SWIFT

Rapid Learning & Just In Time Support

CONTENT

1 Introduction

Info

- This book contains tutorials about syntax of SWIFT Programming Language.
  Syntax is the set of rules that you must obey in order to write proper code in specified language.
- Tutorials are standalone and minimalistic focusing only at one problem at the time.
  Each tutorial includes working example with a complete code needed to test discussed functionality.

How to use this book

- This book can be used as an introduction to SWIFT Programming Language covering all of the basic core functionalities.
- Book is also intended as Just In Time Support so that user can learn what it needs when it needs it.
- This is why tutorials are standalone and minimalistic focusing only at one problem at the time.

Meaning of symbols used in tutorials

- In most tutorials you will find symbols like [R], [E] and [C] whose meanings are described in following table.

Symbols

SYMBOL	NAME	DESCRIPTION
[R]	Reference	Link to reference which was used while creating tutorial.
[E]	Error	Link to solution for an error that might occur while following tutorial.
[C]	Create application	Link to a tutorial which shows how to create an application in order to test tutorial's code.

Chapters organization

Chapter 1

- This chapter explains how to use online compiler to create SWIFT application.

Chapter 2

- This chapter contains tutorials that explain data related syntax covering basic structures used to create and store actual data like data types, literals, variables, constants and optionals.

Chapter 3

- This chapter contains tutorials that explain basic syntax which doesn't include object oriented syntax.

Chapter 4

- This chapter contains tutorials that explain objects oriented syntax of working with classes, objects, structures etc.

Chapter 5

- This chapter contains tutorials on using different tools that might prove helpful while working with SWIFT like online compilers.

Chapter 6

- This chapter contains additional informations.

1.1 Create SWIFT Application

Info

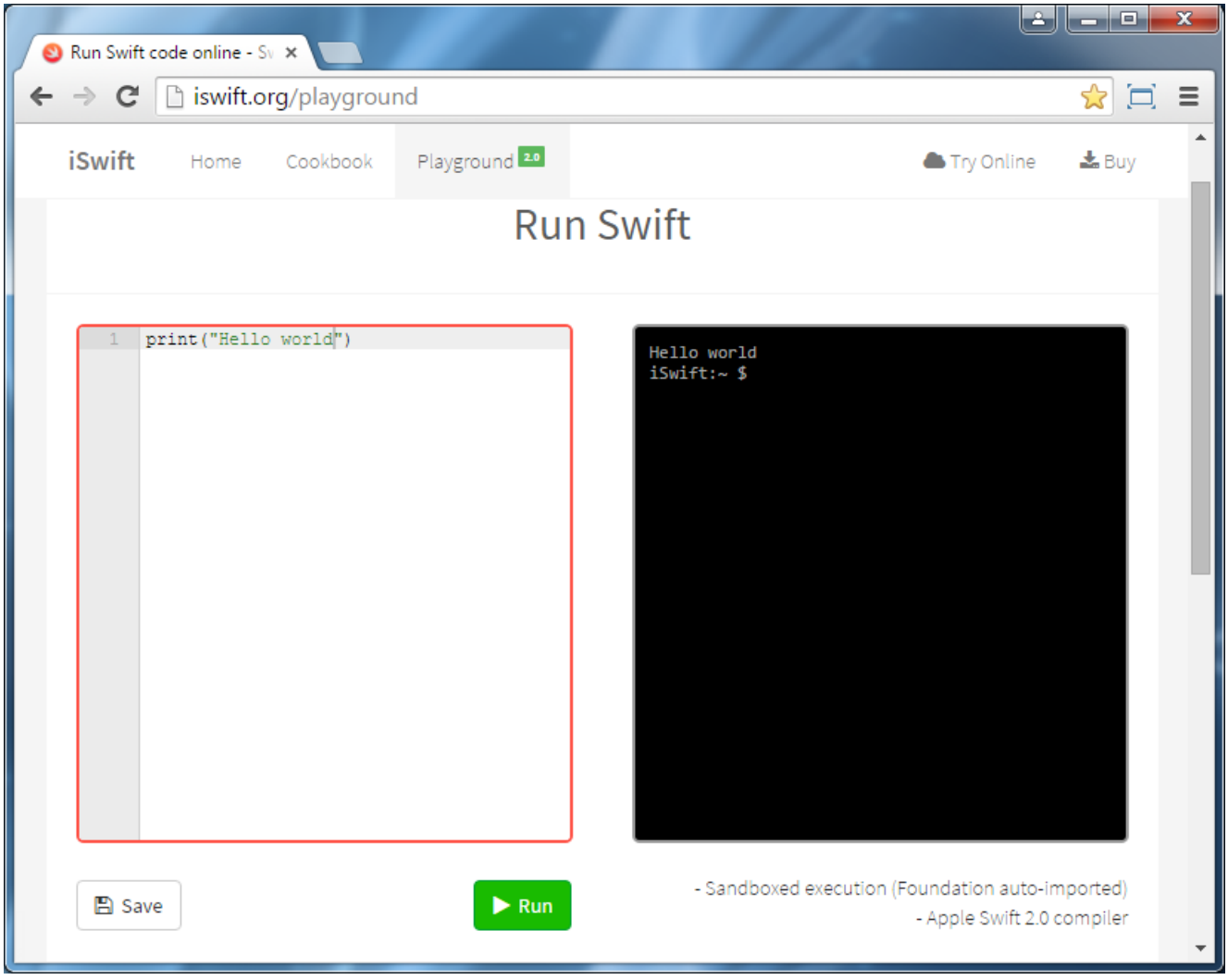
- Following tutorials shows how to create SWIFT application
  - Using online compiler

1.1.1 Using online compiler

Info [R]

- This tutorials shows how to create SWIFT application using online compiler.
- This approach is great for learning and testing syntax.
- Online compilers can be used on any operating system since they run in a Web Browser.
- Here is a list of online compilers (be aware of SWIFT version they support)
  - http://iswift.org/playground (Swift Version 2.0)
  - http://www.runswiftlang.com (Swift Version 1.2)
  - http://swiftstub.com (Swift Version 1.2)

http://iswift.org/playground



Procedure

- http://iswift.org/playground
- (Copy content of Test.swift)
- Run

Test.swift

```
print("Hello World")
```

output

```
Hello World
```

2 Predefined data types and their literals

Info

- Following tutorials focus on predefined data types
  - showing which are available
  - explaining how to use them
  - explaining how to use literals to create data of these types

2.1 Predefined data types

Info

- Following tutorials show how to work with **predefined** data types.
- Predefined data types
  - are implemented as Structures
  - can be categorized as
    - Scalar types that can store single value (Nil, Bool, Int, UInt, Float, Double, String)
    - Collection types that can store multiple values of the same type (Set, Array, Dictionary)

2.1.1 Scalar

Info

- Following tutorials show how to work with scalar data types.
- Scalar datatypes
  - are implemented as Structures
  - can store single value

Scalar data types

TYPE	DESCRIPTION
Nil	Can only have nil value, It can only be assigned to optional.
Bool	Can only have logical `true` or `false` values.
Int	Different signed integer data types can store signed integers of various sizes
UInt	Different unsigned integer data types can store unsigned integers of various sizes
Float	Represents real number as 32-bit floating-point number.
Double	Represents real number as 64-bit floating-point number.
String	Represents sequence of characters treated as a continuous group

2.1.1.1 Nil

Info

- This tutorial shows how to work with nil data type.
- Nil data type
  - can only have `nil` value
  - can be assigned to optional
  - is used to initialize optional which is declared without assigning value to it
  - can be created using Nil Literal

Example [C]

- Example uses nil literal to create nil data of value `nil` which is then stored in optional variables which can have nil data.

Test.swift

```
//TEST VARIABLES.
var age   :Int?        //Optional variable is initialized to nil data.
var weight:Int? = nil  //Optional variable is set         to nil data.

//DISPLAY VARIABLES.
print(age)
```

output

```
nil
```

2.1.1.2 Bool

Info

- This tutorial shows how to work with Bool data type.
- Bool data type
  - can only have two possible values logical `true` or `false`
  - can be created using Boolean Literal

Example [C]

- Example uses nil literal to create nil data of value `nil` which is then stored in optional variables which can have nil data.

Test.swift

```
//TEST VARIABLES.
var access1      = true
var access2:Bool = false

//DISPLAY VARIABLES.
print(access1)
```

output

```
true
```

2.1.1.3 Int

Info

- This tutorial shows how to work with different signed integer data types.
- Int data type
  - can store signed integers of various sizes as shown in the table below
  - can be created using Integer Literal
- Int is alias for Int32 or Int64 depending if the platform is 32-bit or 64-bit respectively.

Singed integers

INTEGER SIZE	DESCRIPTION
Int	Alias for Int32 or Int64 if platform is 32-bit or 64-bit respectively
Int8	Signed integer with 8 bits
Int16	Signed integer with 16 bits
Int32	Signed integer with 32 bits
Int64	Signed integer with 64 bits

Example [C]

- Example shows how to declare variables of different unsigned integer sizes.

Test.swift

```
//TEST VARIABLES.
var a:Int   = -50
var b:Int8  = +50
var c:Int16 =  50
var d:Int32 =  50
var e:Int64 =  50

//DISPLAY VARIABLES.
print(a)
```

output

```
-50
```

2.1.2 Collection

Info

- Following tutorials show how to work with collection data types.
- Collection data types
  - are implemented as Structures
  - allow for multiple items of the same type to be bundled together

Collection data types

TYPE	DESCRIPTION
Set	Unordered container of unique elements
Dictionary	Unordered container of key-value pairs with unique keys
Array	Ordered container of indexed elements starting from index 0

2.1.2.1 Set

Info

- This tutorial shows how to work with Set data type.
- Set data type is
  - an **unordered** container
  - that stores unique elements of the same data type
- Set data can be created using Array Literal in different notations.

Example [C]

- In this example we use `"USD"` and `"EUR"` as keys for `"US Dollar"` and `"Euro"` elements respectively.

Test.swift

```
//TEST VARIABLES.
var cars1:Set         = ["FIAT", "AUDI"] //Implicit element data type. Create set with array literal.
var cars2:Set<String> = ["FIAT", "AUDI"] //Explicit element data type. Create set with array literal.

//DISPLAY VARIABLES.
print(cars1)
```

output

```
["FIAT", "AUDI"]
```

2.1.2.2 Dictionary

Info

- This tutorial shows how to work with Dictionary data type.
- Dictionary data type
  - is an **unordered** container of elements
    - of the same data type
    - that have unique key of the same data type (which can be different from element's data type)

Example [C]

- In this example we use `"USD"` and `"EUR"` as keys for `"US Dollar"` and `"Euro"` elements respectively.

Test.swift

```
//TEST VARIABLES.
var employees1                          = [12: "John", 23: "Lucy"]
var employees2:Dictionary<Int, String> = [12: "John", 23: "Lucy"]

//DISPLAY VARIABLES.
print(employees1)
```

output

```
[12: "John", 23: "Lucy"]
```

2.1.2.3 Array

Info

- This tutorial shows how to work with Array data type.
- Array data type
  - is an **ordered** container of elements
    - of the same data type
    - that have unique integer index starting from 0
  - can be created using Array Literal

Example [C]

- Example uses array literals to create array data containing multiple data.

Test.swift

```
//TEST VARIABLES.
var cars1                = ["AUDI", "FIAT"] //Array literal. Implicit type.
var cars2:[String ]      = ["AUDI", "FIAT"] //Array literal. Explicit type.
var cars3:[String?]      = ["AUDI", nil   ] //Array literal. Explicit type. String Optional.
let cars4:Array<String?> = ["AUDI", nil   ] //Array literal. Explicit type. String Optional.

//DISPLAY VARIABLES.
print(cars1)
```

output

```
["AUDI", "FIAT"]
```

2.2 Literals

Literals

- **Literal** is **string representation** of both **data type** and **data value**.
- Literal can be used to create data of predefined data types like
  - "Ivor Online" is string literal since it represents string data type of value Ivor Online
  - 'Ivor Online' is also string literal, but using single quotes notation, since it also represents string data type
  - 10 is integer literal since it represents integer data type of value 10
  - 0x0A is also integer literal, but using different notation, since it also represents integer data type of value 10
- **Literal notations** are different ways of constructing literals that represent the same data type.
  Following integer literals represent the same data type (integer) using different literal notations: 10, +10, 012 or 0xA.
- Literals can be categorized by the data type they represent as shown in below table.

Literal types

LITERAL	EXAMPLE	DESCRIPTION
Nil	`null`	Represents null data type with value
Boolean	`true`	Represents boolean data type and value
Integer	`10`	Represents integer data type and value
Float	`0.5678`	Represents floating point data type and value
String	`"text"`	Represents string data type and value
Array	`["AUDI", "FIAT"]`	Represents string data type and value

2.2.1 Nil

Info

- This tutorial shows how to work with nil literal.
- Nil literal
  - represents nil data type
  - is written by using case sensitive `nil`
  - can be written in just one notation (format) as shown below

Example [C]

- In this example we use nil literal to create nil data types which is then stored in `age` optional.

Test.swift

```
//CASE SENSITIVE NIL LITERAL.
var age:Int? = nil

//DISPLAY VARIABLES.
print(age)
```

output

```
nil
```

2.2.2 Boolean

Info

- This tutorial shows how to work with Boolean literal.
- Boolean literal
  - represents boolean data type and value
  - is written by using case sensitive `true` or `false`
  - an be written in just one notation (format) as shown below

Example [C]

- In this example we use `true` and `false` literals to create boolean data types which are then stored into variables.

Test.swift

```
//TEST VARIABLES.
var access1      = true
var access2:Bool = false

//DISPLAY VARIABLES.
print(access1)
```

output

```
true
```

2.2.3 Integer

Info

- This tutorial shows how to work with Integer literal.
- Integer literal
  - represents integer data type and value
  - can be written in following notations
    - Decimal Notation where integer is written as decimal number in base of 10 like `-41`
    - Binary Notation where integer is written as binary number in base of 9 like `-0b101001`
    - Octal Notation where integer is written as octal number in base of 8 like `-051`
    - Hexadecimal Notation where integer is written as hexadecimal number in base of 16 like `-0x29`

Example [C]

- In this example we use different integer literals to create integer data types which are then stored into variables.

Test.java

```
//USE INTEGER LITERAL DO STORE INTEGER DATA INTO VARIABLE.
var age = -41           //Decimal     notation (base 10).
    age = -0b101001     //Binary      notation (base 2 ).
    age = -0o51         //Octal       notation (base 8 ).
    age = -0x29         //Hexadecimal notation (base 16).

//DISPLAY BOOLEAN DATA.
print(age)
```

output

```
-41
```

2.2.3.1 Decimal Notation

Info [C]

- Integer literal can be given in decimal notation (base 10).

Test.swift

```
//USE INTEGER LITERAL DO STORE INTEGER DATA INTO VARIABLE.
var age =  41    //Positive.
    age = +41    //Positive.
    age = -41    //Negative.

//DISPLAY INTEGER DATA.
print(age)
```

output

```
-41
```

2.2.3.2 Binary Notation

Info [C]

- Integer literal can be given in binary notation (base 2).

Test.swift

```
//USE INTEGER LITERAL DO STORE INTEGER DATA INTO VARIABLE.
var age =  0b101001    //Positive.
    age = +0b101001    //Positive.
    age = -0b101001    //Negative.

//DISPLAY INTEGER DATA.
print(age)
```

output

```
-41
```

2.2.3.3 Octal Notation

Info [C]

- Integer literal can be written in hexadecimal notation (base 9).

Test.java

```
//USE INTEGER LITERAL DO STORE INTEGER DATA INTO VARIABLE.
var age =  0o51    //Positive.
    age = +0o51    //Positive.
    age = -0o51    //Negative.

//DISPLAY INTEGER DATA.
print(age)
```

output

```
-41
```

2.2.3.4 Hexadecimal Notation

Info [C]

- Integer literal can be given in hexadecimal notation (base 16).

Test.java

```
//USE INTEGER LITERAL DO STORE INTEGER DATA INTO VARIABLE.
var age =  0x29    //Positive.
    age = +0x29    //Positive.
    age = -0x29    //Negative.

//DISPLAY BOOLEAN DATA.
print(age)
```

output

```
-41
```

3 Constants, Variables and Optionals

Info

- Following tutorials explain basic SWIFT syntax.
- Later chapter Object Oriented Syntax explains syntax related to classes and objects.
- Syntax of a programming language is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language.

3.1 Constants

Info **[R]**

- Following tutorials show how to work with constants.
- Constant
  - has value which cannot be changed once it is set (unlike variable which can change its value any number of times)
  - is declared using keyword `let`

3.1.1 Declare

Info

- This tutorial shows how to declare a constant.
- Constant is declared
  - using keyword `let`
  - followed by required constant's name
  - followed by optional data type (not needed if value is given since data type can be referred from the value)
  - followed by optional assignment of value (if value is not specified data type must be specified)

Syntax

```
let name : [DataType] = [value]
```

Example [C]

- This example shows how to declare and reference constants.

Test.swift

```
//DECLARE CONSTANTS.
let const1 = 3.141592    //Data type is reffered from a value.
let const2:Int           //Uninitialized. Can't be used before setting a value.

//REFERENCE INITIALIZED CONSTANT.
print(const1)

//REFERENCE UNINITIALIZED CONSTANT.
const2 = 10              //Set value to uninitialied constant before using it.
print(const2)            //Results in error if no value was given.
```

output

```
3.141592
10
```

3.1.1.1 Name

Info

- This tutorial shows hot to define constant's name.
- Constant's name
  - is used to reference constant when you need to set or retrieve its data
  - must be unique inside a scope where it is declared
  - must follow the same naming rules as identifiers

Example [C]

- This example shows how to
  - specify constant's name when declaring constant
  - use constant's name to reference constant
    - when storing value
    - when retrieving value

Test.swift

```
//DECLARE CONSTANT.
let name : String

//REFERENCE CONSTANT TO STORE VALUE.
name = "John"

//REFERENCE CONSTANT TO RETRIEVE VALUE.
print(name)
```

output

```
John
```

3.1.1.2 Data type

Info

- This tutorial shows how to declare constant's data type (type annotation).
- Data type
  - must be specified if constant is not given initial value during declaration
  - is optional if constant is given initial value during declaration
    - if data type is not declared it will be implied from the value being stored into constant
    - if data type is declared it must match data type of the value being stored into constant

Syntax

```
var constnatName : DataType           //Declare data type without assigning value.
var constnatName : DataType = value   //Declare data type and assigning value of the same data type.
var constnatName            = value   //Data type is implied from the value being stored.
```

Example [C]

- This example shows different ways of declaring variable.

Test.swift

```
//DECLARE VARIABLES.
let height : Int                      //Declare data type. Can't use it before some value is assigned.
let age    : Int = 50                 //Declare data type and assigning value of the same data type.
let name         = "John"             //Data type is implied from the value being stored.

//REFERENCE VARIABLES.
print(age )
print(name)
```

output

```
50
John
```

3.2 Stored Variables

Info [R]

- Following tutorials show how to work with stored variables.
- Stored variable
  - has value which can be changed any number of times (unlike constants which always hold initial value)
  - is declared using keyword `var`
  - is equivalent to term variable as used in other programming languages
  - is called stored to distinguish it from Computed Variables which don't store data
- In the reminder of this chapter Stored Variables are referred as just variables since this is more general name for this type of construct as used in other programming languages.

What is variable

- Memory location is section of memory specified by
  - memory address at which it starts
  - size it occupies
- Variable is a named memory location reserved to store some data.
- Variable is specified by
  - Identifier is the name of the variable (the name of memory location)
  - Reference is memory address at which memory location starts
  - Data Type is type of data that is stored in that memory location defining what these zeroes and ones represent
  - Data Value is the number (zeroes and ones) stored at that memory location, also known as just Data or Value.

Example

- This example shows how to declare and reference String variable which has following characteristics
  - Identifier is `name`
  - Data Value is `"John"` and it is defined with String literal
  - Data Type of data that is actually being stored is defined by String literal `"John"` and it is of String data type
  - Address is assigned automatically by computer and is unknown to us
  - = is assignment operator used to store value into variable
- With string literal `"John"` we have defined both
  - data value (zeroes and ones that should go into memory location)
  - that these zeroes and ones are of String data type

Test.swift

```
//DECLARE VARIABLES.
var name = "John"                       //Simultaneously declare variable and assign data to it.
var age  = 50                           //Simultaneously declare variable and assign data to it.

//REFERENCE VARIABLES.
name = "Bob"                            //Reference variable to assign data to it.
print(name)                             //Reference variable to retrieve data and display it.
```

Strongly/Loosely typed language

- SWIFT can be used both as
  - strongly typed language which means that when you declare variable you can specify its data type
  - loosely typed language which means that when you declare variable you don't need to specify its data type in which case variable data type depends on the data that you store in the variable
- Variable can't change its data type once it was defined either explicitly or implicitly.
  By declaring variable's data type you are saying: "I intend that this variable holds only data of this data type. Warn me if I try to put different data type in it". This allows compiler to raise error if you for instance try to store String into Integer.

Internal representation of variables

- You can imagine that internally SWIFT maintains a table of variables as shown below.
- When new variable is declared
  - new record is inserted containing all informations about the variable
  - if needed new memory location is allocated and data is stored into it
- Table also shows how two variables can reference the same memory location therefore sharing the same peace of data.

Variables

IDENTIFIER	REFERENCE	DATA TYPE	MEMORY SIZE
name	0xE45FF89	String	24
age1	0xFAB43CE	Integer	2
age2	0xFAB43CE	Integer	2

Memory locations

ADDRESS	DATA
0xE45FF89	"John"
0xFAB43CE	50

Identifier

- Variable Identifier (name) is a sequence of characters used to reference variable when we want to
  - declare variable
  - reference variable
    - to assign new data
    - to retrieve current data

Data Type

- Data type tells computer how to intepret number stored in memory location.
- This is needed because number stored in a memory location is just zeros and ones and the same combination of zeros and ones can have different meaning depending on what they are supposed to represent.
- For instance the same combination of zeros and ones can represent either integer 68656c6c6f or string "Hello".
- So in order for print(myvariable) function to properly display data stored in memory location it needs to know if data represents integer or string so that it can display either 68656c6c6f or "Hello" respectivly on the screen.
- Data type also defines the size of memory location so that computer would know how many zeros and ones to take beginning from the address of the memory location.
- There are different ways of defining data, defining both data type and data value, like using
  - literals `10`, `+10`
  - result of mathematical operation `10 + "10"`
  - result of concatenation `"John is" + age + "years old."`

Literal

- **Literal is string representation of both data type and value**
  - "Ivor Online" is string literal since it represents string data type of value Ivor Online
  - 'Ivor Online' is also string literal, but using single quotes notation, since it also represents string data type
  - 10 is integer literal since it represents integer data type of value 10
  - 0x0A is also integer literal, but using different notation, since it also represents integer data type of value 10
- Literal notations are different ways of constructing literals that represent the same data type.
  For example following integer literals represent the same data type (integer) using different literal notations
  10, +10, 012 or 0xA.
- Literals are characterized by the data type they represent like boolean, null, integer, float, string or array.
  Not all data types can be defined using literals. Objects and resources data types can't be written as literals.

3.2.1 Declare

Info

- This tutorial shows how to declare a stored variable.
- Stored variable is declared
  - using keyword `var`
  - followed by required stored variable's name
  - followed by optional data type (not needed if value is given since data type will be referred from the value)
  - followed by optional assignment of value (if value is not specified data type must be specified)

Syntax

```
var name : [DataType] = [value]
```

Example [C]

- This example shows how to declare global variable of data type String.
- Initial value is set to `"John"` and then later changed to `"Bob"`.

Test.swift

```
//DECLARE STORED VARIABLES.
var name = "John"                       //Data type is referred from the value.
var age:Int                             //Uninitialized. Can't be used before setting a value.

//REFERENCE INITIALIZED VARIABLE.
print(name)

//REFERENCE UNINITIALIZED VARIABLE.
age = 50                                //Set value to uninitialized variable before using it.
print(age)                              //Results in error if no value was given.
```

output

```
Bob
```

3.2.1.1 Name

Info

- This tutorial shows hot to define variable name.
- Variable name
  - is used to reference variable when you need to set or retrieve its data
  - must be unique inside a scope where it is declared
  - must follow the same naming rules as identifiers

Syntax

```
var variableName = value   //Data type is implied from the value being stored.
```

Example [C]

- This example shows how to
  - specify variable's name when declaring variable
  - use variable's name to reference variable

Test.swift

```
//DECLARE STORED VARIABLE.
var name : String

//REFERENCE STORED VARIABLE TO STORE VALUE.
name = "John"

//REFERENCE STORED VARIABLE TO RETRIEVE VALUE.
print(name)
```

output

```
John
```

3.2.1.2 Data type

Info

- This tutorial shows how to declare variable's data type (type annotation).
- Data type
  - must be specified if variable is not given initial value during declaration
  - is optional if variable is given initial value during declaration
    - if data type is not declared it will be implied from the value being stored into variable
    - if data type is declared it must match data type of the value being stored into variable

Syntax

```
var variableName : DataType            //Declare data type without assigning value.
var variableName : DataType = value    //Declare data type and assigning value of the same data type.
var variableName            = value    //Data type is implied from the value being stored.
```

Example [C]

- This example shows different ways of declaring variable.

```
//DECLARE VARIABLES.
var height : Int                       //Declare data type. Can't use it before some value is assigned.
var age    : Int = 50                  //Declare data type and assigning value of the same data type.
var name         = "John"              //Data type is implied from the value being stored.

//REFERENCE VARIABLES.
print(age )
print(name)
```

output

```
50
John
```

4 Basic syntax

Info

- Following tutorials explain basic SWIFT syntax.

4.1 Comments

Info

- Comments are pieces of text inserted into source code to clarify it.
- They are not instructions for the computer and are therefore ignored by execution engine.
- Specially formatted comments can be used to generate documentation directly from the source code.
- SWIFT supports single and multiline comments and one types of multiline (block) comment.

Comments

TYPE	EXAMPLE
Single line	`//Single line comment`

Multi line	`/* Multi line comment.`
	`  Second line.`
	`*/`

Multi line nested	`/* First  line of outer comment`
	`  Second line of outer comment`
	`  /* First  line of inner comment`
	`  Second line of inner comment`
	`  */`
	`*/`

4.1.1 Single Line

Info

- Single Line comments can't span over multiple lines.
- Start of single line comment is indicated by especially reserved character `//` or combination of characters `//`.
- End of single line comment is indicated by the end of the line.
- Benefit of single line comments is that they do not need to be terminated with terminator keyword.
- Downside of single line comment is that creating longer comments requires placing indicator for start of comment at the beginning of each line.

Example [C]

- Single line comment is indicated by starting comment with double slash character combo `//`.

Test.swift

```
//This example shows usage of Single line comment.
//These comments must start with double slash character combo .
//They are terminated with new line.
print("Hello World")  //Echo function displays first argument do console.
```

output

```
Hello World
```

4.1.2 Multi Line

Info

- Multi Line comments can span over multiple lines.
- Start of multi-line comment is indicated by combination of slash and multiply character `/*`.
- End of multi-line comment is indicated by combination of multiply and slash character `*/`.
- Benefit of multi-line comments is that they allow you to write a lot of text that spans over multiple lines without having to prefix each line with predefined prefixes as is the case with single line comments.
- Downside of multi-line comments is that they have to be terminated with specific indicator which presents typing overhead compared to single line comments for shorter text.
- SWIFT supports nested multi line comments.

Simple [C]

- This example show to create simple multi line comment.

Test.swift

```
/*
  Multi line comment.
  Block comment.
  Multiple-Line C Syntax.
*/
print("Hello World")  /* Multi line comments can also be used in single line. */
```

output

```
Hello World
```

Nested [C]

- This example show to create nested multi line comments.

Test.swift

```
/*
  First  line of outer comment
  Second line of outer comment
  /*
    First  line of inner comment
    Second line of inner comment
  */
*/
print("Hello World")  /* Multi line comments can also be used in single line. */
```

output

```
Hello World
```

4.2 Operators

Info

- Operators are used to compare or combine variables and their values.

4.2.1 Comparison

Info

- Comparison operators are used to compare values of two variables.
- This way they provide a method to direct program flow depending on the outcome.
- SWIFT comparison operators should be used only for comparing numerical values.
- Strings should be compared using predefined SWIFT functions designed for that purpose.

Operators

TYPE	NAME	DESCRIPTION
`a == b`	Equal	TRUE if a is equal to b.
`a != b`	Not equal	TRUE if a is not equal to b.
`a < b`	Less than	TRUE if a is less than b.
`a > b`	Greater than	TRUE if a is greater than b.
`a <= b`	Less than or equal	TRUE if a is less than or equal to b.
`a >= b`	Greater than or equal	TRUE if a is greater than or equal to b.

Example [C]

- This example shows how to use comparison operators.

Test.swift

```
//TEST VARIABLES.
var left  = 65
var right = 70

//COMPARISON OPERATORS.
if (left == right) { print("Left    equals              right.") }
if (left != right) { print("Left is different from      right.") }

if (left <  right) { print("Left is smaller   then      right.") }
if (left <= right) { print("Left is smaller   or equals right.") }
if (left >  right) { print("Left is greater   then      right.") }
if (left >= right) { print("Left is greater   or equals right.") }
```

output

```
Left is different from      right.
Left is smaller   then      right.
Left is smaller   or equals right.
```

4.2.2 Arithmetic

Info

- Arithmetic operators are used to perform mathematical operations.
- Mathematical operations, not supported by arithmetic operators, can be performed using SWIFT mathematical functions.

Operators

TYPE	NAME
- b	Negation
a + b	Addition
a - b	Subtraction
a * b	Multiplication
a / b	Division
a % b	Modulus
a++	Post-increment
a--	Post-decrement
++b	Pre-increment
--b	Pre-decrement

Example [C]

- This example shows how to use arithmetic operators.

Test.swift

```
//TEST VARIABLES.
var x          = 10
var y          = 20

//ARITHMETIC OPERATORS.
var negate     =    -y    //  -20 = -20
var add        = x + y    //10+20 =  30
var subtract   = x - y    //10-20 = -10
var multiply   = x * y    //10*20 = 200
var divide     = x / y    //10/20 = 0.5
var modulo     = x % y    //10%20 = 0*20+10 = 10
var increment2 =     y++  //Store y into increment2 and then increment y by 1.
var decrement2 =     y--  //Store y into decrement2 and then decrement y by 1.
var increment1 =   ++y    //Increment y by 1 and then store y into increment1.
var decrement1 =   --y    //Decrement y by 1 and then store y into decrement1.

//DISPLAY RESULTS.
print(increment2)         //Replace increment2 with any other variable from above to see its value.
```

output

```
20
```

4.3 Statements

Info

- Statement is smallest part of code which can be executed.
- SWITFT statements are optionally terminated with semicolon `;` indicating end of statement.
- This is required only if there are multiple statements in the same line.
- Statements are classified depending on their function.

4.3.1 Conditional

Info

- Conditional Statements define which part of code should be executed depending on some condition.
- Code is executed only once compared to Looping Statements where code can be execute multiple times.

4.3.1.1 Branching

Info

- Branching is used to control program flow.

4.3.1.1.1 if

Info

- This tutorial shows how to work with if statement.
- If statement
  - executes body if condition is true
  - can be written without round brackets enclosing the condition
  - can't be written without curly brackets enclosing the body

Syntax

```
if (condition) { body }
```

Example [C]

- In this example specific code is executed depending on the value of variable `number`.

Test.swift

```
//TEST VARIABLES.
var number = 10

//IF STATEMENTS.
if (number == 10) { print("Number is 10") }   //Curly   brackets are requiered.
if  number == 10  { print("Number is 10") }   //Rounded brackets are optional.
```

output

```
Number is 10
Number is 10
```

4.3.1.1.2 if case

Info

- This tutorial shows how to work with `if case` statement.
- `If case` statement
  - executes body if condition is true
  - must be written without round brackets around condition
  - must have curly brackets around body
  - allows you to use matching patterns (as shown in Int)
- Following examples show how to use `if case` statement with different data types
  - Int
  - Enumerator

Syntax

```
if case condition { body }
```

Enumerator [C]

- This example shows how to use `if case` statement with `enum` data type.

Test.swift

```
//DECLARE ENUMERATOR.
enum TestEnum {
  case One
  case Two
  case Three
}

//CREATE ENUMERATOR INSTANCE.
let state = TestEnum.One

//IF CASE STATEMENT.
if case TestEnum.One = state { print("1") }
```

output

```
1
```

Int [C]

- This example shows how to use `if case` statement with `Int` data type.

Test.swift

```
//DECLARE VARIABLE.
var cnt = 1

//IF CASE STATEMENTS.
if case 1      = cnt { print("1"      ) }   //Specific value.
if case 2...10 = cnt { print("[2, 10]") }   //Range pattern.
if case 2..<10 = cnt { print("[2, 10]") }   //Range pattern. Same as previous.
```

output

```
1
```

4.3.1.1.3 else if

Info

- Multiple `else if` statements can be used in combination with `if` statement.
- `if...else if` combo works as multiple `if` statements.
- Conditions are evaluated in sequence and only code belonging to the first condition which evaluates to true is executed.
- `if...else if` combo can optionally end with `else` statement.
  If so, then if all conditions evaluate to FALSE only code belonging to `else` statement will be executed.

Syntax

```
if      (condition1) { code1  }
else if (condition2) { code2  }
...
else                 { code10 }
```

Example [C]

- In this example specific code is executed depending on the value of variables `number` and `text`.

Test.swift

```
//TEST VARIABLES.
var number = 10
var text   = "Hello"

//ELSE IF STATEMENTS.
if      (number ==  10    ) { print("Number is 10"     ) }
else if (text   == "Hello") { print("Text equals Hello") }
else if (text   == "World") { print("Text equals World") }
else                        { print("No match found"   ) }
```

output

```
Number is 10
```

4.3.1.2 Looping

Info

- Depending on given condition Looping Statements can execute specific part of code multiple times.
- Existence of condition makes them similar to Conditional Statements but they can execute code only once.

4.3.1.2.1 for

Info

- Depending on the condition, `for` statement can execute specified code multiple times where
  - expression1 is executed at the beginning of the first iteration
  - expression2 is executed at the end of each iteration
  - condition is evaluated at the beginning of each iteration
    - If it is TRUE code is executed
    - If it is FALSE code is NOT executed and program breaks out of for loop
- Expressions and condition can be left empty in which case condition evaluates to TRUE.
- You can use
  - `break` statement to break out of for loop
  - `continue` statement to skip the rest of the body and continue with next iteration from the beginning of for loop.

Syntax

```
for (expression1; condition; expression2) { body }
```

Example [C]

- In this example we show different ways of using `for` statement together with using `break` and `continue` statements.

Test.swift

```
//SIMPLE.   Output is: 1234.
for (var i=1; i<=4; i++) {
  print(i)
}

//BREAK.     Output is: 12  (Since we break out of for loop when i=3)
for (var i=1; i<=4; i++) {
  if(i==3) { break }
  print(i)
}

//CONTINUE. Output is: 124  (Since display of number 3 is skipped)
for (var i=1; i<=4; i++) {
  if(i==3) { continue }
  print(i)
}

//COMPLEX.  Output is: 123.
for (var i=1, j=5 ; i<=4 && j>2 ; i++, j--) {
  print(i)
}
```

output

```
1234
12
124
123
```

4.3.1.2.2 for...in

Info

- This tutorial shows how to use for...in looping statement to iterate through array elements.
- You can use
  - `break` statement to break out of for loop
  - `continue` statement to skip the rest of the body and continue with next iteration from the beginning of for loop.

Syntax

```
for element in array { body }
```

Example [C]

- In this example we show different ways of using `for...in` statement together with using `break` and `continue` statements.

Test.swift

```
//DECLARE ARRAY.
var people = ["John", "Lucy", "Bob"]

//ITERATE THROUGH ARRAY ELEMENTS.
for person in people {
  print(person)
}

//BREAK. Output is: "John" "Lucy"
for person in people {
  print(person)
  if(person == "Lucy") { break }
}

//CONTINUE. Output is: "John" "Bob"
for person in people {
  if(person == "Lucy") { continue }
  print(person)
}
```

output

```
John
Lucy
Bob

John
Lucy

John
Bob
```

4.3.2 Jumping

Info

- Jumping statements are used to unconditionally continue code execution at some other part of code.
- Use of such statements is considered bad practice since they make it harder to follow code execution.
- You should use conditional statements instead.

4.3.2.1 break

Info

- `break` statement is used to end execution of looping statements or switch blocks.
- This means that you can use it to stop execution of for, while, do...while or switch blocks.

Syntax

```
break
```

Example [C]

- This example shows different ways of using `break` statement.

Test.swift

```
//SWITCH.
switch (i) {
  case 1:
    print("Value is 1. Exit from switch. \n")                    //Only selected case is executed.
    if (i==1) { break  }                                         //Break from the rest of the case
    print("This line is skipped. \n")
  default:
    print("Default value is selected if no case was true.\n")
}

//FOR.      Output is: 12  (Since we break out of for loop when i=3)
for(var i=1; i<=4; i++){
  if(i==3) { break }
  print(i)
}

//WHILE.    Output:12      (Since we break out of for loop when i=3)
var i = 0
while(i<=4){
  i++
  if(i==3) { break }
  print(i)
}

//DO WHILE. Output:12      (Since we break out of for loop when i=3)
i = 0
do{
  i++
  if(i==3) { break }
  print(i)
} while(i<=4)
```

output

```
12
12
12
```

```
)                                       //Hello from test.
```

5 Creating custom data types

Info

- Following tutorials show how to create **custom** data types using Classes, Structures, Enumerators and Protocols.
- Note that class, struct, enum and protocol are NOT data types, they are data type types (types of data types).
  So if `MyClass` data type is created using `class` it is of class data type type.

Basic categorization [R]

- In SWIFT all Data Types can be categorizes as
  - Named types
    - are types that can be given a particular name when defined
    - are either predefined in the Swift (Int, String, Array) or can be created by user (MyClass, MyProtocol)
    - are declared using Classes, Structures, Enumerators, and Protocols
  - Compound types
    - are types that can't be given a particular name when defined
    - are either predefined in the Swift or can be created by user ((Int, Int) -> String)
    - may contain named types and other compound types ((Int, (Int, Int))
    - are declared using Tuples and Functions

Term Class is not a data type

- Term Int is a data type.
- Term class is not a data type.
- Instead term class is used to create custom data types (like Int).
- When you declare new class you are actually declaring new data type.
- The same logic equally applies to Structures, Enumerators and Protocols.

Following line declares new data type. Name of this custom data type is MyDataType.

```
class MyDataType {}
```

Type of data type

- Data can be categorized into different types called data types (types of data).
  Examples are Int, Array, MyClas, MyProtocol, etc.
- Data types can also be categorized into different types called data type types (types of data types).
  Type of data type can be either Class, Structure, Enumerator, Protocol, Tuple or Function.
- Int is the name of specific data type.
  Int data type belongs to Structure data type type since it is created using Structure.
- If certain data type was created using Structure then this created data type is of Structure data type type.
  This is why we say that Int, UInt and Float (which are declared using Structure) belong to Structure data type type.
  So Int, UInt and Float are data types created using Structure.
  Structure is not data type, instead it is data type type (the same as Class, Enumerator, Protocol, Tuple or Function).

5.1 Classes

Info

- Class data types are custom made data types that can contain methods and multiple items of different data types.
- **SWIFT has no support for Object literals** which means that you can not use literals to create data of class data types.
- Class is collection of fields and methods used to encapsulate certain functionality for the purpose of
  - Defining specific class/type of objects like person, employee, car, animal, food, etc.
    Such classes are then instantiated to create specific object of that type like person John, animal Rex or food Apple.
  - Organizing functions into separate groups where each class contains one such set.
    Such function might then be accessed directly using class name without the need of instantiating objects.
- An instance of a class is traditionally known as an object.

Instantiation

- When class or structure instance is created all variable and constant fields must have appropriate initial values either by
  - assigning a default value as part of variable and constant field declaration
  - setting an initial value by initializer init()

5.1.1 Declare data type

Info

- This tutorial shows how to declare class data type.
- Class data type is declared
  - using keyword `class`
  - followed by required class's name
  - followed by optional parent class
  - followed by optional list of protocols
  - followed by required body inside curly brackets containing properties and methods

Syntax

```
class className : [ParentClass, Protocols] {
  Properties
  Methods
}
```

Example [C]

- This example shows how to declare simple class `MyClass` with single property `name` and single instance method `sayHello()`.
- Then class instance `myObject` is created and used it to call instance method.
- More complex examples of class declarations are covered in subsequent tutorials each focusing on specific item that can be declared as part of class declaration like initializers, deinitializers, parent class, protocols, etc.

Test.swift

```
//DECLARE CLASS.
class Person {

  //DECLARE FIELD
  var name = "unknown"

  //DECLARE METHOD
  func sayHello() {
    print("Hello world")
  }

}

//CREATE OBJECT.
var john = Person()
    john.sayHello()
```

5.1.2 Designated Initializer

Info

- This tutorial shows how to use `init()` instance method also known as Designated Initializer.
- Designated Initializer has following properties
  - it can only be defined inside **classes or structures**
  - it is used to initialize instances of classes and structures
  - it must initialize all stored properties
  - class or structure can have multiple designated initializers with different parameters
  - it is declared without using keyword `func`
  - it has no return value
  - **all parameters by default have external names equal to their local names**
  - external names can be omitted by using `_` as with normal functions
  - it can only reference **parent's Designated Initializers** (it can't reference other initializers of the same class)
- When class or structure instance is created their
  - optionals are automatically initialized to `nil`
  - variables and constants must have appropriate initial values either by
    - assigning a default value as part of variable and constant field declaration
    - setting an initial value by initializer `init()`
- This tutorial contains following examples on how to work with initializers
  - Declare single initializer
  - Declare multiple initializers
  - Automatically call parent's initializer
  - Explicitly call parent's initializer

Syntax

```
init (parameters) { body }
```

Declare single initializer [C]

- In this example we declare single initializer which accepts two arguments.

Test.swift

```
//DECLARE CLASS.
class Person {

  //DECLARE FIELDS.
  var name:String
  var age :Int

  //DECLARE INITIALIZER.
  init(name:String, age:Int) {           //All properties must be initialized.
    self.name = name
    self.age  = age
  }

}

//CREATE OBJECT USING INITIALIZER.
var john = Person(name:"John", age:50)   //Reference initializer.
print(john.name)
print(john.age)
```

output

```
John
50
```

Declare multiple initializers [C]

- In this example we declare two initializers with different arguments.

Test.swift

```
//DECLARE CLASS.
class Person {

  //DECLARE FIELD.
  var name:String

  //DECLARE INITIALIZER WITH NO PARAMETERS.
  init() {
    name = "John"                          //All properties must be initialized.
    print("Created object \(name)")
  }

  //DECLARE INITIALIZER WITH ONE PARAMETER.
  init(name:String) {
    self.name = name
    print("Created object \(name)")
  }

}

//CREATE OBJECTS.
var john = Person()                        //Call initializer with no  paramters.
var bob  = Person(name:"Bob")              //Call initializer with one paramter.
```

output

```
Created object John
Created object Bob
```

Automatically call parent's initializer [C]

- Parent's initializer is automatically called if child has no initializer.
- We declare `Employee` (with no initializer) which inherits from `Person` (with declared initializer).
- When we create object from class `Employee` parent's constructor is automatically called.

Test.php

```
//DECLARE PARENT CLASS.-------------------------------------------------------------------------------------
class Person {

  //DECLARE FIELD.
  var name:String

  //DECLARE PARENT'S INITIALIZER.
  init(name:String) {
    self.name = name
    print("Created object \(name)")
  }

}

//DECLARE CHILD CLASS.--------------------------------------------------------------------------------------
class Employee:Person { }

//CREATE OBJECTS.-------------------------------------------------------------------------------------------
var bob  = Employee(name:"John")
```

output

```
Created object John
```

Explicitly call parent's initializer [C]

- This example shows how to explicitly call parent initializer when child has its own initializer.
- In this example we create `Employee` which inherits from `Person`.
- Then inside `Employee's` initializer we explicitly call `Person's` initializer using `super.init(name:name)`.
- Deleting `Employee's` initializer will automatically call `Person's` initializer when instantiating `Employee`.

Syntax

```
super.init(parameters)
```

Test.php

```
//DECLARE PARENT CLASS.-------------------------------------------------------------------------------------
class Person {

  //DECLARE FIELD.
  var name:String

  //DECLARE PARENT'S INITIALIZER.
  init(name:String) {
    self.name = name
    print("Created object \(name)")
  }

}

//DECLARE CHILD CLASS.--------------------------------------------------------------------------------------
class Employee:Person {

  //DECLARE INITIALIZER.
  override init(name:String) {
    super.init(name:name)
  }

}

//CREATE OBJECTS.-------------------------------------------------------------------------------------------
var bob  = Employee(name:"John")
```

output

```
Created object John
```

5.1.3 Convenience Initializer

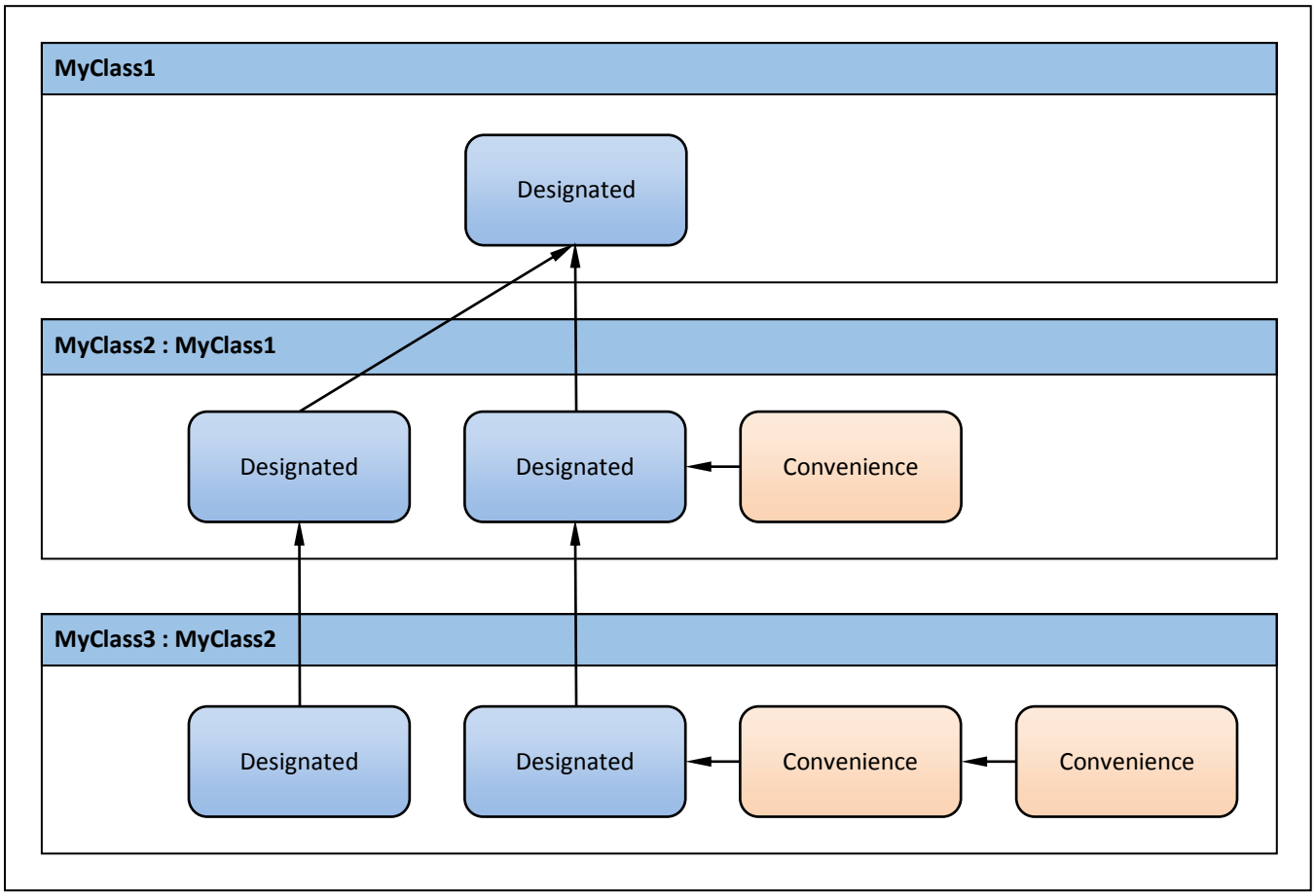
Info

- This tutorial shows how to use `convenience init()` instance method also known as Convenience Initializer.
- Convenience Initializer has the same properties as Designated Initializer with following exceptions
  - it is declared using keyword `convenience`
  - must reference Convenience or Designated Initializer of the same class
  - chain of references to Convenience Initializers must end referencing Designated Initializer of the same class

Syntax

```
convenience init (parameters) { body }
```

Initializers chaining



Declare multiple initializers [C]

- In this example we declare convenience initializer which calls designated initializer.

Test.swift

```
//DECLARE CLASS.
class Person {

  //DECLARE FIELD.
  var name:String
  var age :Int

  //DECLARE INITIALIZER. ACCEPTS NO PARAMETERS.
  init() {
    self.name = "John"
    self.age  = 50
    print("Created object \(name)")
  }

  //DECLARE INITIALIZER. ACCEPTS ONE PARAMETER.
  convenience init(age:Int) {
    self.init()
    self.age = 50
  }

}

//CREATE OBJECTS.
var bob  = Person(age:50)       //Call Convenience Initializer.
```

5.1.4 Deinitializer

Info

- This tutorial shows how to use `deinit` method which is called immediately before a class instance is deallocated.
- Deinitializer `deinit` has following properties
  - it has no return value
  - it is declared without using keyword `func`
  - it can only be declared inside **classes**
  - class can have multiple initializers with different parameters
  - all parameters by default have external names equal to their local names
  - external names can be omitted by using `_` as with normal functions

Syntax

```
deinit { body }
```

Example [C]

- In this example we create object inside function scope.
- When function returns object is destroyed which automatically called deinit.

Test.swift

```
//DECLARE CLASS.
class Person {

  //DECLARE FIELD.
  var name = "John"

  //DECLARE DEINITIALIZER.
  deinit {
    print("\(name) was destroyed")
  }

}

//CREATE FUNCTION SCOPE.
func test() {
  let john = Person()      //Function scope.
  print(john.name)
}

//REFERENCE FUNCTION.
test()
```

Automatically call Parent Denitializer [C]

- If child has deinitializer, parent's deinitializer is called afterwards.
- We declare `Employee` (with no initializer) which inherits from `Person` (with declared initializer).
- When we create object from class `Employee` parent's constructor is automatically called.

Test.php

```
//DECLARE PARENT CLASS.
class Person {

  //DECLARE FIELD.
  var name = "John"

  //DECLARE DEINITIALIZER.
  deinit {
    print("Hello from parent deinitializer")
  }

}

//DECLARE CLASS.
class Employee:Person {

  //DECLARE DEINITIALIZER.
  deinit {
    print("Hello from child deinitializer")
  }

}

//CREATE FUNCTION SCOPE.
func test() {
  let john = Employee()      //Function scope.
  print(john.name)
}

//REFERENCE FUNCTION.
test()
```

output

```
John
Hello from child deinitializer
Hello from parent deinitializer
```

5.1.5 Inherit Class

Info

- This tutorial shows how to declare class which inherits behavior from another class by writing
  - `:` after the class name
  - followed by the name of parent class from which class should inherit fields and methods
- Class can extend only from a single class (which is then called parent) since multiple inheritance is not supported.
- Swift classes do not inherit from a universal base class.
  Classes you declare which don't inherit from another class automatically become base classes for you to build upon.

Syntax

```
class Child : Parent { body }
```

Example [C]

- In this example we declare parent class `Person`.
- Then we declare child class `Soldier` which inherits from `Person` giving Soldier access to `Person's` fields and methods.

Test.swift

```
//DECLARE PARENT CLASS.
class Person {
  var  name = "John"
  func displayName() { print(name) }
}

//DECLARE CHILD CLASS.
class Soldier : Person {
  var  weapon = "Rifle"
  func attack() { print(name) }
}

//ACCESS PARENT FIELDS AND METHODS.
var john = Soldier()                //Create Object from Class Soldier.
    john.displayName()              //Reference inherited method.
print(john.name)                    //Reference inherited field.
```

5.1.6 Conform to protocol

Info

5.2 Functions

Info

- Function is reusable block of code that can be repeatedly called with different parameters and which can return a value.

5.2.1 Declare

Info

- This tutorial shows how to
  - **declare function** (function data) **(We are not declaring function data type)**
  - assign name to it
- Function declaration starts with keyword `func` and declares function's
  - **name** which is used to call the function to perform its task
  - **input parameters** on which function performs its task by declaring their
    - order
    - names
    - data types
    - default values where parameters which have them must follow after parameters without them
  - **data type of return value** which is used to return the result of the function which can be any data type including
    - tuple which is usually used for returning multiple values
    - any other data type either scalar or compound

Syntax for Function Declaration

```
func name (param1, param2, ...) -> ReturnDataType { body }
```

Example [C]

- This example shows how to declare and call a function
- Function declaration defines
  - function name as `sayHello`
  - followed by list of parameters inside round brackets `(name:String, age:Int)` specifying their local names & data types
  - followed by data type of return value as `-> String`

Test.swift

```
//DECLARE FUNCTION.
func sayHello (name:String, age:Int=70) -> String {
  return "\(name) is \(age) years old."
}

//CALL FUNCTION.
var result = sayHello ("John", age:50)
print(result)
```

output

```
John is 50 years old.
```

What is function declaration?

- When you declare new function you are actually declaring function data and NOT declaring new function data type.
  And then you give the name to created function data by specifying function name.
- This is in contrast to when declaring new class in which case you ARE declaring new class data type.
  And then you give the name to newly created class data type.
- **Function data types can't be named and therefore can't be declared for later use (like class data types can).**
  So **we are declaring function data and not function data type** (but function data type is inferred from the declaration).
- During function declaration you are ONLY declaring that your function is of specific function data type.
  That function data type is inferred from data types of input parameters and return value.
  Parameter names and default values are ignored when inferring function data type.
- This is EXACTLY the same as when declaring a variable like `var name:Int = 10` or `var name:(Int, String)->Int`.
  Variable declaration does NOT create new `Int` data type or new `(Int, String)->Int` data type.
  Instead variable declaration only declares that variable can store data of specific data type `Int` or `(Int, String)->Int`.

5.2.2 Reference

Info

- This tutorial shows how to reference/call a function by
  - typing its name
  - followed by list of parameter values separated by comma `,` inside curved brackets

Syntax for Function Call

```
name (value1, value2, ...)
```

Example [C]

- This example shows how to call a function
- Function is called by
  - giving its name `sayHello`
  - followed by list of parameter values inside round brackets `("John", age:50)`

Test.swift

```
//DECLARE FUNCTION.
func sayHello (name:String, age:Int=70) -> String {
  return "\(name) is \(age) years old."
}

//CALL FUNCTION.
var result = sayHello ("John", age:50)
print(result)
```

output

```
John is 50 years old.
```

5.2.3 Name

Info

- Declaration of function name should follow keyword `func`.
- Function name is used to uniquely reference (call) function.
- You can't have two functions with the same name even if they have different number or type of parameters.
  Doing so will result in following error `Fatal error: Cannot redeclare test()`

Example [C]

- Example shows simple function named `test` which displays message when called and has no parameters or return value.

Test.swift

```
//DECLARE FUNCTION.
func test() {
  print("Hello from test.")
}

//CALL FUNCTION.
test()
```

output

```
Hello from test.
```

5.2.4 Scope

Info

- Following tutorials show how to create function with either
  - Global Scope making it visible from any where
  - Function Scope making it visible only inside the function in which it was declared

5.2.4.1 Global

Info

- This tutorial shows how to declare function with global scope (global function).
- Global function
  - is declared outside other functions and types (outside classes, structures, enumerators, etc.)

Example [C]

- Example declares global function by declaring it outside any other structures (global scope).

Test.swift

```
//DECLARE GLOBAL FUNCTION.
func sayHello() {
  print("Hello world")
}

//CALL FUNCTION.
sayHello()
```

output

```
Hello world
```

5.2.4.2 Function

Info

- This tutorial shows how to declare function with function scope (nested function).
- Nested function
  - is declared inside another function
  - can be referenced only inside the function in which it was declared

Example [C]

- Example declares global function by declaring it outside any other structures (global scope).

Test.swift

```
//DECLARE OUTER FUNCTION.
func outerFunction() {

  //DECLARE NESTED FUNCTION.
  func nestedFunction() {                    //Must be declared before called.
    print("Hello from nestedFunction()")
  }

  //CALL NESTED FUNCTION.
  nestedFunction()

}

//CALL OUTER FUNCTION.
outerFunction()
```

output

```
Hello world
```

5.2.5 Return

Info

- Function can optionally return a value in which case return data type must be specified during function declaration.
- If you want to return more than one value you can use tuple or some other compound data type.
- Return value is declared by using `return` keyword which can be used multiple times on different places inside function.
- Alternatively you could use In-Out paramters to return values from functions or to modify them inside the function.

5.2.5.1 Number of values

Info

- Following tutorials show how to return none or more values from a function.

5.2.5.1.1 None

Info

- This tutorial shows how to declare function which returns no value.
- Function which returns no value
  - is declared without specifying return data type
  - exits when end of function is reached
  - can still use `return` keyword inside itself to exit before reaching the end (for instance to return from different places)

Example [C]

- Example shows how to create and call function that has no return value.
- Function simply displays some text without returning any value.

Test.swift

```
//DECLARE FUNCTION.
func test(status:Int) {

  //EXIT EARLY USING RETURN KEYWORD.
  if(status == 10) {
    print("Returning early.")
    return
  }

  //EXIT AT THE END OF FUNCTION WITHOUT USING RETURN KEYWORD.
  print("Hello from end of function.")

}

//CALL FUNCTION.
test(10)
```

output

```
Hello from test.
```

5.2.5.1.2 Single

Info

- To return single value from a function you should
  - Declare return data type (like `-> String`).
  - Use `return` keyword inside a function to specify which value to return.
    Return statement can be used multiple times on different places inside function.

Single return statement [C]

- This example shows how to return single value from a function.

Test.swift

```
//DECLARE FUNCTION.
func test() -> String {
  return "Hello from test."                    //Return string.
}

//CALL FUNCTION.
var result = test()                            //Store function return value into variable result.
print(result)                                  //Hello from test.
```

Multiple return statements [C]

- This example shows how to use `return` statement multiple times on different places inside function.

Test.swift

```
//DECLARE FUNCTION.
func test(age:Int) -> String {
  if   (age==1) { return "First  return value." } //Return string.
  else          { return "Second return value." } //Return string.
}

//CALL FUNCTION.
var result = test(10)                          //Store function return value into variable result.
print(result)                                  //Hello from test.
```

5.2.5.1.3 Multiple

Info

- This tutorial shows how to return multiple values from a function.
- Function can return multiple values in different ways like
  - returning Tuple Data Type
  - using In-Out Input Parameters

Return Tuple [C]

- Example shows how to
  - declare function that returns tuple with named values as alternative to just declaring `-> (String, Int)`
  - return multiple values inside a tuple
- When function returns a tuple
  - it can be stored inside another tuple
  - its values can be extracted into separate variables

Test.swift

```
//DECLARE FUNCTION.
func test() -> (name: String, age: Int) {    //Return tuple
  return ("John", 50)
}

//CALL FUNCTION. SAVE RESULT INTO TUPLE.
var result = test()               //Store tuple.
print(result)                     //Display tuple.
print(result.0)                   //Display specific tuple value using its index.
print(result.1)                   //Display specific tuple value using its index.
print(result.name)                //Display specific tuple value using its name.
print(result.age)                 //Display specific tuple value value its name.

//CALL FUNCTION. SAVE RESULT INTO SEPARATE VARIABLES.
var (nameRet, ageRet) = test()    //Store tuple values into variables.
print(nameRet)                    //Display variable.
print(ageRet)                     //Display variable.
```

output

```
("John", 50)
John
50

John
50

John
50
```

5.2.5.2 Types

Info

- Following tutorials show how to return different types from function.

5.2.5.2.1 Scalar

Info

- This tutorial shows how to return scalar data type from a function.

Example [C]

- This example shows how to return String data type from a function.

Test.swift

```
//DECLARE FUNCTION.
func test() -> String {
  return "Hello from test."                    //Return string.
}

//CALL FUNCTION.
var result = test()                            //Store function return value into variable result.
print(result)                                  //Hello from test.
```

5.2.5.2.2 Tuple

Info

- This tutorial shows how to return tuple from a function.

Example [C]

- Example shows how to
  - declare function that returns tuple with named values as alternative to just declaring `-> (String, Int)`
  - return multiple values inside a tuple
- When function returns a tuple
  - it can be stored inside another tuple
  - its values can be extracted into separate variables

Test.swift

```
//DECLARE FUNCTION.
func test() -> (name: String, age: Int) {    //Return tuple
  return ("John", 50)
}

//CALL FUNCTION. SAVE RESULT INTO TUPLE.
var result = test()               //Store tuple.
print(result)                     //Display tuple.
print(result.0)                   //Display specific tuple value using its index.
print(result.1)                   //Display specific tuple value using its index.
print(result.name)                //Display specific tuple value using its name.
print(result.age)                 //Display specific tuple value value its name.

//CALL FUNCTION. SAVE RESULT INTO SEPARATE VARIABLES.
var (nameRet, ageRet) = test()    //Store tuple values into variables.
print(nameRet)                    //Display variable.
print(ageRet)                     //Display variable.
```

output

```
("John", 50)
John
50

John
50

John
50
```

5.2.5.2.3 Function

Info

- This tutorial shows how to return function.

Example [C]

- Example declares function which returns function of type `(Int,Int)->Int`.

Test.swift

```
//DECLARE FUNCTION.
func test() -> (Int,Int)->Int {                //Declare type of function that should bre returned.

  //DECLARE NESTED FUNCTION.
  func myfunc(s1: Int, s2: Int) -> Int {
    return s1 + s2
  }

  //RETURN NESTED FUNCTION.
  return myfunc                                //Return function.
}

//CALL FUNCTION.
var returnedFunc = test()

//USE RETURNED FUNCTION.
var result = returnedFunc(2,5)
print(result)
```

output

```
7
```

5.2.6 Parameters

Info

- Function can have zero or more parameters.
- It can also have variable number of parameters meaning that not all parameters have to be given when calling function. Parameters which were not given during function call will be given default values.
- For each parameter you must define
  - internal name
  - data type
- Each parameter has
  - internal name which must be declared during function declaration and it used to reference parameter inside function
  - public name which is used to assign value to parameter when calling function (except for the first parameter)
- For the first parameter
  - you can simply define value while calling the function
  - you have to explicitly declare its public name if you want to name the first parameter while calling the function
- For parameters other then the first one
  - you must specify both public name and value while calling the function
  - by default public name is the same as declared internal name
  - you can explicitly declare its public name if you want it to be different from the internal name
- Naming and referencing parameters is discussed in Named parameters.

5.2.6.1 Default values

Info

- Parameters with default values
  - will use that value if value for parameter is not specified during function call
  - must be defined after required parameters (which are those that don't have default values)
  - are called optional parameters and can be used to declare functions with Variable number of parameters

Example [C]

- Example declares function with two parameters with default values declared at the and of parameter list.
- This means that function can be called with 1, 2 or 3 parameters since you can omit second or third optional parameters.
- In the example we omit third parameter which will then use its default value 38 inside the function.

Test.swift

```
//DECLARE FUNCTION.
func sayHello(name: String, age: Int=50, temp: Int = 38) -> String {
  return "\(name) is \(age) years old. \nHis temperature is \(temp) degrees."
}

//CALL FUNCTION.
var result = sayHello("John", temp:40)  //Second argument is omited.
print(result)
```

output

```
John is 50 years old.
His temperature is 40 degrees.
```

5.2.6.2 Number of parameters

Info

- Function can have zero or more parameters.
- It can also have variable number of parameters meaning that not all parameters have to be given when calling function. Parameters which were not defined during function call will be given default values.

5.2.6.2.1 Zero

Info

- To declare function that has no input parameters simply omit the list of parameters between round brackets.

Example [C]

- Example shows function which expects no input parameters and simply displays some text without returning any value.

Test.swift

```
//DECLARE FUNCTION.
func test() {
  print("Hello from test.")
}

//CALL FUNCTION.
test()
```

output

```
Hello from test.
```

5.2.6.2.2 One

Info

- To declare function that has single input parameter simply declare it between round brackets by specifying its
  - required internal name
  - optional public name
  - data type
- When calling a function with single parameter you can
  - simply specify its value between round brackets if public name was not declared
  - optionally specify its public name if ti was explicitly declared
- Naming and referencing parameters is discussed in greater details in Named parameters.

No public name [C]

- In this example we declare a function that accepts single parameter by declaring only its internal name.
- Function is called by providing the value for the parameter without naming the parameter.

Test.swift

```
//DECLARE FUNCTION.
func sayHello(name:String) -> String {
  return "Hello \(name)."
}

//CALL FUNCTION.
var result = sayHello("John")
print(result)
```

output

```
Hello John
```

Declared public name [C]

- In this example we declare function that accepts single parameter for which both public and internal names are declared.
- Since public name was declared then when calling function we have to provide both parameter's public name and value.
- Inside the function parameter is referenced using its private name.

Test.swift

```
//DECLARE FUNCTION.
func sayHello(namePublic namePrivate:String) -> String {
  return "Hello \(namePrivate)."
}

//CALL FUNCTION.
var result = sayHello(namePublic:"John")
print(result)
```

output

```
Hello John
```

5.2.6.2.3 Multiple

Info

- To declare function that has multiple parameters list all of them between round brackets by specifying
  - their data types
  - their local names
  - their optional external names
  - optional underscore _ to omit their external names (can't be used for first parameter)
  - **them in the same order in which they are declared in function**
- Multiple parameters can also be achieved using Variadic parameter or compound data types.
- Naming and referencing parameters is discussed in greater details in Named parameters.

Example [C]

- In this example we declare a function that accepts exactly two parameters for which only internal names are declared.
- Declared parameter names are actually their internal names which can be used inside the function.
  - Public name of the first parameter is not declared
  - Public name of the second parameter is implicitly defined to be the same as its internal name
- Parameters are used to return string which is constructed using string literal.

Test.swift

```
//DECLARE FUNCTION.
func sayHello(name:String, age:Int) -> String {  //Internal names.
  return "\(name) is \(age) years old."
}

//CALL FUNCTION.
var result = sayHello("John", age:50)  //Any argument after the first must be named with its external name.
print(result)
```

output

```
Hello John
```

5.2.6.2.4 Variable

Info

- This tutorial shows how to declare function that can be called with variable number of input parameters.
- This is achieved by defining so called optional parameters which are parameters
  - with defined default values which must be constant expressions (you cannot assign function calls or variables)
  - which must be defined after required variables which are those that do not have default values
- Naming and referencing parameters is discussed in greater details in Named parameters.

Example [C]

- Example declares function with two parameters with default values declared at the and of parameter list.
- This means that function can be called with 1, 2 or 3 parameters since you can omit second or third parameter.
- In the example we omit third parameter which will then use its default value 38 inside the function.

Test.swift

```
//DECLARE FUNCTION.
func sayHello(name: String, age: Int=50, temp: Int = 38) -> String {
  return "\(name) is \(age) years old. \nHis temperature is \(temp) degrees."
}

//CALL FUNCTION.
var result = sayHello("John", temp:40)  //Second argument is omited.
print(result)
```

output

```
John is 50 years old.
His temperature is 40 degrees.
```

6 Advanced topics

Info

- Following tutorials show how to use advanced topics.

6.1 Properties

Info

- Following tutorials show how to use properties.
- Properties are constants, variables and optionals declared inside a class, structure, enumerator or protocol.

6.1.1 Declare

Info

- This tutorial shows how to declare a property.
- Properties are declared in the same way as variables, constants and optionals but inside a class, structure or enumerator.

Example [C]

- This example shows how to declare two fields inside a class.
- Static field can be referenced using class name while instance fields can be referenced through objects.

Test.swift

```
//DECLARE CLASS.
class Person {

  //DECLARE FIELDS.
  var name = "John"        //Variable field.
  let age  = 100           //Constant field.

}

//REFERENCE FIELDS.
var john = Person()        //Create object from class Person.
print(john.name)           //Reference field.
```

output

```
John
```

6.1.2 Reference

Info

- Following tutorials show how to reference property.

6.1.2.1 Static field from static method

Info

- This tutorial shows how to reference static field from static method of the same class.
- The same rules also apply for structures and enumerators.

Syntax

```
fieldName
self.fieldName
ClassName.fieldName
```

Example [C]

- In this example we create two static methods `displayVar()` and `displayString()`.
- Then we call `displayString()` from `displayVar()` with and without `self` prefix.

Test.swift

```
//CREATE CLASS.
class MyClass {

  //DECLARE STATIC FIELD.
  static var name = "John"

  //DECLARE STATIC METHOD.
  static func displayVar() {
    print(name)                          //Reference static field using its name.
    print(self.name)                     //Reference static field using self.
    print(MyClass.name)                  //Reference static field using class name.
  }

}

//REFERENCE STATIC METHOD.
MyClass.displayVar()                     //Reference static field .
```

output

```
John
John
John
```

6.1.2.2 Static field from instance method

Info

- This tutorial shows how to reference static field from instance method of the same class.
- The same rules also apply for structures and enumerators.

Syntax

```
self.dynamicType.fieldName
ClassName.fieldName
```

Example [C]

- Example shows how to reference type method using type name like `TypeName::methodName()`.

Test.swift

```
//CREATE CLASS.
class MyClass {

  //DECLARE STATIC FIELD.
  static var name = "John"

  //DECLARE STATIC METHOD.
  func displayVar() {
    print(self.dynamicType.name)                //Reference static method using its name.
    print(MyClass.name)                         //Reference static method using self.
  }

}

//REFERENCE STATIC METHOD.
//REFERENCE INSTANCE METHOD.
var john = MyClass()
    john.displayVar()                           //Reference instance method.
```

output

```
Hello from type method
```

6.1.2.3 Static field using class name

Info

- This tutorial shows how to reference static field by using class name.
- This can be used from any part of the code where class and field are visible.
- The same rules also apply for structures and enumerators.

Syntax

```
ClassName.fieldName
```

Example [C]

- Example shows how to reference static method using class name like `ClassName::methodName()`.

Test.swift

```
//CREATE CLASS.
class MyClass {

  //DECLARE STATIC FIELD.
  static var name = "John"

  //DECLARE INSTANCE METHOD.
  func sayHello() {
    print(MyClass.name)                     //Reference static field from instance method.
  }

  //DECLARE STATIC METHOD.
  static func displayVar() {
    print(MyClass.name)                     //Reference static field from static method.
  }

}

//REFERENCE STATIC METHOD.
print(MyClass.name)                         //Reference static field outside the class.

//REFERENCE INSTANCE METHOD.
```

```
var john = MyClass()
    john.sayHello()                          //Reference instance method.
```

output

```
John
John
```

6.1.2.4 Instance field from instance method

Info

- This tutorial shows how to reference instance field from instance method of the same class.
- The same rules also apply for structures and enumerators.

Syntax

```
fieldName
self.fieldName
```

Example [C]

- In this example we reference instance method `displayString()` from instance method `displayVar()` of the same class.

Test.swift

```
//CREATE CLASS.
class MyClass {

  //DECLARE INSTANCE FIELD.
  var name = "John"

  //DECLARE INSTANCE METHOD.
  func showName() {
    print(name)
    print(self.name)
  }

}

//REFERENCE INSTANCE FIELD.
var john = MyClass()
    john.showName()
```

output

```
John
John
```

6.1.2.5 Instance field from object

Info

- This tutorial shows how to reference instance field from an object.
- The same rules also apply for instances of structures and enumerators.

Syntax

```
objectName.methodName()
```

Example [C]

- In this example we create instance `john` of class `Person`.
- Then we reference instance field `name` from created instance `john` using `john.name`.

Test.swift

```
//CREATE CLASS.
class Person {

  //DECLARE STATIC FIELD.
  var name = "John"

}

//REFERENCE INSTANCE FIELD.
var john = Person()
print(john.name)
```

output

```
John
```

6.1.2.6 Field using self

Info

- This tutorial shows how to reference static or instance field using `self` keyword.
- When using `self.fieldName` you can only reference either
  - static field from static method
  - instance field form instance method
- When using `self.dynamicType.fieldName` you can reference static field from instance method.
- The same rules also apply for structures and enumerators.

Syntax

```
self.fieldName
```

Example [C]

- In this example we create two static methods `displayVar()` and `displayString()`.
- Then we call `displayString()` from `displayVar()` with and without `self` prefix.

Test.swift

```
//CREATE CLASS.
class MyClass {

  //DECLARE STATIC FIELD.
  static var age = 50

  //DECLARE INSTANCE FIELD.
  var name = "John"

  //DECLARE INSTANCE METHOD.
  func sayHello() {
    print(self.dynamicType.age)                  //Reference static   field from instance method using self.
    print(self.name)                             //Reference instance field from instance method using self.
  }

  //DECLARE STATIC METHOD.
  static func displayVar() {
    print(self.age)                              //Reference static field from static method using self.
  }

}

//REFERENCE STATIC METHOD.
MyClass.displayVar()                             //Reference static method.

//REFERENCE INSTANCE METHOD.
var john = MyClass()
    john.sayHello()                              //Reference instance method.
```

output

```
50
John
50
```

6.1.2.7 Parent's fields

Info

- This tutorial shows how to reference parent's fields.
- When using `super` keyword you can only reference either
  - parent's static field from child's static method
  - parent's instance field form child's instance method
- You can't reference parent's static field from child's instance method.

Syntax

```
super.methodName()
ParentName.methodName()                //Only for parent's static methods.
```

Example [C]

- Example shows how to reference parent's static method from child's static method.

Test.swift

```
//CREATE PARENT CLASS.
class Person {

  //DECLARE STATIC FIELD.
  static var staticField = 50

  //DECLARE INSTANCE FIELD.
  var instanceField = "John"

}

//CREATE CHILD CLASS.
class Employee:Person {

  //DECLARE STATIC METHOD.
  static func staticSayHello() {
    print(super.staticField)           //Reference static field from static method.
    print(Person.staticField)
  }

  //DECLARE INSTANCE METHOD.
  func instanceSayHello() {
    print(Person.staticField)          //Reference static   field from instance method.
    print(super.instanceField)         //Reference instance field from instance method.
  }

}

//REFERENCE STATIC METHOD.
Employee.staticSayHello()              //Reference instance method.

//REFERENCE INSTANCE METHOD.
var john = Employee()
    john.instanceSayHello()            //Reference instance method.
```

output

```
50
50
50
John
```

6.1.3 Scope Modifiers

Info

- SWIFT supports five filed scope modifiers: public, private and internal.

Method Scope Modifiers

ACCESS	PUBLIC	INTERNAL	PRIVATE
Within file	✓	✓	✓
Within module	✓	✓	✗
From anywhere	✓	✗	✗

Syntax - Declare

```
[public|private|internal] func functionName (parameters) { body }
```

6.1.3.1 Public

Info

- Public field is accessible from anywhere which means
  - within the file
  - within the module
  - outside the module
- Field can be declared as public only if class is also public.

Syntax - Declare

```
public [var|let] fieldName
```

Example [C]

- This example shows how to declare public field.

Test.swift

```
//DECLARE CLASS.
public class Person {

  //DECLARE PUBLIC FIELD.
  public var name = "John"

}

//REFERENCE PUBLIC FIELD.
var john = Person()
print(john.name)
```

output

```
Hello
```

6.1.3.2 Internal

Info

- Field is by default declared as internal which makes it accessible from
  - within the file
  - within the module
- Field can be declared as internal only if class is public or internal.

Syntax - Declare

```
internal [var|let] fieldName
```

Example [C]

- This example shows how to declare internal field.

Test.swift

```
//DECLARE CLASS.
class Person {

  //DECLARE INTERNAL FIELD.
  internal var name = "John"

}

//REFERENCE INTERNAL FIELD.
var john = Person()
print(john.name)
```

output

```
Hello
```

6.1.3.3 Private

Info

- Private field is accessible only from
  - within the file
- Field can be declared as private independently of class scope modifier.

Syntax - Declare

```
private [var|let] fieldName
```

Example [C]

- This example shows how to declare private field.

Test.swift

```
//DECLARE CLASS.
class Person {

  //DECLARE PRIVATE FIELD.
  private var name = "John"

}

//REFERENCE PRIVATE FIELD.
var john = Person()
print(john.name)
```

output

```
Hello
```

6.1.4 Other Modifiers

Info

- Following tutorials show how to use additional field modifiers
  - Static fields can only reference static fields and methods

6.1.4.1 Static

Info

- This tutorial shows how to declare field as static by using `static` keyword.
- Static field are known as type fields and they operate inside type context.
  By type we mean class, structure or enumerator inside which static field is declared.
  For instance static field declared inside a class is therefore known as class field and it operates inside class context.
- Static field can
  - be given any of the three scope modifiers with internal being default
  - be referenced both through object or class name
  - only access static fields and methods since you can't use `$this->` as a value for static field

Syntax - Declare

```
static [var|let] fieldName
```

Syntax - Reference

```
self.methodName                         //From inside  the class
super.methodName                        //From child's methods
ClassName.methodName                    //From outside the class using class name
```

Example [C]

- This example shows how to declare static field and reference it within from inside and outside of the class.

Test.swift

```
//DECLARE CLASS.
class Person {

  //DECLARE STATIC FIELD.
  static var name = "John"

  //DECLARE STATIC METHOD.
  static func sayHello() {
    print(name)                         //Reference static field from static method.
  }

  //DECLARE INSTANCE METHOD.
  func saySomething() {
    print(self.dynamicType.name)        //Reference static field from instance method.
  }

}

//REFERENCE STATIC METHOD.
Person.sayHello()
print(Person.name);                     //Reference static field using class name.

//REFERENCE INSTANCE METHOD.
var john = Person()
    john.saySomething()
```

output

```
John
John
John
```

7 Tools

Info

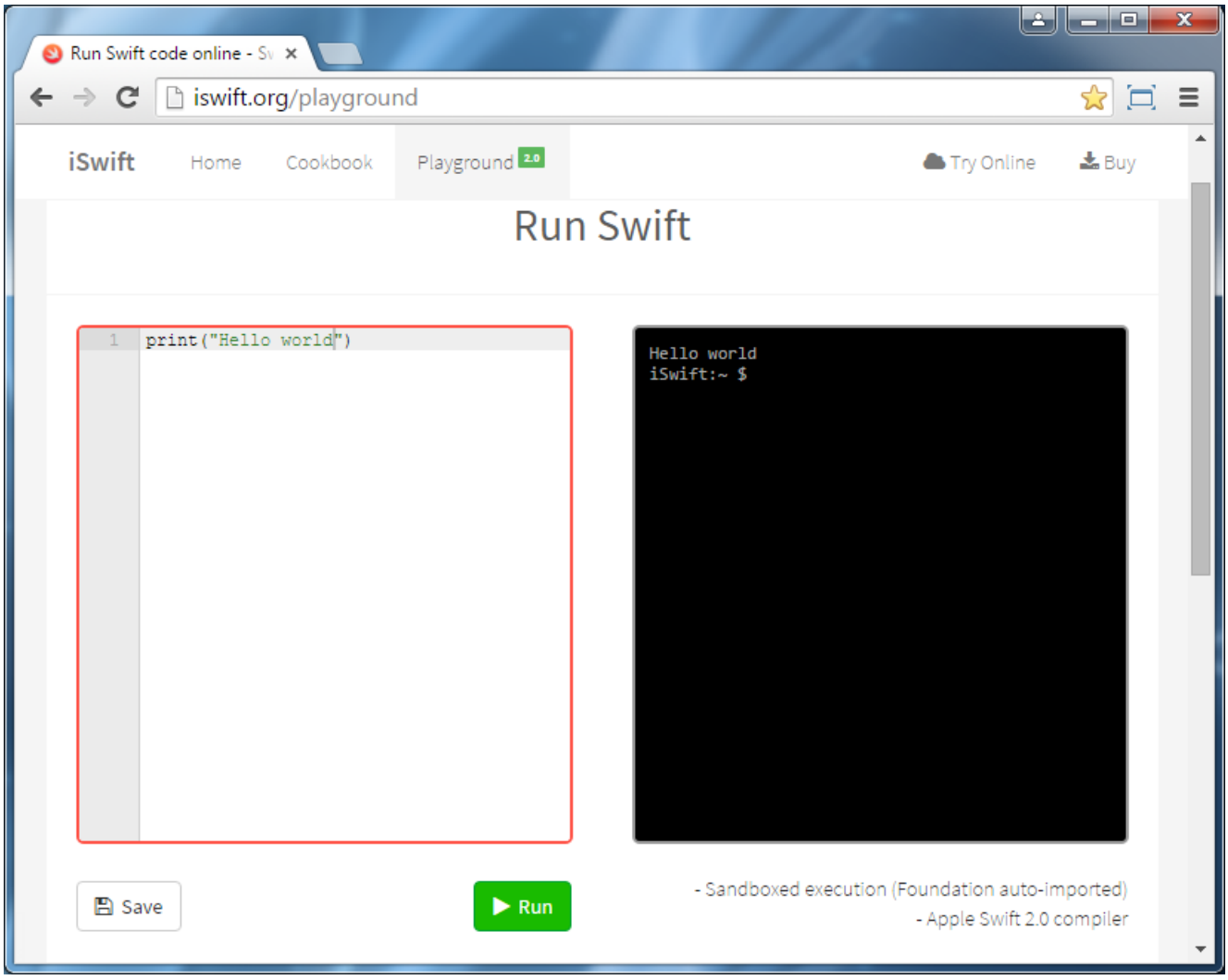
- Following chapters describe different tools that can be used for working with SWIFT.

7.1 Online compiler

Info **[R]**

- This tutorials shows how to create SWIFT application using online compiler.
- This approach is great for learning and testing syntax.
- Online compilers can be used on any operating system since they run in a Web Browser.
- Here is a list of online compilers (be aware of SWIFT version they support)
  - http://iswift.org/playground (Swift Version 2.0)
  - http://www.runswiftlang.com (Swift Version 1.2)
  - http://swiftstub.com (Swift Version 1.2)

http://iswift.org/playground



Procedure

- http://iswift.org/playground
- (Copy content of Test.swift)
- Run

Test.swift

```
print("Hello World")
```

output

```
Hello World
```