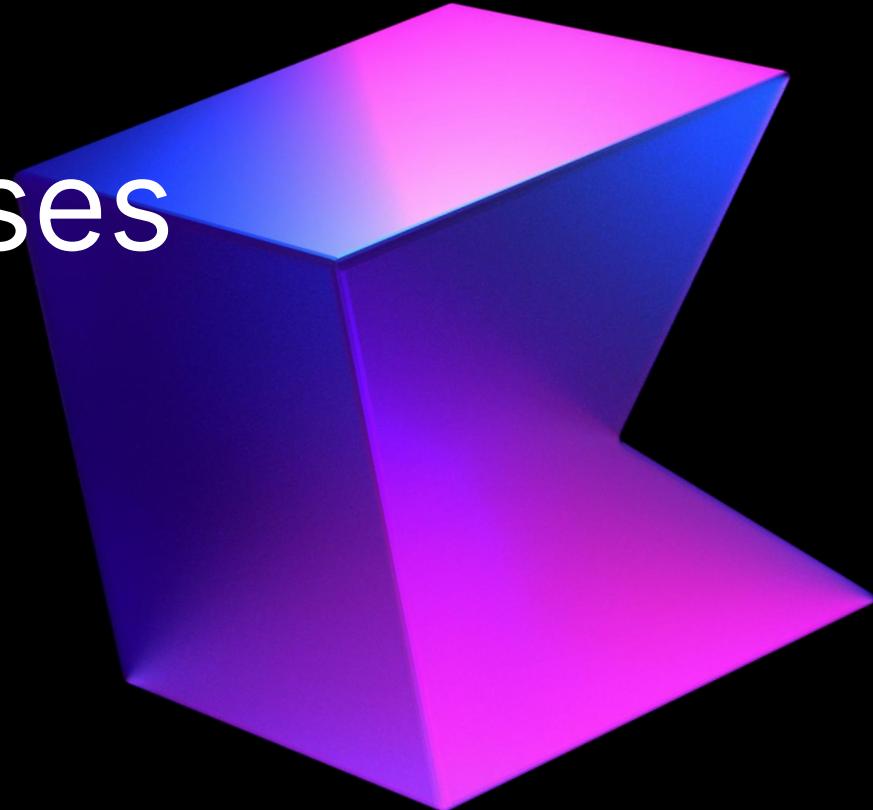


# Kotlin Classes



# Classes in Kotlin

Kotlin offers variety of classes which makes developing applications much easier.  
You can find

- **Normal classes**
- **Abstract classes**
- Data classes
- Enum classes
- Sealed classes
- Singletons (Object declarations)
- Inner classes
- Nested classes
- Anonymous classes (Object expressions)

# Data classes

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

The compiler automatically derives:

- `equals()` and `hashCode()`
- `toString()` of the form `User(name=John, age=42)`
- `componentN()` `functions` corresponding to the properties in their order of declaration.
- `copy()` to copy an object,  
allowing you to alter some of its properties while keeping the rest unchanged

The standard library provides the `Pair` and `Triple` classes,  
but named **data classes** are a much better design choice.

# Data classes

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

fun main() {
    val user = User("Lily", 20)
    println(user) //this will implicitly call toString()
    //Got a copy of all other info except for the name
    val anotherUser: User = user.copy(name="Sally")
    println(anotherUser)
}
```

The output: ➔

User(name=Lily, age=20)

User(name=Sally, age=20)

# Data Classes

- A data class constructor **needs to have at least one property declared.**
- All constructor parameters need to be marked as either **val** or **var**.
- A data class **can NOT be abstract, open, sealed, or inner.**
- Data classes can have body and extra methods. If needed.

# Enum classes

Enum is short for enumeration and refers to a set of predefined constants.

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST  
}
```

Each enum constant is an object.

Each enum is an instance of the enum class separated by a comma, thus it can be initialized as:

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

Enum classes can have methods or even implement interfaces.

# Enum classes Usage

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST; // ; is needed if you intend to add functions inside the enum  
  
    //To Define function inside the Enum  
    fun printDirection(){  
        println("The Chosen direction is ${name.lowercase()})  
    }  
}  
  
fun main() {  
    val direction = Direction.NORTH  
    direction.printDirection()  
}
```

# Creating Singleton in Java

Singleton is a popular **creational design pattern** used to create a single instance of the class during the app's lifetime.

```
public class MySingletonClass {  
  
    //1. private constructor  
    private MySingletonClass(){  }  
  
    //private static reference to the same type of the class  
    private static MySingletonClass instance = null;  
  
    //Create a public static method to check whether the instance is null or not  
    public static MySingletonClass getInstance(){  
        if(instance == null){  
            instance = new MySingletonClass();  
        }  
        return instance;  
    }  
}
```

# Kotlin singleton (Object declarations)

Kotlin provides an easy and concise way to make a singleton.

Using the **object** keyword.

Object declaration (Singleton) is a class that will have a single instance.

As a class, it can extend another class or implement any number of interfaces as needed.

Object declaration may contain initializer block(s), properties, and methods.

Unlike normal classes you can define constants using the **const** keyword.

Constructors are not allowed in the object.

# Kotlin singleton (Object declarations)

```
object DataAccessLayer {  
    fun updateEmployeeData(employee: Employee) {  
        // ...  
    }  
  
    fun removeEmployee(id: String){  
        // ...  
    }  
}  
  
DataAccessLayer.updateEmployeeData(...)
```

# Companion objects

- An ***object declaration*** inside a class can be marked with the ***companion*** keyword.
- The class can contain **only one companion object**
- Companion objects are like static members:
  - The Factory Method
  - Constants
  - Etc.
- Visibility modifiers are applicable.

# Companion objects – Example

```
class Complex(var real: Int, var imaginary: Int){  
  
    companion object{  
        fun printComplex(comp: Complex){  
            println("Current complex is ${comp.real} ${comp.imaginary} i")  
        }  
    }  
}  
  
fun main(){  
    val c1 = Complex(1,2)  
    Complex.printComplex(c1)  
}
```

# Nested Class

In Kotlin, you can declare a class in another class.

Doing so can be useful for **encapsulating a helper class** or placing the code closer to where it's used.

Kotlin's nested classes don't have access to the outer class instance.

```
class Outer {  
    private val num1: Int = 1  
    class Nested {  
        fun meth1(): Int { return 2 } //num1 is NOT accessible  
    }  
}  
val nestedObj = Outer.Nested()  
nestedObj.meth1() // = 2
```

# Inner Classes

A nested class marked as **inner** can access the members of its outer class.  
Inner classes carry a reference to an object of an outer class.

```
class Outer {  
    private val num1: Int = 1  
    inner class Inner {  
        fun foo(): Int { return num1 } //num1 is accessible  
    }  
}  
  
val outerObject = Outer()  
  
val innerObject = outerObject.Inner()  
  
outerObject.foo() // = 1
```

# Inner Classes

From inside the inner class, we can refer to the current instance of the outer class using **this@OuterClassName**

```
class Outer {  
    private val num1: Int = 1  
    inner class Inner {  
        private val num1: Int = 3  
  
        fun foo(): Int {  
            return this@Outer.num1 // this will return 1  
        }  
    }  
}
```

# Nested & Inner Classes

	<b>Nested Class</b>	<b>Inner Class</b>
Reference to Outer Class	No reference to outer class (static).	Has an implicit reference to the outer class.
Usage	When the inner class <b>doesn't need</b> access to the outer class.	When the inner class <b>needs</b> access to the outer class's members.
Syntax	<pre>class Outer {     class Nested { ... } }</pre>	<pre>class Outer {     inner class Inner { ... } }</pre>
Memory	<b>More</b> memory-efficient, doesn't carry a reference to outer class	Slightly <b>less</b> efficient, as it carries a reference to the outer class instance.

# Anonymous Inner class (Object Expression)

The **object** keyword can be used not only for declaring named **singleton-like objects**, but also for declaring **anonymous objects**.

Anonymous objects replace Java's use of anonymous inner classes.

Kotlin anonymous object can implement multiple interfaces or no interfaces.

Unlike object declarations, **anonymous objects aren't singletons**.

Every time an object expression is executed, a new instance of the object is created.

Anonymous object **must** either implement interface(s) or extend a class, or do both.

# Anonymous Inner class (Object Expression)

Anonymous object  
implement an Interface

```
val runnable : Runnable = object : Runnable{
    override fun run() {
        println("Hello form Runnable..")
    }
}
val firstThread : Thread = Thread(runnable)
firstThread.start()
```

Anonymous object  
extends a class

```
val secondThread : Thread = object : Thread(){
    override fun run() {
        println("Hello from Thread")
    }
}
secondThread.start()
```

# Lab

```
class Person{  
    name  
    Address  
    class Address{  
        streetName  
        city  
        building  
    }  
}  
enum building {  
    villa  
    apartment  
}
```

# Today's Challenge

