

CS 473 - MDP

Mobile Device Programming

© 2021 Maharishi International University

All course materials are copyright protected by international copyright laws and remain the property of the Maharishi International University. The materials are accessible only for the personal use of students enrolled in this course and only for the duration of the course. Any copying and distributing are not allowed and subject to legal action.



Maharishi International
University

CS 473 - MDP

Mobile Device Programming

MS.CS Program

Department of Computer Science

Renuka Mohanraj, Ph.D.



Maharishi International
University

CS 473 – MDP

Mobile Device Programming

Lesson 1 (a)

Kotlin Fundamentals



Maharishi International
University

Wholeness of this lesson



Kotlin fundamentals provide the essential building blocks for programming, including variables, null safety, strings, loops, and object-oriented concepts such as classes, inheritance, and interfaces.

Science and Technology of Consciousness: Just as Transcendental Consciousness provides the silent, orderly basis for all expressions of life, these foundational principles provide the structured basis for all advanced Kotlin programming.

Agenda



Software Setup

What is Kotlin?

Kotlin Features

main() function

Mutable and Immutable

Kotlin Strings

Looping

Null safety

Class and Objects

Inheritance, Interface and Data class

Download and Install IntelliJ IDE



- <https://www.jetbrains.com/idea/download>

Windows macOS Linux



The Leading IDE for Professional
Development in Java and Kotlin

Download ▼



Install Gradle on Mac



brew install gradle

If you have already
installed it, upgrade it

- brew upgrade gradle

```
bright~$brew upgrade gradle
```

```
...
```

```
bright~$gradle --version
```

```
-----
```

```
Gradle 9.0.0
```

```
-----
```

```
...
```

```
bright~
```



Releases

Here you can find binaries and reference documentation for current and past versions of Gradle. You can find the next [release candidate](#) or a bleeding edge nightly build for the [release](#) and [master](#) branches on their respective pages.

You can [install Gradle](#) through various other tools, or download a ZIP using the links on this page.

Command-line completion scripts for bash and zsh can be downloaded from the [gradle-completion project](#) page.

Getting Started Resources

There are many [Gradle Build Tool trainings](#) available to help you get started quickly. The new free courses are a part of the [DPE University](#) by Gradle, Inc.

The Gradle team offers free [training](#)

🔍 9.0.0

📅 Jul 31, 2025

- Download: [binary-only](#) or [complete \(checksums\)](#)
- [User Manual](#)
- [API Javadoc](#)
- [Kotlin DSL Reference](#)
- [Groovy DSL Reference](#)
- [Release Notes](#)

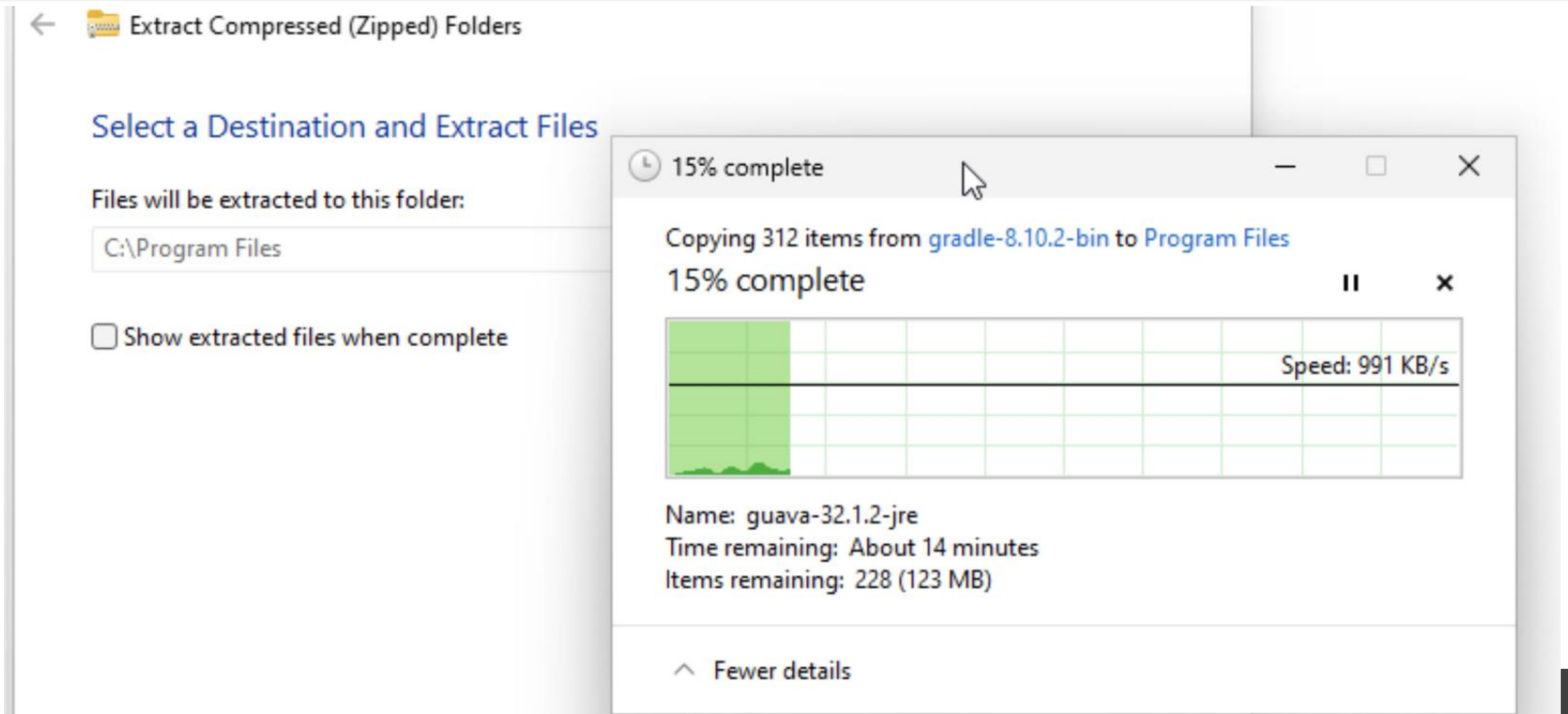


Install Gradle

Windows

<https://gradle.org/releases/>

Download & Extract it to Program Files



Set Environment Variables



The image shows a Windows desktop environment with several windows open, illustrating the process of setting environment variables.

System Properties Window: The "Advanced" tab is selected, showing the "Environment Variables..." button at the bottom.

File Explorer Window: The path is "This PC > Local Disk (C:) > Program Files > gradle-8.10.2 > bin". The file "gradle" is selected.

Environment Variables Window: This window displays the "Environment Variables" for the user "bright".

User variables for bright:

Variable	Value
OneDrive	C:\Users\bright\OneDrive
Path	C:\Users\bright\AppData\Local\Microsoft\
TEMP	C:\Users\bright\AppData\Local\Temp
TMP	C:\Users\bright\AppData\Local\Temp

System variables:

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
MAVEN_HOME	C:\Program Files\apache-maven-3.9.9
NUMBER_OF_PROCESSORS	2
OS	Windows_NT
Path	C:\Program Files\Common Files\Oracle\Java\javapath;C:\Windows...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC

Edit environment variable window: This window is used to edit the "MAVEN_HOME" variable. The value "C:\Program Files\apache-maven-3.9.9\bin" is entered in the text box.

Check the Gradle tool

gradle -v



```
Command Prompt
Microsoft Windows [Version 10.0.22631.4317]
(c) Microsoft Corporation. All rights reserved.

C:\Users\bright>gradle -v

Welcome to Gradle 8.10.2!

Here are the highlights of this release:
- Support for Java 23
- Faster configuration cache
- Better configuration cache reports

For more details see https://docs.gradle.org/8.10.2/release-notes.html

-----
Gradle 8.10.2
-----

Build time:    2024-09-23 21:28:39 UTC
Revision:     415adb9e06a516c44b391edff552fd42139443f7

Kotlin:       1.9.24
Groovy:       3.0.22
Ant:          Apache Ant(TM) version 1.10.14 compiled on August 16 2023
Launcher JVM: 23 (Oracle Corporation 23+37-2369)
Daemon JVM:   C:\Program Files\Java\jdk-23 (no JDK specified, using current Java home)
OS:           Windows 11 10.0 amd64

C:\Users\bright>
```

What is Kotlin?



- Kotlin is a JVM based language developed by JetBrains, a company known for creating IntelliJ IDEA, a powerful IDE for Java and Kotlin development.
- The Android team announced during Google I/O 2017 that Kotlin is an official language to develop Android Apps.
- Android Studio, the official Android IDE, is based on IntelliJ.
- Kotlin was created with Java developers in mind, and with IntelliJ as its main development IDE.
 - Learn more about from <https://developer.android.com/kotlin/index.html>
 - Try online : <https://play.kotlinlang.org/>
 - Find Answers about Kotlin from <https://developer.android.com/kotlin/faq.html>

Kotlin Features



- **Modern and expressive:** You can write more with much less code.
- **It's safer :** Kotlin is null safe, which means that we deal with possible null situations in compile time, to prevent run time exceptions.
- **Functional and object-oriented:** Kotlin is basically an object-oriented language, also gains the benefits of functional programming.
- **Statically typed:** This means the type of every expression in a program is known at compile time, and the compiler can validate that the methods and fields you're trying to access exist on the objects you're using.

Kotlin Features



- **Free and open source:** The Kotlin language, including the compiler, libraries, and all related tooling, is entirely open source and free to use for any purpose.
- **It's highly interoperable:** You can continue using most libraries and code written in Java, because the interoperability between both languages is excellent. It's even possible to create mixed projects, with both Kotlin and Java files coexisting. Easily convert the existing Java code into Kotlin by selecting Code → Convert Java File to Kotlin File
- **Multiplatform Mobile:** The natural way to share code between mobile platforms. <https://kotlinlang.org/docs/multiplatform-mobile-getting-started.html>

Set Kotlin Version



```
1  plugins {  
2      kotlin("jvm") version "2.2.20"  
3  }  
4  
5  group = "com.bright"  
6  version = "1.0-SNAPSHOT"  
7  
8  repositories {  
9      mavenCentral()  
10 }  
11  
12 dependencies {  
13     testImplementation(kotlin("test"))  
14 }  
15  
16 tasks.test {  
17     useJUnitPlatform()  
18 }  
19 kotlin {  
20     jvmToolchain(jdkVersion = 24)  
21 }
```

Ref: <https://kotlinlang.org/docs/releases.html>



Compile and Run

```
1  plugins {  
2      kotlin("jvm") version "2.2.20"  
3      application  
4  }  
5  
6  group = "com.bright"  
7  version = "1.0-SNAPSHOT"  
8  
9  > repositories {...}  
12  
13  > dependencies {...}  
16  
17  > tasks.test {...}  
20  kotlin {  
21      jvmToolchain(jdkVersion = 24)  
22  }  
23  application {  
24      mainClass.set("com.bright.MainKt")  
25  }
```

bright~\$ gradle compileKotlin

BUILD SUCCESSFUL in 9s

bright~\$ gradle run

> Task :run

Hello, Kotlin!

Kotlin Data Types



- Integer Data Types
 - Byte, Short, Int and Long
- Floating Point Data Types
 - Float, Double
- Boolean
 - Accepts true or false
- Character Data Type
 - Char
- String
- All the above data types are actually objects, each of which provides a range of functions and properties that may be used to perform a variety of different type specific tasks. These functions and properties are accessed using so-called dot notation.

main function in Kotlin



This is the main function, which is mandatory in every Kotlin application. The Kotlin compiler starts executing the code from the main function.

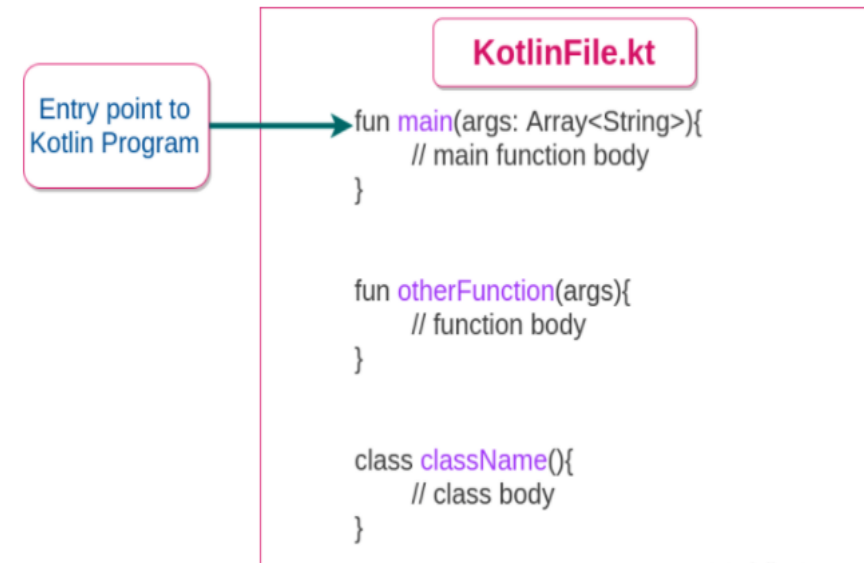
A main() can execute with the args of Array<String> or without arguments will work.

```
fun main() { }
```

```
fun main(args: Array<String>)
```

Replace void without return type or can use keyword Unit

```
fun f(): Unit {  
    println("Nothing return can use Unit similar like Void")  
}  
// If there is no return type mentioned work as void  
fun f1() {  
    println("No return type similar like Void")  
}
```



Mutable and Immutable (Variables and Constants)



- Kotlin is categorized as a statically typed programming language. Uses a technique referred to as *type inference* to identify the type of the variable.
- Mutable/Variable : type is not required.
`var answer = 42` is similar to \rightarrow `var answer: Int = 42`
- Immutable/ Constants :
`val answer = 42` is similar to \rightarrow `val answer: Int = 42`
- Refer : DataTypes.kt, GettingInput.kt

Kotlin Strings



- Strings in Kotlin represent an array of characters. Strings are immutable. It means operations on string produces new strings.

Eg: `val str = "Kotlin Strings"`
`println(str)`

- String templates:** String templates is a powerful feature in Kotlin when a string can contain expression and it will get evaluated.

```
val x = "David"
```

```
val y = "My name is $x"
```

```
println(y)
```

```
println("My name is $x with the length ${x.length}")
```

Output : My name is David

My name is David with the length 5

String operations

```
var s: String = "hello"
```

```
println("Length of string $s is ${s.length}")
```

```
println(
```

```
    "Init cap of string is ${
```

```
        s.replaceFirstChar {
```

```
            if (it.isLowerCase()) it.titlecase()
```

```
            else it.toString()
```

```
        }
```

```
    }"
```

```
)
```

```
println("Lower case is ${s.lowercase()}")
```

```
println("Upper case is ${s.uppercase()}")
```

Output

Length of string hello is 5

Init cap of string is Hello

Lower case is hello

Upper case is HELLO

<https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/index.html>

Kotlin Operators



Type	Operators
Arithmetic Operators	<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>%</code>
Assignment Operators	<code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code>
Increment and decrement operators	<code>++</code> <code>--</code>
Comparison and Equality Operators	<code>></code> <code><</code> <code>>=</code> <code><=</code> <code>==</code> <code>!=</code>
Logical Operators	<code>&&</code> <code> </code>
Sign operators	<code>+</code> and <code>-</code> . They are used to indicate or change the sign of a value.
Range Operator	<code>..</code> (Eg: <code>x..y</code> \rightarrow the range of numbers starting at <code>x</code> and ending at <code>y</code> .)

Kotlin loops for, forEach, repeat, while, do-while



for: for loop iterates through anything that provides an iterator.

Syntax : `for (item in collection) { body }`

Example : `var daysOfWeek =`

`listOf("Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday")`

Note : `listOf()` method returns a read-only list, you cannot add elements.

```
for(day in daysOfWeek){  
    println(day)  
}
```

```
daysOfWeek.forEach{  
    println(it)  
}
```

while and do..while work as usual in Java
Refer : forloop.kt

for with index :

```
for ((index, value) in daysOfWeek.withIndex()) {  
    println("the element at $index is $value")  
}
```

Mutable and Immutable list



```
fun main(args: Array<String>) {  
    // Read only List  
    val myListOfNames = listOf("James", "Paul", "Rafael", "Gina")  
  
    // Read and write mutable list  
    val myMutableList = mutableListOf(12, 34, 45, 123)  
    myMutableList.add(25)  
    myMutableList[0]=100  
    println("Number of elements ${myMutableList.size}")  
    println("Second element ${myMutableList[1]}")  
    println("Index of element \"45\" ${myMutableList.indexOf(45)}")  
  
    for (item in myMutableList) {  
        println(item)  
    }  
    myListOfNames.forEach {  
        println(it)  
    }  
}
```

Output:

Number of elements 5

Second element 34

Index of element "45" 2

100

34

45

123

25

James

Paul

Rafael

Gina

Kotlin - Ranges, when expression



Ranges

- You can create a range in Kotlin via `..` operator.

- Example

```
for(i in 1..5) { print(i) } // Result: 1 2 3 4 5
```

- **downTo** : Using `downTo` function we can go reverse in a range.

```
for(i in 5 downTo 1) { print(i) } // Result: 5 4 3 2 1
```

- **step** : Using `step` function we can increase the step

```
for (i in 1..10 step 2) { print(i) } // Result: 1 3 5 7 9
```

if we want to exclude the last value in a range use `until`

```
for (i in 1 until 5) { print(i) } // Result: 1 2 3 4
```


Kotlin - Ranges, when expression



When expression

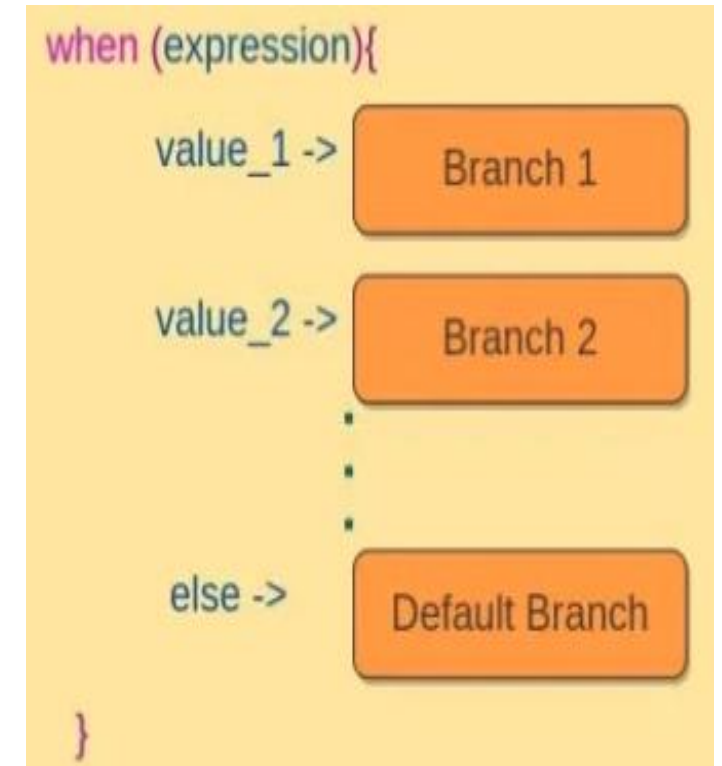
- Kotlin when expression is kind of switch case in Java, but concise in syntax, extended in functionality and more fun. Using “Any” object type with **when** expression makes is really broad in the usage.

Example

```
when (x) {  
    1 -> print("x == 1")  
    2 -> print("x == 2")  
    else -> { // Note the block  
        print("x is neither 1 nor 2")  
    }  
}
```

Refer : WhenExample.kt

<https://kotlinlang.org/docs/control-flow.html#when-expression>



Main Point 1



Lesson covers the fundamentals must know to write Kotlin code. Understand the features, declaring variables, constants and writing methods. The Kotlin language specifies rules for working with null safety. Objects have both data and behavior which support the intended purpose of the object. *Similarly, consciousness is established when the unmanifest value of the object is realized; everything perceived in terms of the Self.*

Kotlin - Null Safety



- Null Safety in Kotlin is to eliminate the risk of occurrence of `NullPointerException` in real time.
- Way to handle Null Safety in Kotlin
 1. Differentiate between nullable references and non-nullable references.
 2. User explicitly checks for a null if conditions
 3. Using a Safe Call Operator (`?.`)
 4. Elvis Operator (`?:`)

Way to handle Null Safety in Kotlin

- **Way – 1** - Differentiate between nullable references and non-nullable references.
 - Kotlin's type system can differentiate between nullable references and non-nullable references. `?` operator is used during variable declaration for the differentiation.

//Non- nullable – You cannot assign null to //the non-null variable

```
var a: String = "Hello"
```

```
a = null // compilation error
```

// If you want to assign null value use ? operator

```
var a: String? = "Hello"
```

```
a = null // OK
```

Way to handle Null Safety in Kotlin

■ Way – 2 – Nullable Check

- The Kotlin system will tell you an error if you want to call a method from a nullable variable. Always check before accessing whether it is null or not.

```
var a: String? = "Hello"
```

```
val l = if (a != null) a.length else -1
```

Way to handle Null Safety in Kotlin

■ Way – 3 - Safe Calls (?.)

- The safe call operator returns the variables property only if the variable is not null, else it returns null. So, the variable holding the return value should be declared nullable.

```
fun main(args: Array){  
    var b: String? = "Hi !" // variable is declared as nullable  
    var len: Int?  
    len = b?.length  
    println("b is : $b")  
    println("length is : $len")  
    b = null  
    len = b?.length  
    println("b is : $b")  
    println("length is : $len")  
}
```

Result for the Code

```
b is : Hi !  
length is : 4  
b is : null  
length is : null
```

Way to handle Null Safety in Kotlin

- Way – 4 – Elvis Operator (?:)
 - If reference to a variable is not null, use the value, else use some default value that is not null. This might sound same as explicitly checking for a null value.

```
fun main(args: Array){  
    var b: String? = " David" // variable is declared as nullable  
    val len = b?.length ?: -1  
    println("length is : $len")  
    b= null  
    val noname = b?.length ?:"No one knows me"  
    println("Name is : $noname")  
}
```

Result for the Code

length is : 5

Name is : No one knows me

The !! Operator (not-null assertion operator)



- Despite the safety measures Kotlin provides to handle NPE, if you need NPE so badly to include in the code use !! operator.
- You can use this operator, if you are 100% sure that variable holds a non null value, or else you will get
NullPointerException(NPE)

The !! Operator (not-null assertion operator)



```
fun main(args: Array){  
    var b: String? = "Hello"    // variable is declared  
    as nullable  
    var blen = b!!.length  
    println("b is : $b")  
    println("b length is : $blen")  
    b = null  
    println("b is : $b")  
    blen = b!!.length // Throws NullPointerException  
    println("b length is : $blen")  
}
```

Result for the code

b is : Hello

b length is : 5

b is : null

Exception in thread "main"

kotlin.KotlinNullPointerException at
ArrayaddremoveKt.main
(Arrayaddremove.kt:9)

Arrays



//boxed Int objects-slower-more memory

```
val array: Array<Int> = arrayOf(1,2,3)
```

//primitive array-faster-less memory

```
val array2: IntArray = intArrayOf(1,2,3)
```

```
println("array has ${array.size} elements")
```

```
println("array2 has ${array2.size} elements")
```

```
println("First element of array is ${array[0]}")
```

```
println("First element of array2 is ${array2[0]}")
```

```
array.forEach { print("$it ") };println()
```

```
array2.forEach { print("$it ") }
```

Output

array has 3 elements

array2 has 3 elements

First element of array is

1

First element of array2 is

1

1 2 3

1 2 3

Arrays - Primitive

```
fun main() {  
    val scores: ByteArray = byteArrayOf(90, 91, 93)  
    val alphabets: CharArray = charArrayOf('a', 'b', 'c', 'd')  
    val rank: ShortArray = shortArrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
    val numbers: IntArray = intArrayOf(1, 2, 3, 4, 5)  
    val mountainHeights: LongArray = longArrayOf(202, 10456)  
    val waterLevel: FloatArray = floatArrayOf(1.0f, 2.0f, 3.0f)  
    val climate: DoubleArray = doubleArrayOf(70.2, 80.3)  
  
    println(scores.contentToString())  
    println(alphabets.contentToString())  
    println(rank.contentToString())  
    println(numbers.contentToString())  
    println(mountainHeights.contentToString())  
    println(waterLevel.contentToString())  
    println(climate.contentToString())  
}
```

```
[90, 91, 93]  
[a, b, c, d]  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
[1, 2, 3, 4, 5]  
[202, 10456]  
[1.0, 2.0, 3.0]  
[70.2, 80.3]
```

String Array



```
val names: Array<String> = arrayOf("Tom","Jerry","Bob")
```

```
names.forEach { print("$it ") }
```

Tom Jerry Bob

ArrayList



```
val names: ArrayList<String> = arrayListOf("Tom", "Jerry", "Mickey")  
  
names.add("Bob")  
  
println(names)  
  
println(names.remove("Tom"))  
  
// names.clear()  
  
println(names.firstOrNull())  
  
println(names.firstOrNull { it.endsWith("ry", true) })
```

```
[Tom, Jerry, Mickey, Bob]  
  
true  
  
Jerry  
  
Jerry
```

ArrayList: Why firstOrNull?



```
val names: ArrayList<String> = arrayListOf("Tom", "Jerry", "Mickey")  
names.clear()  
println(names.first())
```

Exception in thread "main" java.util.NoSuchElementException: List is empty.

ArrayList: Why firstOrNull?



```
val names: ArrayList<String> = arrayListOf("Tom", "Jerry", "Mickey")
```

```
names.clear()
```

```
// println(names.first())
```

```
println(names.firstOrNull())
```



null

ArrayList



```
val cartoonChars: ArrayList<String> = arrayListOf("Tom", "Jerry", "Mickey")
```

```
cartoonChars.add("Donald")
```

```
println(cartoonChars)
```

```
val plants: ArrayList<String> = ArrayList(listOf("Basil", "Cilantro", "Lavender"))
```

```
plants.remove("Lavender")
```

```
plants.add("Rosemary")
```

```
println(plants)
```

[Tom, Jerry, Mickey, Donald]

[Basil, Cilantro, Rosemary]

Define a Function



```
fun name () {  
    body  
}
```

Ref: <https://developer.android.com/codelabs/basic-android-kotlin-compose-functions#1>

Return a value from a function



```
fun name ( ) : return type {  
    body  
    return statement  
}
```

Ref: <https://developer.android.com/codelabs/basic-android-kotlin-compose-functions#2>

Add a parameter to the function



```
fun name ( parameters ) : return type {  
    body  
}
```

Ref: <https://developer.android.com/codelabs/basic-android-kotlin-compose-functions#1>

Declaring own Functions



Function name

Parameters

Return type

```
fun max(a: Int, b: Int): Int {  
  return if (a > b) a else b  
}
```

Function body

Example 1 – functions contain other functions



```
fun greetUser(name: String): String {
```

```
    val greeting = "Hello"
```

```
    fun buildMessage(): String {  
        return "$greeting, $name!"  
    }
```

```
    return buildMessage()
```

```
}
```

```
fun main() {
```

```
    println(greetUser("Kotlin"))
```

```
}
```

The local function can access name and greeting from its enclosing function

Example 2- Variable Number of Function Parameters



Kotlin handles this possibility using the `vararg` keyword to indicate that the function accepts an arbitrary number of parameters of a specified data type.

```
fun greetAll(vararg names: String, greeting: String) {  
    //...  
}  
  
fun greetAllWithPrefix(prefix: String = "Greetings", vararg names: String, ) {  
    //...  
}
```

```
fun greetAll(vararg names: String, greeting: String) {  
    for (name in names) {  
        println("$greeting, $name!")  
    }  
}  
  
fun greetAllWithPrefix(prefix: String = "Greetings", vararg names: String) {  
    for (name in names) {  
        println("$prefix, $name!")  
    }  
}  
  
fun main() {  
    greetAll("Kotlin", "Java", "Go", greeting = "Hello")  
    greetAllWithPrefix("Hey", "Kotlin", "Java", "Go")  
}
```

Hello, Kotlin!
Hello, Java!
Hello, Go!
Hey, Kotlin!
Hey, Java!
Hey, Go!

```
fun greetAll(vararg names: String, greeting: String) {  
    for (name in names) {  
        println("$greeting, $name!")  
    }  
}  
  
fun main() {  
    greetAll(names = arrayOf("Kotlin", "Java", "Go"), greeting = "Hello")  
}
```

Hello, Kotlin!

Hello, Java!

Hello, Go!

Example 3-Declaring Default Function Parameters

Kotlin provides the ability to designate a default parameter value to be used if the value is not provided as an argument when the function is called.

If parameter names are not used within the function call, however, only the trailing arguments may be omitted:

```
fun main() {
```

```
    details("Bright", gender = "Male", age = 40) //valid
```

```
    details(20) //invalid
```

```
    details(age = 20) //valid
```

```
}
```

```
fun details(name: String = "Unknown", age: Int, gender: String = "Male") {  
    println("Name: $name, Age: $age and Gender: $gender")  
}
```

Example 4 :Single Expression Function



- When a function contains single expression, it is not necessary to include the braces around the expression. All that is required is an equal sign (=) after the function declaration followed by the expression.

- Way 1

```
fun multiply(x: Int, y: Int): Int {    return x * y }
```

- Way 2

```
fun multiply(x: Int, y: Int): Int = x * y
```

- Way 3 – no need to specify return type

```
fun multiply(x: Int, y: Int) = x * y
```

```
fun main() {  
    println(sumOfNumbers(1,2,3,4,5))  
}
```

```
//fun sumOfNumbers(vararg numbers: Int): Int { return numbers.sum() }
```

```
fun sumOfNumbers(vararg numbers: Int): Int = numbers.sum()
```

OR

```
fun sumOfNumbers(vararg numbers: Int) = numbers.sum()
```

} Return type is optional

Store Function in a Variable



```
fun add(a: Int, b: Int) = a + b
```

```
fun main() {
```

```
    val result = add
```

Function invocation 'add(...)' expected.

```
    //...
```

```
}
```

Store Function in a Variable



To refer to a function as a value, you need to use the function reference operator (::).

:: function name

```
fun add(a: Int, b: Int) = a + b
```

```
fun main() {
```

```
    val result = ::add
```

```
    println(result(1,2))
```

```
}
```

Redefine Function with a Lambda Expression



Lambda expressions provide a concise syntax to define a function without the `fun` keyword.



We can store a lambda expression directly in a variable without a function reference on another function.

```
val variable name = {  
    function body  
}
```

Redefine Function with a Lambda Expression



```
//fun add(a: Int, b: Int) = a + b
```

```
val add = {a: Int, b: Int ->  
    a + b  
}  
  
fun main() {  
    val result = add  
    println(result(1,2))  
}
```

```
val add = {a: Int, b: Int ->  
    a + b  
}  
  
fun main() {  
    println(add(1,2))  
}
```

```
val add: (Int, Int)-> Int = {a, b ->  
    a + b  
}  
  
fun main() {  
    println(add(1,2))  
}
```

it: Implicit Name of a Single Parameter



- It's very common for a lambda expression to have only one parameter.
- If the compiler can parse the signature without any parameters, the parameter does not need to be declared and `->` can be omitted. The parameter will be implicitly declared under the name **it**

```
val square: (Int) -> Int = {data ->  
  data * data  
}
```



```
val square: (Int) -> Int = { it * it }
```


Main Point 2



- Kotlin is a powerful object oriented and functional programming language features that is easier to use, faster development and more secured. *Science of Consciousness: Transcendental Meditation is an easy and effortless technique to make a mind clear and more powerful so that we can make life easier.*

Quiz



1. Kotlin Strings are mutable.
a) True b) False ✓
2. Kotlin data types are object type.
a) True ✓ b) False

Kotlin Modifiers



- Classes, objects, interfaces, constructors, functions, properties and their setters can have visibility modifiers. There are four visibility modifiers in Kotlin: private, protected, internal and public.
- **public: default**
 - The public modifiers means that the declarations are visible everywhere.
 - In Kotlin the default visibility modifier is public
- **internal (Modules not covered in this course)**
 - internal means that the declarations are visible inside a module. A module in Kotlin is a set of Kotlin files compiled together.
- **private**
 - With private declarations are only visible in the class
- **protected**
 - Declarations are only visible in its class and in its sub classes

Declaring a Kotlin Class



- Kotlin provides extensive support for developing object-oriented applications.
- The basic syntax for a new class is as follows:

```
class ClassName {  
    //properties  
    //methods  
}
```

Adding Properties and Method



```
class BankAccount {  
    var accountNumber: String = ""  
    var balance: Double = 0.0  
  
    fun displayBalance() {  
        println("Account Number: $accountNumber")  
        println("Balance: $balance")  
    }  
}
```

```
fun main() {  
    val account = BankAccount()  
    account.accountNumber = "123456789"  
    account.balance = 100.0  
    account.displayBalance()  
}
```

Output

Account Number: 123456789

Balance: 100.0

Constructors



- A class in Kotlin has
 - a primary constructor and
 - possibly one or more secondary constructors.

Primary Constructor

Class

- Primary Constructor
- Secondary Constructor(s)

- The primary constructor is declared in the class header, and it goes after the class name and optional type parameters.

```
class BankAccount (  
    var accountNumber: String = "",  
    var balance: Double = 0.0  
) {  
    fun displayBalance() {  
        println("Account Number: $accountNumber")  
        println("Balance: $balance")  
    }  
}
```

Note:

In Kotlin, if *all* constructor parameters have default values, the compiler automatically provides a **no-argument constructor** behind the scenes.



Primary Constructor

Class

- Primary Constructor
- Secondary Constructor(s)

- Plain constructor parameters (that are not properties) are accessible in:

- The class header.
- Initialized properties within the class body.
- Initializer blocks.

```
class BankAccount (  
    accountNumber: String = "",  
    openingBonus: Double = 50.0,  
    balance: Double = openingBonus  
) {  
    val interest = 0.01 * balance  
    init {  
        println("Account Number: $accountNumber")  
        println("Balance: $balance")  
    }  
    fun displayBalance() {  
        println("Account Number: $accountNumber")  
        println("Balance: $balance")  
    }  
}
```


Class

- Primary Constructor
- Secondary Constructor(s)

```
class Account { 1 Usage
    accountNo: String = "",
    balance: Double = 0.0
} {
    val accountNo: String = maskAccountNo(accountNo) 1 Usage
    var balance: Double = balance 2 Usages
    init {...}
    fun maskAccountNo(accountNo: String): String {...}
    fun display() { 1 Usage
        println("Account No: $accountNo, Balance: $balance")
    }
}

fun main() {
    val account = Account(accountNo = "1234", balance = 100.0)
    account.balance += 100.0
    account.display()
}
```



Constructors

Class

- Primary Constructor
- Secondary Constructor(s)

- There could be only one **Primary constructor** for a class in Kotlin.
- The primary constructor comes right after the class name in the header part of the class.
- Constructors that are written inside the Body of Class are called **Secondary constructors**.
- Secondary Constructor should call primary constructor or other secondary constructors using **this** keyword.

```
class BankAccount (  
    var accountNumber: String,  
    var balance: Double  
) {  
    constructor(accountNumber: String) : this(accountNumber, 0.0) {  
        println("Secondary constructor called")  
    }  
    constructor(): this("0000") {  
        println("Default constructor called")  
    }  
  
    constructor(balance: Double): this("0000") {  
        this.balance = balance  
    }  
  
    override fun toString() = "$accountNumber, $balance"  
}
```

Secondary Constructors

Note

- Every secondary constructor **must delegate** to the primary constructor (directly or indirectly).
- Ensures the object is **fully initialized** through the primary constructor.

```
class BankAccount (  
    var accountNumber: String,  
    var balance: Double  
) {  
    constructor(accountNumber: String) : this(accountNumber, 0.0)  
    constructor(): this("0000")  
  
    constructor(balance: Double): this("0000") {  
        this.balance = balance  
    }  
  
    override fun toString() = "$accountNumber, $balance"  
}
```

Output

```
0000, 0.0  
1234, 0.0  
5678, 100.0  
0000, 100.0
```

```
fun main() {  
    val account1 = BankAccount()  
    println(account1)  
    val account2 = BankAccount("1234")  
    println(account2)  
    val account3 = BankAccount("5678", 100.0)  
    println(account3)  
    val account4 = BankAccount(100.0)  
    println(account4)  
}
```

Custom Accessors



- By default, Kotlin automatically generates getters and setters.
- We can define our own custom accessors when we need extra logic, such as validation, formatting, or calculations based on other properties.

```
class Point(var x: Int, var y: Int) {  
    var coordinates: String
```

```
    get() {  
        return "Coordinates: ($x, $y)"  
    }  
    set(value) {  
        val parts = value.split(",")  
        if (parts.size == 2) {  
            x = parts[0].trim().toInt()  
            y = parts[1].trim().toInt()  
        } else {  
            throw IllegalArgumentException("Invalid coordinate format")  
        }  
    }  
}
```

Custom getter and setter for
coordinates

```
val distanceFromOrigin: Double
```

```
    get() = sqrt((x * x + y * y).toDouble())
```


- Custom getter for distanceFromOrigin.
- Since its val, setter is not allowed

```
}
```

```

class Point(var x: Int, var y: Int) {
    var coordinates: String
    get() {
        return "Coordinates: ($x, $y)"
    }
    set(value) {
        val parts = value.split(",")
        if (parts.size == 2) {
            x = parts[0].trim().toInt()
            y = parts[1].trim().toInt()
        } else {
            throw IllegalArgumentException("Invalid coordinate format")
        }
    }
    val distanceFromOrigin: Double
    get() = sqrt((x * x + y * y).toDouble())
}

```



```

fun main() {
    val point = Point(10, 20)
    println(point.coordinates)
    point.coordinates = "100, 200"
    println(point.coordinates)
    println(
        DecimalFormat("#.##")
            .format(point.distanceFromOrigin)
    )
}

```

Output

```

Coordinates: (10, 20)
Coordinates: (100, 200)
223.61

```

Changing Visibility



```
class BankAccount ( val accountNumber: String, initialBalance: Double) {
```

```
    var balance = initialBalance  
    // Only the class can modify the balance  
    private set
```

```
    var customerName: String = "Unknown"  
    set(value) {  
        //do validation logic  
        if (value.isNotEmpty())  
            field = value  
    }  
    get() {  
        return field.replaceFirstChar { it.uppercaseChar() }  
    }  
    //...  
}
```

```
fun main() {  
    val account = BankAccount("1234", 100.0)  
    account.balance = 200.0 //error
```


Changing Visibility



```
class Account( 1 Usage
    var accountNo: String = "",
    var balance: Double = 0.0
) {
    var customerName: String = "Unknown" 1 Usage
    private set(value) {
        field = value.uppercase()
    }
    private get() {
        return field.uppercase()
    }

    fun display() {
        println(
            "Name: $ customerName \nAccount No: $accountNo, Balance: $balance"
        )
    }
}
```



Getter visibility must be the same as property visibility.

Backing Field



- In Kotlin, accessors use backing fields to store the property's value in memory.
- Backing fields are useful when you want to add extra logic to a getter or setter, or when you want to trigger an additional action whenever the property changes.
- We cannot declare backing fields directly.
- Kotlin generates them only when necessary.
- We can reference the backing field in accessors using the **field** keyword.

Backing Field



```
class BankAccount ( val accountNumber: String, initialBalance: Double) {  
    // ...  
    var customerName: String = "Unknown"  
    set(value) {  
        if (value.isNotEmpty())  
            field = value  
        println("Customer name is $field")  
    }  
    get() {  
        return field.replaceFirstChar { it.uppercaseChar() }  
    }  
  
    // ...  
}
```

Instead of \$field, if we use \$customerName, it triggers getter method

Inheritance



All classes in Kotlin have a common superclass, **Any**, which is the default superclass for a class with no supertypes declared:

```
class Person // Implicitly inherits from Any
```



```
class Person: Any()
```

- Any has three methods:
 - equals()
 - hashCode()
 - toString()
- Thus, these methods are defined for all Kotlin classes.

Inheritance



- By default, Kotlin classes are final – they cannot be inherited.
- To make a class inheritable, mark it with the open keyword.

open class Person

Parent class with Primary Constructor

- If the derived class has a primary constructor, the base class can (and must) be initialized in that primary constructor according to its parameters.

```
open class Person(val name: String) //{}
```

```
class Student(name: String, val registerNo: String): Person(name) {  
    override fun toString() = "$name, $registerNo"  
}
```

OR

```
class Student(val studentName: String, val registerNo: String): Person(studentName) { /* ... */ }
```

Parent class with Primary & Secondary Constructors



```
open class Person(val name: String){  
    protected var gender: Char? = null  
    constructor(name: String, gender: Char): this(name) {  
        this.gender = gender  
    }  
}
```

```
class Student(name: String, gender: Char, val registerNo: String): Person(name, gender) {  
    override fun toString() = "$name, $registerNo, $gender"  
}
```

```
fun main() {  
    val student = Student("John", 'M',"1234567")  
    println(student)  
}
```

Parent Class with Secondary Constructor



```
open class Person{
    var name: String
    private set
    get() = field.replaceFirstChar { it.uppercase() }
    protected var gender: Char
    constructor(name: String, gender: Char) {
        this.name = name
        this.gender = gender
    }
}

class Student(name: String, gender: Char, val registerNo: String): Person(name, gender) {
    override fun toString() = "$name, $registerNo, $gender"
}

fun main() {
    val student = Student("John", 'M',"1234567")
    println(student)
}
```


Overriding Methods



- Kotlin requires explicit modifiers for overridable members and overrides.

```
open class Animal {  
    open fun makeSound() = println("The Animal is making a sound")  
}
```

```
class Dog : Animal() {  
    override fun makeSound() {  
        super.makeSound()  
        println("Dog says Woof")  
    }  
}
```

To override a method, declare the method as open.

Interface



- Interfaces in Kotlin can contain declarations of abstract methods, as well as method implementations.
- What makes them different from abstract classes is that interfaces cannot store state.
- They can have properties, but these need to be abstract or provide accessor implementations.

Interface



```
interface MathOperation {  
    val pi: Double //non-abstract property  
        get() = 3.14  
    val firstNumber: Double //abstract property  
    val secondNumber: Double //abstract property  
    fun add(a: Double = firstNumber, b: Double = secondNumber): Double {  
        return a + b  
    }  
    fun subtract(a: Double, b: Double): Double //abstract method  
}
```

```
interface MathOperation {  
    val pi: Double           //non-abstract property  
        get() = 3.14  
    val firstNumber: Double  //abstract property  
    val secondNumber: Double //abstract property  
    fun add(a: Double = firstNumber, b: Double = secondNumber): Double {  
        return a + b  
    }  
    fun subtract(a: Double, b: Double): Double //abstract method  
}
```

```
class Calculator(  
    override val firstNumber: Double,  
    override val secondNumber: Double  
): MathOperation {  
    override fun subtract(a: Double, b: Double): Double = a - b  
}
```

Data Class



- Data classes in Kotlin are primarily used to hold data.
- For each data class, the compiler automatically generates additional member functions that allow you to print an instance to readable output, compare instances, copy instances, and more.
- Data classes are marked with data.

```
data class User(  
    val name: String,  
    val age: Int,  
    val gender: Char  
)
```

Data Class



- The compiler automatically derives the following members from all properties declared in the primary constructor:
 - equals()/hashCode() pair.
 - toString() of the form "User(name=John, age=42)".
 - componentN() functions corresponding to the properties in their order of declaration.
 - copy()

```
data class User(  
    val name: String,  
    val age: Int,  
    val gender: Char  
)
```

```
fun main() {  
    val user = User("John", 25, 'M')  
    println(user)  
    println("${user.component1()} is ${user.component2()} years old")  
    println(user.age)  
    println(user.gender)  
    val user2 = user.copy(age = 32)  
    println(user2)  
    println(user.equals(user2))  
}
```

Output

```
User(name=John, age=25, gender=M)  
John is 25 years old  
25  
M  
User(name=John, age=32, gender=M)  
false
```

1. The primary constructor comes right after the class name in the header part of the class.
a) True b) False
2. If you want Inherit, in Kotlin you have declare the parent class with the keyword _____
a) extends b) open c) Inherit

Main Point 3



- In the OO paradigm of programming, execution of a program involves objects interacting with objects. Each object has a type, which is embodied in a Kotlin *class*. The Kotlin language specifies syntax rules for the coding of classes, and also for how objects are to be created based on their type (class). *By using more and more of the intelligence of Nature, we are able to successfully manage all complexity in life, and live a life of success, harmony and fulfillment.*

UNITY CHART

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Take the right angle and let go

1. Understand the fundamentals before writing the code is important to develop Android applications.
 2. Kotlin provides extensive support for developing object-oriented applications.
-

3. *Transcendental consciousness* is when our mind is in contact with the deepest underlying reality, the unified field.
4. *Impulses within the Transcendental field:* the infinitely dynamism of the unified field constantly expresses itself and becomes all aspects of the universe.
5. *Wholeness moving within itself:* In Unity Consciousness, one experiences that this infinite dynamism is nothing but the self.

