The Shark Handbook

Index:

Welcome to the shark handbook! This book will aid you in your mission of becoming a master programmer using shark, an easy to learn and highly portable programming language.

This book is an informal introduction to the shark language. The first two chapters are the informal shark language specification. They will teach you the basics of shark programming without diving into implementation details.

Using shark in the real world requires you to learn how to use tools like a compiler or interpreter, and that is not covered here. Shark can be *implemented* and *dialected* by several authors and/or in several software environments and platforms and these implementations are distributed independently. The first step to apply your newly gained knowledge is to install and learn to use a particular shark distribution.

Chapter 1
The Shark Programming Language

Shark is a multi-paradigm procedure based and object-oriented programming language, this means that programs in shark are structured as sequences of state-changing statements that interacts with data in the form of objects.

Not all values in shark are objects though, things like numbers and strings don't have mutable state or methods, but are more like atoms of data that can be used to compose more complex data types.

Chapter 1.1
Basic Types

Shark is a dynamic, strongly typed language. It has at least 6 basic data types: integral and floating-point numeric values, text characters and strings, the boolean values *true* and *false*, and the special *null* value.

Integers are finite signed numbers without a decimal point, and can be represented with decimal literals (like 0, 3, 1234567890). They have a lower bound of 32-bits of storage (but may be larger in your

implementation) and can represent numbers between -2.147.483.648 (-2^{31}) and 2.147.483.647 ($2^{31} - 1$).

Floating-point numbers can have a decimal point and are *usually* represented internally with 64-bits double precision values. You can represent a floating-point number with or without an exponent (like 3.1415 or 0.123e-12).

Both integers and floating-point numbers support addition (x + y), substraction (x - y) and multiplication (x * y) by both integers and floating-point values. The results of such operations must be considered a floating-point number if either of the operands was a floating-point number, or an integer otherwise. They also support negation (-x).

It's a common idiom to convert an integer into floating-point by adding 0.0 to it, like:

```
x = 3
y = x + 0.0
```

For the inverse (converting a floating-point value into integer) use the *floor* or *ceil* mathematical functions of the standard library, like:

```
x = 3.1415
y = floor(x)
```

Division of two numbers (floating-point or not) should be considered to always yield a floating-point value. It's performed by the slash operator (x / y). Division by zero is undefined behavior. You can get an integer by flooring the result, like:

```
y = floor(x / 2)
```

Shark also supports the modulo or remainder operator (x % y). This operator is only defined for integers and yields the remainder of the division x / y. However, it's not defined what's the remainder of the division by a negative number or zero.

All numbers also support comparisons and equality/inequality checks with the following operators:

```
x < y (x less than y)
```

```
x <= y (x less or equal than y)
   x > y (x greater than y)
x >= y (x greater or equal than y)
   x == y (x equals y)
   x != y (x differs y)
```

The result of those operators is a boolean value (either *true* or *false*) indicating the success or failure of the test.

Integers and floating-point values are considered different types of data, although they may be represented uniformly by the underlaying value representation. Using integers in the place of floats or floats in the place of integers is a source of undefined behavior and errors in most cases and should be avoided. Always be sure of having an integer or float in hand before operating on them.

Characters and strings are the way shark has to represent values and sequences of Unicode text data.

Characters appear with single quotes and span a single Unicode code point (like ‘a’, ‘;’, ‘\n’), while strings are double quoted and may span zero to *potentially infinite* characters (like “Hello.”, “” or “\t\n\t> “).

For the sake of portability, only use ASCII characters in character literals. This must be done in order to run your code in environments with 8-bit character values.

Both characters and strings accept escape sequences to represent special characters. An escape sequence starts with a backslash (\) and is followed by an escape character. Only the following escape sequences are defined by standard shark: \0 (the null character) \r (a carriage return) \n (a newline) \t (a tab character) \\ (a single backslash) \“ (a double quote) and \‘ (a single quote).

Single and double quotes may be escaped in order to be inserted in a character or string literal without causing a compiler error (like ‘\’’, “\”This is a sring.\””).

Like with numbers, characters and strings are considered different types of data, although they may be represented uniformly by the underlaying value representation. Take care.

There's not much you can do with string and characters in standard shark without help of the standard library, we're going to learn more about how to process text data in the proper chapter of the standard library reference.

The boolean values true and false are the representations of truth. They are used widely in code that runs based on a condition, and to represent flags. It can be tested if a certain value is true, false or neither by comparing for equality and inequality against *true* and *false*.

Boolean values support negation with the *not* keyword (not x) and conditions can be composed with the *and* or *or* keywords, like:

```
open and ready
closed or not ready
```

The *and* and *or* operators are special. They are called short-circuit operators because they may or may not evaluate their right-hand expression depending on the result of their left-hand one. The *and* operator will evaluate its right hand if and only if the left-hand yields true. Likewise, the *or* operator will evaluate its right hand if and only if the left-hand yields false. This allows for testing a certain condition without fearing unwanted side effects.

The special value *null* is used as a placeholder in places that require a certain object. It can be tested if a certain value is null or not by comparing for equality and inequality against *null*, like:

```
optional_value == null
required_value != null
```

Chapter 1.2
Variables

Variables are special locations within your program where data can be stored or retrieved by using a name.

A variable name should start with an ASCII alphabetic character or an underscore followed by an arbitrary number of ASCII alphabetic characters, digits or underscores.

You can't name a variable after a shark reserved keyword. For reference, here is a list of all reserved keywords in standard shark:

```
import class pass var function
if then else while do for in range
break continue return not and or
self super new instanceof sizeof
null true false
```

A variable in plain shark is declared with the *var* keyword as follows:

```
var name = value
```

Once defined, you can access its value by using the variable's name or assign the name to a new value:

```
name = null
name = 2
name = name + 0.0
```

A variable in shark has no type: it can hold any type of value at any point in time. This is possible because shark is a dynamically typed language, types are assigned to values, not to the location where they're stored. Take for instance the following code:

```
var x = null
x = 3
x = "Hello."
```

Notice the difference between the assign operator (x = y) and the equality comparison operator (x == y). Unlike regular operators, the assign operator can't be used in composed expressions, each assignment delimits a unique statement in your code.

Sometimes you will want to update the value of a variable by operating on the previous value it contains, you can use a special way of assignment known as augmented assignment to operate and update the value of a

variable at the same time. For instance, if you have a variable *counter* that contains an integer you may want to increase or decrease its value by one. Then, you can write:

```
counter += 1
counter -= 1
```

Which is the same as:

```
counter = counter + 1
counter = counter - 1
```

With the exception that *counter* is evaluated only once (something to take in account if the left-hand expression of your assignment has side effects).

Chapter 1.3
Data Structures

There are at least three types of complex data structures in shark: lists, tables and objects. We will discuss objects in the proper section, for now let's focus in the first two: lists and tables.

Lists are represented with square braces that contains a comma separated list of values, like the following:

```
[null, true, false]
[]
[1, 2, 3]
['a', 'z']
```

Usually, a list contains values of the same type in a particular order so they can be processed by the software. Elements of a list are considered to be sequential in memory and are accessed through an integer index starting from zero. This way the first element of a list is x[0], followed by x[1], x[2] and so on. This zero-based indexing may be strange to you at first, but it becomes very natural once you start to write code that operates on lists.

You can get the size of a list (the number of elements it contains) by using the *sizeof* keyword in the following way:

```
sizeof(x)
sizeof([1, 2, 3]) == 3
```

You can access the last element of a list by substracting one to its size, like this:

```
var x = [1, 2, 3]
var y = x[sizeof(x) - 1]
```

It is undefined what happens when accessing an element smaller than zero or greater/equal than the size of a list, take care.

You can append or insert a value in any position of a list by using the insert operator (x << y), like this:

```
var x = [1, 2, 3]
x << 4
```

This appends 4 to the end of the list.

```
x[0] << 0
```

This inserts 0 at the beginning of the list (position zero), shifting every element one position forward.

You can directly assign to the elements of a list by using an index, this includes augmented assignment:

```
x[0] = x[1] + x[2]
x[0] /= 2
```

The indexes of a list don't have to be immediate integer values, they may be stored in a variable or be computed on-the-fly:

```
var x = [“Hello.”, “Bye.”]
var y = 0
x[y]
x[y + 1]
```

Tables are associative data structures and serve to map keys of any type to values of any type. Table keys may be strings, characters, integers or other objects, they may not be null and is not recommended to use floating-point values as keys. Values can be of any type, with special attention to be paid to the null value.

Tables are represented using braces and contains a comma separated list of key-value pairs, separated by a colon character, like those:

```
{“greeting”: “Hello.”, “apologize”: “Sorry.”}
```

```
{ }
{0: true, 3: false, 123: false}
```

Tables are useful in many situations; they are not sequential like lists so you can have a table mapping huge integer keys without consuming tons of memory.

Indexing a table with a missing key Is undefined behavior, but assigning to a missing key is defined: the key will be inserted in the table and assigned to the given value.

```
var data = { }
data[“missing”] = 2 * 16
```

You can check if a given key is present in a table or not with the *in* and *not in* operators, like this:

```
var table = {“required”: “some value”}
“required” in table == true
“optional” not in table == true
“missing” in table == false
```

This is usually a very fast operation, so tables are well suited to be used as *sets*.

Care must be taken if you assign a given key to the null value: some platforms remove keys assigned to null from tables making them invisible to the *in* and *not in* operators. If you’re using a table as a set make sure to use a value different from null to register used keys, the *true* value is usually a good option.

The *sizeof* keyword should not be used to get the size of a table. In fact, the size of a table is not a significant datum.

Both lists and tables are *mutable* data structures and are passed by reference. This means that when assigning a variable to an existing list or table you’re not creating a copy of the whole data structure, but merely making reference to it. The following code demonstrates this behavior:

```
var list_x = [0, 1, 2]
sizeof(list_x) == 3
var list_y = list_x
```

```
sizeof(list_y) == 3
list_y << 12
sizeof(list_x) == 4
sizeof(list_y) == 4
list_x[3] == 12
```
In this example the variables list_x and list_y are references to the same list in memory so when you append a value to list_y the change becomes visible also from list_x.

You can test if two lists or tables are *the same* by checking for equality or inequality.

The shark standard library contains a number of functions to operate on lists and tables that goes beyond those basic operations, we're going to review them in the proper chapter.

Chapter 1.4
Control Statements

A block of code is composed of declarations and statements and is delimited by indentation (the amount of space at the beginning of each line of code). Every file of source code starts with indentation zero and each nested block increases indentation by a fixed amount of space characters. It's an error to indent your code at the beginning of the code.

Empty lines (lines consisting only of spaces) don't participate in the code and are not indentation delimiters. There are other special cases where indentation is not taken into account, like when one of the three delimiters (parentheses, braces and square braces) are left open at the end of a line. The following code is valid shark code:
```
var data = {
    "x": 0.0,
    "y": 0.0
}
```

Statements and declarations in a block are executed in the order they appear one after the other.

There are three composed statements in standard shark: the if/else statement, the while statement and the for statement.

The if statement serves to branch the execution of the code based on a condition. The most basic form of if statement is formed by the *if* keyword followed by a condition expression followed by the *then* keyword followed by a block of code. The block of code that follows the if construct will be executed if and only if the condition expression is true. For instance:

```
var x = 0
if x < 5 then
    x += 1
```

An if statement may be followed immediately by an else clause. This clause specifies what to do *if* the condition expression is false. For instance:

```
var x = 0
if x > 5 then
    x += 2
else
    x += 1
```

Notice how the else keyword must be placed in its own line, matching the indentation of the *if* statement.

You can chain an if statement right after an else clause, making your code branch again on another condition. This chaining can continue indefinitely for as long as you need another conditional branch. The chained if statements will be executed if and only if all the previous conditions were false. For instance:

```
var x = 3
if x == 0 then
    x += 1
else if x == 1 then
    x += 2
else if x == 2 then
    x += 4
else if x == 3 then
    x += 8
else
```

```
    x += x * 2
```

The while statement will repeatedly execute its body for as long its conditional expression is true. The conditional expression must be followed by the *do* keyword. for instance:

```
var x = 1
while x < 100 do
    x *= 2
```

Notice that if the condition is false from the start the statement will execute its block zero times.

The for statement has three forms in standard shark: the *range(end)* from, the *range(start, end)* form and the *list* form. All three forms take a name followed by the *in* keyword followed by an expression or range expression followed by the *do* keyword. The statement will repeat its block for each element in the range or list expression, assigning each value to the variable named in the statement.

Range expressions are used only in a for loop to indicate the range of integers the loop iterates over, in increments of one. It can take the range(end) form to iterate in the range zero to end - 1 or the range(start, end) from to iterate in the range start to end - 1. For instance:

```
var square_list = [ ]
for value in range(1, 100) do
    square_list << value * value
```

Will iterate starting at 1 and doing the last iteration at 99, effectively stopping when 100 - 1 = 99 iterations occurred.

If the for statement is not given a range expression the compiler will assume a list is provided, and the statement will iterate each value of the list. For instance:

```
var number_list = [0, 32, 128]
var square_list = [ ]
for value in number_list do
    square_list << value * value
```

Notice that if the start of the range is greater or equal than its end or the provided list is empty the statement will execute its block zero times.

If a range start is not provided the statement will start with the zero value. This is compatible with the zero-based indexing of lists; you can iterate over the elements of a list and their indexes with something like:

```
var data = [3, 2, 1, 0]
for x in range(sizeof(data)) do
    var min_value = data[x]
    var min_index = x
    for y in range(x, sizeof(data)) do
        if data[y] < min_value then
            min_value = data[y]
            min_index = y
    data[min_index] = data[x]
    data[x] = min_value
```

This example sorts a list of numbers using two nested for loops.

You can use the *pass* keyword to provide an empty block for a composed statement; the empty block will just do nothing. It's illegal to follow the pass keyword with more statements or declarations. For instance:

```
while not ready do
    pass
```

You can use the *break* and *continue* statements to alter the flow of execution of a loop (either a while or for statement).

The break statement terminates the execution of the loop and jumps to its exit point. The following example searches trough a list and exits at the first number divisible by four:

```
var data = [31, 2, 8, 9]
var first = 0
for value in data do
    if value % 4 == 0 then
        first = value
        break
```

The continue statements jumps to the start of the loop and continues execution with the next iteration. The following example computes the addition of all elements in a list except those that are odd:

```
var data = [0, 1, 2, 3, 4]
var total = 0
for value in data do
    if value % 2 != 0 then
        continue
    total += value
```

Using the break or continue statements outside a loop is a syntax error and will be detected by the compiler.

Chapter 1.5
Functions

A function is a piece of code that can be executed on demand. They may take any number of arguments to parametrize the execution of code and do return a value on exit.

A function in shark is declared by using the *function* keyword, followed by a name and a list of argument names enclosed in parentheses. The following are valid function declarations in shark:

```
function update()
function sort(list, comparator)
function main(args)
```

The function declaration should be followed by a block of code: this is the code that executes every time the function is called. The list of argument names declared in the function becomes local variables in that block, and they’re set to the values provided in the function call.

The following example defines a function that increases the value of a variable by the provided argument:

```
var counter = 0
function increase_counter(amount)
    counter += amount
```

You can then call this function by using its name and passing the arguments you want enclosed in parentheses. For instance:

```
increase_counter(24)
```

The following defines a new function that takes no arguments and increases the counter by 1 every time:

```
function next_counter()
    increase_counter(1)
```

Functions can return a value with the special *return* statement, this statement terminates the execution of the current function and gives the provided value back to the caller. The following defines a function that adds two numbers and returns the result:

```
function add(x, y)
    return x + y
```

The return statement can be used without a value, in this case the null value is returned. Likewise, if the execution reaches its end without reaching a return statement the null value is returned. An instance of both situations:

```
function increase_counter_by_even(amount)
    if amount % 2 != 0 then
        return
    counter += amount
```

It's undefined what happens when you call a function with a different number of arguments than those it was declared.

You can pass functions around like other values, like numbers or strings. This means you can create code that accepts and executes other code in demand, something known as *polymorphism*. The following function sorts a list using a *comparator* function that is passed in by the caller:

```
function sort(data, comparator)
    for x in range(sizeof(data)) do
        var min_value = data[x]
        var min_index = x
        for y in range(x, sizeof(data)) do
            if   comparator(data[y],   min_value)
    then
                min_value = data[y]
```

```
                min_index = y
        data[min_index] = data[x]
        data[x] = min_value
```

You can then create any compatible *comparator* function and feed it to the sort algorithm. For instance, a comparator that compares two numbers:

```
function compare_number(x, y)
    return x < y
var data = [12, 3.1415, 128, 0]
sort(data, compare_number)
```

Unlike other languages, shark doesn't haves nested functions or closures. You can create something similar to closures using regular shark data structures, the following code can call a function with variable parameters using a list to feed the call and a call table to speed up the lookup:

```
function call_0(callee, args)
    return callee()
function call_1(callee, args)
    return callee(args[0])
function call_2(callee, args)
    return callee(args[0], args[1])
function call_3(callee, args)
    return callee(args[0], args[1], args[2])
function call_4(callee, args)
    return callee(args[0], args[1], args[2],
args[3])
var call_table = [
    call_0,
    call_1,
    call_2,
    call_3,
    call_4
]
function call_function(callee, args)
    return call_table[sizeof(args)](callee, args)
```

You can grow the number of arguments this code can accept by creating more call_n functions and adding them to the call_table list.

Chapter 1.6

Classes and Objects

Classes and objects are the third and richest kind of data structure supported by shark. An object is a type of value that has properties (member variables) and methods (member functions). All objects are instances of a class, which defines the interface of methods the object supports.

A class is defined using the *class* keyword, followed by the class name and optionally, the class's superclass enclosed in parentheses. The following are valid class declarations:

```
class Action
class FunctionAction (Action)
```

After a class declaration goes a block of function declarations (methods) owned by the class or the *pass* keyword to create an empty class. A valid and complete class declaration looks like the following:

```
class Action
    function init()
        pass
    function call()
        pass
```

The special *init* function defined in a class defines the initialization code that is executed every time that class is instantiated. If a class defines an init function an object can be created by using the *new* keyword followed by the class's name followed by a list of arguments to feed the init function enclosed in parentheses. For instance, to create a new Action object you can write:

```
var my_action = new Action ()
```

It's undefined what happens when instantiation a class that does not defines an *init* function. Once an object is created, you can call its member functions by separating the function's name with a dot and following it with the list of arguments to the call. To call the newly created Action's *call* member function use:

```
my_action.call()
```

You can set and get member variables of an object by using a dot to separate the object from the variable name, like this:

```
my_action.name = "My Action"
```

```
my_action.id = 0
```
Notice that getting a member variable that has not been assigned before is undefined behavior, but setting to it is defined: the named member variable will be created in the object and assigned to the provided value.

Member functions of a class are different from regular functions in a few ways. Every time you call a member function it will receive a special value that represents the current object that owns the function. This special value can be accessed with the *self* keyword. You can use the self keyword inside methods to lookup properties or other methods of the current object. In this example we create a class Employee with three data members that are accessed using the self keyword:

```
var employee_id = 0
class Employee
    function init(name, lastname)
        self.name = name
        self.lastname = lastname
        self.id = employee_id
        employee_id += 1
    function get_name()
        return self.name
    function get_lastname()
        return self.lastname
    function get_id()
        return self.id
```

Classes can inherit member functions from a superclass. Functions inherited from the superclass are threated like they were defined in the current class, including the *init* special function.

A class can override or add to it's parents member functions with new or more appropiate functions for its use case. The following class extends the previously defined Action class with more concrete member functions:

```
class FunctionAction (Action)
    function init(callee, args)
        self.callee = callee
        self.args = args
    function call()
        call_function(self.callee, self.args)
```

It's recommended (but not necessary) that function overrides preserve the number of arguments of the function being overridden. This makes possible to use the newly created class in places where the superclass was expected but applying all the new behaviors, something known as *polymorphism*. In the previous example, we could use a FunctionAction object wherever a regular Action object was expected. It works because the child class preserves the method signatures of its parent (its *interface*).

You can use polymorphism even if there's no relationship between two classes, all it matters is that the class implements the required interface. The following class doesn't inherit from Action or FunctionAction but does preserves the signature of its call method, thus making possible to exchange it by any Action or FunctionAction object:

```
var counter = 0
class IncreaseCounter
    function init(amount)
        self.amount = amount
    function call()
        counter += self.amount
```

You can use the *super* keyword to call the overridden member function from the function that overrides it. This SquaredIncreaseCounter class overrides the init method of its parent and calls the overridden method from the children's one:

```
class SquaredIncreaseCounter (IncreaseCounter)
    function init(amount)
        super(amount * amount)
```

You can use the *instanceof* keyword to test if an object is instance of a given class. The following code accepts either an Action object or a function and invokes it:

```
function invoke(callee)
    if instanceof(callee, Action) then
        callee.call()
    else
        callee()
```

Modules

A *module* is a location where variables, functions and classes can coexist. In shark every source file delimits a module of its own. Modules can interact with other modules by importing them or being imported by them.

A package is a group of modules that live under the same space, usually delimited by folders in the local filesystem.

You can import a module by giving a path to it to the import declaration, this will initialize the module by running any code contained in it and assign the resulting *module object* into a variable:

```
import system
```

Modules may live inside a package, in this case you should give the full path to the module by separating path items with a dot:

```
import system.math
```

Usually a module will be imported under the name of its last path element so the module *system* will be imported as *system* and *system.math* will be imported as *math*. You can specify a name alias in which the module will be imported by using the arrow (->) specifier:

```
import system -> sys
```

Once imported you can access the module's contents with the double colon (::) operator. Modules are like objects in this respect, but use a different operator to clearly distinguish between the two.

```
var string = sys::string
```

You can also directly import any variable, function or class defined in a module by giving their names to the import statement after a colon (:), like this:

```
import system.io: puts, printf
```

When doing this the module will *not* be imported under its usual name. You can't use the arrow notation to provide an alias to the module either. If you want the module under a name, but also some of its contents you can import the same module twice:

```
import system.io
import system.io: puts, printf
```

Once a module or some of its contents is imported you're ready to use them. The following code will display the string "Hello, world!" in the console:

```
import system.io: puts
puts("Hello, world!")
```

Notice that the import declaration can only be used at the start of a file, and one after another. Imports are absolute: it's not possible in standard shark to import a module based on a condition.

It's also not *well defined* what happens when two modules import each other: they *will* load each other and assign the result to their namespaces but it's not defined what module will be loaded first. When using mutual imports, you should never request a particular variable, function or class of the other module. They should always import the other indirectly and should not use the other's contents until both modules are fully loaded. The opposite can crash your program because one of the requested values is not ready yet.

You can of course create your own modules and packages. You can create a module by creating a new text file under the same directory that your main file, and using the ".shk" file extension. Folders are packages in this structure so a file named "game/sound.shk" and placed under the same directory as your main file can be imported with:

```
import game.sound
```

Chapter 1.8
The Main File

The main file must live in the root of all the other modules and packages that compose your program. There are two types of main files: libraries or top-level programs.

A library main file usually contains an import statement for each sub-module of the library making them visible to the compiler.

A top-level program main file is different in a few ways. It does not usually import all the sub-modules of the program but only those required for the program to run. It can also contain a *main* function.

The main function must be called *main*, it must also live in the main file of the program and is going to be called once the program is run, passing any command-line arguments as its unique argument: a list of strings.

The first element in the list of arguments passed to *main* usually contains the file name of the executable program that contains the running shark code, you can use it to access the relative environment in which you program is executing.

It's up to you what to do with the command-line arguments passed to your program, but one thing is sure: defining a main function is the only way of accessing those command-line arguments. Not all programs will define a main function though, some programs don't need or don't care of command line arguments and will just execute their code in the body of the main file.

An example of complete shark program:

```
import system.io: printf
function main(args)
    for name in args do
        printf("Hello, %!\n", [name])
```

Chapter 1.9
Comments

A comment in shark starts with a hash or number symbol (#) and occupies the rest of the line. There are not multi-line comments in shark. Comments have no effect in the program and can contain any text you want without causing a compile-time error. You can start a comment anywhere on the line and it will take the rest of the line for itself:

```
class empty
    pass          # do nothing
```

An example of commented source code:

```
# sorts the list 'data' using 'comparator' to
determine the element's order.
# 'comparator' is a function that accepts two
arguments and returns if the first argument is
smaller than the second.
function sort(data, comparator)
    # iterates the whole 'data' by index.
    for x in range(sizeof(data)) do
        # min_value starts with the value at the
    current index
        var min_value = data[x]
        # min_index starts at the current index
        var min_index = x
        # iterates the slice of 'data' from the
    current index to the end
        for y in range(x, sizeof(data)) do
            # the current item may be the
    smallest we have found.
            # if so, replace the old 'min_value'
    and 'min_index'
            if comparator(data[y], min_value)
    then
                min_value = data[y]
                min_index = y
        # finally, assign the current item to the
    next smallest item of the list
        data[min_index] = data[x]
        data[x] = min_value
```

Chapter 2
The Shark Standard Library

The shark standard library is a set of basic functionalities available to any shark program in any platform. It's composed of eight modules (all of them under the *system* package) and a library (*system*) that imports the eight.

Chapter 2.1
Error Handling With system.error And system.exit

Module system.exit:

var FAILURE

A constant that means the program has failed.

var SUCCESS

A constant that means the program had success.

function exit(code)

Aborts the execution of the program and returns *code* to the system. This is the only legal way of exiting a program (apart from finishing it's execution properly) that will not generate an error message, all other exits are caused by failures in the software.

Module system.error:

var ERR_NONE

A constant that means no error has occurred.

var ERR_UNKNOWN

A constant that means an unknown error has occurred.

function get_err()

Gets the current error code.

function has_err()

Returns true if an error has been reported, false otherwise.

function set_err(value)

Sets the error code to *value*.

function clear_err()

Sets the error code to ERR_NONE

Use this before an operation or sequence of operations that may trigger a change in the global error code. There are other ways of knowing if an error happened, most functions have special ways of signaling errors like returning the null value.

function error(message)

Reports that an error has occurred. This will jump to the closest call to *pcall* (if any) or terminate the execution of the program by returning FAILURE to the system and displaying the *message* string argument to the user.

function pcall(callee, args)

Makes a protected call to the function *callee* passing the list of arguments *args* to the call. The call will fail if *error* is called anywhere during the execution of *callee*. On success, the value returned from the call to *callee* is returned, and on failure, the message passed to *error* is returned and the global error code is set to ERR_UNKNOWN.

Chapter 2.2
Mathematical Functions With system.math

var pi

The constant π = 3.141592…

var e

The constant *e* = 2.718281…

function abs(x)

Returns the absolute value of *x*.

function acos(x)

Returns the arc cosine (measured in radians) of *x*.

function asin(x)

Returns the arc sine (measured in radians) of *x*.

function atan(x)

Returns the arc tangent (measured in radians) of *x*.

function atan2(y, x)

Returns the arc tangent (measured in radians) of *y / x*.

function cos(x)

Returns the cosine of *x* (measured in radians).

function cosh(x)

Returns the hyperbolic cosine of *x*.

function sin(x)

Returns the sine of *x* (measured in radians).

function sinh(x)

Returns the hyperbolic sine of *x*.

function tan(x)

Returns the tangent of *x* (measured in radians).

function tanh(x)

Returns the hyperbolic tangent of *x*.

function exp(x)

Return *e* raised to the power of *x*.

function log(x)

Returns the natural logarithm (base *e*) of *x*.

function log10(x)

Returns the base 10 logarithm of *x*.

function pow(x, y)

Returns *x* to the power of *y*.

function sqrt(x)

Returns the square root of *x*.

function ceil(x)

Returns the ceiling of *x* as an integer. This is the smallest integer value that is greater or equal than *x*.

function floor(x)

Returns the floor of *x* as an integer. This is the smallest integer value that is less or equal than *x*.

function min(x, y)

Returns the smallest of the arguments *x, y*.

function max(x, y)

Returns the greatest of the arguments *x, y*.

function random(x)

Returns a pseudo-random integer value in the integer range 0 to *x* - 1.

Chapter 2.3
System Time Request With system.time

function clock()

Returns the number of seconds elapsed since an implementation defined epoch (usually the start of the program or the first call to *clock*).

Chapter 2.4
Filesystem Utilities With system.path

function get_base(path)

Gets the base of *path*, this is, the full path minus the last path element. If the path contains no slashes or backslashes the empty string is returned. The base of “root/parent/file.txt” is “root/parent”.

function get_tail(path)

Gets the tail of *path*, this is, the text after the last slash or backslash. If the string contains no slashes or backslashes the whole string is returned. The tail of “root/parent/file.txt” is “file.txt”.

function get_ext(path)

Gets the extension of *path*. The extension is the shortest text sequence that starts in a dot and contains no slashes or backslashes. The extension of "root/parent/file.txt” is “.txt”.

function remove_ext(path)

Removes the extension from *path* and returns the resulting string. The path "root/parent/file.txt" with the extension removed is "root/parent/file"

function join(base, tail)

Joins the two path elements *base* and *tail*, inserting a slash or backslash between them if *base* doesn't end with a slash or backslash. The result is undefined if *tail* does start with a slash or backslash.

function system(command)

Synchronously runs the provided command in the system command line interface and returns the exit code of the process.

function listdir(path)

Lists the contents of the *path* directory and returns the result as a list of names or null if *path* is either not a directory or does not exist. To get a full path out of a name returned by this function use *join*(*path*, *name*).

function mkdir(path)

Creates the directory pointed to by *path*. This will fail if either the parent directory of *path* does not exist or *path* points to an existing file or directory. Returns true on success or false on failure.

function rmdir(path)

Removes the directory pointed to by *path*. The directory must be empty. This fill fail if either *path* does not exist or is not a directory. Returns true on success or false on failure.

function unlink(filename)

Unlinks (deletes) the file pointed to by *filename*. Returns true on success or false on failure.

Chapter 2.5
String Handling And Processing With system.string

var ERR_INVALID_LITERAL

An error code that indicates an invalid literal was feed to a conversion function.

var CHAR_SIZE

The character size supported by the current platform. It may be one of the following:

CHAR_SIZE = 1 Characters are 8-bit wide and strings are encoded in UTF-8.

CHAR_SIZE = 2 Characters are 16-bit wide and strings are encoded in UTF-16.

CHAR_SIZE = 4 Characters are 32-bit wide and string encoding is meaningless.

function itos(x)

Returns a decimal string representation of the integer *x*. The result is always a valid numeric literal following shark's grammar.

function ftos(x)

Returns a decimal string representation of the floating-point number *x*. The result is always a valid numeric literal following shark's grammar.

function ctos(x)

Returns a string that contains the character *x*.

function stoi(x)

Returns the numeric value of the string *x*. The string is expected to contain a valid integer literal following shark's grammar for numeric literals, in other case null is returned and the global error code is set to ERR_INVALID_LITERAL.

function stof(x)

Returns the numeric value of the string *x*. The string is expected to contain a valid floating-point literal following shark's grammar for numeric literals, in other case null is returned and the global error code is set to ERR_INVALID_LITERAL.

function islower(x)

Returns true if the character *x* is an ASCII lower-case alphabetical character, false otherwise.

function isupper(x)

Returns true if the character *x* is an ASCII upper-case alphabetical character, false otherwise.

function isalpha(x)

Returns true if the character *x* is an ASCII alphabetical character, false otherwise.

function isdigit(x)

Returns true if the character *x* is an ASCII digit character, false otherwise.

function isalnum(x)

Returns true if the character *x* is an ASCII alphabetical or digit character, false otherwise.

function isident(x)

Returns true if the character *x* is an ASCII alphabetical or digit character or if it is an underscore, false otherwise.

function ishex(x)

Returns true if the character *x* is a hexadecimal digit character, this is, it is either a decimal digit or an alphabetical character in the range A to F (both inclusive) not taking in account the character's case. It returns false otherwise.

function isascii(x)

Returns true if the character *x* is a valid ASCII character (codepoints 0 to 127), false otherwise.

function issurrogate(x)

Returns true if the character *x* is in the Unicode surrogate-pair range, false otherwise.

function tolower(x)

Returns the character *x* in lower case. It's undefined behavior if the character *x* is not an upper-case ASCII alphabetical character.

function toupper(x)

Returns the character *x* in upper case. It's undefined behavior if the character *x* is not a lower-case ASCII alphabetical character.

function len(data)

Returns the length of the string *data* in characters. See CHAR_SIZE for details on the encoding and size of strings and characters.

function index(data, index)

Returns the character of the string *data* corresponding to position *index*. See CHAR_SIZE for details on the encoding and size of strings and characters.

function slice(data, start, end)

Returns the slice of the string *data* from position *start* to position *end*. See CHAR_SIZE for details on the encoding and size of strings and characters.

function find(x, y)

Returns the smallest index of the string *x* where the string *y* is found, or -1 if *y* is not a substring of *x*.

function concat(x, y)

Returns the concatenation of *x* and *y*.

function join(sep, list)

Joins every string in *list* inserting the string *sep* between items. The string *sep* may be empty, effectively joining all strings into one.

function split(data, sep)

Splits the string *data* into parts on each occurrence of the character *sep*. The size of the returned list is always greater than zero. For instance, the line:

```
split("a.b.c", '.')
```

Will return the list:

```
["a", "b", "c"]
```

function format(data, args)

Formats the string *data* using the list of numbers or stings *args* and returns the result. Formatting is done by replacing each occurrence of the percent symbol (%) by the text representation of the corresponding argument passed in *args*. It's undefined what happens if the size of the list *args* is different to the count of percent symbols in the *data* string.

For instance, the line:

```
format("The student % eat % %s.", ["Malcom", 5,
                    "apple"])
```

Will return the string:

```
"The student Malcom eat 5 apples."
```

function normal(data)

Normalizes the string *data* by removing any starting or ending quotes and double quotes and replacing escape sequences by their corresponding ASCII characters. It's like if the string was feed to a shark compiler and its result were returned.

function quote(data)

Quotes the string *data* by adding a starting and an ending double quote and replacing the unescaped ASCII character with their scape sequences. It's as if the data were prepared to be feed to a shark compiler.

class strbuf

The class strbuf is a stream of characters useful to build large string from smaller parts. It follows an interface similar to text file objects in output mode so you can exchange them in certain situations.

function init()

To create a new strbuf use:

```
new strbuf ()
```

It's undefined what happens when you call strbuf.init() when a strbuf is already created, this may leave the strbuf in an inconsistent state.

function put(value)

Appends the character *value* to the internal string buffer.

function puts(data)

Appends the string *data* to the internal string buffer.

function printf(data, args)

Formats *data* using *args* using the same algorithm than the function format of the system.string module and appends the result to the internal string buffer.

function read_all()

Reads the whole internal string buffer and returns its data as a string. The internal buffer is then cleared.

class bytes

The class bytes is a stream of binary data suitable to build or process any kind of binary data structure.

function init()

To create a new bytes object use:

```
new bytes ()
```

It's undefined what happens when you call bytes.init() when a bytes object is already created, this may leave the bytes object in an inconsistent state.

function put(value)

Appends the integer *value* to the internal byte buffer as a single byte. Its undefined what happens if *value* is not in the range 0 to 255.

function put_short(value)

Appends the integer *value* to the internal byte buffer as a sequence of two bytes. Its undefined what happens if *value* is not in the range 0 to 2^{16}-1.

function put_int(value)

Appends the integer *value* to the internal byte buffer as a sequence of four bytes. Its undefined what happens if *value* is not in the range 0 to 2^{32}-1.

function puts(data)

Appends the internal data of the bytes object *data* to the internal byte buffer.

function tell()

Returns the size of the internal byte buffer as a positive integer. This can be used to compute offsets or positions in the data.

function patch(pos, value)

Sets the byte at the positive integer *pos* to value. Its undefined what happens if *value* is not in the range 0 to 255.

function patch_short(pos, value)

Sets the sequence of two bytes at the positive integer *pos* to *value*. Its undefined what happens if *value* is not in the range 0 to 2^{16}-1.

function patch_int(pos, value)

Sets the sequence of four bytes at the positive integer *pos* to *value*. Its undefined what happens if *value* is not in the range 0 to 2^{32}-1.

function get(index)

Gets the byte value at position *index*.

function encode(data)

Encodes the string *data* as a sequence of bytes in UTF-8 encoding and returns the resulting bytes object.

function decode(data)

Decodes the bytes object *data* as a sequence of UTF-8 characters and returns the resulting string.

Chapter 2.6

Input/Output Functionality With system.io

var ERR_FILE_NOT_FOUND

An error constant that means the requested file was not found.

function put(value)

Displays the character *value* in the console.

function puts(data)

Displays the string *data* in the console.

function printf(data, args)

Formats the string *data* using *args* using the same algorithm than the *format* function of the system.string module and displays the result in the console.

function read_line()

Reads a line of input text from the console.

class text_file

A text_file object binds a file in the filesystem and an internal text buffer that writes or reads to it. It defines no *init* function so it can't be instantiated by usual means.

function put(value)

Writes the character *value* in this text_file object.

function puts(data)

Writes the string *data* in this text_file object.

function printf(data, args)

Formats the string *data* using *args* using the same algorithm than the *format* function of the system.string module and writes the result in this text_file object.

function fetch()

Reads and returns one character from this text_file object.

function read(count)

Reads at max *count* characters from this text_file object and returns the result as a string.

function at_end()

Returns true if this text_file object has reached its end, false otherwise.

function close()

Closes this text_file object and makes it unusable. You should always call close() on every file object you opened during the execution of your program. Leaving a file unclosed at the end of the execution produces an undefined behavior.

class binary_file

A binary_file object binds a file in the filesystem and an internal byte buffer that writes or reads to it. It defines no *init* function so it can't be instantiated by usual means.

function put(value)

Writes the integer *value* in this binary_file object. Its undefined what happens if *value* is not in the range 0 to 255.

function puts(data)

Writes the bytes object *data* in this binary_file object.

function fetch()

Reads and returns one byte from this binary_file object.

function read(count)

Reads at max *count* bytes from this binary_file object and returns the result as a system.string's bytes object.

function at_end()

Returns true if this binary_file object has reached its end, false otherwise.

function close()

Closes this binary_file object and makes it unusable. You should always call close() on every file object you opened during the execution of your program. Leaving a file unclosed at the end of the execution produces an undefined behavior.

function open(filename, mode)

Attempts to open the file pointed to by the string *filename* using the string *mode* to decide what type of file to return, and returns either the opened file object or null on error (use get_err() from system.error to know more). The *mode* argument may be any of the following strings:

"r" Opens the file for reading in text mode (returns a text_file object). Files opened this way should have an UTF-8 encoding, the opposite will trigger an undefined behavior.

"w" Opens the file for writing in text mode (returns a text_file object). Files opened this way will have an UTF-8 encoding. This will override any existing file pointed to by *filename* (if any) and create a new empty file. The *filename* argument may point to a file that doesn't exist, but it should point to a directory that already exists or the function will fail and return null.

"rb" Opens the file for reading in binary mode (returns a binary_file object).

"wb" Opens the file for writing in binary mode (returns a binary_file object). This will override any existing file pointed to by *filename* (if any) and create a new empty file. The *filename* argument may point to a file that doesn't exist, but it should point to a directory that already exists or the function will fail and return null.

Chapter 2.7
Data Structure Utilities With system.util

function copy(object)

Makes and returns a shallow copy of *object* (either a list or table).

function slice(list, start, end)

Returns a list containing the slice of data in the range *start* to *end* of the *list* argument.

function pop(list)

Removes and returns the last element of *list*.

function popindex(list, index)

Removes and returns the element of *list* at position *index*, shifting every element after *index* one position backwards.

function find(list, value)

Returns the smallest index where *list*[index] == *value*, or -1 if *value* is not contained in *list*.

function concat(list, other)

Returns the concatenation of the lists *list* and *other*.

function extend(list, other)

Extends *list* in place with the contents of *other*.

function remove(table, index)

Removes the key *index* (if present) from *table* and returns its associated value.

function update(table, other)

Adds every key-value pair found in *other* to *table*.

Chapter 3
The SharkGame Standard Framework

The SharkGame Framework is a standardized set of functionalities to develop games using shark. The GS standard is very small and somewhat limited at this point. Some important features that are not supported (but may be implemented in the future) are audio playback, networking and 3D graphics.

You can think of making a game in the GS platform as making a game to a small sized handheld console. This is not a real hardware console though, games made for this platform are usually executed by a virtual machine that emulates such device in the best way possible. The specifications of

such device and how to interact with it through shark code are found in the following chapters.

Chapter 3.1
Project Structure

All GS projects have the same structure. A project named “test” will have the following directory/file layout:

test/asset/

Optional. The folder where game assets are placed.

test/asset/icon.png

Optional. An icon image that represents this game, with a max size of 64x64 pixels and no transparency.

test/bin/

Required. The folder where game binaries are placed. This is the go-to destination to the shark compiler output.

test/src/

Required. The folder where the game’s source code is placed. It should only contain shark source files and packages in its sub-directories.

test/src/main.shk

Required. This is the main source file of the game and must contain the *main_activity* class. Unlike regular shark top-level programs, a main function is not required, moreover, defining a main function in a GS project causes undefined behavior.

Chapter 3.2
The Main Activity

A GS activity represents a running process in the GS virtual machine. All GS activity objects must implement the same activity interface, defined by the *activity* class in the *shark.activity* module. It’s not enough to implement this interface though, a true GS activity must *inherit* from the *activity* class in order to work properly, the opposite will trigger undefined behavior.

The main activity of your game must live in the src/main.shk file of your project and be called *main_activity*. An example:

```
import shark.activity: activity
```

```
class main_activity (activity)
    pass
```
This is an empty GS activity that does nothing and draws nothing to the screen. Executing it will *probably* display an infinite black screen.

class activity

function draw(text, x, y)

Draws the texture *text* into this activity's display at position *x, y*. The drawing is done by taking the x and y argument and drawing the given texture by placing its top-left corner at the *x, y* position. Negative or overflowing positions will draw partially (or entirely, if the offset is greater than the texture's size) outside the targets space, leaving visible only the part of the texture that is drawn inside the targets space. This method is not supposed to be overridden by subclasses and, in fact, overriding it will produce an undefined behavior.

function draw_ex(text, x, y, origin_x, origin_y, rotation, scale_x, scale_y)

Draws the texture *text* into this activity's display applying the given transformations. The texture is drawn centered at (*x, y*), position is computed by adding *origin_x* and *origin_y* coordinates to the center of the texture, rotating it *rotation* degrees clockwise and scaling it by the (*scale_x, scale_y*) vector. This method is not supposed to be overridden by subclasses and, in fact, overriding it will produce an undefined behavior.

function draw_text(text, font, x, y)

Draws the text string *text* using the given font into this activity's display at position *x, y*. This method is not supposed to be overridden by subclasses and, in fact, overriding it will produce an undefined behavior.

function launch()

Launches this activity. This method is automatically called once at the start of the activity's lifetime to initialize it's state. You should run any initialization code required by your activity in this method. It's not necessary nor recommended to define an *init* method for your activity class, doing so will trigger undefined behavior.

function event(type, x, y)

This method is going to be called once for each event that is triggered by the user or by software. Events are dispatched once and in the same order they got triggered. The argument *type* signals the type of event received and *x, y* are the positions in the screen where the event happened. We will see more about events and how to properly implement the event method in future chapters.

function update()

This method is going to be called once per game cycle to update your activity's state. A game cycle in GS occurs every *approximately* 41.6 milliseconds, this way a GS application will run at a rate of *approximately* 24 cycles per second. This method is always called *right after* dispatching all events of the cycle and *right before* calling this activity's render method.

function render()

This method is going to be called once per game cycle to update your activity's display. All GS activities have a single display of 480x272 pixels prepared for drawing through the activity's draw methods. Only the *render* function may call the draw methods though, calling them outside the render function is undefined behavior.

Chapter 3.3
Textures

A texture object is used to contain image data that can be drawn into your activity's display.

To load a texture from an image asset use the *load_texture* function of the *shark.asset* module. This function accepts only images in the PNG format.

function load_texture(name)

Loads the image asset *name* into a texture object. The file must exist under the assets directory of the project.

The *texture* class is defined in the *shark.texture* module as follows:
class texture

function get_size_x()

Gets the width of this texture in pixels.

function get_size_y()

Gets the height of this texture in pixels.

This example loads two textures from image assets and draws them to the screen:

```
import shark.activity: activity
import shark.asset: load_texture
import system.math: floor
var bg = load_texture("bg.png")
var logo = load_texture("logo.png")
var logo_x = floor(480 / 2 - logo.get_size_x() / 2)
var logo_y = floor(272 / 2 - logo.get_size_y() / 2)
class main_activity (activity)
    function render()
        self.draw(bg, 0, 0)
        self.draw(logo, logo_x, logo_y)
```

This example animates the *logo* texture around the screen:

```
import shark.activity: activity
import shark.asset: load_texture
import system.math: floor
var bg = load_texture("bg.png")
var logo = load_texture("logo.png")
var logo_max_x = 480 - logo.get_size_x() + 1
var logo_y = floor(272 / 2 - logo.get_size_y() / 2)
class main_activity (activity)
    function launch()
        self.logo_x = 0
    function update()
        self.logo_x += 1
        self.logo_x %= logo_max_x
    function render()
        self.draw(bg, 0, 0)
        self.draw(logo, self.logo_x, logo_y)
```

Chapter 3.4

Text And Fonts

GS supports the rendering of text through the use of TrueType fonts. A TTF font can be loaded with the *load_font* function of the *shark.asset* module.

function load_font(name, size, color)

Loads the font asset *name* into a font object. Here *size* is the height of the font in pixels (an integer) and *color* is the color in which the font will render text (an integer). The file must exist under the assets directory of the project.

Colors are represented using a four-byte (32-bits) integer that contains one byte per color channel (red, green, blue and transparency). This way an opaque green color is represented by (255 * (256 * 256)) + 255. The multiplication 256 * 256 yields 65536 (0x10000 in hex). This multiplied by 255 (0xFF) yields 0xFF0000, plus 255 yields 0xFF00FF. Black can be represented as just 255 (0xFF).

Font objects are defined by the *font* class of the *shark.text* module as follows:

class font

function get_height()

Gets the height of this font object in pixels. This refers to the actual height in pixels of a text line rendered using this font and may differ from the height passed to *load_font* in the *size* argument.

This example renders some text using a loaded font and draws it to the screen:

```
import shark.activity: activity
import shark.asset: load_texture, load_font
var bg = load_texture("bg.png")
var font = load_font("courier_new.ttf", 12, 0xFF)
class main_activity (activity)
    function render()
        self.draw(bg, 0, 0)
```

```
self.draw_text("Hello, world!", 80, 180)
```

Chapter 3.5
Events

Events are the way of handling user input in a GS application.

The GS platform defines six buttons that can be pressed and released, they are: the four buttons of the directional pad (Up, Down, Left and Right) and the special buttons X and Y.

The button X is standardized as the Accept button, and is used to trigger user interface events, meanwhile the button Y is standardized as the Cancel button and is used to go a step back in the user interface.

Those are only guidelines though; you can use the input from the user as you want or even let the user customize how buttons will behave in your application.

Users also have access to a single touch virtual tactile display, that can detect the pressing, moving and releasing of the user at any position of the screen.

Events of any type triggered by the user will be dispatched in order to a GS application through the activity object's *event* method. Just to remind you of what this method looks like, here is its prototype:

```
function event(type, x, y)
```

In the *event* method the *type* argument will receive the type of event that got triggered by the user. It can take the value of any of the 15 special constants defined in the *shark.event* module:

var E_PRESS

Event type that means the tactile screen has been pressed. When passing this event type to the *event* method the accompanying *x* and *y* arguments will hold the current position of the touch in the screen.

var E_MOVE

Event type that means the user input has moved in the tactile screen. When passing this event type to the *event* method the accompanying *x* and *y* arguments will hold the current position of the touch in the screen.

var E_RELEASE

Event type that means the tactile screen has been released. When passing this event type to the *event* method the accompanying *x* and *y* arguments will hold the current position of the touch in the screen.

var E_PRESS_UP

Event type that means the Up button of the directional pad has been pressed.

var E_PRESS_DOWN

Event type that means the Down button of the directional pad has been pressed.

var E_PRESS_LEFT

Event type that means the Left button of the directional pad has been pressed.

var E_PRESS_RIGHT

Event type that means the Right button of the directional pad has been pressed.

var E_PRESS_X

Event type that means the X button has been pressed.

var E_PRESS_Y

Event type that means the Y button has been pressed.

var E_REL_UP

Event type that means the Up button of the directional pad has been released.

var E_REL_DOWN

Event type that means the Down button of the directional pad has been released.

var E_REL_LEFT

Event type that means the Left button of the directional pad has been released.

var E_REL_RIGHT

Event type that means the Right button of the directional pad has been released.

var E_REL_X

Event type that means the X button has been released.

var E_REL_Y

Event type that means the Y button has been released.

The following example displays touch and button events on the screen:

```
import shark.activity: activity
import shark.asset
import shark.event
import system.math: floor
var touch_table = {
    event::E_PRESS: “pressed”,
    event::E_MOVE: “moved”,
    event::E_RELEASE: “released”,
}
var button_table = {
    event::E_PRESS_UP: “pressed up”
    event::E_PRESS_DOWN: “pressed down”
    event::E_PRESS_LEFT: “pressed left”
    event::E_PRESS_RIGHT: “pressed right”
    event::E_PRESS_X: “pressed X”
    event::E_PRESS_Y: “pressed Y”
    event::E_REL_UP: “released up”
    event::E_REL_DOWN: “released down”
    event::E_REL_LEFT: “released left”
    event::E_REL_RIGHT: “released right”
    event::E_REL_X: “released X”
```

```
    event::E_REL_Y: “released Y”
}
var bg = asset::load_texture(“bg.png”)
var font = asset::load_font(“courier_new.ttf”, 12,
0xFF)
var center_x = floor(480 / 2)
var center_y = floor(272 / 2)
var touch_text = null
var touch_x = 12
var touch_y = 12
var button_text = null
var button_x = 12
var button_y = 36
function set_touch_text(text)
    touch_text = text
function set_button_text(text)
    button_text = text
set_touch_text(“You have not pressed the screen
yet.”)
set_button_text(“You have not pressed any buttons
yet.”)
class main_activity (activity)
    function event(type, x, y)
        if type in touch_table then
            set_touch_text(format(“You have % in
the screen at (%, %).”, [touch_table[type], x,
y]))
        else
            set_button_text(format(“You have %.”,
[button_table[type]]))
    function render()
        self.draw(bg, 0, 0)
        self.draw_text(touch_text,        touch_x,
touch_y)
        self.draw_text(button_text,      button_x,
button_y)
```

Chapter 3.6

Data Persistence

Data persistence is achieved in GS through the *shark.persistent* module. This module contains a single function that's used to request the save file for the current GS application to the system:

function get_save_file(mode)

Opens the save file for the current GS application and returns the resulting file object. The *mode* argument is used in the same way than the *open* function of the *system.io* module. This function returns null if a read or read binary mode is requested and the save file has not been created yet for the current application, this is *not* considered an error but a way to signal that no save data is available (however, the global error code may be set to ERR_FILE_NOT_FOUND).

Once the save file is opened correctly you can write or read data from it like a regular text_file or binary_file object. This is considered the only legal way of storing and reading persistent data for a GS application, opening and using files directly using *system.io* functions in a GS application is undefined behavior.