# Swift User Guide

**REVISION HISTORY**

| NUMBER | DATE | DESCRIPTION | NAME |
|---|---|---|---|
| | | | |

# Contents

# 1 Overview

Swift is a data-flow oriented coarse grained scripting language that supports dataset typing and mapping, dataset iteration, conditional branching, and procedural composition.

Swift programs (or workflows) are written in a language called Swift.

Swift scripts are primarily concerned with processing (possibly large) collections of data files, by invoking programs to do that processing. Swift handles execution of such programs on remote sites by choosing sites, handling the staging of input and output files to and from the chosen sites and remote execution of programs.

# 2 Getting Started

This section will provide links and information to new Swift users about how to get started using Swift.

## 2.1 Quickstart

This section provides the basic steps for downloading and installing Swift.

- Swift requires that a recent version of Oracle Java is installed. More information about installing Java can be found at http://www.oracle.com/technetwork/java.
- Download Swift 0.95 at http://swiftlang.org/packages/swift-0.95.tar.gz.
- Extract by running "tar xfz swift-0.95.tar.gz"
- Add Swift to $PATH by running "export PATH=$PATH:/path/to/swift-0.95/bin"
- Verify swift is working by running "swift -version"

## 2.2 Tutorials

There are a few tutorials available for specific clusters and supercomputers.

Swift on Clouds and Ad Hoc collections of workstations

Swift on OSG Connect

Swift on Crays

Swift on RCC Midway Cluster at UChicago / Slurm

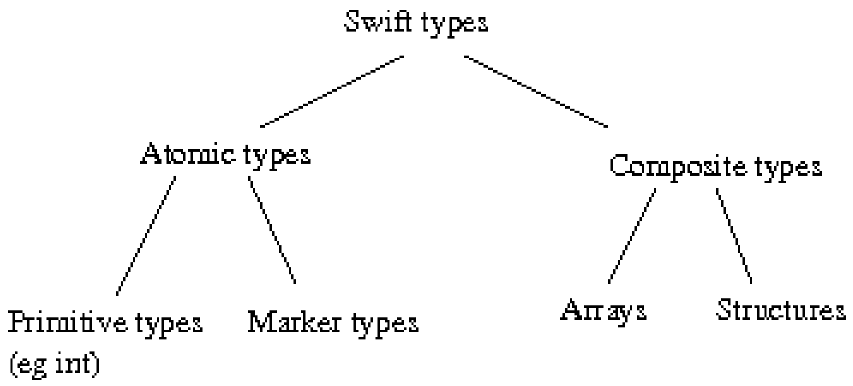
# 3 The Swift Language

## 3.1 Language Basics

A Swift script describes data, application components, invocations of applications components, and the inter-relations (data flow) between those invocations.

Data is represented in a script by strongly-typed single-assignment variables. The syntax superficially resembles C and Java. For example, { and } characters are used to enclose blocks of statements.

Types in Swift can be atomic or composite. An atomic type can be either a primitive type or a mapped type. Swift provides a fixed set of primitive types, such as integer and string. A mapped type indicates that the actual data does not reside in CPU addressable memory (as it would in conventional programming languages), but in POSIX-like files. Composite types are further subdivided into structures and arrays. Structures are similar in most respects to structure types in other languages. In Swift, structures are defined using the *type* keyword (there is no struct keyword). Arrays use numeric indices, but are sparse. They can contain elements of any type, including other array types, but all elements in an array must be of the same type. We often refer to instances of composites of mapped types as datasets.



Atomic types such as string, int, float and double work the same way as in C-like programming languages. A variable of such atomic types can be defined as follows:

```
string astring = "hello";
```

A struct variable is defined using the *type* keyword as discussed above. Following is an example of a variable holding employee data:

```
type Employee{
    string name;
    int id;
    string loc;
}
```

The members of the structure defined above can be accessed using the dot notation. An example of a variable of type Employee is as follows:

```
Employee emp;
emp.name="Thomas";
emp.id=2222;
emp.loc="Chicago";
```

Arrays of structures are allowed in Swift. A convenient way of populating structures and arrays of structures is to use the *readData()* function.

Mapped type and composite type variable declarations can be annotated with a mapping descriptor indicating the file(s) that make up that dataset. For example, the following line declares a variable named photo with type image. It additionally declares that the data for this variable is stored in a single file named shane.jpg.

```
image photo <"shane.jpg">;
```

Component programs of scripts are declared in an app declaration, with the description of the command line syntax for that program and a list of input and output data. An app block describes a functional/dataflow style interface to imperative components.

For example, the following example lists a procedure which makes use of the ImageMagick http://www.imagemagick.org/- convert command to rotate a supplied image by a specified angle:

```
app (image output) rotate(image input) {
  convert "-rotate" angle @input @output;
}
```

A procedure is invoked using the familiar syntax:

```
rotated = rotate(photo, 180);
```

While this looks like an assignment, the actual unix level execution consists of invoking the command line specified in the app declaration, with variables on the left of the assignment bound to the output parameters, and variables to the right of the procedure invocation passed as inputs.

The examples above have used the type image without any definition of that type. We can declare it as a marker type which has no structure exposed to Swift script:

```
type image;
```

This does not indicate that the data is unstructured; but it indicates that the structure of the data is not exposed to Swift. Instead, Swift will treat variables of this type as individual opaque files.

With mechanisms to declare types, map variables to data files, and declare and invoke procedures, we can build a complete (albeit simple) script:

```
type image;
image photo <"shane.jpg">;
image rotated <"rotated.jpg">;

app (image output) rotate(image input, int angle) {
   convert "-rotate" angle @input @output;
}

rotated = rotate(photo, 180);
```

This script can be invoked from the command line:

```
  $ ls *.jpg
  shane.jpg
  $ swift example.swift
  ...
  $ ls *.jpg
  shane.jpg rotated.jpg
```

This executes a single convert command, hiding from the user features such as remote multisite execution and fault tolerance that will be discussed in a later section.

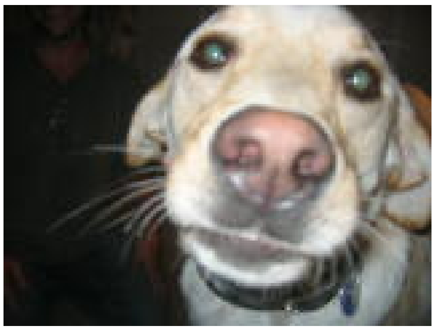


**Figure 1. shane.jpg**

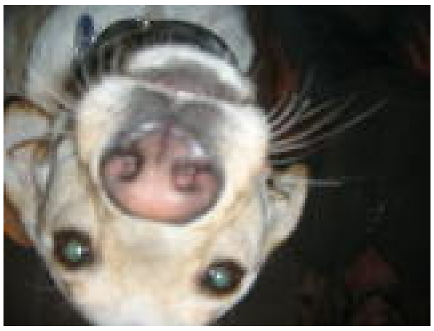


**Figure 2. rotated.jpg**

## 3.2 Arrays and Parallel Execution

Arrays of values can be declared using the [] suffix. Following is an example of an array of strings:

```
string pets[] = ["shane", "noddy", "leo"];
```

An array may be mapped to a collection of files, one element per file, by using a different form of mapping expression. For example, the filesys_mapper maps all files matching a particular unix glob pattern into an array:

```
file frames[] <filesys_mapper; pattern="*.jpg">;
```

The foreach construct can be used to apply the same block of code to each element of an array:

```
foreach f,ix in frames {
  output[ix] = rotate(f, 180);
```

Sequential iteration can be expressed using the iterate construct:

```
step[0] = initialCondition();
iterate ix {
  step[ix] = simulate(step[ix-1]);
}
```

This fragment will initialise the 0-th element of the step array to some initial condition, and then repeatedly run the simulate procedure, using each execution's outputs as input to the next step.

## 3.3 Associative Arrays

By default, array keys are integers. However, other primitive types are also allowed as array keys. The syntax for declaring an array with a key type different than the default is:

```
<valueType>[<keyType>] array;
```

For example, the following code declares and assigns items to an array with string keys and float values:

```
float[string] a;
a["one"] = 0.2;
a["two"] = 0.4;
```

In addition to primitive types, a special type named **auto** can be used to declare an array for which an additional **append** operation is available:

```
int[auto] array;

foreach i in [1:100] {
  array << (i*2) ;
}

foreach v in array {
  trace(v);
}
```

Items in an array with **auto** keys cannot be accessed directly using a primitive type. The following example results in a compile-time error:

```
int[auto] array;
array[0] = 1;
```

However, it is possible to use **auto** key values from one array to access another:

```
int[auto] a;
int[auto] b;

a << 1;
a << 2;

foreach v, k in a {
  b[k] = a[k] * 2;
}
```

### 3.4 Ordering of execution

Non-array variables are single-assignment, which means that they must be assigned to exactly one value during execution. A procedure or expression will be executed when all of its input parameters have been assigned values. As a result of such execution, more variables may become assigned, possibly allowing further parts of the script to execute.

In this way, scripts are implicitly parallel. Aside from serialisation implied by these dataflow dependencies, execution of component programs can proceed in parallel.

In this fragment, execution of procedures p and q can happen in parallel:

```
y=p(x);
z=q(x);
```

while in this fragment, execution is serialised by the variable y, with procedure p executing before q.

```
y=p(x);
z=q(y);
```

Arrays in Swift are more monotonic - a generalisation of being assignment. Knowledge about the content of an array increases during execution, but cannot otherwise change. Each element of the array is itself single assignment or monotonic (depending on its type). During a run all values for an array are eventually known, and that array is regarded as closed.

Statements which deal with the array as a whole will often wait for the array to be closed before executing (thus, a closed array is the equivalent of a non-array type being assigned). However, a foreach statement will apply its body to elements of an array as they become known. It will not wait until the array is closed.

Consider this script:

```
file a[];
file b[];
foreach v,i in a {
  b[i] = p(v);
}
a[0] = r();
a[1] = s();
```

Initially, the foreach statement will have nothing to execute, as the array a has not been assigned any values. The procedures r and s will execute. As soon as either of them is finished, the corresponding invocation of procedure p will occur. After both r and s have completed, the array a will be closed since no other statements in the script make an assignment to a.

### 3.5 Compound procedures

As with many other programming languages, procedures consisting of Swift script can be defined. These differ from the previously mentioned procedures declared with the app keyword, as they invoke other Swift procedures rather than a component program.

```
(file output) process (file input) {
  file intermediate;
  intermediate = first(input);
  output = second(intermediate);
}

file x <"x.txt">;
file y <"y.txt">;
y = process(x);
```

This will invoke two procedures, with an intermediate data file named anonymously connecting the first and second procedures.

Ordering of execution is generally determined by execution of app procedures, not by any containing compound procedures. In this code block:

```
(file a, file b) A() {
  a = A1();
  b = A2();
}
file x, y, s, t;
(x,y) = A();
s = S(x);
t = S(y);
```

then a valid execution order is: A1 S(x) A2 S(y). The compound procedure A does not have to have fully completed for its return values to be used by subsequent statements.

## 3.6 More about types

Each variable and procedure parameter in Swift script is strongly typed. Types are used to structure data, to aid in debugging and checking program correctness and to influence how Swift interacts with data.

The image type declared in previous examples is a marker type. Marker types indicate that data for a variable is stored in a single file with no further structure exposed at the Swift script level.

Arrays have been mentioned above, in the arrays section. A code block may be applied to each element of an array using foreach; or individual elements may be references using [] notation.

There are a number of primitive types:

| **type** | **contains** |
|---|---|
| int | integers |
| string | strings of text |
| float | floating point numbers, that behave the same as Java doubles |
| boolean | true/false |

Complex types may be defined using the type keyword:

```
type headerfile;
type voxelfile;
type volume {
  headerfile h;
  voxelfile v;
}
```

Members of a complex type can be accessed using the . operator:

```
volume brain;
```

```
o = p(brain.h);
```

Sometimes data may be stored in a form that does not fit with Swift's file-and-site model; for example, data might be stored in an RDBMS on some database server. In that case, a variable can be declared to have external type. This indicates that Swift should use the variable to determine execution dependency, but should not attempt other data management; for example, it will not perform any form of data stage-in or stage-out it will not manage local data caches on sites; and it will not enforce component program atomicity on data output. This can add substantial responsibility to component programs, in exchange for allowing arbitrary data storage and access methods to be plugged in to scripts.

```
type file;

app (external o) populateDatabase() {
  populationProgram;
}

app (file o) analyseDatabase(external i) {
  analysisProgram @o;
}

external database;
file result <"results.txt">;

database = populateDatabase();
result = analyseDatabase(database);
```

Some external database is represented by the database variable. The populateDatabase procedure populates the database with some data, and the analyseDatabase procedure performs some subsequent analysis on that database. The declaration of database contains no mapping; and the procedures which use database do not reference them in any way; the description of database is entirely outside of the script. The single assignment and execution ordering rules will still apply though; populateDatabase will always be run before analyseDatabase.

### 3.7 Data model

Data processed by Swift is strongly typed. It may be take the form of values in memory or as out-of-core files on disk. Language constructs called mappers specify how each piece of data is stored.

### 3.8 More technical details about Swift script

The syntax of Swift script has a superficial resemblance to C and Java. For example, { and } characters are used to enclose blocks of statements.

A Swift script consists of a number of statements. Statements may declare types, procedures and variables, assign values to variables, and express operations over arrays.

### 3.9 Variables

Variables in Swift scripts are declared to be of a specific type. Assignments to those variables must be data of that type. Swift script variables are single-assignment - a value may be assigned to a variable at most once. This assignment can happen at declaration time or later on in execution. When an attempt to read from a variable that has not yet been assigned is made, the code performing the read is suspended until that variable has been written to. This forms the basis for Swift's ability to parallelise execution - all code will execute in parallel unless there are variables shared between the code that cause sequencing.

### 3.10 Variable Declarations

Variable declaration statements declare new variables. They can optionally assign a value to them or map those variables to on-disk files.

Declaration statements have the general form:

```
typename variablename (<mapping> | = initialValue ) ;
```

The format of the mapping expression is defined in the Mappers section. initialValue may be either an expression or a procedure call that returns a single value.

Variables can also be declared in a multivalued-procedure statement, described in another section.

## 3.11 Assignment Statements

Assignment statements assign values to previously declared variables. Assignments may only be made to variables that have not already been assigned. Assignment statements have the general form:

```
variable = value;
```

where value can be either an expression or a procedure call that returns a single value.

Variables can also be assigned in a multivalued-procedure statement, described in another section.

## 3.12 Procedures

There are two kinds of procedure: An atomic procedure, which describes how an external program can be executed; and compound procedures which consist of a sequence of Swift script statements.

A procedure declaration defines the name of a procedure and its input and output parameters. Swift script procedures can take multiple inputs and produce multiple outputs. Inputs are specified to the right of the function name, and outputs are specified to the left. For example:

```
(type3 out1, type4 out2) myproc (type1 in1, type2 in2)
```

The above example declares a procedure called myproc, which has two inputs in1 (of type type1) and in2 (of type type2) and two outputs out1 (of type type3) and out2 (of type type4).

A procedure input parameter can be an optional parameter in which case it must be declared with a default value. When calling a procedure, both positional parameter and named parameter passings can be passed, provided that all optional parameters are declared after the required parameters and any optional parameter is bound using keyword parameter passing. For example, if myproc1 is defined as:

```
(binaryfile bf) myproc1 (int i, string s="foo")
```

Then that procedure can be called like this, omitting the optional

```
parameter s:
binaryfile mybf = myproc1(1);
```

or like this supplying a value for the optional parameter s:

```
binaryfile mybf = myproc1 (1, s="bar");
```

### 3.12.1 Atomic procedures

An atomic procedure specifies how to invoke an external executable program, and how logical data types are mapped to command line arguments.

Atomic procedures are defined with the app keyword:

```
app (binaryfile bf) myproc (int i, string s="foo") {
    myapp i s @filename(bf);
}
```

which specifies that myproc invokes an executable called myapp, passing the values of i, s and the filename of bf as command line arguments.

### 3.12.2 Compound procedures

A compound procedure contains a set of Swift script statements:

```
(type2 b) foo_bar (type1 a) {
    type3 c;
    c = foo(a);    // c holds the result of foo
    b = bar(c);    // c is an input to bar
}
```

## 3.13 Control Constructs

Swift script provides if, switch, foreach, and iterate constructs, with syntax and semantics similar to comparable constructs in other high-level languages.

### 3.13.1 foreach

The foreach construct is used to apply a block of statements to each element in an array. For example:

```
check_order (file a[]) {
    foreach f in a {
        compute(f);
    }
}
```

foreach statements have the general form:

```
foreach controlvariable (,index) in expression {
    statements
}
```

The block of statements is evaluated once for each element in expression which must be an array, with controlvariable set to the corresponding element and index (if specified) set to the integer position in the array that is being iterated over.

### 3.13.2 if

The if statement allows one of two blocks of statements to be executed, based on a boolean predicate. if statements generally have the form:

```
if(predicate) {
    statements
} else {
    statements
}
```

where predicate is a boolean expression.

### 3.13.3 switch

switch expressions allow one of a selection of blocks to be chosen based on the value of a numerical control expression. switch statements take the general form:

```
switch(controlExpression) {
    case n1:
        statements2
    case n2:
        statements2
    [...]
    default:
        statements
}
```

The control expression is evaluated, the resulting numerical value used to select a corresponding case, and the statements belonging to that case block are evaluated. If no case corresponds, then the statements belonging to the default block are evaluated.

Unlike C or Java switch statements, execution does not fall through to subsequent case blocks, and no break statement is necessary at the end of each block.

Following is an example of a switch expression in Swift:

```
int score=60;
switch (score){
case 100:
    tracef("%s\n", "Bravo!");
case 90:
    tracef("%s\n", "very good");
case 80:
    tracef("%s\n", "good");
case 70:
    tracef("%s\n", "fair");
default:
    tracef("%s\n", "unknown grade");
    }
```

#### 3.13.4 iterate

iterate expressions allow a block of code to be evaluated repeatedly, with an iteration variable being incremented after each iteration.

The general form is:

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

Here *var* is the iteration variable. Its initial value is 0. After each iteration, but before *terminationExpression* is evaluated, the iteration variable is incremented. This means that if the termination expression is a function of only the iteration variable, the body will never be executed while the termination expression is true.

Example:

```
iterate i {
    trace(i); // will print 0, 1, and 2
} until (i == 3);
```

Variables declared inside the body of *iterate* can be used in the termination expression. However, their values will reflect the values calculated as part of the last invocation of the body, and may not reflect the incremented value of the iteration variable:

```
iterate i {
    trace(i);
    int j = i; // will print 0, 1, 2, and 3
} until (j == 3);
```

### 3.14 Operators

The following infix operators are available for use in Swift script expressions.

| operator | purpose |
|---|---|
| + | numeric addition; string concatenation |
| - | numeric subtraction |
| * | numeric multiplication |
| / | floating point division |
| %/ | integer division |
| %% | integer remainder of division |
| == != | comparison and not-equal-to |
| < > ⇐ >= | numerical ordering |
| && \|\| | boolean and, or |
| ! | boolean not |

## 3.15 Global constants

At the top level of a Swift script program, the global modified may be added to a declaration so that it is visible throughout the program, rather than only at the top level of the program. This allows global constants (of any type) to be defined.

## 3.16 Imports

The import directive can be used to import definitions from another Swift file.

For example, a Swift script might contain this:

```
import "defs";
file f;
```

which would import the content of defs.swift:

```
type file;
```

Imported files are read from two places. They are either read from the path that is specified from the import command, such as:

```
import "definitions/file/defs";
```

or they are read from the environment variable SWIFT_LIB. This environment variable is used just like the PATH environment variable. For example, if the command below was issued to the bash shell:

```
export SWIFT_LIB=${HOME}/Swift/defs:${HOME}/Swift/functions
```

then the import command will check for the file defs.swift in both "${HOME}/Swift/defs" and "${HOME}/Swift/functions" first before trying the path that was specified in the import command.

Other valid imports:

```
import "../functions/func"
import "/home/user/Swift/definitions/defs"
```

There is no requirement that a module is imported only once. If a module is imported multiple times, for example in different files, then Swift will only process the imports once.

Imports may contain anything that is valid in a Swift script, including the code that causes remote execution.

## 3.17 Mappers

Mappers provide a mechanism to specify the layout of mapped datasets on disk. This is needed when Swift must access files to transfer them to remote sites for execution or to pass to applications.

Swift provides a number of mappers that are useful in common cases. This section details those mappers. For more complex cases, it is possible to write application-specific mappers in Java and use them within a Swift script.

### 3.17.1 The Single File Mapper

The single_file_mapper maps a single physical file to a dataset.

| Swift variable | Filename |
|---|---|
| f | myfile |
| f [0] | INVALID |
| f.bar | INVALID |

| parameter | meaning |
|---|---|
| file | The location of the physical file including path and file name. |

Example:

```
file f <single_file_mapper;file="plot_outfile_param">;
```

There is a simplified syntax for this mapper:

```
file f <"plot_outfile_param">;
```

### 3.17.2 The Simple Mapper

The simple_mapper maps a file or a list of files into an array by prefix, suffix, and pattern. If more than one file is matched, each of the file names will be mapped as a subelement of the dataset.

| Parameter | Meaning |
|---|---|
| location | A directory that the files are located. |
| prefix | The prefix of the files |
| suffix | The suffix of the files, for instance: ".txt" |
| padding | The number of digits used to uniquely identify the mapped file. This is an optional parameter which defaults to 4. |
| pattern | A UNIX glob style pattern, for instance: "*foo*" would match all file names that contain foo. When this mapper is used to specify output filenames, pattern is ignored. |

```
type file;
file f <simple_mapper;prefix="foo", suffix=".txt">;
```

The above maps all filenames that start with foo and have an extension .txt into file f.

| Swift variable | Filename |
|---|---|
| f | foo.txt |

```
type messagefile;

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

messagefile outfile <simple_mapper;prefix="foo",suffix=".txt">;
```

```
outfile = greeting("hi");
```

This will output the string *hi* to the file foo.txt.

The simple_mapper can be used to map arrays. It will map the array index into the filename between the prefix and suffix.

```
type messagefile;

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

messagefile outfile[] <simple_mapper;prefix="baz",suffix=".txt", padding=2>;

outfile[0] = greeting("hello");
outfile[1] = greeting("middle");
outfile[2] = greeting("goodbye");
```

| **Swift variable** | **Filename** |
|---|---|
| outfile[0] | baz00.txt |
| outfile[1] | baz01.txt |
| outfile[2] | baz02.txt |

simple_mapper can be used to map structures. It will map the name of the structure member into the filename, between the prefix and the suffix.

```
type messagefile;

type mystruct {
  messagefile left;
  messagefile right;
};

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

mystruct out <simple_mapper;prefix="qux",suffix=".txt">;

out.left = greeting("hello");
out.right = greeting("goodbye");
```

This will output the string "hello" into the file qux.left.txt and the string "goodbye" into the file qux.right.txt.

| **Swift variable** | **Filename** |
|---|---|
| out.left | quxleft.txt |
| out.right | quxright.txt |

### 3.17.3 Concurrent Mapper

The concurrent_mapper is almost the same as the simple mapper, except that it is used to map an output file, and the filename generated will contain an extract sequence that is unique. This mapper is the default mapper for variables when no mapper is specified.

| Parameter | Meaning |
|---|---|
| location | A directory that the files are located. |
| prefix | The prefix of the files |
| suffix | The suffix of the files, for instance: ".txt" pattern A UNIX glob style pattern, for instance: "*foo*" would match all file names that contain foo. When this mapper is used to specify output filenames, pattern is ignored. |

Example:

```
file f1;
file f2 <concurrent_mapper;prefix="foo", suffix=".txt">;
```

The above example would use concurrent mapper for f1 and f2, and generate f2 filename with prefix "foo" and extension ".txt"

### 3.17.4 Filesystem Mapper

The filesys_mapper is similar to the simple mapper, but maps a file or a list of files to an array. Each of the filename is mapped as an element in the array. The order of files in the resulting array is not defined.

TODO: note on difference between location as a relative vs absolute path w.r.t. staging to remote location - as mihael said: It's because you specify that location in the mapper. Try location="." instead of location="/sandbox/. . . "

| parameter | meaning |
|---|---|
| location | The directory where the files are located. |
| prefix | The prefix of the files |
| suffix | The suffix of the files, for instance: ".txt" |
| pattern | A UNIX glob style pattern, for instance: "*foo*" would match all file names that contain foo. |

Example:

```
file texts[] <filesys_mapper;prefix="foo", suffix=".txt">;
```

The above example would map all filenames that start with "foo" and have an extension ".txt" into the array texts. For example, if the specified directory contains files: foo1.txt, footest.txt, foo__1.txt, then the mapping might be:

| Swift variable | Filename |
|---|---|
| texts[0] | footest.txt |
| texts[1] | foo1.txt |
| texts[2] | foo__1.txt |

### 3.17.5 Fixed Array Mapper

The fixed_array_mapper maps from a string that contains a list of filenames into a file array.

| parameter | Meaning |
|---|---|
| files | A string that contains a list of filenames, separated by space, comma or colon |

Example:

```
file texts[] <fixed_array_mapper;files="file1.txt, fileB.txt, file3.txt">;
```

would cause a mapping like this:

| Swift variable | Filename |
|---|---|
| texts[0] | file1.txt |
| texts[1] | fileB.txt |
| texts[2] | file3.txt |

### 3.17.6 Array Mapper

The array_mapper maps from an array of strings into a file

| parameter | meaning |
|---|---|
| files | An array of strings containing one filename per element |

Example:

```
string s[] = [ "a.txt", "b.txt", "c.txt" ];

file f[] <array_mapper;files=s>;
```

This will establish the mapping:

| Swift variable | Filename |
|---|---|
| f[0] | a.txt |
| f[1] | b.txt |
| f[2] | c.txt |

### 3.17.7 Regular Expression Mapper

The regexp_mapper transforms one file name to another using regular expression matching.

| parameter | meaning |
|---|---|
| source | The source file name |
| match | Regular expression pattern to match, use |
| () | to match whatever regular expression is inside the parentheses, and indicate the start and end of a group; the contents of a group can be retrieved with the |
| \\number | special sequence (two backslashes are needed because the backslash is an escape sequence introducer) |
| transform | The pattern of the file name to transform to, use \number to reference the group matched. |

Example:

```
file s <"picture.gif">;
file f <regexp_mapper; source=s,
  match="(.*)gif", transform="\\1jpg">;
```

This example transforms a file ending gif into one ending jpg and maps that to a file.

| Swift variable | Filename |
|---|---|
| f | picture.jpg |

### 3.17.8 Structured Regular Expression Mapper

The structured_regexp_mapper is similar to the regexp_mapper with the only difference that it can be applied to arrays while the regexp_mapper cannot.

| **parameter** | **meaning** |
|---|---|
| source | The source file name |
| match | Regular expression pattern to match, use |
| () | to match whatever regular expression is inside the parentheses, and indicate the start and end of a group; the contents of a group can be retrieved with the |
| \\number | special sequence (two backslashes are needed because the backslash is an escape sequence introducer) |
| transform | The pattern of the file name to transform to, use \number to reference the group matched. |

Example:

```
file s[] <filesys_mapper; pattern="*.gif">;

file f[] <structured_regexp_mapper; source=s,
          match="(.*)gif", transform="\\1jpg">;
```

This example transforms all files in a list that end in gif to end in jpg and maps the list to those files.

### 3.17.9 CSV Mapper

The csv_mapper maps the content of a CSV (comma-separated value) file into an array of structures. The dataset type needs to be correctly defined to conform to the column names in the file. For instance, if the file contains columns: name age GPA then the type needs to have member elements like this:

```
type student {
  file name;
  file age;
  file GPA;
}
```

If the file does not contain a header with column info, then the column names are assumed as column1, column2, etc.

| **Parameter** | **Meaning** |
|---|---|
| file | The name of the CSV file to read mappings from. |
| header | Whether the file has a line describing header info; default is |
| true | |
| skip | The number of lines to skip at the beginning (after header line); default is 0. |
| hdelim | Header field delimiter; default is the value of the |
| delim | parameter |
| delim | Content field delimiters; defaults are space, tab and comma |

Example:

```
student stus[] <csv_mapper;file="stu_list.txt">;
```

The above example would read a list of student info from file "stu_list.txt" and map them into a student array. By default, the file should contain a header line specifying the names of the columns. If stu_list.txt contains the following:

```
name,age,gpa
```

```
101-name.txt, 101-age.txt, 101-gpa.txt
name55.txt, age55.txt, age55.txt
q, r, s
```

then some of the mappings produced by this example would be:

| **stus[0].name** | **101-name.txt** |
|---|---|
| stus[0].age | 101-age.txt |
| stus[0].gpa | 101-gpa.txt |
| stus[1].name | name55.txt |
| stus[1].age | age55.txt |
| stus[1].gpa | gpa55.txt |
| stus[2].name | q |
| stus[2].age | r |
| stus[2].gpa | s |

### 3.17.10 External Mapper

The external mapper, ext maps based on the output of a supplied Unix executable.

| parameter | meaning |
|---|---|
| exec | The name of the executable (relative to the current directory, if an absolute path is not specified) |
| * | Other parameters are passed to the executable prefixed with a - symbol |

The output (stdout) of the executable should consist of two columns of data, separated by a space. The first column should be the path of the mapped variable, in Swift script syntax (for example [2] means the 2nd element of an array) or the symbol $ to represent the root of the mapped variable. The following table shows the symbols that should appear in the first column corresponding to the mapping of different types of swift constructs such as scalars, arrays and structs.

| Swift construct | first column | second column |
|---|---|---|
| scalar | $ | file_name |
| anarray[] | [] | file_name |
| 2dimarray[][] | [][] | file_name |
| astruct.fld | fld | file_name |
| astructarray[].fldname | [].fldname | file_name |

Example: With the following in mapper.sh,

```
#!/bin/bash
echo "[2] qux"
echo "[0] foo"
echo "[1] bar"
```

then a mapping statement:

```
student stus[] <ext;exec="mapper.sh">;
```

would map

| **Swift variable** | **Filename** |
|---|---|
| stus[0] | foo |
| stus[1] | bar |
| stus[2] | qux |

Advanced Example: The following mapper.sh is an advanced example of an external mapper that maps a two-dimensional array to a directory of files. The files in the said directory are identified by their names appended by a number between 000 and 099. The first index of the array maps to the first part of the filename while the second index of the array maps to the second part of the filename.

```
#!/bin/sh

#take care of the mapper args
while [ $# -gt 0 ]; do
  case $1 in
    -location)          location=$2;;
    -padding)           padding=$2;;
    -prefix)            prefix=$2;;
    -suffix)            suffix=$2;;
    -mod_index)         mod_index=$2;;
    -outer_index)       outer_index=$2;;
    *) echo "$0: bad mapper args" 1>&2
       exit 1;;
  esac
  shift 2
done

for i in `seq 0 ${outer_index}`
do
 for j in `seq -w 000 ${mod_index}`
 do
  fj=`echo ${j} | awk '{print $1 +0}'` #format j by removing leading zeros
  echo "["${i}"]["${fj}"]" ${location}"/"${prefix}${j}${suffix}
 done
done
```

The mapper definition is as follows:

```
file_dat dat_files[][] < ext;
                              exec="mapper.sh",
                              padding=3,
                              location="output",
                              prefix=@strcat( str_root, "_" ),
                              suffix=".dat",
                              outer_index=pid,
                              mod_index=n >;
```

Assuming there are 4 files with name aaa, bbb, ccc, ddd and a mod_index of 10, we will have 4x10=40 files mapped to a two-dimensional array in the following pattern:

| Swift variable | Filename |
|---|---|
| stus[0][0] | output/aaa_000.dat |
| stus[0][1] | output/aaa_001.dat |
| stus[0][2] | output/aaa_002.dat |
| stus[0][3] | output/aaa_003.dat |
| ... | ... |
| stus[0][9] | output/aaa_009.dat |
| stus[1][0] | output/bbb_000.dat |
| stus[1][1] | output/bbb_001.dat |
| ... | ... |
| stus[3][9] | output/ddd_009.dat |

## 3.18 Executing app procedures

This section describes how Swift executes app procedures, and requirements on the behaviour of application programs used in app procedures. These requirements are primarily to ensure that the Swift can run your application in different places and with the various fault tolerance mechanisms in place.

### 3.18.1 Mapping of app semantics into unix process execution semantics

This section describes how an app procedure invocation is translated into a (remote) unix process execution. It does not describe the mechanisms by which Swift performs that translation; that is described in the next section.

In this section, this example Swift script is used for reference:

```
type file;

app (file o) count(file i) {
  wc @i stdout=@o;
}

file q <"input.txt">;
file r <"output.txt">;
```

The executable for wc will be looked up in tc.data.

This unix executable will then be executed in some application procedure workspace. This means:

Each application procedure workspace will have an application workspace directory. (TODO: can collapse terms application procedure workspace and application workspace directory ?

This application workspace directory will not be shared with any other application procedure execution attempt; all application procedure execution attempts will run with distinct application procedure workspaces. (for the avoidance of doubt: If a Swift script procedure invocation is subject to multiple application procedure execution attempts (due to Swift-level restarts, retries or replication) then each of those application procedure execution attempts will be made in a different application procedure workspace. )

The application workspace directory will be a directory on a POSIX filesystem accessible throughout the application execution by the application executable.

Before the application executable is executed:

- The application workspace directory will exist.
- The input files will exist inside the application workspace directory (but not necessarily as direct children; there may be subdirectories within the application workspace directory).
- The input files will be those files mapped to input parameters of the application procedure invocation. (In the example, this means that the file input.txt will exist in the application workspace directory)
- For each input file dataset, it will be the case that @filename or @filenames invoked with that dataset as a parameter will return the path relative to the application workspace directory for the file(s) that are associated with that dataset. (In the example, that means that @i will evaluate to the path input.txt)
- For each file-bound parameter of the Swift procedure invocation, the associated files (determined by data type?) will always exist.
- The input files must be treated as read only files. This may or may not be enforced by unix file system permissions. They may or may not be copies of the source file (conversely, they may be links to the actual source file).

During/after the application executable execution, the following must be true:

- If the application executable execution was successful (in the opinion of the application executable), then the application executable should exit with unix return code 0; if the application executable execution was unsuccessful (in the opinion of the application executable), then the application executable should exit with unix return code not equal to 0.

- Each file mapped from an output parameter of the Swift script procedure call must exist. Files will be mapped in the same way as for input files.
- The output subdirectories will be precreated before execution by Swift if defined within a Swift script such as the location attribute of a mapper. App executables expect to make them if they are referred to in the wrapper scripts.
- Output produced by running the application executable on some inputs should be the same no matter how many times, when or where that application executable is run. *The same* can vary depending on application (for example, in an application it might be acceptable for a PNG→JPEG conversion to produce different, similar looking, output jpegs depending on the environment)

Things to not assume:

- Anything about the path of the application workspace directory
- That either the application workspace directory will be deleted or will continue to exist or will remain unmodified after execution has finished
- That files can be passed between application procedure invocations through any mechanism except through files known to Swift through the mapping mechanism (there is some exception here for external datasets - there are a separate set of assertions that hold for external datasets)
- That application executables will run on any particular site of those available, or than any combination of applications will run on the same or different sites.

### 3.19 How Swift implements the site execution model

This section describes the implementation of the semantics described in the previous section.

Swift executes application procedures on one or more sites.

Each site consists of:

- worker nodes. There is some execution mechanism through which the Swift client side executable can execute its wrapper script on those worker nodes. This is commonly GRAM or Falkon or coasters.
- a site-shared file system. This site shared filesystem is accessible through some file transfer mechanism from the Swift client side executable. This is commonly GridFTP or coasters. This site shared filesystem is also accessible through the posix file system on all worker nodes, mounted at the same location as seen through the file transfer mechanism. Swift is configured with the location of some site working directory on that site-shared file system.

There is no assumption that the site shared file system for one site is accessible from another site.

For each workflow run, on each site that is used by that run, a run directory is created in the site working directory, by the Swift client side.

In that run directory are placed several subdirectories:

- shared/ - site shared files cache
- kickstart/ - when kickstart is used, kickstart record files for each job that has generated a kickstart record.
- info/ - wrapper script log files
- status/ - job status files
- jobs/ - application workspace directories (optionally placed here - see below)

Application execution looks like this:

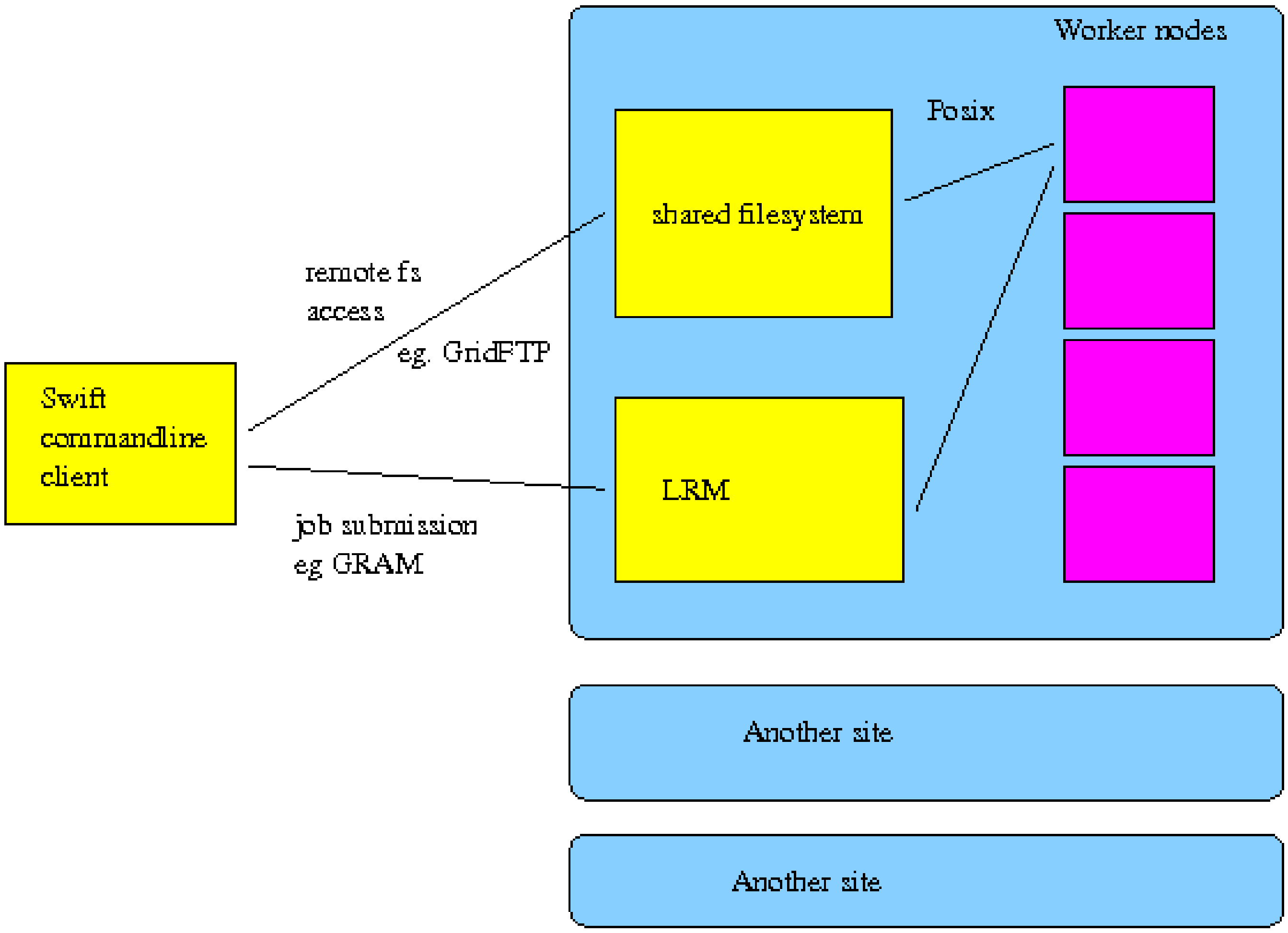
For each application procedure call:

The Swift client side selects a site; copies the input files for that procedure call to the site shared file cache if they are not already in the cache, using the file transfer mechanism; and then invokes the wrapper script on that site using the execution mechanism.

The wrapper script creates the application workspace directory; places the input files for that job into the application workspace directory using either cp or ln -s (depending on a configuration option); executes the application unix executable; copies output files from the application workspace directory to the site shared directory using cp; creates a status file under the status/ directory; and exits, returning control to the Swift client side. Logs created during the execution of the wrapper script are stored under the info/ directory.

The Swift client side then checks for the presence of and deletes a status file indicating success; and copies files from the site shared directory to the appropriate client side location.

The job directory is created (in the default mode) under the jobs/ directory. However, it can be created under an arbitrary other path, which allows it to be created on a different file system (such as a worker node local file system in the case that the worker node has a local file system).



## 3.20 Technical overview of the Swift architecture

This section attempts to provide a technical overview of the Swift architecture.

### 3.20.1 Execution layer

The execution layer causes an application program (in the form of a unix executable) to be executed either locally or remotely.

The two main choices are local unix execution and execution through GRAM. Other options are available, and user provided code can also be plugged in.

The kickstart utility can be used to capture environmental information at execution time to aid in debugging and provenance capture.

#### 3.20.2 Swift script language compilation layer

Step i: text to XML intermediate form parser/processor. parser written in ANTLR - see resources/VDL.g. The XML Schema Definition (XSD) for the intermediate language is in resources/XDTM.xsd.

Step ii: XML intermediate form to Karajan workflow. Karajan.java - reads the XML intermediate form. compiles to karajan workflow language - for example, expressions are converted from Swift script syntax into Karajan syntax, and function invocations become karajan function invocations with various modifications to parameters to accomodate return parameters and dataset handling.

#### 3.20.3 Swift/karajan library layer

Some Swift functionality is provided in the form of Karajan libraries that are used at runtime by the Karajan workflows that the Swift compiler generates.

### 3.21 Function reference

This section details functions that are available for use in the Swift language.

#### 3.21.1 arg

Takes a command line parameter name as a string parameter and an optional default value and returns the value of that string parameter from the command line. If no default value is specified and the command line parameter is missing, an error is generated. If a default value is specified and the command line parameter is missing, @arg will return the default value.

Command line parameters recognized by @arg begin with exactly one hyphen and need to be positioned after the script name.

For example:

```
trace(arg("myparam"));
trace(arg("optionalparam", "defaultvalue"));
```

```
$ swift arg.swift -myparam=hello
Swift v0.3-dev r1674 (modified locally)

RunID: 20080220-1548-ylc4pmda
Swift trace: defaultvalue
Swift trace: hello
```

#### 3.21.2 extractInt

extractInt(file) will read the specified file, parse an integer from the file contents and return that integer.

#### 3.21.3 extractFloat

Similar to extractInt, extractFloat(file) will read the specified file, parse a float from the file contents and return that float.

#### 3.21.4 filename

filename(v) will return a string containing the filename(s) for the file(s) mapped to the variable v. When more than one filename is returned, the filenames will be space separated inside a single string return value.

#### 3.21.5 filenames

filenames(v) will return multiple values containing the filename(s) for the file(s) mapped to the variable v.

#### 3.21.6 length

length(array) will return the length of an array in Swift. This function will wait for all elements in the array to be written before returning the length.

#### 3.21.7 readData

readData will read data from a specified file and assign it to Swift variable. The format of the input file is controlled by the type of the return value. For scalar return types, such as int, the specified file should contain a single value of that type. For arrays of scalars, the specified file should contain one value per line. For complex types of scalars, the file should contain two rows. The first row should be structure member names separated by whitespace. The second row should be the corresponding values for each structure member, separated by whitespace, in the same order as the header row. For arrays of structs, the file should contain a heading row listing structure member names separated by whitespace. There should be one row for each element of the array, with structure member elements listed in the same order as the header row and separated by whitespace. The following example shows how readData() can be used to populate an array of Swift struct-like complex type:

```
type Employee{
    string name;
    int id;
    string loc;
}

Employee emps[] = readData("emps.txt");
```

Where the contents of the "emps.txt" file are:

```
name id loc
Thomas 2222 Chicago
Gina 3333 Boston
Anne 4444 Houston
```

This will result in the array "emps" with 3 members. This can be processed within a Swift script using the foreach construct as follows:

```
foreach emp in emps{
    tracef("Employee %s lives in %s and has id %d", emp.name, emp.loc, emp.id);
}
```

#### 3.21.8 readStructured

readStructured will read data from a specified file, like readdata, but using a different file format more closely related to that used by the ext mapper.

Input files should list, one per line, a path into a Swift structure, and the value for that position in the structure:

```
rows[0].columns[0] = 0
rows[0].columns[1] = 2
rows[0].columns[2] = 4
rows[1].columns[0] = 1
rows[1].columns[1] = 3
rows[1].columns[2] = 5
```

which can be read into a structure defined like this:

```
type vector {
        int columns[];
}

type matrix {
        vector rows[];
}

matrix m;

m = readStructured("readStructured.in");
```

(since Swift 0.7, was readData2(deprecated))

#### 3.21.9 regexp

regexp(input,pattern,replacement) will apply regular expression substitution using the Java java.util.regexp API http://java.sun.com/-j2se/1.4.2/docs/api/java/util/regex/Pattern.html. For example:

```
string v =  regexp("abcdefghi", "c(def)g","monkey");
```

will assign the value "abmonkeyhi" to the variable v.

#### 3.21.10 sprintf

sprintf(spec, variable list) will generate a string based on the specified format.

```
Example: string s = sprintf("\t%s\n", "hello");
```

Format specifiers

| | |
|---|---|
| %% | % sign |
| %M | Filename output (waits for close) |
| %p | Format variable according to an internal format |
| %b | Boolean output |
| %f | Float output |
| %i | int output |
| %s | String output |
| %k | Variable sKipped, no output |
| %q | Array output |

#### 3.21.11 strcat

strcat(a,b,c,d,. . . ) will return a string containing all of the strings passed as parameters joined into a single string. There may be any number of parameters.

The + operator concatenates two strings: strcat(a,b) is the same as a + b

### 3.21.12 strcut

strcut(input,pattern) will match the regular expression in the pattern parameter against the supplied input string and return the section that matches the first matching parenthesised group.

For example:

```
string t = "my name is John and i like puppies.";
string name = strcut(t, "my name is ([^ ]*) ");
string out = strcat("Your name is ",name);
trace(out);
```

This will output the message: Your name is John.

### 3.21.13 strjoin

strjoin(array, delimiter) will combine the elements of an array into a single string separated by a given delimiter. The array passed to strjoin must be of a primitive type (string, int, float, or boolean). It will not join the contents of an array of files.

Example:

```
string test[] = ["this", "is", "a", "test" ];
string mystring = strjoin(test, " ");
tracef("%s\n", mystring);
```

This will print the string "this is a test".

### 3.21.14 strsplit

strsplit(input,pattern) will split the input string based on separators that match the given pattern and return a string array.

Example:

```
string t = "my name is John and i like puppies.";
string words[] = strsplit(t, "\\s");
foreach word in words {
    trace(word);
}
```

This will output one word of the sentence on each line (though not necessarily in order, due to the fact that foreach iterations execute in parallel).

### 3.21.15 toInt

toInt(input) will parse its input string into an integer. This can be used with arg() to pass input parameters to a Swift script as integers.

### 3.21.16 toFloat

toFloat(input) will parse its input string into a floating point number. This can be used with arg() to pass input parameters to a Swift script as floating point numbers.

### 3.21.17 toString

toString(input) will parse its input into a string. Input can be an int, float, string, or boolean.

#### 3.21.18 trace

trace will log its parameters. By default these will appear on both stdout and in the run log file. Some formatting occurs to produce the log message. The particular output format should not be relied upon.

#### 3.21.19 tracef

`tracef(`*`spec`*`,` *`variable list`*`)` will log its parameters as formatted by the formatter *spec*. *spec* must be a string. Checks the type of the specifiers arguments against the variable list and allows for certain escape characters.

Example:

```
int i = 3;
tracef("%s: %i\n", "the value is", i);
```

Specifiers:

**`%s`**
: Format a string.

**`%b`**
: Format a boolean.

**`%i`**
: Format a number as an integer.

**`%f`**
: Format a number as a floating point number.

**`%q`**
: Format an array.

**`%M`**
: Format a mapped variable's filename.

**`%k`**
: Wait for the given variable but do not format it.

**`%p`**
: Format variable according to an internal format.

Escape sequences:

**`\n`**
: Produce a newline.

**`\t`**
: Produce a tab.

**Known issues:**
: Swift does not correctly scan certain backslash sequences such as `\\`.

#### 3.21.20 java

java(class_name, static_method, method_arg) will call a java static method of the class class_name.

#### 3.21.21 writeData

writeData will write out data structures in the format described for readData. The following example demonstrates how one can write a string "foo" into a file "writeDataPrimitive.out":

```

```

# 4 Configuration

Swift uses a single configuration file called swift.properties. The swift.properties file is responsible for:

1. Defining how to interface with schedulers
2. Defining app names and locations
3. Defining various other swift settings and behavior

Here is an example swift.properties file.

```
# Define a site named sandyb
site.sandyb {
   tasksPerWorker=16
   taskWalltime=00:05:00
   jobManager=slurm
   jobQueue=sandyb
   maxJobs=1
   workdir=/scratch/midway/$USER/work
   filesystem=local
}

# Define sandyb apps
app.sandyb.echo=/bin/echo

# Define other swift properties
sitedir.keep=true
wrapperlog.always.transfer=true

# Select which site to run on
site=sandyb
```

The details of this file will be explained more later. Let's first look at an example of running Swift. Using the swift.properties the new Swift command a user would run is:

```
$ swift script.swift
```

That is all that is needed. Everything Swift needs to know is defined in swift.properties.

## 4.1 Location of swift.properties

Swift searches for swift.properties files in multiple locations:

1. The etc/swift.properties file included with the Swift distribution.
2. $SWIFT_SITE_CONF/swift.properties - used for defining site templates.
3. $HOME/.swift/swift.properties
4. swift.properties in your current directory.

5. Any property file you point to with the command line argument "-properties <file>"

Settings get read in this order. Definitions in the later files will override any previous definitions. For example, if you have execution.retries=10 in $HOME/.swift/swift.properties, and execution.retries=0 in the swift.properties in your current directory, execution.retries will be set to 0.

To verify what files are being read, and what values will be set, run:

```
$ swift -listconfig
```

## 4.2 Selecting a site

There are two ways Swift knows where to run. The first is via swift.properties. The site command specified which site entries should be used for a particular run.

```
site=sandyb
```

Sites can also be selected on the command line by using the -site option.

```
$ swift -site westmere script.swift
```

The -site command line argument will override any sites selected in swift.properties.

## 4.3 Selecting multiple sites

To use multiple sites, use a list of site names separated by commas. In swift.properties:

```
site=westmere,sandyb
```

The same format can be used on the command line:

```
$ swift -site westmere,sandyb script.swift
```

**Note**
You can also use "sites=" in swift.properties, and "-sites x,y,z" on the command line.

## 4.4 Run directories

When you run Swift, you will see a run directory get created. The run directory has the name of runNNN, where NNN starts at 000 and increments for every run.

The run directories can be useful for debugging. They contain: .Run directory contents

| | |
|---|---|
| apps | An apps generated from swift.properties |
| cf | A configuration file generated from swift.properties |
| runNNN.log | The log file generated during the Swift run |
| scriptname-runNNN.d | Debug directory containing wrapper logs |
| scripts | Directory that contains scheduler scripts used for that run |
| sites.xml | A sites.xml generated from swift.properties |
| swift.out | The standard out and standard error generated by Swift |

## 4.5 Using site templates

Swift recognizes an environmnet variable called $SWIFT_SITE_CONF, which points to a directory containing a swift.properties file. This swift.properties can contain multiple site definitions for the various queues available on the cluster you are using.

Your local swift.properties then does not need to define the entire site. It may contain only differences you need to make that are specific to your application, like walltime.

## 4.6 Backward compatability

New users are encouraged to use the configuration mechanisms described in this documentation. However, if you are migrating from an older Swift release to 0.95, the older-style configurations using sites.xml and tc.data should still work. If you notice an instance where this is not true, please send an email to swift-support@ci.uchicago.edu.

## 4.7 The swift.properties file format

### 4.7.1 Site definitions

Site definitions in the swift.properties files begin with "site".

The second word is the name of the site you are defining. In these examples we will define a site called westmere.

The third word is the property.

For example:

```
site.westmere.jobQueue=fast
```

Before the site properties are listed, it's important to understand the terminology used.

A **task**, or **app task** is an instance of a program as defined in a Swift app() function.

A **worker** is the program that launches app tasks.

A **job** is related to schedulers. It is the mechanism by which workers are launched.

Below is the list of valid site properties with brief explanations of what they do, and an example swift.properties entry.

Table 1: swift.properties site properties

| Property | Description | Example |
|---|---|---|
| condor | Pass parameters directly through to the submit script generated for the condor scheduler. For example, the setting "site.osgconnect.condor.+projectname=Swift" will generate the line "+projectname = Swift". | site.osgconnect.condor.+projectname=Swift |
| filesystem | Defines how files should be accessed | site.westmere.filesystem=local |
| jobGranularity | Specifies the granularity of a job, in nodes | site.westmere.jobGranularity=2 |
| jobManager | Specifies how jobs will be launched. The supported job managers are "cobalt", "slurm", "condor", "pbs", "lsf", "local", and "sge". | site.westmere.jobManager=slurm |
| jobProject | Set the project name for the job scheduler | site.westmere.project=myproject |
| jobQueue | Set the name of the scheduler queue to use. | site.westmere.jobQueue=westmere |

Table 1: (continued)

| Property | Description | Example |
|---|---|---|
| jobWalltime | The maximum number amount of time allocated in a scheduler job, in hh:mm:ss format. | site.westmere.jobWalltime=01:00:00 |
| maxJobs | Maximum number of scheduler jobs to submit | site.westmere.maxJobs=20 |
| maxNodesPerJob | The maximum number of nodes to request per scheduler job. | site.westmere.maxNodesPerJob=2 |
| pe | The parallel environment to use for SGE schedulers | site.sunhpc.pe=mpi |
| providerAttributes | Allows user to pass attributes through directly to scheduler submit script. Currently only implemented for sites that use PBS. | site.beagle.providerAttributes=pbs.aprun;pbs.mp |
| slurm | Pass parameters directly through to the submit script generated for the slurm scheduler. For example, the setting "site.midway.slurm.mail-user=username" generates the line "#SBATCH --mail-user=username". | site.midway.slurm.mail-user=username |
| stagingMethod | When provider staging is enabled, this option will specify the staging mechanism for use for each site. If set to *file*, staging is done from a filesystem accessible to the coaster service (typically running on the head node). If set to *proxy*, staging is done from a filesystem accessible to the client machine that swift is running on, and is proxied through the coaster service. If set to *sfs* (short for "shared filesystem"), staging is done by copying files to and from a filesystem accessible by the compute node (such as an NFS or GPFS mount) | site.osg.stagingMethod=file |
| taskDir | Tasks will be run from this directory. In the absence of a taskDir definition, Swift will run the task from workdir. | site.westmere.taskDir=/scratch/local/$USER/wo |
| tasksPerWorker | The number of tasks that each worker can run simultaneously. | site.westmere.tasksPernode=12 |
| taskThrottle | The maximum number of active tasks across all workers. | site.westmere.taskThrottle=100 |
| taskWalltime | The maximum amount of time a task may run, in hh:mm:ss. | site.westmere.taskWalltime=01:00:00 |
| site | Name of site or sites to run on. This is the same as running with swift -site <sitename> | site=westmere |
| userHomeOverride | Sets the Swift user home. This must be a shared filesystem. This defaults to $HOME. For clusters where $HOME is not accessible to the worker nodes, you may override the value to point to a shared directory that you own. | site.beagle.userHomeOverride=/lustre/beagle/use |

Table 1: (continued)

| Property | Description | Example |
|---|---|---|
| workdir | The workdirectory element specifies where on the site files can be stored. This directory must be available on all worker nodes that will be used for execution. A shared cluster filesystem is appropriate for this. Note that you need to specify absolute pathname for this field. | site.westmere.workdir=/scratch/midway/$USER/ |

## 4.8 Grouping site properties

The example swift.properties in this document listed the following site related properties:

```
site.westmere.provider=local:slurm
site.westmere.jobsPerNode=12
site.westmere.taskWalltime=00:05:00
site.westmere.queue=westmere
site.westmere.initialScore=10000
site.westmere.filesystem=local
site.westmere.workdir=/scratch/midway/$USER
```

However, you can also simplify this by grouping site properties together with curly brackets.

```
site.westmere {
   provider=local:slurm
   jobsPerNode=12
   taskWalltime=00:05:00
   queue=westmere
   initialScore=10000
   filesystem=local
   workdir=/scratch/midway/$USER/work
}
```

## 4.9 App definitions

In 0.95, applications wildcards will be used by default. This means that $PATH will be searched and pathnames to application do not have to be defined.

In the case where you have multiple sites defined, and you want control over where things run, you will need to define the location of apps. In this scenario, you will can define apps in swift.properties with something like this:

```
app.westmere.cat=/bin/cat
```

When an app is defined in swift.properties for any site you are running on, wildcards will be disabled, and all apps you want to use must be defined.

## 4.10 General Swift properties

Swift behavior can be configured through general Swift properties. Below is a list of properties:

| Name | Valid Values | Default Value | Description |
|---|---|---|---|
| config.rundirs | true, false | true | By default, Swift will generate a run directory that contains logs, scheduler submit scripts, debug directories, and other files associated with a particular Swift run. Setting this value to false disables the creation of run directories and causes all logs and directories to be created in the current working directory. |
| execution.retries | Positive integer | 2 | The number of time a job will be retried if it fails (giving a maximum of 1 + execution.retries attempts at execution) |
| file.gc.enabled | true, false | true | Files mapped by the concurrent mapper (i.e. when you don't explicitly specify a mapper) are deleted when they are not in use any more. This property can be used to prevent files mapped by the concurrent mapper from being deleted. |
| foreach.max.threads | Positive integer | 1024 | Limits the number of concurrent iterations that each foreach statement can have at one time. This conserves memory for swift programs that have large numbers of iterations (which would otherwise all be executed in parallel) |

| Name | Valid Values | Default Value | Description |
|---|---|---|---|
| lazy.errors | true, false | false | Swift can report application errors in two modes, depending on the value of this property. If set to false, Swift will report the first error encountered and immediately stop execution. If set to true, Swift will attempt to run as much as possible from a Swift script before stopping execution and reporting all errors encountered. When developing Swift scripts, using the default value of false can make the program easier to debug. However in production runs, using true will allow more of a Swift script to be run before Swift aborts execution. |
| swift.home | String | | Points to the Swift installation directory ($SWIFT_HOME). In general, this should not be set as Swift can find its own installation directory, and incorrectly setting it may impair the correct functionality of Swift. |

| **Name** | **Valid Values** | **Default Value** | **Description** |
|---|---|---|---|
| pgraph | true, false | false | Swift can generate a Graphviz http://www.graphviz.org/ file representing the structure of the Swift script it has run. If this property is set to true, Swift will save the provenance graph in a file named by concatenating the program name and the instance ID (e.g. helloworld-ht0adgi315l61.dot). If set to false, no provenance graph will be generated. If a file name is used, then the provenance graph will be saved in the specified file. The generated dot file can be rendered into a graphical form using Graphviz http://www.graphviz.org/, for example with a command-line such as: $ swift -pgraph graph1.dot q1.swift $ dot -ograph.png -Tpng graph1.dot |
| pgraph.graph.options | String | splines="compound", rankdir="TB" | This property specifies a Graphviz http://www.graphviz.org specific set of parameters for the graph. |
| pgraph.node.options | String | color="seagreen", style="filled" | Used to specify a set of Graphviz http://www.graphviz.org specific properties for the nodes in the graph. |
| provenance.log | true, false | false | This property controls whether the log file will contain provenance information enabling this will increase the size of log files, sometimes significantly. |
| provider.staging.pin.swiftfiles | true, false | false | When provider staging is enabled and provider.staging.pin.swiftfiles is set, cache some small files needed by Swift to avoid the cost of staging more than once. |

| Name | Valid Values | Default Value | Description |
|---|---|---|---|
| sitedir.keep | true, false | false | Indicates whether the working directory on the remote site should be left intact even when a run completes successfully. This can be used to inspect the site working directory for debugging purposes. |
| status.mode | files, provider | files | Controls how Swift will communicate the result code of running user programs from workers to the submit side. In files mode, a file indicating success or failure will be created on the site shared filesystem. In provider mode, the execution provider job status will be used. provider mode requires the underlying job execution system to correctly return exit codes. |
| tcp.port.range | none | <start>,<end> where start and end are integers | A TCP port range can be specified to restrict the ports on which GRAM callback services are started. This is likely needed if your submit host is behind a firewall, in which case the firewall should be configured to allow incoming connections on ports in the range. |
| throttle.file.operations | <int>, off | 8 | Limits the total number of concurrent file operations that can happen at any given time. File operations (like transfers) require an exclusive connection to a site. These connections can be expensive to establish. A large number of concurrent file operations may cause Swift to attempt to establish many such expensive connections to various sites. Limiting the number of concurrent file operations causes Swift to use a small number of cached connections and achieve better overall performance. |

| Name | Valid Values | Default Value | Description |
|---|---|---|---|
| throttle.host.submit | <int>, off | 2 | Limits the number of concurrent submissions for any of the sites Swift will try to send jobs to. In other words it guarantees that no more than the value of this throttle jobs sent to any site will be concurrently in a state of being submitted. |
| throttle.score.job.factor | <int>, off | 4 | The Swift scheduler has the ability to limit the number of concurrent jobs allowed on a site based on the performance history of that site. Each site is assigned a score (initially 1), which can increase or decrease based on whether the site yields successful or faulty job runs. The score for a site can take values in the (0.1, 100) interval. The number of allowed jobs is calculated using the following formula: 2 + score*throttle.score.job.factor This means a site will always be allowed at least two concurrent jobs and at most 2 + 100*throttle.score.job.factor. With a default of 4 this means at least 2 jobs and at most 402. This parameter can also be set per site using the jobThrottle profile key in a site catalog entry. |

| Name | Valid Values | Default Value | Description |
|---|---|---|---|
| throttle.submit | <int>, off | 4 | Limits the number of concurrent submissions for a run. This throttle only limits the number of concurrent tasks (jobs) that are being sent to sites, not the total number of concurrent jobs that can be run. The submission stage in GRAM is one of the most CPU expensive stages (due mostly to the mutual authentication and delegation). Having too many concurrent submissions can overload either or both the submit host CPU and the remote host/head node causing degraded performance. |
| throttle.transfers | <int>, off | 4 | Limits the total number of concurrent file transfers that can happen at any given time. File transfers consume bandwidth. Too many concurrent transfers can cause the network to be overloaded preventing various other signaling traffic from flowing properly. |
| ticker.date.format | String | | Describes how to format the ticker date output. The format of this string is documented in the Java SimpleDateFormat class, at http://docs.oracle.com/-javase/6/docs/api/java/text/-SimpleDateFormat.html |
| ticker.disable | true, false | false | When set to true, suppresses the output progress ticker that Swift sends to the console every few seconds during a run |
| ticker.prefix | String | Progress: | String to prepend to ticker output |
| tracing.enabled | true, false | true | Enables tracing of procedure invocations, assignments, iteration constructs, as well as certain dataflow events such as data intialization and waiting. This is done at a slight decrease in performance. Traces will be available in the log file. |

| Name | Valid Values | Default Value | Description |
|---|---|---|---|
| use.wrapper.staging | true, false | false | Determines if the Swift wrapper should do file staging. |
| use.provider.staging | true, false | false | If true, files will be staged by Swift over the network. |
| wrapper.invocation.mode | absolute, relative | absolute | Determines if Swift remote wrappers will be executed by specifying an absolute path, or a path relative to the job initial working directory. In most cases, execution will be successful with either option. However, some execution sites ignore the specified initial working directory, and so absolute must be used. Conversely on some sites, job directories appear in a different place on the worker node file system than on the filesystem access node, with the execution system handling translation of the job initial working directory. In such cases, relative mode must be used. |
| wrapper.parameter.mode | args,files | args | Controls how Swift will supply parameters to the remote wrapper script. args mode will pass parameters on the command line. Some execution systems do not pass commandline parameters sufficiently cleanly for Swift to operate correctly. files mode will pass parameters through an additional input file. This provides a cleaner communication channel for parameters, at the expense of transferring an additional file for each job invocation. |
| wrapperlog.always.transfer | true, false | false | This property controls when output from the Swift remote wrapper is transfered back to the submit site. When set to false, wrapper logs are only transfered for jobs that fail. If set to true, wrapper logs are transfered after every job is completed or failed. |

### 4.11 Using shell variables

Any value in swift.properties may contain environment variables. For example:

```
workdir=/scratch/midway/$USER/work
```

Environment variables are expanded locally on the machine where you are running Swift.

Swift will also define a variable called $RUNDIRECTORY that is the path to the run directory Swift creates. In a case where you'd like your work directory to be in the runNNN directory, you may do something like this:

```
workdir=$RUNDIRECTORY
```

# 5 Debugging

## 5.1 Retries

If an application procedure execution fails, Swift will attempt that execution again repeatedly until it succeeds, up until the limit defined in the execution.retries configuration property.

Site selection will occur for retried jobs in the same way that it happens for new jobs. Retried jobs may run on the same site or may run on a different site.

If the retry limit execution.retries is reached for an application procedure, then that application procedure will fail. This will cause the entire run to fail - either immediately (if the lazy.errors property is false) or after all other possible work has been attempted (if the lazy.errors property is true).

With or without lazy errors, each app is re-tried <execution.retries> times before it is considered failed for good. An app that has failed but still has retries left will appear as "Failed but can retry".

Without lazy errors, once the first (time-wise) app has run out of retries, the whole run is stopped and the error reported.

With lazy errors, if an app fails after all retries, its outputs are marked as failed. All apps that depend on failed outputs will also fail and their outputs marked as failed. All apps that have non-failed outputs will continue to run normally until everything that can proceed completes.

For example, if you have:

```
foreach x in [1:1024] {
   app(x);
}
```

If the first started app fails, all the other ones can still continue, and if they don't otherwise fail, the run will only terminate when all 1023 of them will complete.

So basically the idea behind lazy errors is to run EVERYTHING that can safely be run before stopping.

Some types of errors (such as internal swift errors happening in an app thread) will still stop the run immediately even in lazy errors mode. But we all know there are no such things as internal swift errors :)

## 5.2 Restarts

If a run fails, Swift can resume the program from the point of failure. When a run fails, a restart log file will be left behind in the run directory called restart.log. This restart log can then be passed to a subsequent Swift invocation using the -resume parameter. Swift will resume execution, avoiding execution of invocations that have previously completed successfully. The Swift source file and input data files should not be modified between runs.

Normally, if the run completes successfully, the restart log file is deleted. If however the workflow fails, swift can use the restart log file to continue execution from a point before the failure occurred. In order to restart from a restart log file, the -resume logfile argument can be used after the Swift script file name. Example:

```
$ swift -resume runNNN/restart.log example.swift.
```

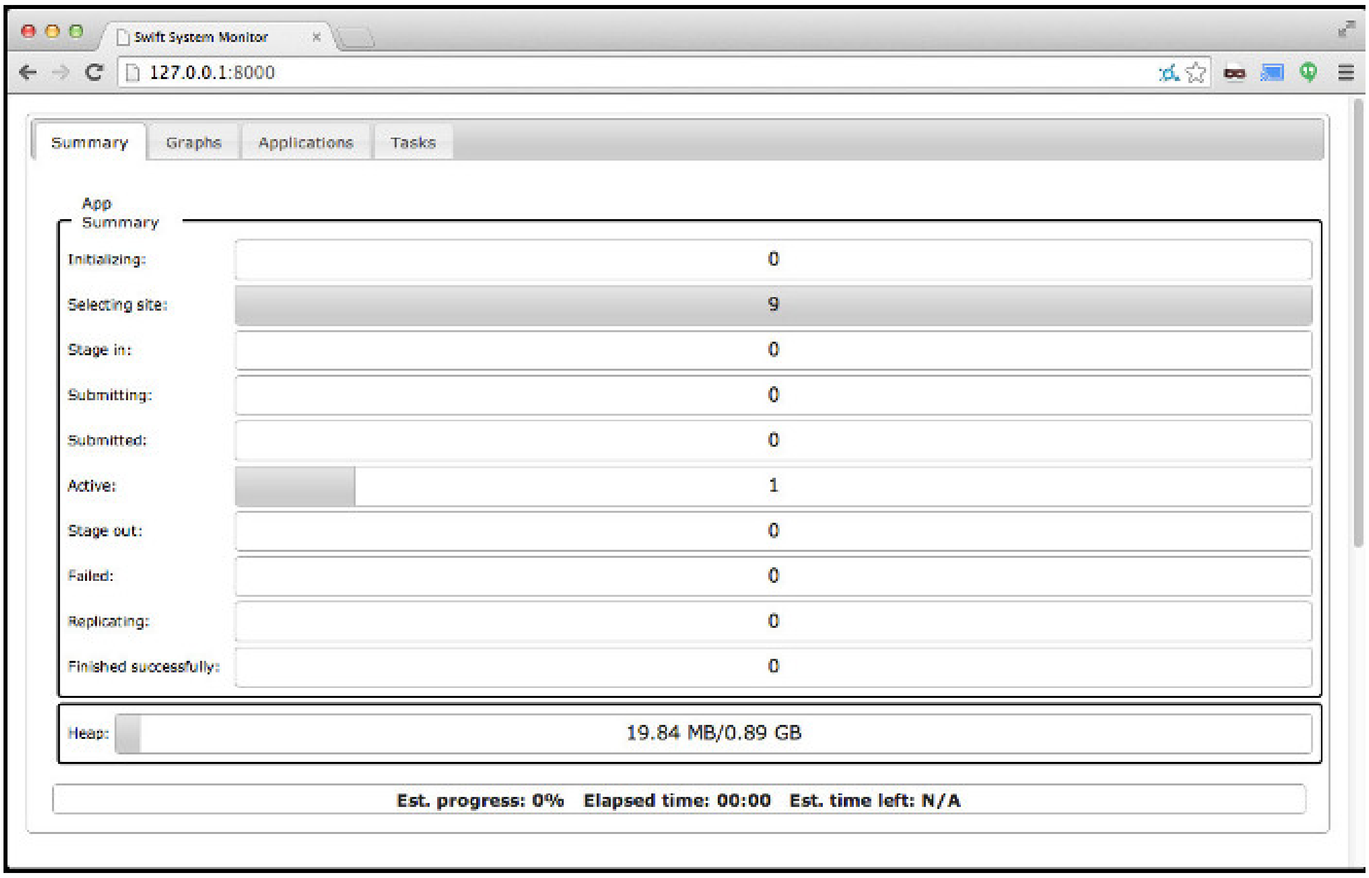
## 5.3 Monitoring Swift

Swift runs can be monitored for progress and resource usage. There are three basic monitors available: Swing, TUI, and http.

### 5.3.1 HTTP Monitor

The HTTP monitor will allow for the monitoring of Swift via a web browser. To start the HTTP monitor, run Swift with the `-ui http:<port>` command line option. For example:

```
swift -ui http:8000 modis.swift
```



App Summary

| | |
|---|---|
| Initializing: | 0 |
| Selecting site: | 9 |
| Stage in: | 0 |
| Submitting: | 0 |
| Submitted: | 0 |
| Active: | 1 |
| Stage out: | 0 |
| Failed: | 0 |
| Replicating: | 0 |
| Finished successfully: | 0 |

Heap: 19.84 MB/0.89 GB

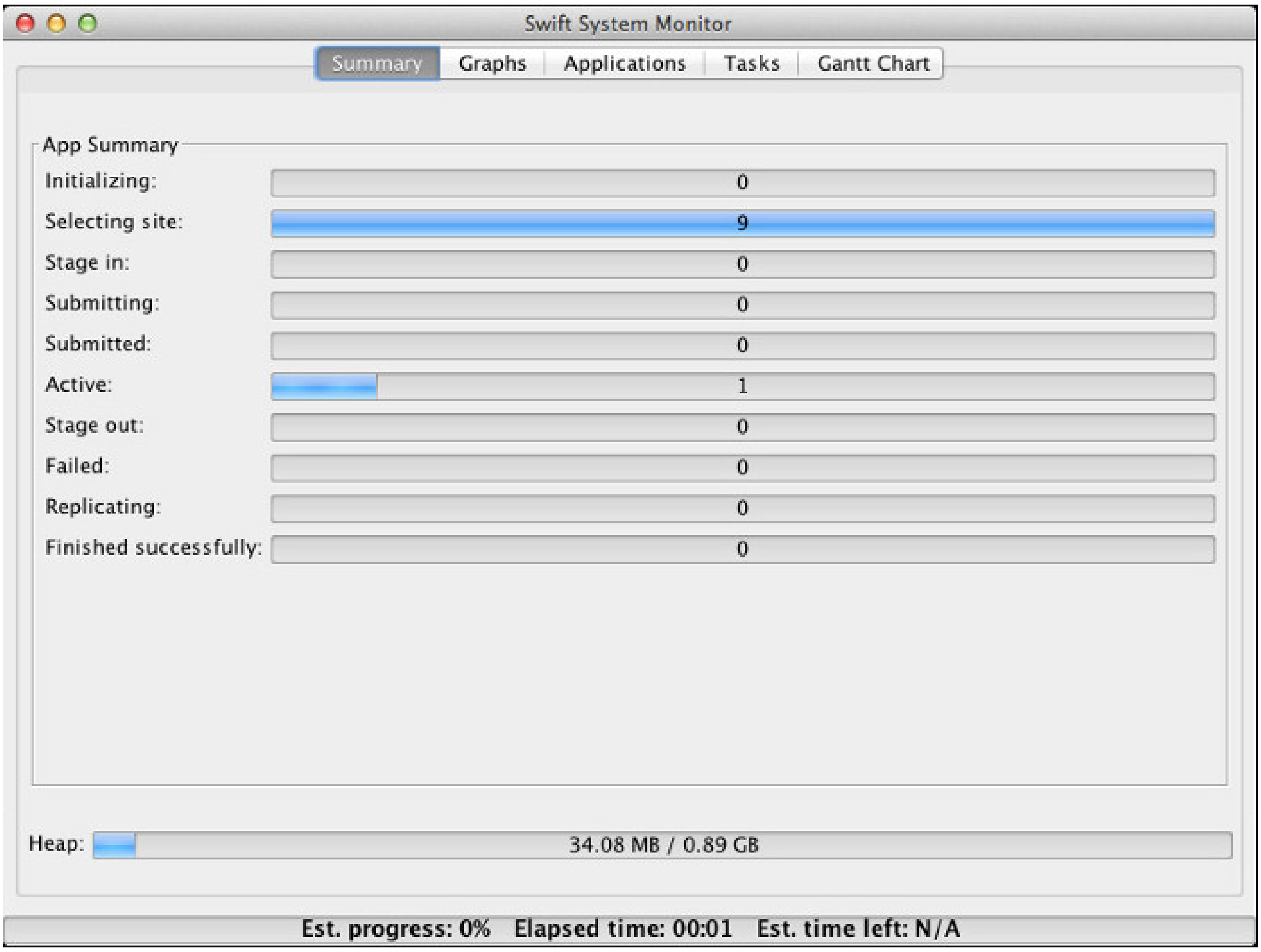
Est. progress: 0%   Elapsed time: 00:00   Est. time left: N/A

This will create a server running on port 8000 on the machine where Swift is running. Point your web browser to http://<ip_address>:8000 to view progress.

### 5.3.2 Swing Monitor

The Swing monitor displays information via a Java gui/X window. To start the Swing monitor, run Swift with the `-ui Swing` command line option. For example:

```
swift -ui Swing modis.swift
```



| | |
|---|---|
| Initializing: | 0 |
| Selecting site: | 9 |
| Stage in: | 0 |
| Submitting: | 0 |
| Submitted: | 0 |
| Active: | 1 |
| Stage out: | 0 |
| Failed: | 0 |
| Replicating: | 0 |
| Finished successfully: | 0 |

Heap: 34.08 MB / 0.89 GB

Est. progress: 0% Elapsed time: 00:01 Est. time left: N/A

This will produce a gui/X window consisting of the following tabs:

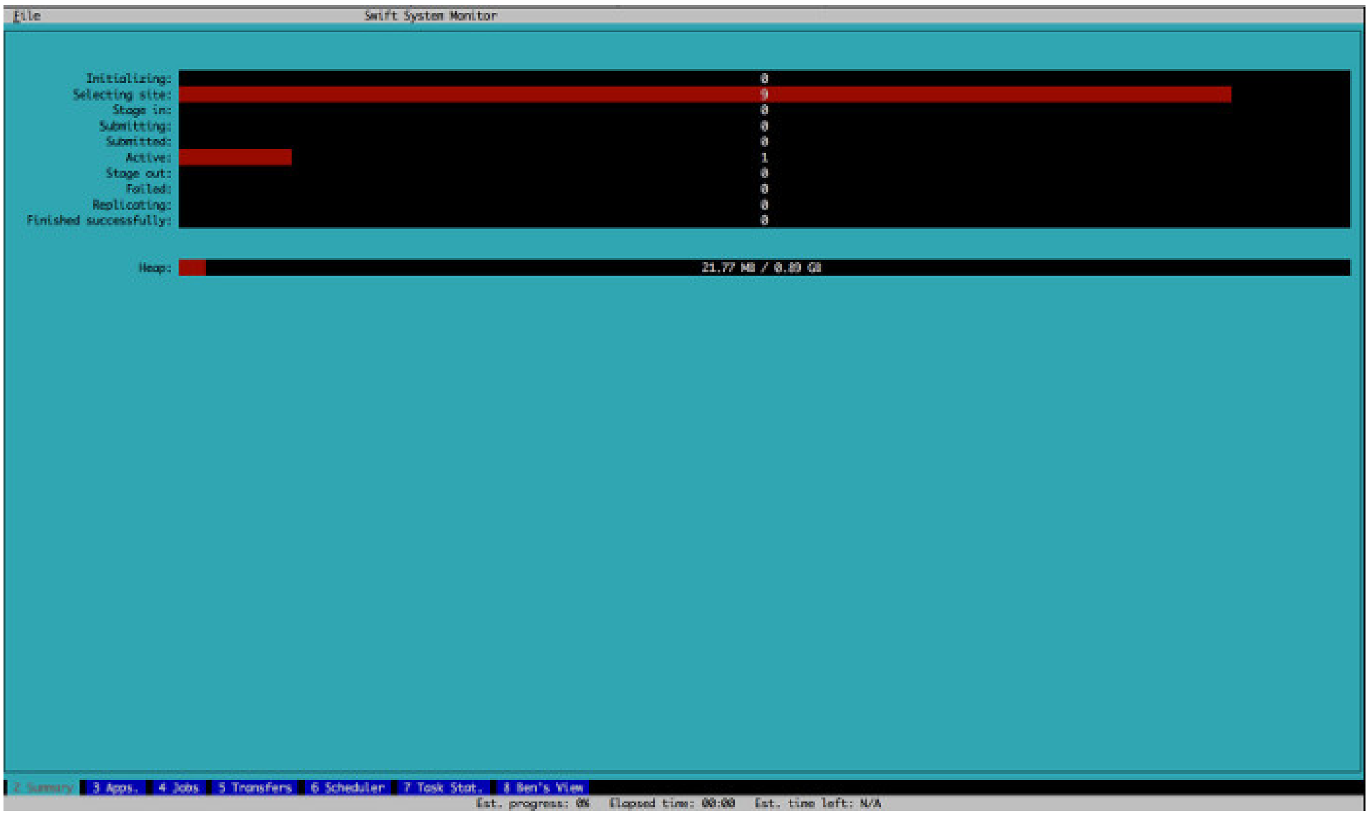
- Summary
- Graphs
- Applications
- Tasks
- Gantt Chart

### 5.3.3 TUI Monitor

The TUI (textual user interface) monitor is one option for monitoring Swift on the console using a curses-like library.

The progress of a Swift run can be monitored using the `-ui TUI` option. For example:

```
swift -ui TUI modis.swift
```


```
File                      Swift System Monitor

         Initializing:                0
       Selecting site:                9
             Stage in:                0
           Submitting:                0
            Submitted:                0
               Active:                1
            Stage out:                0
               Failed:                0
          Replicating:                0
Finished successfully:                0

                 Heap:         21.77 MB / 0.89 GB

2 Summary  3 Apps.  4 Jobs  5 Transfers  6 Scheduler  7 Task Stat.  8 Ben's View
          Est. progress: 0%   Elapsed time: 00:00   Est. time left: N/A
```

This will produce a textual user interface with multiple tabs, each showing the following features of the current Swift run:

- A summary view showing task status
- An apps tab
- A jobs tab
- A transfer tab
- A scheduler tab
- A Task statistics tab
- A customized tab called *Ben's View*

Navigation between these tabs can be done using the function keys f2 through f8.

## 5.4 Log analysis

Swift logs can contain a lot of information. Swift includes a utility called "swiftlog" that analyzes the log and prints a nicely formatted summary of all tasks of a given run.

**swiftlog usage**

```
$ swiftlog run027
Task 1
        App name = cat
        Command line arguments = data.txt data2.txt
        Host = westmere
        Start time = 17:09:59,607+0000
        Stop time = 17:10:22,962+0000
        Work directory = catsn-run027/jobs/r/cat-r6pxt6kl
        Staged in files = file://localhost/data.txt file://localhost/data2.txt
        Staged out files = catsn.0004.outcatsn.0004.err

Task 2
        App name = cat
        Command line arguments = data.txt data2.txt
        Host = westmere
        Start time = 17:09:59,607+0000
        Stop time = 17:10:22,965+0000
        Work directory = catsn-run027/jobs/q/cat-q6pxt6kl
        Staged in files = file://localhost/data.txt file://localhost/data2.txt
        Staged out files = catsn.0010.outcatsn.0010.err
```

home