Chu Anh Lợi -20215280

Lesson 3.1  
[1. Terminology](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#0)

The following programming terms should already be familiar to you:

* *Classes* are blueprints for objects. For example, an Aquarium class is the blueprint for making an Aquarium object.
* *Objects* are instances of classes; an aquarium object is one actual Aquarium that exists in memory.
* *Properties* are characteristics of classes, such as the length, width, and height of an Aquarium.
* *Methods*, also called *member functions*, are the functionality of the class. Methods are what you can "do" with the object. For example, you can fillWithWater() an Aquarium object.
* An *interface* is a specification that a class can implement. For example, cleaning is common to objects other than aquariums, and cleaning generally happens in similar ways for different objects. So you could have an interface called Clean that defines a clean() method. The Aquarium class could implement the Clean interface to clean the aquarium with a soft sponge.
* *Packages* are a way to group related code to keep it organized, or to make a library of code. Once a package is created, you can use import to allow you to directly reference classes in that package.

## [2. Create a class](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#1)

Step 1: Create a package  
A screenshot of a computer

Description automatically generated

Step 2: Create a class with properties  
A screenshot of a computer

Description automatically generated

Step 3: Create a main() function

Define a buildAquarium() function and inside create an instance of Aquarium. To create an instance, reference the class as if it were a function, Aquarium(). This calls the constructor of the class and creates an instance of the Aquarium class, similar to using a new keyword in other languages.

Define a main() function and call buildAquarium().

A screenshot of a computer

Description automatically generated

Step 4: Add a method

1. In the Aquarium class, add a method to print the aquarium's dimension properties.

A screen shot of a computer

Description automatically generated

1. In Main.kt, in buildAquarium(), call the printSize() method on myAquarium.

A screenshot of a computer

Description automatically generated

1. Run program

A screenshot of a computer

Description automatically generated

1. In buildAquarium(), add code to set the height to 60 and print the changed size attributes.
2. Run the program

A screenshot of a computer

Description automatically generated

[3. Add class constructors](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#2)

Step 1: Create a constructor

In the Aquarium class you created earlier, change the class definition to include three constructor parameters with default values for length, width and height, and assign them to the corresponding properties.

A computer screen shot of a computer program

Description automatically generated

1. The more compact Kotlin way is to define the properties directly with the constructor, using var or val, and Kotlin also creates the getters and setters automatically. Then you can remove the property definitions in the body of the class.
2. When you create an Aquarium object with that constructor, you can specify no arguments and get the default values, or specify just some of them, or specify all of them and create a completely custom-sized Aquarium. In the buildAquarium() function, try out different ways of creating an Aquarium object using named parameters.

A screenshot of a computer program

Description automatically generated

4.Run the program

A screenshot of a computer program

Description automatically generated

Step 2: Add init blocks

1.In the Aquarium class, add an init block to print that the object is initializing, and a second init block to print the volume in liters. Note init blocks can have multiple statements.

A screenshot of a computer program

Description automatically generated

2. Run the program  
A screenshot of a computer program

Description automatically generated

Step 3: Learn about secondary constructors

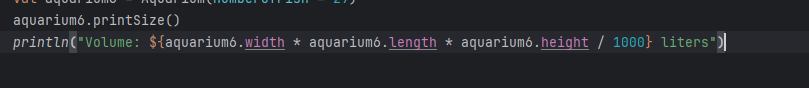
1.In the Aquarium class, add a secondary constructor that takes a number of fish as its argument, using the constructor keyword. Create a val tank property for the calculated volume of the aquarium in liters based on the number of fish. Assume 2 liters (2,000 cm^3) of water per fish, plus a little extra room so the water doesn't spill.

2.Inside the secondary constructor, keep the length and width (which were set in the primary constructor) the same, and calculate the height needed to make the tank the given volume.

A screen shot of a computer code

Description automatically generated

3.In the buildAquarium() function, add a call to create an Aquarium using your new secondary constructor. Print the size and volume.



4.Run the program  
A screenshot of a computer

Description automatically generated

Step 4: Add a new property getter

1.In the Aquarium class, define an Int property called volume, and define a get() method that calculates the volume in the next line.

A computer screen with text and numbers

Description automatically generated

2.Remove the init block that prints the volume.

3.Remove the code in buildAquarium() that prints the volume.

4.In the printSize() method, add a line to print the volume.

A screenshot of a computer code

Description automatically generated

5.Run the program

A screenshot of a computer program

Description automatically generated

Step 5: Add a property setter

1.In the Aquarium class, change volume to a var so it can be set more than once.

2.Add a setter for the volume property by adding a set() method below the getter, which recalculates the height based on the supplied amount of water. By convention, the name of the setter parameter is value, but you can change it if you prefer.

A screenshot of a computer

Description automatically generated

3.In buildAquarium(), add code to set the volume of the Aquarium to 70 liters. Print the new size.

A computer screen with text and numbers

Description automatically generated

4.Run the program

A screenshot of a computer program

Description automatically generated

[4. Learn about visibility modifiers](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#3)

There have been no visibility modifiers, such as public or private, in the code so far. That's because by default, everything in Kotlin is public, which means that everything can be accessed everywhere, including classes, methods, properties, and member variables.

In Kotlin, classes, objects, interfaces, constructors, functions, properties, and their setters can have *visibility modifiers*:

* private means it will only be visible in that class (or source file if you are working with functions).
* protected is the same as private, but it will also be visible to any subclasses.
* internal means it will only be visible within that module. A [module](https://kotlinlang.org/docs/reference/visibility-modifiers.html#modules) is a set of Kotlin files compiled together, for example, a library, a client or application, a server application in an IntelliJ project. Note the usage of "module" here is unrelated to Java modules that were introduced in Java 9.
* public means visible outside the class. Everything is public by default, including variables and methods of the class.

See [Visibility Modifiers](https://kotlinlang.org/docs/reference/visibility-modifiers.html) in the Kotlin documentation for more information.

Member variables

Properties within a class, or member variables, are public by default. If you define them with var, they are mutable, that is, readable and writable. If you define them with val, they are read-only after initialization.

If you want a property that your code can read or write, but outside code can only read, you can leave the property and its getter as public and declare the setter private, as shown below.

A screenshot of a computer program

Description automatically generated

[5. Learn about subclasses and inheritance](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#4)

Step 1: Make the Aquarium class open

1.Mark the Aquarium class and all its properties with the open keyword.

2.Add an open shape property with the value "rectangle".

3.Add an open water property with a getter that returns 90% of the volume of the Aquarium.

4.Add code to the printSize() method to print the shape, and the amount of water as a percentage of the volume.

A computer screen shot of a program

Description automatically generated

5.In buildAquarium(), change the code to create an Aquarium with width = 25, length = 25, and height = 40.

A screenshot of a computer

Description automatically generated

6.Run the program

A screen shot of a computer

Description automatically generated

Step 2: Create a subclass

1.Create a subclass of Aquarium called TowerTank, which implements a rounded cylinder tank instead of a rectangular tank. You can add TowerTank below Aquarium, because you can add another class in the same file as the Aquarium class.

2.In TowerTank, override the height property, which is defined in the constructor. To override a property, use the override keyword in the subclass.

**Note:** Subclasses must declare their constructor parameters explicitly.

3.Make the constructor for TowerTank take a diameter. Use the diameter for both length and width when calling the constructor in the Aquarium superclass.

4.Override the volume property to calculate a cylinder. The formula for a cylinder is pi times the radius squared times the height. Note IntelliJ may flag PI as undefined. You need to import the constant PI from java.lang.Math at the top of Main.kt.

5.In TowerTank, override the water property to be 80% of the volume.

6.Override the shape to be "cylinder".

7.Your final TowerTank class should look something like the code below.

A screen shot of a computer

Description automatically generated

8.In buildAquarium(), create a TowerTank with a diameter of 25 cm and a height of 45 cm.

A computer screen shot of a program

Description automatically generated

9.Run your program and observe the output

A screen shot of a computer

Description automatically generated

[6. Compare abstract classes and interfaces](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#5)

Sometimes you want to define common behavior or properties to be shared among some related classes. Kotlin offers two ways to do that, interfaces and abstract classes. In this task, you create an abstract AquariumFish class for properties that are common to all fish. You create an interface called FishAction to define behavior common to all fish.

* Neither an abstract class or an interface can be instantiated, Abstract classes can have constructors.
* Since they are not classes, interfaces can't have any constructor logic
* Interfaces cannot store any state.

Step 1. Create an abstract class

1.Under **example.myapp**, create a new file, AquariumFish.kt.

2.Create a class, also called AquariumFish, and mark it as abstract.

3.Add one String property, color, and mark it as abstract.

4.Create two subclasses of AquariumFish, Shark and Plecostomus.

5.Because color is abstract, the subclasses must implement it. Make Shark grey and Plecostomus gold.

A screenshot of a computer

Description automatically generated

6.In **Main.kt**, create a makeFish() function to test your classes. Instantiate a Shark and a Plecostomus, then print the color of each.

7.Delete your earlier test code in main() and add a call to makeFish(). Your code should look something like the code below.

A screenshot of a computer

Description automatically generated

8.Run your program and observe the output.

A screen shot of a computer code

Description automatically generated

Step 2. Create an interface

1. In **AquariumFish.kt**, create an interface called FishAction with a method eat().
2. Add FishAction to each of the subclasses, and implement eat() by having it print what the fish does.

A screenshot of a computer program

Description automatically generated

1. In the makeFish()function in Main.kt, have each fish you created eat something by calling eat().

A screenshot of a computer

Description automatically generated

1. Run your program and observe the output.

A black screen with white text

Description automatically generated

When to use abstract classes versus interfaces

The examples above are simple, but when you have a lot of interrelated classes, abstract classes and interfaces can help you keep your design cleaner, more organized, and easier to maintain.

As noted above, abstract classes can have constructors and interfaces cannot, but otherwise they are very similar. So, when should you use each?

When you use interfaces to design a class, the class's functionality is extended by the methods in the interfaces that it implements. Using the traits defined in interfaces will tend to make code easier to reuse and understand than inheritance from an abstract class. Also, you can implement multiple interfaces in a class, but you can only subclass from one class. The rule of thumb is to favor composition (ie, interfaces and instance references) over subclassing where possible.

* Use an abstract class any time you can't complete a class. For example, going back to the AquariumFish class, you can make all AquariumFish implement FishAction, and provide a default implementation for eat while leaving color abstract, because there isn't really a default color for fish.

[7. Use interface delegation](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#6)

The previous task introduced abstract classes and interfaces. *Interface delegation* is an advanced design technique where the methods of an interface are implemented by a helper (or delegate) object, which is then used by a class. This technique can be useful when you use an interface in a series of unrelated classes. You implement the needed interface functionality in a separate helper class. Each of the unrelated classes then uses an instance of that helper class to obtain the functionality.

In this task, you use interface delegation to add functionality to a class.

Step 1: Make a new interface

1. In **AquariumFish.kt**, remove the AquariumFish class. Instead of inheriting from the AquariumFish class, Plecostomus and Shark are going to implement interfaces for both the fish action and their color.
2. Create a new interface, FishColor, that defines the color as a string.
3. Change Plecostomus to implement two interfaces, FishAction, and FishColor. You need to override the color from FishColor and eat() from FishAction.
4. Change your Shark class to also implement the two interfaces, FishAction and FishColor, instead of inheriting from AquariumFish.
5. Your finished code should look something like this:

A screenshot of a computer

Description automatically generated

Step 2: Make a singleton class

Next, you implement the setup for the delegation part by creating a helper class that implements FishColor. You create a basic class called GoldColor that implements FishColor—all it does is say that its color is gold.

It doesn't make sense to make multiple instances of GoldColor, because they'd all do exactly the same thing. So Kotlin lets you declare a class where you can only create one instance of it by using the keyword object instead of class. Kotlin will create that one instance, and that instance is referenced by the class name. Then all other objects can just use this one instance. You cannot create other instances of this class. If you're familiar with the [singleton pattern](https://en.wikipedia.org/wiki/Singleton_pattern), this is how you implement singletons in Kotlin.

1. In **AquariumFish.kt**, create an object for GoldColor. Override the color.

A screenshot of a computer

Description automatically generated

Step 3: Add interface delegation for FishColor

Now you're ready to use interface delegation.

1. In **AquariumFish.kt**, remove the override of color from Plecostomus.
2. Change the Plecostomus class to get its color from GoldColor. You do this by adding by GoldColor to the class declaration, creating the delegation. What this says is that instead of implementing FishColor, use the implementation provided by GoldColor. So every time color is accessed, it is delegated to GoldColor.

A computer screen shot of a computer code

Description automatically generated

With the class as is, all instances of Plecostomus will be "gold". But these fish actually come in many colors. You can address this by adding a constructor parameter for the color with GoldColor as the default color for Plecostomus.

1. Change the Plecostomus class to take a passed in fishColor with its constructor, and set its default to GoldColor. Change the delegation from by GoldColor to by fishColor.

A screen shot of a computer

Description automatically generated

Step 4: Add interface delegation for FishAction

In the same way, you can use interface delegation for the FishAction.

1. In **AquariumFish.kt** make a PrintingFishAction class that implements FishAction, which takes a String food for its constructor parameter, then prints what the fish eats.

A black screen with white text

Description automatically generated

1. In Plecostomus class, remove the override function eat(), because you will replace it with a delegation.
2. In the declaration of Plecostomus, delegate FishAction to PrintingFishAction, passing "eat algae".
3. With all that delegation, there's no code in the body of the Plecostomus class, so remove the {}, because all the overrides are handled by interface delegation.

A screen shot of a computer

Description automatically generated

If you created a similar design for Shark, the following diagram would represent both the Shark and the Plecostomus classes. They are both composed of the PrintingFishAction and FishColor interfaces, but delegating the implementation to them.

Interface delegation is powerful, and you should generally consider how to use it whenever you might use an abstract class in another language. It lets you use composition to plug in behaviors, instead of requiring lots of subclasses, each specialized in a different way.

Composition often leads to better [encapsulation](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)), lower [coupling](https://en.wikipedia.org/wiki/Coupling_(computer_programming)) (interdependence), cleaner interfaces, and more usable code. For these reasons, using composition with interfaces is the preferred design. On the other hand, inheritance from an abstract class tends to be a natural fit for some problems. So you should prefer composition, but when inheritance makes sense Kotlin lets you do that too!

[8. Create a data class](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#7)

A data class is similar to a struct in some other languages. It exists mainly to hold some data. Kotlin [data](https://kotlinlang.org/docs/reference/data-classes.html) classes come with some extra benefits, such as utilities for printing and copying. In this task, you create a simple data class and learn about the support Kotlin provides for data classes.

Step 1: Create a data class

1. Add a new package decor under the **example.myapp** package to hold the new code. Right-click on **example.myapp** in the **Project** pane and select **File > New > Package**.
2. In the package, create a new class called Decoration.
3. To make Decoration a data class, prefix the class declaration with the keyword data.
4. Add a String property called rocks to give the class some data.

A screenshot of a computer

Description automatically generated

1. In the file, outside the class, add a makeDecorations() function to create and print an instance of a Decoration with "granite".
2. Add a main() function to call makeDecorations(), and run your program. Notice the sensible output that is created because this is a data class.

A screen shot of a computer

Description automatically generated

1. In makeDecorations(), instantiate two more Decoration objects that are both "slate" and print them.
2. In makeDecorations(), add a print statement that prints the result of comparing decoration1 with decoration2, and a second one comparing decoration3 with decoration2. Use the equals() method that is provided by data classes.

A screenshot of a computer

Description automatically generated

1. Run your code.

A screen shot of a computer program

Description automatically generated

**Note:** You could have used == to check whether decoration1 == decoration2 and decoration3 == decoration2. In Kotlin, using == on data class objects is the same as using equals() (structural equality). If you need to check whether two variables refer to the same object (referential equality), use the === operator. Read more about [equality in Kotlin](https://kotlinlang.org/docs/reference/equality.html) in the language documentation.

**Note:** Although they are similar to structs in some languages, remember that data class objects are objects. Assigning a data class object to another variable copies the reference to that object, not the contents. To copy the contents to a new object, use the copy() method.

**Warning:** The copy(), equals(), and other data class utilities only refer to properties defined in the primary constructor.

Step 2. Use destructuring

To get at the properties of a data object and assign them to variables, you could assign them one at a time, like this.

Instead, you can make variables, one for each property, and assign the data object to the group of variables. Kotlin puts the property value in each variable.

This is called [destructuring](https://kotlinlang.org/docs/reference/multi-declarations.html" \l "destructuring-declarations" \t "_blank) and is a useful shorthand. The number of variables should match the number of properties, and the variables are assigned in the order in which they are declared in the class. Here is a complete example you can try in **Decoration.kt**.

If you don't need one or more of the properties, you can skip them by using \_ instead of a variable name, as shown in the code below.

    val (rock, \_, diver) = d5

[9. Learn about singletons and enums](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#8)

In this task, you learn about some of the special-purpose classes in Kotlin, including the following:

* Singleton classes
* Companion objects
* Enums

Step 1: Recall singleton classes

Recall the earlier example with the GoldColor class.

A black background with white text

Description automatically generated

Because every instance of GoldColor does the same thing, it is declared as an object instead of as a class to make it a singleton. There can be only one instance of it.

Step 2: Create an enum

Kotlin also supports enums which are a set of named values or constants. Enums are a special type of class in Kotlin which allows you to refer to the value by name, much like in other languages. They can enhance the readability of your code. Each constant in the enum is an object. Declare an enum by prefixing the declaration with the keyword enum. A basic enum declaration only needs a list of names, but you can also define one or more fields associated with each name.

1. In **Decoration.kt**, try out an example of an enum.

A black background with blue text

Description automatically generated

Enums are similar to singletons—there can be only one, and only one of each value in the enumeration. For example, there can only be one Color.RED, one Color.GREEN, and one Color.BLUE. In this example, the RGB values are assigned to the rgb property to represent the color components. There are other useful characteristics of enums. For example, you can get the ordinal value of an enum using the ordinal property, and its name using the name property.

1. Try out another example of an enum in the REPL.

A screen shot of a computer

Description automatically generated

[10. Summary](https://developer.android.com/codelabs/android-development-kotlin-3.1?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-development-with-kotlin-3%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fandroid-development-kotlin-3.1#9)

This lesson covered a lot of ground. While much of it should be familiar from other object-oriented programming languages, Kotlin adds some features to keep code concise and readable.

Classes and constructors

* Define a class in Kotlin using class.
* Kotlin automatically creates setters and getters for properties.
* Define the primary constructor directly in the class definition. For example: class Aquarium(var length: Int = 100, var width: Int = 20, var height: Int = 40)
* If a primary constructor needs additional code, write it in one or more init blocks.
* A class can define one or more secondary constructors using constructor, but Kotlin style is to use a factory function instead.

Visibility modifiers and subclasses

* All classes and functions in Kotlin are public by default, but you can use modifiers to change the visibility to internal, private, or protected.
* To make a subclass, the parent class must be marked open.
* To override methods and properties in a subclass, the methods and properties must be marked open in the parent class.

Data classes, singletons, and enums

* Make a data class by prefixing the declaration with data.
* *Destructuring* is a shorthand for assigning the properties of a data object to separate variables.
* Make a singleton class by using object instead of class.
* Define an enum using enum class.

Abstract classes, interfaces, and delegation

* Abstract classes and interfaces are two ways to share common behavior between classes.
* An *abstract class* defines properties and behavior, but leaves the implementation to subclasses.
* An *interface* defines behavior, and may provide default implementations for some or all of the behavior.
* When you use interfaces to compose a class, the class's functionality is extended by way of the class instances that it contains.
* Interface delegation uses composition by delegating the implementation to the interface classes.
* Composition is a powerful way to add functionality to a class using interface delegation. In general composition is preferred, but inheritance from an abstract class is a better fit for some problems.