Fibor

An Interpreted Language

Tristan Barnes

Contents

Overview

What is Fibor?

Fibor is an interpreted, object-oriented programming language with explicit typing. Fibor is also the name of the Swift framework that provides tools to run Fibor programs in a virtual context and to link said context to native Swift code (similar to what the CoreJavaScript framework does for JavaScript and Swift code).

Origins

Fibor was originally conceived as the language in which plugins for an iOS app, *CardDeck*[1], would be written. The interpreter logic was initially written in the *CardDeck* app project itself, but as the language, and interpreter, became more complex, the code associated with the interpreter was migrated to an independent framework.

Why object-oriented?

I'm comfortable with object-oriented paradigms. It's as simple as that.

[1] *CardDeck* is an iOS app that I am currently working on. The concept is that the app can be thought of as a "deck of cards" with which a user can play "any" card game that he/she desires with friends over Bluetooth. In order to do this, a user must write a program representing the rules of a game in Fibor, using Fibor libraries and objects provided by *CardDeck*. A game could then be interpreted on a host device and played by several people on connected devices.

Why explicit typing?

To experienced programmers, explicit typing can often feel like syntactic salt. Though, at times I agree, I believe that explicit typing can be a big time and headache saver when it comes to resolving (and ideally, preventing) bugs. I believe this is especially true for lesser experienced programmers, the target demographic for *CardDeck*.

See It In Action!

The rest of this document explores the syntax of the Fibor programming language. The current Fibor framework source code in Swift can be viewed on GitHub (https://github.com/Tristan67/Fibor). However, what's the fun in staring at lines of code? Some might argue "a lot", but for those that disagree, and even those that don't, an iOS project that actually puts the current Fibor framework to use can be downloaded from GitHub (https://github.com/Tristan67/Fibor-REPL). Enjoy!

Language Semantics

Fibor source code can be conceptualized as a tree composed of components of one of three types: expressions, statements, and definitions. Expressions are source code that represent a value, and can be evaluated to return said value (e.g. literals, method calls, operations, etc.). Statements are source code that perform some action (e.g. declaring a variable, returning a value from the scope of a method or block, performing certain statements based on a condition, etc.). Definitions are source code that define a class, extension, property, method, or operator.

Expressions

Expressions are source code that represent a value and return said value when they are evaluated. Any expression can be wrapped in a pair of parenthesis, which is equivalent to just the expression inside (this is done to use the return value of a method call as part of another expression, or to make more clear if an operator is postfix, infix, or prefix when they appear adjacent in code).

Assignments

After a variable has been declared, the value it points to can be changed using the *assignment operator expression*. To do this, simply type the name of the defined variable, followed by the assignment operator (**`=`**), followed by an expression representing the new value.

```
<variable name> = <expression>
```

The new value must be of a valid type (for objects, this means that the value must be an object of the same class or of a subclass of the class assigned to the variable when it was declared).

Bool Literals

Objects of type **`Bool`** can be initialized using one of the two boolean literals: **`true`** or **`false`**.

Integer Literals

Objects of type **`Int`** can be initialized using an integer literal. Integer literals are represented by one or more numerical characters (0 - 9).

String Literals

Objects of the type **`String`** can be initialized using a string literal. String literals are a series of characters, enclosed by a pair of double quotes (").

References

Reference expressions represent the underlying value of a variable. A reference expression is simply the name of a variable.

Initialization

Object instances are created, or initialized, by calling a class's initializer. In order to call a classes initializer, simply type the at sign (**`@`**) immediately followed by the class name of the desired object and then the **`init`** keyword.

```
@<class name> init
```

The process of initializing an object can be thought of as calling the class's global "init" method.

Calling a Method

In order to call a method of an object, simply write the name of the method behind an expression representing an object (typically a reference expression).

```
 <object expression> <method name>
```

In real code, this might look like:

```
 someObjectVariable someMethod
```

If the method expects one or more arguments simply list out the arguments, separated by a comma, following the name of the method and a colon. For a method call with 3 arguments, this might look like:

```
someObjectVariable someMethod: argA, argB, argC
```

Parenthesis can be used to indicate different . For example, the following code snippet calls the method call **`doSomething`** on **`someObject`** with an argument of the value of the **`name`** property of `anotherObject`.

```
someObject doSomething: anotherObject.name
```

In the example below, the **`name`** property of the object returned by calling **`doSomething`** on **`someObject`** with an argument of the value of **`anotherObject`** is being accessed.

```
(someObject doSomething: anotherObject).name
```

If a method does not expect any arguments and some operation (e.g. accessing a property) is to be performed on the returned value, the call must still be wrapped in a pair of parenthesis. This is a syntax error:

```
someObjectVariable someMethod.name
```

Block Literals

Blocks are an instance type that represent a statement and can be passed around in code. A block literal is indicated by the carrot operator (**`^`**). Following the carrot operator, is a capture list. A capture list is a list of values that a block should capture from its surrounding scope so that they can be used within the block's statement. A capture list may be omitted if no values should be captured. To use a capture list, use a pair of square brackets to wrap a name and value pair separated by a comma, where the name of represents the captured value in the block's statement.

Following the capture list, or carrot operator if there is no capture list, is a pair of parenthesis. Inside the parenthesis is a series of name and type pairs separated by a comma, where the name represents the respective parameter in the block's statement and the type is the expected type of the respective argument. If no arguments should be passed to a block, the block is written with an empty pair of parenthesis.

Lastly, the parenthesis is followed by the return operator (**->**), which is followed by the return type of the block.

```
^[<name> <value>, <name> <value>, <...>](<name> <type>, <name>
     <type>, <...>) -> <type> <statement>
```

Here is an example of a block literal that has one captured value and one parameter of type `Int` and returns an object of type **`Int`**:

```
^[c capturedValue](param Int) -> Int return c + param
```

In order to perform more than one statement when a block is invoked, use a *statement group*.

Use the *void symbol* (**#**) to indicate that a block does not return anything. The block literal below does nothing when it is invoked.

```
^() -> # return
```

Invocating a Block

To get a block to perform its underlying statement, the block must be invoked. To invoke a block, write a pair of parenthesis behind an expression that represents a block instance (e.g. a block literal, a variable name, a method call that returns a block) with the expected arguments inside separated by a comma. (Just use an empty pair of parenthesis to invoke a block that does not expect any arguments).

In the snippet below, the block instance referenced by the variable block is invoked without any arguments.

```
block()
```

In order to invoke a block literal, the block's statement must be a group statement, or a statement that contains a group statement as the last statement. Otherwise, Fibor will think that the returned value is being invoked. Here is an example:

```
^(param Int) -> Int {
    return param
}(4)
```

In this example, the value returned from a method call is being invoked:

```
(someObject makeBlock)()
```

Inspecting Properties

Object instances can hold onto and reference other values via their properties. In order to reference the underlying value of an object instance, type the a dot and the name of the desired property immediately after an expression that represents the object instance.

```
<expression>.<property name>
```

Globals

Globals refer to properties and methods of a class, instead of an instance of said class. In order to access a global property, simply type the at sign (`@`) immediately followed by the desired class name and then access the property with the same syntax for accessing properties of an object instance.

```
@<class name>.<property name>
```

In order to call a global property, again, simply type the at sign (`@`) immediately followed by the desired class name and then call the desired method using the same syntax as for an object instance.

```
@<class name> <method name>
```

This Literal

In order to reference the object instance that is performing a method call from within said method's implementation, use the **`this`** keyword.

Super Call

There are times when a subclass's overriding implement of a superclass's method should also call the superclass's implementation of that method. To do so inside the subclass's implementation, simply type the **`super`** keyword, followed by the name of the method.

```
super <method name>
```

Statements

Statements are source code that perform some action. They do not return a value. Statements are found within definitions, other statements, and block literals.

Variables (declaration)

In order to declare a variable in Fibor, use the **`var`** keyword, followed by the name of the variable, the type, and an initial value from an expression, preceeded by the assignment operator.

```
var <name> <type> = <expression>
```

For example, in order to declare a variable called **`number`** of type **`Int`** and with an initial value of four, represented by an integer literal, one would type:

```
var number Int = 4
```

In Fibor, declarations are statements and do not return a value, like expressions do.

Group Statements

There are cases where it makes since for multiple statements to be performed at the same time, as a group. In order to create a *group statement*, enclose one or more statements with a pair of curly-braces, separated by newlines.

```
{
    <statement 1>
    <statement 2>
    <statement 3>
}
```

Conditions (if-else)

The *if* statement is used if another statement should only be performed if a certain condition is met. To use an *if* statement, type the **`if`** keyword, followed by an expression that returns a **`Bool`** type, followed by the statement to be performed, followed by the **`else`** keyword, followed by the alternative statement.

```
if <Bool expression> <statement> else <statement>
```

If no statement should be performed if the expression condition is false, the **`else`** keyword and alternative statement may be omitted.

Unlike in other languages, *if* statements in Fibor are not followed by curly-braces as part of their structure. However, if multiple statements should be performed based on a condition, a *group statement* should be used as the *if* statement's statement.

```
if <Bool expression> {
    <...>
}
```

Looping (while)

In order to perform a statement several times, while a Bool expression is true, use a *while loop*. *While loops* are started with the **`while`** keyword, followed by a condition expression of type **`Bool`**, followed by the statement to be performed.

```
while <Bool expression> <statement>
```

Calling Methods (do)

Since method calls are expressions, they always return a value. However, there are times when a method should be called so that its implementation is performed, but the value it returns is not important and should not be kept in memory. In order to evaluate an expression, but to not hold a reference to the value it returns, use a *do* statement. A do statement is begun by the do keyword and followed by the expression that should be evaluated.

```
do <expression>
```

Returning

Us the *return* statement to return a value from a method or block. To use a return statement, simply write the **`return`** keyword, followed by the expression that evaluates to the value that should be returned.

```
return <expression>
```

If a method or block is not expected to return any value, but it should still return before performing any more statements, the **`return`** keyword can simply be written without any following value.

Definitions

Definitions are source code that define a class, whether it be the whole class, using a *class* definition, or simply extending a class, using an *extension* definition.

Definitions are broken down into two categories: primary and secondary. Primary definitions include *classes* and *extensions*, which are made up of multiple secondary definitions, which include *properties*, *methods*, *operators*, and *initializers* (only classes, not extensions, can have a single initializer).

Properties

Properties are what allow object instances to hold onto other values. To define a property, write the prop keyword, the name of the property, the type of the property, an equals sign (**`=`**), and an expression that evaluates to the initial value of the property.

```
prop <name> <type> = <expression>
```

Use the **`global`** and/or **`private`** keyword before the **`prop`** keyword in order to define a global and/or private property, respectively.

Methods

In order to define a method, type the **`meth`** keyword followed by the name of the method. Immediately after the name, a pair of parenthesis encloses a list of name and type pairs, representing the name of each parameter and its associated type. (Type an empty pair of parenthesis to express that a method takes no parameters). The parenthesis is followed by the return operator (**->**), which is followed by the return type of the method, which is followed by the statement that the method should perform when it is called.

```
meth <name>(<name> <type>, <name> <type>, <...>) -> <type>
<statement>
```

If a method should perform more than one statement in its implementation, use a group statement.

Use the **`global`** and/or **`private`** keyword before the **`meth`** keyword in order to define a global and/or private method, respectively.

Infix Operators

There are three different types of operators: infix, prefix, and postfix, each of which has its own definition syntax. In order to define an infix operator, type the **`infix`** keyword followed by the operator's symbols (**+**, **-**, *****, **/**, **%**, **=**, **!**, **&**, **|**, **^**, **~**, **<**, **>**, **?**, ****, **'**, **:**, and **#**). Immediately after the operator symbol, a single pair of parenthesis encloses a name and type pair, representing the name of the right-hand-side parameter of the operation and its associated type. (The object instance on the left-hand-side of the operation is the object that that is performing the operation and can be accessed within the implementation of the operation using the **`this`** keyword). After the closing parenthesis is the return operator (**->**), which is followed by the return type of the operator, which is followed by the statement that the operator should perform.

```
infix <symbols>(<name> <type>) -> <type> <statement>
```

Use the **`global`** and/or **`private`** keyword before the **`infix`** keyword in order to define a global and/or private operator, respectively.

Prefix Operators

In order to define a prefix operator, type the **`prefix`** keyword followed by the operator's symbols (**+**, **-**, *****, **/**, **%**, **=**, **!**, **&**, **|**, **^**, **~**, **<**, **>**, **?**, \, **'**, **:**, and **#**), the return operator (**->**), the return type of the operator, and then is followed by the statement that the operator should perform. (Prefix operators take no arguments).

```
prefix <symbols> -> <type> <statements>
```

Use the **`global`** and/or **`private`** keyword before the **`prefix`** keyword in order to define a global and/or private operator, respectively.

Postfix Operators

The same syntax used for defining a prefix operator is used to define a postfix operator, except the **`postfix`** keyword is used instead.

```
postfix <symbols> -> <type> <statements>
```

Use the **`global`** and/or **`private`** keyword before the **`postfix`** keyword in order to define a global and/or private operator, respectively.

Initializers

After an object has been initialized, the initializer of the class of said object, and of every superclass of said object, starting with that of the most super, is called. Every class must have a single initializer had part of its definition. In order to define an initializer, simply type the **`init`** keyword followed by the statement that the initializer should perform.

```
init <statement>
```

Initializers do not return any instance. Therefore, in order to return from an initializer early, simply use the return statement without any value.

Initializers cannot be modified by the **`global`** or **`private`** keywords.

Classes

Classes are the instructions for defining all data types in the Fibor programming language (except for blocks). In order to define a class, type the **`class`** keyword, the class name, a colon, the name of the class's superclass, and a pair of curly braces that enclose all the secondary definitions that define said class.

```
class <name>: <supername> {
    <secondary definition>
    <secondary definition>
    <secondary definition>
    ...
}
```

All class definitions must include one, and only one, initializer.

If a class has no superclass, simply omit the colon and superclass name.

Extensions

There may be times when the functionality of an already defined class should be extended to include more. The extension definition is used to do just that. In order to define an extension, type the **`extension`** keyword followed by the name of the class that is to be extended and a pair of curly braces that enclose secondary definitions that should be added to the class.

```
extension <name> {
    <secondary definition>
    <secondary definition>
    <secondary definition>
    ...
}
```

Extensions cannot include an initializer since the class's initializer was already defined when the class was defined.

An extension can only be typed in source code that follows the class's initial definition.

What’s Next?

The syntax of the Fibor language, as well as the Fibor framework, are still just a work in progress. Here is a list of things that are still being implemented into the language:

- Protocols
- Generic types
- Arrays (the above is needed to do so)