Lesson 4:

Implementing a quiz

Instructor: Ahmet Geymen

About this lesson

- Lesson 4:
  - More Kotlin fundamentals
    - Inheritance
    - Generics
    - Objects and Extensions
    - Collections
- Workshop: More Kotlin fundamentals

Get started

More Kotlin fundamentals

Inheritance

Inheritance

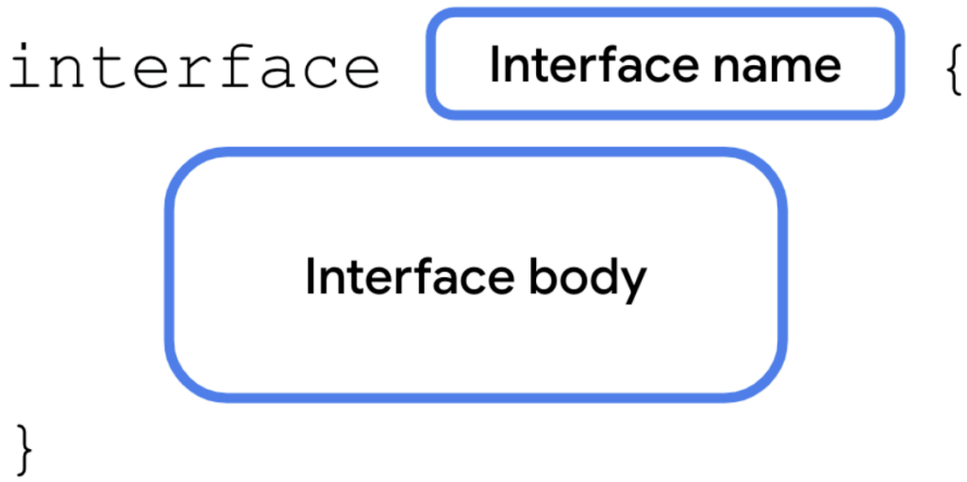
- Kotlin has single-parent class inheritance
- Each class has exactly one parent class, called a superclass
- Each subclass inherits all members of its superclass including ones that the superclass itself has inherited
- All classes in Kotlin have a common superclass, `Any`

If you don't want to be limited by only inheriting a single class, you can define an interface since you can implement as many of those as you want.

Interfaces

- Provide a contract all implementing classes must adhere to
- Can contain method signatures and property names
- Can derive from other interfaces

Interfaces



Interfaces

```
interface  Interface name  {
    Interface body
}

class  Class name  :  Interface name  {
    Class body
}
```

Interface example

```
interface Shape {
    fun computeArea() : Double
}

class Circle(val radius:Double) : Shape {
    override fun computeArea() = Math.PI * radius * radius
}

val c = Circle(3.0)
println(c.computeArea())
=> 28.274333882308138
```

Extending a class

To extend a class:

- Create a new class that uses an existing class as its core (subclass)
- Add functionality to a class without creating a new one (extension functions)

Creating a new class

- Kotlin classes by default are not subclassable
- Use `open` keyword to allow subclassing
- Properties and functions are redefined with the `override` keyword

Classes are final by default

Declare a class

```
class A
```

Try to subclass A

```
class B : A

=>Error: A is final and cannot be inherited from
```

Use open keyword

Use `open` to declare a class so that it can be subclassed.

Declare a class

```
open class C
```

Subclass from C

```
class D : C
```

Overriding

- Must use `open` for properties and methods that can be overridden (otherwise you get compiler error)
- Must use `override` when overriding properties and methods
- Something marked `override` can be overridden in subclasses (unless marked `final`)

Abstract classes

- Class is marked as `abstract`
- Cannot be instantiated, must be subclassed
- Similar to an interface with the added the ability to store state
- Properties and functions marked with `abstract` must be overridden
- Can include non-abstract properties and functions

Example abstract classes

```
abstract class Food {
    abstract val kcal : Int
    abstract val name : String
    fun consume() = println("I'm eating ${name}")
}
class Pizza() : Food() {
    override val kcal = 600
    override val name = "Pizza"
}
fun main() {
    Pizza().consume()    // "I'm eating Pizza"
}
```

When to use each

- Defining a broad spectrum of behavior or types? Consider an interface.
- Will the behavior be specific to that type? Consider a class.
- Need to inherit from multiple classes? Consider refactoring code to see if some behavior can be isolated into an interface.
- Want to leave some properties / methods abstract to be defined by subclasses? Consider an abstract class.
- You can extend only one class, but implement one or more interfaces.

Data classes

Data class

- Special class that exists just to store a set of data
- Mark the class with the `data` keyword
- Generates `toString()`, `equals()`, `hashCode()`, `copy()` methods, and destructuring operators

A data class needs to have at least one parameter in its constructor, and all constructor parameters must be marked with val or var

Data class

```
data class class name ( ... )
```

Data class example

Define the data class:

```
data class Player(val name: String, val score: Int)
```

Use the data class:

```
val firstPlayer = Player("Lauren", 10)
println(firstPlayer)

=> Player(name=Lauren, score=10)
```

Pair and Triple

- `Pair` and `Triple` are predefined data classes that store 2 or 3 pieces of data respectively

- Access variables with `.first`, `.second`, `.third` respectively

- Usually named data classes are a better option (more meaningful names for your use case)

Pair and Triple example

```
val bookAuthor = Pair("Harry Potter", "J.K. Rowling")
println(bookAuthor)

=> (Harry Potter, J.K. Rowling)
```

```
val bookAuthorYear = Triple(
   "Harry Potter”,
   "J.K. Rowling”,
   1997
)
println(bookAuthorYear)
println(bookAuthorYear.third)

=> (Harry Potter, J.K. Rowling, 1997)
    1997
```

Pair to

`Pair`'s special `to` variant lets you omit parentheses and periods (infix function).

It allows for more readable code

```
val bookAuth1 = "Harry Potter".to("J. K. Rowling")
val bookAuth2 = "Harry Potter" to "J. K. Rowling"
=> bookAuth1 and bookAuth2 are Pair (Harry Potter, J. K.
Rowling)
```

Also used in collections like Map and HashMap

```
val map = mapOf(1 to "x", 2 to "y", 3 to "zz")
=> map of Int to String {1=x, 2=y, 3=zz}
```

Enum classes

User-defined data type for a set of named values

- Use `this` to require instances be one of several constant values
- The constant value is, by default, not visible to you
- Use `enum` before the `class` keyword

Enum class

```
enum class [enum name] {
    [Case 1], [Case 2], [Case 3]
}
```

Enum class example

Define directions

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

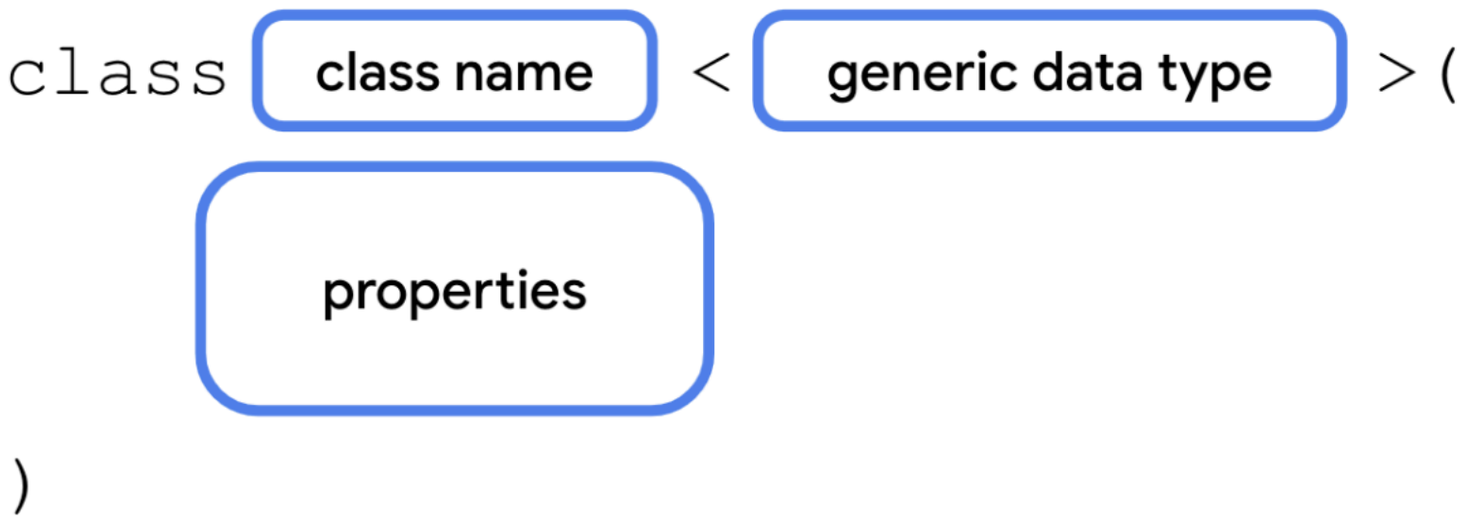
Define an `enum` with red, green, and blue colors.

```
enum class Color(val r: Int, val g: Int, val b: Int) {
    RED(255, 0, 0), GREEN(0, 255, 0), BLUE(0, 0, 255)
}

println("" + Color.RED.r + " " + Color.RED.g + " " + Color.RED.b)
=> 255 0 0
```

Generics

Generics

- Create classes or functions that are parameterized by type
- Allow a data type, such as a class, to specify an unknown placeholder data type that can be used with its properties and methods.



Generics

- The placeholder name can then be used wherever you use a real data type within the class, such as for a property.

```
class [class name]<[generic data type]>(
    val [property name]:[generic data type]
)
```

Generics

- The placeholder name can then be used wherever you use a real data type within the class, such as for a property.

```
class [class name]<[generic data type]>(
    val [property name]:[generic data type]
)
```

```
val [instance name] = [class name]<[generic data type]>([parameters])
```

Generics

- The placeholder name can then be used wherever you use a real data type within the class, such as for a property.

```
class [class name] < [generic data type] >(
    val [property name] : [generic data type]
)
```

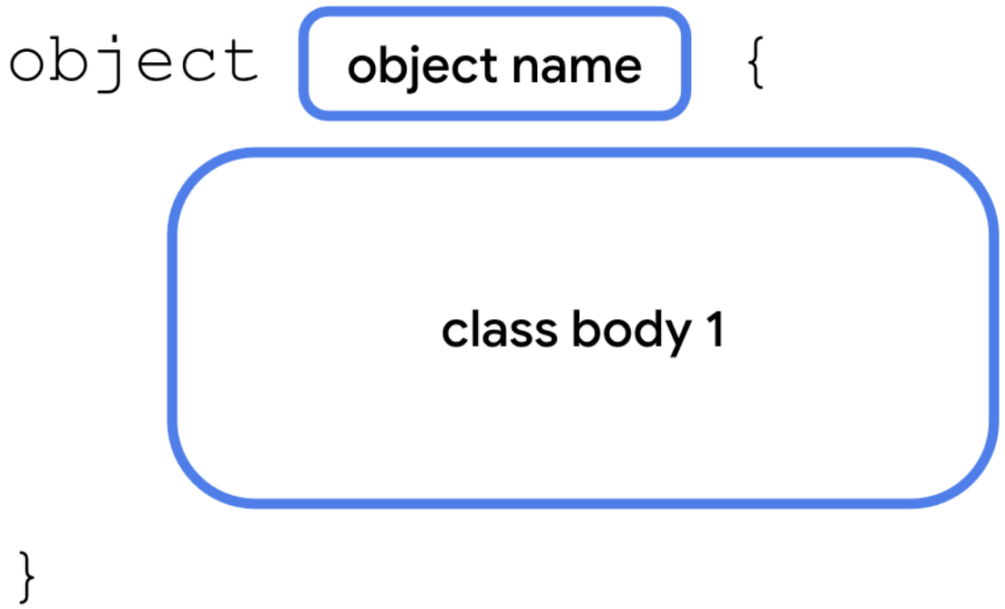
```
val [instance name] = [class name] < [generic data type] >( [parameters] )
```

Usually generic type named as T (short for type), or other capital letters if the class has multiple generic types.

Objects

Object / singleton

- Sometimes you only want one instance of a class to ever exist
- Use the `object` keyword instead of the `class` keyword



Object / singleton example

```
object Calculator {
    fun add(n1: Int, n2: Int): Int {
        return n1 + n2
    }
}

println(Calculator.add(2,4))
=> 6
```

Companion objects

- Lets all instances of a class share a single instance of a set of variables or functions
- Use `companion` keyword

```
companion object object name
```

Companion object example

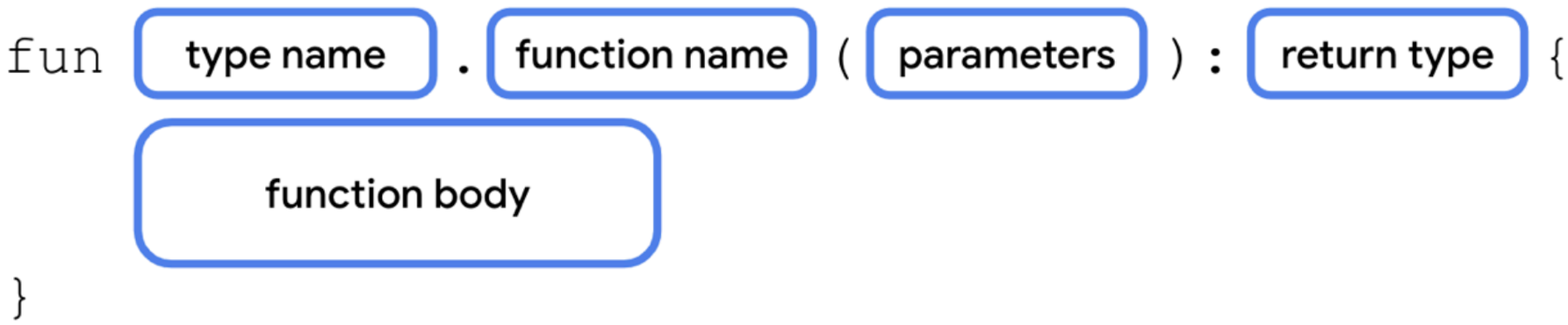
```
class PhysicsSystem {
    companion object WorldConstants {
        val gravity = 9.8
        val unit = "metric"
        fun computeForce(mass: Double, accel: Double): Double {
            return mass * accel
        }
    }
}
println(PhysicsSystem.WorldConstants.gravity)
println(PhysicsSystem.WorldConstants.computeForce(10.0, 10.0))

=> 9.8
   100.0
```

Extensions

Extension functions

Add functions to an existing class that you cannot modify directly.

- Appears as if the implementer added it
- Not actually modifying the existing class
- Cannot access private instance variables



Extension function example

Add `isOdd()` to `Int` class:

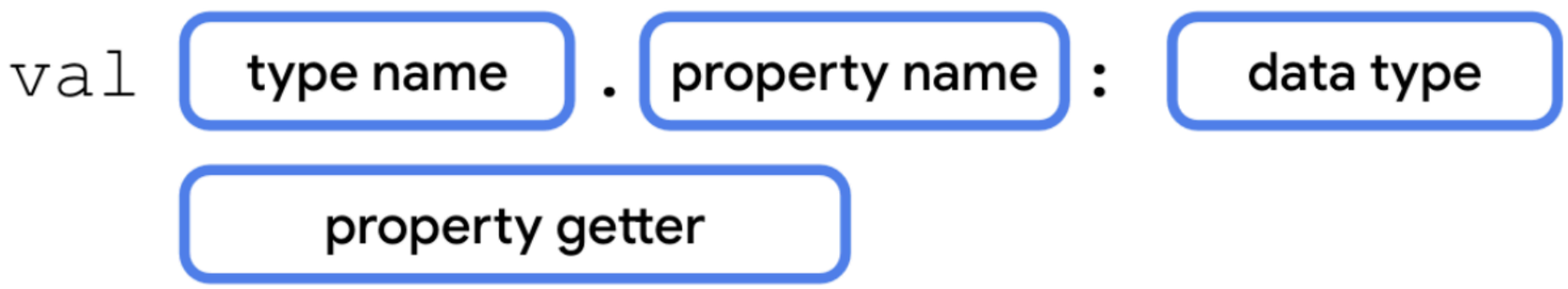
```
fun Int.isOdd(): Boolean { return this % 2 == 1 }
```

Call `isOdd()` on an `Int`:

```
3.isOdd()
```

Extension properties

- Just like extension functions, you can add additional properties to existing classes without modifying their source code.
- When creating extension properties, remember that they do not store data themselves but provide a getter



Extension properties can't store data, so they must be get-only.

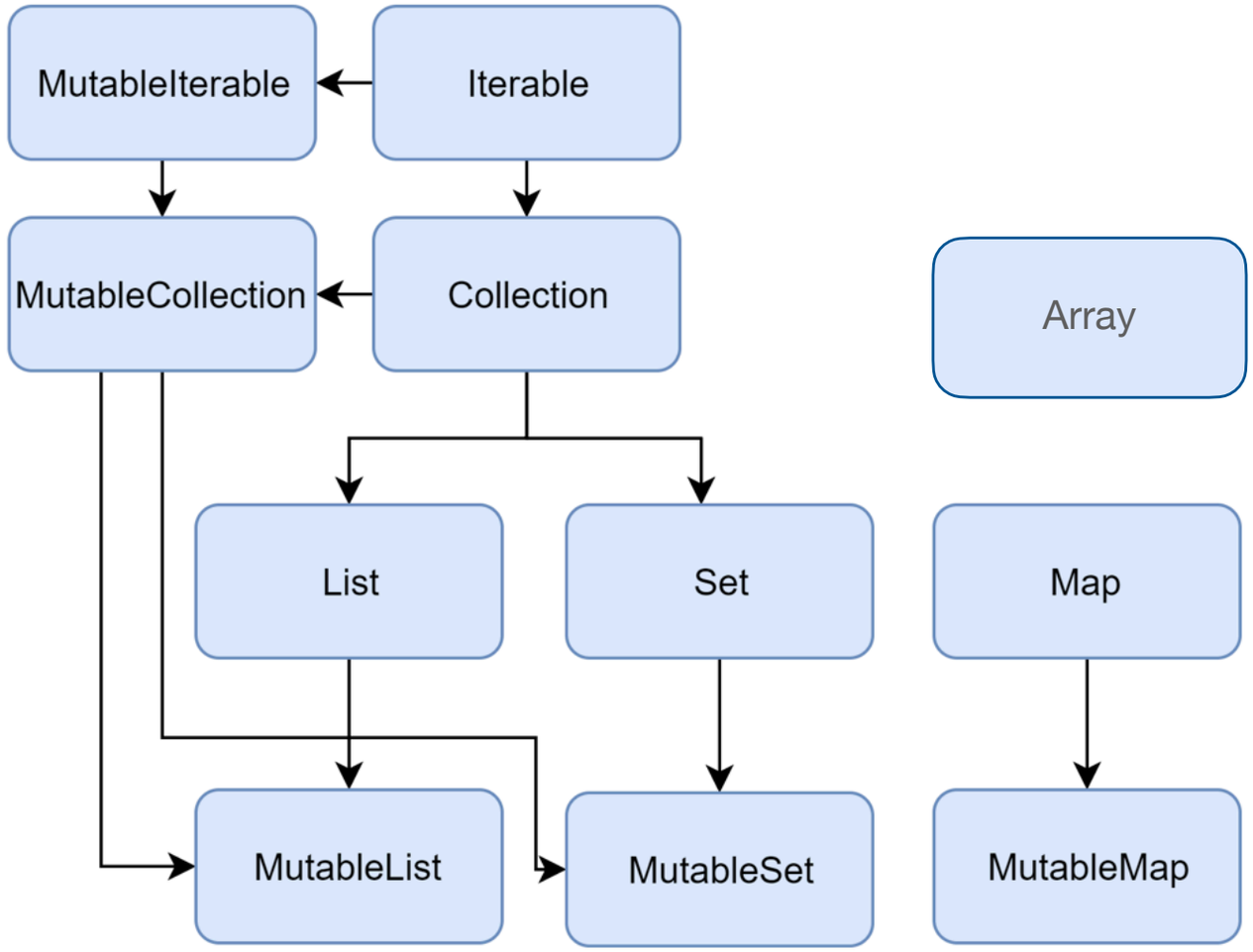
Why use extensions?

- Add functionality to classes that are not open
- Add functionality to classes you don’t own
- Separate out core API from helper methods for classes you own

Define extension functions in an easily discoverable place such as in the same file as the class, or a well-named function.

Collections

Overall view of collections



Arrays

- Arrays store multiple items
- Array elements can be accessed programmatically through their indices
- Array elements are mutable
- Array size is fixed

Array using arrayOf()

An array of strings can be created using `arrayOf()`

```
val pets = arrayOf("dog", "cat", "canary")
println(pets.joinToString())
```

```
⇒ dog, cat, canary
```

Access and modify elements of array

```
val simpleArray = arrayOf(1, 2, 3)

// Accesses the element and modifies it
simpleArray[0] = 10

// Prints the modified element
println(simpleArray[0].toString())

=> 10
```

Lists

- Lists are ordered collections of elements
- List elements can be accessed programmatically through their indices
- Elements can occur more than once in a list

Immutable list using listOf()

Declare a list using `listOf()` and print it out.

```
val instruments = listOf("trumpet", "piano", "violin")
println(instruments)
```

```
⇒ [trumpet, piano, violin]
```

Mutable list using mutableListOf()

Lists can be changed using mutableListOf()

```
val myList = mutableListOf("trumpet", "piano", "violin")
myList.remove("violin")
```

⇒ kotlin.Boolean = true

With a list defined with val, you can't change which list the variable refers to, but you can still change the contents of the list.

Sets

- Set stores unique elements
- Elements’ order is generally undefined.
- `null` elements are unique as well: a Set can contain only one `null`

Set using setOf()

Declare two sets and compare them

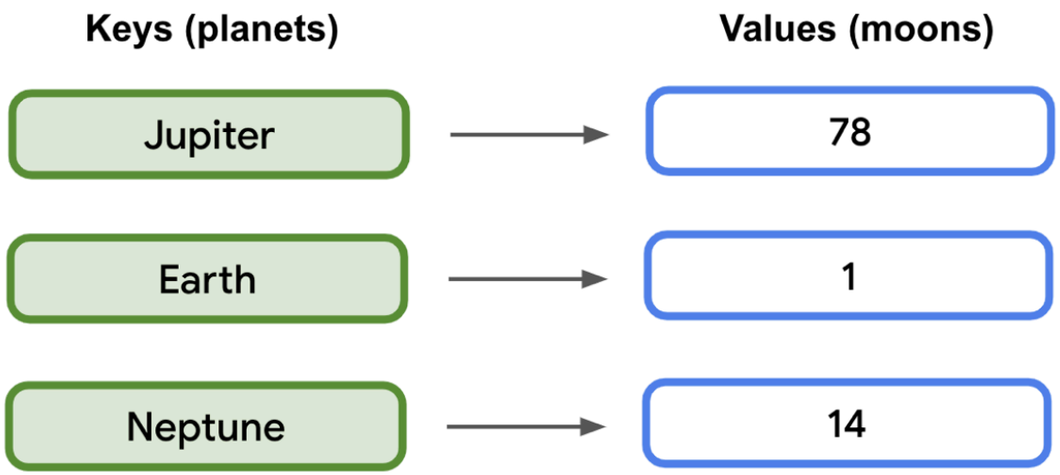
```
val numbers = setOf(1, 2, 3, 4)
println("Number of elements: ${numbers.size}")
if (numbers.contains(1)) println("1 is in the set")
```

```
val numbersBackwards = setOf(4, 3, 2, 1)
println("The sets are equal: ${numbers == numbersBackwards}”)
```

```
=> Number of elements: 4
    1 is in the set
    The sets are equal: true
```

Maps

- Map stores key-value pairs (or entries)
- Keys are unique, but different keys can be paired with equal values.
- Provides specific functions, such as access to value by key, searching keys and values, and so on.



Immutable map using mapOf()

```
val [map name] = mapOf(
    [key] to [value] ,
    [key] to [value] ,
    [key] to [value] ,
)
```

Immutable map using mapOf()

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3)

println("All keys: ${numbersMap.keys}")
println("All values: ${numbersMap.values}")
if ("key2" in numbersMap) {
    println(“Value by key \"key2\": ${numbersMap[“key2”]}”)
}
```

```
=> All keys: [key1, key2, key3]
   All values: [1, 2, 3]
   Value by key "key2": 2
```

Mutable map using mutableMapOf()

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
numbersMap["one"] = 11

println(numbersMap)
```

```
=> {one=11, two=2, three=3}
```

Workshop