CP3406_CP5307 Codelab 1.3: Using Classes and Objects in Kotlin

Contents

1. Terminology

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

In this task, you create a new package and a class with some properties and a method.

Step 1: Create a package

Packages can help you keep your code organized.

1. In the **Project** pane, under the **Hello Kotlin** project, right-click on the **src > main > kotlin** folder.
2. Select **New > Package** and call it `example.myapp`.

Step 2: Create a class with properties

Classes are defined with the keyword `class`, and class names by convention start with a capital letter.

1. Right-click on the **example.myapp** package.
2. Select **New > Kotlin File / Class**.
3. Under **Kind**, select **Class**, and name the class **`Aquarium`**. IntelliJ IDEA includes the package name in the file and creates an empty `Aquarium` class for you.
4. Inside the `Aquarium` class, define and initialize `var` properties for the width, height, and length (in centimeters). Initialize the properties with default values.

```
package example.

myapp


class Aquarium {
    var width: Int = 20
    var height: Int = 40
    var length: Int = 100
}
```

Under the hood, Kotlin automatically creates getters and setters for the properties you defined in the `Aquarium` class, so you can access the properties directly, for example, `myAquarium.length`.

Note: If you declared these properties with `val` instead of `var`, the properties would be immutable. You could only set them once, and all the instances of `Aquarium` would have the same dimensions.

Also note that IntelliJ IDEA underlines the name of each `var` in your code, but not each `val`. Kotlin coding style prefers immutable data when possible, so IntelliJ IDEA draws your attention to mutable data so you can minimize its use.

Step 3: Create a main() function

Create a new file called `Main.kt` to hold the `main()` function.

1. In the **Project** pane on the left, right-click on the **example.myapp** package.
2. Select **New > Kotlin File / Class**.
3. Under the **Kind** dropdown, keep the selection as **File**, and name the file `Main.kt`. IntelliJ IDEA includes the package name, but doesn't include a class definition for a file.
4. 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.
5. Define a `main()` function and call `buildAquarium()`.

```
package example.

myapp


fun buildAquarium() {
    val myAquarium = Aquarium()
}fun main() {
    buildAquarium()
}
```

Step 4: Add a method

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

```
    fun printSize() {
        println("Width: $width cm " +
                "Length: $length cm " +
                "Height: $height cm ")
    }
```

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

```
fun buildAquarium() {
    val myAquarium = Aquarium()
    myAquarium.printSize()
}
```

3. Run your program by clicking the green triangle next to the `main()` function. Observe the result.

```
⇒ Width: 20 cm Length: 100 cm Height: 40 cm
```

4. In `buildAquarium()`, add code to set the height to 60 and print the changed dimension properties.

```
fun buildAquarium() {
    val myAquarium = Aquarium()
    myAquarium.printSize()
    myAquarium.height = 60
    myAquarium.printSize()
}
```

5. Run your program and observe the output.

```
⇒ Width: 20 cm Length: 100 cm Height: 40 cm
Width: 20 cm Length: 100 cm Height: 60 cm
```

3. Add class constructors

In this task, you create a constructor for the class, and continue working with properties.

Step 1: Create a constructor

In this step, you add a constructor to the `Aquarium` class you created in the first task. In the earlier example, every instance of `Aquarium` is created with the same dimensions. You can change the dimensions once it is created by setting the properties, but it would be simpler to create it the correct size to begin with.

In some programming languages like Java, the constructor is defined by creating a method within the class that has the same name as the class. In Kotlin, you define the constructor directly in the class declaration itself, specifying the parameters inside parentheses as if the class was a method. As with functions in Kotlin, those parameters can include default values.

1. 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.

```
class Aquarium(length: Int = 100, width: Int = 20, height: Int = 40) {
   // Dimensions in cm
   var length: Int = length
   var width: Int = width
   var height: Int = height
...
}
```

2. 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.

```
class Aquarium(var length: Int = 100, var width: Int = 20, var height: Int
= 40) {
...
}
```

3. 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.

```
fun buildAquarium() {
    val aquarium1 = Aquarium()
    aquarium1.printSize()
    // default height and length
    val aquarium2 = Aquarium(width = 25)
    aquarium2.printSize()
    // default width
    val aquarium3 = Aquarium(height = 35, length = 110)
    aquarium3.printSize()
    // everything custom
    val aquarium4 = Aquarium(width = 25, height = 35, length = 110)
    aquarium4.printSize()
}
```

4. Run the program and observe the output.

```
⇒ Width: 20 cm Length: 100 cm Height: 40 cm
 Width: 25 cm Length: 100 cm Height: 40 cm
Width: 20 cm Length: 110 cm Height: 35 cm
Width: 25 cm Length: 110 cm Height: 35 cm
```

Notice that you didn't have to overload the constructor and write a different version for each of these cases (plus a few more for the other combinations). Kotlin creates what is needed from the default values and named parameters.

Step 2: Add init blocks

The example constructors above just declare properties and assign the value of an expression to them. If your constructor needs more initialization code, it can be placed in one or more `init` blocks. In this step, you add some `init` blocks to `Aquarium` class.

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.

```
class Aquarium (var length: Int = 100, var width: Int = 20, var height: Int
= 40) {
    init {
        println("aquarium initializing")
    }
    init {
        // 1 liter = 1000 cm^3
        println("Volume: ${width * length * height / 1000} liters")
    }
    ...
}
```

2. Run the program and observe the output.

```
aquarium initializing
Volume: 80 liters
Width: 20 cm Length: 100 cm Height: 40 cm
aquarium initializing
Volume: 100 liters
```

```
Width: 25 cm Length: 100 cm Height: 40 cm
aquarium initializing
Volume: 77 liters
Width: 20 cm Length: 110 cm Height: 35 cm
aquarium initializing
Volume: 96 liters
Width: 25 cm Length: 110 cm Height: 35 cm
```

Notice that the `init` blocks are executed in the order in which they appear in the class definition, and all of them are executed when the constructor is called.

Note: Parameters of the primary constructor can be used in the initializer blocks. Any properties used in initializer blocks must be declared previously.

Step 3: Learn about secondary constructors

In this step, you learn about secondary constructors and add one to your class. In addition to a primary constructor, which can have one or more `init` blocks, a Kotlin class can also have one or more secondary constructors. This feature allows constructor overloading, that is, constructors with different arguments.

Note: Kotlin coding style suggests each class should have only one constructor, using default values and named parameters. This is because using multiple constructors, especially overloaded ones, leads to more code paths and the likelihood that one or more paths will go untested. If you do need multiple constructors, consider whether a factory function would work to keep the class definition clean.

Note: Every secondary constructor must call the primary constructor first, either directly using `this()`, or indirectly by calling another secondary constructor. This means that any `init` blocks in the primary will be executed for all constructors, and all the code in the primary constructor will be executed first.

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.

```
constructor(numberOfFish: Int) : this() {
    // 2,000 cm^3 per fish + extra room so water doesn't spill
    val tank = numberOfFish * 2000 * 1.1
}
```

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.

```
    // calculate the height needed
    height = (tank / (length * width)).toInt()
```

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

```
fun buildAquarium() {
    val aquarium6 = Aquarium(numberOfFish = 29)
    aquarium6.printSize()
    println("Volume: ${aquarium6.width * aquarium6.length *
aquarium6.height / 1000} liters")
}
```

4. Run your program and observe the output.

```
⇒ aquarium initializing
Volume: 80 liters
Width: 20 cm Length: 100 cm Height: 31 cm
Volume: 62 liters
```

Notice that the volume is printed twice, once by the `init` block in the primary constructor before the secondary constructor is executed, and once by the code in `buildAquarium()`.

You could have included the `constructor` keyword in the primary constructor, too, but it's not necessary in most cases.

Step 4: Add a new property getter

In this step, you add an explicit property getter. Kotlin automatically defines getters and setters when you define properties, but sometimes the value for a property needs to be adjusted or calculated. For example above, you printed the volume of the `Aquarium`. You can make the volume available as a property by defining a variable and a getter for it. Because `volume` needs to be calculated, the getter needs to return the calculated value, which you can do with a one-line function that immediately follows the property name and type.

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

```
val volume: Int
    get() = width * height * length / 1000  // 1000 cm^3 = 1 liter
```

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.

```
fun  printSize()  {
    println("Width: $width cm " +
            "Length: $length cm " +
            "Height: $height cm "
    )
    // 1 liter = 1000 cm^3
    println("Volume: $volume liters")
}
```

5. Run your program and observe the output.

```
  ⇒aquarium initializing
Width: 20 cm Length: 100 cm Height: 31 cm
Volume: 62 liters
```

The dimensions and volume are the same as before, but the volume is only printed once after the object is fully initialized by both the primary constructor and the secondary constructor.

Step 5: Add a property setter

In this step, you create a new property setter for the volume.

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.

```
var volume: Int
    get() = width * height * length / 1000
    set(value) {
        height = (value * 1000) / (width * length)
    }
```

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

```
fun buildAquarium() {
    val aquarium6 = Aquarium(numberOfFish = 29)
    aquarium6.printSize()
    aquarium6.volume = 70
    aquarium6.printSize()
}
```

4. Run your program again and observe the changed height and volume.

```
* aquarium initialized
Width: 20 cm Length: 100 cm Height: 31 cm
Volume: 62 liters
Width: 20 cm Length: 100 cm Height: 35 cm
Volume: 70 liters
```

4. Learn about visibility modifiers

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 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.

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.

```
var volume: Int
    get() = width * height * length / 1000
    private set(value) {
        height = (value * 1000) / (width * length)
    }
```

5. Learn about subclasses and inheritance

In this task, you learn how subclasses and inheritance work in Kotlin. They are similar to what you've seen in other languages, but there are some differences.

In Kotlin, by default, classes cannot be subclassed. You must mark a class as `open` to allow it to be subclassed. In those subclasses, you must also mark properties and member variables as `open`, in order to override them in the subclass. The `open` keyword is required, to prevent accidentally leaking implementation details as part of the class's definition.

Step 1: Make the Aquarium class open

In this step, you make the `Aquarium` class `open`, so that you can override it in the next step.

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

```
open class Aquarium (open var length: Int = 100, open var width: Int = 20,
open var height: Int = 40) {
    open var volume: Int
        get() = width * height * length / 1000
        set(value) {
            height = (value * 1000) / (width * length)
        }
```

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

```
   open val shape = "rectangle"
```

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

```
    open var water: Double = 0.0
        get() = volume * 0.9
```

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

```
fun printSize() {
    println(shape)
    println("Width: $width cm " +
            "Length: $length cm " +
            "Height: $height cm ")
    // 1 l = 1000 cm^3
    println("Volume: $volume liters Water: $water liters (${water / volume
* 100.0}% full)")
}
```

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

```
fun buildAquarium() {
    val aquarium6 = Aquarium(length = 25, width = 25, height = 40)
    aquarium6.printSize()
}
```

6. Run your program and observe the new output.

```
⇒ aquarium initializing
rectangle
```

```
Width: 25 cm Length: 25 cm Height: 40 cm
Volume: 25 liters Water: 22.5 liters (90.0% full)
```

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.

```
class TowerTank (override var height: Int, var diameter: Int):
Aquarium(height = height, width = diameter, length = diameter) {
```

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`.

```
    override var volume: Int
    // ellipse area = π * r1 * r2
    get() = (width/2 * length/2 * height / 1000 * PI).toInt()
    set(value) {
        height = ((value * 1000 / PI) / (width/2 * length/2)).toInt()
    }
```

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

```
override var water = volume * 0.8
```

6. Override the `shape` to be `"cylinder"`.

```
override val shape = "cylinder"
```

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

`Aquarium.kt`:

```
package example.

myapp
```

```
import java.lang.Math.

PI


... // existing Aquarium class
class TowerTank (override var height: Int, var diameter: Int):
Aquarium(height = height, width = diameter, length = diameter) {
    override var volume: Int
    // ellipse area = π * r1 * r2
    get() = (width/2 * length/2 * height / 1000 * PI).toInt()
    set(value) {
        height = ((value * 1000 / PI) / (width/2 * length/2)).toInt()
    }
    override var water = volume * 0.8
    override val shape = "cylinder"
}
```

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

`Main.kt:`

```
package example.

myapp

fun buildAquarium() {
    val myAquarium = Aquarium(width = 25, length = 25, height = 40)
    myAquarium.printSize()
    val myTower = TowerTank(diameter = 25, height = 40)
    myTower.printSize()
}
```

9. Run your program and observe the output.

```
⇒ aquarium initializing
rectangle
Width: 25 cm Length: 25 cm Height: 40 cm
Volume: 25 liters Water: 22.5 liters (90.0% full)
aquarium initializing
cylinder
Width: 25 cm Length: 25 cm Height: 40 cm
Volume: 18 liters Water: 14.4 l (80.0% full)
```

6. Compare abstract classes and interfaces

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.

Note: Abstract classes are classes that are partially defined. It is the responsibility of the subclasses of the abstract class to define its methods and properties. Abstract classes are always open; you don't need to mark them with `open`. Properties and methods of an abstract class are non-abstract unless you explicitly mark them with the `abstract` keyword. If these properties and methods do not have the `abstract` keyword, then subclasses can use them as given. If properties or methods are abstract, the subclasses must implement them.

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`.

```
package example.

myapp


abstract class AquariumFish {
    abstract val color: String
}
```

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.

```
class Shark: AquariumFish() {
    override val color = "grey"
}
class Plecostomus: AquariumFish() {
    override val color = "gold"
}
```

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.

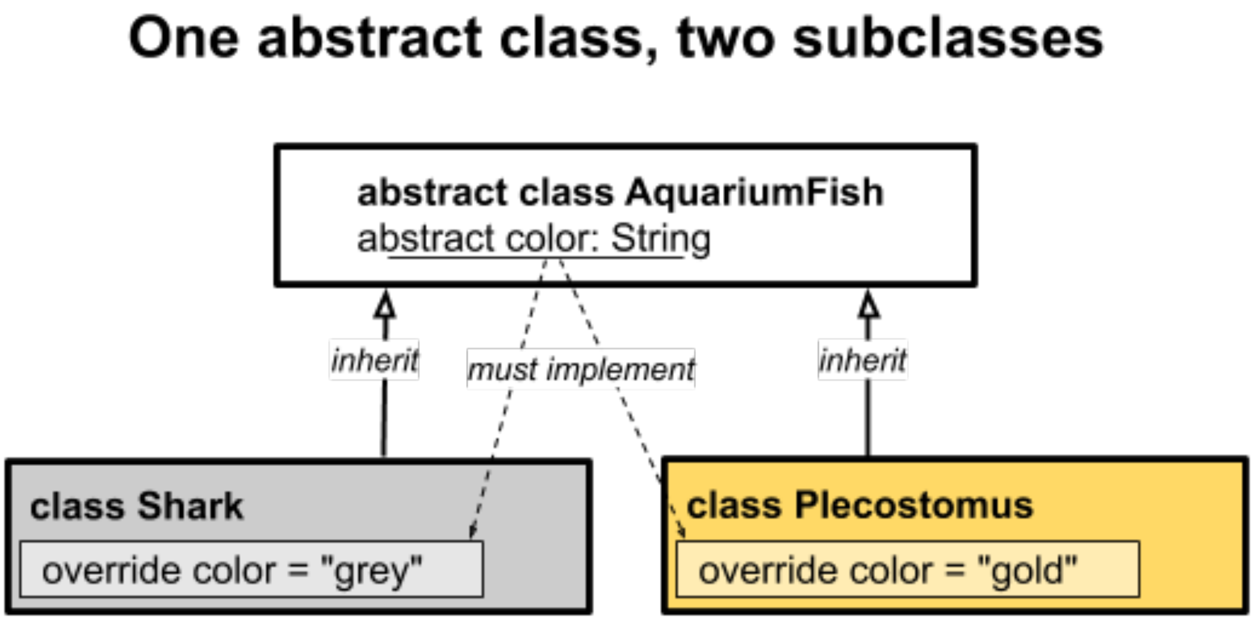
`Main.kt`:

```
package example.

myapp

fun makeFish() {
    val shark = Shark()
    val pleco = Plecostomus()    println("Shark: ${shark.color}")
    println("Plecostomus: ${pleco.color}")
}fun main () {
    makeFish()
}
```

8. Run your program and observe the output.

```
⇒ Shark: grey
Plecostomus: gold
```

The following diagram represents the class hierarchy of our application. Both the `Shark` class and `Plecostomus` class subclass the abstract class, `AquariumFish`.



Step 2. Create an interface

1. In **AquariumFish.kt**, create an interface called `FishAction` with a method `eat()`.

```
interface FishAction {
    fun eat()
}
```

2. Add `FishAction` to each of the subclasses, and implement `eat()` by having it print what the fish does.

```
class Shark: AquariumFish(), FishAction {
    override val color = "grey"
    override fun eat() {
        println("hunt and eat fish")
    }
}
class Plecostomus: AquariumFish(), FishAction {
    override val color = "gold"
    override fun eat() {
        println("eat algae")
    }
}
```

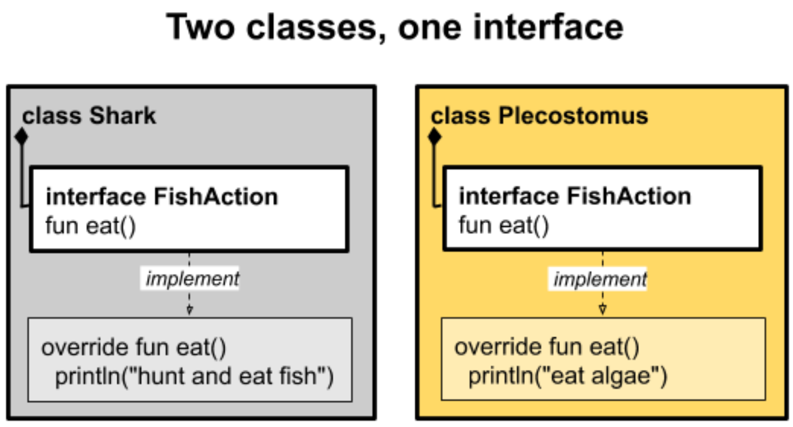
3. In the `makeFish()` function in Main.kt, have each fish you created eat something by calling `eat()`.

```
fun makeFish() {
    val shark = Shark()
    val pleco = Plecostomus()
    println("Shark: ${shark.color}")
    shark.eat()
    println("Plecostomus: ${pleco.color}")
    pleco.eat()
}
```

4. Run your program and observe the output.

```
⇒ Shark: grey
hunt and eat fish
Plecostomus: gold
eat algae
```

The following diagram represents the `Shark` class and the `Plecostomus` class, both of which implement the `FishAction` interface.



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.

```
interface FishAction {
    fun eat()
}
abstract class AquariumFish : FishAction {
   abstract val color: String
   override fun eat() = println("yum")
}
```

7. Use interface delegation

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.

```
interface FishColor {
    val color: String
}
```

3. Change `Plecostomus` to implement two interfaces, `FishAction`, and `FishColor`. You need to override the `color` from `FishColor` and `eat()` from `FishAction`.

```
class Plecostomus: FishAction, FishColor {
    override val color = "gold"
    override fun eat() {
        println("eat algae")
    }
}
```

4. Change your `Shark` class to also implement the two interfaces, `FishAction` and `FishColor`, instead of inheriting from `AquariumFish`.

```
class Shark: FishAction, FishColor {
    override val color = "grey"
    override fun eat() {
        println("hunt and eat fish")
    }
}
```

5. Your finished code should look something like this:

```
package example.

myapp


interface FishAction {
    fun eat()
}
interface FishColor {
    val color: String
}
class Plecostomus: FishAction, FishColor {
    override val color = "gold"
    override fun eat() {
        println("eat algae")
    }
}
class Shark: FishAction, FishColor {
    override val color = "grey"
    override fun eat() {
        println("hunt and eat fish")
    }
}
```

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, this is how you implement singletons in Kotlin.

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

```
object GoldColor : FishColor {
   override val color = "gold"
}
```

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`.

```
class Plecostomus: FishAction, FishColor by GoldColor {
   override fun eat() {
       println("eat algae")
```

```
    }
}
```

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`.

3. 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`.

```
class Plecostomus(fishColor: FishColor = GoldColor): FishAction,
       FishColor by fishColor {
   override fun eat() {
       println("eat algae")
   }
}
```

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.

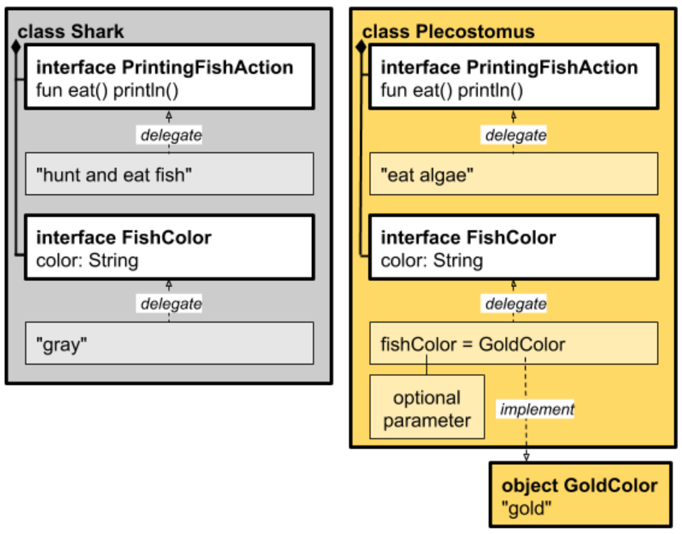
```
class PrintingFishAction(val food: String) : FishAction {
    override fun eat() {
        println(food)
    }
}
```

2. In `Plecostomus` class, remove the override function `eat()`, because you will replace it with a delegation.
3. In the declaration of `Plecostomus`, delegate `FishAction` to `PrintingFishAction`, passing `"eat algae"`.
4. 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.

```
class Plecostomus (fishColor: FishColor = GoldColor):
        FishAction by PrintingFishAction("eat algae"),
        FishColor by fishColor
```

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.

Two classes, two interfaces with delegation



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, lower coupling (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

A `data` class is similar to a `struct` in some other languages. It exists mainly to hold some data. Kotlin `data` 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-clickon **example.myapp** in the **Project** pane and select **File > New > Package**.
2. In the package, create a new class called `Decoration`.

```
package example.myapp.

decor


class 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.

```
data class Decoration(val rocks: String) {
}
```

5. In the file, outside the class, add a `makeDecorations()` function to create and print an instanceof a `Decoration` with `"granite"`.

```
fun makeDecorations() {
    val decoration1 = Decoration("granite")
    println(decoration1)
}
```

6. Add a `main()` function to call `makeDecorations()`, and run your program. Notice the sensibleoutput that is created because this is a data class.

⇒ `Decoration(rocks=granite)`

7. In `makeDecorations()`, instantiate two more `Decoration` objects that are both "slate" and printthem.

```
fun makeDecorations() {
    val decoration1 = Decoration("granite")
    println(decoration1)  val  decoration2  =  Decoration("slate")
    println(decoration2)  val  decoration3  =  Decoration("slate")
    println(decoration3)
}
```

8. 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.

```
    println  (decoration1.equals(decoration2))
    println  (decoration3.equals(decoration2))
```

9. Run your code.

```
⇒ Decoration(rocks=granite)
Decoration(rocks=slate)
Decoration(rocks=slate)
false
true
```

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 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.

```
val rock = decoration.rock
val wood = decoration.wood
val diver = decoration.diver
```

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.

```
val (rock, wood, diver) = decoration
```

This is called destructuring 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**.

```
// Here is a data class with 3 properties.
data class Decoration2(val rocks: String, val wood: String, val diver:
String){
}fun makeDecorations() {
    val d5 = Decoration2("crystal", "wood", "diver")
    println(d5)
// Assign all properties to variables.
    val (rock, wood, diver) = d5
    println(rock)
    println(wood)
    println(diver)
}
```

```
⇒ Decoration2(rocks=crystal, wood=wood, diver=diver)
crystal
wood
diver
```

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

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.

```
object GoldColor : FishColor {
   override val color = "gold"
}
```

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.

```
enum class Color(val rgb: Int) {
   RED(0xFF0000), GREEN(0x00FF00), BLUE(0x0000FF);
}
```

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.

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

```
enum class Direction(val degrees: Int) {
    NORTH(0), SOUTH(180), EAST(90), WEST(270)
}fun main() {
    println(Direction.EAST.name)
    println(Direction.EAST.ordinal)
    println(Direction.EAST.degrees)
}
```

```
⇒ EAST2
90
```

10. Summary

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.