Exercise 1: Lambdas and Higher-Order Functions

Task: Write a higher-order function that takes two integers and a lambda expression as parameters and applies the lambda to these integers.
Expected Output (for inputs 6, 3, and a lambda that multiplies its inputs):

```
18
```

Answer

```
fun applyOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
    return operation(a, b)
}

val result = applyOperation(6, 3) { x, y -> x * y }
println(result)
```

Explanation: This exercise introduces lambda expressions and higher-order functions. Common issues include incorrect lambda syntax or misunderstanding how to pass functions as parameters.

Exercise 2: Collection Operations

Task: Given a list of strings, use a single collection function to convert all strings to uppercase and then join them into a single string separated by commas.
Expected Output (for input ["apple", "banana", "cherry"]):

```
APPLE,BANANA,CHERRY
```

Answer

```
fun joinUpperCase(strings: List<String>): String {
    return strings.map { it.uppercase() }.joinToString(",")
}

println(joinUpperCase(listOf("apple", "banana", "cherry")))
```

Exercise 3a: Basic Class

Task: Create a simple class `Person` with two properties: `name` (String) and `age` (Int). Create an instance of this class and print its properties.

Expected Output:

```
Name: Alice, Age: 30
```

Answer

```
class Person(val name: String, val age: Int)

fun main() {
    val person = Person("Alice", 30)
    println("Name: ${person.name}, Age: ${person.age}")
}
```

Explanation: This exercise introduces basic class creation, including constructor and property declaration.

Exercise 3b: Adding Methods

Task: Add a method `greet` to the `Person` class that prints a greeting message.

Expected Output:

```
Hello, my name is Alice and I am 30 years old.
```

Answer

```
class Person(val name: String, val age: Int) {
    fun greet() {
        println("Hello, my name is $name and I am $age years old.")
    }
}

fun main() {
    val person = Person("Alice", 30)
    person.greet()
}
```

Explanation: This exercise shows how to add methods to a class.

Exercise 3c: Secondary Constructor

Task: Add a secondary constructor to the Person class that only takes name as a parameter and sets a default age.

Expected Output:

```
Name: Bob, Age: 25
```

Answer

```
class Person(val name: String, val age: Int = 25) {
    // Secondary constructor
    constructor(name: String) : this(name, 25)
}

fun main() {
    val person = Person("Bob")
    println("Name: ${person.name}, Age: ${person.age}")
}
```

Explanation: Demonstrates the use of secondary constructors and default parameter values.

Exercise 3d: Encapsulation

Task: Implement encapsulation in the Person class by making age private and adding a public method to increment it.

Expected Output:

```
Name: Alice, Age: 31
```

Answer

```
class Person(val name: String, private var age: Int) {
    fun incrementAge() {
        age++
    }

    fun displayInfo() {
        println("Name: $name, Age: $age")
    }
}

fun main() {
    val person = Person("Alice", 30)
    person.incrementAge()
    person.displayInfo()
}
```

Explanation: Introduces the concept of encapsulation by making properties private and providing public methods to interact with them.

Exercise 3e: Inheritance

Task: Create a Student class that inherits from Person and adds a new property studentId.
Expected Output:

```
Name: Charlie, Age: 20, Student ID: 12345
```

Answer

```
open class Person(val name: String, val age: Int)

class Student(name: String, age: Int, val studentId: Int) : Person(name,
age)

fun main() {
    val student = Student("Charlie", 20, 12345)
    println("Name: ${student.name}, Age: ${student.age}, Student ID:
${student.studentId}")
}
```

Explanation: Demonstrates basic inheritance, where `Student` extends `Person` and adds an additional property.

Exercise 4: Data Class
Task: Create a Kotlin data class Book with two properties: title (String) and author (String). Instantiate it and print its toString() representation.
Expected Output:

```
Book(title=Kotlin for Beginners, author=Jane Doe)
```

Answer

```
data class Book(val title: String, val author: String)

fun main() {
    val book = Book("Kotlin for Beginners", "Jane Doe")
    println(book)
}
```

Explanation: This introduces data classes in Kotlin, which are used for classes that primarily serve to hold data. They automatically provide `toString()`, `equals()`, and `hashCode()` methods.

Exercise 5: Class with Init Block

Task: Create a `User` class with an `init` block that prints a message when an instance is created.

Expected Output:

```
User created: Alice
```

Answer

```kotlin
class User(val name: String) {
    init {
        println("User created: $name")
    }
}

fun main() {
    val user = User("Alice")
}
```

Explanation: The `init` block in Kotlin is executed when the class instance is created. It's commonly used for initialization logic.

Exercise 6: Abstract Class

Task: Create an abstract class `Shape` with an abstract method `area()`. Implement this method in a derived class `Circle`.

Expected Output:

```
Area of the circle: 78.54
```

Answer

```
abstract class Shape {
    abstract fun area(): Double
}

class Circle(private val radius: Double) : Shape() {
    override fun area(): Double {
        return Math.PI * radius * radius
    }
}

fun main() {
    val circle = Circle(5.0)
    println("Area of the circle: %.2f".format(circle.area()))
}
```

Explanation: Abstract classes and methods define a contract for derived classes. They cannot be instantiated and must be implemented by subclasses.

Exercise 7: Interface Implementation

Task: Define an interface `Clickable` with a method `click()`. Implement this interface in a class `Button`.

Expected Output:

```
Button clicked
```

Answer

```kotlin
interface Clickable {
    fun click()
}

class Button : Clickable {
    override fun click() {
        println("Button clicked")
    }
}

fun main() {
    val button = Button()
    button.click()
}
```

Explanation: Interfaces in Kotlin are used to define a contract (like abstract classes) but can contain default method implementations. Classes implementing interfaces must provide implementations for all abstract methods.

Exercise 8: Sealed Classes

Task: Create a sealed class `Expr` with two subclasses `Const` (holding an integer) and `Sum` (holding two `Expr`s). Write a function `eval` that evaluates an `Expr` and returns its value.

Expected Output (for input `Sum(Const(1), Const(2))`):

```
3
```

```
sealed class Expr
data class Const(val number: Int) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()

fun eval(expr: Expr): Int = when (expr) {
    is Const -> expr.number
    is Sum -> eval(expr.e1) + eval(expr.e2)
}

println(eval(Sum(Const(1), Const(2))))
```

Exercise 9: Generics

Task: Write a generic function that takes an element and a list, and returns a new list with the element appended to the end of the list.

Expected Output (for input `"orange"` and `["apple", "banana"]`):

```
[apple, banana, orange]
```

Answer

```
fun <T> appendToList(element: T, list: List<T>): List<T> {
    return list + element
}

println(appendToList("orange", listOf("apple", "banana")))
```