Competency Based Learning Material (CBLM)

Android Mobile Application Development

Level-4

Module: Working with OOP (Object Oriented Programming) and Designing Pattern

Code: CBLM-OU-ICT-AMAD-02-L4-V1

National Skills Development Authority
Prime Minister's Office
Government of the People's Republic of Bangladesh

Copyright

National Skills Development Authority
Prime Minister's Office
Level: 10-11, Biniyog Bhaban,
E-6 / B, Agargaon, Sher-E-Bangla Nagar Dhaka-1207, Bangladesh.
Email: ec@nsda.gov.bd
Website: www.nsda.gov.bd.
National Skills Portal: http:\\skillsportal.gov.bd

This Competency Based Learning Materials (CBLM) on "Work with OOP (Object Oriented Programming) and Design Pattern" under the Android Mobile Application Development, Level-4" qualification is developed based on the national competency standard approved by National Skills Development Authority (NSDA)

This document is to be used as a key reference point by the competency-based learning materials developers, teachers/trainers/assessors as a base on which to build instructional activities.

It serves as the document for providing training consistent with the requirements of industry in order to meet the qualification of individuals who graduated through the established standard via competency-based assessment for a relevant job.

This document has been developed by NSDA with the assistance of related specialist/trainer /related employee**.**

Public and private institutions may use the information contained in this CBLM for activities benefitting Bangladesh.

Approved by __ th Authority Meeting of NSDA Held on --------------

How to use this Competency Based Learning Material (CBLM)

The module, Work with OOP (Object Oriented Programming) and Design Pattern contains training materials and activities for you to complete. These activities may be completed as part of structured classroom activities or you may be required you to work at your own pace. These activities will ask you to complete associated learning and practice activities in order to gain knowledge and skills you need to achieve the learning outcomes.

1. Review the **Learning Activity** page to understand the sequence of learning activities you will undergo. This page will serve as your road map towards the achievement of competence.

2. Read the **Information Sheets.** This will give you an understanding of the jobs or tasks you are going to learn how to do. Once you have finished reading the **Information Sheets** complete the questions in the **Self-Check.**

3. **Self-**Checks are found after each **Information Sheet**. **Self-Checks** are designed to help you know how you are progressing. If you are unable to answer the questions in the **Self-Check** you will need to re-read the relevant **Information Sheet**. Once you have completed all the questions check your answers by reading the relevant **Answer Keys** found at the end of this module.

4. Next move on to the **Job Sheets. Job Sheets** provide detailed information about *how to do the job* you are being trained in. Some **Job Sheets** will also have a series of **Activity Sheets**. These sheets have been designed to introduce you to the job step by step. This is where you will apply the new knowledge you gained by reading the Information Sheets. This is your opportunity to practise the job. You may need to practice the job or activity several times before you become competent.

5. Specification **sheets**, specifying the details of the job to be performed will be provided where appropriate.

6. A review of competency is provided on the last page to help remind if all the required assessment criteria have been met. This record is for your own information and guidance and is not an official record of competency

When working though this Module always be aware of your safety and the safety of others in the training room. Should you require assistance or clarification please consult your trainer or facilitator.

When you have satisfactorily completed all the Jobs and/or Activities outlined in this module, an assessment event will be scheduled to assess if you have achieved competency in the specified learning outcomes. You will then be ready to move onto the next Unit of Competency or Module

Table of Contents

Module Content

Unit of Competency	**Work with OOP (Object Oriented Programming) and Design Pattern**
Unit Code	**OU-ICT-AMAD-02-L4-V1**
Module Title	**Work with OOP (Object Oriented Programming) and Design Pattern**
Module Descriptor	This module covers the knowledge, skills and attitudes required to work with OOP and Design Pattern It includes the task of describe class properties, constants and visibility, apply encapsulation, apply inheritance, use Polymorphism and use design pattern
Nominal Hours	45 Hours
Lerning Outcome	After completing the practice of the module, the trainees will be able to perform the following jobs: 1. Describe class properties, constants, and visibility 2. Apply encapsulation 3. Apply inheritance 4. Use Polymorphism 5. Use design pattern

Assessment Criteria

1. Kotlin built-in methods is explained.
2. Field, property, and method inside a class are interpreted.
3. Advantages and limitations of OOP are described.
4. Encapsulation is described.
5. Language mechanism is explained for restricting access to object component.
6. Language construction is explained which facilitates bundling of data.
7. Association relationship is defined.
8. Association relationship between two classes is created.
9. Data and its functionality are encapsulated.
10. The inheritance is explained.
11. Types of inheritance are identified.
12. Subclasses and super classes of inheritance are explained.
13. Essence of inheritance relationship is described.
14. Inheritance relationship between classes is created.
15. Inheritances vs sub typing is explained.
16. Static Polymorphism is used.
17. Dynamic Polymorphism is applied.
18. Design pattern categories is explained
19. Aspects of design is recognized
20. Design pattern is applied

Learning Outcome 1: Describe Class Properties, Constants, And Visibility

Assessment Criteria	1. Kotlin built-in methods is explained. 2. Field, property, and method inside a class are interpreted. 3. Advantages and limitations of OOP are described.
Conditions and Resources	• Actual workplace or training environment • CBLM • Handouts • Job related tools, equipment, and materials • Multimedia Projector • Paper, Pen, Pencil, and Eraser • Internet Facilities • Whiteboard and Marker
Contents	1. Kotlin built-in methods ▪ String Methods ▪ Number Methods ▪ Character Methods ▪ Array Methods ▪ Object method ▪ List method 2. Field, property, and method inside a class 3. Advantages of OOP ▪ Code reusability ▪ Memory and performance management ▪ Data access restriction providing better data security. ▪ Class hierarchies are helpful in the design process allowing increased extensibility. ▪ Modularity ▪ Data abstraction 4. Limitations of OOP ▪ Lengthy learning process ▪ Requires intensive testing procedures. 5. Field, property, and method inside a class
Activities/job/Task	1. Write a Kotlin program that uses at least three different built-in methods. For example, create a string, convert it to uppercase, find its length, and print the result. 2. Create a Kotlin class named Person with fields for name and age. Implement a property for is Adult that returns true if the age is 18 or older. Also, create a method print Details that prints the person's name and age.

	3. Discuss a real-world scenario where OOP would be advantageous and another scenario where it might not be the best fit. Consider factors like system complex
Training Methods	• Blended • Discussion • Presentation • Demonstration • Guided Practice • Individual Practice • Project Work • Problem Solving • Brainstorming
Assessment Methods	Assessment methods may include but not limited to • Written Test • Demonstration • Oral Questioning

Learning Experience 1: Describe Class properties, Constants, and Visibility

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials Describe class properties, constants, and visibility
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Describe class properties, constants, and visibility"	2. Read Information sheet 1: Describe class properties, constants, and visibility 3. Answer Self-check 1: Describe class properties, constants, and visibility 4. Check your answer with Answer key 1: Describe class properties, constants, and visibility
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet **Job Sheet 1.1:** Write a Kotlin program that uses at least three different built-in methods. For example, create a string, convert it to uppercase, find its length, and print the result. **Specification Sheet 1.1:** Write a Kotlin program that uses at least three different built-in methods. For example, create a string, convert it to uppercase, find its length, and print the result. **Job Sheet 1.2:** Create a Kotlin class named Person with fields for name and age. Implement a property for isAdult that returns true if the age is 18 or older. Also, create a method print Details that prints the person's name and age. **Specification Sheet 1.2:** Create a Kotlin class named Person with fields for name and age. Implement a property for isAdult that returns true if the age is 18 or older. Also, create a method print Details that prints the person's name and age.

	Job Sheet 1.3: Discuss a real-world scenario where OOP would be advantageous and another scenario where it might not be the best fit. Consider factors like system complexity, maintenance, and scalability in your discussion. **Specification Sheet 1.3**: Discuss a real-world scenario where OOP would be advantageous and another scenario where it might not be the best fit. Consider factors like system complexity, maintenance, and scalability in your discussion.

Information Sheet 1: Describe class properties, constants, and visibility

Learning Objective: After completion of this information sheet, the learners will be able to explain, define and interpret the following contents

1.1 Kotlin built-in methods
1.2 Field, property, and method inside a class
1.3 Advantages of OOP
1.4 Limitations of OOP

1.1 Kotlin built-in methods

Kotlin is a modern programming language that provides a variety of built-in methods to simplify common tasks and enhance the development experience. These methods are categorized into different classes, functions, and extensions. Some of the essential built-in methods in Kotlin include:

a. String Methods
b. Number Methods
c. Character Methods
d. Array Methods
e. Object method
f. List method Etc.

a. String Methods

String methods refer to a collection of functions or operations that can be applied to strings in programming languages. These methods are specifically designed to manipulate, modify, or extract information from string data types. They allow developers to perform various operations on strings, such as transforming their appearance, extracting substrings, or checking specific conditions.

Conversion: Convert a string to uppercase or lowercase using *`toUpperCase()`* and *`toLowerCase()`* methods, respectively.
Manipulation: Modify strings by replacing, trimming, or splitting them using methods like *`replace()`, `trim()`, and `split()`*.
Extraction: Extract specific parts of a string, such as a substring or a portion between two indices, using methods like *`substring()` and `slice()`*.
Comparison: Compare strings for equality, similarity, or order using methods like *`equals()`, `contains()`, and `compareTo()`*.
Formatting: Apply formatting to strings, such as adding padding or justification, using methods like *`padStart()`, `padEnd()`, and `align()`*.
Concatenation: Combine multiple strings into a single string using methods like `plus()` and `plusAssign()`.

Here is a simple Kotlin program that demonstrates the usage of some string methods:

```kotlin
fun main() {
    val myString = "Hello, World!"

    // Convert to uppercase
    val uppercaseString = myString.toUpperCase()
    println("Uppercase: $uppercaseString")

    // Convert to lowercase
    val lowercaseString = myString.toLowerCase()
    println("Lowercase: $lowercaseString")

    // Replace "World" with "Kotlin"
    val replacedString = myString.replace("World", "Kotlin")
    println("Replaced: $replacedString")

    // Trim leading and trailing spaces
    val trimmedString = myString.trim()
    println("Trimmed: $trimmedString")

    // Split the string into a list of words
    val words = myString.split(" ")
    println("Words: $words")
}
```

When you run this program, it will produce the following output

```
Uppercase: HELLO, WORLD!
Lowercase: hello, world!
Replaced: Hello, Kotlin!
Trimmed: Hello, World!
Words: [Hello, World!]
```

This program demonstrates the usage of various string methods, such as `toUpperCase()`, `toLowerCase()`, `replace()`, `trim()`, and `split()`. It showcases how these methods can be used to manipulate and transform string data in Kotlin.
Number Methods

b. Number Methods

Kotlin number data types are used to define variables which hold numeric values and they are divided into two groups: **(i) Integer types** store whole numbers, positive or negative **(ii) Floating point types** represent numbers with a fractional part, containing one or more decimals.

Following table list down all the Kotlin number data types, keywords to define their variable types, size of the memory taken by the variables, and a value range which can be stored in those variables.

Data Type	Size (bits)	Data Range
Byte	8 bit	-128 to 127
Short	16 bit	-32768 to 32767
Int	32 bit	-2,147,483,648 to 2,147,483,647
Long	64 bit	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
Float	32 bit	1.40129846432481707e-45 to 3.40282346638528860e+38
Double	64 bit	4.94065645841246544e-324 to 1.79769313486231570e+308

If we will try to store a value more than permitted value in a variable of particular data type, the Kotlin compiler will complain because an overflow would occur at runtime.

Example
Following example shows how to define and access different Kotlin number data types

```
fun main(args: Array<String>) {
        val a: Int = 10000
         val d: Double = 100.00
         val f: Float = 100.00f
        val l: Long = 1000000004
        val s: Short = 10
        val b: Byte = 1

         println("Int Value is " + a)
          println("Double  Value is " + d)
        println("Float Value is " + f)
        println("Long Value is " + l )
        println("Short Value is " + s)
        println("Byte Value is " + b)
}
```

When you run the above Kotlin program, it will generate the following output:

```
Int Value is 10000
Double  Value is 100.0
Float Value is 100.0
```

```
Long Value is 1000000004
Short Value is 10
Byte Value is 1
```

c. Character Methods

Kotlin character data type is used to store a single character and they are represented by the type Char keyword. A Char value must be surrounded by single quotes, like 'A' or '1'.

Example

Following example shows how to define and access a Kotlin Char data type:

```
fun main(args: Array<String>) {
  val letter: Char    // defining a Char variable
  letter = 'A'        // Assigning a value to it
  println("$letter")
}
```

When you run the above Kotlin program, it will generate the following output: A

Kotlin supports several escape sequences of characters. When a character is preceded by a backslash (\), it is called an escape sequence and it has a special meaning to the compiler. For example, \n in the following statement is a valid character and it is called a new line character

```
println('\n') //prints a newline character
println('\$') //prints a dollar $ character
println('\\') //prints a back slash \ character
```

The following escape sequences are supported in Kotlin: \t, \b, \n, \r, \', \", \\ and \$.

d. Array Methods

Kotlin arrays are a collection of homogeneous data. Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
To create an array, use the *arrayOf()* function, and place the values in a comma-separated list inside it:

```
val cars = arrayOf("Volvo", "BMW", "Wafiq", "Parves")
```

Access the Elements of an Array

You can access an array element by referring to the index number, inside square brackets. In this example, we access the value of the first element in cars:

Example

```
val cars = arrayOf("Volvo", "BMW", "Wafiq", "Parves")
println(cars[0])
// Outputs Volvo
```

To change the value of a specific element, refer to the index number:

Example

```
val cars = arrayOf("Volvo", "BMW", "Wafiq", "Parves")
cars[0] = "Opel"
println(cars[0])
// Now outputs Opel instead of Volvo
```

Array Length / Size

To find out how many elements an array has, use the size property

Example

```
val cars = arrayOf("Volvo", "BMW", "Wafiq", "Parves")
println(cars.size)
// Outputs 4
```

Check if an Element Exists

You can use the in operator to check if an element exists in an array:

Example

```
val cars = arrayOf("Volvo", "BMW", "Ford", "Mazda")
if ("Volvo" in cars) {
  println("It exists!")
} else {
  println("It does not exist.")
}
```

e. Object method

The objects are created from the Kotlin class and they share the common properties and behaviours defined by a class in form of data members (properties) and member functions (behaviours) respectively.
The syntax to declare an object of a class is

```
var varName = ClassName()
```

We can access the **properties** and methods of a class using the **. (dot)** operator as shown below:

```
var varName = ClassName()
varName.property = <Value>
varName.functionName()
```

Example

Following is an example where we will create one Kotlin class and its object through which we will access different data members of that class.

```
class myClass {
   // Property (data member)
   private var name: String = "Tutorialspoint.com"
   // Member function
   fun printMe() {
      print("The best Learning website - " + name)
   }
}
fun main(args: Array<String>) {
   val obj = myClass() // Create object obj of myClass 
class
   obj.printMe() // Call a member function using object
}
```

The above piece of code will yield the following output in the browser, where we are calling **printMe()** method of **myClass** with the help of its own object **obj**.

Multiple Objects

You can create multiple objects of one class:

Example

```
val c1 = Car()
c1.brand = "waifq"
c1.model = "Mustang"
c1.year = 1969

val c2 = Car()
c2.brand = "BMW"
c2.model = "X5"
c2.year = 1999

println(c1.brand)  // waifq
println(c2.brand)  // BMW
```

Output:
```
waifq
BMW
```

f. List method

Kotlin list is an ordered collection of items. A Kotlin list can be either mutable (mutableListOf) or read-only (listOf). The elements of list can be accessed using indices. Kotlin mutable or immutable lists can have duplicate elements.

Creating Kotlin Lists

For list creation, use the standard library functions *listOf()* for read-only lists and *mutableListOf()* for mutable lists.

Example

```
fun main() {
    val theList = listOf("one", "two",
"three","four")
    println(theList)

    val theMutableList = mutableListOf("one", "two",
"three", "four")
    println(theMutableList)
}
```

Output:

```
[one, two, three, four]
[one, two, three, four]
```

Using Iterator

Example

```
fun main() {
   val theList = listOf("one", "two", "three", "four")
      val itr = theList.listIterator()
   while (itr.hasNext()) {
      println(itr.next())
   }
}
```

Output:

```
one
two
three
four
```

List Addition

We can use + operator to add two or more lists into a single list. This will add second list into first list, even duplicate elements will also be added.

Example

```
fun main() {
val firstList = listOf("one", "two", "three")
val secondList = listOf("four", "five", "six")
val resultList = firstList + secondList
println(resultList)
                }
```

Output: [one, two, three, four, five, six]

Filtering Elements

We can use filter() method to filter out the elements matching with the given predicate.

Example

```
fun main() {
   val theList = listOf(10, 20, 30, 31, 40, 50, -1, 0)
   val resultList = theList.filter{ it > 30}

   println(resultList)
}
```

Output: [31, 40, 50]

Dropping First N Elements

We can use drop() method to drop first N elements from the list.

Example

```
fun main() {

   val theList = listOf(10, 20, 30, 31, 40, 50, -1, 0)

   val resultList = theList.drop(3)

    println(resultList)

}
```

Output: [31, 40, 50, -1, 0]

Grouping List Elements

We can use groupBy() method to group the elements matching with the given predicate.

Example

```
    fun main() {
    val theList = listOf(10, 12, 30, 31, 40, 9, -3, 0)
    val resultList = theList.groupBy{ it % 3}
   println(resultList)
   }
```

Output: {1=[10, 31, 40], 0=[12, 30, 9, -3, 0]}

Mapping List Elements

We can use map() method to map all elements using the provided function

Example

```
   fun main() {
 val theList = listOf(10, 12, 30, 31, 40, 9, -3, 0)
 val resultList = theList.map{ it / 3 }
   println(resultList)
          }
```

Output: [3, 4, 10, 10, 13, 3, -1, 0]

1.2 Field, property, and method inside a class

Field: A field is a variable or attribute that is associated with an object or class. It holds a specific value and can be accessed by other parts of the program. Fields are used to store data related to the object or class.

To create a simple Kotlin program using fields, let's create a class called "Person" with fields for name, age, and occupation. We'll also include getter and setter methods for each field.

Example:

```
class Person(var name: String, var age: Int, var occupation: String)
fun main() {
  val person = Person("John Doe", 30, "Software Engineer")
    println("Name: ${person.name}, Age: ${person.age}, Occupation:
${person.occupation}")
```

```
    person.name = "Jane Doe"
    person.age = 35
    person.occupation = "Project Manager"
    println("Updated Name: ${person.name}, Age: ${person.age}, Occupation: ${person.occupation}")
}
```

Compile and run the program. You should see the following output:

```
Name: John Doe, Age: 30, Occupation: Software Engineer
Updated Name: Jane Doe, Age: 35, Occupation: Project Manager
```

Note:

- In the code above, we defined a class called "Person" with three fields: "name" (String), "age" (Int), and "occupation" (String). We used the `var` keyword to make these fields mutable, meaning their values can be changed after they are initialized.
- Inside the `main` function, we created a new instance of the `Person` class and assigned initial values to its fields. We then printed the initial values using string interpolation.
- We updated the values of the `person` object's fields and printed the updated values.

Property: In object-oriented programming languages like C# or Java, a property is a combination of a field and a method that allows controlled access to the field's value. Properties are used to encapsulate data and provide a way to get and set the value of a field. They can also be used to add validation or perform other actions when the value is being accessed or modified.

While methods define the actions that a class can perform, the properties define the class's characteristics or data attributes. For example, a smart device has these properties:

- **Name.** Name of the device.
- **Category.** Type of smart device, such as entertainment, utility, or cooking.
- **Device status**. Whether the device is on, off, online, or offline. The device is considered online when it's connected to the internet. Otherwise, it's considered offline.

Properties are basically variables that are defined in the class body instead of the function body. This means that the syntax to define properties and variables are identical. You define an immutable property with the val keyword and a mutable property with the var keyword.

Implement the afore mentioned characteristics as properties of the Smart Device class: On the line before the turnOn()method, define the name property and assign it to an "Android TV" string:

Example:

```
class SmartDevice {

   val name = "Android TV"
   val category = "Entertainment"
   var deviceStatus = "online"

   fun turnOn() {
      println("Smart device is turned on.")
   }

   fun turnOff() {
      println("Smart device is turned off.")
   }
}
```

Output:

```
Smart device is turned on.
Smart device is turned off.
```

On the line after the name property, define the category property and assign it to an "Entertainment" string, and then define a deviceStatus property and assign it to an "online" string:

Example

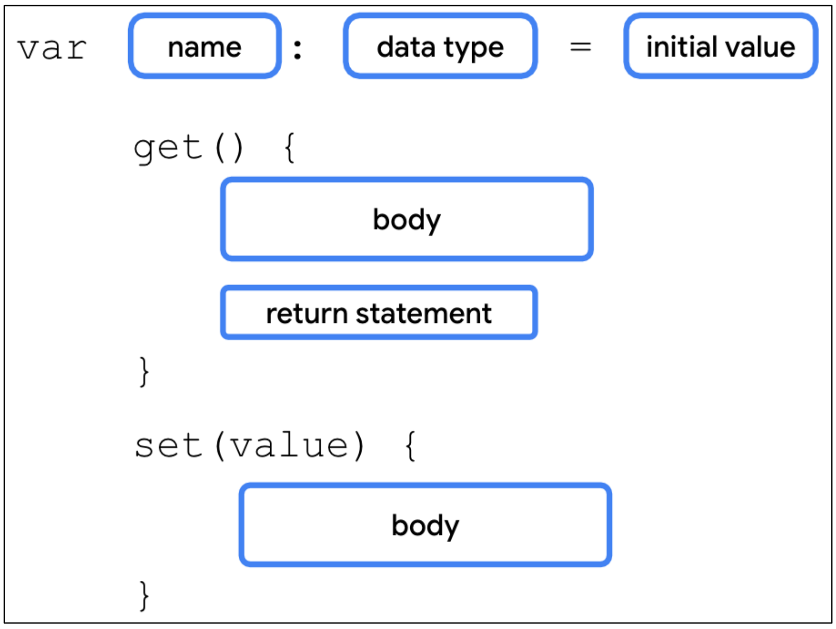
```
class SmartDevice {

   val name = "Android TV"
   val category = "Entertainment"
   var deviceStatus = "online"

   fun turnOn() {
      println("Smart device is turned on.")
   }

   fun turnOff() {
      println("Smart device is turned off.")
   }
}
```

Output:

```
Smart device is turned on.
Smart device is turned off.
```

Getter and setter functions in properties

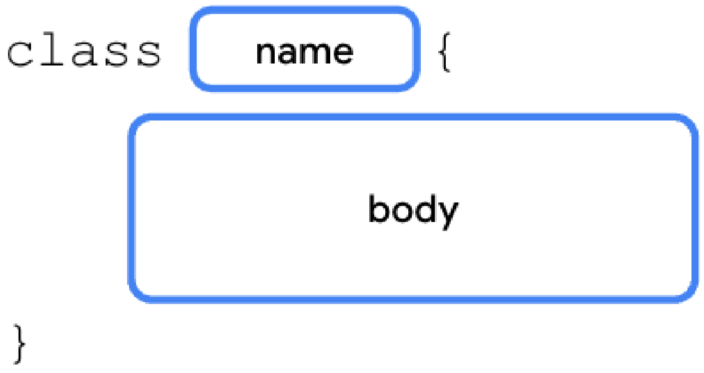
Properties can do more than a variable can. For example, imagine that you create a class structure to represent a smart TV. One of the common actions that you perform is increase and decrease the volume. To represent this action in programming, you can create a property named “speakerVolume”, which holds the current volume level set on the TV speaker, but there is a range in which the value for volume resides. The minimum volume one can set is 0, while the maximum is 100. To ensure that the “speakerVolume” property never exceeds 100 or falls below 0, you can write a setter function. When you update the value of the property, you need to check whether the value is in the range of 0 to 100. As another example, imagine that there is a requirement to ensure that the name is always in uppercase. You can implement a getter function to convert the name property to uppercase.



When you do not define the getter and setter function for a property, the Kotlin compiler internally creates the functions. For example, if you use the var keyword to define a “speakerVolume” property and assign it a 2 value, the compiler autogenerates the getter and setter functions as you can see in this code snippet:

```
var speakerVolume = 2
  get() = field
  set(value) {
    field = value
  }
```

Define Classes When you define a class, you specify the properties and methods that all objects of that class should have. A class definition starts with the class keyword, followed by a name and a set of curly braces. The part of the syntax before the opening curly brace is also referred to as the class header. In the curly braces, you can specify properties and functions for the class. You learn about properties and functions soon. You can see the syntax of a class definition in this diagram



These are the recommended naming conventions for a class:

- You can choose any class name that you want, but do not use Kotlin keywords as a class name, such as the fun keyword.
- The class name is written in PascalCase, so each word begins with a capital letter and there are no spaces between the words. For example, in SmartDevice, the first letter of each word is capitalized and there is not a space between the words.

A class consists of three major parts:

- **Properties.** Variables that specify the attributes of the class's objects.
- **Methods.** Functions that contain the class's behaviors and actions.
- **Constructors.** A special member function that creates instances of the class throughout the program in which it is defined.

This is not the first time that you have worked with classes. In previous codelabs, you learned about data types, such as the Int, Float, String, and Double data types. These data types are defined as classes in Kotlin. When you define a variable as shown in this code snippet, you create an object of the Int class, which is instantiated with a 1 value:

```
val number: Int = 1
```

Define a SmartDevice class:
In Kotlin Playground, replace the content with an empty main() function:

```
fun main() {
}
```

On the line before the main() function, define a SmartDevice class with a body that includes an // empty body comment:

```
class SmartDevice {
   // empty body
}
fun main() {
  }
```

1.3 Advantages of OOP

a. Code Reusability

Objects can be reused in different parts of a program or even in different programs altogether. This reusability saves time and effort as developers can leverage existing code rather than reinventing the wheel.

b. Memory and performance management

in object-oriented programming (OOP) are essential considerations to ensure efficient utilization of system resources and optimal program execution. Here are some aspects related to memory and performance management in OOP:

c. Memory Allocation and Deallocation

OOP languages typically handle memory allocation and deallocation automatically through mechanisms like garbage collection (e.g., in Java, C#) or automatic memory management (e.g., in Python). This relieves developers from manually managing memory and reduces the risk of memory leaks and dangling pointers.

However, it's important to be mindful of memory usage patterns, especially in resource-constrained environments or when dealing with large datasets. In such cases, explicit memory management techniques may be employed (e.g., C++ with `new` and `delete`).

Object Creation and Destruction

- Creating and destroying objects can have performance implications, especially in scenarios where objects are frequently instantiated and discarded.
- Object pools or caching mechanisms can be employed to reuse objects instead of recreating them from scratch, reducing the overhead of memory allocation and deallocation.
- Object destruction should be handled efficiently to release resources promptly and avoid memory leaks. Destructors or finalizers may be used to perform cleanup operations when objects are no longer needed.

Optimizing Data Structures and Algorithms

- Choosing appropriate data structures and algorithms can significantly impact the performance of OOP programs.
- Understanding the characteristics and usage patterns of different data structures (e.g., arrays, linked lists, hash maps, trees) helps in selecting the most suitable ones for a given problem.

- Algorithms should be designed to minimize time complexity (e.g., using efficient sorting algorithms) and space complexity (e.g., avoiding unnecessary memory overhead).

Minimizing Object Overhead

- Each object in OOP comes with its own overhead, including metadata for type information, v-table pointers (in languages with virtual functions), and memory padding for alignment.
- Minimizing the size of objects (e.g., by using efficient data types, avoiding unnecessary fields) reduces memory consumption and can improve cache locality, leading to better performance.

Avoiding Performance Antipatterns

- Certain coding practices in OOP can lead to performance bottlenecks. For example, excessive use of inheritance and deep class hierarchies can impact method dispatching efficiency.
- Virtual function calls, while providing polymorphic behavior, incur overhead compared to non-virtual calls. Careful use of virtual functions is necessary in performance-critical sections of code.
- Iterating over large collections of objects can be inefficient if not done properly. Using optimized iteration techniques (e.g., iterators, range-based loops) and considering parallelism where applicable can improve performance.

Profiling and Optimization

- Profiling tools can help identify performance hotspots in OOP code by analyzing runtime behavior, memory usage, and execution times of different functions and methods.
- Optimization strategies such as algorithmic optimizations, parallelization, and memory usage optimizations can be applied based on profiling results to address performance bottlenecks and improve overall efficiency.

d. Data access restriction providing better data
security. Data access restriction indeed plays a significant role in enhancing data security in object-oriented programming (OOP). Here's how:

Encapsulation

- Encapsulation, one of the key principles of OOP, involves bundling data (attributes) and methods (functions) that operate on the data into a single unit (an object) and restricting access to the inner workings of the object.
- By encapsulating data within objects and providing controlled access through methods (getters and setters), OOP languages like Java, C++, and Python enforce data hiding, preventing direct manipulation of internal state from outside the object.

- This helps in protecting sensitive data from unauthorized access and manipulation, thereby enhancing data security.

Access Control Modifiers

- OOP languages provide access control modifiers (e.g., private, protected, public) that allow developers to specify the visibility and accessibility of class members (attributes and methods).
- Private members can only be accessed within the same class, while protected members can also be accessed within subclasses. Public members are accessible from anywhere.
- By restricting access to sensitive data through private or protected members, OOP promotes the principle of least privilege, limiting exposure to potential security vulnerabilities.

Inheritance and Polymorphism

- Inheritance allows subclasses to inherit properties and methods from super classes. While inheritance can facilitate code reuse and promote modularity, it also raises concerns about data security.
- OOP languages offer mechanisms to control access to inherited members, allowing developers to selectively expose or hide inherited data and behaviors.
- Polymorphism allows objects of different classes to be treated interchangeably through a common interface. Access restrictions enforced by the superclass apply to polymorphic objects, ensuring consistent data security across different object types.

Data Validation and Sanitization

- OOP encourages encapsulating data validation and sanitization logic within object methods, ensuring that data integrity and security constraints are enforced at the point of entry.
- By encapsulating validation logic within objects, developers can prevent invalid or malicious data from corrupting internal state or compromising system security.

Information Hiding

- OOP promotes information hiding by exposing only essential interfaces and concealing implementation details. This reduces the attack surface and limits the exposure of sensitive information to potential attackers.
- By hiding implementation details behind well-defined interfaces, OOP mitigates the risk of information leakage and protects against reverse engineering and exploitation of system vulnerabilities.

e. Class hierarchies

A class hierarchy has parent classes and child classes. Child classes can inherit data and methods from parent classes, modify these data and methods, and add their own data and methods.

f. Modularity

Modularity OOP breaks down a program into smaller, more manageable chunks called objects. This modularity makes it easier to maintain, update, and debug code as changes can be made to one part of the code without affecting others.

g. Data abstraction

Data abstraction is a programming and design tool that displays basic information about a device while hiding its internal functions. Users can look at an interface and determine how a machine works, but they are unable to see how the machine can respond to their commands. In other words, they can understand what a machine does, but not how it is doing it.

For example, when using a cell phone, you can figure out how to answer incoming calls and respond to text messages. Thanks to data abstraction, you cannot tell how the phone itself transmits signals. The purpose of data abstraction is to expose only the essential elements of a device.

Types of data abstraction

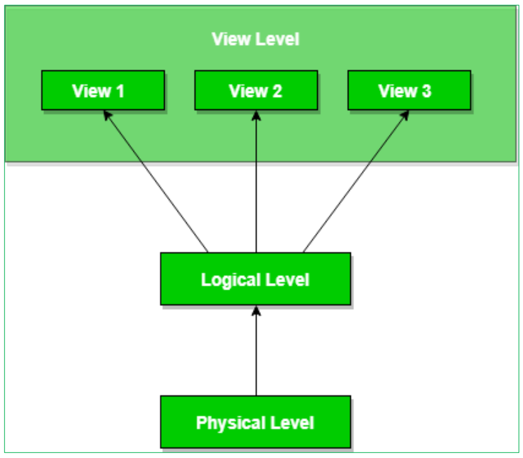
There are two types of data abstraction. They include:

Abstraction using classes: A class organizes the data into categories. With access specifiers, the classes determine which functions users can see and which functions remain hidden.

Abstraction in header files: Header files obscure all the inner functions from the user.

There are also three layers of data abstraction. They include:

Physical: The physical layer is the lowest level of data abstraction. It dictates the way a system stores the data.



Logical: The logical level indicates the specific types of data in the storage and the connections between the data. Professionals may look at the logical layer to determine what data to keep.

View: The view layer represents the highest level of data abstraction. It explains a portion of the entire database, allowing professionals to access the information they need.

1.4 Limitations of OOP

While object-oriented programming (OOP) offers numerous advantages, it does come with certain limitations. Here are two common limitations associated with OOP:

a. Lengthy Learning Process

- OOP introduces several new concepts, such as classes, objects, inheritance, polymorphism, and encapsulation, which can be challenging for beginners to grasp.
- Understanding the principles and best practices of OOP may require a significant investment of time and effort, especially for programmers transitioning from procedural or other paradigms.
- Learning to design effective class hierarchies, manage object relationships, and utilize advanced OOP features can further increase the learning curve.

b. Requires Intensive Testing Procedures

- While OOP promotes modularity and code reuse, it also introduces complexities that can make testing more challenging.
- Inheritance and polymorphism, for example, can lead to unexpected behaviors if not carefully designed and tested.
- Unit testing individual classes and integration testing interactions between classes become crucial to ensure the correctness and reliability of the software.
- Additionally, changes to one part of the codebase may have unintended consequences on other parts, necessitating thorough regression testing to catch any regressions.

Self-Check - 1: Describe class properties, constants, and visibility

1. Kotlin Built-in Methods

   Answer:
2. What is a built-in method in Kotlin?

   Answer:
3. Name a commonly used built-in method for string manipulation in Kotlin.

   Answer:
4. What built-in method in Kotlin is used to check if a collection is empty?

   Answer:
5. **Q4: Which built-in method can be used to filter elements in a collection?

   Answer:
6. How do you convert a string to uppercase in Kotlin?

   Answer:
7. What is a field in a class?

   Answer:
8. What is a property in Kotlin?

   Answer:
9. How is a mutable property defined in Kotlin?

   Answer:
10. How is an immutable property defined in Kotlin?

    Answer:
11. What is the difference between a method and a function in the context of classes?

    Answer:
12. What is encapsulation?

    Answer:
13. What advantage does inheritance provide in OOP?

    Answer:
14. Define polymorphism.

    Answer: What does abstraction help achieve in OOP?

    Answer:
15. Mention one performance-related limitation of OOP.

    Answer:

Answer Key - 1: Describe Class properties, Constants, and Visibility

1. Kotlin Built-in Methods

   Answer: Kotlin provides several built-in methods to make programming more efficient and concise. These methods are part of the Kotlin Standard Library and cover a wide range of functionalities, from collections to string manipulation and more.

2. What is a built-in method in Kotlin?

   Answer: A built-in method in Kotlin is a pre-defined function provided by the Kotlin Standard Library to perform common tasks.

3. Name a commonly used built-in method for string manipulation in Kotlin.

   Answer: `substring()`

4. What built-in method in Kotlin is used to check if a collection is empty?

   Answer: `isEmpty()`

5. Which built-in method can be used to filter elements in a collection?

   Answer: `filter()`

6. How do you convert a string to uppercase in Kotlin?

   Answer: Using the built-in method `toUpperCase()`

7. What is a field in a class?

   Answer: A field is a variable that is a member of a class or object.

8. What is a property in Kotlin?

   Answer: A property in Kotlin combines a field with getter and setter methods.

9. How is a mutable property defined in Kotlin?

   Answer: Using the keyword `var`.

10. How is an immutable property defined in Kotlin?

    Answer: Using the keyword `val`.

11. What is the difference between a method and a function in the context of classes?

    Answer: A method is a function defined within a class that operates on instances of that class.

12. What is encapsulation?

Answer: Encapsulation is the bundling of data with the methods that operate on that data.

13. What advantage does inheritance provide in OOP?

    Answer: Inheritance allows the creation of a new class using the properties and methods of an existing class.

14. Define polymorphism.

    Answer: Polymorphism is the ability to present the same interface for different underlying forms (data types).

15. What does abstraction help achieve in OOP?

    Answer: Abstraction helps in hiding complex implementation details and showing only the necessary features.

16. Mention one performance-related limitation of OOP.

    Answer: Object creation and garbage collection can add overhead.

Job Sheet-1.1: Write a Kotlin program that uses at least three different built-in methods. For example, create a string, convert it to uppercase, find its length, and print the result.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Create a new Kotlin program.
10. Define a string variable str with a value.
11. Use the toUpperCase() method to convert the string to uppercase.
12. Use the length property to get the length of the string.
13. Use the println() function to print the result.
14. Print the Values to the Console:
15. Show your work to the trainer.
16. Close IntelliJ IDEA, Android Studio as per standard procedure.
17. Clean your Work Place as per standard procedure.
18. Turn off the computer and clean your workplace.

Specification Sheet-1.1: Write a Kotlin program that uses at least three different built-in methods. For example, create a string, convert it to uppercase, find its length, and print the result.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Job Sheet-1.2: Create a Kotlin class named Person with fields for name and age. Implement a property for is Adult that returns true if the age is 18 or older. Also, create a method print Details that prints the person's name and age.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Create a new Kotlin program.
10. Define a string variable str with a value.
11. Use the toUpperCase() method to convert the string to uppercase.
12. Use the length property to get the length of the string.
13. Use the println() function to print the result.
14. Print the Values to the Console:
15. Show your work to the trainer.
16. Close IntelliJ IDEA, Android Studio as per standard procedure.
17. Clean your Work Place as per standard procedure.
18. Turn off the computer and clean your workplace.

Specification Sheet-1.2: Create a Kotlin class named Person with fields for name and age. Implement a property for is Adult that returns true if the age is 18 or older. Also, create a method print Details that prints the person's name and age.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Job Sheet-1.3: Discuss a real-world scenario where OOP would be advantageous and another scenario where it might not be the best fit. Consider factors like system complexity, maintenance, and scalability in your discussion.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. **Define the Problem:**
   a. Identify the requirements and functionalities needed.
   b. For an e-commerce platform: user management, product catalog, shopping cart, order processing.
10. **Plan the Classes:**
    a. Determine the main entities: User, Product, Order, ShoppingCart.
    b. Identify relationships and inheritance hierarchies.
11. **Design Class Structure:**
    a. Define properties and methods for each class.
    b. Plan for encapsulation, inheritance, and polymorphism.
12. **Implement Classes:**
    a. Write the class definitions in Kotlin.
    b. Implement properties, constructors, and methods.
13. **Test the System:**
    a. Create instances of classes and test interactions.
    b. Ensure all functionalities work as expected.
14. **Refactor and Optimize:**
    a. Review the code for improvements.
    b. Optimize for performance and readability.
15. **Deploy and Maintain:**
    a. Deploy the application.
    b. Regularly update and maintain the codebase.
16. Print the Values to the Console:
17. Show your work to the trainer.
18. Close IntelliJ IDEA, Android Studio as per standard procedure.
19. Clean your Work Place as per standard procedure.
20. Turn off the computer and clean your workplace.

Specification Sheet-1.3: Discuss a real-world scenario where OOP would be advantageous and another scenario where it might not be the best fit. Consider factors like system complexity, maintenance, and scalability in your discussion.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Learning Outcome 2: Apply encapsulation

Assessment Criteria	1. Encapsulation is described. 2. Language mechanism is explained for restricting access to object component. 3. Language construction is explained which facilitates bundling of data. 4. Association relationship is defined. 5. Association relationship between two classes is created. 6. Data and its functionality are encapsulated.
Conditions and Resources	• Actual workplace or training environment • CBLM • Handouts • Job related tools, equipment, and materials • Multimedia Projector • Paper, Pen, Pencil, and Eraser • Internet Facilities • Whiteboard and Marker Whiteboard and Marker
Contents	1. Encapsulation 2. Language mechanism 3. Language construction 4. Association relationship ▪ One-to-one ▪ One-to-many ▪ Many-to-many 5. Data and its functionality
Activities/job/Task	1. Create two Kotlin classes, Student and Course. Establish an association relationship between them, where a student can be associated with multiple courses, and a course can have multiple students. 2. Design a Kotlin class representing a Bank Account. Encapsulate the account balance as a private field and provide public methods for deposit, withdrawal, and checking the balance. Ensure proper encapsulation and validation of transactions.
Training Methods	• Blended • Discussion • Presentation • Demonstration • Guided Practice

	• Individual Practice • Project Work • Problem Solving • Brainstorming
Assessment Methods	Assessment methods may include but not limited to • Written Test • Demonstration • Oral Questioning

Learning Experience 2: Apply encapsulation

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials Apply encapsulation
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Apply encapsulation"	2. Read Information sheet 1: Apply encapsulation, and vi Apply encapsulation 3. Answer Self-check 1: Apply encapsulation 4. Check your answer with Answer key 1: Apply encapsulation
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet **Job Sheet 2.1:** Create two Kotlin classes, Student and Course. Establish an association relationship between them, where a student can be associated with multiple courses, and a course can have multiple students. **Specification Sheet 2.1:** Create two Kotlin classes, Student and Course. Establish an association relationship between them, where a student can be associated with multiple courses, and a course can have multiple students. **Job Sheet 2.2:** Design a Kotlin class representing a Bank Account. Encapsulate the account balance as a private field and provide public methods for deposit, withdrawal, and checking the balance. Ensure proper encapsulation and validation of transactions. **Specification Sheet 2.2:** Design a Kotlin class representing a Bank Account. Encapsulate the account balance as a private field and provide public methods for deposit, withdrawal, and checking the balance. Ensure proper encapsulation and validation of transactions.

Information Sheet 2: Apply encapsulation

Learning Objective: After completion of this information sheet, the learners will be able to explain, define and interpret the following contents

2.1 Encapsulation
2.2 Language mechanism
2.3 Language construction
2.4 Association relationship
2.5 Data and its functionality

2.1 Encapsulation

Encapsulation is a fundamental concept in object-oriented programming (OOP) that involves bundling or combining related data and methods (functions) into a single unit, known as a class. This technique helps in achieving data abstraction and hiding the internal details of an object, providing a more secure and organized way of managing data and functionality.

By encapsulating data and methods, you can control access to the internal components of a class by defining specific rules for accessing or modifying them. This is usually done through access modifiers (like public, private, or protected) that determine the visibility of class members. Encapsulation promotes data integrity, makes code more maintainable, and helps in creating a clear separation of concerns between different parts of a program.

Example

```
// Person class with encapsulation
class Person {
  private var name: String = ""
  private var age: Int = 0

  // Getter and setter for name
  fun getName(): String {
    return name
  }

  fun setName(newName: String) {
    name = newName
  }
  // Getter and setter for age
  fun getAge(): Int {
    return age
  }
  fun setAge(newAge: Int) {
    age = newAge
  }
}
fun main() {
    // Create a Person object
    val john = Person()

    // Set name and age using setter methods
    john.setName("John")
    john.setAge(30)

    // Retrieve name and age using getter methods
    println("Name: ${john.getName()}")
    println("Age: ${john.getAge()}")
}
```

Output :

```
Name: John
Age: 30
```

Example

Here is a Kotlin class representing a Bank Account with encapsulation and validation of transactions

```
class BankAccount {
    private var balance: Double = 0.0

    // Public method to deposit money into the account
    fun deposit(amount: Double) {
        if (amount > 0) {
            balance += amount
            println("Deposited $amount successfully. New balance: $balance")
        } else {
            println("Invalid deposit amount. Amount should be greater than zero.")
        }
    }

    // Public method to withdraw money from the account
    fun withdraw(amount: Double) {
        if (amount > 0 && amount <= balance) {
            balance -= amount
            println("Withdrew $amount successfully. New balance: $balance")
        } else {
            if (amount <= 0) println("Invalid withdrawal amount. Amount should
be greater than zero.")
            else println("Insufficient balance. Cannot withdraw $amount.")
        }
    }

    // Public method to check the account balance
    fun checkBalance(): Double {
        return balance
    }
}

// Example usage:
fun main() {
    val account = BankAccount()

    account.deposit(1000.0)
    account.withdraw(500.0)
    println("Current balance: ${account.checkBalance()}")
}
```

In this class, the account balance is encapsulated as a private field `balance`, which can only be accessed through public methods. The `deposit`, `withdraw`, and `checkBalance` methods ensure proper encapsulation and validation of transactions.

The `deposit` method checks if the amount to be deposited is greater than zero, and if so, adds the amount to the balance. The `withdraw` method checks if the amount to be withdrawn is greater than zero and less than or equal to the current balance. If both conditions are met, it subtracts the amount from the balance. If any of the conditions are not met, it displays an error message. The `checkBalance` method simply returns the current balance.

2.2 Language mechanism

Language mechanism, as explained earlier, refers to the structural and functional aspects of language that enable communication and understanding. Let's consider an example of a simple sentence and its breakdown in terms of grammar, vocabulary, syntax, and semantics.

Example: "The cat is chasing the mouse."

Grammar: The sentence follows the subject-verb-object (SVO) structure, which is a common pattern in English.

Vocabulary: The words "cat," "is," "chasing," "the," and "mouse" are used to convey the meaning.

Syntax: The arrangement of words in the sentence follows the rules of English grammar, with proper word order and punctuation.

Semantics: The meaning of the sentence is understood by the reader or listener, who can interpret the action and relationship between the cat and the mouse.

Now, let's create a simple Kotlin program that demonstrates the language mechanism in action by printing a greeting message.

```
fun main() {
    val name = "John" // vocabulary: name of a person
    val greeting = "Hello, " // grammar: subject-verb structure
    println("$greeting $name!") // syntax: proper word order and punctuation
        }
```

In this program, we use the vocabulary "Hello," "John," and a variable named "name" to store the person's name. The grammar structure includes a subject ("Hello") and a verb ("print"). The syntax is maintained by correctly placing the variables and strings within the println function.

2.3 Language Construction

Language construction involves designing or modifying a language to ensure effective communication and understanding. Let's consider an example of creating a simple programming language called "Kotlet" with basic commands.

Example: Kotlet Language

Commands

- PRINT: Displays a message on the screen.
- VAR: Declares a variable.
- SET: Assigns a value to a variable.
- ADD: Adds two numbers together.
- SUB: Subtracts one number from another.

```
/ PRINT command

fun printMessage(message: String) {

    println(message)

}

// VAR command

fun declareVariable(name: String) {

    println("VAR $name")

}

// SET command

fun setVariable
```

2.4 Association relationship

An Association relationship in object-oriented programming refers to a relationship between two or more classes where objects of those classes are connected or associated with each other. This relationship allows objects to interact and exchange data. One common example of an Association relationship is the "has-a" or "composition" relationship, where one class contains an instance of another class as a member.

Let's create a simple Kotlin program that demonstrates an Association relationship between a "Person" class and an "Address" class. In this example, a person can have an associated address.

Example

```
// Address class
data class Address(val street: String, val city: String, val zipCode: String)
// Person class with an associated Address
class Person(val name: String, val age: Int, val address: Address)
fun main() {
   // Create an Address object
   val myAddress = Address("123 Main St", "New York", "10001")
   // Create a Person object with the associated Address
   val john = Person("John", 30, myAddress)
   // Print the Person and Address details
   println("Person: ${john.name}, Age: ${john.age}")
   println("Address: ${john.address.street}, ${john.address.city},
${john.address.zipCode}")
}
```

When you run this program, the output will be:

```
Person: John, Age: 30
Address: 123 Main St, New York, 10001
```

One-to-One

An Association relationship in a one-to-one relationship means that two classes have a direct connection, where one instance of one class is related to one instance of the other class. In Kotlin, we can represent this association using reference variables.

Let's create a simple Kotlin program that demonstrates a one-to-one association between two classes, `Student` and `Address`. The `Student` class will have a reference to the `Address` class.

Example

```
// Address class
data class Address(val street: String, val city: String, val zipCode: String)
// Student class with an association to Address
class Student(val name: String, val age: Int, val address: Address)
fun main() {
   // Create an Address object
   val address = Address("123 Main St", "New York", "10001")
   // Create a Student object with the address
   val john = Student("John", 30, address)
   // Print the student's information
   println("Student Name: ${john.name}")
   println("Student Age: ${john.age}")
   println("Address:           ${john.address.street},           ${john.address.city},
${john.address.zipCode}")
}
```

In this program, we create a `data` class `Address` to represent a student's address. The `Student` class has a one-to-one association with `Address`, as it includes an `address` property of type `Address`. When we create a `Student` object, we pass an `Address` object to establish the association.

The output of this program will be

```
Student Name: John
Student Age: 30
Address: 123 Main St, New York, 10001
```

One-to-Many

In Kotlin, an Association relationship in a One-to-Many scenario can be represented using classes and lists or arrays. Let's create a simple example to demonstrate this relationship. Suppose we have two classes: `Course` and `Student`. A course can have many students enrolled in it, and each student can be enrolled in multiple courses.

```
// Course class
data class Course(val id: Int, val name: String)
// Student class
data   class   Student(val   id:   Int,   val   name:   String,   val   courses:
MutableList<Course>)
fun main() {
   // Create some courses
   val course1 = Course(1, "Kotlin Programming")
   val course2 = Course(2, "Data Structures")
   val course3 = Course(3, "Algorithms")
   // Create some students
```

```
    val student1 = Student(1, "John", mutableListOf(course1, course2))
    val student2 = Student(2, "Jane", mutableListOf(course2, course3))
    // Associate students with courses
    student1.courses.add(course3)
    student2.courses.add(course1)
    // Print the courses for each student
    println("Student 1: ${student1.courses.map { it.name }.joinToString(", ")}")
    println("Student 2: ${student2.courses.map { it.name }.joinToString(", ")}")
}
```

In this program, we define a `Course` class with `id` and `name` properties, and a `Student` class with `id`, `name`, and a `courses` property, which is a mutable list to represent the One-to-Many relationship.

When you run the program, the output will be:

```
Student 1: Kotlin Programming, Data Structures, Algorithms
Student 2: Data Structures, Algorithms, Kotlin Programming
```

Many-to-Many relationship

In a Many-to-Many relationship, two tables are directly related to each other, where one record in Table A can be related to multiple records in Table B, and vice versa. To represent this relationship in Kotlin, we can create two classes (let's call them `TableA` and `TableB`) and use a `List` or `Set` to store the associations between them.
Here's a simple Kotlin program demonstrating a Many-to-Many relationship between `TableA` and `TableB`:

```
// Define TableA class
data class TableA(val id: Int, val name: String)

// Define TableB class
data class TableB(val id: Int, val name: String)

// Define Association class
data class Association(val tableA: TableA, val tableB: TableB)

fun main() {
    // Create instances of TableA and TableB
    val tableA1 = TableA(1, "A1")
    val tableA2 = TableA(2, "A2")
    val tableB1 = TableB(1, "B1")
    val tableB2 = TableB(2, "B2")
    // Create associations between TableA and TableB
    val association1 = Association(tableA1, tableB1)
    val association2 = Association(tableA1, tableB2)
    val association3 = Association(tableA2, tableB1)

    // Store associations in lists
    val tableAAssociations = mutableListOf<Association>()
    tableAAssociations.add(association1)
    tableAAssociations.add(association2)
    tableAAssociations.add(association3)

    // Display associations for TableA
    println("Associations for TableA:")
    tableAAssociations.forEach { println("TableA: ${it.tableA}, TableB:
${it.tableB}") }
}
```

In this program, we define `TableA` and `TableB` classes using the `data` keyword, which automatically creates getter and setter methods for the properties. We also define an `Association` class to represent the relationship between `TableA` and `TableB`.

The `main` function creates instances of `TableA` and `TableB`, and then creates associations between them. These associations are stored in a `mutableListOf<Association>()`, which can be replaced with a `Set

2.5 Data and its functionality

Data and functionality can be encapsulated within classes, objects, functions, and other constructs. Here is a brief overview of how data and functionality are structured in Kotlin:

Data

- **Properties/Fields:** Data is stored in properties or fields, which can be declared within classes or objects. Properties can have getters and setters, or can be read-only (val) or mutable (var).
- **Data Classes:** Kotlin provides data classes, which are classes specifically designed to hold data. These classes automatically generate `equals()`, `hashCode()`, `toString()`, `copy()`, and `componentN()` functions based on the properties declared in the primary constructor.

Functionality

- **Functions:** Kotlin supports functions, which are blocks of code that can be called with specified parameters and can return a value. Functions can be declared at the top level, within classes, or within objects.
- **Methods:** Functions declared within classes or objects are referred to as methods. These methods can operate on the data stored in properties and provide functionality to manipulate that data.

- **Companion Objects:** Kotlin allows the declaration of companion objects within classes. These objects can hold functions and properties that are associated with the class itself rather than instances of the class.
- **Extension Functions:** Kotlin supports extension functions, which allow adding new functions to existing classes without modifying their source code. This is particularly useful for adding functionality to classes from external libraries or classes that you don't control.

Example

Here is an example demonstrating data and functionality in Kotlin

```
// Data class representing a person
data class Person(val name: String, var age: Int)

// Function to print person's details
fun Person.printDetails() {
    println("Name: $name, Age: $age")
}
// Extension function to check if person is an adult
fun Person.isAdult(): Boolean {
    return age >= 18
}
fun main() {
    val person = Person("Alice", 25)
    // Accessing properties
    println("Name: ${person.name}, Age: ${person.age}")
    // Modifying property
    person.age = 30
    println("Updated Age: ${person.age}")
    // Using method
    person.printDetails()
    // Using extension function
    println("Is adult: ${person.isAdult()}")
}
```

This example demonstrates:

- Declaration of a data class `Person` with properties `name` and `age`.
- Definition of a top-level function `printDetails()` to print person's details.
- Definition of an extension function `isAdult()` to check if a person is an adult.
- Usage of properties, methods, and extension functions within the `main()` function.

Self-Check Sheet - 2: Apply encapsulation

Questionnaire

1. What is encapsulation?

   Answer:

2. Name one language mechanism used for encapsulation in Kotlin.

   Answer:

3. What is a language construct in programming languages?

   Answer:

4. How is an association relationship represented in object-oriented programming?

   Answer:

5. Give an example of a one-to-one association relationship.

   Answer:

6. Provide an example of a one-to-many association relationship.

   Answer:

7. What is an example of a many-to-many association relationship?

   Answer:

8. How is data represented in programming languages?

   Answer:

9. What functionality does encapsulation provide in terms of data?

   Answer:

10. How do language mechanisms like access modifiers enhance data functionality?

    Answer:

Answer Key - 2: Apply encapsulation

1. What is encapsulation?
   Answer: Encapsulation is the principle of bundling data and methods that operate on the data within a single unit, typically a class. It restricts direct access to some of the object's components and allows access only through designated methods.
2. Name one language mechanism used for encapsulation in Kotlin.
   Answer: Access modifiers (e.g., \`private\`, \`protected\`, \`internal\`, \`public\`) are used in Kotlin to enforce encapsulation by controlling the visibility of classes, interfaces, properties, methods, and constructors.
3. What is a language construct in programming languages?
   Answer: A language construct refers to a syntactical element or feature provided by a programming language to perform specific tasks or represent certain programming concepts. Examples include variables, loops, conditionals, functions, classes, and interfaces.
4. How is an association relationship represented in object-oriented programming?
   Answer: An association relationship in object-oriented programming represents how objects are related to each other. It can be one-to-one, one-to-many, or many-to-many.
5. Give an example of a one-to-one association relationship.
   Answer: An example of a one-to-one association relationship is the relationship between a person and their social security number.
6. Provide an example of a one-to-many association relationship.
   Answer: An example of a one-to-many association relationship is the relationship between a teacher and their students.
7. What is an example of a many-to-many association relationship?
   Answer: A classic example of a many-to-many association relationship is the relationship between students and courses, where a student can enroll in multiple courses, and a course can have multiple students.
8. How is data represented in programming languages?
   Answer: Data in programming languages is represented using variables, constants, or data structures such as arrays, lists, maps, and custom-defined data types like classes or structs.
9. What functionality does encapsulation provide in terms of data?
   Answer: Encapsulation provides control over the access to data by hiding the internal state of an object and only allowing manipulation through well-defined methods. This ensures data integrity and facilitates modular programming.
10. How do language mechanisms like access modifiers enhance data functionality?
    Answer: Language mechanisms like access modifiers allow developers to specify the visibility and accessibility of data and methods. This ensures that data is accessed and modified only in ways that are safe and appropriate, thereby enhancing data functionality and maintaining code integrity.

Job Sheet-2.1: Create two Kotlin classes, Student and Course. Establish an association relationship between them, where a student can be associated with multiple courses, and a course can have multiple students.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. **Define the Student class**
   a. Create a new Kotlin file Student.kt and define the Student classPlan the Classes
10. **Define the Course class**
    a. Create a new Kotlin file Course.kt and define the Course class
11. **Establish the association relationship**
    In both classes, we have added a reference to the other class. In the Student class, we have a courses list to store multiple courses a student is enrolled in. In the Course class, we have a student's list to store multiple students enrolled in a course.
12. **Create instances of the classes**
    a. Create instances of the Student and Course classes:
13. **Verify the association relationship**
    a. Print the courses associated with each student and the students enrolled in each course:
14. Print the Values to the Console
15. Show your work to the trainer.
16. Close IntelliJ IDEA, Android Studio as per standard procedure.
17. Clean your Work Place as per standard procedure.
18. Turn off the computer and clean your workplace.

Specification Sheet-2.1: Create two Kotlin classes, Student and Course. Establish an association relationship between them, where a student can be associated with multiple courses, and a course can have multiple students.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Job Sheet-2.2: Design a Kotlin class representing a Bank Account. Encapsulate the account balance as a private field and provide public methods for deposit, withdrawal, and checking the balance. Ensure proper encapsulation and validation of transactions.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Create a Kotlin Program Encapsulate the account balance as a private field and provide public methods for deposit, withdrawal,
10. **Define the BankAccount class**
    a. Create a new Kotlin file BankAccount.kt and define the BankAccount class:
11. **Define the InsufficientFundsException exception**
    a. Create a new Kotlin file InsufficientFundsException.kt and define the InsufficientFundsException exception:
12. **Test the BankAccount class**
    a. Create a test class BankAccountTest.kt and test the BankAccount class:
13. **Verify the association relationship**
    a. Print the courses associated with each student and the students enrolled in each course:
14. Print the Values to the Console
15. Show your work to the trainer.
16. Close IntelliJ IDEA, Android Studio as per standard procedure.
17. Clean your Work Place as per standard procedure.

Turn off the computer and clean your workplace.

Specification Sheet-2.2 Design a Kotlin class representing a Bank Account. Encapsulate the account balance as a private field and provide public methods for deposit, withdrawal, and checking the balance. Ensure proper encapsulation and validation of transactions.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Learning Outcome 3: Apply inheritance

Assessment Criteria	1. The inheritance is explained. 2. Types of inheritance are identified. 3. Subclasses and super classes of inheritance are explained. 4. Essence of inheritance relationship is described. 5. Inheritance relationship between classes is created. 6. Inheritances vs sub typing is explained.
Conditions and Resources	• Actual workplace or training environment • CBLM • Handouts • Job related tools, equipment, and materials • Multimedia Projector • Paper, Pen, Pencil, and Eraser • Internet Facilities • Whiteboard and Marker Whiteboard and Marker
Contents	1. Types of inheritance 2. Subclasses and super classes of inheritance 3. Essence of inheritance relationship 4. Inheritances vs sub typing
Activities/job/Task	1. Define a Kotlin class hierarchy for a zoo. Create a base class named Animal with properties like name, age, and a method makeSound(). Implement two subclasses, Mammal and Bird, each with specific properties and sound implementations. Demonstrate the use of inheritance in creating instances of animals. 2. Create a Kotlin program that demonstrates subtyping without explicit inheritance. Define an interface named Drawable with a method draw(). Implement two classes, Circle and Square, that do not share a common inheritance but both implement the Drawable interface. Show how a function can accept both Circle and Square objects without explicitly inheriting from a common base class.
Training Methods	• Blended • Discussion • Presentation • Demonstration • Guided Practice • Individual Practice • Project Work • Problem Solving

	• Brainstorming
Assessment Methods	Assessment methods may include but not limited to • Written Test • Demonstration • Oral Questioning

Learning Experience 3: Apply inheritance

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials Apply encapsulation
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Apply inheritance"	2. Read Information sheet 3: Apply inheritance 3. Answer Self-check 3: Apply inheritance 4. Check your answer with Answer key 3: Apply inheritance
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet **Job Sheet 3.1:** Define a Kotlin class hierarchy for a zoo. Create a base class named Animal with properties like name, age, and a method makeSound(). Implement two subclasses, Mammal and Bird, each with specific properties and sound implementations. Demonstrate the use of inheritance in creating instances of animals. **Specification Sheet 3.1:** **Job Sheet 3.2:** Create a Kotlin program that demonstrates subtyping without explicit inheritance. Define an interface named Drawable with a method draw(). Implement two classes, Circle and Square, that do not share a common inheritance but both implement the Drawable interface. Show how a function can accept both Circle and Square objects without explicitly inheriting from a common base class.. **Specification Sheet 3.2** :

Information Sheet 3: Apply inheritance

Learning Objective: After completion of this information sheet, the learners will be able to explain, define and interpret the following contents

3.1 The inheritance.
3.2 Types of inheritance
3.3 Subclasses and super classes of inheritance
3.4 Essence of inheritance relationship
3.5 Inheritances vs sub typing

3.1 The Inheritance

Inheritance can be defined as the process where one class acquires the members (methods and properties) of another class. With the use of inheritance, the information is made manageable in a hierarchical order.

In Kotlin, inheritance is the mechanism by which a class can inherit properties and behavior (methods) from another class. Kotlin supports single class inheritance, meaning a class can inherit from only one superclass. Here's how you can declare a simple inheritance relationship:

```
/ Superclass
open class Animal(val name: String) {
   open fun makeSound() {
      println("$name makes a sound")
   }
}
// Subclass
class Dog(name: String) : Animal(name) {
   override fun makeSound() {
      println("$name barks")
   }
}
```

In this example:

- `Animal` is the superclass, and `Dog` is the subclass.
- `Dog` inherits from `Animal` using the colon (`:`) syntax.
- `open` keyword is used on the superclass to allow subclasses to override its members. Without `open`, the members cannot be overridden.
- `override` keyword is used on the subclass to indicate that it's overriding the `makeSound` function from the superclass.

You can then create instances of these classes and call their methods:

```
fun main() {
    val animal = Animal("Animal")
    animal.makeSound() // Output: Animal makes a sound

    val dog = Dog("Buddy")
    dog.makeSound() // Output: Buddy barks
}
```

Remember that Kotlin classes are `final` by default, meaning they cannot be subclassed unless explicitly marked with `open`. Similarly, functions and properties are also `final` by default and need to be marked with `open` if you want subclasses to override them.

3.2 Types of inheritance

Object-oriented programming languages, inheritance can take various forms. Here are the common types of inheritance supported in Kotlin:

a. Single Inheritance

- A subclass inherits from only one superclass.
- This is the simplest form of inheritance and is supported in Kotlin as demonstrated in the previous example.

```
open class Animal {
    // superclass
}

class Dog : Animal() {
    // subclass
}
```

b. Hierarchical Inheritance

- Multiple subclasses inherit from a single superclass.
- Each subclass may have its own unique behavior or attributes.

```
open class Animal {
    // superclass
}
class Dog : Animal() {
    // subclass
}

class Cat : Animal() {
    // another subclass
}
```

c. **Multilevel Inheritance:**
   - A subclass inherits from another subclass, creating a chain of inheritance.

```
open class Animal {
   // superclass
}

open class Mammal : Animal() {
   // subclass
}

class Dog : Mammal() {
   // subclass
}
```

d. **Multiple Inheritance through Interfaces:**
   - Kotlin does not support multiple class inheritance, but it does support multiple inheritance through interfaces.
   - A class can implement multiple interfaces, allowing it to inherit behavior and/or properties from each interface.

```
interface Swimmer {
   fun swim()
}

interface Walker {
   fun walk()
}

class Dog : Swimmer, Walker {
   override fun swim() {
      // implementation
   }

   override fun walk() {
      // implementation
   }
}
```

e. Hybrid Inheritance:

- A combination of two or more types of inheritance.
- For instance, a subclass may inherit from multiple interfaces and a superclass simultaneously.

```
open class Animal {
   // superclass
}

interface Swimmer {
   fun swim()
}

interface Walker {
   fun walk()
}

class Dolphin : Animal(), Swimmer {
   override fun swim() {
      // implementation
   }
}
```

3.3 Subclasses and super classes of inheritance

In Kotlin, when you talk about subclasses and superclasses in inheritance, you are referring to the relationships between classes where one class (the subclass) inherits from another class (the superclass). Let us delve into each:

a. Superclass

- A superclass is the class from which other classes inherit properties and behavior.
- It's also called a base class or a parent class.
- Superclasses are declared using the `open` keyword if you intend for them to be subclassed.

```
open class Animal(val name: String) {
   fun makeSound() {
      println("$name makes a sound")
   }
}
```

b. Subclass

- A subclass is a class that inherits properties and behavior from a superclass.
- It can add its own additional properties and behavior, and override existing ones if they are marked as `open`.
- Subclasses are declared by specifying the superclass in the class declaration.

```
class Dog(name: String) : Animal(name) {
    override fun makeSound() {
        println("$name barks")
    }
}
```

In this example, `Animal` is the superclass, and `Dog` is the subclass. `Dog` inherits from `Animal` by using the colon (`:`) followed by the superclass name.

You can create instances of the subclass and use its inherited behavior, along with any additional behavior defined in the subclass:

```
fun main() {

    val animal = Animal("Animal")

    animal.makeSound() // Output: Animal makes a sound

    val dog = Dog("Buddy")

    dog.makeSound() // Output: Buddy barks

}
```

This illustrates the relationship between a superclass and its subclass. The subclass `Dog` inherits the `name` property and `makeSound()` function from the superclass `Animal`. It can also override the `makeSound()` function to provide its own implementation.

3.4 Essence of inheritance relationship

The essence of inheritance in Kotlin lies in the ability to create hierarchical relationships between classes, allowing for code reuse and polymorphic behavior. Let's illustrate this essence with a simple program:

Suppose we're building a basic system to manage different types of vehicles. We have a superclass `Vehicle`, and two subclasses `Car` and `Motorcycle`, each inheriting from `Vehicle`. Here's how we can structure the classes:

```
// Superclass
open class Vehicle(val brand: String, val model: String) {
    open fun start() {
        println("$brand $model starts")
    }
    open fun stop() {
        println("$brand $model stops")
    }
}
// Subclasses
class Car(brand: String, model: String) : Vehicle(brand, model) {
    override fun start() {
        println("$brand $model starts like a car")
    }
    override fun stop() {
        println("$brand $model stops like a car")
    }
}
class Motorcycle(brand: String, model: String) : Vehicle(brand, model) {
    override fun start() {
        println("$brand $model starts like a motorcycle")
    }
    override fun stop() {
        println("$brand $model stops like a motorcycle")
    }
}
```

In this example

- `Vehicle` is the superclass representing common properties and behavior of all vehicles.
- `Car` and `Motorcycle` are subclasses that inherit from `Vehicle`.
- Each subclass can override the `start()` and `stop()` methods with its own specific implementation.

Now, let's use these classes

```
fun main() {
    val car = Car("Toyota", "Corolla")
    car.start() // Output: Toyota Corolla starts like a car
    car.stop()  // Output: Toyota Corolla stops like a car
    val motorcycle = Motorcycle("Honda", "CBR")
    motorcycle.start() // Output: Honda CBR starts like a motorcycle
    motorcycle.stop()  // Output: Honda CBR stops like a motorcycle
}
```

Here is the essence of the inheritance relationship:

- Both `Car` and `Motorcycle` inherit common properties and behavior from `Vehicle`, such as the `brand` and `model` properties, as well as the `start()` and `stop()` methods.
- Each subclass can provide its own specialized behavior by overriding methods from the superclass.

- This allows for code reuse and flexibility. You can treat objects of `Car` and `Motorcycle` types as `Vehicle`s, enabling polymorphism. For example, you can pass a `Car` object to a function expecting a `Vehicle` parameter.

This essence of inheritance facilitates the creation of a hierarchical structure, where subclasses specialize or extend the behavior of their superclasses, promoting code organization and reusability.

3.5 Inheritances vs sub typing

In Kotlin, inheritance and subtyping are related concepts but serve slightly different purposes:

a. Inheritance

- Inheritance refers to the mechanism by which a class can inherit properties and behavior from another class, known as its superclass.
- It allows for code reuse and promotes a hierarchical organization of classes.
- In Kotlin, classes can inherit from only one superclass, supporting single inheritance.
- Inheritance is primarily used to model an "is-a" relationship, where a subclass is a specialized version of its superclass.
-

Example

```
open class Animal {
    // superclass
}
class Dog : Animal() {
    // subclass
}
```

b. Subtyping

- Subtyping refers to the relationship between types, where a value of one type can be substituted for a value of another type based on their relationship.
- In Kotlin, subtyping is closely related to the concept of type hierarchies created through inheritance.
- Subtyping allows for polymorphism, where objects of different subtypes can be treated interchangeably based on their common supertype.
- Kotlin supports both class inheritance and interface inheritance, enabling subtyping through classes and interfaces.

Example

```
open class Animal {
   fun makeSound() {
      println("Animal makes a sound")
   }
}
class Dog : Animal() {
   override fun makeSound() {
      println("Dog barks")
   }
}
fun main() {
   val animal: Animal = Dog()
   animal.makeSound() // Output: Dog barks
}
```

In this example, `Dog` is a subtype of `Animal`, meaning a `Dog` object can be treated as an `Animal`. This is made possible through inheritance and subtyping.

In summary, inheritance is a mechanism for code reuse and hierarchy organization, while subtyping deals with the relationship between types and enables polymorphism and interchangeability of objects based on their type hierarchy. In Kotlin, inheritance and subtyping often go hand in hand, with inheritance facilitating the creation of type hierarchies that enable subtyping relationships.

Self-Check Sheet - 3: Apply inheritance

Questionnaire

1. What are the different types of inheritance in Kotlin?

   Answer:

2. Define subclasses and super classes in the context of inheritance.

   Answer:

3. What is the essence of the inheritance relationship?

   Answer:

4. What is the difference between inheritance and subtyping?

   Answer:

5. What is the main purpose of using inheritance in object-oriented programming?

   Answer:

6. Does Kotlin support multiple inheritance from classes?

   Answer:

7. How do you indicate that a class can be inherited from in Kotlin?

   Answer:

8. Explain how method overriding works in Kotlin.

   Answer:

9. Why do we use interfaces along with inheritance in Kotlin?

   Answer:

10. Demonstrate hierarchical inheritance with a simple Kotlin code snippet.

    Answer:

Answer Key - 3: Apply inheritance

1. What are the different types of inheritance in Kotlin?

   Answer: In Kotlin, there are five types of inheritance: single inheritance, multi-level inheritance, hierarchical inheritance, multiple inheritance through interfaces, and hybrid inheritance (a combination of multiple and hierarchical).

2. Define subclasses and super classes in the context of inheritance.

   Answer: In inheritance, a subclass (or derived class) is a class that inherits properties and behavior from another class, known as the superclass (or base class).

3. What is the essence of the inheritance relationship?

   Answer: The essence of inheritance is the ability of a subclass to inherit properties and methods from its superclass, allowing for code reuse, extension, and specialization.

4. What is the difference between inheritance and subtyping?

   Answer: Inheritance is a mechanism for code reuse and extension, where a subclass inherits properties and behavior from its superclass. Subtyping, on the other hand, is a relationship between types, where a subtype can be used wherever its supertype is expected.

5. What is the main purpose of using inheritance in object-oriented programming?
   Answer: The primary purpose of inheritance is to promote code reuse and abstraction. It allows for the creation of specialized classes that inherit common properties and behavior from a superclass.

6. Does Kotlin support multiple inheritance from classes?

   Answer: No, Kotlin does not support multiple inheritance from classes. However, it supports multiple inheritance through interfaces, where a class can implement multiple interfaces.

7. How do you indicate that a class can be inherited from in Kotlin?

   Answer: In Kotlin, the `open` keyword is used to mark a class as inheritable. Classes without the `open` keyword are final by default and cannot be inherited from.

8. Explain how method overriding works in Kotlin.
   Answer: To override a method in Kotlin, you use the `override` keyword in the subclass. This indicates that the method in the subclass is replacing the implementation of the method in the superclass.

9. Why do we use interfaces along with inheritance in Kotlin?

Answer: Interfaces allow for multiple inheritance in Kotlin, as a class can implement multiple interfaces. This promotes code flexibility and enables classes to exhibit multiple behaviors.

10. Demonstrate hierarchical inheritance with a simple Kotlin code snippet.

Answer:

```
open class Animal {
    fun eat() {
        println("Animal is eating")
    }
}
class Dog : Animal() {
    fun bark() {
        println("Dog is barking")
    }
}
class Cat : Animal() {
    fun meow() {
        println("Cat is meowing")
    }
}
```

In this example, both `Dog` and `Cat` inherit from the `Animal` class, showcasing hierarchical inheritance.

Job Sheet-3.1: Define a Kotlin class hierarchy for a zoo. Create a base class named Animal with properties like name, age, and a method makeSound(). Implement two subclasses, Mammal and Bird, each with specific properties and sound implementations. Demonstrate the use of inheritance in creating instances of animals.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Create a Kotlin Program class hierarchy for a zoo
10. **Define the Base Class Animal**
    a. First, we will create the base class Animal with properties name and age, and a method makeSound().Plan the Classes:
11. **Define the Subclass Mammal**
    a. Next, we will create a subclass Mammal that extends Animal. This class can have additional properties specific to mammals, such as furColor.
12. **Define the Subclass Bird**
    a. Now, we will create a subclass Bird that extends Animal. This class can have additional properties specific to birds, such as wingSpan.Test the System:
13. **Demonstrate the Use of Inheritance**
    a. Finally, we will create instances of Mammal and Bird and demonstrate the use of inheritance by calling their methods and accessing their properties.
14. Run the Program and show output
15. Show your work to the trainer.
16. Close IntelliJ IDEA, Android Studio as per standard procedure.
17. Clean your Work Place as per standard procedure.
18. Turn off the computer and clean your workplace.

Specification Sheet-3.1: Define a Kotlin class hierarchy for a zoo. Create a base class named Animal with properties like name, age, and a method makeSound(). Implement two subclasses, Mammal and Bird, each with specific properties and sound implementations. Demonstrate the use of inheritance in creating instances of animals

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Job Sheet-3.2: Create a Kotlin program that demonstrates subtyping without explicit inheritance. Define an interface named Drawable with a method draw(). Implement two classes, Circle and Square, that do not share a common inheritance but both implement the Drawable interface. Show how a function can accept both Circle and Square objects without explicitly inheriting from a common base class.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Create a Kotlin program that demonstrates subtyping without explicit inheritance
10. **Define the Drawable interface**
    a. Create a new Kotlin file Drawable.kt and define the Drawable interface:
11. **Implement Circle class**
    a. Create a new Kotlin file Circle.kt and implement the Circle class:
12. **Implement Square class**
    a. Create a new Kotlin file Square.kt and implement the Square class:
13. **Create a function that accepts both Circle and Square objects**
    a. Create a new Kotlin file Main.kt and define a function that accepts both Circle and Square objects:
14. Run the Program and show output
15. Show your work to the trainer.
16. Close IntelliJ IDEA, Android Studio as per standard procedure.
17. Clean your Work Place as per standard procedure.
18. Turn off the computer and clean your workplace.

Specification Sheet-3.2 Create a Kotlin program that demonstrates subtyping without explicit inheritance. Define an interface named Drawable with a method draw(). Implement two classes, Circle and Square, that do not share a common inheritance but both implement the Drawable interface. Show how a function can accept both Circle and Square objects without explicitly inheriting from a common base class.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Learning Outcome 4: Use Polymorphism

Assessment Criteria	1. Static Polymorphism is used. 2. Dynamic Polymorphism is applied.
Conditions and Resources	• Actual workplace or training environment • CBLM • Handouts • Job related tools, equipment, and materials • Multimedia Projector • Paper, Pen, Pencil, and Eraser • Internet Facilities • Whiteboard and Marker Whiteboard and Marker
Contents	1. Static Polymorphism 2. Apply Dynamic Polymorphism
Activities/job/Task	1. Create a Kotlin class named MathOperations with multiple overloaded methods for addition. Implement methods for adding two integers, two doubles, and a combination of an integer and a double. Demonstrate the use of static polymorphism by calling these methods with different argument types. 2. Create a Kotlin class hierarchy representing geometric shapes, including a base class Shape with a method calculateArea(). Implement two subclasses, Circle and Rectangle, that override the calculateArea() method with specific formulas for each shape. Demonstrate the use of dynamic polymorphism by calling the calculateArea() method on instances of both subclasses.
Training Methods	• Blended • Discussion • Presentation • Demonstration • Guided Practice • Individual Practice • Project Work • Problem Solving • Brainstorming
Assessment Methods	Assessment methods may include but not limited to • Written Test • Demonstration • Oral Questioning

Learning Experience 4: Use Polymorphism

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials Apply encapsulation
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Use Polymorphism"	2. Read Information sheet 3: Use Polymorphism 3. Answer Self-check 3: Use Polymorphism 4. Check your answer with Answer key 3: Use Polymorphism
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet **Job Sheet 4.1:** Create a Kotlin class named MathOperations with multiple overloaded methods for addition. Implement methods for adding two integers, two doubles, and a combination of an integer and a double. Demonstrate the use of static polymorphism by calling these methods with different argument types. **Specification Sheet 4.2 :** Job Sheet 4.1 **Job Sheet 4.2:** Create a Kotlin class hierarchy representing geometric shapes, including a base class Shape with a method calculateArea(). Implement two subclasses, Circle and Rectangle, that override the calculateArea() method with specific formulas for each shape. Demonstrate the use of dynamic polymorphism by calling the calculateArea() method on instances of both subclasses. **Specification Sheet 4.2** : Job Sheet 4.2

Information Sheet 4: Use Polymorphism

Learning Objective: After completion of this information sheet, the learners will be able to explain, define and interpret the following contents

4.1 Static Polymorphism
4.2 Dynamic Polymorphism

4.1 Static Polymorphism

Static Polymorphism, also known as Compile-Time Polymorphism, is a concept in programming languages where a single method or function can take on multiple forms based on the context in which it is used. In Kotlin, static polymorphism is achieved primarily through inheritance and method overriding.

a. Inheritance

In Kotlin, classes can be derived from a parent class or interface, allowing the child class to inherit properties and methods from the parent. This inheritance relationship enables static polymorphism, as the child class can be treated as an instance of the parent class.

Example

```
open class Animal {
  fun makeNoise() {
    println("An animal makes a noise.")
  }
}
class Dog : Animal() {
  override fun makeNoise() {
    println("Dog barks.")
  }
}

fun main() {
  val animal: Animal = Dog()
  animal.makeNoise() // Output: Dog barks.
}
```

In this example, the `Dog` class is a child of the `Animal` class. The `makeNoise()` method is overridden in the `Dog` class, demonstrating static polymorphism.

b. Method Overriding

Method overriding is another way to achieve static polymorphism in Kotlin. When a subclass has a method with the same signature as a method in its parent class, it overrides the parent's method. The subclass's method is called when an object of the subclass is used, demonstrating polymorphic behavior.

Example:

```
open class Shape(open val name: String)
class Rectangle(override val name: String) : Shape(name)
class Square(name: String) : Rectangle(name) {
   override val name: String
      get() = "Square"
}
fun printShapeName(shape: Shape) {
   println("Shape Name: ${shape.name}")
}
fun main() {
   val rectangle = Rectangle("Rectangle")
   val square = Square("Square")

   printShapeName(rectangle) // Output: Shape Name: Rectangle
   printShapeName(square) // Output: Shape Name: Square
}
```

In this example, the `Square` class overrides the `name` property from the `Rectangle` class, which itself is a subclass of the `Shape` class. The `printShapeName()` function demonstrates static polymorphism by treating both `Rectangle` and `Square` objects as instances of the `Shape` class.

In summary, static polymorphism in Kotlin is achieved through inheritance and method overriding, allowing a single method or function to take on multiple forms based on the context in which it is used.

4.2 Apply Dynamic Polymorphism

Dynamic Polymorphism, also known as Runtime Polymorphism, is a concept in programming languages where the method to be executed is determined during the execution of a program, not at compile-time. In Kotlin, dynamic polymorphism is primarily achieved through function overriding and interfaces.

a. Function Overriding

In Kotlin, when a subclass has a function with the same name and parameters as a function in its parent class, it overrides the parent's function. The subclass's function is called when an object of the subclass is used, demonstrating dynamic polymorphism.

Example

```
open class Animal {
   fun speak(sound: String) {
      println("An animal speaks $sound.")
   }
}
class Dog : Animal() {
   override fun speak(sound: String) {
      when (sound) {
         "bark" -> println("Dog barks.")
         "woof" -> println("Dog woofs.")
         else -> println("Dog doesn't understand the sound.")
      }
   }
}
fun main() {
   val animal: Animal = Dog()
   animal.speak("bark") // Output: Dog barks.
}
```

In this example, the `Dog` class overrides the `speak()` function from the `Animal` class. When the `speak()` function is called on a `Dog` object, the overridden version in the `Dog` class is executed, demonstrating dynamic polymorphism.

b. Interfaces and Dynamic Polymorphism

Interfaces in Kotlin allow for the implementation of dynamic polymorphism by defining a contract that must be followed by any class implementing the interface. When a class implements an interface and provides its own implementation of the interface's functions, it demonstrates dynamic polymorphism.

Example

```
interface Flyable {
   fun fly()
}
class Bird : Flyable {
   override fun fly() {
      println("Bird flies.")
   }
}
class Penguin : Flyable {
   override fun fly() {
      println("Penguin can't fly, but swims.")
   }
}
fun showFlying(flyable: Flyable) {
   flyable.fly()
}
fun main() {
   val bird = Bird()
   val penguin = Penguin()

   showFlying(bird) // Output: Bird flies.
   showFlying(penguin) // Output: Penguin can't fly, but swims.
}
```

In this example, the `Flyable` interface defines a contract for flying. The `Bird` and `Penguin` classes implement the `Flyable` interface and provide their own implementations of the `fly()` function. The `showFlying()` function demonstrates dynamic polymorphism by treating both `Bird` and `Penguin`

Self-Check - 4: Use Polymorphism

Questionnaire

1. What is Static Polymorphism?

   Answer: Static Polymorphism, or Compile-Time Polymorphism, is a programming concept where a single method or function can take on multiple forms based on the context in which it is used. In Kotlin, it is primarily achieved through inheritance and method overriding.

2. How does Inheritance contribute to Static Polymorphism in Kotlin?

   Answer: In Kotlin, inheritance allows a child class to inherit properties and methods from a parent class. This enables static polymorphism, as a child class can be treated as an instance of the parent class.

3. Can you explain Method Overriding in Kotlin and its relation to Static Polymorphism?

   Answer: Method Overriding is when a subclass has a method with the same signature as a method in its parent class. It overrides the parent's method, demonstrating static polymorphism. The subclass's method is called when an object of the subclass is used.

4. What is Dynamic Polymorphism?

   Answer: Dynamic Polymorphism, or Runtime Polymorphism, is a programming concept where the method to be executed is determined during the execution of a program, not at compile-time. In Kotlin, it is primarily achieved through function overriding and interfaces.

5. How does Function Overriding contribute to Dynamic Polymorphism in Kotlin?

   Answer: In Kotlin, when a subclass has a function with the same name and parameters as a function in its parent class, it overrides the parent's function. The subclass's function is called when an object of the subclass is used, demonstrating dynamic polymorphism.

6. How do Interfaces support Dynamic Polymorphism in Kotlin?

   Answer: Interfaces in Kotlin allow for the implementation of dynamic polymorphism by defining a contract that must be followed by any class implementing the interface. When a class implements an interface and provides its own implementation of the interface's functions, it demonstrates dynamic polymorphism.

Answer Key - 4: Use Polymorphism

1. What is Static Polymorphism?
   Answer: Static Polymorphism, or Compile-Time Polymorphism, is a programming concept where a single method or function can take on multiple forms based on the context in which it is used. In Kotlin, it is primarily achieved through inheritance and method overriding.

2. How does Inheritance contribute to Static Polymorphism in Kotlin?
   Answer: In Kotlin, inheritance allows a child class to inherit properties and methods from a parent class. This enables static polymorphism, as a child class can be treated as an instance of the parent class.

3. Can you explain Method Overriding in Kotlin and its relation to Static Polymorphism?
   Answer: Method Overriding is when a subclass has a method with the same signature as a method in its parent class. It overrides the parent's method, demonstrating static polymorphism. The subclass's method is called when an object of the subclass is used.

4. What is Dynamic Polymorphism?
   Answer: Dynamic Polymorphism, or Runtime Polymorphism, is a programming concept where the method to be executed is determined during the execution of a program, not at compile-time. In Kotlin, it is primarily achieved through function overriding and interfaces.

5. How does Function Overriding contribute to Dynamic Polymorphism in Kotlin?
   Answer: In Kotlin, when a subclass has a function with the same name and parameters as a function in its parent class, it overrides the parent's function. The subclass's function is called when an object of the subclass is used, demonstrating dynamic polymorphism.

6. How do Interfaces support Dynamic Polymorphism in Kotlin?
   Answer: Interfaces in Kotlin allow for the implementation of dynamic polymorphism by defining a contract that must be followed by any class implementing the interface. When a class implements an interface and provides its own implementation of the interface's functions, it demonstrates dynamic polymorphism.

Job Sheet-4.1: Create a Kotlin class named MathOperations with multiple overloaded methods for addition. Implement methods for adding two integers, two doubles, and a combination of an integer and a double. Demonstrate the use of static polymorphism by calling these methods with different argument types.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Create a Kotlin class named MathOperations with multiple overloaded methods for addition
10. Define the MathOperations Class: Create a class named MathOperations.
11. Implement Overloaded Methods: Add methods for adding two integers, two doubles, and a combination of an integer and a double.
12. Demonstrate Static Polymorphism: Create an instance of MathOperations and call the overloaded methods with different argument types.
13. Run the Program and show output
14. Show your work to the trainer.
15. Close IntelliJ IDEA, Android Studio as per standard procedure.
16. Clean your Work Place as per standard procedure.
17. Turn off the computer and clean your workplace.

Specification Sheet-4.1: Create a Kotlin class named MathOperations with multiple overloaded methods for addition. Implement methods for adding two integers, two doubles, and a combination of an integer and a double. Demonstrate the use of static polymorphism by calling these methods with different argument types.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Job Sheet-4.2: Create a Kotlin class hierarchy representing geometric shapes, including a base class Shape with a method calculateArea(). Implement two subclasses, Circle and Rectangle, that override the calculateArea() method with specific formulas for each shape. Demonstrate the use of dynamic polymorphism by calling the calculateArea() method on instances of both subclasses.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Create a Kotlin class hierarchy representing geometric shapes, including a base class Shape with a method calculateArea()
10. **Define the Base Class Shape**
    a. Create an abstract class Shape with an abstract method calculateArea().
11. **Define the Circle Subclass**
    a. Create a subclass Circle that inherits from Shape.
    b. Add a property radius to the Circle class.
    c. Override the calculateArea() method to calculate the area of a circle.
12. **Define the Circle Subclass**
    a. Create a subclass Circle that inherits from Shape.
    b. Add a property radius to the Circle class.
    c. Override the calculateArea() method to calculate the area of a circle.
13. **Define the Circle Subclass**
    a. Create a subclass Circle that inherits from Shape.
    b. Add a property radius to the Circle class.
    c. Override the calculateArea() method to calculate the area of a circle.
14. Run the Program and show output
15. Show your work to the trainer.
16. Close IntelliJ IDEA, Android Studio as per standard procedure.
17. Clean your Work Place as per standard procedure.
18. Turn off the computer and clean your workplace.

Specification Sheet-4.2: Create a Kotlin class hierarchy representing geometric shapes, including a base class Shape with a method calculateArea(). Implement two subclasses, Circle and Rectangle, that override the calculateArea() method with specific formulas for each shape. Demonstrate the use of dynamic polymorphism by calling the calculateArea() method on instances of both subclasses.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Learning Outcome 5: Use design pattern

Assessment Criteria	1. Design pattern categories is explained 2. Aspects of design is recognized 3. Design pattern is applied
Conditions and Resources	• Actual workplace or training environment • CBLM • Handouts • Job related tools, equipment, and materials • Multimedia Projector • Paper, Pen, Pencil, and Eraser • Internet Facilities • Whiteboard and Marker Whiteboard and Marker
Contents	1. Design pattern o Singleton pattern o Builder pattern o Factory pattern o Observer pattern 2. Aspects of design 3. Apply design pattern
Activities/job/Task	3. Choose a design pattern (e.g., Observer, Factory Method, Strategy) and apply it to a specific problem or scenario. Provide a detailed implementation in Kotlin, explaining how the design pattern enhances the structure or behavior of the system. 4. Write a detailed explanation of the three main categories of design patterns: Creational, Structural, and Behavioral. Provide examples of design patterns within each category and describe scenarios where they are commonly used. 5. Identify and describe the key aspects of design, including scalability, maintainability, flexibility, and reusability. Discuss how each aspect contributes to the overall quality of a software design and provide examples of design decisions that address these aspects. 6. Choose a design pattern (e.g., Singleton, Observer, Factory Method) and apply it to a specific problem or scenario. Provide a detailed implementation in Kotlin, explaining how the design pattern enhances the structure or behavior of the system.
Training Methods	• Blended • Discussion

	• Presentation • Demonstration • Guided Practice • Individual Practice • Project Work • Problem Solving • Brainstorming
Assessment Methods	Assessment methods may include but not limited to • Written Test • Demonstration • Oral Questioning

Learning Experience 5: Use Design Pattern

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about the learning materials	1. Instructor will provide the learning materials Apply encapsulation
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Use Design Pattern"	2. Read Information sheet 3: Use Design Pattern 3. Answer Self-check 3: Use Design Pattern 4. Check your answer with Answer key 3: Use Design Pattern
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job/Task Sheet and Specification Sheet **Job Sheet 5.1:** Choose a design pattern (e.g., Observer, Factory Method, Strategy) and apply it to a specific problem or scenario. Provide a detailed implementation in Kotlin, explaining how the design pattern enhances the structure or behavior of the system. **Specification Sheet 5.4 :** **Job Sheet 5.2:** Write a detailed explanation of the three main categories of design patterns: Creational, Structural, and Behavioral. Provide examples of design patterns within each category and describe scenarios where they are commonly used. **Specification Sheet 5.4 :** **Job Sheet 5.3:** Identify and describe the key aspects of design, including scalability, maintainability, flexibility, and reusability. Discuss how each aspect contributes to the overall quality of a software design and provide examples of design decisions that address these aspects. **Specification Sheet 5.4 :**

Information Sheet 5: Use Design Pattern

Learning Objective: After completion of this information sheet, the learners will be able to explain, define and interpret the following contents

5.1 Design pattern categories
5.2 Aspects of design
5.3 Design pattern

5.1 Design Pattern

A design pattern is a general repeatable solution to a commonly occurring problem in software design. In this blog post, we will delve into various design patterns and explore how they can be effectively implemented in Kotlin.

a. Singleton pattern

The Singleton design pattern ensures that a class has only one instance and provides a global point of access to that instance. Every single place where it is used will make use of the same instance, hence reducing memory usage and ensuring consistency. It is useful when exactly one object is needed to coordinate actions across the system, such as managing a shared resource or controlling a single point of control (e.g., a configuration manager or a logging service). The pattern typically involves a private constructor, a method to access the instance, and lazy initialization to create the instance only when it's first requested.

In Kotlin, the Singleton design pattern can be implemented in several ways. Here are two common approaches.

Object Declaration
The most straightforward way to implement a Singleton in Kotlin is by using an object declaration. An object declaration defines a singleton class and creates an instance of it at the same time. The instance is created lazily when it's first accessed.

Here is a code example of using the object declaration methos:

```
object MySingleton {
   // Singleton properties and methods go here
   fun doSomething() {
      println("Singleton is doing something")
   }
}
To use our singleton:
MySingleton.doSomething()
```

Companion Object
Another approach is to use a companion object within a class. This approach allows us to have more control over the initialization process, and we can use it when we need to perform some additional setup.

Let's see how we can make use of the companion object method:

```
class MySingleton private constructor() {
   companion object {
      private val instance: MySingleton by lazy { MySingleton() }

      fun getInstance(): MySingleton {
         return instance
      }
   }
   // Singleton properties and methods go here
   fun doSomething() {
      println("Singleton is doing something")
   }
}
To use the singleton:
val singletonInstance = MySingleton.getInstance()
singletonInstance.doSomething()
```

By using by lazy, the instance is created only when it's first accessed, making it a lazy-initialized singleton.

b. Builder pattern

The Builder design pattern is used for constructing complex objects by separating the construction process from the actual representation. It is particularly useful when an object has many parameters, and we want to provide a more readable and flexible way to construct it.
Here is an example of implementing the Builder design pattern in Kotlin:

```
// Product class
data class Computer(
   val cpu: String,
   val ram: String,
   val storage: String,
   val graphicsCard: String
)
// Concrete builder class
class ComputerBuilder {
   private var cpu: String = ""
```

```
    private var ram: String = ""
    private var storage: String = ""
    private var graphicsCard: String = ""
    fun cpu(cpu: String): ComputerBuilder {
        this.cpu = cpu
        return this
    }
    fun ram(ram: String): ComputerBuilder {
        this.ram = ram
        return this
    }
    fun storage(storage: String): ComputerBuilder {
        this.storage = storage
        return this
    }
    fun graphicsCard(graphicsCard: String): ComputerBuilder {
        this.graphicsCard = graphicsCard
        return this
    }
    fun build(): Computer {
        return Computer(cpu, ram, storage, graphicsCard)
    }
}
fun main() {
    // Build the computer with a specific configuration
    val builder = ComputerBuilder()
    val gamingComputer = builder
        .cpu("Intel Core i9")
        .ram("32GB DDR4")
        .storage("1TB SSD")
        .graphicsCard("NVIDIA RTX 3080")
        .build()
}
```

In this code, the computer class serves as the product to be built, encapsulating attributes like CPU, RAM, storage, and graphics card. The Computer Builder interface declares methods for configuring each attribute, while the Computer Builder class implements this interface, progressively setting the values. In the client code within the main function, a Computer Builder instance is utilized to construct a computer object with a specific configuration by method chaining. This approach enhances readability and flexibility, especially when dealing with objects with numerous optional or interchangeable components, as the Builder pattern facilitates a step-by-step construction process.

Note that the Builder pattern is not as commonly used in Kotlin as it is in Java, for example, because Kotlin provides named parameters, which can be used in a constructor to a very similar effect to a builder:

```
fun main() {
    // Without using the Builder pattern
   val simpleComputer = Computer(
      cpu = "Intel Core i5",
      ram = "16GB DDR4",
      storage = "512GB SSD",
      graphicsCard = "NVIDIA GTX 1660"
   )
}
```

Using named parameters in a constructor as in the example above is better for null safety, because it does not accept null values and the values do not have to be set to empty strings ("") as in the Builder example.

c. Factory pattern

The Factory Design Pattern is a creational pattern that provides an interface for creating objects in a super class but allows subclasses to alter the type of objects that will be created. This pattern is often used when a class cannot anticipate the class of objects it must create.

Here is an example of a simple Factory Design Pattern in Kotlin:

```
// Product interface
interface Product {
   fun create(): String
}
// Concrete Product A
class ConcreteProductA : Product {
   override fun create(): String {
      return "Product A"
   }
}
// Concrete Product B
class ConcreteProductB : Product {
   override fun create(): String {
      return "Product B"
   }
}
// Factory interface
interface ProductFactory {
   fun createProduct(): Product
}
// Concrete Factory A
class ConcreteFactoryA : ProductFactory {
   override fun createProduct(): Product {
      return ConcreteProductA()
   }
}
// Concrete Factory B
class ConcreteFactoryB : ProductFactory {
   override fun createProduct(): Product {
      return ConcreteProductB()
   }
}
// Client code
fun main() {
   val factoryA: ProductFactory = ConcreteFactoryA()
   val productA: Product = factoryA.createProduct()
   println(productA.create())

   val factoryB: ProductFactory = ConcreteFactoryB()
   val productB: Product = factoryB.createProduct()
   println(productB.create())
}
```

In this example, we have a Product interface representing the product to be created. We have two concrete product classes, ConcreteProductA and ConcreteProductB, which implement the Product interface. We also have a ProductFactory interface with a method createProduct() and two concrete factory classes, ConcreteFactoryA and ConcreteFactoryB which implement this interface and return instances of the respective concrete products.

d. Observer pattern

The Observer design pattern is a behavioral design pattern where an object, known as the subject, maintains a list of its dependents, known as observers, that are notified of any state changes. This pattern is often used to implement distributed event handling systems.

Here is a simple example:

```
// Define an interface for the observer
interface Observer {
  fun update(value: Int)
}
// Define a concrete observer that implements the Observer interface
```

```
class ValueObserver(private val name: String) : Observer {
  override fun update(value: Int) {
    println("$name received value: $value")
  }
}
// Define a subject that emits values and notifies observers
class ValueSubject {
  private val observers = mutableListOf<Observer>()
  fun addObserver(observer: Observer) {
    observers.add(observer)
  }
  fun removeObserver(observer: Observer) {
    observers.remove(observer)
  }
  private val observable: Flow<Int> = flow {
    while (true) {
      emit(Random.nextInt(0..1000))
      delay(100)
    }
  }
  fun startObserving() {
    val observerJob = coroutineScope.launch {
      observable.collect { value ->
        notifyObservers(value)
      }
    }
  }
  private fun notifyObservers(value: Int) {
    for (observer in observers) {
      observer.update(value)
    }
  }
}
```

In summary, this code sets up a system where multiple observers can be attached to a subject ValueSubject. The subject emits random values in a continuous stream and each attached observer ValueObserver is notified whenever a new value is emitted. The observer then prints a message indicating that it received the new value.

Sure, let's explore some aspects of design in Kotlin along with corresponding code examples:

5.2 Aspects of design

a. Immutability

mutability ensures that once an object is created, its state cannot be changed.
Program Example:

```
data class ImmutablePerson(val name: String, val age: Int)
fun main() {
    val person = ImmutablePerson("Alice", 30)
    println(person)

    // Error: Cannot change properties of ImmutablePerson
    // person.age = 35
}
```

b. Null Safety

Kotlin's type system distinguishes between nullable and non-nullable types, reducing the risk of NullPointerExceptions.

Program Example:

```
fun main() {
    var nullableString: String? = "Hello"
    nullableString = null // Allowed

    val nonNullableString: String = "World"
    // Error: Null cannot be a value of a non-null type String
    // nonNullableString = null
}
```

c. Extension Functions

Extension functions allow adding new functionality to existing classes without modifying their source code.

Program Example:

```
fun String.addExclamation(): String {
    return "$this!"
}
fun main() {
    val greeting = "Hello"
    println(greeting.addExclamation()) // Prints "Hello!"
}
```

d. Sealed Classes

Sealed classes represent restricted class hierarchies, ensuring that all subclasses are known and defined within the same file.

Program Example:

```
sealed class Result
data class Success(val message: String) : Result()
data class Error(val errorMessage: String) : Result()

fun handleResult(result: Result) {
   when (result) {
      is Success -> println("Success: ${result.message}")
      is Error -> println("Error: ${result.errorMessage}")
   }
}
fun main() {
   val successResult = Success("Data loaded successfully")
   val errorResult = Error("Failed to load data")

   handleResult(successResult) // Prints "Success: Data loaded successfully"
   handleResult(errorResult)   // Prints "Error: Failed to load data"
}
```

e. Data Classes

Data classes automatically provide methods like `toString()`, `equals()`, `hashCode()`, and `copy()` based on properties defined in the primary constructor.

Program Example

```
data class Person(val name: String, val age: Int)
fun main() {
   val person1 = Person("Alice", 30)
   val person2 = Person("Alice", 30)
   println(person1 == person2) // Prints true (equals() comparison)
}
```

5.3 Apply design pattern

Following 5.1

Self-Check - 5: Use Design Pattern

Questionnaire

1. What is a design pattern?

   Answer:

2. Explain the Singleton pattern.

   Answer:

3. How does the Builder pattern work?

   Answer:

4. What is the purpose of the Factory pattern?

   Answer:

5. Describe the Observer pattern.

   Answer:

6. What are the aspects of design patterns?

   Answer:

7. When should you use the Singleton pattern?

   Answer:

8. How does the Builder pattern promote flexibility?

   Answer:

9. In what scenarios is the Factory pattern applicable?

   Answer:

Answer Key - 5: Use Design Pattern

1. What is a design pattern?

   Answer: A design pattern is a reusable solution to a commonly occurring problem in software design. It provides a general template for organizing code and managing data and objects in a specific way.

2. Explain the Singleton pattern.

   Answer: The Singleton pattern ensures that a class has only one instance throughout the application. It provides a global point of access to this unique instance, which can be useful for managing resources or maintaining a single state across the application.

3. How does the Builder pattern work?

   Answer: The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations. It uses a builder object to assemble the object step-by-step, providing flexibility in creating various object structures.

4. What is the purpose of the Factory pattern?

   Answer: The Factory pattern is used to hide the complexity of object creation and provide a simplified interface for clients to obtain objects. It defines an interface for creating objects, but the specific class of the object created is deferred until runtime.

5. Describe the Observer pattern.

   Answer: The Observer pattern establishes a one-to-many dependency between objects, so that when the state of one object changes, all its dependents are notified and updated automatically. It is useful for maintaining consistency between related objects and promoting loose coupling.

6. What are the aspects of design patterns?

   Answer: Aspects of design patterns include their purpose, structure, behavior, and relationships with other patterns. These aspects help developers understand and apply design patterns effectively in their software designs.

7. When should you use the Singleton pattern?

   Answer: Use the Singleton pattern when you need to ensure that only one instance of a class exists throughout the application, and you want to provide a global point of access to that instance. It is useful for managing resources, logging, or maintaining a single application state.

8. How does the Builder pattern promote flexibility?

   Answer: The Builder pattern promotes flexibility by separating the construction of a complex object from its representation. This separation allows clients to choose different representations of the same object structure without changing the construction process.

9. In what scenarios is the Factory pattern applicable?

   Answer: The Factory pattern is applicable when you have a family of related classes that should be treated uniformly, but you want to delay the decision about which specific class to instantiate until runtime. It helps simplify object creation and promote loose coupling.

Job Sheet-5.1: Choose a design pattern (e.g., Observer, Factory Method, Strategy) and apply it to a specific problem or scenario. Provide a detailed implementation in Kotlin, explaining how the design pattern enhances the structure or behavior of the system.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Create a Kotlin class Choose a design pattern (e.g., Observer, Factory Method, Strategy) and apply it to a specific problem or scenario
10. **Define the Observer interface**
    a. Create a new Kotlin file Observer.kt and define the Observer interface
11. **Implement the WeatherStation class**
    a. Create a new Kotlin file WeatherStation.kt and implement the WeatherStation class
12. **Implement the WeatherDisplay class**
    a. Create a new Kotlin file WeatherDisplay.kt and implement the WeatherDisplay class:
13. **Create instances and register observers**
    a. Create instances of the WeatherStation, WeatherDisplay, and register them with each other:
14. Run the Program and show output
15. Show your work to the trainer.
16. Close IntelliJ IDEA, Android Studio as per standard procedure.
17. Clean your Work Place as per standard procedure.
18. Turn off the computer and clean your workplace.

Specification Sheet-5.1: Choose a design pattern (e.g., Observer, Factory Method, Strategy) and apply it to a specific problem or scenario. Provide a detailed implementation in Kotlin, explaining how the design pattern enhances the structure or behavior of the system.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Job Sheet-5.2: Write a detailed explanation of the three main categories of design patterns: Creational, Structural, and Behavioral. Provide examples of design patterns within each category and describe scenarios where they are commonly used**.**

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Create a Kotlin a detailed explanation of the three main categories of design patterns
10. **Creational Patterns**
    Creational patterns focus on the creation of objects. They provide a way to create objects in a way that simplifies the process and makes it more maintainable. The main goal of creational patterns is to decouple object creation from the specific implementation details.
    a. Singleton: Ensures that only one instance of a class is created.
    b. Factory Method: Provides a way to create objects without specifying the exact class of object that will be created.
    c. Abstract Factory: Provides a way to create families of related objects without specifying the exact class of object that will be created.
11. **Structural Patterns**
    Structural patterns focus on the composition of classes and objects. They provide a way to compose objects to form larger structures and improve the flexibility and maintainability of the system
    a. Adapter: Converts one interface to another interface, allowing incompatible classes to work together.
    b. Composite: Allows clients to treat individual objects and compositions of objects uniformly.
    c. Bridge: Separates an object's abstraction from its implementation so that the two can vary independently.
12. **Behavioral Patterns**
    Behavioral patterns focus on the interactions between objects. They provide a way to define the behavior of objects and how they interact with each other.
    a. Observer: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.
    b. Observer: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

    c. Observer: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

13. Run the Program and show output
14. Show your work to the trainer.
15. Close IntelliJ IDEA, Android Studio as per standard procedure.
16. Clean your Work Place as per standard procedure.
17. Turn off the computer and clean your workplace.

Specification Sheet-5.2: Write a detailed explanation of the three main categories of design patterns: Creational, Structural, and Behavioral. Provide examples of design patterns within each category and describe scenarios where they are commonly used**.**

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Job Sheet-5.3: Identify and describe the key aspects of design, including scalability, maintainability, flexibility, and reusability. Discuss how each aspect contributes to the overall quality of a software design and provide examples of design decisions that address these aspects.

Working Procedure/ Steps:

1. Follow OSH and use Personal Protective Equipment (PPE).
2. Check Electricity & Internet Connections to your Computer.
3. Start the Computer.
4. Create a folder on the desktop with your registration number and name.
5. Collect the resources from your trainer as per the job requirement.
6. Open IntelliJ IDEA or android studio
7. Set Up Your IntelliJ IDEA Environment
8. Ensure you have a Kotlin compiler or an IDE like IntelliJ IDEA that supports Kotlin.
9. Identify and describe the key aspects of design, including scalability, maintainability, flexibility, and reusability
10. Understand Requirements: Gather and analyze the requirements to identify the core functionalities and constraints.
11. Design the System Architecture: Choose an appropriate architecture that supports scalability and flexibility.
12. Define the Class Hierarchy: Use OOP principles to define classes and relationships
13. Implement Key Features with Reusability in Mind: Write reusable functions and classes.
14. Incorporate Design Patterns: Apply design patterns to address maintainability and flexibility.
15. Test Thoroughly: Ensure the software works correctly and efficiently under different conditions.
16. Example: Unit tests and integration tests using Kotlin's testing libraries.
17. Document and Review: Maintain clear documentation and conduct code reviews to ensure code quality and maintainability.
18. Run the Program and show output
19. Show your work to the trainer.
20. Close IntelliJ IDEA, Android Studio as per standard procedure.
21. Clean your Work Place as per standard procedure.
22. Turn off the computer and clean your workplace.

Specification Sheet-5.3: Identify and describe the key aspects of design, including scalability, maintainability, flexibility, and reusability. Discuss how each aspect contributes to the overall quality of a software design and provide examples of design decisions that address these aspects.

Conditions for the job: Work must be carried out in a safe manner and according to relevant competency standards.

List of required PPE's

S/N	Name of PPE	Specification	Unit	Required Quantity
01	Ergonomic Chair	Wood and foam	No	1
02	Eye protective glass	Metal and Glass	No	1
03	Mask	Surgical mask	1	1

List of required Software

S/N	Name of Materials	Specification	Unit	Required Quantity
01	IntelliJ IDEA	Latest version	…	1
02	Android Studio	Latest version	…	1

List of required Tools & Equipment's

S/N	Name	Specification	Unit	Required Quantity
01	Personal Computer or Laptop	Minimum Core i5 7th gen Processor	No	1
04	Internet connection		…	1

Review of Competency

Below is yourself assessment rating for module "Work with OOP (Object Oriented Programming) and Design Pattern

Assessment of performance Criteria	Yes	No
Kotlin built-in methods is explained.		
Field, property and method inside a class are interpreted.		
Advantages and limitations of OOP are described.		
Encapsulation is described.		
Language mechanism is explained for restricting access to object component.		
Language construction is explained which facilitates bundling of data.		
Association relationship is defined.		
Association relationship between two classes are created.		
Data and its functionality is encapsulated.		
The inheritance is explained.		
Types of inheritance are identified.		
Subclasses and super classes of inheritance are explained.		
Essence of inheritance relationship is described.		
Inheritance relationship between classes is created.		
Inheritances vs sub typing is explained.		
Static Polymorphism is used.		
Dynamic Polymorphism is applied.		
Design pattern categories is explained		
Aspects of design is recognized		
Design pattern is applied		

I now feel ready to undertake my formal competency assessment.

Signed:

Date:

Development of CBLM

The Competency based Learning Material (CBLM) of '**Work with OOP (Object Oriented Programming) and Design Pattern' (Occupation: Android Mobile Application Development, Level-4)** for National Skills Certificate is developed by NSDA with the assistance of SIMEC System Ltd., ECF Consultancy & SIMEC Institute of Technology JV (Joint Venture Firm) in the month of July, 2024 under the contract number of package SD-9B dated 15th January 2024.

SL No.	Name & Address	Designation	Contact Number
1	Engr. Md. Zuwel Parves	Writer	01737-278906
2	Md. Abdul Al Hossain	Editor	01778-926438
3	Engr Md. Zuwel Parves	Co-Ordinator	01737-278906
4	Md. Abdur Razzaque	Reviewer	01713-304824

Reference

1. https://reflectoring.io/kotlin-design-patterns/
2. https://www.w3schools.com/kotlin/index.php
3. https://www.tutorialspoint.com/kotlin/index.htm
4. https://kotlinlang.org/docs/control-flow.html#for-loops
5. https://www.programiz.com/kotlin-programming/examples/factors-number