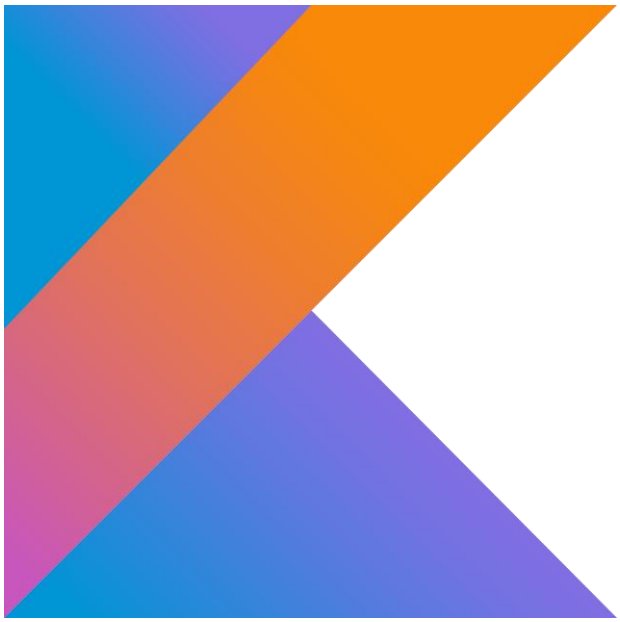


KOTLIN

Quick Start

V 1.0

CONTENT

1 Quick Start

Info

- Kotlin is new modern programming language (similar to SwiftUI).
- It is used for building Android Applications using Android Studio IDE and Jetpack Compose declarative UI language.

1.1 Create Application

Info

- Following tutorials show different ways of creating Kotlin Application.

1.1.1 Online Compiler

Info

- This tutorial shows how to create Kotlin Application using Online Compiler (lacks functionality compared to other options)
- This is a great way to test Kotlin Code snippets without having to install anything on the computer.
- Code that is to be executed needs to be written inside main() function.

Create Application

- https://play.kotlinlang.org
- Run

Code

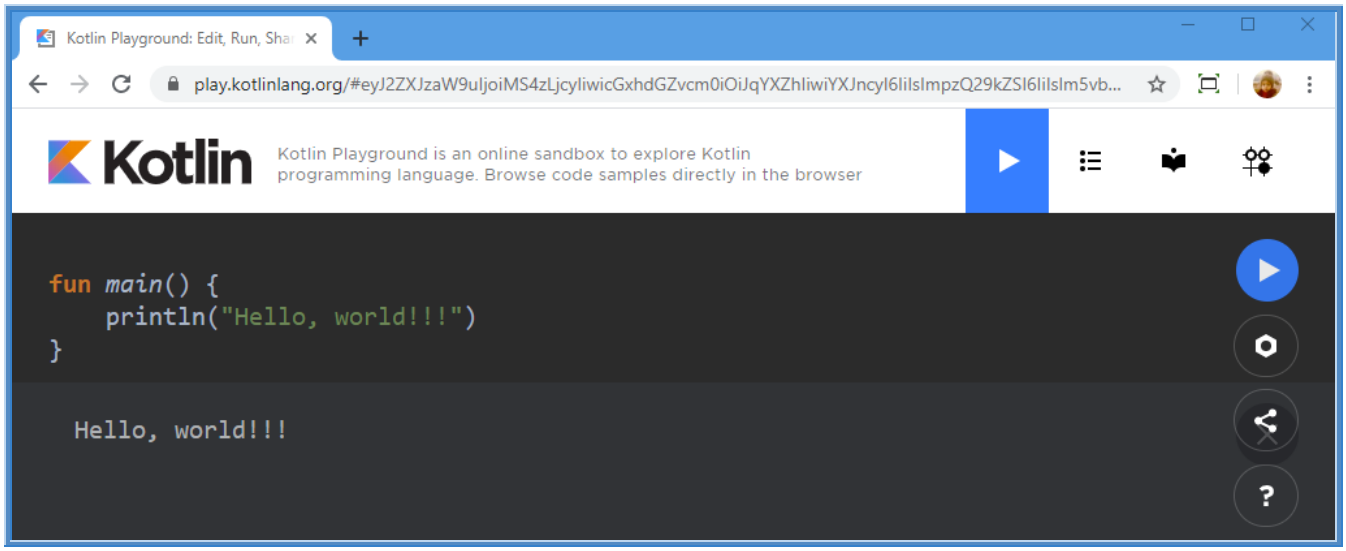
```
//=====================================================================================================================
// CLASS: Person
//=====================================================================================================================
class Person {
  fun sayHello() { println("Hello") }
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {
  var john = Person()
      john.sayHello()
}
```

Output

```
Hello
```

https://play.kotlinlang.org



1.1.2 Android Studio Scratch File

Info [R]

- To test Kotlin Code snippets you can also use Android Studio Scratch File.
- Code that is to be executed needs to be written outside and Function.
- Code is written on the left side and output is displayed on the right side.

Run Code

- Start Android Studio
- File
- New
- Scratch File
- Kotlin

Code

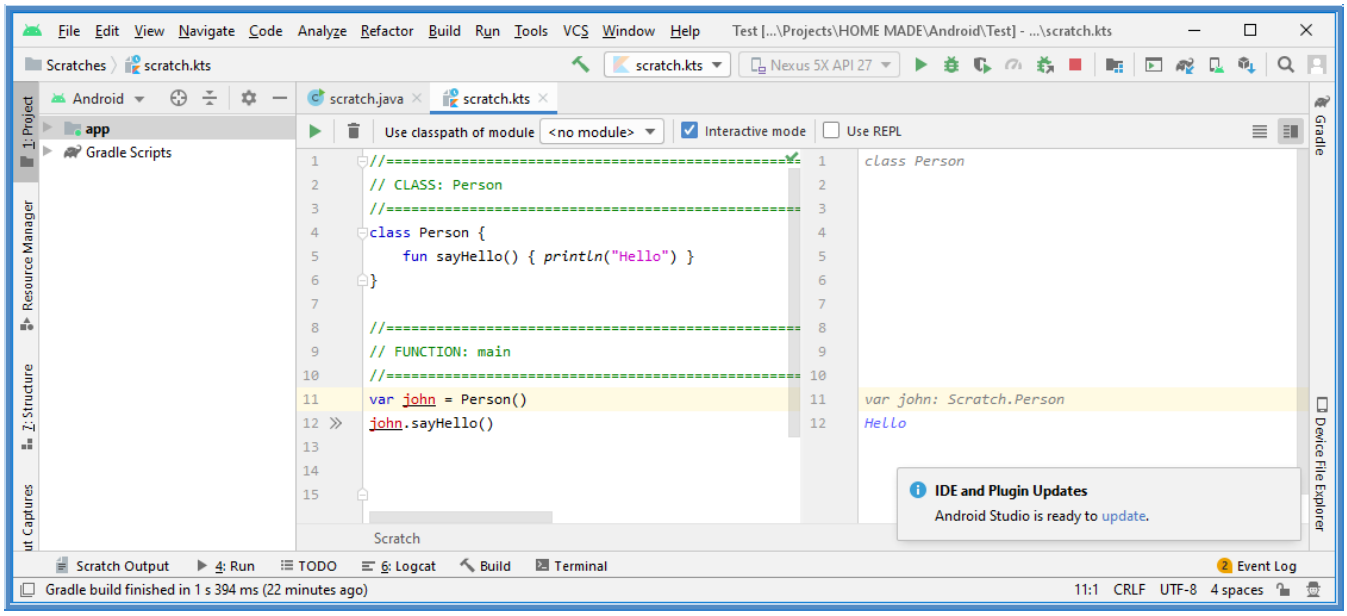
```
//=====================================================================================================
// CLASS: Person
//=====================================================================================================
class Person {
  fun sayHello() { println("Hello") }
}

//=====================================================================================================
// FUNCTION: main
//=====================================================================================================
fun main() {
  var john = Person()
      john.sayHello()
}
```

Output

```
Hello
```

Android Studio Scratch File



2 Main Concepts

Info

- This Chapter give overview of the main concepts of JAVA Syntax.
- In subsequent Chapters these will be covered in more detail.

2.1 Data Types

Info

- Literal are string representation of both Data Type and data value: 10, "John", ["AUDI", "FIAT"].
- Scalar Data Types are implemented as Structures and can only store single value.
- Collection Data Types are implemented as Structures and allow us to group multiple items of the same Data Type.

2.1.1 Literals

Info

- **Literal** is **string representation** of both **Data Type** and **Data Value**: 10, "John", 'A'
- **Literal notations** are different ways of constructing literals that represent the same Data Type: 10, +10, 0B1010 or 0xA.
- Escape Sequences are used for Character & String Literals to symbolize certain characters that can't be written.

Literal Types

LITERAL	EXAMPLE	NOTATIONS
Null	`null`	
Boolean	`true, false`	
Character	`'A'`	
Integer	`65`	Decimal: `65` Hexadecimal: `0x41` Binary: `0b100_0001`
Long	`65L`	Decimal: `65L` Hexadecimal: `0x41l` Binary: `0b100_0001L`
Float	`65.2F`	Basic: `65.2F` Scientific: `0.652E2f`
Double	`65.2`	Basic: `65.2` Scientific: `0.652E2`
String	`"Hello ${name} \n"`	

Null

- Null Literal represents Null Data Type which can have only one value null.

Null

```
var age : Int? = null
```

Boolean

- Boolean Literal represents Boolean Data Type which can have only two values true or false.

Boolean Literals

```
var access2 : Boolean = true
var access2 : Boolean = false
```

Integer

- Integer Literal represents signed Integer Data Type and value and supports multiple notations.

Integer Literal Notations

```
var value : Int = 65                    //Decimal     notation (base 10).
var value : Int = 0x41                  //Hexadecimal notation (base 16).
var value : Int = 0b1000001             //Binary      notation (base 2 ).
var value : Int = 10_000                //You can use underscore any way you like: 10_000, 0b10__00_001
```

Long

- Long Literal represents Long Data Type and value and supports multiple notations.
- It is written by adding letter **"L"** as suffix (in upper or lower case).

Long Literal Notations

```
var value : Long = 65L              //Decimal     notation (base 10).
var value : Long = 0x45L            //Hexadecimal notation (base 16).
var value : Long = 0b10000011       //Binary      notation (base 2 ).
var value : Long = 10_000l          //You can use underscore any way you like: 10_000L, 0b10__00_001L
```

Float [R]

- Float Literal represents Float Data Type and value and supports multiple notations.
- Float literal is used to represent real number.
- It is written by adding letter **"F"** as suffix (in upper or lower case).

Float Literal Notations

```
var value : Float = 65.23f                    //Basic      notation.
var value : Float = 0.6523E2F                 //Scientific notation.
```

Double [R]

- Double Literal represents Double Data Type and value and supports multiple notations.

Double Literal Notations

```
var value : Double = 65.23                   //Basic      notation.
var value : Double = 0.6523E2                //Scientific notation.
```

Character

- Character literal
  - represents character data type and value where value is 16-bit Unicode character
  - is constructed by enclosing character or escape sequence inside single quotes
- Escape sequence
  - can represent any character, including special ones, as shown by the table below
  - starts with backslash / escape character followed by specific combinations representing a character
- **Using octal or Unicode character representations is equivalent to typing that character inside your source code. This differs from escape sequences \n, \' or \" which will not throw error while '\u000D', '\u0027' or '\u0022' might.**

Character

```
//BASIC CHARACTER LITERAL.
var value : Char = 'A'          //'A' character.
var value : Char = '\t'         //Tab character.
var value : Char = '\"'         //Double quote character.
var value : Char = '\''         //Single quote character.

//USING OCTAL ESCAPE SEQUENCE.
var value : Char = '\101'       //'A' character.
var value : Char = '\10'        //Tab quote character.
var value : Char = '\42'        //Double quote character.
var value : Char = '\47'        //Single quote character.

//USING UNICODE ESCAPE SEQUENCE. USES HEXADECIMAL ASCII VALUES WITH LEADING ZEROS.
var value : Char = '\u0041'     //'A' character.
var value : Char = '\u0009'     //Tab character
var value : Char = '\u0022'     //Double quote character
var value : Char = '\u0027'     //Single quote character reports error.
```

String

- String literal represents string data type and value.
- String literal is used to represent strings.
- String literal is written by enclosing sequence of characters inside double quotes.
- **Using octal or Unicode character representations is equivalent to typing that character inside your source code. This differs from escape sequences \n, \' or \" which will not throw error while '\u000D', '\u0027' or '\u0022' might.**

String

```
//BASIC STRING LITERAL.
var name : String = "Display letter A";                  //"Display letter A" is string literal.
var name : String = "First line \nSecond \t line.";      //Escape sequences \n and \t for new line and tab.

//USING OCTAL ESCAPE SEQUENCE.
var name : String = "Display letter \101";               //"Display letter A".
var name : String = "First line \15Second \11 line.";    //"First line \nSecond \t line.".

//USING OCTAL ESCAPE SEQUENCE.
var name : String = "Display letter \u0041";             //"Display letter A".
var name : String = "First line \nSecond \u0009 line."; //Replacing \n with unicode causes error.
```

Escape Sequences

- Escape Sequences are used for Character & String Literals to symbolize certain characters that can't be written.

Escape Sequences

SEQUENCE	DESCRIPTION
\n	Linefeed
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\e	Escape
\f	Form feed
\\	Backslash
\$	Dollar sign
\"	Double quote (only for Double Quotes and Heredoc)
\'	Single quote (only for Single Quotes and Nowdoc)
\d	Octal representation of character as defined in ASCII Table (For 'A' it is '\101').
\ud	Unicode representation of character as defined in ASCII Table (For 'A' it is '\u0041').

2.1.2 Scalars

Info [R]

- **Scalar Data Types** can only store single value. In Kotlin every Data Type is an Object.
  Unlike Java which has primitive Data Types (int, double) and related wrapper Objects (Integer, Double).

Scalar Data Types

TYPE	EXAMPLE	DESCRIPTION	SIZE
Null	`null`	undefined/unknown value	
Bool	`true, false`	logical TRUE or FALSE	
Char		Unicode character	16-bit
Byte	20, -54	Signed integer number	8-bit two's complement
Short	20, -54	Signed integer number	16-bit two's complement
Int	20, -54	Signed integer number	32-bit two's complement
Long	20, -54	Signed integer number	64-bit two's complement
Float	-45.87	Real number	32-bit IEEE 754
Double	-45.87	Real number	64-bit IEEE 754

Null

- Null Data Type can only have **null** value and can only be assigned to **Optional** either by
  - explicitly setting Optional to null using Null Literal
  - declaring Optional without assigning value to it in which case it is initialized to null

Null Data Type

```
var age    : Int?        //Optional variable is initialized to null data type.
var weight : Int? = null //Optional variable is set         to null data type using Null Literal.
```

Boolean

- Bool Data Type can only have two possible values logical **true** or **false** and can be created using Boolean Literal.

Bool Data Type

```
var access2 : Boolean = false              //Explicit Data Type declaration.
var access1           = true               //Implicit Data Type declaration.
```

Char

- Float data type is type of data that represents real number and can be created using Float Literal.
- Check out float cheat sheet containing code samples for working with float data type.

Char

```
var value : Char = 'A'
```

Byte

- Byte data type is type of data that represents integer number and can be created using Integer Literal.
- Check out byte cheat sheet containing code samples for working with byte data type.

byte

```
var value: Byte = 65 //Create byte data using Integer Literal which is then implicitly converted to byte
data.
```

Short

- Short data type is type of data that represents integer number and can be created using Integer Literal.
- Check out short cheat sheet containing code samples for working with short data type.

short

```
var value: Short = 65 //Create short data using integer literal which is implicitly converted to short data.
```

Int

- Int data type is type of data that represents integer number and can be created using Integer Literal.
- Check out int cheat sheet containing code samples for working with int data type.

int

```
//CREATE INTEGER DATA USING INTEGER LITERAL IN DIFFERENT NOTATIONS.
var value : Int = 65                        //Decimal.
    value = 0x41                            //Hexadecimal. 0X41
    value = 0b1000001                       //Binary.      0B1000001
    value = 10_000                          //You can use underscore any way you like: 10_000, 0b10__00_001
```

Long

- Long data type is type of data that represents integer number and can be created using Long Literal.
- Check out long cheat sheet containing code samples for working with long data type.

Test.java

```
//CREATE LONG DATA USING INTEGER LITERAL IN DIFFERENT NOTATIONS.
var longValue = 1000L         //Use "L" sufix to specify Long value when Data Type is not declared
var value : Long = 65         //Decimal.
    value = 0x41              //Hexadecimal. 0X41
    value = 0b1000001         //Binary.      0B1000001
    value = 10_000            //You can use underscore any way you like: 10_000, 0b10__00_001
```

Float

- Float Data Type is type of data that represents real number and can be created using Float Literal.
- Check out float cheat sheet containing code samples for working with float data type.

float

```
//CREATE FLOAT DATA USING FLOAT LITERAL IN DIFFERENT NOTATIONS.
var value : Float = 65.23f                    //Basic      notation. 65F
    value = 0.6523E2f                         //Scientific notation. 0.6523E2 F
```

Double

- Double data type is type of data that represents real number and can be created using Double Literal.
- Check out double cheat sheet containing code samples for working with double data type.

double

```
//CREATE DOUBLE DATA USING DOUBLE LITERAL IN DIFFERENT NOTATIONS.
var value : Double = 65.23                    //Basic      notation. 65F
    value = 0.6523E2                          //Scientific notation. 0.6523E2 F
```

2.1.3 Collections

Info

- Collection Data Types are implemented as Structures and allow us to group multiple items of the same Data Type.

Collection Data Types

TYPE	EXAMPLE	DESCRIPTION
String		Class for storing constant strings.
Array		Data of the same Data Type.

String

- String Data Type represents **Array of Characters** and can be created using String Literal.
- Check out String cheat sheet containing code samples for working with String data type.

String Data Type

```
//DECLARE STRINGS.
var name : String   = "John"            //Explicit Data Type declaration.
var name1           = "John"            //Implicit Data Type declaration.

//ACESS CHARACTERS.
var length          = name.length       // 4
var firstCharInName = name[0]           // 'J'
var lastCharInName  = name[length - 1]  // 'n'

//CONCATENATE & REFERENCE VARIABLES.
var age : Int = 20
print(name + " is " + "$age years old.")
```

Array [R] [R]

- In Kotlin Arrays are represented using Array<T> class. (Array<Int> is translated to Integer[])
  There is also IntArray(1,2,3) which is translated as int[] and can be converted using IntArray.toTypedArray().
- Array can be created using
  - arrayOf<Int> () library Function
  - Array() Constructor
    - first parameter is the size of Array
    - second parameter is function that takes array index and returns element to be inserted at that index

Array

```
//DECLARE ARRAYS USING: arrayOf() library function
var integers1 : Array<Int> = arrayOf<Int>(1, 2, 3, 4, 5)
var integers2 : Array<Int> = arrayOf     (1, 2, 3, 4, 5)
var integers3              = arrayOf<Int>(1, 2, 3, 4, 5)
var integers               = arrayOf     (1, 2, 3, 4, 5)
var strings                = arrayOf     ("Cat", "Dog", "Lion", "Tiger")
var objects                = arrayOf     (1, true, 3, "Hello", 'A')       //Mixed Data Types

//DECLARE ARRAYS USING: Array() Constructor
var mySquareArray = Array(5, {i -> i * i})                              // [0, 1, 4, 9, 16]

//ACCESS ELEMENTS.
val size  = integers.size
val first = integers[0]
val last  = integers[size - 1]
integers[0] = 10
```

2.2 Variables

Info

- Following tutorials cover some basic operations related to single value variables.

2.2.1 Constants

Info

- Constant has value which cannot be changed once it is set (unlike Variable which can change its value multiple times)
- Referencing Constant that has no Value gives error: variable 'age' must be initialized.
  This is because only Optional Constants are allowed to be nil. This is why Optionals were introduced in the first place to detect these kind of errors at compile time rather than during run time.

Syntax [R]

- Constant is declared by
  - using required Keyword val val
  - followed by required Constant's Name name
  - followed by optional Constant's Data Type String (Optional only if Value is given from which it can be inferred)
  - followed by optional Constant's Value "John"

Example

- In this example we declare
  - Constant name by specifying Data Type as String and by giving it an initial value of "John"
  - Constant age by specifying Data Type as Int since no initial Value was given from which Type could be inferred
  - Constant weight by omitting Data Type since it can be inferred as Int from the given initial Value of 180

Syntax

```
//DECLARE CONSTANT.
val name   : String = "John"  //Full Syntax with both Data Type and initial value
val age    : Int              //Must Declare Data Type since there is no initial value
val weight          = 180     //Data Type can be ommited since it is infered from initial value

//ASSIGN NEW VALUE TO CONSTANT.
age = 20                      //Constant's value can be set only once
//age = 30                    //This line would throw error since we are trying to change value of a Constant

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

2.2.2 Variables

Info

- Variable has value which can be changed any number of times (unlike Constants which always hold initial value).
- Referencing Variable that has no Value gives error: variable 'age' must be initialized.
  This is because only Optional Variables are allowed to be nil. This is why Optionals were introduced in the first place to detect these kind of errors at compile time rather than during run time.

Variable Syntax [R]

- Variable is declared by
  - using required Keyword var var
  - followed by required Variable's Name name
  - followed by optional Variable's Data Type String (Optional only if Value is given from which it can be inferred)
  - followed by optional Variable's Value "John"

Example

- In this example we declare
  - Variable name by specifying Data Type as String and by giving it an initial value of "John"
  - Variable age by specifying Data Type as Int since no initial Value was given from which Type could be inferred
  - Variable weight by omitting Data Type since it can be inferred as Int from the given initial Value of 180

Variable Syntax

```
//DECLARE VARIABLE.
var name   : String = "John"  //Full Syntax with both Data Type and initial value
var age    : Int              //Must Declare Data Type since there is no initial value
var weight          = 180     //Data Type can be ommited since it can be inferred as Int from initial value

//ASSIGN NEW VALUE TO VARIABLE.
age = 20                      //Variable's value can be changed any number of times

//DISPLAY VARIABLE.
print(name)
```

2.2.3 Optionals

Info [R]

- Optional is Constant or Variable that can be **null** to indicate absence of value (hence the name since value is optional).
  This is symbolized by appending **question mark '?'** to Data Type: String**?**, Int**?**.
  Question mark '?' symbolizes that we are wondering if Optional contains Value of specified Data Type or if it is empty.
  If Optional is of type String?, and we set it to null, it means that Optional is empty (it has no String value stored in it).
- When you reference Optional which
  - is null you get **null**
  - has Value you get that Value **"John"**
  - is not initialized you get variable 'age' must be initialized

Syntax

- Optional is declared by

• using required Keyword	let or var	let or var	(to declare Constant or Variable)
• followed by required	Name	name	
• followed by required	Data Type	String	
• followed by required **question mark**	?	?	
• followed by optional	Value	"John"	

Syntax

```
//DECLARE OPTIONAL.
let name   : String? = "John" //Optional Constant
var age    : Int?    = 20     //Optional Variable
var height : Int?    = null   //null
var weight : Int?             //Not initialized. Throws error if referenced without asigning a value.

//REFERENCE OPTIONAL.
name = "Bill"
println(name)                 //Bill
println(height)               //null
```

2.3 Basic Syntax

Info

- Following tutorials cover some basic Syntax for creating code.

2.3.1 Comments

Info

- Comment is a text that describes the purpose of the source code (it is not being executed).

Comment Syntax

- Comments are created by writing text
  - after double slash **"//"** for single line Comments //Single line comment.
  - between **"/* ... */"** for multi-line Comments /* Multi line comment. */

Comments

```
//Single line comment.
var name = "John"

/* Multi line comment.
   Second line.     */
var age =20
```

2.3.2 Print to Console

Info

- To print to Console you can use **print()** Functions.
- Inside the String you can reference Properties by using **$name**.

Print

```
//DECLARE VARIABLES.
var name = "John"
var age  = 20

//DISPLAY VARIABLES.
println("Hello")
print  ("$name is $age years old")         //John is 20 years old
print  (name + " is " + "$age years old")  //John is 20 years old
```

2.3.3 Statements - Conditional

Info

- Conditional Statements execute its Body single time if Condition is TRUE
  - if ... else Statement executes single section for which condition is TRUE
  - when Statement is used instead if ... else Statement when all conditions evaluate the same variable

if ... else

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

//IF ELSE STATEMENT (CAN RETURN VALUE).
var result : String =
if      (number ==  10    )   "Number is 10"                       //Can ommit {} for single line
else if (text   == "Hello") { "Text equals Hello"                } //Don't use return to return value
else if (text   == "World") { "Text equals World"                }
else                        { print("No match"); "No match found" }
print(result)
```

when *[R]*

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

//WHEN STATEMENT (CAN RETURN VALUE).
var result : String =
when(number) {
  10     ->   "number = 10"                            //Can ommit {} for single line
  20, 30 -> { "number = 20 or 30"                  }   //It doesn't fall through to subsequent case
  else   -> { print("No match"); " No match found" }   //Don't use return to return value
}
print(result)
```

2.4 Structures

Info

- Following tutorials show how to work with more complex structures that can hold multiple variables or Methods.

2.4.1 Objects

Info [R] [R]

- Kotlin Object represents a **single static instance of its Properties and Methods**.
- Kotlin Object is like a Class that only has static Properties and Methods and from which you can't create Instances.
  Since you can't create Instances **it doesn't have Constructor** and you can't create it using Parameters.
  Constructor isn't needed since when you create it you immediately set all the Properties to wanted Values.
- Kotlin Objects are useful when you don't need multiple instances of the same Class.
  They are similar to Closures since they are also used when we don't need to call the same Function multiple times.
- Properties and Methods are **public** by default but you can declare them as **private**.
- Object can **extend single Class** and **Implement multiple Interfaces**.

Basic Object Syntax

- Object is declared by
  - using Keyword `object` `object`
  - followed by the Object `Name` `Person`
  - followed by the Object `Body` `{ ... }`
    - which can have Properties `val age = 20` (Variables, Constants, Enumerators,...)
    - which can have Methods `fun greet() { ... }`

Basic Object Syntax

```
object Person { ... }
```

Basic Object Syntax

```
//=====================================================================================================================
// OBJECT: Person
//=====================================================================================================================
object Person {
  val age = 20
  fun greet(name: String) = "$name is $age years old"
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {
  var result = Person.greet("John")
  println(result)
}
```

2.4.2 Classes

Info [R]

- Classes
  - can contain **Properties** & **Methods** (add **private** Keyword to those that should not be accessible from the outside)
  - must have at least one **constructor(...)** Method to create its Instance
- Content
  - Basic Class Syntax
  - Extend Class
  - Constructors (Parameters become Properties only if you use var or val.)
  - Companion Object (Workaround for implementing static Methods and Properties)

Basic Class Syntax

- Class is declared by
  - using Keyword `class` `class`
  - followed by the Class Name `Name` `Person`
  - followed by the Class Body `Body` `{ ... }`
    - which can have `constructor` `constructor()` (Method used to create Class Instance)
    - which can have Properties `var name : String` (Variables, Constants, Enumerators,...)
    - which can have Methods `fun sayHello() { ... }`

Basic Class Syntax

```
class Person { ... }
```

Basic Class Syntax

```
//=====================================================================================================================
// CLASS: Person
//=====================================================================================================================
class Person {

  //DECLARE PROPERTIES.
  public  var name : String = ""  //Default public Properties can be accessed from the outside
  private var age  : Int    = 0   //Private Properties can only be accessed from Class Properties & Methods

  //CONSTRUCTOR.
  constructor(name: String, age: Int) {
    this.name = name
    this.age  = age
  }

  //DECLARE METHODS.
  fun sayHello() {
    println("$name is $age years old")
  }

}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {
var john = Person("John", 20)  //Create Class Instance by calling its constructor() Method
    john.sayHello()            //John is 20 years old
    print(john.name)           //John
}
```

Extend Class [R]

- Child Class can **extend single** Parent Class by adding **: ParentClassName** after Child Class Name.
- Use Keyword **override** to create Method that has the same signature as an existing Parent's one.
  When overriding method with default parameter values, these values must be omitted from the signature.
- Use Keyword **super** to access Parent's Properties and Methods `super.displayName()`.
- You can only extend from a Class or override a Method that were declared using Keyword **open**.

Extend Class Syntax

```
class Soldier : Person { ... }
```

Extend Class

```
//=====================================================================================================================
// PARENT CLASS: Person
//=====================================================================================================================
open class Person {
       var name = ""
  open fun displayName() { print(name) }
  constructor(name: String) { this.name = name }                              //Secondary Constructor

}

//=====================================================================================================================
// CHILD CLASS: Soldier
//=====================================================================================================================
class Soldier : Person {                                                      //Extend from Person Class
  override fun displayName() { println("Hello $name") }                       //Override Method
           fun greet() { super.displayName() }                                //Call Parent's Method
  constructor(name: String) : super(name) { println("Secondary Constructor") } //Call Parent's Constructor
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {

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

  //DISPLAY NAME.
  print(name)

}
```

Constructors [R] [R]

- Kotlin Class must have Constructor
  - Default Constructor is created by compiler if you don't explicitly declare any Constructor
  - Primary Constructor is declared before the body and its implementation is given inside init{} block
    - there can be only one Primary Constructor
    - you don't have to declare one but if there is one, Secondary Constructors must eventually call it
  - Secondary Constructor is declared inside the Body
    - you can have multiple Secondary Constructors with or without Primary Constructor
    - if there is Primary Constructor, Secondary Constructors must call it `constructor(name: String) : this(name, 20) {`

Just Primary Constructor

- **Input parameters of Primary Constructor** are declared after Class Name and before the Class Body.
  - If you declare them with **var** or **val** they automatically become **Class Properties**
    You can reference them with their **name** or inside the string with **$name** or **${name}** (for complex expressions)
    So just like in the Function you don't need to redeclare them inside the Class.
    Additionally you can specify them to be private (like you can with any Class Property).
  - Without var or val they are **just Constructor Parameters** (and not Class Properties.)
    Then you can only reference them inside `init{}` blocks and during Properties initialization (in the same way as above).
- Actual implementation of Primary Constructor can be defined inside multiple `init{}` blocks and Properties Declarations which are all executed in the given order (inside the Class Body as if representing a Function Body).
- Using **constructor keyword** is optional. Without it declaration syntax looks more like that of a Function.

Primary Constructor Syntax

```
class Person constructor(public var name: String,     age: Int) { ... init{} ... init{} ... } //Optional
class Person           (       var name: String, val age: Int) { ... init{} ... init{} ... } //Properties
class Person           (           name: String,     age: Int) { ... init{} ... init{} ... } //Parameters
```

Just Primary Constructor

```
//=====================================================================================================================
// CLASS: Person
//=====================================================================================================================
class Person constructor(name: String = "", public var age: Int = 0) {

  //PRIMARY CONSTRUCTOR IMPLEMENTATION.
  init { println("name is $name") }  //name is only Constructor Parameter

  //DECLARE & INITIALIZE PROPERTIES.
  var greet : String = "Hello $name" //Use Constructor Parameter name to initialize Property

  //PRIMARY CONSTRUCTOR IMPLEMENTATION.
  init { println("age is ${age}") }  //age is both Constructor Parameter & Class Property

  //DECLARE METHODS.
  fun sayHello() {
    println("$greet. You are ${age} years old") //Can't reference name since it is not Property
  }
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {
var john = Person("John", 20)  //Create Class Instance by calling its constructor() Method
    john.sayHello()            //Hello John. You are 20 years old
    print(john.age)            //20 Because of var, age Parameter was created as Class Property

}
```

Just Secondary Constructor

- Secondary Constructor is declared inside the Body by using Keyword constructor.
- It is automatically used if number and type of Parameters don't match Primary Constructor.

Secondary Constructor Syntax

```
constructor(name: String, age: Int) { ... }
```

Just Secondary Constructor

```
//=============================================================================================================
// CLASS: Person
//=============================================================================================================
class Person {

  //DECLARE PROPERTIES.
  public  var name : String = ""  //Default public Properties can be accessed from the outside
  private var age  : Int    = 0   //Private Properties can only be accessed from Class Properties & Methods

  //CONSTRUCTOR.
  constructor(name: String, age: Int) {
    this.name = name
    this.age  = age
  }

  //DECLARE METHODS.
  fun sayHello() {
    println("$name is $age years old")
  }

}

//=============================================================================================================
// FUNCTION: main
//=============================================================================================================
fun main() {

//CREATE OBJECT.
var john = Person("John", 20)  //Create Class Instance by calling its constructor() Method
    john.sayHello()            //John is 20 years old
    print(john.name)           //John

}
```

Both Primary & Secondary Constructors

- If both Primary and Secondary Constructors are defined, Secondary Constructor must call Primary Constructor before executing its own Body by using **this** Keyword (highlighted in Red).
- Since name is declared as Variable in Primary Constructor it automatically becomes Property (as in Functions).

Primary & Secondary Constructor Syntax

```
class Person(name: String, age: Int) {
  init { ... }
  constructor(name: String) : this(name, 20) { ... } //Call Primary Constructor before executing Secondary
}
```

Both Primary & Secondary Constructors

```
//=============================================================================================================
// CLASS: Person
//=============================================================================================================
class Person(public var name: String, age: Int) { //Primary constructor without constructor Keyword.

  //DECLARE PROPERTIES.
  private var age  : Int    = 0   //Private Properties can only be accessed from Class Properties & Methods

  //PRIMARY CONSTRUCTOR IMPLEMENTATION.
  init {
    println("PRIMARY CONSTRUCTOR IMPLEMENTATION")
    this.age  = age
  }

  //SECONDARY CONSTRUCTOR.
  constructor(name: String) : this(name, 20) {
    println("SECONDARY CONSTRUCTOR")
  }

  //DECLARE METHODS.
  fun sayHello() {
    println("$name is $age years old")
  }

}

//=============================================================================================================
// FUNCTION: main
//=============================================================================================================
fun main() {

//CREATE OBJECT.
var john = Person("John")      //Create Class Instance by calling its constructor() Method
    john.sayHello()            //John is 20 years old
    print(john.name)           //John

}
```

Companion Object [R]

- Companion Object is a workaround to **add static Methods & Properties to a Class** (which Kotlin doesn't support).
- They are accessed using Class name and are shared across all Class Instances.
- A Companion Object is always declared inside of another Class with the default name of Companion.
- Use **@JvmStatic** annotation on the Object's Methods & Properties that you want to make static for that Class.

Companion Object

```
//=============================================================================================================
// CLASS: Person
//=============================================================================================================
class Person {
  companion object Companion {
    @JvmStatic val name = "John"
    @JvmStatic fun greet(age: Int) : String { return "$name is $age years old" }
  }
}

//=============================================================================================================
// FUNCTION: main
//=============================================================================================================
fun main() {
  var name   = Person.name         //Call static Property using Class name.
  var result = Person.greet(20)    //Call static Method   using Class name.
  println("Hello $name")
  println(result)
}
```

2.4.3 Fields

Info

- **Field** is a **Constant** or **Variable** declared inside a Class or Object
  - **public** Fields can be referenced **inside** and **outside** of the Class or Object (by behaviour)
  - **private** Fields can be referenced only **inside** the Class or Object (from Class Fields, Properties, Methods)
- Use **this** to reference Fields from within Class (Fields, Properties, Methods).

Fields

```
//=====================================================================================================================
// CLASS: Person
//=====================================================================================================================
class Person {

  //DECLARE FIELD.
  public var name : String = ""

  //CONSTRUCTOR.
  constructor(name: String) { this.name = name }  //Use this to reference Field

}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {

    //CREATE OBJECT.
    var john = Person("John")
        john.name = "Bill"    //Set  Field
    var greet = john.name     //Read Field

    //DISPLAY.
    print(greet)              //Bill

}
```

2.4.4 Properties

Info

- **Property** is a **named** pair of **get()** and **set()** Methods which are accessed using Field Syntax
  - name = "John" actually calls name.set("John")
  - result = name actually calls name.get("John")
- This means that **Property doesn't need to store any Value**. Its Methods can simply call other Fields, Properties and Methods. Therefore both get() and set() Methods can store multiple Values into multiple other Fields or Properties. And get() Method can return value which is just a combination of different other Fields, Properties and Methods.
- But if needed Property can have a so called **Backing Field** to store a single Value.
  Backing Field is referenced from get() and set() Methods using **field** Keyword.
  Backing Field is not visible in the Kotlin Code. Instead it is added to a generated Java Code.
- Both **get()** and **set()** Methods are optional. If none is defined we get a Field.
  But in Java generated code, Kotlin might add default get() and set() Methods implementing Field as a Property.
- Use **this** to reference Properties from within Class (Fields, Properties, Methods).
- Properties can also be
  - **public** Properties can be referenced **inside** & **outside** of the Class or Object (by default)
  - **private** Properties can be referenced only **inside** the Class or Object (from Fields, Properties, Methods)

Properties

```
//=====================================================================================================
// CLASS: Person
//=====================================================================================================
class Person {

  //DECLARE PROPERTY.
  public var name : String = ""                    //Data Type returned by get() & input into set(value).
             get()      { return "Hello $field" }  //Optional
             set(value) { field = value         }  //Optional

  //CONSTRUCTOR.
  constructor(name: String) { this.name = name }   //Use this to reference Property

}

//=====================================================================================================
// FUNCTION: main
//=====================================================================================================
fun main() {

    //CREATE OBJECT.
    var john = Person("John")
        john.name = "Bill"    //name.set("Bill")
    var greet = john.name     //name.get()

    //DISPLAY.
    print(greet)              //Hello Bill

}
```

2.5 Functions

Info

- **Function** is
  - block of code
  - that accepts Input Parameters
  - and returns a single value
- In Kotlin Function can be declared using following **Programming Syntaxes**
  - **Lambda Expression** declares **Anonymous Function** (Function that has no Name)
  - **Function Declaration** declares **Named Function** (Function that has a Name)
- **Closure** is Object that is create when Function needs to be returned or used as Input Parameter of another Function
  - has a single Method
  - has Properties which are all referenced from that Method and contains current Values of outer Variables

2.5.1 Function Declaration - Named

Info [R] [R]

- Function Declaration declares **Named Function** (Function that has a Name)
- Each Function has **Signature** which defines
  - Data Types of Input Parameters (name:String, age:Int)
  - Data Type of Return Value (String)
  - Colon **":"** inside Signature symbolizes that these Input Parameters are converted into this Output Parameter.
- Since Functions can be stored inside Variables or given as Input Parameters into other Functions, **Signature is used** to say
  - what kind of Function can be stored into a Variable
  - what kind of Function can be given as Input Parameter into another Function
- For instance `var name : Int` says that this Variable can store Int Data Type.
  Similarly `var name : (name:String, age:Int) -> (String)` says that this Variable can store Function that
  - accepts String & Int as Input Parameters
  - returns String
- Content
  - Syntax - Block `fun greet (name:String, age:Int) : (String) { ... }`
  - Syntax - Assignment `fun greet (name:String, age:Int) : (String) = Single Line`
  - Input Parameters `fun greet (name:String, age:Int, height: Int = 170)` `greet("John", age=20)`
  - Return Value `fun greet1(name: String, age: Int) : Unit { ... }`
  - Function Scope (Top Level, Local, Member, Extension Functions)
  - Function as Input Parameter
  - Function as Return Value
  - Infix Methods (Methods with single Parameter, no default value, no vararg)

Syntax

```
//NAMED FUNCTION DECLARATIONS.
fun greet (name:String, age:Int = 20) : (String)   { return("$name is ${age} years old") }//Block      Syntax
fun greet (name:String, age:Int)      : (String) = "$name is ${age} years old"            //Assignment Syntax

//CALL FUNCTION BY NAME.
greet(name="John", age=50)                                  //Named   Parameters
greet("John", age=50)                                       //Indexed before Named Parameters
greet("John", 50)                                           //Indexed Parameters
greet("John")                                               //Default Parameters
```

Syntax - Block

- Function Declaration
  - uses Keyword `fun` `fun`
  - followed by Function's Name `greet`
  - followed by Function's Signature `(name:String, age:Int) : (String)` (accepts 2 Parameters & returns String)
  - followed by Function's Body `{ return("$name is $age years old.") }` (requires return Keyword)

Block Syntax

```
//DECLARE & IMPLEMENT FUNCTION.
fun greet (name:String, age:Int) : (String) { return("$name is $age years old.") }

//CALL FUNCTION.
var result = greet("John", 50)

//DISPLAY RESULT.
print(result)
```

Syntax - Assignment

- Function is declared by
  - using Keyword `fun` `fun`
  - followed by Function's Name `greet`
  - followed by Function's Signature `(name:String, age:Int) : (String)` (accepts 2 Parameters & returns String)
  - followed by equal sign `"="` `"="`
  - followed by single statement `"$name is $age years old."` (forbidden return Keyword)

Assignment Syntax

```
//DECLARE & IMPLEMENT FUNCTION.
fun greet (name:String, age:Int) : (String) = "$name is $age years old."

//CALL FUNCTION.
var result = greet("John", 50)

//DISPLAY RESULT.
print(result)
```

Input Parameters [R]

- Function Input Parameters
  - can have **default values** `greet (height: Int = 170)`
  - can be called without their names `greet("John", 20)`
    - such **Indexed Parameters** must be given in the same order in which they are declared in a Function
  - can be called with their names in which case `greet(age=20, name="John")`
    - order of such **Named Parameters** doesn't matter
    - default parameters can be at any position
  - can be called with the **combination** of Indexed & Named Parameters in which case `greet("John", age=20)`
    - Indexed Parameters need to come before the Named Parameters

Default, Indexed & Named Parameters

```
//DECLARE FUNCTION WITH DEFAULT PARAMETER.
fun greet (name:String, age:Int, height: Int = 170) : Unit {
  print("$name is $age years old and $height cm tall.")
}

//CALL FUNCTION.
greet("John", 20)           //Parameters can be specified without their names. Then order matters.
greet("John", age=20)       //When mixing Indexed & Named Parameters, Indexed Parameters need to come first.
greet(age=20, name="John")  //When using named Parameters order doesn't matter
```

Variable Number of Arguments

- Function can have one **vararg** Parameter
  - vararg must be first `fun sum(vararg numbers: Int, base: Int)`
  - vararg is internally represented as Array<T>
  - other Parameters must be called with their names (Named Parameters) `sum(1, 2, 3, 4, base = 100)`
  - Array elements can be passed using star **"*"** prefix ***numbers** `sum(*numbers, base = 100)`

Variable Number of Arguments

```
//DECLARE FUNCTION WITH VARARG PARAMETER.
fun sumOfNumbers(vararg numbers: Int, baseNumber: Int) : Unit {
  var sum : Int = baseNumber
  for(number in numbers) { sum += number }
  println(sum)
}

//CALL FUNCTION WITH VARIABLE NUMBER OF ARGUMENTS.
sumOfNumbers(1, 2, 3, 4, baseNumber = 100)    //Other Parameters must be called with their names
```

- When using Array to provide elements to vararg Parameter you can use spread operator **(*)** which only works on `intArray`.

From intArray

```
//CALL FUNCTION WITH ARRAY ELEMENTS.
val numbers : IntArray = intArrayOf(1, 2, 3, 4)
sumOfNumbers(*numbers, baseNumber = 100)              //vararg Parameter can be loaded with Array Elements
```

- If you have Array<Int> you have to convert it first.

From Array<Int>

```
//CALL FUNCTION WITH ARRAY ELEMENTS.
var numbers : Array<Int> = arrayOf<Int>(1, 2, 3, 4)
sumOfNumbers(*numbers.toIntArray(),baseNumber=10)      //vararg Parameter can be loaded with Array Elements
```

Return Value [R]

- Functions only have a single Return Value. Return Value Data Type is part of their Signature.

Unit

- Functions which don't return anything have a return type of Unit (corresponds to void in Java).
- If Function doesn't return any value you don't need specify Unit in the Signature and you don't need return Keyword.

Unit

```
fun greet1(name: String, age: Int) : Unit { print("$name is $age years old") }
fun greet2(name: String, age: Int)        { print("$name is $age years old") }
greet1("John", 20)
```

Function Scope [R]

- Based on Scope Kotlin Functions can be
  - **Top Level** Functions (Can be defined at the top level **inside** a source **file** outside any Object or Class)
  - **Local** Functions (Functions defined **inside** another **Function**)
  - **Member** Functions (Methods defined **inside** an Object or **Class**)
  - **Extension** Functions (Methods **added to** an existing Object or **Class** through Extensions)

Function Scope

```
//=====================================================================================================================
// DECLARE FUNCTIONS
//=====================================================================================================================
fun topLevelFunction() { println("Top level Function") }                //Top level Function

fun outerFunction() {
  fun localFunction() { println("Local/Inner Function") }               //Local/Inner Function
  localFunction()
}

class Person {
 fun memberFunction() { println("Member Function") }                   //Member Function (Method)
}

fun Person.extensionFunction() { println("Extension Function") }        //Extension Function

//=====================================================================================================================
// CALL FUNCTIONS
//=====================================================================================================================
fun main() {

  topLevelFunction()              //Top level Function
  outerFunction()                 //Calls Local/Inner Function
  var person = Person()
      person.memberFunction()     //Member Function (Method)
      person.extensionFunction()  //Extension Function

}
```

Function as Input Parameter

- When you compare this code with Lambda Expressions as Function Parameter you see that
  - using Lambda Expression allows us to declare and use Anonymous Function in the place of Input Parameter
  - using Function Declaration we first need to declare the Named Function and then use its Name as Input Parameter

Lambda Expressions as Function Parameter

```
executeFunction({ name: String, age: Int -> "$name IS $age YEARS OLD." })
```

Function as Input Parameter

```
//=====================================================================================================================
// FUNCTION: executeFunction
//=====================================================================================================================
fun executeFunction(receivedFunction: (name: String, age: Int) -> String) {
  var result = receivedFunction("John", 20)
  println(result)
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {

  //DECLARE LOCAL VARAIBLE.
  var weight = 150

  //DECLARE NAMED FUNCTION.
  fun namedFunction (name: String, age: Int) : String {
    return("$name is $age years old and $weight kg.")
  }

  //USE NAMED FUNCTION AS PARAMETER (SENDS CLOSURE WITH weight Property).
  executeFunction(::namedFunction)

}
```

Function as Return Value

- Function getFunction() actually returns a Closure with
  - this Named Function as its Method
  - Property **weight** which is declared outside of the Scope of Named Function
- When you compare this code with Lambda Expressions as Return Value you see that
  - using Lambda Expression allows us to declare and return Anonymous Function in one line
  - using Function Declaration we first need to declare the Named Function and then use its Name to return it

Lambda Expressions as Return Value

```
return { name: String, age: Int -> "$name is $age years old and $weight kg." }
```

Function as Return Value

```
//=====================================================================================================================
// FUNCTION: getFunction
//=====================================================================================================================
fun getFunction (weight: Int) : (name: String, age: Int) -> String {

  //DECLARE NAMED FUNCTION.
  fun returnedFunction(name: String, age: Int) : String {
    return "$name is $age years old and $weight kg."
  }

  //RETURN NAMED FUNCTION.
  return ::returnedFunction

}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {

  //DECLARE VARIABLE TO HOLD FUNCTION DATA TYPE.
  var returnedFunction : (name: String, age: Int) -> String

  //DECLARE ANONYMOUS FUNCTION USING LAMBDA EXPRESSION.
  returnedFunction = getFunction(150)            //Returns Closure with partialy initialized Function

  //CALL RETURNED FUNCTION.
  var result = returnedFunction("John", 20)      //Call Function with missing Parameters

  //DISPLAY RESULT.
  print(result)

}
```

Infix Methods [R]

- Infix Method is declared like any other Method except that it has Keyword infix.
  You can also call it like a normal Method but also in this special infix way by **omitting dot** and **parentheses**.
- Infix Methods
  - **must** be **member** or **extension functions** (Methods)
  - **must** have a **single parameter**
    - Parameter **can't** have **default value**
    - Parameter **can't** be **vararg** (can't accept variable number of arguments)

Infix Method

```
//=====================================================================================================================
// CLASS: Person
//=====================================================================================================================
class Person(var age: Int) {
  infix fun addAge(age: Int) : Int {
    return this.age + age
  }
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {

//CALL METHOD.
var person = Person(20)
var newAge = person.addAge(10) //Normal call
    newAge = person addAge 10  //Infix  call

//DISPLAY RESULT.
print(newAge)

}
```

2.5.2 Function Declaration - Anonymous

Info **[R]**

- Anonymous Function Declaration declares **Anonymous Function** (Function that has a Name).
- This is alternative to Lambda Expressions which also declares Anonymous Function and it was introduced because
  - **return** keyword returns from the nearest function declared with the **fun** keyword
  - so if return is used inside Lambda it will return from the enclosing function
  - and to return just from Lambda requires the return@label syntax (which is somewhat clunky)
  - to avoid clunkiness Anonymous Function Declarations allows you to use return to return just from the Function
- A secondary reason is that it's indeed impossible to fit the return type declaration into the syntax of a lambda.
- Anonymous Function Declarations is used in the same way as Named Function Declarations except that
  - you **avoid specifying name**
  - just like with Lambda Expressions you either (which means you can't call it with **named Parameter** either)
    - save Anonymous Function in a Variable
    - or declare it directly at the place of Function Input Parameter (so that it gets stored in that Parameter)
  - supports both **Block** and **Assignment Syntax**
  - you can specify **Data Type of Return Value**
- Content
  - Syntax - Block
  - Syntax - Assignment
  - Function as Input Parameter
  - Function as Return Value

Syntax

```
//ANONYMOUS FUNCTION DECLARATIONS.
fun (name:String, age:Int) : (String)   { return("$name is ${age} years old.") } //Block      Syntax
fun (name:String, age:Int) : (String) = "$name is ${age} years old."              //Assignment Syntax

//CALL FUNCTION THROUGH VARIABLE.
greet("John", 50)                                       //No Default Values or Named Parameters
```

Syntax - Block

- Function Declaration
  - uses Keyword `fun` `fun`
  - followed by Function's Signature `(name:String, age:Int) : (String)` (accepts 2 Parameters & returns String)
  - followed by Function's Body `{ return("$name is $age years old.") }` (required return Keyword)

Block Syntax

```
//DECLARE VARIABLE THAT CAN HOLD FUNCTION DATA TYPE (OF SPECIFIED SIGNATURE).
var greet: (String, Int) -> (String)        //Closure that accepts 2 Parameters & returns String

//DECLARE & IMPLEMENT FUNCTION.
greet = fun (name:String, age:Int) : (String) { return("$name is $age years old.") }

//CALL FUNCTION.
var result = greet ("John", 50)

//DISPLAY RESULT.
print(result)
```

Syntax - Assignment

- Function is declared by
  - using Keyword `fun` `fun`
  - followed by Function's Signature `(name:String, age:Int) : (String)` (accepts 2 Parameters & returns String)
  - followed by equal sign `"="` `"="`
  - followed by single statement `"$name is $age years old."` (forbidden return Keyword)

Assignment Syntax

```
//DECLARE VARIABLE THAT CAN HOLD FUNCTION DATA TYPE (OF SPECIFIED SIGNATURE).
var greet: (String, Int) -> (String)        //Closure that accepts 2 Parameters & returns String

//DECLARE & IMPLEMENT FUNCTION.
greet = fun (name:String, age:Int) : (String) = "$name is $age years old."

//CALL FUNCTION.
var result = greet("John", 50)

//DISPLAY RESULT.
print(result)
```

Function as Input Parameter

- When you compare this code with Lambda Expressions as Function Parameter you see that it is as compact while
  - allowing us to specify Data Type of Return Value
  - use normal return
  - use Syntax that is similar to Named Function Declarations (unlike Lambda that uses completely different Syntax)

Lambda Expressions as Function Parameter

```
executeFunction({ name: String, age: Int -> "$name IS $age YEARS OLD." })
```

Function as Input Parameter

```
//=====================================================================================================================
// FUNCTION: executeFunction
//=====================================================================================================================
fun executeFunction(receivedFunction: (name: String, age: Int) -> String) {
  var result = receivedFunction("John", 20)
  println(result)
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {

  //DECLARE LOCAL VARIABLE.
  var weight = 150

  //DECLARE VARIABLE TO HOLD FUNCTION DATA TYPE.
  var anonymousFunction : (name: String, age: Int) -> String

  //DECLARE ANONYMOUS FUNCTION USING LAMBDA EXPRESSION.
  anonymousFunction = fun (name: String, age: Int) : String {return "$name is $age years old and $weight
kg."}

  //CALL FUNCTION WITH FUNCTION AS PARAMETER (SENDS CLOSURE WITH weight Property).
  executeFunction(anonymousFunction)

  //USE LAMBDA EXPRESSION DIRECTLY AS PARAMETER (SENDS CLOSURE WITH weight Property).
  executeFunction(fun (name: String, age: Int) : String { return "$name is $age years old and $weight kg." })

}
```

Function as Return Value

- Function getFunction() actually returns a Closure with
  - this Anonymous Function as its Method
  - Property **weight** which is declared outside of the Scope of Anonymous Function
- When you compare this code with Lambda Expressions as Function Parameter you see that it is as compact while
  - allowing us to specify Data Type of Return Value
  - use normal return
  - use Syntax that is similar to Named Function Declarations (unlike Lambda that uses completely different Syntax)

Lambda Expressions as Return Value

```
return { name: String, age: Int -> "$name is $age years old and $weight kg." }
```

Function as Return Value

```
//=============================================================================================================
// FUNCTION: getFunction
//=============================================================================================================
fun getFunction (weight: Int) : (name: String, age: Int) -> String {

    //DECLARE VARIABLE TO HOLD FUNCTION DATA TYPE.
    var anonymousFunction : (name: String, age: Int) -> String

    //DECLARE ANONYMOUS FUNCTION.
    anonymousFunction = fun (name: String, age: Int) : String {
        return "$name is $age years old and $weight kg."
    }

    //RETURN ANONYMOUS FUNCTION.
    return anonymousFunction

    //USE ANONYMOUS FUNCTION DECLARATION DIRECTLY AS RETURN VALUE.
    return fun (name: String, age: Int) : String { return "$name is $age years old and $weight kg." }

}

//=============================================================================================================
// FUNCTION: main
//=============================================================================================================
fun main() {

//DECLARE VARIABLE TO HOLD FUNCTION DATA TYPE.
var returnedFunction : (name: String, age: Int) -> String

//DECLARE ANONYMOUS FUNCTION USING LAMBDA EXPRESSION.
returnedFunction = getFunction(150)            //Returns Closure with partialy initialized Function

//CALL RETURNED FUNCTION.
var result = returnedFunction("John", 20)      //Call Function with missing Parameters

//DISPLAY RESULT.
print(result)

}
```

2.5.3 Lambda Expressions

Info [R] [R]

- **Lambda Expression** (Lambda for short) declares **Anonymous Function** (Function that has no Name)
- Each Function has **Signature** which defines
  - names and Data Types of Input Parameters (name:String, age:Int) (specifying Data Types is optional)
  - Data Type of Return Value (String) (specifying Data Type is optional)
- Since Functions can be stored inside Constants and Variables (including Input Function Parameters) Signature is used to say what kind of Closure can be stored into a Constant or Variable.
  For instance `var name : Int` says that Variable `name` can store Int Data Type.
  Similarly `var closure : (String, Int) -> (String)` says that Variable `closure` can store Closure Data Type of specified Signature - only Closures which accept String & Int as Input Parameters and return String.
- When you store Function inside Constant or Variable you can use them to call Function **(without Named Parameters)**.
  Parameter Names are only for internal use inside Function Body since Anonymous Functions can't have Named Parameters when called.
- Inside lambda we can only use qualified **return** syntax `return@filter`.
  Otherwise, the value of the last expression is implicitly returned.
- Content
  - Syntax
  - Lambda Expressions as Function Parameter
  - Lambda Expressions as Return Value

Lambda Expression

```
  (String    ,     Int) -> (String)                    //SIGNATURE OF ANONYMOUS FUNCTION
{ name:String, age:Int  -> "$name is $age years old." } //DECLARATION & IMPLEMENTATION OF ANONYMOUS FUNCTION
{ name       , age      -> "$name is $age years old." } //DECLARATION & IMPLEMENTATION OF ANONYMOUS FUNCTION
greet("John"     , 50)                                  //Call with Indexed Parameters
```

Syntax

- Lambda Expression is given
  - inside curly brackets { ... } { ... }
  - starting by Function's Signature (name:String, age:Int) -> (accepts 2 Parameters)
  - followed by Function's Statements print("Hello"); "$name is $age years old."

Closure Syntax

```
//DECLARE VARIABLE THAT CAN HOLD FUNCTION DATA TYPE (OF SPECIFIED SIGNATURE).
var closureVariable : (String, Int) -> (String)        //Closure that accepts 2 Parameters & returns String

//DECLARE ANONYMOUS FUNCTION. ASSIGN FUNCTION TO VARIABLE.
closureVariable = { name:String, age:Int -> print("Hello"); "$name is $age years old." }
closureVariable = { name       , age     -> "$name is $age years old."                 }

//CALL FUNCTION. STORE RETURN VALUE INTO VARIABLE.
var result : String = closureVariable("John", 20)      //Closures don't have Named Parameters.

//DISPLAY RESULT.
print(result)
```

Lambda Expressions as Function Parameter

- In this example Function executeFunction() accepts Function as Input Parameter. Function Signature is in red.
- We will call this function twice
  - first we use Lambda Expression to store Anonymous Function into Variable and use this Variable as Parameter
  - then we skip the Variable part and use Lambda Expression directly in function call which stores declared Anonymous Function directly into Function Input Parameter receivedFunction.

Lambda Expressions as Function Parameter

```
//=====================================================================================================================
// FUNCTION: executeFunction
//=====================================================================================================================
fun executeFunction(receivedFunction: (name: String, age: Int) -> String) {
  var result = receivedFunction("John", 20)
  println(result)
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {

  //DECLARE LOCAL VARIABLES.
  var weight = 150

  //DECLARE VARIABLE TO HOLD FUNCTION DATA TYPE.
  var anonymousFunction : (name: String, age: Int) -> String

  //DECLARE ANONYMOUS FUNCTION USING LAMBDA EXPRESSION.
  anonymousFunction = { name: String, age: Int -> "$name is $age years old and $weight kg." }

  //CALL FUNCTION WITH FUNCTION AS PARAMETER (SENDS CLOSURE WITH weight Property).
  executeFunction(anonymousFunction)

  //USE LAMBDA EXPRESSION DIRECTLY AS PARAMETER.
  executeFunction({ name: String, age: Int -> "$name IS $age YEARS OLD AND $weight KG." })

}
```

Lambda Expressions as Return Value

- Body of the getFunction() can be replaced with just the Green part where in the single line
  - we use Lambda Expression to declare Anonymous Function
  - return Closure with
    - this Anonymous Function as its Method
    - Property **weight** which is declared outside of the Scope of Anonymous Function

Lambda Expressions as Return Value

```
//=======================================================================================================================
// FUNCTION: getFunction
//=======================================================================================================================
fun getFunction (weight: Int) : (name: String, age: Int) -> String {

  //DECLARE VARIABLE TO HOLD FUNCTION DATA TYPE.
  var anonymousFunction : (name: String, age: Int) -> String

  //DECLARE ANONYMOUS FUNCTION USING LAMBDA EXPRESSION.
  anonymousFunction = { name: String, age: Int -> "$name is $age years old and $weight kg." }

  //RETURN CLOSURE WITH ANONYMOUS FUNCTION.
  return anonymousFunction

  //USE LAMBDA EXPRESSION DIRECTLY AS RETURN VALUE.
  return { name: String, age: Int -> "$name is $age years old and $weight kg." }

}

//=======================================================================================================================
// FUNCTION: main
//=======================================================================================================================
fun main() {

  //DECLARE VARIABLE TO HOLD FUNCTION DATA TYPE.
  var returnedFunction : (name: String, age: Int) -> String

  //DECLARE ANONYMOUS FUNCTION USING LAMBDA EXPRESSION.
  returnedFunction = getFunction(150)            //Returns Closure with partialy initialized Function

  //CALL RETURNED FUNCTION.
  var result = returnedFunction("John", 20)      //Call Function with missing Parameters

  //USE LAMBDA EXPRESSION AS PARAMETER.
  print(result)

}
```

2.5.4 Inline Functions

Info [R] [R]

- Higher-order functions either (all other functions are first-order functions)
  - accept function as argument
  - return a function
- Higher order functions are stored as Objects (which uses memory).
  When declaring Function, use **inline** keyword to tell compiler to simply copy/inject function code at the calling place.
  Inline functions must be named and can't be declared inside another function (only as Methods or top level Functions).
- If inline Function has Input Parameter that accepts function, then provided Lambda expression can have basic **return**.
  This will exit the function inside which inlined function was called (because it was copied inside it).
  Such Lambda expression needs to be given at the place of Input Parameter so that Compiler knows that Lambda Expression will be used to call Inline Function and that it can therefore support simple return.
  You can't store such Lambda Expression into a Variable if it has simple return.

Lambda Expression with simple return

```
//=====================================================================================================================
// FUNCTION: executeFunction
//=====================================================================================================================
inline fun executeFunction(receivedFunction: (name: String, age: Int) -> Unit)  {
  var result = receivedFunction("John", 20)                                          //return
  println("Not executed")
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {
  executeFunction({ name: String, age: Int -> print("$name IS $age YEARS OLD."); return })  //return
  println("Not executed")
}
```

Output

```
John IS 20 YEARS OLD.
```

2.5.5 Function vs Lambda vs Closure

Function vs Lambda vs Closure

- **Function** is
  - block of code
  - that accepts Input Parameters
  - and returns a single value
- To **Declare a Function** means to
  - specify block of code that should be executed when Function is called
  - specify Names and Data Types of Input Parameters (Names or indexes are used to reference Parameters)
  - specify Data Type of Return Value (or it is inferred by return value)
- In Kotlin Function can be declared using following **Programming Syntaxes**
  - **Lambda Expression** declares **Anonymous Function** (Function that has no Name)
  - **Anonymous Function Declaration** declares **Anonymous Function** (Function that has no Name)
  - **Named Function Declaration** declares **Named Function** (Function that has a Name)
- **Closure** is Object that has
  - a single Method
  - Properties which are all referenced from that Method
- **Function Declaration** is used to create Class & Object Methods and blocks of code that need to be called multiple times.
- **Lambda Expression** is used to declare Function that is used as Input Parameter or Return Value of another Function.
- **Closure** is created
  - when Function needs to return a Function (together with values of referenced Variables from the enclosing Scope)
  - when Function is used as Input Parameter to another Function

Lambda Expressions

- **Lambda Expression** is Programming Syntax that allows us to
  - declare a block of code
  - that accepts Input Parameters
  - and returns a single value
- To be able to **call** and execute that block of code we need to
  - store Lambda Expression into a Variable (or Constant or Function Input Parameter)
  - use that Variable's Name to call that code (can't use Named Parameters)
- Unlike Functions, Lambda Expressions can't have Name which is why they are sometimes referred to as **Anonymous Functions**. But since Lambda Expressions can be stored into Variables, and then we can use that Variable's Name to call that Lambda Expression, the difference is negligible (as long as you don't care about Named Parameters during calls).

Lambda Expression

```
{ name:String, age:Int -> "$name is $age years old." }
```

Using Lambda Expressions to store Anonymous Function into Variable (and call it through Variable Name)

```
//DECLARE CLOSURE. STORE IT INTO VARIABLE
var closureVariable = { name:String, age:Int  -> "$name is $age years old." }
var closureVariable = { name       , age      -> "$name is $age years old." }

//CALL CLOSURE THROUGH VARIABLE.
closureVariable ("John", 50)                                                  //Indexed Parameters
```

Named Function Declarations

- **Function declaration** is also a Programming Syntax that allows us to do the same as with Lambda Expression.
  But in additional we can also
  - give the Name to that block of code
  - use that Name to call that block of code
  - use Named Parameters when calling that block of code (depending on the Language but Kotlin supports this)
- If Lambda Expression and Function Declaration have the same **Signature** then you can store Reference to a Function into the same Variable. Now you can call that Function in the same way as you would the Lambda Expression, by using that Variable's Name (but without using Named Parameters).
- You can think of Function as a Wrapper around Lambda Expression allowing you to assign a name to a Lambda Expression and call Lambda Expression with Named Parameters. But when you store Function reference into a Variable then when using this Variable your Function starts behaving the same as an Lambda Expression (you can't have Named Parameters).

Function Declaration

```
//DECLARE FUNCTION.
fun greet (name:String, age:Int = 20) : (String) { return("$name is $age years old.") } //Block      Syntax
fun greet (name:String, age:Int)      : (String) = "$name is $age years old."           //Assignment Syntax

//CALL FUNCTION.
greet(name="John", age=50)                                                              //Named   Parameters
greet("John"     , 50)                                                                  //Indexed Parameters
```

- Such Variables need to be declared to accept **Function Data Type** specific to the Closure we want to store.
  Function Data Type (also known as **Signature**) defines **Data Types** of **Input Parameters** and **Return Value**.
  These can be inferred from the Closure Signature (so you don't need to explicitly specify them in Variable declaration).
  But once Closure is stored into Variable, this Variable is set to accept only this Function Data Type so you can't store in it other Data Types or Functions with different Signatures.
- Variables that accept **Function Data Type** can store **Closure**, **Function** & **Anonymous Function** with the same **Signature**.
  To store Function into Variable you actually have to store a reference to a Function using "::" operator as a prefix before the Function name: `closureVariable = ::greet`. You can now use Variable to call Function. But you can't use Named Parameters anymore because this information is not stored into Variable (it is part of Function Declaration).

Declare Variable that holds specific Function Data Type

```
var closureVariable : (String, Int) -> (String)        //Function that accepts 2 Parameters & returns String
```

Store Function Reference into Variable

```
//DECLARE FUNCTION.
fun greet (name:String, age:Int) : (Unit) { println("$name is $age years old.") }

//DECLARE VARIABLE. (THAT HOLDS SPECIFIC FUNCTION DATA TYPE)
var closureVariable : (String, Int) -> (Unit)

//STORE REFERENCE TO FUNCTION INTO VARIABLE.
closureVariable = ::greet

//CALL FUNCTION THROUGH VARIABLE (NO NAMED PARAMETERS).
closureVariable("John", 20)
```

Closure

- **Closure Object** (or just Closure for short) is a special type of Object that has following basic characteristics
  - it has a single public Method (optional private Methods from other inner Functions)
  - it only has Properties that are referenced from that Method (optional Properties referenced from private Methods)
  - it is created when
    - **returning a Function** from another Function
    - **using Function as Parameter** into another Function
- When Closure is created
  - Function that is being returned or used as parameter is added as a Method to the Closure Object
  - enclosing Function Variables, that are referenced by that Function, are stored as Properties with their current Values
- Such Closure Object can then be passed around and its Method can be freely executed at any time since Closure contains all Variables used by the Method. **Multiple executions change their Values?**
- So Closure Objects are implicitly created to allow this kind of functionality. You could rewrite that Functionality by explicitly creating Class, Object, Properties and Method and explicitly pass it around.
- Method returned inside Closure Object is said to be partially applied since Properties already have some value. To get the final result from the method you need to provide values for its Input Parameter (this is the part that is missing).
- Outer Function behaves exactly like a Class. By calling it you get its instances - Closure Objects that contain
  - Outer Function Variables as its Properties (referenced by either returned or other inner Functions)
  - returned Function as its public Method (so that it can be called from outside)
  - all other Inner Functions as its private Methods (since they might be referenced from the returned public Method)

Function that returns Closures

```
//=============================================================================================================
// FUNCTION: factory
//=============================================================================================================
fun factory (name:String, age:Int) : ( (Int) -> String ) {
    fun greet(weight: Int) : (String) {
      return "$name is $age years old and $weight kg heavy."
    }
    return ::greet
}

//=============================================================================================================
// FUNCTION: main
//=============================================================================================================
fun main() {

//DECLARE VARIABLE. (THAT HOLDS CLOSURE)
var greet : (Int) -> (String)

//GET CLOSURE.
greet = factory("John", 20)  //CLOSURE PROPERTIES: name="John", age=20

//CALL CLOSURE METHOD.
var result = greet(150)

//DISPLAY RESULT.
println(result)

//GET ANOTHER CLOSURE.
var greet2 : (Int) -> (String) = factory("Bill", 50)
println(greet2(100))  //DIFFERENT CLOSURE COPY: name="Bill", age=50
println(result)       //PREVIOUS CLOSURE IS UNCHANGED

}
```

- Here we implement the same behaviour by declaring a class that creates Closure-like Object that only has one Method and Properties referenced by that Method (like Input Parameters to Outer Function/Class Constructor).
  Returned Method looks the same but the way we call it from the Closure if different then when calling from Object.
  Also Variable that stores Closure accepts **Function Data Type** and the one that stores Object accepts **Class Data Type**.
  If outer Function had some additional code you would transfer that inside init{} blocks of the Primary Constructor.

Class that returns Closure-like Objects

```
//=====================================================================================================================
// FUNCTION: factory
//=====================================================================================================================
class Factory (var name:String, var age:Int) {
    fun greet(weight: Int) : (String) {
      return "${name} is $age years old and $weight kg heavy."
    }
}

//=====================================================================================================================
// FUNCTION: main
//=====================================================================================================================
fun main() {

//DECLARE VARIABLE. (THAT HOLDS OBJECT)
var greet : Factory

//GET OBJECT.
greet = Factory("John", 20)  //OBJECT PROPERTIES: name="John", age=20

//CALL OBJECT METHOD.
var result = greet.greet(150)

//DISPLAY RESULT.
println(result)

//GET ANOTHER OBJECT.
var greet2 : Factory = Factory("Bill", 50)
println(greet2.greet(100))  //DIFFERENT OBJECT: name="Bill", age=50
println(result)             //PREVIOUS OBJECT IS UNCHANGED

}
```

Output

```
John is 20 years old and 150 kg heavy.
Bill is 50 years old and 100 kg heavy.
John is 20 years old and 150 kg heavy.
```

Higher-Order Functions and Lambdas [R]

- Kotlin Functions are **First-Class Citizens** which means that they can be
  - assigned to **Variables**
  - passed as **Input** Parameters to other Functions
  - **returned** from other Functions
- **Higher-Order Function** is function that
  - **returns** Function
  - takes Functions as **Input** parameter

3 Summary

Info

- Following chapters present a quick summery of the subjects covered in this book.
- Online Compiler: https://play.kotlinlang.org

3.1 Data Types

Literals

Literals

LITERAL	EXAMPLE	NOTATIONS
Null	`null`	
Boolean	`true, false`	
Character	`'A'`	
Integer	`65`	Decimal: `65` Hexadecimal: `0x41` Binary: `0b100_0001`
Long	`65L`	Decimal: `65L` Hexadecimal: `0x41l` Binary: `0b100_0001L`
Float	`65.2F`	Basic: `65.2F` Scientific: `0.652E2f`
Double	`65.2`	Basic: `65.2` Scientific: `0.652E2`
String	`"Hello ${name} \n"`	

Data Types

- Kotlin doesn't support Array Literal or Data Type. Instead function `arrayOf<T>` is used to create it and `Array<T>` to store it.

Scalar Data Types

TYPE	EXAMPLE	DESCRIPTION	SIZE
Null	`var value : Int? = null`	undefined/unknown value	
Boolean	`var value : Boolean = true`	logical TRUE or FALSE	
Char	`var value : Char = 'A'`	Unicode character	16-bit
Byte	`var value : Byte = 65`	Signed integer number	8-bit two's complement
Short	`var value : Short = 65`	Signed integer number	16-bit two's complement
Int	`var value : Int = 65`	Signed integer number	32-bit two's complement
Long	`var value : Long = 65L`	Signed integer number	64-bit two's complement
Float	`var value : Float = 65.2F`	Real number	32-bit IEEE 754
Double	`var value : Double = 65.2`	Real number	64-bit IEEE 754

Collection Data Types

TYPE	EXAMPLE	DESCRIPTION
String	`var value : String = "Hello ${name} \n"`	Class for storing constant strings.
Array	`var value : Array<Int> = arrayOf<Int>(1, 2, 3, 4, 5)`	Class for storing Objects.

Data Types & Literals

```
//GENERAL
var value1 : Int?    = null                         //Null      Literal
var value2 : Boolean = true                         //Boolean   Literal: true, false
var value3 : Char    = 'A'                          //Character Literal (16-bit)

//NUMBERS
var value4 : Byte    = 65
var value5 : Short   = 65
var value6 : Int     = 65                           //Integer Literal: 0x41  0b100_0001
var value7 : Long    = 65L                          //Long    Literal: 0x41L 0b100_0001L
var value8 : Float   = 65.2F                        //Float   Literal: 0.652E2F
var value9 : Double  = 65.2                        //Double  Literal: 0.652E2

//COLLECTIONS
var value10 : String = "Hello ${name} \n"           //Collection of characters
var value11 : Array<Int> = arrayOf(1, 2, 3, 4, 5)   //Generic Class
```

3.2 Variables

Info

- Scalars hold single value and can be: Constants, Variables, Optionals.

Scalars

```
val value1 : Int  = 65    //CONSTANT          (value)
var value2 : Int  = 65    //VARIABLE          (variable)
```

Optionals

- If Scalar can have null value it is called Optional (indicated by a question mark '?').

Optional Scalars

```
val value3 : Int? = null  //OPTIONAL CONSTANT ('?' indicates that constant might not have value)
var value3 : Int? = null  //OPTIONAL VARIABLE ('?' indicates that variable might not have value)
```

3.3 Basic Syntax

Comments & Print

Comments

```
//Single line comment.

/* Multi line
   comment.  */
```

Print

```
//DECLARE VARIABLES.
var name = "John"
var age  = 20

//DISPLAY VARIABLES.
println("Hello" + name)                    //Hello John
print  ("${name} is $age years old")       //John is 20 years old
```

Statements - Conditional

if

```
var result : String =
if      (name == "John")   "Name is John"                        //Can ommit {} for single line
else if (name == "Bill")   "Name is Bill"                        //Don't use return to return value
else if (name == "Lucy")   "Name is Lucy"
else                    { print("No match"); "No match found" }
```

when

```
var result : String =
when(name) {
  "John"          ->   "Name is John"                            //Can ommit {} for single line
  "Bill", "Lucy" -> { "Name is Bill or Lucy"               }     //It doesn't fall through to subsequent
case
  else            -> { print("No match"); " No match found" }    //Don't use return to return value
}
```

for

```
//ITERATE THROUGH SEQUENCE.
for (i in 1..5       step 2) { println(i) }  // 1 3 5
for (i in 6 downTo 0 step 2) { println(i) }  // 6 4 2 0

//ITERATE THROUGH ARRAY.
val people = arrayOf("John", "Bill", "Lucy")                                  // DECLARE ARRAY
for (person          in people            ) { println(person)             }  // John Bill Lucy
for (index           in people.indices    ) { println(people[index])      }  // John Bill Lucy
for ((index, person) in people.withIndex()) { println("$index = $person") }  // 0 = John, 1 = Bill, 2 = Lucy
```

while

```
var i = 0
while(i < 5) {
  i++
  println(i)
}
```

do

```
var i = 0
do {
  i++
  println(i)
} while(i < 5)
```

Break

```
//BREAK FROM NEAREST LOOP.
for(i in 1..5) {
  if(i==3) { break }                //Break from nearest Loop.
  print(i)                          //1, 2
}

//BREAK FROM LABELED LOOP.
beginLoop@
for(i in 1..5) {
  for(j in 10..15) {
    if(j==13) { break@beginLoop }   //Break from labeled Loop.
    println(j)                      //10, 11, 12
  }
}
```

Continue

```
//CONTINUE WITH NEAREST LOOP.
for(i in 1..5) {
  if(i==3) { continue }             //Continue with nearest Loop.
  print(i)                          //1, 2, 4, 5
}

//CONTINUE WITH LABELED LOOP.
beginLoop@
for(i in 1..3) {
  for(j in 10..15) {
    if(j==13) { continue@beginLoop } //Continue with labeled Loop.
    println(j)                      //(10, 11, 12) (10, 11, 12) (10, 11, 12) (10, 11, 12) (10, 11, 12)
  }
}
```

Return

```
//DECLARE FUNCTION.
fun test() : String {
  return("Hello from test.")        //Return from the nearest enclosing function.
  print("Never executed")           //Skipped line.
}

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