CS 473 - MDP
Mobile Device Programming

Maharishi International University

CS 473 - MDP
Mobile Device Programming

MS.CS Program

Department of Computer Science

Renuka Mohanraj, Ph.D.

Maharishi International University

Course Overview Chart

	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Week 1	Course Overview; **Lesson 1(a):** Kotlin Fundamentals	**Lesson 1(b) & 2:** More Kotlin Fundamentals & Introduction to Android	**Lesson 3:** Jetpack Compose Basics	**Lesson 4:** Lists & Grid	**Lesson 5:** App Architecture and Dependency Injection	**Lesson 5 contd:** App Architecture and Dependency Injection
	Lesson 1(a): cont'd	Lab	Lab	Lab	Lab	
Week 2	Revision for Midterm	Midterm Exam	**Lesson 6:** Coroutine and Testing ViewModel with Coroutine	**Lesson 7:** Navigating Android Apps: Activities and Jetpack Navigation	**Lesson 7 contd:** Navigating Android Apps: Activities and Jetpack Navigation	**Lesson 8:** Full Stack App Development
			Lab/Project	Lab/Project	Lab/Project	
Week 3	**Lesson 8 cont'd:** Full Stack App Development	**Lesson 9:** Data Persistence	**Lesson 9 cont'd:** Data Persistence	**Lesson 10:** Work Manager	**Lesson 11:** Generative AI on Android	**Lesson 11 cont'd:** Generative AI on Android
	Lab/Project	Lab/Project	Lab/Project	Lab/Project	Lab/Project	Project
Week 4	Revision for Final	Final Exam	Project	Project Presentation	Project Presentation	Project Presentation
			Early Bird Project Presentation	Project Presentation	Project Presentation	Project Presentation

Lesson-1(b)
More Kotlin Fundamentals

The nature of life is to grow

Lesson 4

DATA LAYOUTS IN ANDROID USING LISTS AND GRIDS:

Order is present everywhere

WHOLENESS OF THE LESSON

This lesson provides a deeper understanding of advanced Kotlin concepts such as Higher-Order Functions, Generics, Objects, Extensions, and Scope Functions. Mastery of these topics equips us with the ability to write more powerful, reusable, and concise code, forming a solid foundation for creating flexible and maintainable applications.
Science and Technology of Consciousness: Similarly, the principle "Life is found in layers" from the Science of Creative Intelligence reminds us that knowledge and skills unfold progressively. Just as each layer of experience reveals greater depth and understanding, each advanced Kotlin concept builds upon the previous one, enriching our capacity as developers.

MAIN POINTS

1. Higher-Order Functions and Generics expand the expressive power and reusability of Kotlin by treating functions as first-class citizens and creating type-safe, adaptable code structures. *Science and Technology of Consciousness:* Growth unfolds in layers, where one principle supports many expressions. Generics and higher-order functions reflect this layered growth, enabling simple foundations to generate broader possibilities.
2. Objects and Extensions provide mechanisms to centralize shared behavior and add new functionality to existing classes without modifying their original design. *Science and Technology of Consciousness:* The field of pure consciousness is the unified basis for diversity. Similarly, objects unify shared behavior, while extensions allow growth without disturbing the underlying essence.
3. Scope Functions (let, apply, run, also, with) simplify initialization, transformation, and configuration, leading to more concise and readable programs. *Science and Technology of Consciousness:* Nature's organizing power brings order to complexity; scope functions mirror this by streamlining code and revealing clarity within layered structures.

Wholeness

This lesson provides a deeper understanding of advanced Kotlin concepts such as Higher-Order Functions, Generics, Objects, Extensions, and Scope Functions. Mastery of these topics equips us with the ability to write more powerful, reusable, and concise code, forming a solid foundation for creating flexible and maintainable applications.

Science and Technology of Consciousness: Similarly, the principle "Life is found in layers" from the Science of Creative Intelligence reminds us that knowledge and skills unfold progressively. Just as each layer of experience reveals greater depth and understanding, each advanced Kotlin concept builds upon the previous one, enriching our capacity as developers.

Agenda

- Higher Order Function
- Generics
- objects
- Extensions
- Scope Functions

Higher Order Function

A higher-order function is a function that takes functions as parameters or returns a function.

```
fun operation(a: Int, b: Int, myFunc: (Int, Int) -> Int) {
    //...
}
```

The function operation accepts **three inputs**:

- First parameter: an integer (a)
- Second parameter: an integer (b)
- Third parameter: a function (myFunc) that takes two integers and returns an integer

Inside operation, we call myFunc(a, b).

Higher Order Function › Passing Function as Parameter

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

fun operation(a: Int, b: Int, myFunc: (Int, Int) -> Int) {
    println(myFunc(a, b))
}
```

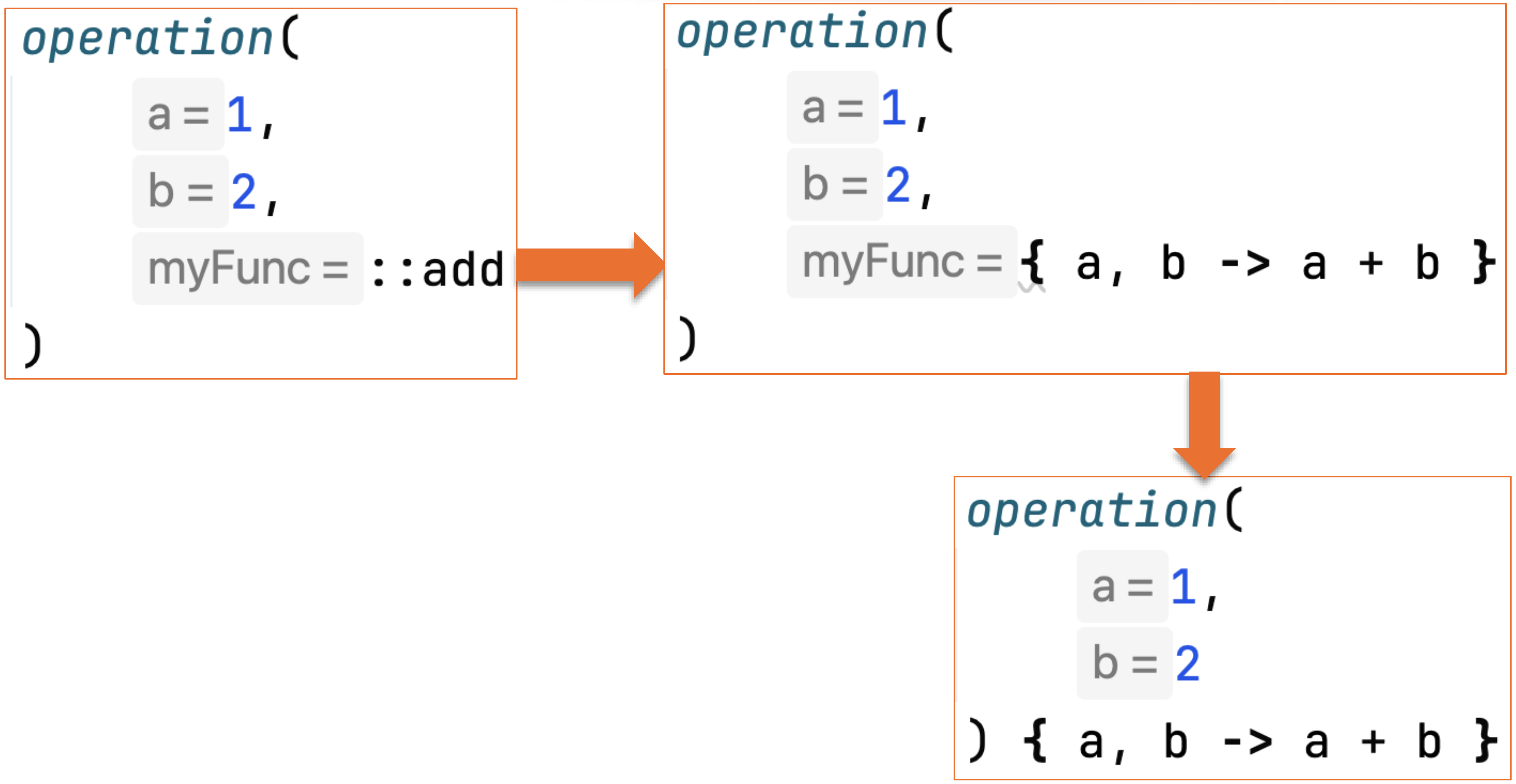
Note:
Only one lambda expression is allowed outside a parenthesized argument list.

```
fun main() {
    operation(1, 2, ::multiply)
    operation(1, 2, ::add)
    operation(10, 20) { a, b -> a * b }
}
```

Output

```
2
3
200
```

Higher Order Function ❯ Passing Function as Parameter



```
operation(
    a = 1,
    b = 2,
    myFunc = ::add
)
```

```
operation(
    a = 1,
    b = 2,
    myFunc = { a, b -> a + b }
)
```

```
operation(
    a = 1,
    b = 2
) { a, b -> a + b }
```

Note:
Only one lambda expression is allowed outside a parenthesized argument list.

Higher Order Function ❯ Returns a Function

```
//fun operation(myFunc: (Int, Int) -> Int): (Int, Int) -> Int = myFunc

fun operation(myFunc: (Int, Int) -> Int) = myFunc
```

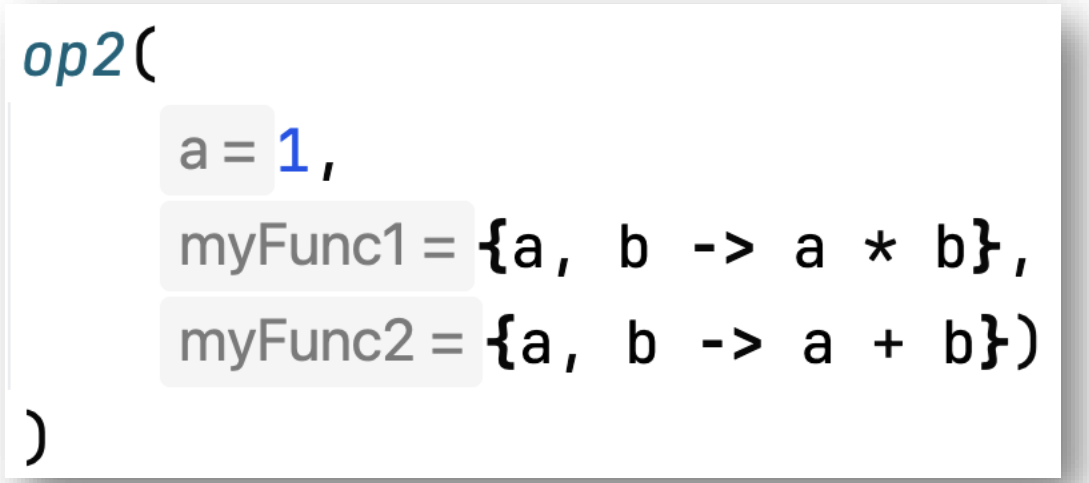
```
fun main() {
    println(operation(::multiply)(1, 2))
    println(operation(::add)(1, 2))
    println(operation { a, b -> a + b }(1, 2))
}
```

Output

```
2
3
3
```

Predict the Output

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

fun main() {
    println(
        op2(1, {a, b -> a * b}) {a, b -> a + b}
    )
}

fun op2(a: Int, myFunc1: (Int, Int) -> Int, myFunc2: (Int, Int) -> Int): Int{
    return myFunc1(a, myFunc2(a, a))
}
```



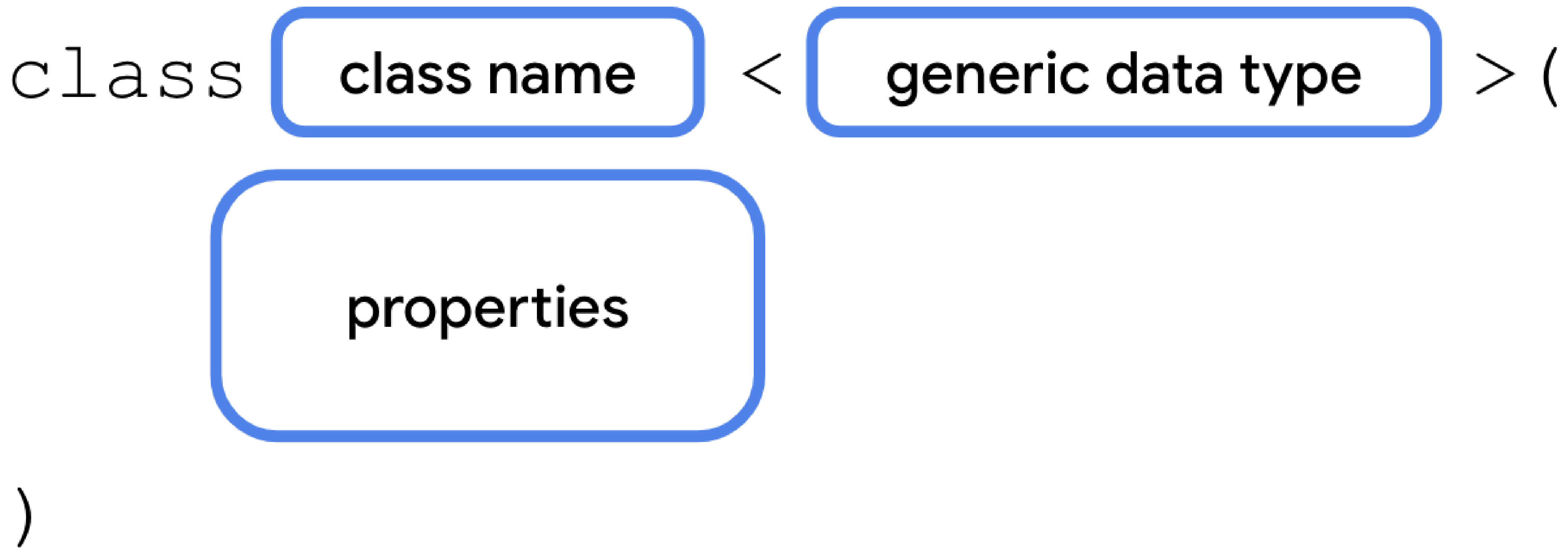
```
op2(
        a = 1,
        myFunc1 = {a, b -> a * b},
        myFunc2 = {a, b -> a + b})
)
```

Output

2

Make a Reusable Class with Generics

- Generics allow us to create classes, interfaces, and functions that can work with any type, not just a specific one.
- This is achieved by specifying a type parameter in angle brackets <>.



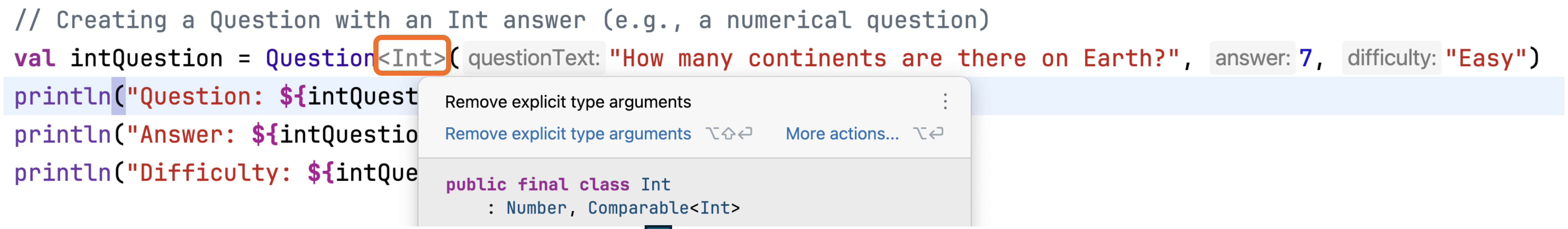
Ref: https://developer.android.com/codelabs/basic-android-kotlin-compose-generics?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-1%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-andro

Sample code for Generic class

```
// A generic Question class that can hold any type of answer
class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: String
)
```

```
class [class name] < [generic data type] > (
        [properties]
)
```

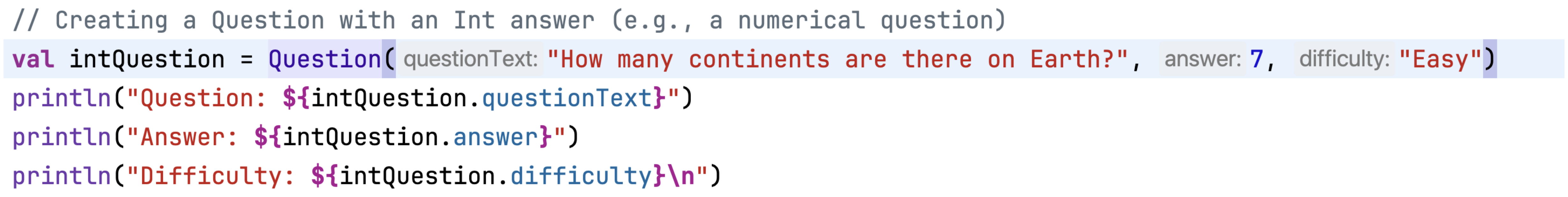
```kotlin
fun main() {
    // Creating a Question with an Int answer
    val intQuestion = Question<Int>("How many continents are there on Earth?", 7, "Easy")
    println("Question: ${intQuestion.questionText}")
    println("Answer: ${intQuestion.answer}")
    println("Difficulty: ${intQuestion.difficulty}\n")

    // Creating a Question with a String answer
    val stringQuestion = Question<String>("What is the largest ocean on Earth?", "Pacific Ocean", "Medium")
    println("Question: ${stringQuestion.questionText}")
    println("Answer: ${stringQuestion.answer}")
    println("Difficulty: ${stringQuestion.difficulty}\n")

    // Creating a Question with a Boolean answer
    val booleanQuestion = Question<Boolean>("Is Mount Everest the tallest mountain on Earth?", true, "Hard")
    println("Question: ${booleanQuestion.questionText}")
    println("Answer: ${booleanQuestion.answer}")
    println("Difficulty: ${booleanQuestion.difficulty}")
}
```



```
// Creating a Question with an Int answer (e.g., a numerical question)
val intQuestion = Question<Int>( questionText: "How many continents are there on Earth?",  answer: 7,  difficulty: "Easy")
println("Question: ${intQuest
println("Answer: ${intQuestio
println("Difficulty: ${intQue
```

Remove explicit type arguments
Remove explicit type arguments ⌥⇧↩
More actions... ⌥↩

```
public final class Int
    : Number, Comparable<Int>
```



```
// Creating a Question with an Int answer (e.g., a numerical question)
val intQuestion = Question( questionText: "How many continents are there on Earth?",  answer: 7,  difficulty: "Easy")
println("Question: ${intQuestion.questionText}")
println("Answer: ${intQuestion.answer}")
println("Difficulty: ${intQuestion.difficulty}\n")
```

Use an enum class

```
Question("How many continents are there on Earth?", 7, "Easy")

Question("What is the largest ocean on Earth?", "Pacific Ocean", "Medium")

Question("Is Mount Everest the tallest mountain on Earth?", true, "Hard")
```

- Mistype one of the three possible strings, it could introduce bugs.
- If the values change, for example, "*medium*" is renamed to "*average*", then we need to update all usages of the string.
- The code is harder to maintain if we add more difficulty levels.

Syntax with sample code

```
enum class Difficulty {
   EASY, MEDIUM, HARD
}
```

```
enum class enum name {
     Case 1 , Case 2 , Case 3

}
```

Refactor the Question class

```
class Question<T>(
    val questionText: String,
    val answer: T,
    val difficulty: Difficulty
)
```

```
enum class Difficulty {
    EASY, MEDIUM, HARD
}
```

```
Question( questionText: "How many continents are there on Earth?",  answer: 7,  difficulty: "Easy")
Question( questionText: "What is the largest ocean on Earth?",  answer: "Pacific Ocean",  difficulty: "Medium")
Question( questionText: "Is Mount Everest the tallest mountain on Earth?",  answer: true,  difficulty: "Hard")
```

```
Question( questionText: "How many continents are there on Earth?",  answer: 7, Difficulty.EASY)
Question( questionText: "What is the largest ocean on Earth?",  answer: "Pacific Ocean", Difficulty.MEDIUM)
Question( questionText: "Is Mount Everest the tallest mountain on Earth?",  answer: true, Difficulty.HARD)
```

Use a data class

- If a class primarily holds data and has no significant methods for behavior, it can be defined as a data class.

```
data class class name ( ... )
data class Question<T>(
   val questionText: String,
   val answer: T,
   val difficulty: Difficulty
)
```

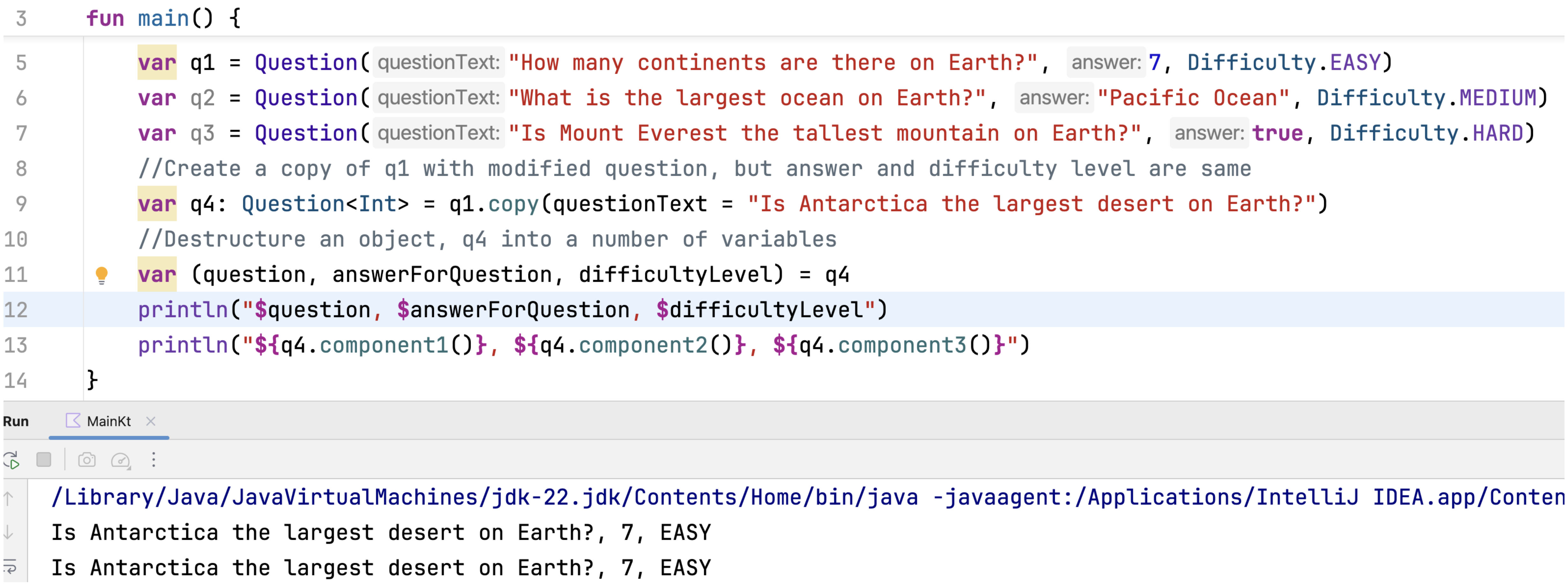
```
class Question<T>(
   val questionText: String,
   val answer: T,
   val difficulty: Difficulty
)
```

When a class is defined as a data class, the following methods are implemented.

- equals()
- hashCode()
- toString()
- componentN(): component1(), component2(), etc.
- copy()

Note:

- A data class needs to have at least one parameter in its constructor, and all constructor parameters must be marked with val or var.
- A data class also cannot be abstract, open, sealed, or inner.



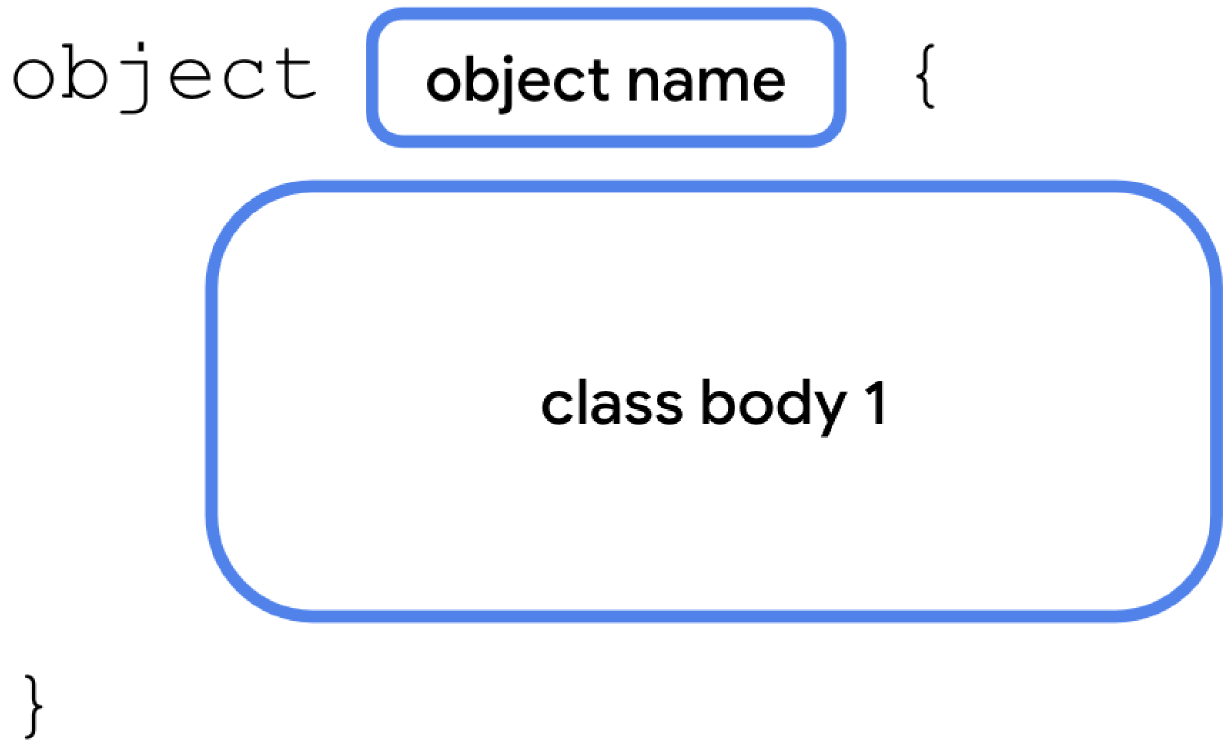
```kotlin
fun main() {
    var q1 = Question( questionText: "How many continents are there on Earth?",  answer: 7, Difficulty.EASY)
    var q2 = Question( questionText: "What is the largest ocean on Earth?",  answer: "Pacific Ocean", Difficulty.MEDIUM)
    var q3 = Question( questionText: "Is Mount Everest the tallest mountain on Earth?",  answer: true, Difficulty.HARD)
    //Create a copy of q1 with modified question, but answer and difficulty level are same
    var q4: Question<Int> = q1.copy(questionText = "Is Antarctica the largest desert on Earth?")
    //Destructure an object, q4 into a number of variables
    var (question, answerForQuestion, difficultyLevel) = q4
    println("$question, $answerForQuestion, $difficultyLevel")
    println("${q4.component1()}, ${q4.component2()}, ${q4.component3()}")
}
```

```
/Library/Java/JavaVirtualMachines/jdk-22.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Conten
Is Antarctica the largest desert on Earth?, 7, EASY
Is Antarctica the largest desert on Earth?, 7, EASY
```

Use a singleton object

There are many scenarios where we want a class to only have one instance.

- For example:
  - Player status in a mobile game for the current user.
  - An object to access a remote data source (such as a Firebase database).

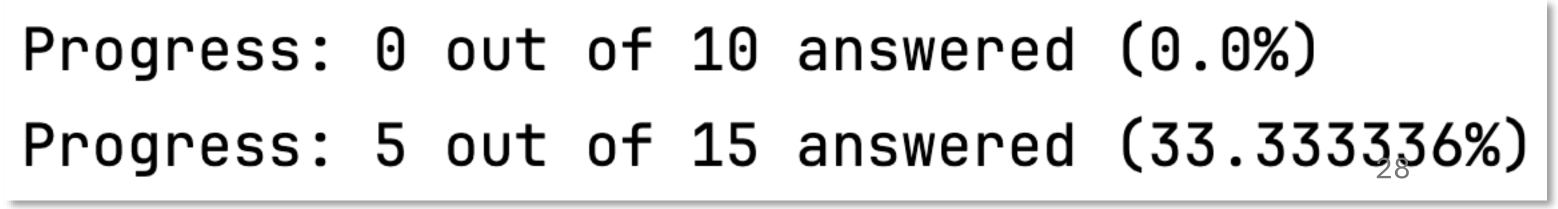


Scenario

- For a quiz, it would be great to have a way to keep track of the total number of questions, and the number of questions the student answered so far.
- We only need one instance of this class to exist, so instead of declaring it as a class, declare it as a singleton object.

```kotlin
object StudentProgress {
    private var total:Int = 10
    private var answered:Int = 0

    fun updateTotal(newTotal: Int) {
        total = newTotal
    }
    fun updateAnswered(newAnswer: Int) {
        answered = newAnswer
    }
    fun showProgress() {
        val percentage = if (total > 0) {
            answered.toFloat() / total.toFloat() * 100
        } else {
            0.0
        }
        println("Progress: $answered out of $total answered ($percentage%)")
    }
}
```

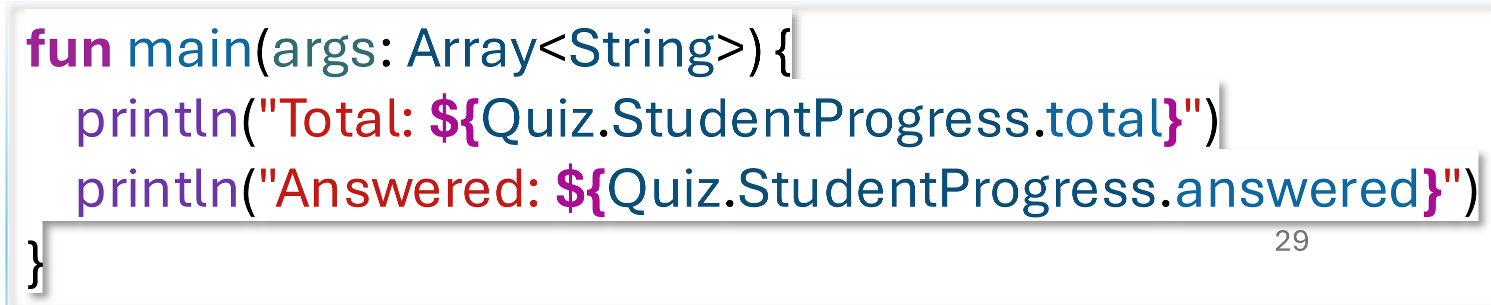
```kotlin
fun main(args: Array<String>) {
    StudentProgress.showProgress()
    // Simulating updates
    StudentProgress.updateTotal(15)
    StudentProgress.updateAnswered(5)

    // Showing updated progress
    StudentProgress.showProgress()
}
```



```
Progress: 0 out of 10 answered (0.0%)
Progress: 5 out of 15 answered (33.333336%)
```

Move StudentProgress to Quiz

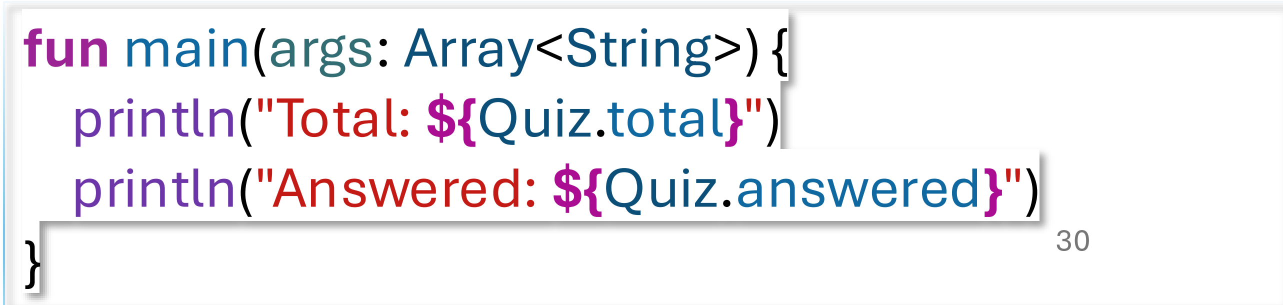
```
class Quiz {
  var q1 = Question("How many continents are there on Earth?", 7, Difficulty.EASY)
  var q2 = Question("What is the largest ocean on Earth?", "Pacific Ocean", Difficulty.MEDIUM)
  var q3 = Question("Is Mount Everest the tallest mountain on Earth?", true, Difficulty.HARD)
  object StudentProgress {
      var total: Int = 10
      var answered: Int = 3
  }
}
```



```
fun main(args: Array<String>) {
   println("Total: ${Quiz.StudentProgress.total}")
   println("Answered: ${Quiz.StudentProgress.answered}")
}
```

Mark StudentProgress with companion keyword

```
class Quiz {
  var q1 = Question("How many continents are there on Earth?", 7, Difficulty.EASY)
  var q2 = Question("What is the largest ocean on Earth?", "Pacific Ocean", Difficulty.MEDIUM)
  var q3 = Question("Is Mount Everest the tallest mountain on Earth?", true, Difficulty.HARD)
 companion object StudentProgress {
     var total: Int = 10
     var answered: Int = 3
 }
}
```



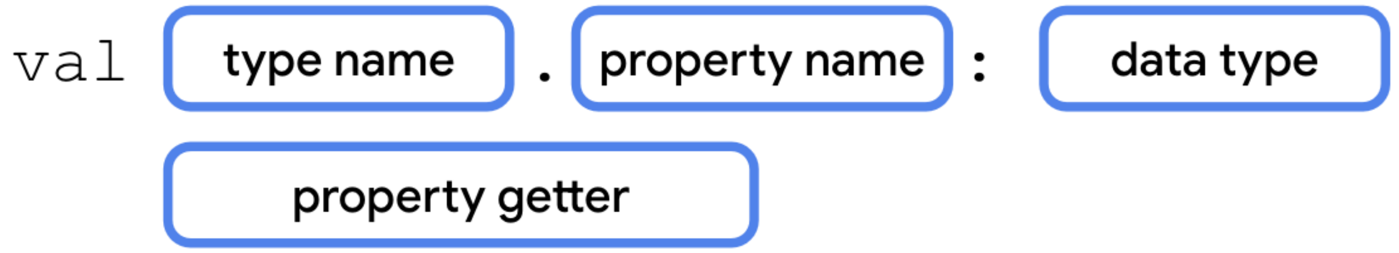
```
fun main(args: Array<String>) {
   println("Total: ${Quiz.total}")
   println("Answered: ${Quiz.answered}")
}
```

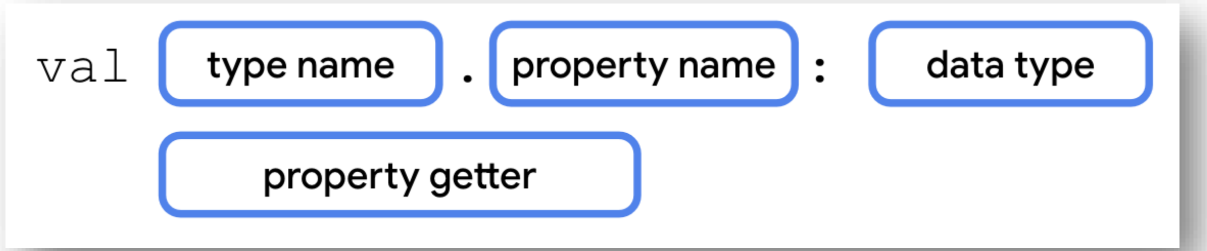
```kotlin
class Stack {
    // Companion object acts like a "static object" tied to the class
    companion object {
        var top = -1
            private set
        val items = mutableListOf<Int>()
        fun push(item: Int) {
            items.add(item)
            ++top
        }
        fun pop(): Int {
            return items.removeAt(top--)
        }
        fun isEmpty(): Boolean {
            return top == -1
        }

        override fun toString(): String {
            return items.toString()
        }
    }
}
```

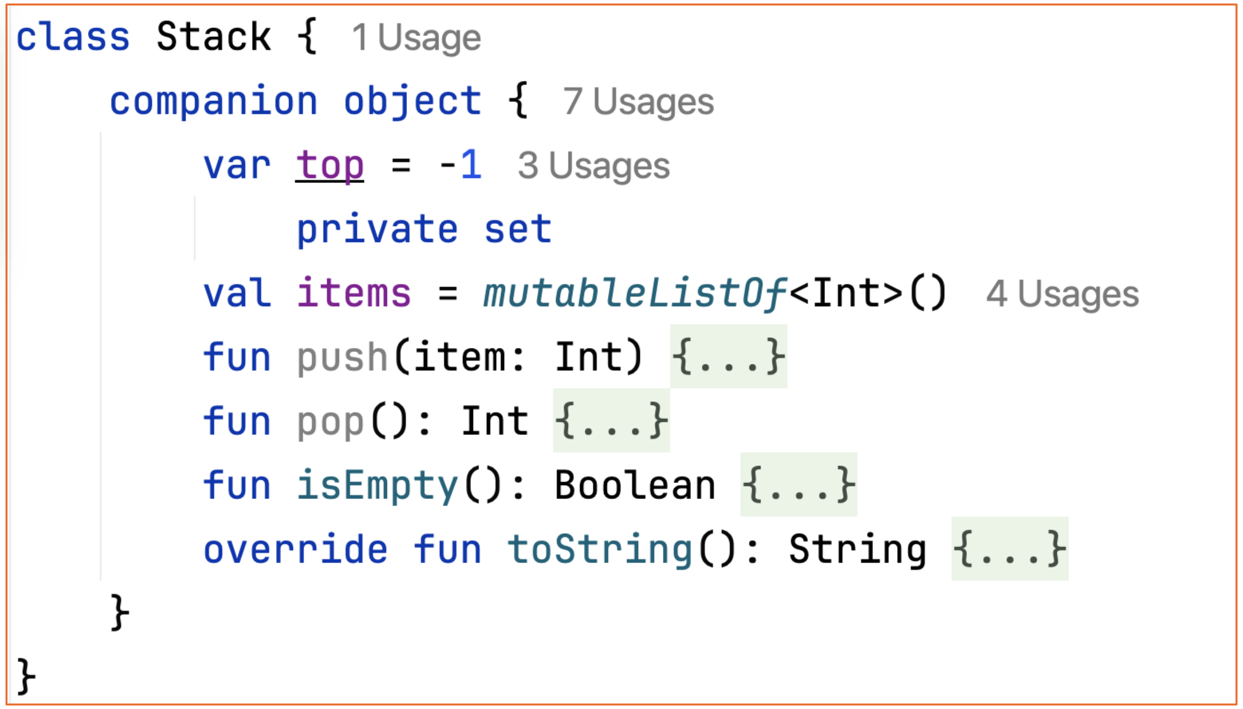
```kotlin
fun main() {
    // Explicitly calling push via Companion name
    Stack.Companion.push(1)
    // Shortcut syntax: members of the companion object can be called directly on the class
    Stack.push(2)
    println(Stack.Companion) //[1, 2]
    println(Stack) //[1, 2]
    println(Stack.items) //[1, 2]

}
```

Extend classes with new properties and methods

Kotlin language gives other developers the ability to extend existing data types, adding properties and methods that can be accessed with dot syntax, as if they were part of that data type.







```
class Stack {  1 Usage
    companion object {  7 Usages
        var top = -1  3 Usages
            private set
        val items = mutableListOf<Int>()  4 Usages
        fun push(item: Int) {...}
        fun pop(): Int {...}
        fun isEmpty(): Boolean {...}
        override fun toString(): String {...}
    }
}
```

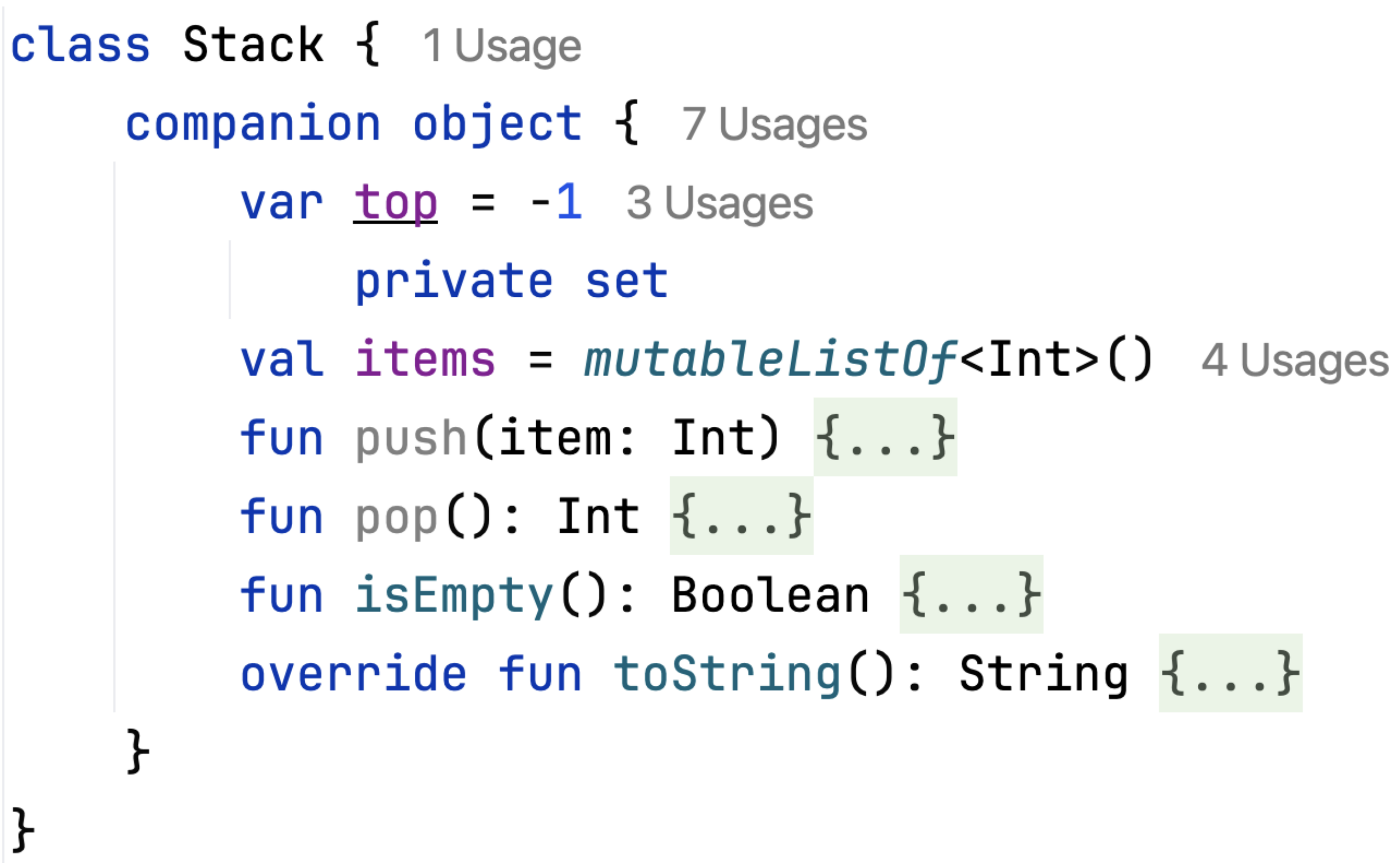
```
val Stack.size
    get() = items.size

fun main() {
    Stack.push(1)
    Stack.push(2)
    val stack = Stack()
    println(stack.sumOfItems()) //3
    println("Size of stack is ${stack.size}") //2
}
```

Add an extension function

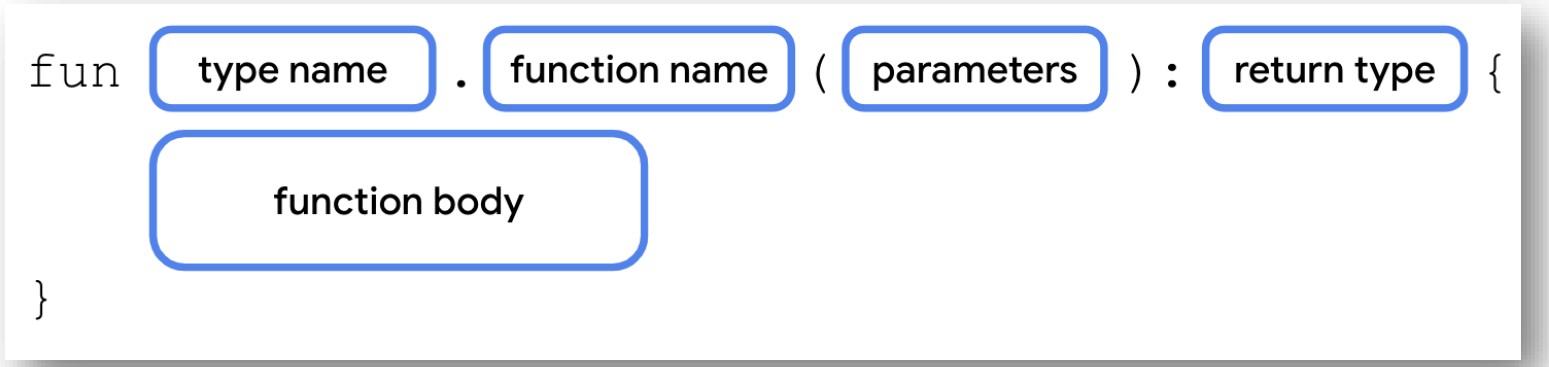
Add the type name and a dot operator (.) before the function name.

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

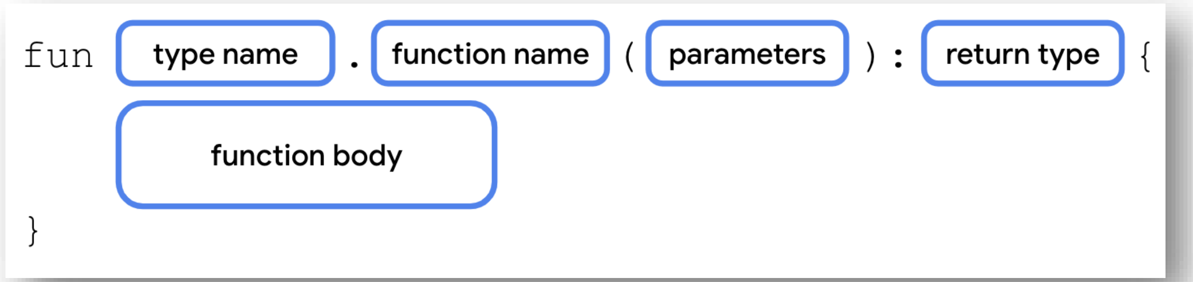
How to create an extended method for Stack class?

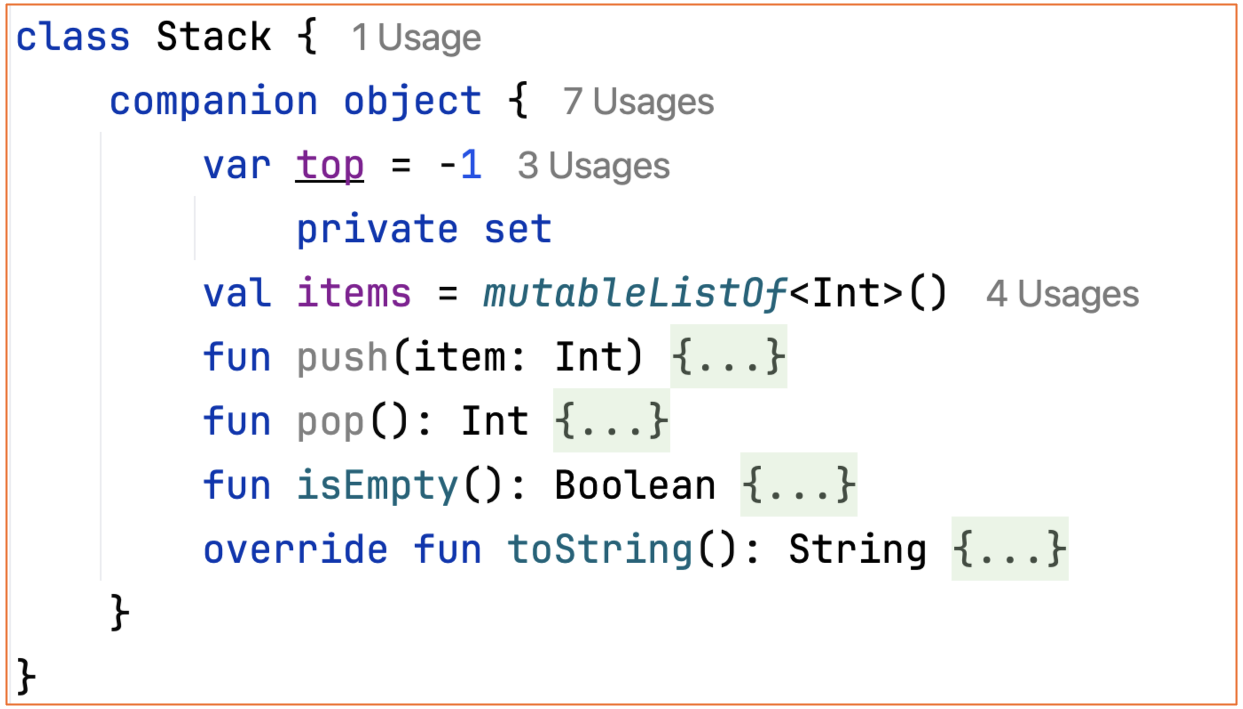


```
class Stack {  1 Usage
    companion object {  7 Usages
        var top = -1  3 Usages
            private set
        val items = mutableListOf<Int>()  4 Usages
        fun push(item: Int) {...}
        fun pop(): Int {...}
        fun isEmpty(): Boolean {...}
        override fun toString(): String {...}
    }
}
```



```
fun [type name] . [function name] ( [parameters] ) : [return type] {
    [function body]
}
```

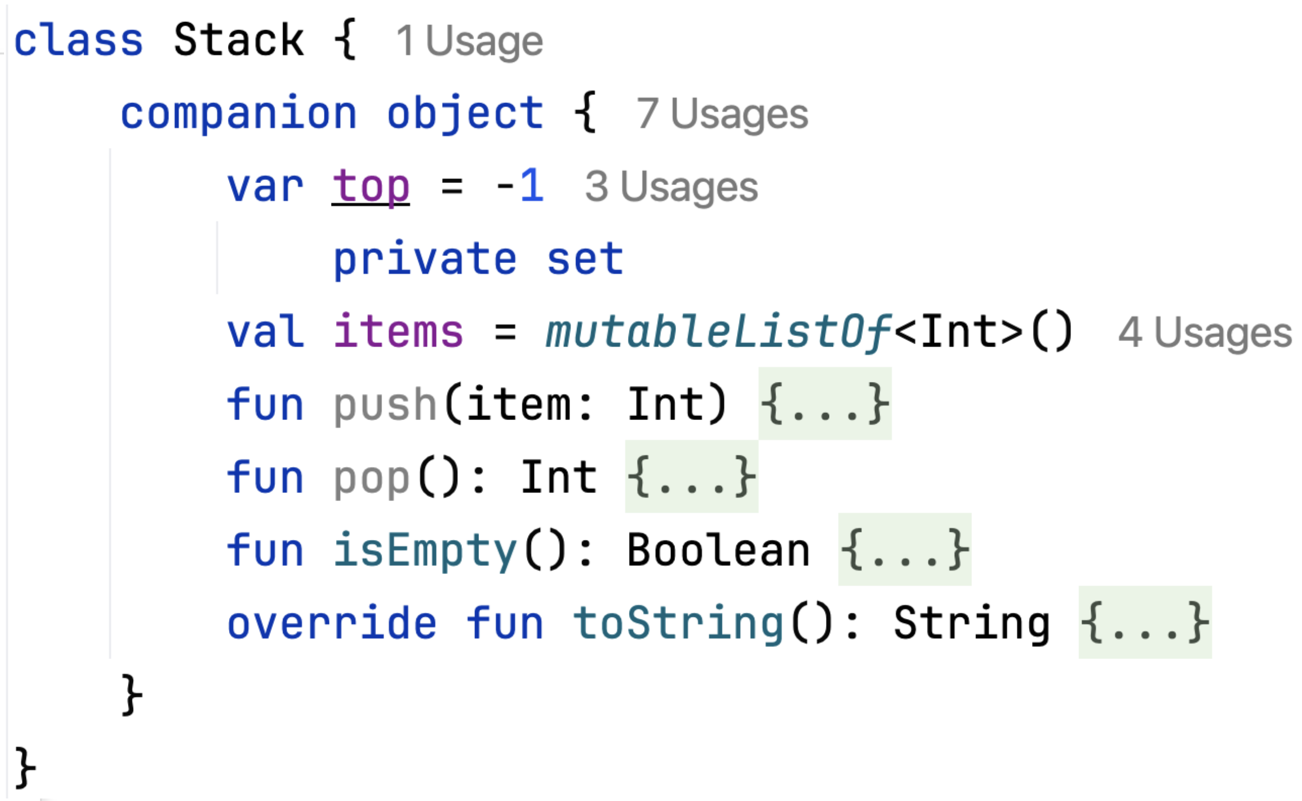




```
class Stack {  1 Usage
    companion object {  7 Usages
        var top = -1  3 Usages
            private set
        val items = mutableListOf<Int>()  4 Usages
        fun push(item: Int) {...}
        fun pop(): Int {...}
        fun isEmpty(): Boolean {...}
        override fun toString(): String {...}
    }
}
```

```
//extended method for Stack class
fun Stack.sumOfItems()  = items.sum() // or Stack.items.sum()

fun main() {
    Stack.Companion.push(1)
    Stack.push(2)
    val stack = Stack()
    println(stack.sumOfItems())
}
```

Extended method for object



```
class Stack {  1 Usage
    companion object {  7 Usages
        var top = -1  3 Usages
            private set
        val items = mutableListOf<Int>()  4 Usages
        fun push(item: Int) {...}
        fun pop(): Int {...}
        fun isEmpty(): Boolean {...}
        override fun toString(): String {...}
    }
}
```

```
//extended method for Companion object in Stack class
fun Stack.Companion.printStackTrace() {
    items.reversed().forEach { print("$it ") }
    println()
}
```

Scope Functions

Kotlin

Scope Functions

- The Kotlin standard library contains several functions whose sole purpose is to execute a block of code within the context of an object.
- When we call such a function on an object with a lambda expression provided, it forms a temporary scope. In this scope, we can access the object without its name. Such functions are called scope functions.

Scope Functions

There are five of them:

- let
- apply
- run
- also
- with

Basically, these functions all perform the same action: execute a block of code on an object.

What's different is how this object becomes available inside the block and what the result of the whole expression is.

```kotlin
class Person(
    val name: String,
    var age: Int,
    var city: String
) {
    fun incrementAge() = ++age
    fun moveTo(city: String) {...}
    override fun toString(): String {...}
}
```

let

Context Object: it

Returns: The lambda result.

```kotlin
fun main() {
    val alice = Person("Alice", 20, "Amsterdam")
    val info = alice.let {
        it.moveTo("Paris")
        it.incrementAge()
        "Name: ${it.name}, Age: ${it.age}, City: ${it.city}"
    }
    println(alice)
    println(info)
}
```

Output

```
Name: Alice, Age: 21, City: Paris
Person(name='Alice', age=21, city='Paris')
```

apply

```
class Person {
    var name: String? = null
    var age: Int? = null
    override fun toString(): String {
        return "Employee(name=$name, age=$age)"
    }
}
```

Context Object: this

Returns: The context object itself.

```
fun main() {
    val person = Person()
    person.apply {
        this.name = "John" //this is optional
        age = 25
    }
    println(person)
}
```

run

Context Object: this

Returns: The lambda result.

It's useful when you need to both configure an object (accessing its members as this) and then return a result from the operation.

```
class Person {
    var name: String? = null
    var age: Int? = null
    override fun toString(): String {
        return "Employee(name=$name, age=$age)"
    }
}
```

```
fun main() {
    val person = Person()
    val bio = person.run {
        name = "John"
        age = 25
        "My name is $name, I'm $age years old."
    }
    println(bio) // My name is John, I'm 25 years old.
}
```

also

```
class Person {
    var name: String? = null
    var age: Int? = null
    override fun toString(): String {
        return "Employee(name=$name, age=$age)"
    }
}
```

Context Object: it

Returns: The context object itself.

```
fun main() {
    var person = Person()
    person = person.also {
        it.name = "John"
        it.age = 25
        println("Employee created: $it")
    }
    println(person)
}
```

Output

```
Employee created: Employee(name=John, age=25)
Employee(name=John, age=25)
```

with

Context Object: this

Returns: The lambda result.

```
class Person {
    var name: String? = null
    var age: Int? = null
    override fun toString(): String {
        return "Employee(name=$name, age=$age)"
    }
}
```

```
fun main() {
    val person = Person()
    val status = with(person){
        name = "John"
        age = 25
        "Employee created: $this"
    }
    println(status)
}
```

Output

```
Employee created: Employee(name=John, age=25)
```

Use scope functions to access class properties and methods

Higher-order functions to access properties and methods of an object without referring to the object's name.

- Replace long object names using let()
    - The let() function allows you to refer to an object in a lambda expression using the identifier it, instead of the object's actual name.
- Call an object's methods without a variable using apply()

Replace long object names using let()

```
class Quiz {
  var question1 = Question("How many continents are there on Earth?", 7, Difficulty.EASY)
  var question2 = Question("What is the largest ocean on Earth?", "Pacific Ocean", Difficulty.MEDIUM)
  var question3 = Question("Is Mount Everest the tallest mountain on Earth?", true, Difficulty.HARD)

  companion object StudentProgress {
    var total: Int = 10
    var answered: Int = 3
  }
  fun printQuiz() {
    println(question1.questionText)
    println(question1.answer)
    println(question1.difficulty)
  }
}
```

```
fun printQuiz() {
  question1.let {
    println(it.questionText)
    println(it.answer)
    println(it.difficulty)
  }
}
```