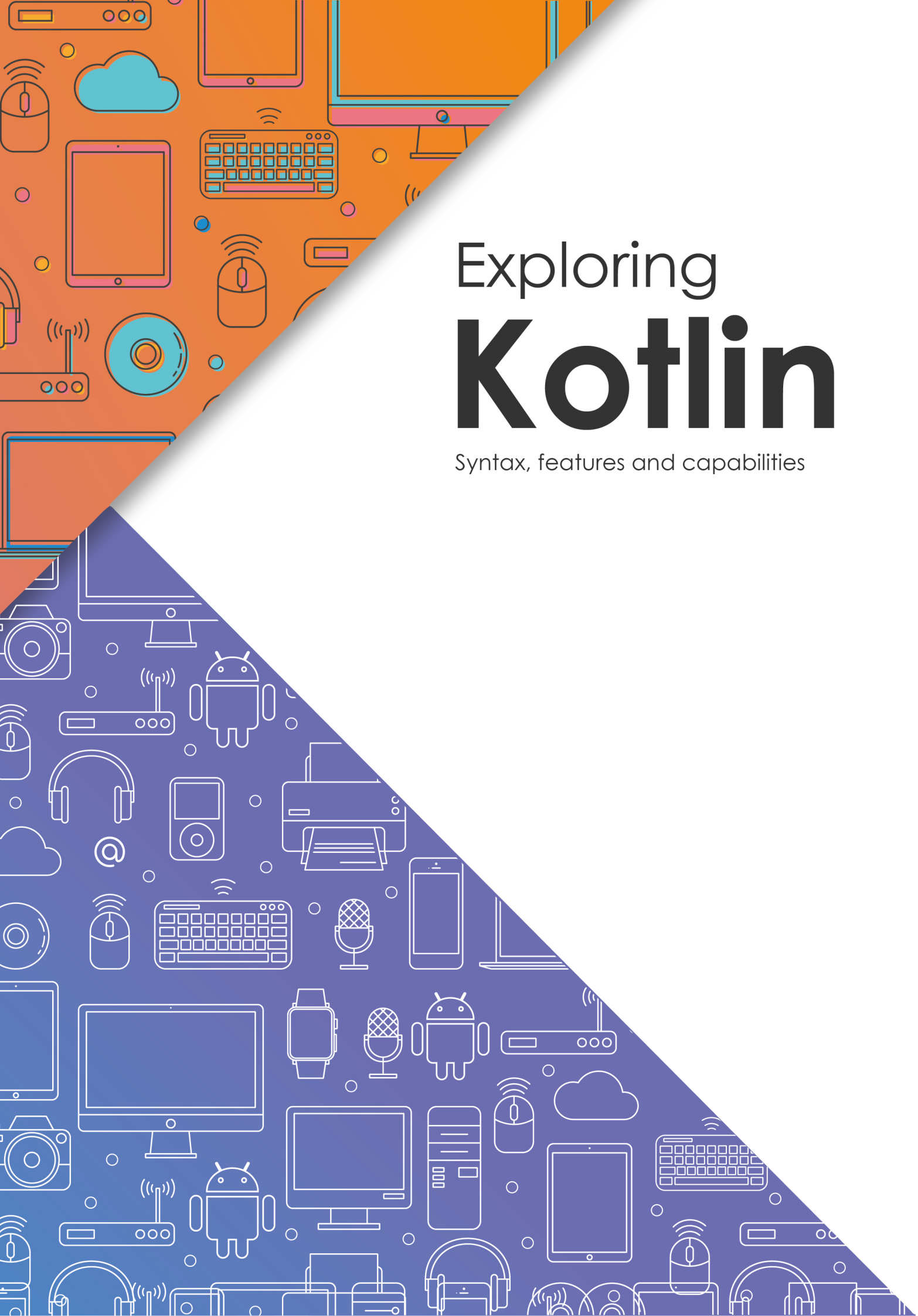

Exploring Kotlin

Syntax, features and capabilities

Exploring Kotlin

Syntax, features & capabilites

Hani M K

This book is for sale at http://leanpub.com/exploring-kotlin

This version was published on 2020-08-21

Tweet This Book!

Please help Hani M K by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Exploring #Kotlin Book Less than 100 pages covering Kotlin syntax and features in straight and to the point explanation. @hmkcode

The suggested hashtag for this book is #kotlin.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#kotlin

Thank you for downloading Exploring Kotlin.

This book is dedicated for those who spent a good amount of their career writing Java! Finally, here is a language that makes you be yourself :)

The first version of this book, is a result of individual effort. Writing this mini book was a good way to explore Kotlin. I hope reading it does the same thing too.

Your comments and suggestions are welcome.

Please, feel free to contact and follow if you like.

- *Email: hmkcode@gmail.com*
- *Github: github.com/hmkcode*
- *Twitter: twitter.com/hmkcode*
- *Blog: hmkcode.com*

Hani M. K.

Contents

CHAPTER 1 | Getting Started

- This chapter will cover the basic syntax of Kotlin and will introduce some of the tools used for coding in Kotlin.

1.1 | Tools

- There are many tools that will help you start coding in Kotlin.
- Tools listed below shall cover the needs for most Kotlin learners to start coding in Kotlin.
- Based on your goal from learning Kotlin, you may use one or more of these tools.
- In this book we will mainly be using Kotlin online playground.

Kotlin playground

- Kotlin Playground[1] is an online sandbox to explore Kotlin.
- The playground is the easiest way to start exploring Kotlin.

```
1  fun main() {
2      println("Hello, world!!!")
3  }
```

IntelliJ IDEA

- IntelliJ IDEA is the preferred IDE for developing projects in Kotlin.
- Kotlin is bundled with IntelliJ IDEA. It has all what you need to start coding in Kotlin.
- You can download the free Community Edition[2]
- You can follow the steps in this tutorial from Kotlin official site Getting Started with IntelliJ IDEA [3]

Android studio

- Android Studio fully supports Kotlin.
- You can create a new project with Kotlin-first support, add Kotlin files to existing project with Java files or converting existing Java to Kotlin.

[1] https://play.kotlinlang.org
[2] http://www.jetbrains.com/idea/download/index.html
[3] https://kotlinlang.org/docs/tutorials/getting-started.html

1.2 | Kotlin Basic Syntax

`main()` function

- The starting point for running Kotlin app is `main()` function.

```kotlin
fun main() {
    println("Hello world!")
}
```

`val` & `var` variables

- Read-only variables should be defined with `val`
- Variables that can be updated or changed is defined as `var`
- Variables must either have a type annotation or be initialized

```kotlin
val w: Int = 0 // immediate assignment
val x = 6 // type `Int` is inferred "no type provided"
val y: Int // Type is required "no initial value"
y = 7 // deferred assignment "Type of y defined before"
val z = 3 // this is OK

z = z + 1 // This is an error "Val cannot be reassigned"

var counter = 0 // this variable can be updated
counter++;
```

Comments

- Comments can be added in Kotlin as following

```kotlin
// single line comment

/* Multi-line
    comments */
```

String templates

- String template is a neat way to build a dynamic string.

```
fun main() {
    var n = 1
          println("n = $n")
    val result = "n is ${if(n > 0) "positive" else "not positive"}"
    println(result)
}
```

Output:

```
n = 1
n is positive
```

- Notice: Kotlin expression inside the curly brackets.

Functions `fun`

- Functions is a block of reusable code that is used to perform an action and may return result.
- Functions are declared using `fun` keyword.
- Functions may take variables or even other functions as parameters.

```
fun multiply(a: Int, b: Int): Int{
    return a * b
}
```

- Function with expression body and inferred return type can be written as:

```
fun multiply(a: Int, b: Int) = a * b
```

- Function parameters can have default values.

```
fun multiply(a: Int=2, b: Int=4) = a * b
```

- Function with no return value use `Unit` as return type which can be also omitted.

```
fun printResult(a: Int, b: Int): Unit{

    println("result = "+multiply(a, b))

}
```

`if` & `when` Control Flow

- `if` and `when` are used to control the flow of the program. They are used to select which code will be executed based on some conditions.

```
// if a is even return true
fun isEven(a:Int): Boolean{
    if(a % 2 == 0) return true else return false
}
```

- `if` can be written as an expression

```
// if as expression i.e. returns result
fun isEven(a:Int) =  if(a % 2 == 0) true else false
```

- *Even simpler version of the function above is possible*

```
fun isEven(a:Int) =  (a % 2 == 0)
```

- `when` tries sequentially to match its argument against all branches.
- if `when` is used as an expression, `else` branch is mandatory unless exhaustiveness of possible cases is guaranteed as in the case of `enum class` and `sealed class`.

```
fun check(a:Int){
    when(a%2){
        0 -> println("Even")  // i.e. if(a%2 == 0)
        else -> println("Odd")
    }
}
```

`null` saftey using `?`

- Object type should be marked with `?` if its reference is nullable i.e. can possibily receive `null` value.

```
val str:String? = null  // str is nullable

// getObj function can take null and return null
fun getObj(str: String?) : String? {...}
```

- Also, to safely call a nullable object, `?.` operator can be used to check if an object is `null` before calling a function on that object.

```
// str is nullable
val str:String? = null

// explicit way to check if str is null
 val length = if(str != null) str.length else 0

// alternative way to check for null using ?
val length = str?.length
```

`is` type checks and `as` type casting

- `is` operator is used to check if an object is an instance of a certain type.
- `is` will automatically cast checked object into the type being checked.

```
val str:Any = "" // str of type Any initialized with String

if(str is String) // true
    println(str.length) // str is cast to type String
```

- `as` operator is used to explicitly cast an object into a specific type.

```
val str:Any = ""
println((str as String).length) // str is cast from Any to String
```

`for` & `while` loop

- `for` loop is used to iterates over any iterable object e.g. collection, sequence, range, ...

```
val collection = listOf(1,2,3,4)

for(e in collection)
    println(e)
```

- To loop over a range of numbers we can use range operator e.g. `1..9`

```
for(i in 1..9)
    prinlnt(i)
```

- `while` loop is another way to repeat block of code as long as a certain condition is true.

```
while(true){ // this will loop forever!

}
```

Class and classes types

- A class is the basic building block of object-oriented language.
- A class represents an entity that has properties and can do some action.
- `class` keyword is used to declare a class.
- A class is used to create instances or objects of the class type.

```
//minimum valid calss
class SomeClass
```

- A class can have its own properties, or “variables”, and functions.

```
class SomeClass(val id:Int){ // class with primary constructor
    var name:String = ""   // parameter

    function doSomething(){ println("doing something!")} // function
}
```

- To create an instance, call class name with its primary constructor parameters.

```
val someClass = SomeClass(3) // create instance of class SomeClass
someClass.doSomething() // call class method.
```

- There are many other types of classes in Kotlin such as `abstract class`, `inline class`, `sealed class`, ...etc.

Collections

- Collection is a container for objects of similar type.
- There are three basic types of collections namely, `List`, `Set` and `Map`

```
val list = listOf("a", "b", "c") // collection of type list
val map = mapOf("key1" to "one",
    "key2" to "two",
    "key3" to "three") // map colleciton
```

- An element of a collection can be retrieved by its index or by key for map.

```
println(list[0])  // a
println(map["key3"]) // three
```

- To iterate over the whole collectoin elements you can use `for` loop

```
for(e in list)
    println(e)
```

Basic types

- In Kotlin, everything is an object that has functions and properties.
- Some types such numbers, characters and boolean can be presented as primitive at runtime.

Numbers

Type	Size(bits)
Byte	8
Short	16
Int	32
Long	64
Float	32
Double	64

```
val x = 1 // Type inferred as Int
val y = 5000000000 // Type inferred as Long
val z = 1L // Explicit type Long
val b: Byte = 1

val PI = 3.14 // Type inferred as Double
val exp = 2.7182818284 // Double
val expFloat = 2.7182818284f // Float, actual value is 2.7182817
```

Character

- A character is defined using `Char`.
- Character value is single quoted e.g. `'A'`.

- Some special characters include `'\t'`, `'\b'`, `'n'`,...etc

Boolean

- `Boolean` has two values `true` and `false`
- Booleans operators are `||` for OR, `&&` for AND and `!` for negation.

CHAPTER 2 | Classes and Inheritance

2.1 | Class Anatomy

- Class is the basic building block of object-oriented programming languages.
- A Class is a representation of an entity that may have properties and can do actions.
- Classes define the blueprint for creating objects.

Declaring a class

- `class` keyword is used to declare a new class.

```
1  class SomeClass{ /* class memebers */ }
```

- Curly brackets `{}` can be omitted if the class has no body.

```
1  class SomeClass
```

Class members

- Class can contain:
  - Constructors declared with `constructor`
  - Properties declared with `val` and `var`
  - Functions declared with `fun`
  - Nested & inner classes *more in chapter 6...*
  - Object declaration and companion objects declared using `object`

Constructor

- A class can have zero or more constructors.
- Constructors can be declared in the class header or body using `constructor` keyword.
- Constructors can have zero or more parameters with optional default values.
- A constructor declared in the class header is called **primary** constructor.
- A class can have one **primary** constructor.

```
class SomeClass constructor(id:Int) {/* ... */} // primary constructor
```

- Keyword `constructor` can be omitted if not preceded by a modifiers of annotation.

```
class SomeClass(id:Int) {/* ... */} // constructor is omitted
class SomeClass @Inject constructor(id:Int) {/*...*/} // constructor is required
```

- Constructors declared in the class body are called **secondary** constructors.
- A class can have multiple **secondary** constructors.
- **Secondary** constructor should call **primary** constructor "if exists" using `this` keyword.

```
class SomeClass(id:Int){
    var id:Int = id
    var name: String = ""

    // secondary constructor
    constructor(id:Int, name:String): this(id){ // this calls primary
        this.name = name
    }
}
```

Creating class instance

- Once a class is declared, instances can be created by calling class name followed by brackets `()` containing values for constructor parameters if any.

```
class SomeClass(id:Int) // declare a class

val someClass:SomeClass = SomeClass(3) // create instance
```

Class properties

- A class can have variables defined within its body called properties.
- Properties are define using `val` or `var` keywords.
- *More about properties in chapter 4*

```
class SomeClass(id:Int) {
    val id:Int = id
}
```

- Properties can be defined in **primary** constructor by marking its parameters with `val` or `var` keywords.

```
class SomeClass(val id:Int) // this equivalent to the previous class above
```

Class functions

- A class can define functions within its body too.
- Functions are declared using `fun` keywords.
- Functions can have parameters with optional default values.
- *More about functions in chapter 12*.

`init` Block

- initializing blocks are used to execute some code during instance initialization.
- `init` keyword is used to declare initialization block.
- Initialization blocks are executed in the order they appear in the class body.

```
class SomeClass{
    init{ println("Class in initializing...")}

    val property1 = "1st property".also(::println)
    init{ println("2nd init...")}
    val property2 = "2nd property".also(::println)

    override fun toString() = "Class is created!"
}

fun main() {
    println(SomeClass())
}
```

Output:

```
Class in initializing...
1st property
2nd init...
2nd property
Class is created!
```

- If a class is created using secondary constructor, then delegation to the primary constructor is executed before all init blocks i.e. `this` will be called first.

```
fun main() {
    println(SomeClass("Primary"))
    println("--------------------")
    println(SomeClass(1, "Secondary"))
}
class SomeClass(name:String){

    constructor(id:Int, name:String):this(name.also(::println)) {}
    init{ println("Class in initializing...")}

    override fun toString() = "Class is created!"
}
```

Output:

```
Class in initializing...
Class is created!
--------------------
Secondary
Class in initializing...
Class is created!
```

Constructor parameters vs. class properties

- Variables defined in the constructors are called constructor parameters.
- Variables declared within class body are called properties.
- Parameters can be used to initialized class properties.

```
class SomeClass(id:Int) { // id is a parameter
    val id:Int = id // id is class property
}
```

- Properties can be declared and initialized in the **primary** constructor by marking constructor parameters with `val` and `var`

```
class SomeClass(var id:Int = 0)
```

2.2 | Class Inheritance

- A class can inherit from another class or be inherited by other classes.
- A class we are inheriting from is called superclass while the inheriting class is called subclass.
- By inheriting from a superclass, subclasses can reuse superclass `open` members or `override` them.
- By default all classes in Kotlin are subtypes of supertype `Any`.

`open` for inheritance

- A class should be marked as `open` to allow other classes to inherit its members.

```
open class ParentClass
```

Extending a Class

- A class can inherit or extend other class by using `:`.
- The superclass should be marked as `open`

```
class ChildClass : ParentClass()
```

Initializing superclass

- Superclass should be initialized by subclass.
- If the subclass has a **primary** constructor, then superclass will be initialized in class header.

```
open class ParentClass(open val id:Int)
class ChildClass(id:Int) : ParentClass(id)
```

- If a subclass has NO **primary** constructor, then each **secondary** constructors should directly or indirectly call `super` which initialize superclass constructor.

```
open class ParentClass(open val id:Int)
class ChildClass : ParentClass{

    constructor(id:Int):super(id)

    // super is called by delegating to other constructor
    constructor(id:Int, name:String):this(id)
}
```

`override` functions

- A subclass can call superclass functions marked as `open`.
- A subclass can also `override` functions superclass functions marked as `open`.
- By overriding we mean using the exact name and signature of a superclass open function to declare a function in a subclass.
- Functions marked with `override` are `open` by default unless marked as `final`
- Functions in superclass with NO `open` modifier can't be overridden or even declared in subclasses.
- `private` functions can't be declared as `open`.

```
open class ParentClass{
    open fun doSomething() { println("ParentClass.doSomething()")}
    fun doSomethingElse() { println("ParentClass.doSomething()")}
}
class ChildClass : ParentClass(){

   override fun doSomething() { println("ChildClass.doSomething()")}
   // fun doSomethingElse() { println("ChildClass.doSomething()")} can't be decla\
red or overriden
}
```

`override` properties

- `open` properties of a superclass can be overridden in a subclass using `override`.
- `val` properties can be overridden with `var` but not the other way around.
- `private` properties can't be declared as `open`.

```
open class ParentClass{
    open val name:String = "ParentClass"
}
class ChildClass : ParentClass(){
    override var name:String = "ChildClass"
}
```

- Concise syntax

```
open class ParentClass(open val name:String = "ParentClass")
class ChildClass(override var name:String = "ChildClass") : ParentClass(name)
```

Initializing order

- Initialization of superclass is done before the initialization of subclass.
- Initialization of superclass properties are also done before subclass properties.
- Parameter evaluation of a superclass preceding all initializations.
- `init` block and properties initialization are executed in the order they appear in the code.

```
open class ParentClass(name:String){
    open val name = name.also{ println("Initializing Parent name=$name")}
    init{ println("init ParentClass...")}
}
class ChildClass(name:String) : ParentClass(name.also{
    println("Passing to Parent name=$name")}){

    override var name = name.also{ println("Initializing Child name=$name")}
     init{ println("init ChildClass...")}
}

fun main() {
    ChildClass("ChildClass")
}
```

Output:

```
Passing to Parent name=ChildClass
Initializing Parent name=ChildClass
init ParentClass...
Initializing Child name=ChildClass
init ChildClass...
```

Accessing superclass members

- Subclasses can access its superclass non-private properties and functions using `super` keyword.

```
open class ParentClass(open val name:String){
    open fun doSomething() {
        println("ParentClass.doSomething")
        println("ParentClass.name= "+this.name)
    }
}
class ChildClass(name:String) : ParentClass("Parent"){
     override val name = name+" "+super.name

     override fun doSomething() {
         super.doSomething()
         println("ChildClass.doSomething with")
         println("ChildClass.name= "+this.name)
     }
}

fun main() {
```

```
18      ChildClass("Child").doSomething()
19  }
```

Output:

```
1  ParentClass.doSomething
2  ParentClass.name= Child Parent
3  ChildClass.doSomething with
4  ChildClass.name= Child Parent
```

CHAPTER 3 | Packages & Visibility Modifiers

3.1 | Packages

- Kotlin source files may be organized into packages using `package` keyword.
- Packages help avoiding naming collision.
- Content of a source file declared with a `package` are part of that package.
- Package name will be part of the source file content e.g. functions and classes full name.

```
1  package com.hmkcode
2
3  fun doSomething() { /*...*/ } // com.hmkcode.doSomething
4  class SomeClass { /*...*/ } // com.hmkcode.SomeClass
```

- A package should be imported if its content is used in another source file.

```
1  package com.kotlin
2
3  import com.hmkcode.*
```

- Source files without package declaration are part of the default package that has no name.

3.2 | Visibility Modifiers

- Visibility modifiers control the accessibility of classes, objects, interfaces, constructors, functions and properties.
- Kotlin has four visibility modifiers: `private`, `protected`, `internal` and `public`.
- `public` is the default modifiers.

Top-level

- Classes, functions, properties, ...etc. declared directly inside a package are called top-level.
- Top level classes, functions, properties,...etc. marked as
  - `public` are visible everywhere.
  - `private` are visible within the source file they declared in.
  - `internal` are visible in the same module.
  - `protected` not available for top-level declarations.

```kotlin
package com.kotlin

private fun doSomething(){  // visible in this file
    println("doSomething...")
}
protected val name:String = "Some Name" // protected not allowed

fun doAnotherThing(){   // visible everywhere
    println("doAnotherThing...")
}
```

```kotlin
package com.hmkcode

import com.kotlin.*

fun main(){
    // doSomething(); // not accessible "private"
    doAnotherThing() // public is accessible
}
```

Class members

- Class members such as functions, properties and constructs marked as:
  - `public` are visible everywhere.
  - `private` are visible within the class.
  - `internal` are visible within the module.
  - `protected` are visible within the class and subclasses.

```kotlin
package com.kotlin

open class SomeClass{
    private val name:String = "SomeName"
    protected val id:Int = 2

    internal open fun doSomething(){
        println("name = $name")
    }
}

class SubClass : SomeClass(){
    override fun doSomething(){
        // println("name = $name") "name" is private -> not visible
        println("id = $id") // "id" is protected -> visible in subclass
    }
}
```

```
1  package com.hmkcode
2
3  import com.kotlin.*
4
5  fun main(){
6      // SomeClass().name // "name" is private -> not visible
7      SomeClass().doSomething() // "doSomething" is internal -> visible same module
8      SubClass().doSomething()  // overridden "doSomething" is public
9  }
```

CHAPTER 4 | Properties & Fields

4.1 | Class Properties

- `val` and `var` variables declared within a class are called class properties.

```
class SomeClass{
    var name:String = ""
}
```

- Property can be access by its name

```
SomeClass().name
```

Getters and Setters

- Kotlin, automatically generates accessors for properties with default implementation.
- `var` properties have `get()` & `set()` accessors while `val` properties have `get()` acceesor only.
- Within the implementation of the `get()` & `set()`, you can refer to the property using backing field `field` keyword.

```
class SomeClass{

    var name:String = ""
            get() = field  // "field" is a backing field refer to name
        set(value) {
            field = value
        }
}
```

- You can override default implementation of `get()` & `set()`
- Notice that, `set()` is NOT called on initialization of property.

```
class SomeClass{
    var name:String = "" // set() is not called in initialization
            get() = field.toUpperCase()
        set(value) {
            println("Setting name to $value ...")
            field = value
        }
}

fun main() {
    val someClass = SomeClass()
    someClass.name = "somename" // call custom set()
    println("name = "+someClass.name) // call custom get()
}
```

4.2 | Compile-Time Constants

- Kotlin allows the declaration of compile time constants using `const` modifier.
- Compile time constants have **primitive** values known at compile time.
- A constant is allowed to be declared as a top-level, a member of object declaration or companion object.
- Constants should be initialized.
- No custom getter is allowed.

```
const val VERSION: Int = 2
```

4.3 | Late-Initialization

- Properties and variables should be intialized with a value or `null` for nullable type.

```
class SomeClass{
    var name:String = "SomeName" // property initialized
    var id:Int? = null  // nullable property intialized with null
    override fun toString():String{
        return "SomeClass name=$name"
    }
}
```

```
fun main() {
    var someClass:SomeClass = SomeClass() // variable initialized
    println(someClass)
}
```

- In case you cannot supply an initializer value you can mark the property with `lateinit`.
- `lateinit` can be used on `var` properties declared in the class body, top-level properties and local variables.

- `lateinit` modifier is not allowed on properties of primitive types.

```
lateinit var someClass:SomeClass


class SomeClass{
    var id:Int? = null // lateinit NOT allowed on primitive types
    lateinit var name:String

    override fun toString():String{
        return "SomeClass (id = $id , name = $name)"
    }
}
```

- Accessing a `lateinit` property before it has been initialized throws an exception.
- To avoid the exception, you can check if the `lateinit var` has already been initialized. using `.isInitialized`.

> *`.isInitialized` 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 le.*

```
lateinit var someClass:SomeClass

class SomeClass{
    var id:Int? = null
    lateinit var name:String

    fun isNameInitialized():Boolean{
        return this::name.isInitialized
    }
    override fun toString():String{
        return "SomeClass (id = $id , name = $name)"
```

```
    }
}

fun main() {
    println(::someClass.isInitialized) // false
    someClass = SomeClass()

    println(someClass.isNameInitialized()) // false
    someClass.name = "SomeName"

    println(someClass.isNameInitialized()) // true
    println(someClass) // SomeClass (id = null , name = SomeName)
}
```

CHAPTER 5 | Abstract Classes and Interface

5.1 | Abstract Class

- Abstract class is declared with `abstract class` keywords.
- A class should be declared as `abstract`, if it has abstract members.
- Abstract members are unimplemented functions or properties.
- `abstract` class can have both abstract and concrete members.
- Any non-abstract class inheriting an `abstract` class must implement abstract members.
- A class can only extend one abstract class.
- Abstract classes cannot be instantiated.
- Abstract classes are `open` by default.

```
abstract class AbstractClass {
    abstract val name:String  // abstract property

    val id:Int = 9
            get() = field

    abstract fun doSomething() // abstract function

    fun doSomethingElse(){ println("AbstractClass.doSomethingElse()")}
}

class SomeClass(override val name:String) : AbstractClass() {
    override fun doSomething() { println("ChildClass.doSomething()") }
}
```

5.2 | Interface

- • Interfaces are declared with `interface` keywords.
- Similar to abstract classes, interfaces can contain abstract and non-abstract members.
- All unimplemented members are abstract by default i.e. no need to mark members as `abstract`.
- A class in Kotlin can implement more than one interface.
- Properties in the interface can NOT have backing fields, so they can NOT maintain state. In other words, you don't have access to `field`.

```
interface SomeInterface {
    val name:String // NO abstract keyword

    val id:Int
            get() = 9 // field not accessiable

    fun doSomething() // NO abstract keyword

    fun doSomethingElse(){ println("AbstractClass.doSomethingElse()")}
}

class SomeClass(override val name:String) : SomeInterface {
    override fun doSomething() { println("ChildClass.doSomething()") }
}
```

Multiple implementation inheritance

- Since a class can inherit from multiple interface, we may inherit more than one *implemented* function of the same name.
- To resolve this conflict, the inheriting class must override this method.
- Inherited implemented functions can be accessed using qualified `super<...>`.

```
interface A {
    fun doSomething(){ println("A.doSomething()")}
}

interface B{
    fun doSomething(){ println("B.doSomething()")}
}

class SomeClass() : A, B {
    override fun doSomething() {
        super<A>.doSomething()
        super<B>.doSomething()
        println("SomeClass.doSomething()")
    }
}
```

CHAPTER 6 | Nested Class, Inner Class, Local Class & Anonymous Class

6.1 | Nested Class

- A class can be nested within another class.
- Nested class can also be defined within abstract classes and interfaces.

```
class SomeClass {
    class NestedClass{
        fun doNestedThing() {println("SomeClass.NestedClass.doNestedThing()")}
    }
}
interface SomeInterface {
    class NestedClass{
        fun doNestedThing() {
            println("SomeInterface.NestedClass.doNestedThing()")
        }
    }
}
```

- A nested class can be instantiated as following:

```
fun main() {
    SomeClass.Nested().doNestedThing()
    SomeInterface.Nested().doNestedThing()
}
```

6.2 | Inner Class

- Inner class is nested class marked with `inner`.
- Inner classes can access member of outer class instances.

```
class SomeClass {
    val name:String = "SomeName"
    inner class InnerClass{
        fun doInnerThing() {println("$name.InnerClass.doNestedThing()")}
    }
}
```

- An inner class can be instantiated as following:

```
fun main() {
    SomeClass().InnerClass().doInnerThing()
}
```

- `inner` is not allowed inside interfaces.
- `inner` is allowed within abstract classes.
- Abstract class with inner class should be extended to create an instance of the inner class.

```
abstract class AbstractClass {
    val name:String = "AbstractClass"
    inner class InnerClass{
        fun doInnerThing() {println("$name.InnerClass.doNestedThing()")}
    }
}

class ChildClass : AbstractClass(){}
```

6.3 | Local Class

- Local class is an inner class defined within a function or constructor scope.

```
class SomeClass {
    val name:String = "SomeClass"

    constructor(){

        class InnerClass{
                fun doInnerThing() { println("$name.constructor") }
        }
        InnerClass().doInnerThing()
    }
    fun doSomething(){
```

```

        class InnerClass{
                fun doInnerThing() { println("$name.doSomething") }
        }
        InnerClass().doInnerThing()
    }
}

fun main() {
    SomeClass().doSomething()
}
```

Output:

```
SomeClass.constructor
SomeClass.doSomething
```

6.4 | Anonymous Class

- Abstract classes and interfaces are not instantiable by themselves.
- Abstract classes and interfaces are usually extended to create a reusable implementation i.e. classes.
- Anonymous inner classes can be used to create an implementation of an abstract classes or interfaces.
- Anonymous inner class instances can be created using `object` expressions.

```
interface EventHandler{
    fun handle()
}

class EventListener {
   fun listen(eventHandler: EventHandler){
       eventHandler.handle()
   }
}

fun main() {
    EventListener().listen(object : EventHandler {
        override fun handle(){
            println("Handling event...")
        }
    })
}
```

CHAPTER 7 | Data Classe, Enum Class, Sealed Class, Inline Class

7.1 | Data Class

- Data classes are mainly created to hold data.
- Usually such kind of classes are called model class, POJO in Java, or value object.
- `data` keyword is used to mark a class as a data class.

```
data class DataClass(val id:Int, val name: String)
```

- Data class should have at least one parameter in the primary constructor.
- Parameters in the primary constructor need to be marked as `val` or `var`
- `abstract`, `open`, `sealed` & `inner` keyword are not allowed with `data class`.
- Data classes can inherit from other classes.
- Compiler will auto implement `toString()`, `equals()`, `hasCode()`, `copy()` & `componentN()`

toString()

- `toString()` function is auto generated for data classes.
- Parameters inside the primary constructor is used in the generated `toString()` function.

```
data class DataClass(val id:Int = 0, val name: String = "SomeName")

fun main() {
    println(DataClass())
    println(DataClass(1))
    println(DataClass(1, "AnotherName"))
}
```

Output:

```
DataClass(id=0, name=SomeName)
DataClass(id=1, name=SomeName)
DataClass(id=1, name=AnotherName)
```

copy()

- `copy()` generate a copy of an existing object with ability to change some of its properties.

```
fun main() {
    val dataClass = DataClass()
    val copiedDataClass = dataClass.copy(id=3)
    println(copiedDataClass)
}
```

Output:

```
DataClass(id=3, name=SomeName)
```

7.2 | Enum Class

- Enum classes are used to define a set of named *type-safe* constants.
- Each constant is an instance of the enum class.
- Enum classes can be declared using `enum` keyword.

```
enum class Color {
        RED, YELLOW, BLUE
}
```

```
val color:Color = Color.BLUE
```

- Enum classes can have properties, functions and implement interfaces.

```
interface Painter{
    fun paint()
}
enum class Color(val hex:String) : Painter {
    RED("#FF0000"),
    YELLOW("#FFFF00"),
    BLUE("#0000FF");

    override fun paint(){ println("painting...")}
}
```

- Enum constants can have properties and functions defined within anonymous classes.

```
enum class Color(val hex:String) {
    RED("#FF0000"){
        override fun paint() { println("painting RED...")}
    },
    YELLOW("#FFFF00"){
        override fun paint() { println("painting YELLOW...")}
    },
    BLUE("#0000FF"){
        override fun paint() { println("painting BLUE...")}
    };

    abstract fun paint()
}

fun main() {
   Color.YELLOW.paint()
}
```

Using `values()` & `valuesOf()`

- You can use `values()` to get the list of enum constants.
- To get an enum constant by its name use `valueOf()`

```
enum class Color(val hex:String) {
    RED("#FF0000"){
        override fun paint() { println("painting RED...")}
    },
    YELLOW("#FFFF00"){
        override fun paint() { println("painting YELLOW...")}
    },
    BLUE("#0000FF"){
        override fun paint() { println("painting BLUE...")}
    };

    abstract fun paint()
}

fun main() {
   Color.values().forEach(Color::paint)
   println(Color.valueOf("RED"))
}
```

Generic way `enumValues<T>()` & `enumValueOf<T>()`

- `enumValues<T>()` is a generic verion of `values()`
- `enumValueOf<T>()` is a generic verion of `valueOf()`.

```
fun main() {
   enumValues<Color>().forEach(Color::paint)

   println(enumValueOf<Color>("RED"))
}
```

`ordinal` & `name`

- Enum classes have two default properties `ordinal` & `name`
- `name` returns constant name.
- `ordinal` returns constant position.

```
fun main() {
   enumValues<Color>().forEach {
       println("position: "+it.ordinal +" - name: "+it.name)
    }
}
```

7.3 | Sealed Class

- A sealed class can be used to define an exclusive set of classes extending this sealed class.
- In other words, having a limited set of classes extending the sealed class.
- The limitation can be ensured by defining the set of classes in the same file where the sealed class is declared.
- Sealed class is declared using `sealed` keyword.
- A sealed class is abstract and can have abstract members.

```
// in the same file
sealed class SealedClass
data class SomeClass: Expr()
object someObject : Expr()
```

Enum Class vs. Sealed Class

- Enum class has a set of constant objects which are instances of the enum class.
- Sealed class has a set of subclasses which can be used to generate multiple instances.

7.4 | Inline Class *experimental*

- Inline class creates a wrapper around other type without runtime overhead.
- Inline class is declared using `inline` keyword.
- Inline class must have a single property in the primary constructor.

```
inline class Tempreature(final val value:Double)
```

- At runtime, created instances of inline class can be represented as the wrapper calss or as *the single property*

```
val temperature = Temperature(35.2)

// for the JVM this will be represented as
// val temperature = 35.2
```

- Inline class can declare properties and functions.
- Properties accessors don't have access to backing fields.
- No `lateinit` or `delegated` properties allowed.
- Inline class can implement interfaces, but can NOT extend other classes.
- Inline class is `final`.

```
interface Printable{
    fun print()
}
inline class Tempreature(final val value:Double){

    fun fahrenhite(): Double{
        return value/24
    }
}
```

CHAPTER 8 | Object Expression and Declarations & Companion objects

8.1 | Object Expression

- Object expression allow creating an object without declaring a class or subclass.

```
fun main() {

    val obj = object {
        val id:Int = 0
        val name:String = "ObjectName"
    }

    println(obj.name)
}
```

- Also, it can be used to create an object of anonymous class.

```
interface A { fun call() }

fun main() {

    val obj = object : A {
        val id:Int = 0
        val name:String = "ObjectName"

        override fun call(){
            println("calling...")
        }
    }
    obj.call()
}
```

- Object created using object expression can be used in local and private declarations.
- The actual type of anonymous object will be declared supertype or `Any` if used as return type of `public` function or `public` property.
- Object expressions can access variables from the outer enclosing scope.

```
interface A { }

class SomeClass(val name:String) {

    // Private memebers
    private val obj = object {
        val x: String = name
    }
    // Public memeber
    public val obj2 = object: A {
        val x: String = name
    }
    fun access() {
        val x1 = obj.x // this is  OK!

        // Type is Supertype "A" or Any
        val x2 = obj2.x // ERROR: Unresolved reference 'x'
    }
}
```

8.2 | Object Declarations

- Object declarations enable us to declare a *named* object without declaring a class.
- This is useful to declare singletons.
- `object` to declare object keyword can be used just like `var` and `val` do for variables declare.
- Declared object is singleton so it cannot be local.

```
object obj {
   fun call(){
       println("calling")
   }
}

fun main() {
    obj.call()
}
```

8.3 | Companion Objects

- `companion` keyword can be used for object declaration inside a class.
- Members of the companion object can be called using class name.

- Companion object name is optional and can be omitted.
- Class name acts as a reference to the companion object.
- Companion object can extend other classes and implement interfaces.

```
interface A

class SomeClass{
   companion object obj : A {
                  fun call(){ println("calling") }
   }
}

fun main() {
    SomeClass.call()
}
```

CHAPTER 9 | Delegation & Property Delegation

9.1 | Delegation

- Usually, inheritance is a common solution to code reusability. Extending class implementation will allow subclasses to reuse implemented functions and properties.
- Delegation design pattern, provides an alternative solution to inheritance using object composition.
- In Kotlin a class can delegate the implementation of an interface to **specified object** instead of explicitly implementing the abstract members.
- Kotlin provides support for this pattern by generating all interface members that delegate to the **specified object**.

    - To better understand the idea let us assume that we have two display modes for an application, dark mode `DarkMode` and light mode `LightMode`. Each mode is implementing an interface `Mode`.
    - Now, if we want to build a *custom* dark mode that inherits some functionalities from the *original* dark mode, we can simply extend `DarkMode` and then override members that need to be customized.
    - However, if we need to do the same thing for the light mode, we will end up creating two new custom modes or subclasses.
    - To avoid extending many classes, delegation can come into rescue.
    - Instead of using inheritance to solve this problem we can use delegation pattern to achieve the same goal in a cleaner way using composition.
    - We will create one custom mode class `MyCustomMode` that implements `Mode` interface but delegates the implementation of the abstract members to an object which is also a of type `Mode`.
    - The delegate object is passed to `MyCustomMode` as a parameter.
    - Kotlin provides a keyword `by` to define a delegate.

```
1  interface Mode{
2      val color:String
3      fun display()
4  }
5
6  class DarkMode(override val color:String) : Mode{
7      override fun display(){
8          println("Dark Mode..."+color)
9      }
10 }
11 class LightMode(override val color:String) : Mode {
```

```
12      override fun display() {
13          println("Light Mode..."+color)
14      }
15  }
16
17  // delegating class can override memebers of the interface
18  class MyCustomMode(mode: Mode): Mode by mode {
19      override val color:String = "CUSTOM_COLOR"
20  }
```

`override` interface memeber

- Delegating class can `override` abstract members of the interface which is already implemented in the delegate object.
- Note that delegate object does not have access to the overridden members in the delegating class.

```
1  fun main() {
2     val darkMode = DarkMode("BLACK")
3     val lightMode = LightMode("WHITE")
4
5     val myCustomMode = MyCustomMode(lightMode)
6     myCustomMode.display() // Light Mode...WHITE
7
8     println(myCustomMode.color) // CUSTOM_COLOR
9  }
```

9.2 | Delegated Properties

- Kotlin supports delegated properties which delegate some common actions executed upon getting or setting properties values.
- Some common use cases include computing property value upon first access, observing property change and storing properties in a map.
- `by` keyword is used for property delegation.
- The **delegate** has to provide `getValue()` function for `val` & `var` properties and `setValue()` function for `var` properties.
- `getValue()` will be called every time a property value is retrieved, while `setValue()` will be called every time the property value is set.
- `getValue(thisRef:Any?, property:KProperty<*>): T`, takes two parameters. First one is `this` object which contains the property we are reading. The second parameter is a metadata about the property such as property name. The function returns object of the property type.
- `setValue(thisRef: Any?, property: KProperty<*>, value: String)` takes three parameters. The the first two are similar to the `getValue()` parameters. The third parameter is the valued assigned to the property.

```
import kotlin.reflect.KProperty

class SomeClass{
    var name:String by Delegate()

    override fun toString():String{
        return "SomeClass"
    }
}

class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        println("getValue() is called,
            $thisRef.${property.name} property delegated to Delegate!")
        return "SomeValue"

    }
    operator fun setValue(thisRef: Any?,
                          property: KProperty<*>,
                          value: String) {

        println("setValue() is called,
            $value has been assigned to $thisRef.${property.name}")
    }
}
```

```
fun main() {

    // getValue() will be called
        println(SomeClass().name)

    // setValue() will be called
    SomeClass().name = "NewValue"

}
```

Output:

```
getValue() is called, SomeClass.name property delegated to Delegate!
SomeValue
setValue() is called, NewValue has been assigned to SomeClass.name
```

9.3 | Standard Delegates

Lazy

- `lazy()` is a function that takes a lambda and return a delegate which is an instance of `Lazy<T>`.
- `lazy()` is used to implement lazy property, the first call to `get()` execute the lambda passed to `lazy()` function and store the value.
- Subsequent calls to `get()` returned the stored value.

```
fun main() {
   println(lazyProperty)
   println(lazyProperty)
}

val lazyProperty: String by lazy {
    println("first call")
    "I am Lazy!"
}
```

Output:

```
first call
I am Lazy!
I am Lazy!
```

Observable

- `Delegates.observable()` takes two parameters: initial value and handler.
- The handler is called every time the property is set.
- The handler has three parameters: the property, old value and new value.

```
import kotlin.properties.Delegates
fun main() {
   SomeClass().observableProperty = "bar"
}

class SomeClass{
    var observableProperty: String by Delegates.observable("foo") {
         prop, old, new -> println("old value: $old -> new value: $new")
    }
}
```

Output:

```
old value: foo -> new value: bar
```

Storing properties in a map

- A property can delegate to a `Map`.
- Delegated properties takes their values from this map.
- for `var` property use `MutableMap`

```
fun main() {
    val someClass = SomeClass(mapOf("name" to "Marko", "id" to 1 ))
    println(someClass.name) // print Marko
}

class SomeClass(val map: Map<String, Any?>){
   val id: Int      by map
   val name: String by map
}
```

CHAPTER 10 | Extensions

- Inheritance allow us to extend a given class with new functions and properties in new subclasses.
- Kotlin, however, provides a different extension mechanism which enables us to extend the original class with new functions and properties without using inheritance.
- So, classes that we have or don't have access to their source code can be extended with extension functions and properties.
- Functions and properties added using extension are accessible in the same way as if they were members of the original class.

10.1 | Extension Functions

- To add an extension function to a given class, we add the class name before the function.
- For example, below we are adding a new function `print()` to `String` class.

```
fun String.print(){
    println(this)
}
```

- Now, this `print()` function is available to any string object.

```
 "Hello, world!!!".print()
```

- Extension functions of a superclass will be available to subclasses.

```
 open class Parent
class Child : Parent()

fun Parent.print() { println("Parent") }

fun main() {
    Child().print() // print Parent
}
```

- Extension functions are dispatched statically, that means the static type of the receiver object is used to call the function not the type evaluated at runtime.

- Below, `obj` type is `Parent`, so `Parent.print()` will be called.

```
open class Parent
class Child : Parent()

fun Parent.print() { println("Parent") }
fun Child.print() { println("Child") }

fun main() {
    val obj:Parent = Child()
    obj.print() // prints --> Parent
}
```

- In case, extension function name is already taken by a member function of the class being extended, the member function will be executed.

```
class SomeClass{
    fun print(){ // member function
        println("Memeber function...")
    }
}

fun SomeClass.print():Int{ // extension function
    println("Extension function...")
        return 1
}

fun main() {
    SomeClass().print() // prints -->  Memeber function...
```

- Member functions of a given class can't execute on a `null` receiver.
- Extension function, however, can execute on a nullable receiver with `null` value.

```
class SomeClass{
    fun doSomething(){
        println("member function...")
    }
}

fun SomeClass?.print(){  // extension function
    println("Extension function...")
}
```

```
fun main() {
    val someClass:SomeClass? = null
    someClass.print() // prints --> Extension function...

    someClass!!.doSomething() // throws exception
    someClass?.doSomething() // safe call but does not execute
}
```

10.2 | Extension Properties

- Properties can also be added to a class using extension.
- Extension properties must have accessors or be abstract.
- Extension properties do not have backing field. So, the property cannot hold a value.

```
class Class{
    var _id:Int = 0
}

var Class.id:Int
        get() = _id
        set(value) { _id = value }

fun main() {
    val clazz = Class()
    clazz.id = 5
    print(clazz.id)
}
```

10.3 | More about Extension

- Companion object can also have its own extension function and properties.
- Extensions should be imported if declared outside the calling package.
- Extensions can be declared inside a class for other classes. Qualified `this` can be used to resolve naming conflict if exist.
- Extensions declared as a member inside a class can be overridden in subclass if marked as `open`
- Extensions cannot access `private` members of its receiver.

```
class OtherClass{
    override fun toString() = "OtherClass"
}

class Class {

    // adding extension function to OtherClass
    fun OtherClass.print() {
        println(toString()) // OtherClass.toString()
        println(this@Class.toString()) // Class.toString()
    }

    fun call(otherClass:OtherClass){
        otherClass.print()
    }
    override fun toString() = "Class"
}
```

CHAPTER 11 | Generics

11.1 | Generic Classes

- Kotlin classes can take generic type parameters.
- The actual type of this parameter can be specified when an instance is created.

```
class Container<T>(val t: T){ /* ... */ }
```

- To create an instance of classes with generic type, we need to provide the type of generic parameter.
- If the type can be inferred we may omit the parameterization `<T>`.

```
val container = Container<SomeClass>(SomeClass())

val otherContainer = Container(SomeClass()) // type is inferred
```

`in` and `out`

- We know that assigning an object of a certain type to a reference of its supertype is allowed.

```
interface Shape
class Circle : Shape
class Rectangle : Shape

val shape: Shape = Circle() // this is OK
```

- However, assigning a class with generic parameter of a subtype to a reference with generic parameter of a supertype is not allowed.

```
// this is NOT OK
val contShape:Container<Shape> = Container<Circle>(Circle())
```

- To resolve this issue Kotlin introduced **declaration-site variance**.
- Simply we annotate the generic type parameter `T` with `out` modifier.
- However, the class with the `<out T>` can't have a member that consume `T`, for example a member method cannot take T as parameter.
- Members of class `Container<out T>` can only return `T`.

```
class Container<out T>(val t: T){
    //fun consume(t:T){ } consuming T is not allowed
    fun produdce():T{ /* ... */}  // returing T is allowed
}

// not it is OK
val contShape:Container<Shape> = Container<Circle>(Circle())
```

- The opposite case is also not allowed where a class with generic parameter of supertype is assigned to a reference with generic parameter of subtype.

```
// this is NOT OK yet!
fun display(shape:Container<Shape>){
    val circle: Container<Circle> = shape
}
```

- To resolve this issue Kotlin introduced complementary variance annotation `in`.
- In contrast to `out`, class with generic parameter annotated with `in` can't have members that produce `T`.

```
class Container<in T>(t: T){
    fun consume(t:T){/* ... */}
    // fun produdce(): T { ... } not allowed to return T
}
```

11.2 | Generic Functions

- Functions can take generic type parameters and also return generic type.

```
interface Shape
class Circle : Shape
class Rectangle : Shape

fun <T: Shape> generic(t:T): T{
    return t
}

fun main() {
     generic(Circle())
}
```

11.3 | Generic Constraints

- Generic type can berestricted by generic constraints.
- An upper bound can be specified on generic type so that only subtypes can be used for generic type.
- The upper bound is specified inside the angle brackets as following.

```
fun <T: Shape> generic(t:T){ }
```

- If more than one upper bound is needed we can use `where` clause.

```
fun <T> generic(t:T): T where
        T: Shape,
        T: Any?
    { return t }
```

CHAPTER 12 | Functions and Lambdas

12.1 | Functions Basics

- `fun` keyword is used to declare a function.

```
fun foo(){
    println("foo is called...")
}
```

- A declared function is called by its name.
- Functions declared within a class are called using dot notation.

```
fun main() {
    foo()
    SomeClass().doSomething()  // dot notation
}

class SomeClass{
          fun doSomething(){
        println("doing something...")
    }
}
```

Function parameters

- Functions can have parameters.
- Parameters' types should be declared explicitly.

```
fun sum(a: Int, b: Int) { println(a+b) }
```

- Parameters can have default values.
- Parameters with default values can be omitted when calling the function or overridden with new values.
- Default values should be omitted when overriding the function.

```
fun sum(a: Int = 1, b: Int = 2) { println(a+b) }

fun main() {
    sum() // output: 3 --> a=1, b=4
    sum(3, 4) // output: 7 --> a=3, b=4
}
```

- Parameter names can be used when calling a function.
- Parameters with no default values should be passed as named argument if positioned after ones with default values. this is also true for lambda.

```
fun sum(a: Int = 1, b: Int) { println(a+b) }
```

```
fun main() {
    sum(a = 3, b = 4) // output: 7 --> a=3, b=4
    sum(b = 3) // output: 4 --> a = 1, b =3
}
```

- In case a lambda is the last parameter & preceding default parameters, it can be passed as named argument or outside the parentheses.

```
fun sum(a: Int = 1, lambda1: () -> Unit ,  b: Int, lambda2: () -> Unit) {
    lambda1()
    println(a+b)
    lambda2()
}

fun main() {
    sum(lambda1 = { println("sum is called") } ,
        b = 4) { println("sum is done") }
}
```

Variable number of arguments `vararg`

- A function can have a variable number of arguments using `vararg`.
- Only one parameter can be marked as `vararg`.
- `vararg` is normally the last function parameter. If not, then other parameters should be passed as named argument.
- `vararg` arguments can be passed one-by-one, or as an array with **spread** operator *.

```
fun foo(vararg args:String, a:Int) {
    for (arg in args)
                println(arg)
}

fun main() {
    foo("a","b","c", a = 3)

    val array = arrayOf("A", "B", "C")
    foo(*array, a = 3)
}
```

Single-expression functions

- Function returning single expression can omit curly braces `{}` and define the body after `=`.
- Return type can be omitted too.

```
fun sum(x: Int, y:Int): Int = x + y
```

Return type

- A function return type is `Unit` if it does not return useful value.
- Function returning `Unit` can optionally omit the return type.
- Kotlin does not infer return types for functions with block bodies.
- So, unless returning `Unit`, all function with block bodies shold always sepcify in the return type.

`infix` Function

- Member functions and extension functions with a single parameter that is NOT `varage` and has no default value can be makred as `infix`.
- Function marked with `infix` keyword can be called using infix notation.
- Infix notation all calling a function without dot or parentheses.
- `this` should be used explicitly when infix function is called within current object.

```
// extension function
infix fun Int.gt(x: Int): Boolean { return this > x }

class List {
    var myList: MutableList<String> = mutableListOf<String>()

    infix fun push(s: String) {
        myList.add(s)
    }

    fun addTwice(s: String) {
        this push s // this must be used
        push(s) // this can be omitted
    }

    override fun toString():String{
        return myList.toString();
    }
}

fun main() {
    println(3 gt 6) // flase

    val list = List()
    list.addTwice("A")
    list push "B" // infix function
    println(list) // [A, A, B]

}
```

Function Scope

- Function in Kotlin can be declared as a top level in a file which requires no class.
- Member functions & extension functions are called on their receiver only.
- Local functions are nested function.

```
fun topLevel() { println("Top level...")}

class SomeClass {
        fun member() {
        print("Member calls ")
        topLevel()

        fun local(){
            println("Local...")
```

```
        }
        print("Member calls ")
        local()
    }
}

fun main() {
    SomeClass().member()
}
```

Output:

```
Member calls Top level...
Member calls Local...
```

| Tail Recursive Functions

- A recursive function is a function calling itself.
- Tail recursive is special case of recursion where the last action of a function is a call to itself.
- Tail recursive function are avoiding the risk of stack overflow.
- A Recursive function marked as `tailrec` and meeting the conditions of tail recursive function is converted by the compiler into an optimized loop-based version.

```
fun main() {
    println(sum(5))
}

// sum(N) sums first N integers e.g. sum(3) = 1 + 2 + 3
tailrec fun sum(x:Int, total:Int = 0):Int
    = if(x == 0) total else sum(x-1, total+x)
```

12.2 | Function Types & Higher-Order Functions

- In Kotlin, functions can be stored in variables, passed as arguments to another function and returned from other functions.
- The question now is, how to declare the type of the variable storing a function?

Function types

- Function types is way to define the type of a function using its parameters and return type.

- Special notation is used to define the signature of the functions based on their parameters and return types.
- A Function type is defined by listing the types of the parameters between parentheses `(A, B, ...)` followed by an arrow `->` and ended by the return type `C`.
- For example, `(Int, Int) -> Boolean` is a function type representing all functions that take two arguments of type `Int` and return `Boolean`.
- `gt` function below has a function type `(Int, Int) -> Boolean`

```
fun gt(a: Int, b: Int): Boolean = a > b
```

- Function type with now parameters can be written as following `() -> A` where `A` is a return type.
- Function type can also specify receiver type e.g. `Int.(Int) -> Int` is a function type that is called on a receiver object of type `Int` , takes one parameter of type `Int` and returns `Int`

- `sum` function below has a function type `Int.(Int) -> Int`

```
fun Int.sum(a:Int): Int = this + a
```

- Receiver can interchange with first parameter so `(Int, Int) -> Boolean` can be interchanged with `Int.(Int) -> Boolean`
- `suspend` keyword can be added to the function type to represent `suspend` function e.g. `suspend (Int, Int) -> Boolean`
- Named parameters can be optionally included `(a:Int, b:Int) -> Booelan`
- nullable function type is defined as following `((Int, Int) -> Boolean)?`
- Nested function types is allowed `(Int) -> ((Int) -> Unit)`
- `->` notation is right-associative `(Int) -> (Int) -> Unit` = `(Int) -> ((Int) -> Unit)`
- To improve code readability, function type can be named using `typealiaas` keyword.

```
typealiaas checkType = (Int, Int) -> Boolean
```

Higher-order functions

- Higher-order functions can take functions as parameters or return a function.
- The type of the parameter accepting the function or the return type is declared using **function type**.
- The `compare` function below takes two variable and a function as parameters.

```
fun compare(a:Int, b:Int, f: (Int, Int) -> Boolean): Boolean{
    return f(a,b)
}
```

Instantiating a function type

- Similar to class type, function type can be instantiated.
- There are serval ways to create an instance of a function type.
- For example to pass an instance of function type `(Int, Int) -> Boolean` to `compare()` function “declared above”, we can use the follwoing ways:
  - Lambda `{a, b -> a > b}`

```
    compare(2,3, {a, b -> a > b})
```

- Anonymous function `fun(a:Int, b:Int):Boolean = a > b`

```
    compare(2,3, fun(a:Int, b:Int):Boolean{ return a > b})
```

- Callable reference `::gt`

```
    fun gt(a: Int, b: Int): Boolean = a > b
    compare(2,3, ::gt)
```

- Instance of a class implementing a function type

```
    class GreaterThan: (Int, Int) -> Boolean {
        override operator fun invoke(a: Int, b:Int) = a > b
    }

    compare(2,3, GreaterThan())
```

Calling an instance of a function type

- Instance of function type can be invoked using `invoke(a, b, ...)` function.
- Or directly passing the parameter `(a, b, ...)`
- For receiver type, the receiver should be the first argument.

```
{a:Int, b:Int -> a + b}.invoke(2,3) // 5
{a:Int, b:Int -> a > b}(2,3) // 5

val f: Int.(Int) -> Int = { b -> this + b}
f.invoke(2,3) //5
f(2,3) // 5
2.f(3) //5
```

12.3 | Lambdas

- Lambda is a literal function which means it is not declared but passed as an expression.
- Lambda expression is always surrounded by curly braces `{...}`
- Parameters types are optional.
- Lambda's body goes after the arrow `->`.

```
val sum: (Int, Int) -> Int = {a:Int, b:Int -> a + b}
```

- Which can be written without parameter types.

```
val sum: (Int, Int) -> Int = {a,b -> a + b}
```

- Equivalent function

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

Trailing Lambda

- If lambda is passed as the last parameter of a function, then lambda can be placed outside the function parentheses, this is know as **trailing lambda**.

```
fun compare(a:Int, b:Int, f: (Int, Int) -> Boolean): Boolean{
    return f(a,b)
}
```

- Normal way

```
compare(2,3, { a , b -> a > b})
```

- Or, trailing lambda

```
compare(3,2) { a , b -> a > b }
```

Lambda with single paramete

- Lambda with single parameter can be written in a shorter way. Omit the single parameter, the arrow `->` and use `it` as a reference to the single paramter.

```
fun isEven(a:Int, f: (Int) -> Boolean):Boolean{
    return f(a)
}
```

- Normal way

```
isEven(4){ a -> a % 2 == 0}
```

- Using `it`

```
isEven(4){ it % 2 == 0}
```

Lambda qualified return

- Lambda returns from the enclosing function.
- By default, the last expression of a lambda is implicitly returned.
- Explicit return should be qualified `return@label`

```
//implicit return
isEven(4){ it % 2 == 0}

//explicit return
isEven(4){ return@isEven (it % 2 == 0)}
```

Unused parameter _

- If the function type takes unused parameter, we can replace them with _

```
val p: (Int, Int) -> Unit = { a , _ -> println("a: $a")}
p(3,4) // output a: 3
```

Destructuring in Lambdas

- Lambda parameter with appropriate `componentN` function can be destructed using destructuring declaration.

```
data class SomeClass(val id:String, val name:String)

fun print(c: SomeClass, f: (SomeClass) -> String){
    println(f(c))
}

fun main() {
    print(SomeClass("2","John"), { c -> "id: ${c.id} - name: ${c.name}"})

    // using destructuring declaration
    print(SomeClass("2","John"), { (id, name) -> "id: $id - name: $name"})

}
```

Output:

```
id: 2 - name: John
id: 2 - name: John
```

Anonymous functions

- Anonymous function is a regular function without a name.

```
fun(x: Int, y: Int): Int = x + y
```

12.4 | Inline Functions

- Using Lambda in higher-order functions give us abstraction but introduce run-time overhead.
- inlining a function using `inline` keyword, eliminate runtime overhead imposed by higher-order functions.

- For example, the function below is a non-inlined higher-order function.

```
fun nonInlined(body: () -> Unit) {
    println("before")
    body()
    println("after")
}
```

- When the **nonInlined()** function above is called, it *may* be translated as shown below which include the creation of an instance of the function type.

```
nonInlined{ println("Body is here") }
```

```
println("before")

// overhead of creating an object
object : () -> Unit {
    override operator fun invoke() {
        println("Body is here")
    }
}.invoke()

println("after")
```

- However, when the same function marked as `inline`, the calling to the function will be inlined. So, it will be running as following

```
inline fun nonInlined(body: () -> Unit) {
    println("before")
    body()
    println("after")
}
```

```
println("before")
println("Body is here") // no object created
println("after")
```

Enforcing `noinline`

- If a function is marked as `inline`, we still can enforce no-inlining of a lambda function using `noinline`

```
inline fun nonInlined(noinline body: () -> Unit) {
    println("before")
    body()
    println("after")
}
```

`reified` & `inline`

- Generic type T is not accessible in the body of a generic function because it's only available at compile time but erased at runtime.
- If you create an `inline` function with a `reified` T though, the type of T can be accessed even at runtime.

```
class Class

inline fun <reified T> chekcType(p: Any): Boolean {
    return (p is T)
}

fun main() {
    println(chekcType<Class>(Class())) // true
}
```

CHAPTER 13 | Collections & Sequences

- Collection is a data structure that contains a number of objects of the same type.
- Object in a collection is called an element or item.
- Kotlin Standard Library provides comprehensive set of tools for managing collections.
- Kotlin Standard Library offers generic interfaces, classes,

and functions for creating, populating, and managing collections of any type.

13.1 | Collection

- `Collection<T>` is an interface representing a supertype in collection hierarchy.
- `Collection<T>` is a read-only collection interface.
- `List` & `Set` are subtypes of `Collection<T>`.

| Collection Types

- **List**
  - Ordered elements by indices
  - Elements can occur more than once.
- **Set**
  - Collection of unique elements.
  - Order is not important
- **Map**
  - A set of key-value pairs.
  - Key is unique and map to one value.
  - Values can be duplicates.
  - `Map` is NOT subtype of `Collection<T>`

Read-only vs. Mutable Collection

- Each type of collection has **read-only** and **mutable** version.
- **read-only** for accessing collection elements.
- **mutable** for reading and modifying collection elements.

Creating Collection

- A collection can be created by instantiating an implementation of collection such as `ArrayList`, `HashSet`, `HashMap`, ...
- A more convenient way is using Kotlin STD functions `listOf<T>`,`SetOf<T>`,`mapOf<K,V>`, `mutableListOf<T>`, `mutableSetOf<T>` and `mutableMap<K,V>`.
- There are also functions for creating empty collections: `emptyList()`,`emptySet()` and `emptyMap()`.

```
fun main() {

    // Collection
    print(listOf(1,2,3)) // [1, 2, 3]
    print(mutableSetOf(1,2,3)) // [1, 2, 3]
    print(ArrayList<Int>()) // empty ArrayList

    // List
    val list = listOf(1,2,3)
    val mutableList = mutableListOf(1,2,3)
    mutableList.add(4)
    val arrayList = ArrayList<Int>()
    val emptyList = emptyList<Int>()

    // Set
    val s = setOf(1,2,3)
    val hashSet = HashSet<Int>()
    hashSet.add(4)
    print(hashSet) // [4]

    // Map
    val m = mapOf(1 to "one", 2 to "two")
    val hashMap = HashMap<Int,String>(m)
    hashMap.put(3, "three")
    print(hashMap) // {1=one, 2=two, 3=three}
}

fun print(c:MutableCollection<Any>){
    for(e in c)
            println(e)
}
```

- Collection operations such as `filter()`, `map()`,...etc. create new collection as a returned result from existing ones.

```
val list = listOf(1,2,3)
val result = list.filter{ it > 2} // return elements > 2

println(result) // [3]
```

Copying

- Collection copying functions, such as `toList()`, `toMutableList()`, `toSet()`.

```
val fromList = listOf(1,2,3)
val toList = fromList.toList()
```

- A `List` can be copied to `Set` or vice versa.

```
val fromList = listOf(1,2,3,3)
val toSet = fromList.toSet() // 1,2,3
```

- We can simply create a new variable of collection by assigning an existing one to that variable. However, variables created this way are pointing to the same collection reference i.e. an update to one of them will be reflected to others.

```
val fromList = mutableListOf(1,2,3)
val toList = fromList // toList is copying by reference
toList.add(4)

println(fromList)
```

13.2 | Iterations

- Iterators are used to sequentially traverse elements of collections.
- `Iterable<T>` interface defines functions for iterating elements.
- `Collection<T>` and its sub-types such as `List` and `Set` are inheriting from `Iterable<T>`
- Iterators can be obtained by calling `iterator()` function.
- Obtained iterator is pointing to first element in the collection.
- `next()` function retrieve current element and move pointer to next.

```
fun main() {
    val list = listOf(1,2,3)
    val map = mapOf(1 to "one", 2 to "two")

    val listIterator = list.iterator()
    val mapIterator = map.iterator()

    while(listIterator.hasNext())
            println(listIterator.next())

    while(mapIterator.hasNext())
            println(mapIterator.next().value)
}
```

Output:

```
1
2
3
one
two
```

- `for` loop can be used to iterate over a collection

```
val list = listOf(1,2,3)

for(n in list)
    println(n)

for(i in 0..list.size-1)
    println(list[i])
```

- `forEach()` function for iterating over a collection.
- `it` is used to reference current element.

```
val list = listOf(1,2,3)

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

- `List` has special iterator `ListIterator` supporting bi-directional iteration using `next()` and `previous()`.
- `MutableIterator` for mutable collection allow element removal while iterating.

13.3 | Sequences

- Sequences are another type of elements container.
- Sequences works like iterable but with different processing approach.

```
val numbersSequence = sequenceOf(1,2,3,4)
```

- The first difference between iterables and sequences is that iterables are executed eagerly, they complete and return results.
- In contrast, sequences are executed lazily i.e. actual calling to operations happens only when the result is requested.

```
fun main() {
          val seq = sequenceOf(1,2,3)
    val itr = listOf(1,2,3)

    // sequence executed laziy
    seq.filter{println("Sequence filter: $it"); it > 2}

    // iterable executed eagerly
    itr.filter{println("Iterable filter: $it"); it > 2}
}
```

Output:

```
// sequence filter was not executed since the result was not requested.

Iterable filter: 1
Iterable filter: 2
Iterable filter: 3
```

- Sequences execute operations when result is requested.

```
fun main() {
          val seq = sequenceOf(1,2,3)
    val itr = listOf(1,2,3)

    val result = seq.filter{println("Sequence filter: $it"); it > 2}
    println("Sequence Result: "+result)

    val itrResult = itr.filter{println("Iterable filter: $it"); it > 2}
          println("Iterable Result: "+itrResult)

          println("Sequence Result: "+result.toList()) // result requested
}
```

Output:

```
Sequence Result: kotlin.sequences.FilteringSequence@448139f0
Iterable filter: 1
Iterable filter: 2
Iterable filter: 3
Iterable Result: [3]

Sequence filter: 1
Sequence filter: 2
Sequence filter: 3
Sequence Result: [3]
```

- The second difference between iterable and sequence is that sequence performs all the processing steps one-by-one for every single element.
- Iterable, on the other hand, completes each step for the whole collection and then proceeds to the next step.

```
fun main() {
          val seq = sequenceOf(1,2,3)
    val itr = listOf(1,2,3)

    val result = seq
        .filter{println("Sequence filter: $it"); it > 1}
        .map{ println("Sequence map: $it"); it * 2}


    val itrResult = itr
        .filter{println("Iterable filter: $it"); it > 1}
        .map{ println("Iterable map: $it"); it * 2}

         println("Iterable Result: "+itrResult)
         println("Sequence Result: "+result.toList())
}
```

Output:

```
Iterable filter: 1
Iterable filter: 2
Iterable filter: 3
Iterable map: 2
Iterable map: 3
Iterable Result: [4, 6]

Sequence filter: 1
Sequence filter: 2
Sequence map: 2
Sequence filter: 3
Sequence map: 3
Sequence Result: [4, 6]
```

- The third difference is that, Sequences avoid creating intermediate collection, which will be positively reflected on the processing performance.

Creating a Sequence

- Using `sequenceOf()`

```
val numbersSequence = sequenceOf(1,2,3,4)
```

- Using `asSequence()`, which will convert itrable to sequence.

```
val list = listOf(1,2,3)
val sequence = list.asSequence()
```

- Using `generateSequence()`.
- `generateSequence()` can generate infinite sequence.
- The value of the element is calculated based on lambda function.
- The generation of the sequence stop when a `null` is returned.

```
fun main() {
          val seq1 = generateSequence(0){ it + 1} // return 0,1,2,3,4,...
    println(seq1.take(4).toList()) // [0, 1, 2, 3] extract first 4

    val seq2 = generateSequence("A"){ if(it.length < 3) it+"A" else null}
     println(seq2.take(10).toList()) // [A, AA, AAA] stops when length >=3
}
```

13.4 | Collection Operations

- Kotlin provides built-in functions to perform operations on collections.
- These operations may include filtering a collection, transforming collection content into different types, sorting, grouping or simply retrieving a single element.
- Some of the common operations are explained next.

Retrieving Single Elements

- `elementAt()` function retrieves an element at specific position from `List` and `Set`.
- `get()` and `[]` operators can also be used to access `List` elements at specific position.
- `Map` elements are retrieved by key using `get(key)`, `[key]` or `getValue(key)` which unlike first two will throw exception if key not found.
- `getOrElse()` can be used with `List` and `Map` where value of non-existent elemement is returned from lambda.

```
fun main() {
        val list = listOf(1,5,3,4)
    val unsortedSet = setOf(2,3,1)
    val sortedSet = sortedSetOf(3,2,1)
    val map = sortedMapOf(1 to "one", 2 to "two" ,3  to "three")

    // List
        println(list.elementAt(0)) // 1
    println(list.get(0)) // 1
    println(list.getOrElse(5){it}) // it = 5, passed index
    println(list[0]) // 1

    // Set
    println(unsortedSet.elementAt(0)) // 2
    println(sortedSet.elementAt(0)) // 1

    // Map
    println(map.get(0)) // "one" or null
    println(map.getOrElse(5){"does not exist"}) // "one" or null
    println(map[0]) // "one" or null
    println(map.getvalue(0)) // "one" or exception

}
```

- Safe versions of `elementAt()` are:
  - `elementAtOrNull()`: returns null if index is out of bounds.
  - `elementAtOrElse()`: takes a lambda with index as a single argument which returns result if index is out-of bounds.

```
val list = listOf(1,5,3,4)

println(list.elementAtOrNull(4)) // null
println(list.elementAtOrElse(4){it}) // 4
```

- `first()` & `last()` to retrieve first element and last one.

```
val list = listOf(1,5,3,4)
val unsortedSet = setOf(2,3,1)
val sortedSet = sortedSetOf(3,2,1)

list.first() // 1
unsortedSet.first() // 2
sortedSet.last() // 3
```

- `first()` and `last()` can take a predicate. So, `first()` will return first matching and `last()` will return the last element matching the predicate.
- `firstOrNull()` and `lastOrNull()` are safe versions that return `null` if no match found.

```
val list = listOf(1,5,3,4)

list.first{ it > 2 } // 5
list.last{ it < 4 } // 3

list.firstOrNull{ it > 5 } // null
```

- `find()` is an alias function that works exactly like `firstOrNull()`, while `findLast()` is an alias for `lastOrNull()`

Retrieving collection parts

- Parts of a collection can be retrieved using element indices or size of wanted parts.
- `slice()` function returns a list of elements with given indices.
- `slice()` throws an exception if an index is out of bound.

```
val list = listOf(1,5,3,4,2)
println(list.slice(0..2)) // [1, 5, 3]
println(list.slice(setOf(0,2))) //[1,3]
```

- `List` provides `subList(s,e)` function to retrieve parts by indices.

```
val list = listOf(1,5,3,4,2)
println(numbers.subList(0, 2)) // [1, 5]
```

- `take(n)` returns *n* elements starting from the first one.
- `takeLast(n)` returns the last *n* elements.
- `drop(n)` returns all except first *n* elements.
- `dropLast(n)` returns all elements except last *n* elements

```
val list = listOf(1,5,3,4,2)
println(list.take(2)) // [1, 5]
println(list.dropLast(2)) // [1, 5, 3]
```

- Taking and dropping elements can also be specified by predicates.
- `takeWhile()` keeps taking elements from the beginning of a collection until the predicate is `false`.
- `takeLastWhile()` keeps taking elements from the end of a collection until the predicate is `false`.
- `dropWhile()` keeps dropping elements from the beginning of a collection until the predicate is `false`
- `dropLastWhile()` keeps drop elements from the end of the collection until the predicate is `false`

```
val list = listOf(1,5,3,4,2)

println(list.takeWhile { true }) // [1, 5, 3, 4, 2] i.e. take all
println(list.takeLastWhile { it < 3}) // [2]
println(list.dropWhile { it != 4 }) // [4, 2]
println(list.dropLastWhile { true }) // [] i.e. drop all
```

- Chunking a collection into multiple collections of given size can be done using `chunked(n)`.
- All chunks are equal in size. However, last may be smaller if remaining elements are less than the *n*.
- `chunked(){ lambda }` can also take a lambda

```
val list = listOf(1,5,3,4,2)

println(list.chunked(2)) // [[1, 5], [3, 4], [2]]
println(list.chunked(2){it.sum()}) // [6, 7, 2]
```

- A collection can also be broken into parts using `windowed(n)`.
- `windowed(n)` returns lists of size *n* created by sliding a window over the collection.
- By default, `windowed(n)` has step of size 1 which will return the first n elements starting with element of index 0, then it will return the next n elements starting from element of index 1 and so on.
- step of the function can be changed using `step` parameter.
- The function also provides an option to keep the window sliding till the end using `partialWindows`.

```
val list = listOf(1,5,3,4,2)

println(list.windowed(3))
// [[1, 5, 3], [5, 3, 4], [3, 4, 2]]

println(list.windowed(3, partialWindows=true))
// [[1, 5, 3], [5, 3, 4], [3, 4, 2], [4, 2], [2]]
```

Checking existence

- `contains()` function checks if a collection include an element.
- `in` operator works like `contains()`
- To test the existence of more than one elements use `containsAll()`
- `isEmpty()` and `isNotEmpty()` are used to check if the collection has any element.

```
val list = listOf(1,5,3,4)
val unsortedSet = setOf(2,3,1)
val map = sortedMapOf(1 to "one", 2 to "two" ,3  to "three")

println(list.contains(2)) // false
println(3 in list) // true
println(unsortedSet.containsAll(setOf(2,3))) // true
println(map.isEmpty()) // false
```

Filtering

- `filter()` is the most used function for filtering a collection.
- `filter()` takes a lambda as a filtering condition "predicate" and returns collection matching the condition.
- Since `filter()` takes a lambda of a single parameter we can implicitly reference it using `it` which is a reference to the element.

```
val list = listOf(1,2,3,4)
val filtered = list.filter{ it  > 2 }
println(filtered) // [3, 4]
```

- `filterNot()` is used to return the elements not matching the predicate.

```
val list = listOf(1,2,3,4)
val filtered = list.filterNot{ it  > 2 }
println(filtered) // [1, 2]
```

- `filterIndexed()` gives you access to element index.

```
val list = listOf(1,2,3,4)
val filtered = list.filterIndexed{ index, value -> index > 1 && value < 4}
println(filtered) // [3]
```

Partitioning

- Filters only return matching or not matching elements. But what if we want to return both?!
- `partition()` is a way to partition a collection of elements into two separate collections.
- The return type is `Pair` of `List` one for matching elements and one for not matching ones.

```
val list = listOf(1,2,3,4)
val partitions = list.partition{ it  > 2 }
println(partitions) // ([3, 4], [1, 2])
```

Testing predicates

- Sometimes we want to test if one, all or no elements in a collection matching a given predicate.
- To test such cases, Kotlin provides `any()`, `none()` and `all()` functions.
- `any()` function returns true if at least one element matches the predicate.
- `none()` function returns true if all elements are NOT matching the predicate.
- `all()` function returns true if all elements are matching the predicate.

```
fun main() {
        val list = listOf(1,2,3,4)
        println(list.any{ it  > 4 }) // false
    println(list.none{ it  > 4 }) // true
    println(list.all{ it  > 4 }) // false
}
```

`plus` and `minus` Operators

- A handy way to add or remove an element or collection of elements from a collection is by using `+` and `-` operators.
- However, the result of adding or subtracting elements is a new *read-only* collection.
- The second operand can be a single element or a collection.
- When second operand is a collection, all occurrences of matching elements are removed.

```
val list = listOf(1,5,3,4,3)

list + 2 // [1, 5, 3, 4, 3, 2] 2 added
list - 3 // [1, 5, 4, 3] one "3" removed
list - listOf(3) // [1, 5, 4] all "3"s removed
```

Mapping

- Mapping takes elements of an existing collection, transforms them and produce a new collection.
- The transformation is defined using lambda function.
- `map()` takes a lambda that uses element as single parameter to apply mapping.
- `mapIndexed()` takes a lambda with element value and index passed as parameters.

```
val list = listOf(1,5,3,4,3)

list.map{ it + 2 } // [3, 7, 5, 6, 5]
list.mapIndexed{ index, value -> index } // [0, 1, 2, 3, 4]
```

- To remove element transormed to `null`, `mapNotNull()` and `mapIndexedNotNull()` can be used.

```
// transform 3 to null
val mappedNotNull = list.mapNotNull{ if(it==3) null else it} // [1, 5, 4]
```

- Map can be mapped by keys using `mapKeys()` or by values using `mapValues()`

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3, "d" to 4)

map.mapKeys { it.key.toUpperCase() } // {A=1, B=2, C=3, D=4}
map.mapValues { it.value*2 } // {a=2, b=4, c=6, d=8}
```

Zipping

- Zipping takes elements with the same position from two collections and build a pair out of them.
- The result of zipping is a list of elements of type `Pair`.
- `zip()` can be called as `infix` function.

```
val num = listOf(1,2,3,4)
val str = listOf("one", "two", "three")

num.zip(str) // [(1, one), (2, two), (3, three)]
num zip str // [(1, one), (2, two), (3, three)]
```

- `zip()` can take a lambda that take two elements as parameters. The result in this case is a list of the returned type.

```
num.zip(str){ n, s -> s.toUpperCase()+":"+n} // [ONE:1, TWO:2, THREE:3]
```

- Unzipping is also allowed for a list of `Pair`s.
- `unzip()` returns two lists from the original list.

```
val pairs = listOf(Pair(1,"A"), Pair(2,"B"))

pairs.unzip() // ([1, 2], [A, B])
```

Association

- Association is a way to build a map from a collection and associated values.
- `associateWith()` creates `Map` where the collection elements are the keys.
- `associateBy()` creates `Map` where the collection elements are the values.
- `associate()` allowing creating a Map where keys and values can be produced from the collection.

```
val list = listOf("one", "two", "three", "four")
println(list.associateWith { it.first().toUpperCase() })
println(list.associateBy { it.first().toUpperCase() })
println(list.associate{n-> Pair(n.toUpperCase(), n)})
```

Output:

```
{one=O, two=T, three=T, four=F}
{O=one, T=three, F=four}
{ONE=one, TWO=two, THREE=three, FOUR=four}
```

Flattening

- Flattening takes a collection of collections and return a single list of elements.
- `flatten()` function extracts elements from nested collections into a single list.

```
val sets = listOf(setOf(1, 2), setOf(3, 4), setOf(1))
println(sets.flatten()) // [1, 2, 3, 4, 1]
```

- To map elements of nested collection before flattening use `flatMap()`

```
println(sets.flatMap{s -> s.map{ it * 2 }})        // [2, 4, 6, 8, 2]
```

Grouping

- Grouping return a map of grouped elements.
- Keys are the lambda results while values are the matching list of elements.
- `groupBy()` function takes a lambda that is applied on each element.
- The result of the lambda is a key.
- Elements returning the same key are grouped together.
- `groupBy()` function can take a second lambda parameter that transform elements values.

```
// group elements by the remainder of dividing element by 2
// if remainder = 0 then element is Even i.e. 0,2,4,6,...
// if remainder = 1 then element is Odd i.e. 1,3,5,...

val list = listOf(1, 2, 3, 4, 5)
println(list.groupBy { it%2 }) // {1=[1, 3, 5], 0=[2, 4]}
println(list.groupBy(keySelector = { if(it%2==0) "Even" else "Odd" },
                        valueTransform = { it })) // {Odd=[2, 6, 10], Even=[4, 8]}
```

- `groupingBy()` is another grouping function that does the grouping and then apply operation to all groups.
- `groupingBy()` is a lazy function that return an instance of `Grouping` type.
- `Grouping` instance does not execute, it waits for some operation to be executed on top of it.
- `Grouping` can execute the following operations:
  - `eachCount()`: counts number of elements in each group.
  - `fold()` & `reduce()` applies a lambda on each group and return accumulated results.
  - `aggregate()`: is a generic way to perform a custom operation on a `Grouping`.

```
val list = listOf(1, 2, 3, 4, 5)

println(list.groupBy{ if(it%2==0) "Even" else "Odd" })
println(list.groupingBy{ if(it%2==0) "Even" else "Odd" }.eachCount())

println(list.groupingBy{ if(it%2==0) "Even" else "Odd" }
                .reduce{ key, total, e -> total + e})

println(list.groupingBy{ if(it%2==0) "Even" else "Odd" }
                .fold(2){total, e -> total + e})

println(list.groupingBy{ if(it%2==0) "Even" else "Odd" }
                .aggregate{ k, total: Int? , e, b ->
                    println("k=$k e=$e total=$total b=$b");
                    if(b) e else total?.plus(e)
                })
```

Ordering

`Comparable`, `Comparator` and `compareBy()`

- Collection element should implement `Comparable` interface to define ordering logic.
- Most built-in types are implementing `Comparable`.
- User defined class should implement `Comparable` to define a natural order.

```
class Person(val age:Int) : Comparable<Person>{

    override fun compareTo(other: Person): Int{
        return this.age - other.age
    }
}

println(Person(48) < Person(30)) // false
```

- Classes that are not subtype of `Comparable` can be sorted using `Comparator`

```
class Person(val age:Int) {
    override fun toString():String{
        return "Person:$age"
    }
}

class MyComparator : Comparator<Person>{
    override fun compare(a: Person, b:Person ): Int{
        return a.age - b.age
    }
}

val myComparator = MyComparator()
println(listOf(Person(48), Person(30)).sortedWith(myComparator))

// [Person:30, Person:48]
```

- `Comparator` can be defined as anonymous class.

```
val myComparator = Comparator { a: Person, b: Person ->  a.age - b.age }
```

- Or, even shorter way using `compareBy()` function

```
println(listOf(Person(48), Person(30)).sortedWith(compareBy{ it.age }))
```

Natural order

- `sorted()` and `sortedDescending()` return a sorted collection in ascending or descending sequence.
- Elements should be of type `Comparable`

```
 val list = listOf(1, 5, 3, 4, 2)
println(list.sorted()) // [1, 2, 3, 4, 5]
println(list.sortedDescending()) // [5, 4, 3, 2, 1]
```

Custom orders

`sortedBy()` and `sortedByDescending()` are used to sort non-Comparable elements.

- The functions map elements into `Comparable` values and sort them in natural order.

`sortedWith()` can take a `Comparator`.

```
val list = listOf("one", "two", "three", "four")

// sort by second letter
println(list.sortedBy { it[1] } ) // [three, one, four, two]
println(list.sortedWith(compareBy{it[1]}) ) // [three, one, four, two]
```

Reverse & random order

- `reverse()` returns the current ordering of a collection in reverse order.
- `shuffled()` returns a new `List` with randomly ordered elements.

```
val list = listOf(1, 5, 3, 4, 2)
println(list.reversed()) // [2, 4, 3, 5, 1]
println(list.shuffled()) // order changes randomly
```

Aggregation

- Aggregate operations return a single value from a collection.
- Some common operations are `sum()`, `min()`, `max()`, `average()`, `count()`

```
val list = listOf(1, 5, 3, 4, 2)

println("Sum: ${list.sum()}") // Sum: 15
println("Count: ${list.count()}") // Count: 5
println("Max: ${list.max()}") // Max: 5
println("Average: ${list.average()}") // Average: 3.0
```

Fold & reduce

- `fold()` and `reduce()` are generic aggregation functions that can be used to define a custom aggregate operation.
- Both functions take a lambda as an operation. The operation takes two arguments, accumulated value and current element.
- `fold()` works like `reduce()` except that `fold(n)` takes an initial value.

```
val list = listOf(1, 5, 3, 4, 2)

 val list = listOf(1, 5, 3, 4, 2)

println("Sum by reduce: "+list.reduce{total, e ->
    println("total=$total e=$e");
    total + e})

println("Sum by fold: "+list.fold(2){total, e ->
    println("total=$total e=$e");
    total + e})
```

Output:

```
total=1 e=5
total=6 e=3
total=9 e=4
total=13 e=2
Sum by reduce: 15

total=2 e=1
total=3 e=5
total=8 e=3
total=11 e=4
total=15 e=2
Sum by fold: 17
```

- `fold()` and `reduce()` aggregate elements starting from the left to right.
- To start aggregation from the right use `foldRight()` and `reduceRight()`
- To element index as a parameter to the lambda operation use `foldIndexed()` and `reducedIndexed()`

```
 val list = listOf(1, 5, 3, 4, 2)

println("Sum by reduce: "+list.reduceIndexed{i, total, e ->
    println("index=$i total=$total e=$e");
    total + e + i})
```

Write Operations

- Mutable collection can be changed by adding, removing and updating elements.

Adding elements

- `add()` add an element to `List` or `Set`

```
val list = mutableListOf(1,5,3,4,2)
list.add(5)
println(list) // [1, 5, 3, 4, 2, 5]

val s = mutableSetOf(1,5,3,4,2)
s.add(5)
println(s) // [1, 5, 3, 4, 2]
```

- A collection of elements can also be added to an existing collection using `addAll()`
- By default, `addAll()` adds the new collection to the end of the current one. However, we can specify the location too.

```
val list = mutableListOf(1,5,3,4,2)
list.addAll(listOf(0,7))
println(list) // [1, 5, 3, 4, 2, 0, 7]

list.addAll(0, listOf(0,7))
println(list) // [0, 7, 1, 5, 3, 4, 2, 0, 7]
```

- `plusAssign (+=)` operator can be used to add elements.

```
val list = mutableListOf(1,5,3,4,2)
list += listOf(0,7)
println(list) //  [1, 5, 3, 4, 2, 0, 7]
```

- `Map` has its own adding function.
- `put()` and `putAll()` are used to add elements to a map.
- `plusAssign (+=)` operator can be used to add entries to a map.
- `[]` is an alias operator for `put()`

```
val map = mutableMapOf("one" to 1, "two" to 2)
map.put("three", 3)
map["four"]= 4
println(map) // {one=1, two=2, three=3, four=4}
```

Removing elements

- `remove()` removes a single matching element from the collection.
- `removeAll()` remove all occurrences of passed set of elements.
- `clear()` will remove all elements.
- `minusAssign (-=)` operator can be used to remove elements.

```
val list = mutableListOf(1,5,3,4,2,4)
list.removeAll(setOf(4))
println(list) \\ [1, 5, 3, 2]
list.clear()
println(list) \\ []
```

- `Map`'s `remove()` function takes a key or whole key-value to remove a map entry.

```
val map = mutableMapOf("one" to 1, "two" to 2)
map.remove("one")
println(map) // {two=2}
```

- To remove an entry by values, call `remove()` function on map's values.

```
val map = mutableMapOf("one" to 1, "two" to 2)
map.values.remove(1)
println(map) // {two=2}
```

CHAPTER 14 | Coroutines

- Coroutines are light-weight threads that enable you to write asynchronous code in a sequential style.
- Kotlin Coroutines is not part of the standard library.
- `kotlinx-coroutines-core` should be added as a dependency to your project.

14.1 | Blocking vs. non-Blocking

- A blocking call to a function will block the parent thread until the function is complete.

```
fun main(){
    runForever() // this function blocks the thread forever!
    println("Hello!") // not reachable line!
}

fun runForever(){
    while (true){ }
}
```

- On the other hand, calling a non-block function on a thread does not necessarily block that thread. It can be delegated to another thread in case of heavy operation or it can suspend its execution when e.g. waiting for a result. Suspension will free the thread for other function.

```
fun main(){

    thread {
        runForever() // running on different thread so main is not blocked
    }

    println("Hello!") // "Hello!" will be printed!
}

fun runForever(){
    while (true){ }
}
```

14.2 | Coroutine Basics

- Coroutines are light-weight threads running in a context `CoroutineContext` of some **Coroutine Scope**.

Coroutine scope

- **Coroutine Scope** specifies the lifetime of the coroutines.
- **Coroutine Scope** is used to start a coroutine.
- `GlobalScope` is a coroutine scope which runs on the whole application lifetime.

Coroutine builders

- **Coroutine builders** are functions used to run coroutines.
- `launch()`, `async()` & `runBlocking` are coroutine builder provided by Kotlin.
- `launch()` function returns an object of type `Job` that can be used to cancel the running coroutine.

```
1  import kotlinx.coroutines.*
2
3  fun main(){
4      val job = GlobalScope.launch {  // GlobalScope
5          runForever() // blocking funcion
6          println("Me Too!")  // will NOT be printed
7      }
8
9      println("Hello!")  // will be printed
10
11 }
12
13 fun runForever(){
14     while (true){ }
15 }
```

Suspending Function

- Suspending functions are regular functions marked with `suspend`.
- Suspending functions may suspend the execution of the current coroutine but does not necessary block the thread.
- Suspending functions do NOT have special return types.
- Suspending functions should be called only from a coroutine or another suspend function.

```
suspend fun delay(time:Long): Unit {...}
```

```
import kotlinx.coroutines.*
fun main() = runBlocking{ // this: CoroutineScop

    val job = launch {  // launch a coroutine
        runForever() // when suspends, anther coroutine will run
    }
    launch {  // launch another coroutine
        println("Me Too!")
    }

    println("Hello!")
    job.cancel()
}

suspend fun runForever(){
    while (true){
        delay(2000L)  // delay is suspend function
    }
}
```

CoroutineDispatcher

- Coroutine context include a coroutine dispatcher.
- Coroutine dispatcher specifies the thread or threads used by a coroutine for its execution.
- Dispatcher can be explicitly specified by an optional parameter of type `CoroutineContext` passed to coroutine builder such as `launch()` and `async()`
- Kotlin provides standard implementation for `CoroutineDispatcher`
  - **`Dispatchers.Default`**: is the default dispatcher is used by coroutine builders if no dispatcher is specified. Also, default dispatcher is the appropriate choice for heavy operations that consume CPU resources.
  - **`Dispatchers.IO`** is used for IO intensive blocking operations.
  - **`Dispatchers.Main`** is used to run a coroutine on the main thread.

```
import kotlinx.coroutines.*
fun main() = runBlocking{ // this: CoroutineScop

    val job = launch {Dispatchers.Default  // launch a coroutine
        runForever() // when suspends, anther coroutine will run
    }
    launch {Dispatchers.Main  // launch another coroutine in Main thread
        println("Me Too!")
    }
```

```
10
11      println("Hello!")
12      job.cancel()
13  }
14
15  suspend fun runForever(){
16      while (true){
17          delay(2000L)  // delay is suspend function
18      }
19  }
```