


Kotlin

A decorative horizontal bar composed of various colored segments (black, blue, yellow, light blue, dark blue, teal) is positioned above the text.

Presented by
VAISHALI TAPASWI

FANDS INFONET Pvt.Ltd.

www.fandsindia.com

fands@vsnl.com

Ground Rules

- ❑ **Turn off cell phone. If you cannot, please keep it on silent mode. You can go out and attend your call.**
- ❑ **If you have questions or issues, please let me know immediately.**
- ❑ **Let us be punctual.**

A decorative vertical bar on the left side of the slide, composed of numerous horizontal segments in various shades of blue, black, and yellow, creating a striped effect.

Agenda

Kotlin

- ❑ The most strongly supported JVM language in the Android ecosystem—aside from Java—is **Kotlin**, an open-source, statically-typed language developed by JetBrains.
- ❑ One other huge benefit of **Kotlin** is that most of its language design decisions focused on maintaining backward compatibility with many Java and Android projects.

Kotlin

- Kotlin is a statically typed programming language that runs on the Java virtual machine and also can be compiled to JavaScript source code or use the LLVM compiler infrastructure. It is sponsored and developed by JetBrains.

Kotlin

- ❑ Kotlin came up in 2011 from a team of developers at JetBrains.
- ❑ It's a statically typed language for modern multi-platform applications.
- ❑ Kotlin runs on the Java Virtual Machine and compiles to the JVM Bytecode. It can also be compiled to the Javascript source code. Thus, Kotlin is 100% interoperable with Java. This means that you can call Kotlin code in Java and vice-versa.
- ❑ Interoperability feature is one of the major reasons that's led to the wide adoption of the kotlin programming language.

Kotlin

- Functional Programming
- Reduced
 - Plumbing Code
 - Boiler Code
- Type Inference
 - define the type or set the value

Functional Programming



Functional Programming

- ❑ Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.
- ❑ Two Categories
 - Pure Functional Languages – These types of functional languages support only the functional paradigms. For example – Haskell.
 - Impure Functional Languages – These types of functional languages support the functional paradigms and imperative style programming. For example – LISP.

Characteristics

- ❑ Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- ❑ Functional programming supports higher-order functions and lazy evaluation features.
- ❑ Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- ❑ Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

Advantages

- ❑ Bugs-Free Code – Functional programming does not support state, so there are no side-effect results and we can write error-free codes.
- ❑ Efficient Parallel Programming – Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- ❑ Efficiency – Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- ❑ Supports Nested Functions – Functional programming supports Nested Functions.
- ❑ Lazy Evaluation – Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

Disadvantages

- ❑ As a downside, functional programming requires a large memory space. As it does not have state, you need to create new objects every time to perform actions.
- ❑ Functional Programming is used in situations where we have to perform lots of different operations on the same set of data.
- ❑ Lisp is used for artificial intelligence applications like Machine learning, language processing, Modeling of speech and vision, etc.
- ❑ Embedded Lisp interpreters add programmability to some systems like Emacs.

OOP Vs FP

OOP	Functional Programming
Uses Mutable data.	Uses Immutable data.
Follows Imperative Programming Model.	Follows Declarative Programming Model.
Focus is on "How you are doing"	Focus is on: "What you are doing"
Not suitable for Parallel Programming	Supports Parallel Programming
Its methods can produce serious side effects.	Its functions have no-side effects
Flow control is done using loops and conditional statements.	Flow Control is done using function calls & function calls with recursion
It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java	It uses "Recursion" concept to iterate Collection Data.
Execution order of statements is very important.	Execution order of statements is not so important.
Supports only "Abstraction over Data".	Supports both "Abstraction over Data" and "Abstraction over Behavior".

<https://kotlinlang.org/docs/reference/faq.html>

- Is Kotlin an object-oriented language or a functional one?
 - Kotlin has both object-oriented and functional constructs. You can use it in both OO and FP styles, or mix elements of the two. With first-class support for features such as higher-order functions, function types and lambdas, Kotlin is a great choice if you're doing or exploring functional programming.

Vocabulary

- Arity
- Higher-Order Functions (HOF)
- Closure
- Partial Application
- Currying
- Auto Currying
- Function Composition
- Morphism
- Endomorphism
- Isomorphism

Arity

- The number of arguments a function takes. From words like unary, binary, ternary, etc. This word has the distinction of being composed of two suffixes, "-ary" and "-ity."
- Addition, for example, takes two arguments, and so it is defined as a binary function or a function with an arity of two.
- Likewise, a function that takes a variable number of arguments is called "variadic," whereas a binary function must be given two and only two arguments, currying and partial application notwithstanding (see below).

Higher-Order Functions (HOF)

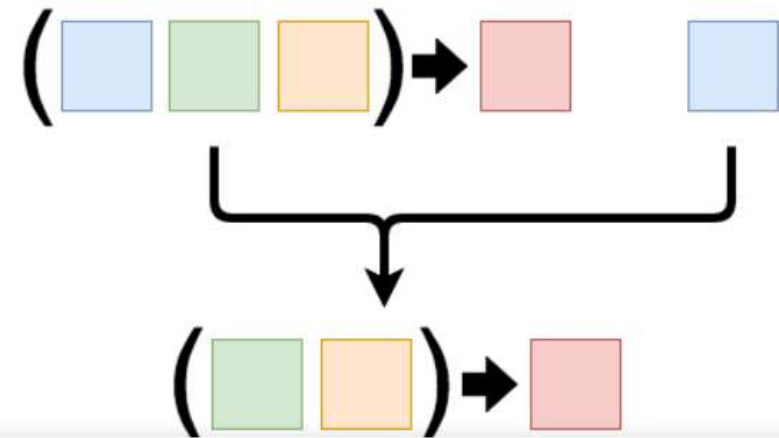
- A function which takes a function as an argument and/or returns a function.
 - `const filter = (predicate, xs) => xs.filter(predicate)`
 - `const is = (type) => (x) => Object(x) instanceof type`
 - `filter(is(Number), [0, '1', 2, null]) // [0, 2]`

Closure

```
const addTo = (x) => {  
  return (y) => {  
    return x + y  
  }  
}
```

- A closure is a scope which retains variables available to a function when it's created. This is important for partial application to work.
- Usage
 - var addToFive = addTo(5)
 - addToFive(3) // => 8

Partial Application



- Is a technique of fixing a number of arguments to a function, producing another function of smaller arguments i.e binding values to one or more of those arguments as the chain of function progressed.

Partial Application

- Partially applying a function means creating a new function by pre-filling some of the arguments to the original function.

```
fun discShm(mymap:Map<String,
Int>):Int{
    return gettotal3(mymap,
discCalc3)
}
```

```
val discCalc3 = { sum: Int,
percent: Int ->.....}
```

```
fun gettotal3(mymap:Map<String,
Int>,disc:(Int,Int)->Double ):Int{
    ....
}
```

Currying

```
const sum = (a, b) => a + b
```

```
const curriedSum = (a) => (b) => a + b
```

```
curriedSum(40)(2) // 42.
```

```
const add2 = curriedSum(2) // (b) => 2 + b
```

```
add2(10) // 12
```

- The process of converting a function that takes multiple arguments into a function that takes them one at a time.
- Each time the function is called it only accepts one argument and returns a function that takes one argument until all arguments are passed.

Auto Currying

```
const add = (x, y) => x + y
```

```
const curriedAdd = _.curry(add)
```

```
curriedAdd(1, 2) // 3
```

```
curriedAdd(1) // (y) => 1 + y
```

```
curriedAdd(1)(2) // 3
```

- Transforming a function that takes multiple arguments into one that if given less than its correct number of arguments returns a function that takes the rest. When the function gets the correct number of arguments it is then evaluated.

Function Composition

- The act of putting two functions together to form a third function where the output of one function is the input of the other.

```
const compose = (f, g) => (a) => f(g(a)) // Definition
const floorAndToString = compose((val) => val.toString(),
Math.floor) // Usage
floorAndToString(121.212121) // '121'
```

Purity

- A function is pure if the return value is only determined by its input values, and does not produce side effects.

```
const greet = (name) =>  
  `Hi, ${name}`
```

```
greet('Brianne') // 'Hi,  
Brianne'
```

```
window.name = 'Brianne'
```

```
const greet = () => `Hi,  
${window.name}`  
greet() // "Hi, Brianne"
```


Morphism, Endomorphism

- Morphism - A transformation function
- Endomorphism - A function where the input type is the same as the output

```
// uppercase :: String -> String
const uppercase = (str) => str.toUpperCase()
// decrement :: Number -> Number
const decrement = (x) => x - 1
```

Isomorphism

- Isomorphism - A pair of transformations between 2 types of objects that is structural in nature and no data is lost.

// Providing functions to convert in both directions makes them isomorphic.

```
const pairToCoords = (pair) => ({ x: pair[0], y: pair[1] })  
const coordsToPair = (coords) => [coords.x, coords.y]  
coordsToPair(pairToCoords([1, 2])) // [1, 2]  
pairToCoords(coordsToPair({ x: 1, y: 2 })) // { x: 1, y: 2 }
```

Language Basics



Getting Started

A decorative horizontal bar consisting of a series of colored segments in shades of blue, teal, yellow, and black, spanning the width of the slide.

Basic Syntax

Defining Packages

- Package specification should be at the top of the source file:
 - package my.demo
 - import java.util.*
- It is not required to match directories and packages: source files can be placed arbitrarily in the file system.
- If the package is not specified, the contents of such a file belong to "default" package that has no name.

Imports

□ Default Imports

kotlin.*
kotlin.annotation.*
kotlin.collections.*
kotlin.comparisons.* (since 1.1)
kotlin.io.*
kotlin.ranges.*
kotlin.sequences.*
kotlin.text.*

□ Additional packages are imported depending on the target platform:

JVM:

java.lang.*
kotlin.jvm.*

JS:

kotlin.js.*

Imports

□ Import Syntax

- We can import either a single name, e.g.
 - `import foo.Bar` // Bar is now accessible without qualification
- All the accessible contents of a scope (package, class, object etc):
 - `import foo.*` // everything in 'foo' becomes accessible
- If there is a name clash, we can disambiguate by using `as` keyword to locally rename the clashing entity:
 - `import foo.Bar` // Bar is accessible
 - `import bar.Bar as bBar` // bBar stands for 'bar.Bar'

Imports

- The import keyword is not restricted to importing classes; you can also use it to import other declarations:
 - top-level functions and properties;
 - functions and properties declared in object declarations;
 - enum constants.
- No import static support (like Java)

Defining functions

- Explicitly defining input and return types

```
fun sum(a: Int, b: Int): Int {
```

- Function with an expression body and inferred return type

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

- Function returning no meaningful value, Unit need not be mentioned

```
fun printSum(a: Int, b: Int): Unit {  
    println("sum of $a and $b is ${a + b}")  
}
```

Defining variables

- Read-only local variables are defined using the keyword `val`. They can be assigned a value only once.

```
val a: Int = 1 // immediate assignment
```

```
val b = 2 // `Int` type is inferred
```

```
val c: Int // Type required when no initializer is provided
```

```
c = 3 // deferred assignment
```

- Variable Values can't be re-assigned.

```
– c = 3 //Will give error
```

Comments

- Just like Java and JavaScript, Kotlin supports end-of-line and block comments.

`// This is an end-of-line comment`

`/* This is a block comment
on multiple lines. */`

Using String Templates

- Simple Name in Template

```
var a = 1
```

```
val s1 = "a is $a"
```

- Arbitrary Expression in Template

```
a = 2
```

```
val s2 = "${s1.replace("is", "was")}, but now is  
$a"
```

const vs val

- ❑ const is like val, except that they are compile-time constants.
- ❑ const are allowed only as a top-level or member of an object

`const val x = "Hello, Kotlin" //would compile`

`val y = printFunction() //works.`

`const val z = printFunction() //won't work. valid for compile time constant only.`

`class A {`

`const val x = "Hello, Kotlin" //won't compile.`

`//const can be only used at top level or in an object.`

Lazy

- Lazy is lazy initialization.
- Val property won't be initialized until you use it for the first time in your code. Henceforth, the same value would be used. The value is present in the lazy() function which takes the value in the form of a lambda.

```
val x: Int by lazy { 10 }
```

Lateinit

- ❑ lateinit modifier is used with var properties and is used to initialize the var property at a later stage. Normally, Kotlin compiler requires you to specify a value to the property during initialization. With the help of lateinit you can delay it.

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

    var a = A()
    a.x = "Hello Kotlin"
}
```

```
class A {
    lateinit var x: String
}
```

Data Types

- Numbers
 - Double – 64, Float – 32(F), Long – 64(L)
 - Int – 32, Short – 16, Byte – 8
 - No type inference for short and byte
- Characters
 - specified in single quotes
- Boolean
- Arrays
- Strings

Operators

- Equality Operators
 - ==, !=
- Comparison operators
 - $a < b$, $a > b$, $a \leq b$, $a \geq b$
- Range instantiation and range checks
 - $a..b$, $x \text{ in } a..b$ (checks if x is in the range of a to b),
 $x \text{ !in } a..b$ (checks if x is not in the range of a to b)
- Increment and decrement operators
 - ++ and -- . Also, we can use the function `inc()` and `dec()` too (Both are used as post increment/decrement)

Bitwise operators

- shl(bits) – signed shift left (Java's <<)
- shr(bits) – signed shift right (Java's >>)
- ushr(bits) – unsigned shift right (Java's >>>)
- and(bits) – bitwise and
- or(bits) – bitwise or
- xor(bits) – bitwise xor
- inv() – bitwise complement

String

- ❑ String is an array of characters. Kotlin Strings are more or less similar to the Java Strings
- ❑ Strings are immutable.
- ❑ Empty String - `var s = String()`
- ❑ Properties and Functions
 - Length, `get(index)`, `compare...`
- ❑ String Templates
 - `var len = str.length`
 - `var newStr = "Length of str is ${str.length}"`
- ❑ Raw Strings – Multiline String `"""`
- ❑ String Equality
 - Referential Equality: Checks if the pointers for two objects are the same. `===` operator is used.
 - Structural Equality : Checks if the contents of both the objects are equal. `==` operator is used.

Using Conditional Expressions

□ Normal If

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

□ If Expression

– fun maxOf(a: Int, b: Int) = if (a > b) a else b

If branches can be blocks, and the last expression is the value of a block:

```
val max = if (a > b) {  
    print("Choose a")  
    a  
} else {  
    print("Choose b")  
    b  
}
```

- Using an expression rather than a statement (for example, returning its value or assigning it to a variable), the expression is required to have an else branch

When

- When replaces the switch operator of C-like languages. In the simplest form it looks like this

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

When

```
when (x) {  
    in 1..10 -> print("x is in the range")  
    in validNumbers -> print("x is valid")  
    !in 10..20 -> print("x is outside the range")  
    else -> print("none of the above") }  
}
```

- ❑ Matches its argument against all branches sequentially until some branch condition is satisfied.
- ❑ Used as an Expression / Statement.
- ❑ If when is used as an expression, the else branch is mandatory

```
when (x) {  
    0, 1 -> print("x == 0 or x == 1")  
    else -> print("otherwise")  
}
```

```
when (x) {  
    parseInt(s) -> print("s encodes x")  
    else -> print("s does not encode x")  
}
```

When

```
fun hasPrefix(x: Any) = when(x) {  
    is String -> x.startsWith("prefix")  
    else -> false  
}
```

```
when {  
    x.isOdd() -> print("x is odd")  
    x.isEven() -> print("x is even")  
    else -> print("x is funny")  
}
```

```
fun Request.getBody()  
= when (val response = executeRequest()) {  
    is Success -> response.body  
    is HttpError -> throw HttpException(response.status)  
}
```


For Loops

- For loop iterates through anything that provides an iterator.

```
for (item in collection) print(item)
```

```
for (item: Int in ints) {}
```

- To iterate over a range of numbers, use a range expression:

```
for (i in 1..3) { println(i); }
```

```
for (i in 6 downTo 0 step 2) { println(i) }
```

For Loop

- ❑ A for loop over a range or an array is compiled to an index-based loop that does not create an iterator object.

```
for (i in array.indices) { println(array[i]) }
```
- ❑ Index and Value together

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

While Loops

- while and do..while work as usual

```
while (x > 0) {
```

```
    x--
```

```
}
```

```
do {
```

```
    val y = retrieveData()
```

```
} while (y != null) // y is visible here!
```

Returns and Jumps

- Kotlin has three structural jump expressions:
 - Return
 - By default returns from the nearest enclosing function or anonymous function.
 - Break
 - Terminates the nearest enclosing loop.
 - Continue
 - Proceeds to the next step of the nearest enclosing loop.

Array (class of the type Array)

□ Declaration

- `arrayOf("India","USA","China","Australia","Sri Lanka")`
- `arrayOf<String>("India","USA","China","Australia","Sri Lanka")`
- `arrayOf("Hi",1,1L,1.2,false)`

□ Accessing and Modifying Elements in Array

- `array3.get(0)`, `array3[0]`
- `array1[1] = 6`, `array1.set(1,6)`

□ Utility functions to initialise arrays of primitives using functions such as :

- `charArrayOf()`, `booleanArrayOf()`, `longArrayOf()`,
`shortArrayOf()`, `byteArrayOf()`

Break and Continue Labels

- Any expression in Kotlin may be marked with a label. Labels have the form of an identifier followed by the @ sign

```
loop@ for (i in 1..100) {  
    for (j in 1..100) {  
        if (...) break@loop  
    }  
}
```

Array

Given –

```
var array1 = arrayOf(1,2,3,4)
```

- ❑ Kotlin Array() Constructor (size,init)
 - `val array = Array(6) {i-> i*2} //or`
 - `val array = Array(6,{i-> i*2})`
- ❑ Reversing an array
 - `array1 = array1.reversedArray()`
- ❑ Reversing an Array in place
 - `array1.reverse()`
- ❑ Sum of elements of an Array
 - `println(array1.sum())`
- ❑ Appending an Element in an Array
 - `array1 = array1.plus(5) //or`
 - `array1 = array1.plusElement(5)`

Null Safety

- ❑ Compiler by default doesn't allow any types to have a value of null at compile-time.
- ❑ For Kotlin, Nullability is a type. At a higher level, a Kotlin type fits in either of the two
 - Nullability Type
 - Non-Nullability Type
 - `var a : String? = null`
- ❑ `var streetName : String? = address?.locality?.street?.addressLine1`

Operators

❑ !! Operator

- `a = null`
- `println(a!!.length)` // runtime error.

❑ Elvis Operator (?:0)

- Up until now, whenever the safe call operator returns a null value, we don't perform an action and print null instead. The Elvis operator `?:` allows us to set a default value instead of the null
- `print(newStr?.length?:0)`

❑ Safe Casting

- The `?` operator can be used for preventing `ClassCastException` as well
- `var b : String = "2"`
- `var x : Int? = b as? Int`
- `println(x)` //prints null

Operators

□ Using let()

- Let function executes the lamda function specified only when the reference is non-nullable as shown below.
- `newString?.let { println("The string value is: $it") }`

□ Using also()

- also behaves the same way as let except that it's generally used to log the values. It can't assign the value of it to another variable.
- `newString?.let { c = it }.also { println("Logging the value: $it") }`

□ Filtering Out Null Values

- `var array2: Array<Any?> = arrayOf("1", "2", "3", null)`
- `var newArray = array2.filterNotNull()`

Control Flow

- forEach loop
 - (2..5).forEach{ println(it) }

Control Flow

- Range
 - 1..10, x in 1..10, xlin 1..10, 10 downTo 1,
 - 1 until 4(exclude 4), 1..5 step 3
- while loop / do while
- break and continue
- repeat and when
 - repeat allows us to execute the statements in the loop N number of times(where N is the number specified as the argument)..
 - when in Kotlin is equivalent to switch in other languages, though with a different syntax and more power!

Classes and Objects



Important Points

- ❑ New is no longer a keyword
- ❑ All classes are by default final
- ❑ A class is a blueprint for the object.
- ❑ Visibility Modifiers
 - public, protected, internal, private
 - Internal (
 - Setting a declaration as internal means that it'll be available in the same module only.
 - By module in Kotlin, we mean a group of files that are compiled together.

Class

- ❑ Classes in Kotlin are declared using the keyword

```
class Sample { ... }
```
- ❑ Declaration consists of
 - class name, the class header (specifying its type parameters, the primary constructor etc.)
 - class body, surrounded by curly braces.
 - Both the header and the body are optional
 - if the class has no body, curly braces can be omitted.

Constructors

- A constructor is a concise way to initialize class properties.
- It is a special member function that is called when an object is instantiated (created).
- Two Constructors
 - Primary constructor - concise way to initialize a class
 - Secondary constructor - allows you to put additional initialization logic

Primary Constructor

- The primary constructor is part of the class header. Constructor keyword is not needed if it doesn't contain visibility modifiers

```
class Person constructor(firstName: String) { ... }
```

```
class Person(val firstName: String, var age: Int) {  
    // class body  
}
```

- The primary constructor has a constrained syntax, and cannot contain any code.
- May contain default properties

Primary Constructor with Init

- Allows you to put additional initialization logic in init block

```
fun main(args: Array<String>) {  
    val person1 = Person("joe", 25)  
}  
class Person(fn: String, pAge: Int) {  
    val firstName: String  
    var age: Int  
    // initializer block  
    init {  
        firstName = fn.capitalize()  
        age = pAge  
        println("First Name = $firstName")  
        println("Age = $age")  
    }  
}
```

Initializer

- During an instance initialization, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers

```
class InitOrderDemo(name: String) {
    val firstProperty = "First property:
$name"
    init {
        println("First initializer block that
prints ${name}")
    }
    val secondProperty = "Second
property: ${name.length}"
    init {
        println("Second initializer block
that prints ${name.length}")
    }
}
```

Default Value in Primary Constructor

- You can provide default values in constructor like functions
- If all the default values are provided, JVM will automatically add default constructor to class
- `class Person(_firstName: String = "UNKNOWN", _age: Int = 0)`

Secondary Constructor

- ❑ Class can also contain one or more secondary constructors. They are created using constructor keyword.
- ❑ Secondary Constructors are not that common in Kotlin. The most common use of secondary constructor comes up when you need to extend a class that provides multiple constructors that initialize the class in different ways.

Secondary Constructor

```
class Log {  
    constructor(data: String)  
    {  
        // some code  
    }  
    constructor(data: String,  
        numberOfData: Int) {  
        // some code  
    }  
}
```

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s). Delegation to another constructor of the same class is done using the `this` keyword

<https://www.programiz.com/kotlin-programming/constructors>

Instantiate a class

- ❑ **No New Keyword**
- ❑ To create an instance of a class, call the constructor just like a function
 `val invoice = Invoice()`
 `val customer = Customer("Joe Smith")`

Properties

- Undoubtedly the most important element in any programming language
- Every Kotlin property declaration begins with the keyword `var` or `val`.
 - `val` is used when the variable is immutable i.e. read-only. Can set the value only once
 - `Var` is mutable

Properties Types And Sytnax

- The initializer, getter and setter are optional. Property type is optional if it can be inferred from the initializer (or from the getter return type, as shown below).

```
var <propertyName>[: <PropertyType>] [=
    <property_initializer>]
    [<getter>]
    [<setter>]
```

Properties

- ❑ Override Get/Set
- ❑ Change the visibility of an accessor or to annotate it, but don't need to change the default implementation, you can define the accessor without defining its body
 - `var setterVisibility: String = "abc"`
 - `private set // private with default implementation`
- ❑ Try returning and setting field values

Lateinit

- ❑ lateinit modifier is used with var properties and is used to initialize the var property at a later stage. Normally, Kotlin compiler requires you to specify a value to the property during initialization. With the help of lateinit you can delay it.

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

    var a = A()
    a.x = "Hello Kotlin"
}
```

```
class A {
    lateinit var x: String
}
```

Checking of lateinit var

- This check is only available for the properties that are lexically accessible, i.e. declared in the same type or in one of the outer types, or at top level in the same file.

```
if (foo::bar.isInitialized) {  
    println(foo.bar)  
}
```

Interfaces

- Similar to Java 8
- Contain declarations of abstract methods, as well as method implementations.
- Abstract classes Vs Interfaces
 - cannot store state.
- They can have properties but these need to be abstract or to provide accessor implementations.

Interfaces

```
interface MyInterface
{
    fun bar()
    fun foo() {
        // optional body
    }
}
```

```
class Child :
MyInterface {
    override fun bar() {
        // body
    } }
```

```
interface MyInterface {
    val prop: Int // abstract
    val propertyWithImplementation:
String
    get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

Interfaces Inheritance and Conflicts

```
interface Named {
    val name: String
}
interface Person : Named {
    val firstName: String
    val lastName: String
    override val name: String
    get() = "$firstName
$lastName"
}
```

```
interface A {
    fun foo() { print("A") }
    fun bar() }
interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") } }
class C : A {
    override fun bar() { print("bar") } }
class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }
    override fun bar() {
        super<B>.bar()
    }
}
```

Implementation by Delegation

- The Delegation pattern has proven to be a good alternative to implementation inheritance, and Kotlin supports it natively requiring zero boilerplate code. A class Derived can implement an interface Base by delegating all of its public members to a specified object:

Implementation by Delegation

```
interface Base {  
    fun print()  
}  
  
class BaseImpl(val x: Int) : Base {  
    override fun print() { print(x) }  
}  
  
class Derived(b: Base) : Base by b  
  
fun main() {  
    val b = BaseImpl(10)  
    Derived(b).print()  
}
```

Abstract Class

```
abstract class Person(name: String)
{
    abstract fun displayAge()
}
```

- Abstract keyword is used to declare abstract classes in Kotlin. An Abstract class can't be instantiated. However, it can be inherited by subclasses. By default, the members of an abstract class are non-abstract unless stated otherwise

Visibility Modifiers

- ❑ Classes, objects, interfaces, constructors, functions, properties and their setters can have visibility modifiers.
- ❑ (Getters always have the same visibility as the property.)
- ❑ Four visibility modifiers in Kotlin
 - private, protected, internal and public.

Visibility Modifiers

- ❑ Public is used by default, in case of no visibility modifier, which means that your declarations will be visible everywhere
- ❑ If you mark a declaration private, it will only be visible inside the file containing the declaration
- ❑ If you mark it internal, it is visible everywhere in the same module
- ❑ Protected is not available for top-level declarations.
- ❑ To marks a constructor private
 - `class C private constructor(a: Int) { ... }`

Extensions

- Similar to C# and Gosu, the ability to extend a class with new functionality without having to inherit from the class or use any type of design pattern such as Decorator
- Done using special declarations called extensions. Kotlin supports extension functions and extension properties.

Extensions

- To declare an extension function, we need to prefix its name with a *receiver type*, i.e. the type being extended.
- Note : **Extensions are resolved statically**

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {  
    val tmp = this[index1] // 'this' corresponds to the  
    list  
    this[index1] = this[index2]  
    this[index2] = tmp  
}
```

InFix

```
fun main(args: Array<String>) {  
    val s:String = "aaa"  
    println(s)  
    println(s.shout("ss"))  
    println(s shout "AAA")  
}  
  
infix fun String.shout(s:String):String  
{return this.toUpperCase()+s}
```

Data Class

- ❑ You need to append the class with the keyword data
- ❑ The primary constructor needs to have at least one parameter.
- ❑ Each parameter of the primary constructor must have a val or a var assigned.
- ❑ This isn't the case with a normal class, where specifying a val or a var isn't compulsory.
- ❑ Data classes cannot be appended with abstract, open, sealed or inner

Data Class built-in methods

- Kotlin Data class automatically creates the following functions for you.
 - equals() and hashCode()
 - toString() of the form
"Book(name=JournalDev,
authorName=Anupam)"
 - componentN() functions for each of the parameters in the order specified. This is known as destructuring declarations.
 - copy()

Data Class Features

- ❑ To create a parameterless constructor, specify default values to each of the parameters present in the primary constructor.
- ❑ A Data Class allows subclassing (No need to mention the keyword open).
- ❑ You can provide explicit implementations for the functions `equals()`, `hashCode()` and `toString()`
- ❑ Explicit implementations for `copy()` and `componentN()` functions are not allowed.
- ❑ We can control the visibility of the getters and setters by specifying the visibility modifiers in the constructor

Destructuring Declarations

- ❑ `componentN()` function lets us access each of the arguments specified in the constructor, in the order specified. N is the number of parameters in the constructor.
- ❑ `val (deptno,dname,loc) = deptobj`

Sealed Class

- ❑ Sealed classes are used for representing restricted class hierarchies, when a value can have one of the types from a limited set, but cannot have any other type.
- ❑ Extension of enum classes
- ❑ Subclass of a sealed class can have multiple instances which can contain state.

Sealed Class

- To declare a sealed class, you put the sealed modifier before the name of the class. A sealed class can have subclasses, but all of them must be declared in the same file as the sealed class itself

sealed class Expr

data class Const(val number: Double) : Expr()

data class Sum(val e1: Expr, val e2: Expr) : Expr()

object NotANumber : Expr()

Generics

- Same as Java

```
class Box<T>(t: T) {  
    var value = t  
}
```

Singleton Object

- A Singleton is a software design pattern that guarantees a class has one instance only and a global point of access to it is provided by that class
- object MyObject

```
object DataProviderManager {  
    fun registerDataProvider(provider: DataProvider) {  
        // ...    }  
    val allDataProviders: Collection<DataProvider>  
        get() = // ...  
}
```

Companion Objects

- ❑ An object declaration inside a class can be marked with the companion keyword
- ❑ Members of the companion object can be called by using simply the class name as the qualifier
- ❑ The name of the companion object can be omitted

```
class MyClass {  
    companion object Factory {  
        fun create(): MyClass =  
            MyClass()  
    }  
}
```

```
val instance =  
    MyClass.create()
```

```
class MyClass {  
    companion object { }  
    val x = MyClass.Companion
```


Enum Classes

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST  
}  
  
enum class Color(val rgb: Int) {  
    RED(0xFF0000),  
    GREEN(0x00FF00),  
    BLUE(0x0000FF)  
}
```

```
enum class ProtocolState {  
    WAITING {  
        override fun signal() =  
TALKING  
    },  
  
    TALKING {  
        override fun signal() =  
WAITING  
    };  
  
    abstract fun signal():  
ProtocolState
```

Classes and Inheritance



Inheritance

- All classes in Kotlin have a common superclass Any, that is the default superclass for a class with no supertypes declared
- `open class Base(p: Int)`
- `class Derived(p: Int) : Base(p)`

Inheritance

□ Override

- Properties and functions
- A member marked override is itself open, i.e. it may be overridden in subclasses. If you want to prohibit re-overriding, use final

□ Calling the superclass implementation

- Code in a derived class can call its superclass functions and property accessors implementations using the super keyword

Abstract Classes

- A class and some of its members may be declared abstract. An abstract member does not have an implementation in its class. Note that we do not need to annotate an abstract class or function with open – it goes without saying.
- We can override a non-abstract open member with an abstract one

Type Checks

- Type Checks

 - is and !is

- if (obj is String) {

 - print(obj.length)

 - }

- if (obj !is String) { // same as !(obj is String)

 - print("Not a String")

www.fandsindia.com

 - }

Smart Casts

- In many cases, one does not need to use explicit cast operators in Kotlin, because the compiler tracks the is-checks and explicit casts for immutable values and inserts (safe) casts automatically when needed:

```
fun demo(x: Any) {  
    if (x is String) {  
        print(x.length) // x is automatically cast to String  
    }  
}
```

"Unsafe" cast operator

- Usually, the cast operator throws an exception if the cast is not possible. Thus, we call it unsafe. The unsafe cast in Kotlin is done by the infix operator as

```
val x: String = y as String
```

```
val x: String? = y as String?
```

```
val x: String? = y as? String
```




A **F**ast **AND** **S**teady Approach

Interoperability



Interoperability

- Kotlin is designed with Java Interoperability in mind. Existing Java code can be called from Kotlin in a natural way, and Kotlin code can be used from Java rather smoothly as well. In this section we describe some details about calling Java code from Kotlin.
- Demo

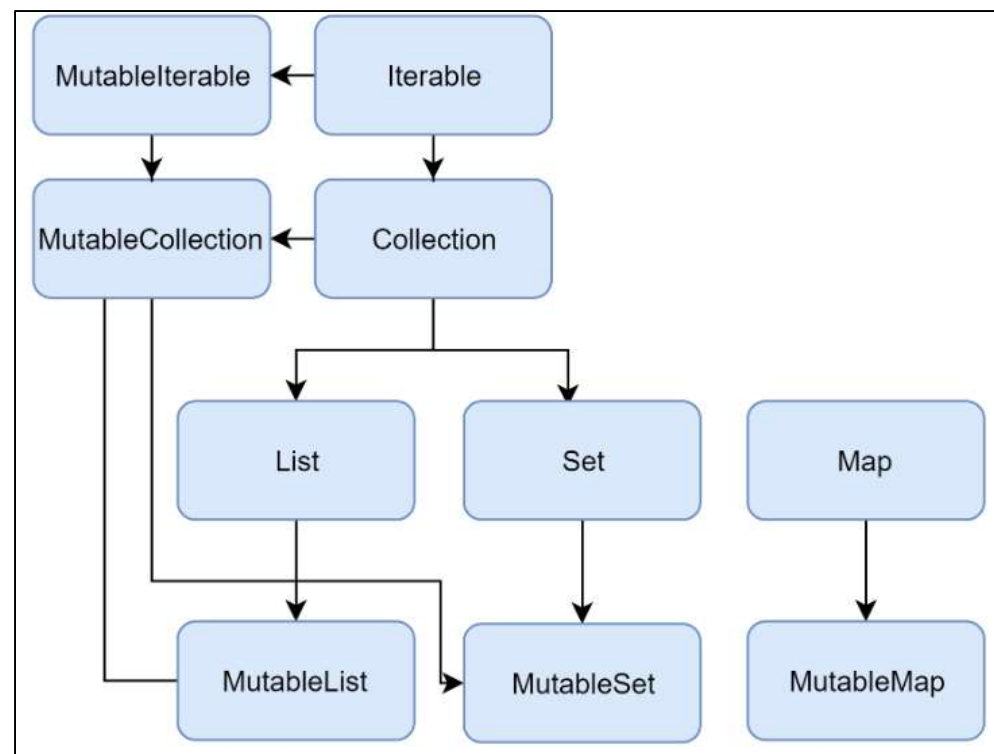


A **F**ast **AND** **S**teady Approach

Collections



Collections



- Unlike many languages, Kotlin distinguishes between mutable and immutable collections (lists, sets, maps, etc). Precise control over exactly when collections can be edited is useful for eliminating bugs, and for designing good APIs.

Collections: List, Set, Map

- It is important to understand up front the difference between a read-only view of a mutable collection, and an actually immutable collection. Both are easy to create, but the type system doesn't express the difference
- The Kotlin `List<out T>` type is an interface that provides read-only operations like `size`, `get` and so on. Like in Java, it inherits from `Collection<T>` and that in turn inherits from `Iterable<T>`. Methods that change the list are added by the `MutableList<T>` interface.

Collections: List, Set, Map

- ❑ Kotlin does not have dedicated syntax constructs for creating lists or sets. Use methods from the standard library, such as `listOf()`, `mutableListOf()`, `setOf()`, `mutableSetOf()`. Map creation in NOT performance-critical code can be accomplished with a simple idiom: `mapOf(a to b, c to d)`.

Collections: List, Set, Map

□ List

- `val theList = listOf("one", "two", "three")`
- `val theMutableList = mutableListOf("one", "two", "three")`

□ Set

- `val theSet = setOf("one", "two", "three")`
- `val theMutableSet = mutableSetOf("one", "two", "three")`

□ Map

- `val theMap = mapOf(1 to "one", 2 to "two", 3 to "three")`
- `val theMutableMap = mutableMapOf(1 to "one", 2 to "two", 3 to "three")`

Useful Operators

- Kotlin's Collections API is much richer than the one we can find in Java – it comes with a set of overloaded operators.
- The “*in*” Operator
 - We can use expression “*x in collection*” which can be translated to *collection.contains(x)*
- The “+” Operator / “-” Operator
 - We can add an element or entire collection to another using “+” operator:

Useful Methods

- ❑ `sublist - theList.slice(1..2)`
- ❑ `remove all nulls - theList.filterNotNull()`
- ❑ `filter - theList.filter{ it > 0 }`
- ❑ `drop first N items - theList.drop(2)`
- ❑ `drop items first if they satisfy the given condition - theList.dropWhile{ it.length < 4 }`
- ❑ `group elements - theList.groupBy{ it % 3 }`

Useful Methods

- ❑ map all elements using the provided function
 - `theList.map{ it * it }`
- ❑ flatmap() to flatten nested collections. Here, we are converting Strings to List<String> and avoiding ending up with List<List<String>>:
 - `theList.flatMap{ it.toLowerCase().toList() }`
- ❑ We can perform fold/reduce operation:
 - `theList.fold(0, {acc, i -> acc + (i * i)})`
 - The difference between the two is that **fold** takes an explicit initial value, whereas **reduce** uses the first element from the list as the initial value.