: 20070918-1434-16bt8x4a
echo started
echo completed
wcl started
extract int value 16.0
wcl completed
wcl started
extract int value 2.0
wcl completed
wcl started
extract int value 1.0
wcl completed

$ ls foldout.*
foldout.0 foldout.1 foldout.2 foldout.3
```

4. Runtime features

4.1. Visualising 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 graph1.dot
```

which can then be viewed using your favourite image viewer.

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.

WARNING: This example is necessarily more vague than previous examples, because its requires access to remote resources. You should ensure that you can submit a job using the globus-job-run (or globusrun-ws?) command(s).

We do not need to modify any SwiftScript code to run on another resource. Instead, we must modify another catalog, the 'site catalog'. This catalog provides details of the location that applications will be run, with the default settings referring to the local machine. We will modify it to refer to a remote resource - the UC Teraport cluster. If you are not a UC Teraport user, you should use details of a different resource that you do have access to.

The site catalog is located in etc/sites.xml and is a relatively straightforward XML format file. We must modify each of the following three settings: gridftp (which indicates how and where data can be transferred to the remote resource), jobmanager (which indicates how applications can be run on the remote resource) and workdirectory (which indicates where working storage can be found on the remote resource).

4.3. Writing a mapper

This section will introduce writing a custom mapper so that Swift is able to access data files laid out in application-spe-

cific ways.

An application-specific mapper must take the form of a Java class that implements the [Mapper3](#) interface.

Usually you don't need to implement this interface directly, because Swift provides a number of more concrete classes with some functionality already implemented.

The hierarchy of helper classes is:

[Mapper4](#) - This is the abstract interface for mappers in Swift. You must implement methods to provide access to mapper properties, to map from a SwiftScript dataset path (such as foo[1].bar) to a file name, to check whether a file exists. None of the default Swift mappers implement this interface directly - instead they use one of the following helper classes.

[AbstractMapper5](#) - This provides helper methods to manage mapper properties and to handle existence checking. Examples of mappers which use this class are: [array_mapper6](#), [csv_mapper7](#), [fixed_array_mapper8](#), [regex_mapper9](#) and [single_file_mapper10](#).

[AbstractFileMapper11](#) - This provides a helper class for mappers which select files based on selecting files from a directory listing. It is necessary to write some helper methods that are different from the above mapper methods. Examples of mappers which use this class are: [simple_mapper12](#), [filesystem_mapper13](#) and the (undocumented) `Structure-dRegularExpressionMapper`.

In general, to write a mapper, choose either the `AbstractMapper` or the `AbstractFileMapper` and extend those. If your mapper will generally select the files it returns based on a directory listing and will convert paths to filenames using some regular conversion (for example, in the way that `simple_mapper` maps files in a directory that match a particular pattern), then you should probably use the `AbstractFileMapper`. If your mapper will produce a list of files in some other way (for example, in the way that `csv_mapper` maps based on filenames given in a CSV file rather than looking at which files are in a directory), then you should probably use the `AbstractMapper`.

4.3.1. Writing a very basic mapper

In this section, we will write a very basic (almost useless) mapper that will map a SwiftScript dataset into a hard-coded file called `myfile.txt`, like this:

```
Swift variable                               Filename
var <----->                               myfile.txt
```

We should be able to use the mapper we write in a SwiftScript program like this:

```
type file;
file f <my_first_mapper>;
```

3 <http://www.ci.uchicago.edu/swift/javadoc/vdsk/org/griphyn/vdl/mapping/Mapper.html>
4 <http://www.ci.uchicago.edu/swift/javadoc/vdsk/org/griphyn/vdl/mapping/Mapper.html>
5 <http://www.ci.uchicago.edu/swift/javadoc/vdsk/org/griphyn/vdl/mapping/AbstractMapper.html>
6 http://www.ci.uchicago.edu/swift/guides/userguide.php#mapper.array_mapper
7 http://www.ci.uchicago.edu/swift/guides/userguide.php#mapper.csv_mapper
8 http://www.ci.uchicago.edu/swift/guides/userguide.php#mapper.fixed_array_mapper
9 http://www.ci.uchicago.edu/swift/guides/userguide.php#mapper.regex_mapper
10 http://www.ci.uchicago.edu/swift/guides/userguide.php#mapper.single_file_mapper
11 <http://www.ci.uchicago.edu/swift/javadoc/vdsk/org/griphyn/vdl/mapping/file/AbstractFileMapper.html>
12 http://www.ci.uchicago.edu/swift/guides/userguide.php#mapper.simple_mapper
13 http://www.ci.uchicago.edu/swift/guides/userguide.php#mapper.filesys_mapper

First we must choose a base class - `AbstractMapper` or `AbstractFileMapper`. We aren't going to use a directory listing to decide on our mapping - we are getting the mapping from some other source (in fact, it will be hard coded). So we will use `AbstractMapper`.

So now onto the source code. We must define a subclass of `AbstractMapper` and implement several mapper methods: `isStatic`, `existing`, and `map`. These methods are documented in the javadoc for the `Mapper` interface.

Here is the code implementing this mapper. Put this in your source `vdsk` directory, make a directory `src/tutorial/` and put this file in `src/tutorial/MyFirstMapper.java`

```
package tutorial;

import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;

import org.griphyn.vdl.mapping.AbsFile;
import org.griphyn.vdl.mapping.AbstractMapper;
import org.griphyn.vdl.mapping.Path;
import org.griphyn.vdl.mapping.PhysicalFormat;

public class MyFirstMapper extends AbstractMapper {

    AbsFile myfile = new AbsFile("myfile.txt");

    public boolean isStatic() {
        return false;
    }

    public Collection existing() {
        if (myfile.exists())
            return Arrays.asList(new Path[] {Path.EMPTY_PATH});
        else
            return Collections.EMPTY_LIST;
    }

    public PhysicalFormat map(Path p) {
        if(p.equals(Path.EMPTY_PATH))
            return myfile;
        else
            return null;
    }
}
```

Now we need to inform the Swift engine about the existence of this mapper. We do that by editing the `MapperFactory` class definition, in `src/org/griphyn/vdl/mapping/MapperFactory.java` and adding a `registerMapper` call alongside the existing `registerMapper` calls, like this:

```
registerMapper("my_first_mapper", tutorial.MyFirstMapper.class);
```

The first parameter is the name of the mapper that will be used in `SwiftScript` program. The second parameter is the new `Mapper` class that we just wrote.

Now rebuild Swift using the 'ant `redist`' target.

This new Swift build will be aware of your new mapper. We can test it out with a hello world program:

```
type messagefile;
(messagefile t) greeting() {
    app {
        echo "hello" stdout=@filename(t);
    }
}
messagefile outfile <my_first_mapper>;
outfile = greeting();
```

Run this program, and hopefully you will find the "hello" string has been output into the hard coded output file `myfile.txt`:

```
$ cat myfile.txt
hello
```

So that's a first very simple mapper implemented. Compare the source code to the `single_file_mapper` in [Single-FileMapper.java](#)¹⁴. There is not much more code to the `single_file_mapper` - mostly code to deal with the file parameter.

4.4. 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`.

```
type file;
(file f) touch() {
    app {
        touch @f;
    }
}
(file f) processL(file inp) {
    app {
        echo "processL" stdout=@f;
    }
}
(file f) processR(file inp) {
    app {
        broken "process" stdout=@f;
    }
}
(file f) join(file left, file right) {
    app {
        echo "join" @left @right stdout=@f;
    }
}
```

¹⁴ <http://www.ci.uchicago.edu/trac/swift/browser/trunk/src/org/griphyn/vdl/mapping/file/SingleFileMapper.java>

```
}  
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 v0.1-dev  
RunID: php8y7ydim8x0  
touch started  
echo started  
broken started  
touch completed  
broken completed  
echo completed  
echo started  
echo completed
```

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 v0.1-dev  
RunID: 6y3urvnm5kch1  
touch started  
broken started  
touch completed  
echo started  
echo completed  
broken failed  
The following errors have occurred:  
1. Application echo not executed due to errors in dependencies  
2. Application "broken" failed (Job failed with an exit code of 1)  
   Arguments: "process"  
   Host: localhost  
   Directory: restart-6y3urvnm5kch1/broken-4empvyci
```

```
STDERR:  
STDOUT:
```

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 *6y3urvnm5kch1*rlog  
restart-6y3urvnm5kch1.0.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-6y3urvnm5kch1.0.rlog restart.swift  
  
Swift v0.1-dev  
  
RunID: nyrg0squeudnul  
broken started  
broken failed  
The following errors have occurred:  
1. Application echo not executed due to errors in dependencies  
2. Application "broken" failed (Job failed with an exit code of 1)  
   Arguments: "process"  
   Host: localhost  
   Directory: restart-nyrg0squeudnul/broken-i2guyvci  
   STDERR:  
   STDOUT:
```

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-6y3urvnm5kch1.0.rlog restart.swift  
  
Swift v0.1-dev  
  
RunID: x0318zci03i11  
broken started  
echo started  
broken completed  
echo completed
```

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:

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

When we invoke the procedure, we can specify values for the parameters by name:

```
french = greeting(s="bonjour");
```

or we can let the default value apply:

```
english = greeting();
```