The Sapphire Programming Language

Official Documentation v1.0.3

Table of Contents

Chapter 1: Introduction

What is Sapphire?

Sapphire is a dynamic, object-oriented scripting language designed for simplicity, readability, and embeddability. It features a familiar C-style syntax, making it easy to learn for developers coming from languages like C++, Java, or JavaScript.

The language is compiled to a custom bytecode format and executed on its own lightweight Virtual Machine (VM). This architecture allows Sapphire to be integrated into larger host applications (currently a C++ application using SFML and ImGui) to provide flexible scripting capabilities.

Design Philosophy

- **Simplicity:** The syntax is clean and avoids unnecessary complexity.
- **Object-Oriented:** Everything is designed around the concept of objects, from simple numbers to complex user-defined classes.
- **Embeddable:** The language is built to be controlled by a host application, allowing it to script graphics, application logic, or any other host-provided functionality.

Hello, World!

The traditional first program in any language is simple in Sapphire. It uses the built-in print function to output text to the console. All statements must end with a semicolon.

```
// My first Sapphire program
print("Hello, World!");
```

Chapter 2: Getting Started

Running Scripts

The Sapphire interpreter is a command-line application named sapphire.exe. To execute a script file (which typically has a .sp extension), you provide the file path as a command-line argument.

```
C:\Users\YourUser\Projects\Sapphire>sapphire my_script.sp
```

If the script is in a different directory, you can provide the full or relative path.

The --info Command

To display version information about the interpreter, use the --info flag. This is useful for verifying your installation.

C:\Users\YourUser\Projects\Sapphire>sapphire —info

This will display the Sapphire ASCII art logo and version details.

Chapter 3: Language Fundamentals

Comments

Comments are used to annotate code and are ignored by the interpreter. Sapphire supports single-line comments, which start with // and continue to the end of the line.

```
// This is a full-line comment.
double pi = 3.14; // This is an inline comment.
```

Variables and Data Types

Variables must be declared with their type before use. The declaration syntax is type variable_name; or type variable_name = initial_value;.

Core Data Types

- **number**: A 64-bit floating-point number. Sapphire uses double for all numeric values, but you can use the keywords int, double, and float for declaration clarity.
  ```
  int integer_value = 100;
  double float_value = 99.5;
  double sum = integer_value + float_value; // sum is 199.5
  ```

- **string**: A sequence of characters. String literals are enclosed in double quotes (").
  ```
  string greeting = "Hello";
  string name = "Sapphire";
  string message = greeting + ", " + name + "!"; // "Hello, Sapphire!"
  ```

- **bool**: A boolean value, which can be either true or false.
  ```
  bool is_complete = false;
  is_complete = true;
  ```

- **nil**: A special value representing "no value" or "nothing". If a variable is declared without an initial value, it defaults to nil. In conditional statements, nil is considered "falsey".
  ```
  string my_variable; // my_variable is currently nil
  if (my_variable == nil) {
      print("The variable is nil.");
  }
  ```

Operators

Sapphire provides a standard set of operators for performing calculations and logic.

Arithmetic Operators

Operator	Description	Example
+	Addition	10 + 5
-	Subtraction	10 - 5
*	Multiplication	10 * 5
/	Division	10 / 4
- (unary)	Negation	-10

String Concatenation

The + operator is also used to join two strings.

```
string first = "Sapphire";
string second = " Language";
print(first + second); // Outputs: Sapphire Language
```

Comparison and Equality Operators

These operators always evaluate to a bool (true or false).
Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
<	Less than	a < b

<=	Less than or equal	a <= b
>	Greater than	a > b
>=	Greater than or equal	a >= b

Logical Operators

Used to combine boolean expressions.

Operator	Description	Example
! (unary)	Logical NOT	!true -> false
and	Logical AND	(a > 5) and (b < 10)
or	Logical OR	(a > 5) or (b < 10)

Short-Circuiting: The and and or operators are short-circuited.

- For and, if the left-hand side is false, the right-hand side is never evaluated.
- For or, if the left-hand side is true, the right-hand side is never evaluated.

Scope and Blocks

A block is a series of statements enclosed in curly braces {}. A block creates a new scope. Variables declared inside a block are local to that block and are destroyed when the block is exited.

```
double global_var = 10;

if (true) {
   double local_var = 20; // Only exists inside this if-block
   print(global_var); // Prints 10 (can access outer scope)
   print(local_var);  // Prints 20
}

print(global_var); // Prints 10
// print(local_var); // This would cause an error, local_var is out of scope.
```

Chapter 4: Control Flow

If-Else Statements

The if statement executes a block of code if its condition is "truthy" (not false or nil). It can be followed by an optional else block, which executes if the condition is "falsey".

```
int score = 85;

if (score >= 90) {
```

```
    print("Grade: A");
} else if (score >= 80) {
    print("Grade: B"); // This block will be executed
} else {
    print("Grade: C or lower");
}
```

While Loops

The while loop repeatedly executes a block of code as long as its condition remains truthy.

```
int countdown = 3;

while (countdown > 0) {
    print("Countdown: " + valueToString(countdown));
    countdown = countdown - 1;
}

print("Liftoff!");
```

Chapter 5: Functions

Defining and Calling Functions

Functions are defined using the function keyword, followed by a return type, a name, a list of parameters in parentheses, and a body in curly braces.

```
// A function that takes two numbers and returns their product
function double multiply(double a, double b) {
    return a * b;
}

// Calling the function
double result = multiply(7, 6);
print(result); // Outputs: 42
```

Parameters and Return Values

Functions can accept zero or more parameters, each with a declared type. The return statement exits the function and provides a value back to the caller. The type of the returned value should match the function's declared return type.

The void Return Type

If a function does not return a value, its return type should be declared as void. A return; statement with no value can be used to exit a void function early. If a void function reaches the end of its body, it implicitly returns nil.

```
function void greet(string name) {
    if (name == nil) {
        print("Hello, stranger!");
        return; // Exit early
    }
    print("Hello, " + name + "!");
}

greet("Sapphire"); // Outputs: Hello, Sapphire!
greet(nil);        // Outputs: Hello, stranger!
```

Chapter 6: Data Structures

Arrays

Arrays are ordered, indexable collections of values.

Declaration and Initialization

Arrays are declared with a type followed by []. They are initialized using a comma-separated list of values inside square brackets.

```
int[] numbers = [10, 20, 30, 40];
string[] names = ["Alice", "Bob", "Charlie"];
```

Accessing and Modifying Elements

Elements are accessed using square bracket notation with a zero-based index.

```
string[] names = ["Alice", "Bob", "Charlie"];

// Get the first element
```

```
string first_name = names[0]; // "Alice"
print(first_name);

// Modify the third element
names[2] = "Charles";
print(names[2]); // Outputs: Charles
```

Chapter 7: Object-Oriented Programming

Classes and Instances

A class is a blueprint for creating objects. An instance is a specific object created from that blueprint.

```
// Define a simple class
class Point {
    double x;
    double y;
}

// Create an instance of the class
Point p = Point();
```

Fields

Fields are variables that belong to an instance of a class. They are declared inside the class body and are accessed using dot notation.

```
Point p = Point();

// Assign values to the fields of the instance 'p'
p.x = 10;
p.y = 20;

print("Point coordinates:");
print(p.x); // Outputs: 10
```

Methods and this

Methods are functions defined inside a class. They operate on the data of the instance they are called on. The special keyword this is used inside a method to refer to the instance itself.

```
class Counter {
    int value;

    // A method to initialize the counter
    function void init() {
        this.value = 0;
    }

    // A method to increment the value
    function void increment() {
        this.value = this.value + 1;
    }

    // A method to get the current value
    function int getValue() {
        return this.value;
    }
}

// Using the class
Counter c = Counter();
c.init();
c.increment();
c.increment();
print(c.getValue()); // Outputs: 2
```

Chapter 8: The Standard Library

Global Functions

Function Signature	Description
print(value)	Prints a string representation of any value to the console.

clock()	Returns the number of seconds since the program started.
parseDouble(string)	Converts a string to a number. Returns 0.0 on failure.
valueToString(value)	Converts any value into its string representation.
evaluate(string)	Executes a string of Sapphire code and returns the result.

Example:

```
string num_str = "123.45";
double num = parseDouble(num_str);
print(num + 10); // Outputs: 133.45

string code = "5 * 8";
double result = evaluate(code);
print(result); // Outputs: 40
```

The Math Library

Provides common mathematical functions.

- **Math.sqrt(number)**: Returns the square root of a number.
  ```
  print(Math.sqrt(144)); // Outputs: 12
  ```

The IO Library

Provides input/output functionality.

- **IO.readLine()**: Pauses execution and waits for the user to type a line of text into the console and press Enter. Returns the entered text as a string.
  ```
  print("What is your name?");
  string name = IO.readLine();
  print("Hello, " + name + "!");
  ```

The UI Library (ImGui Bindings)

This library allows creating graphical user interfaces.

Function Signature	Description
UI.Begin(string title, number flags)	Begins a new window.
UI.End()	Ends the current window.
UI.Text(string)	Renders text.
UI.Button(string label, ...)	Creates a button.
UI.SameLine()	Places the next widget on the same line.
UI.Separator()	Renders a horizontal line.
UI.Spacing()	Adds a small vertical space.
UI.SetNextWindowPos(number x, number y)	Sets the position for the next window.
UI.SetNextWindowSize(number w, number h)	Sets the size for the next window.
UI.PushStyleColor(id, r, g, b, a)	Changes the color of UI elements.
UI.PopStyleColor(count)	Restores the previous color.

Example:

```
// This code, when run in a loop, creates a simple window with a button.
function void updateUI() {
    if (UI.Begin("My Window", 0)) {
        UI.Text("This is a simple UI.");
        if (UI.Button("Click Me")) {
            print("Button was clicked!");
        }
    }
    UI.End();
}
```

Chapter 9: Advanced Topics

Language Internals

Sapphire is not an interpreted language in the traditional sense. It follows a more sophisticated, multi-stage process:

1. **Lexing:** The source code string is scanned and broken down into a series of tokens (e.g., KEYWORD, IDENTIFIER, NUMBER).
2. **Parsing/Compiling:** The stream of tokens is parsed to build an Abstract Syntax Tree (AST) and simultaneously compiled into a custom, low-level **bytecode**. This bytecode is a set of simple instructions optimized for a stack-based virtual machine.
3. **Execution:** The generated bytecode is run on the **Sapphire Virtual Machine (VM)**. The VM is a C++ program that reads and executes each bytecode instruction, manipulating its own stack to perform calculations and operations.
4. **Garbage Collection:** Sapphire includes an automatic **Mark-and-Sweep Garbage Collector (GC)**. The GC periodically runs to find and free memory for objects (like strings, functions, and class instances) that are no longer being used, preventing memory leaks.