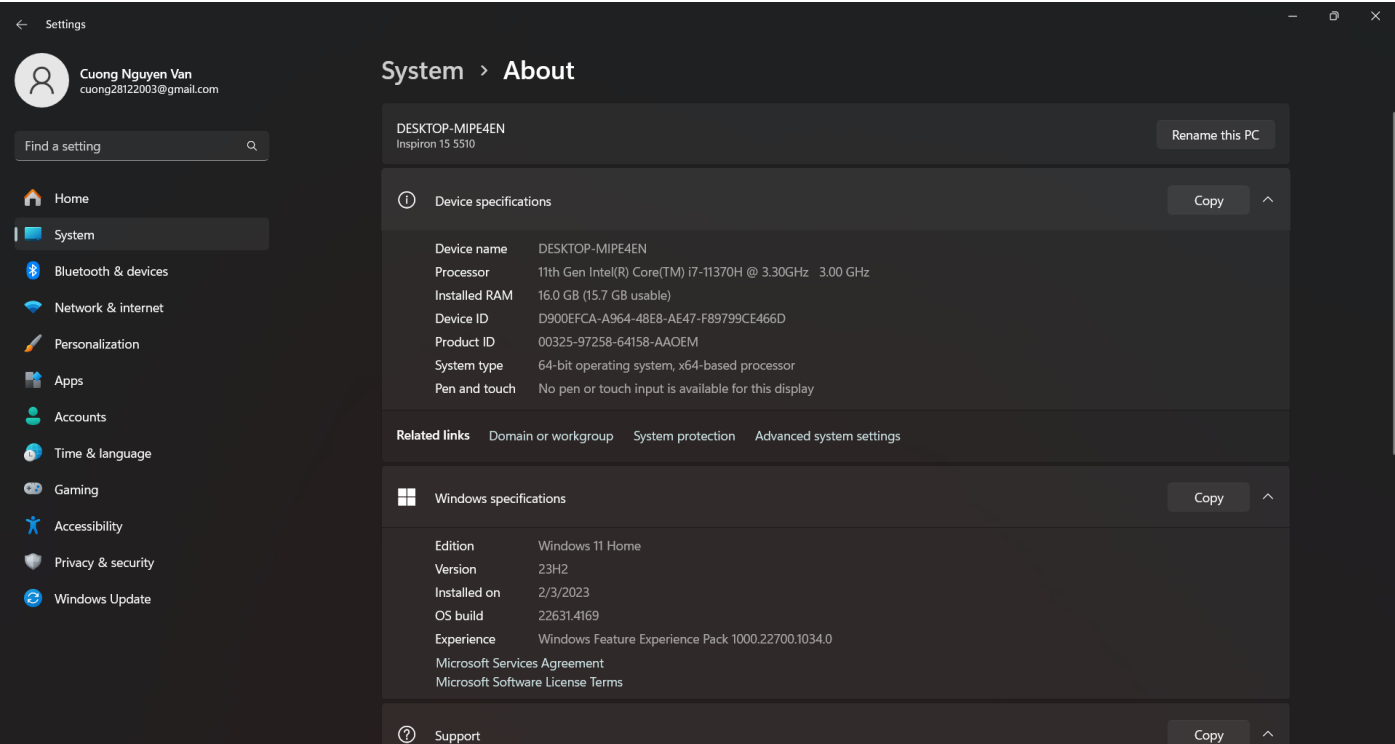
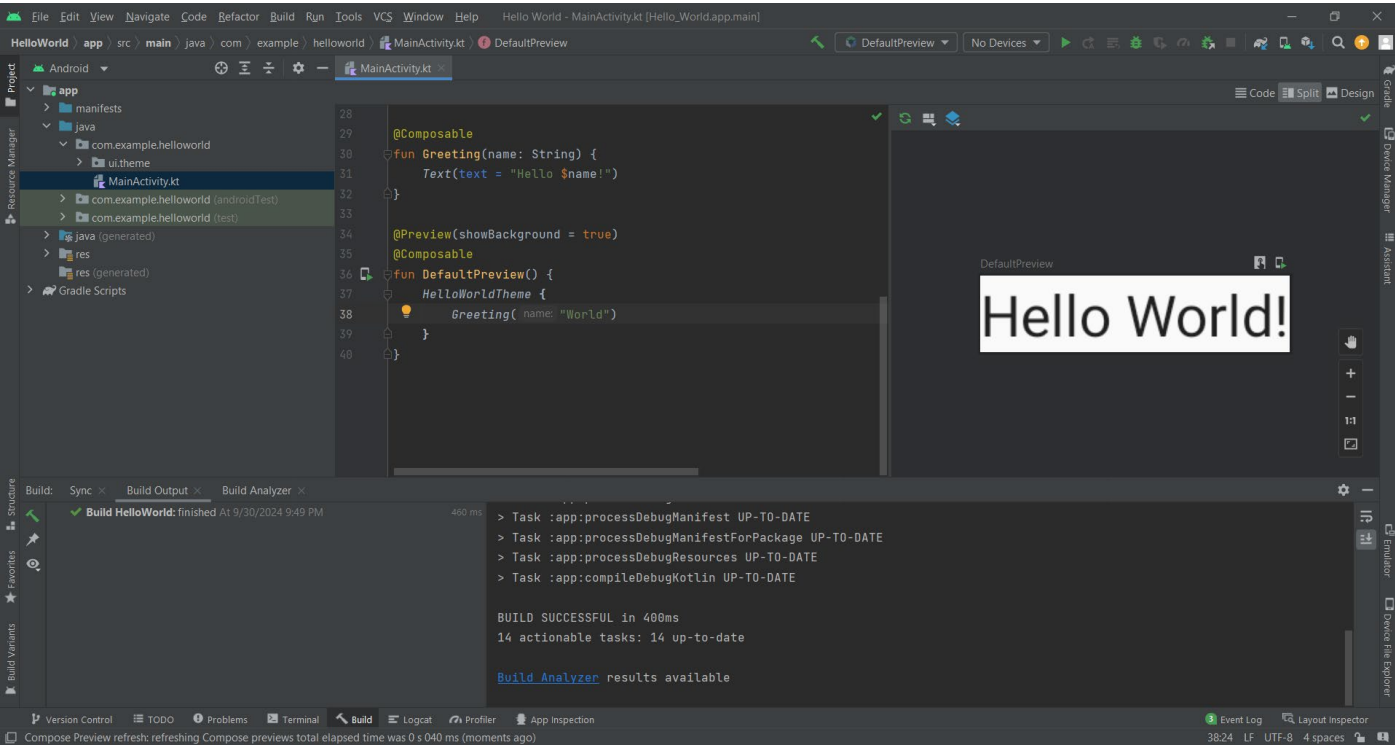


Report – Lesson 3

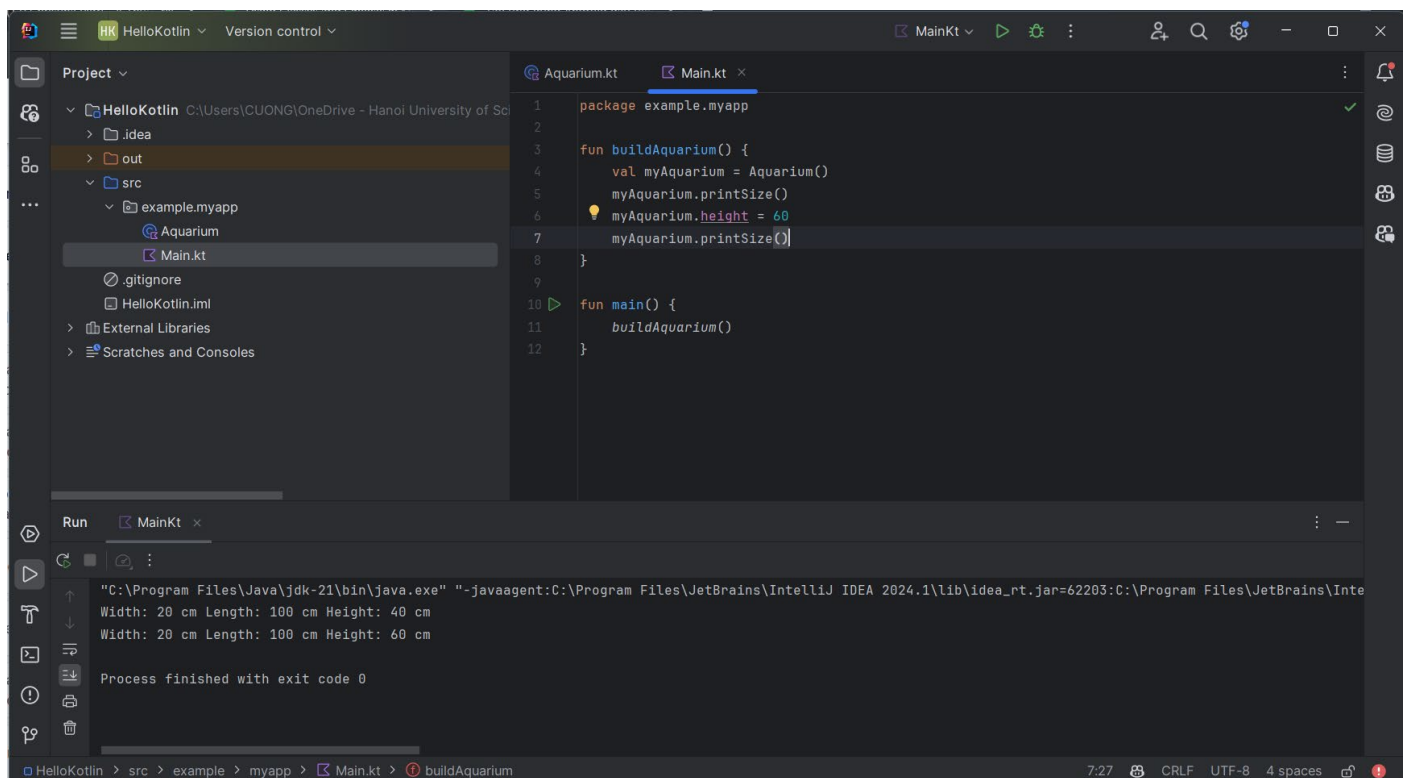
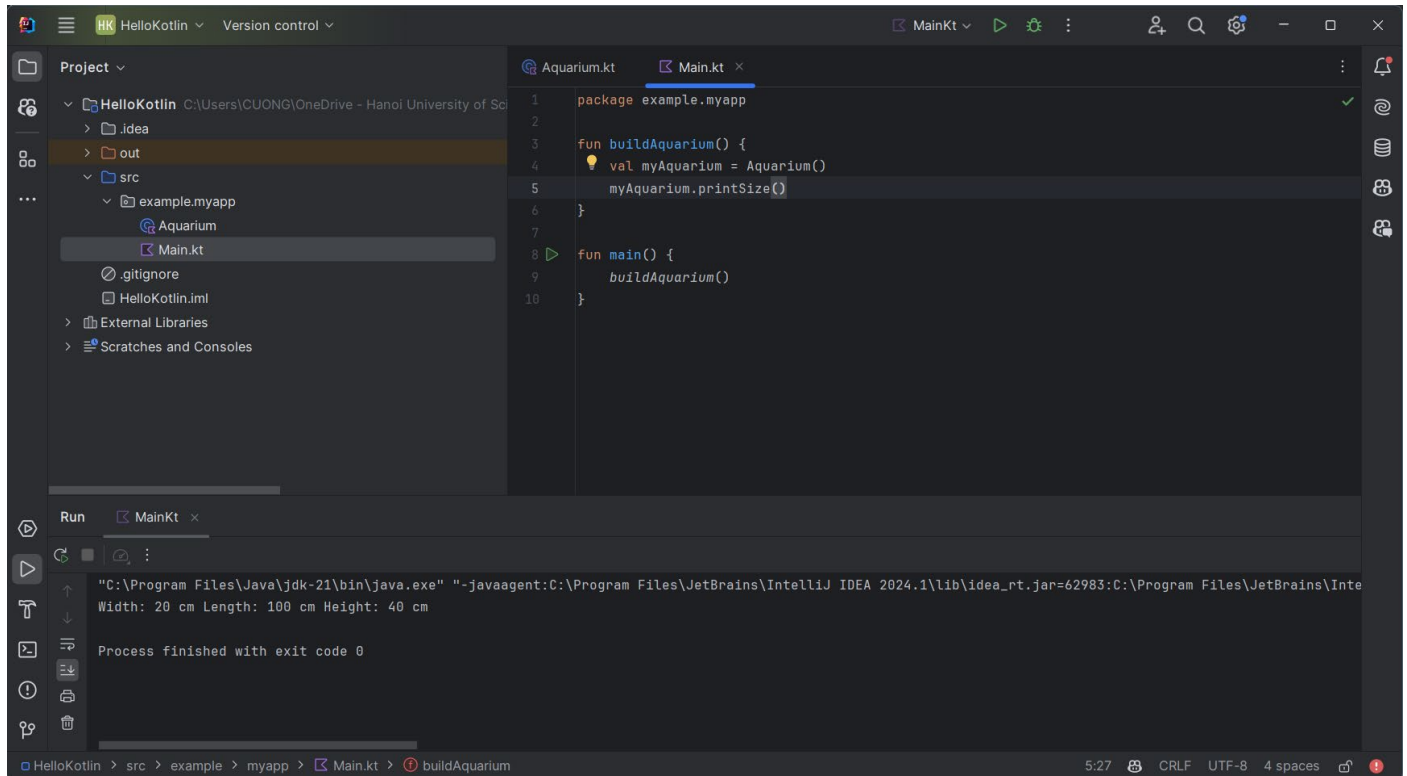
Demo Hello World.....	2
3.1 Using Classes and Objects in Kotlin	3
Create a class.....	3
Add class constructors	4
Learn about visibility modifiers	6
Learn about subclasses and inheritance.....	7
Compare abstract classes and interfaces	8
Use interface delegation	10
Create a data class	11
Learn about singletons and enums	12
3.2 Pairs/triples, collections, constants, and writing extension functions	13
Create a Companion Object.....	13
Learn about pairs and triples	13
Learn more about collections.....	14
Organize and define constants	17
Understand extension functions.....	18

Demo Hello World



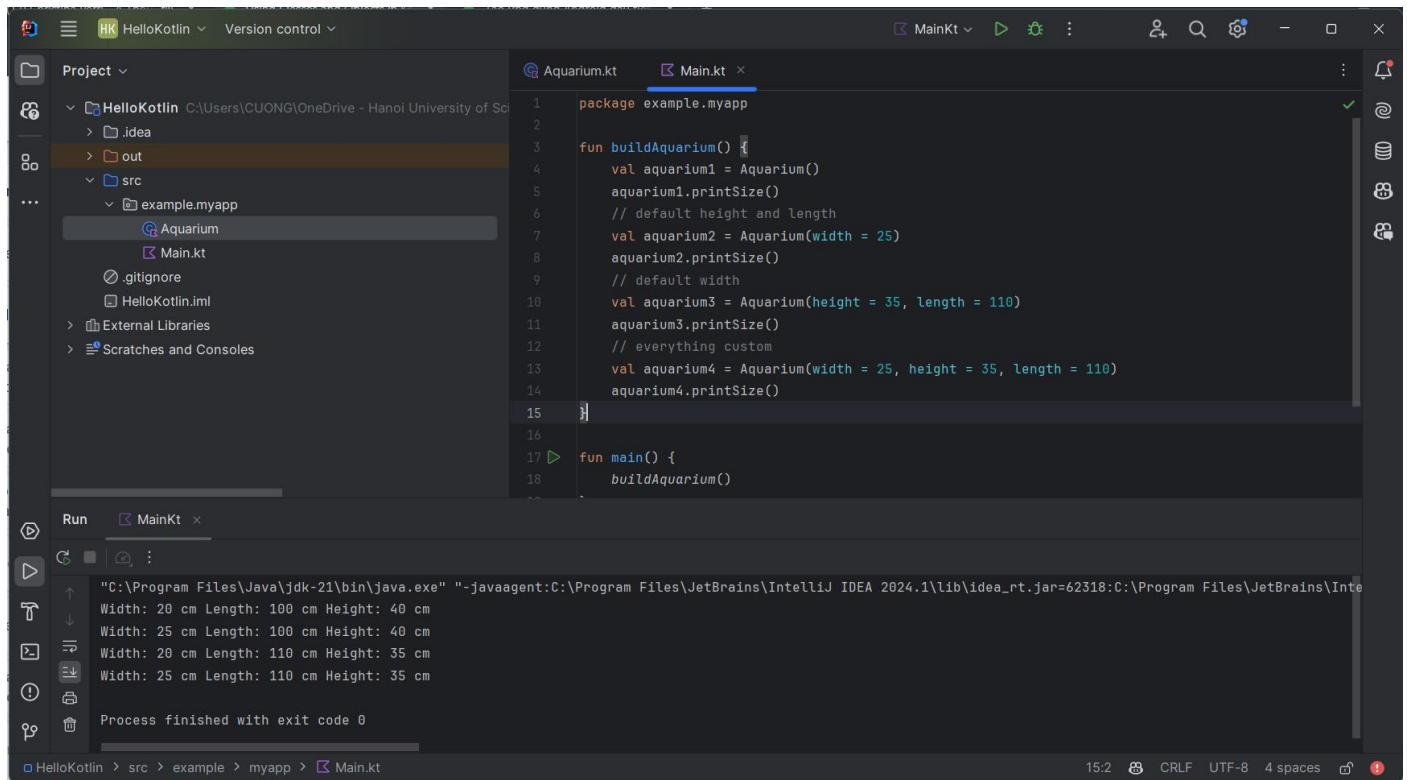
3.1 Using Classes and Objects in Kotlin

Create a class



Add class constructors

Step 1: Create a constructor

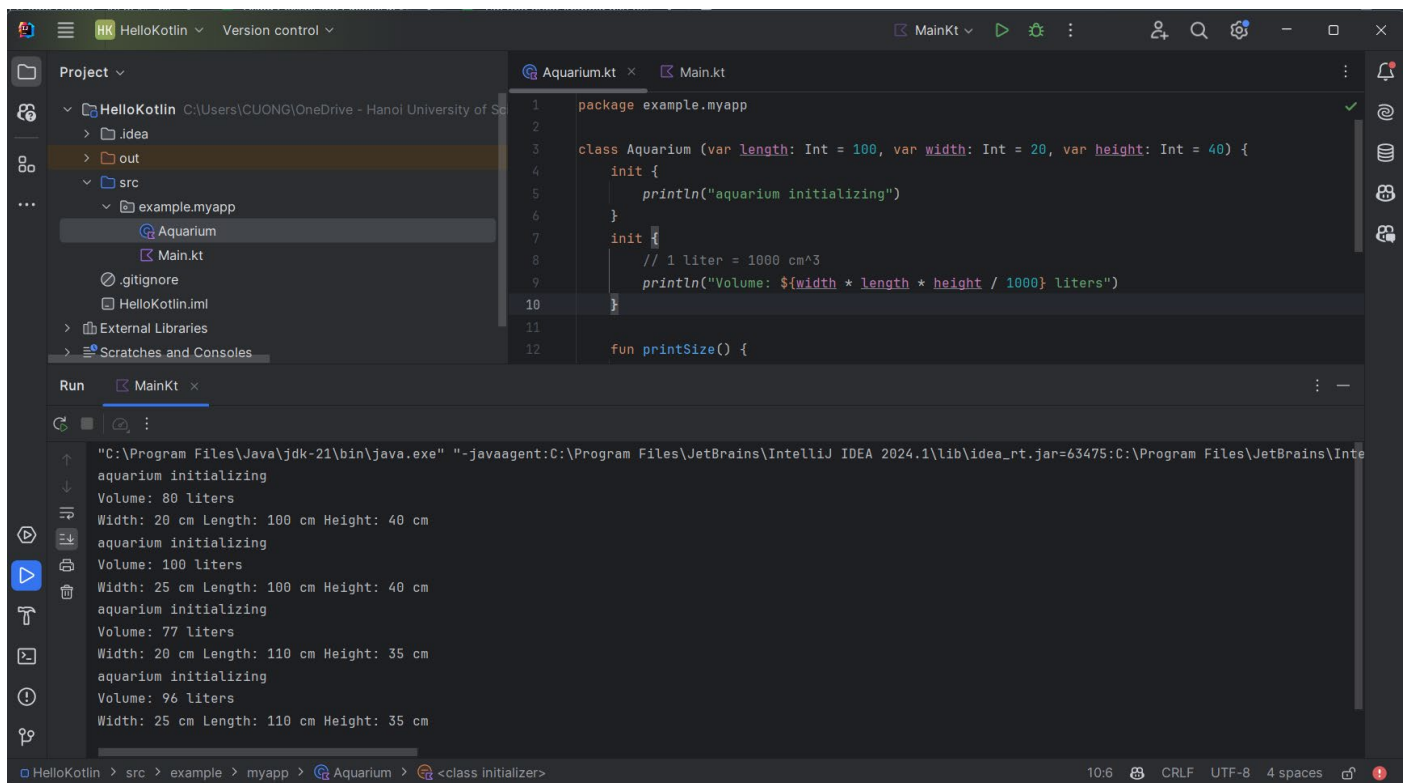


```
1 package example.myapp
2
3 fun buildAquarium() {
4     val aquarium1 = Aquarium()
5     aquarium1.printSize()
6     // default height and length
7     val aquarium2 = Aquarium(width = 25)
8     aquarium2.printSize()
9     // default width
10    val aquarium3 = Aquarium(height = 35, length = 110)
11    aquarium3.printSize()
12    // everything custom
13    val aquarium4 = Aquarium(width = 25, height = 35, length = 110)
14    aquarium4.printSize()
15 }
16
17 fun main() {
18     buildAquarium()
19 }
```

Run MainKt

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\lib\idea_rt.jar=62318:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin" -Didea.config.path=C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\conf -Didea.home.path=C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin -Dfile.encoding=UTF-8
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
Process finished with exit code 0
```

Step 2: Add init blocks



```
1 package example.myapp
2
3 class Aquarium (var length: Int = 100, var width: Int = 20, var height: Int = 40) {
4     init {
5         println("aquarium initializing")
6     }
7     init {
8         // 1 liter = 1000 cm^3
9         println("Volume: ${width * length * height / 1000} liters")
10    }
11
12    fun printSize() {
```

Run MainKt

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\lib\idea_rt.jar=63475:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin" -Didea.config.path=C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\conf -Didea.home.path=C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin -Dfile.encoding=UTF-8
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
```

Step 3: Learn about secondary constructors

The screenshot shows the IntelliJ IDEA interface. The Project view on the left shows the file structure: `src > example > myapp > Aquarium`. The main editor displays `Main.kt` with the following code:

```
3 fun buildAquarium() {  
11     // aquarium3.printSize()  
12     // // everything custom  
13     // val aquarium4 = Aquarium(width = 25, height = 35, length = 110)  
14     // aquarium4.printSize()  
15     val aquarium6 = Aquarium(numberOfFish = 29)  
16     aquarium6.printSize()  
17     println("Volume: ${aquarium6.width * aquarium6.length * aquarium6.height / 1000} liters")  
18 }  
19  
20 fun main() {  
21     buildAquarium()  
22 }
```

The Run tool window at the bottom shows the output of the program:

```
aquarium initializing  
Volume: 80 liters  
Width: 20 cm Length: 100 cm Height: 31 cm  
Volume: 62 liters  
Process finished with exit code 0
```

Step 4: Add a new property getter

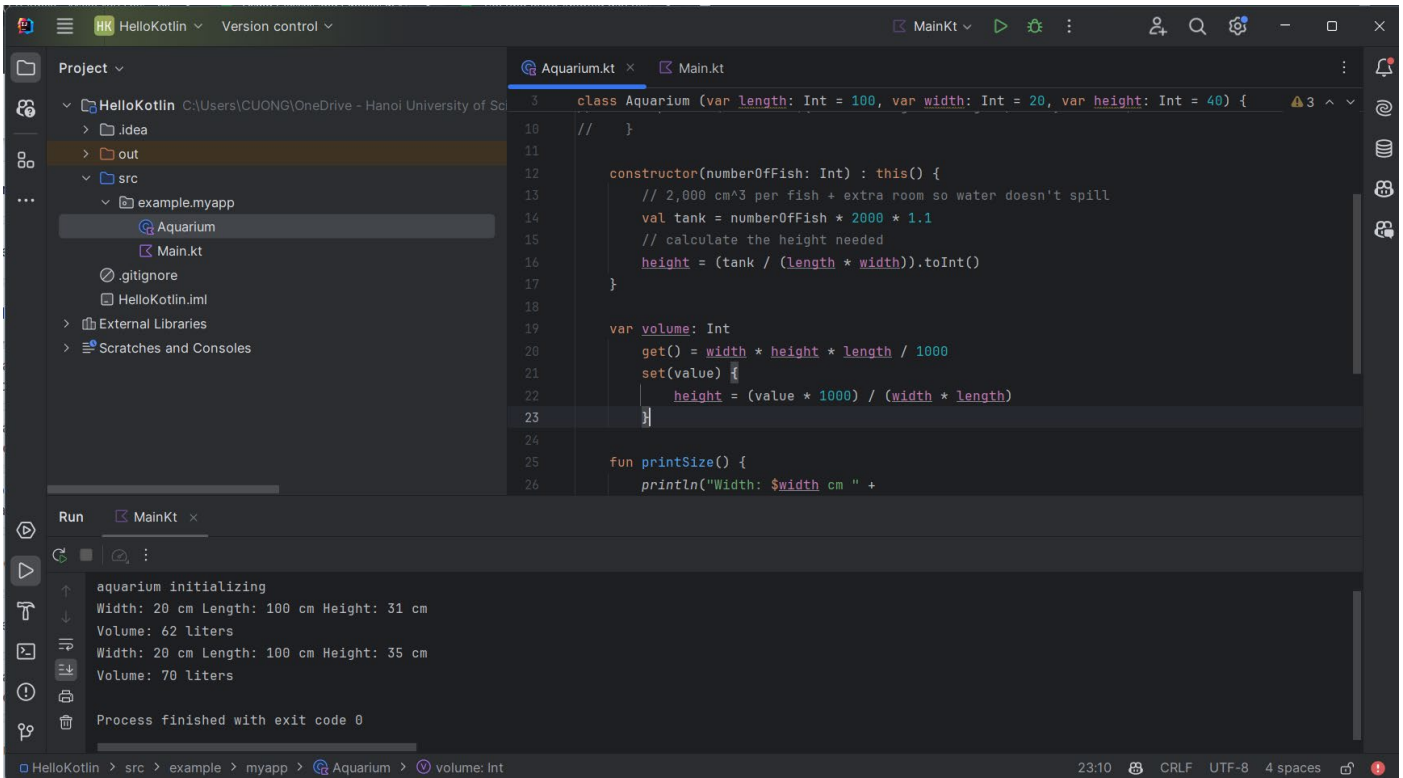
The screenshot shows the IntelliJ IDEA interface. The Project view on the left shows the file structure: `src > example > myapp > Aquarium`. The main editor displays `Aquarium.kt` with the following code:

```
3 class Aquarium (var length: Int = 100, var width: Int = 20, var height: Int = 40) {  
12     constructor(numberOfFish: Int) : this() {  
13         // 2,000 cm^3 per fish + extra room so water doesn't spill  
14         val tank = numberOfFish * 2000 * 1.1  
15         // calculate the height needed  
16         height = (tank / (length * width)).toInt()  
17     }  
18  
19     val volume: Int  
20     get() = width * height * length / 1000 // 1000 cm^3 = 1 liter  
21  
22     fun printSize() {  
23         println("Width: $width cm " +  
24             "Length: $length cm " +  
25             "Height: $height cm "  
26         )  
27         // 1 liter = 1000 cm^3  
28         println("Volume: $volume liters")  
29     }  
30 }
```

The Run tool window at the bottom shows the output of the program:

```
aquarium initializing  
Width: 20 cm Length: 100 cm Height: 31 cm  
Volume: 62 liters  
Volume: 62 liters  
Process finished with exit code 0
```

Step 5: Add a property setter



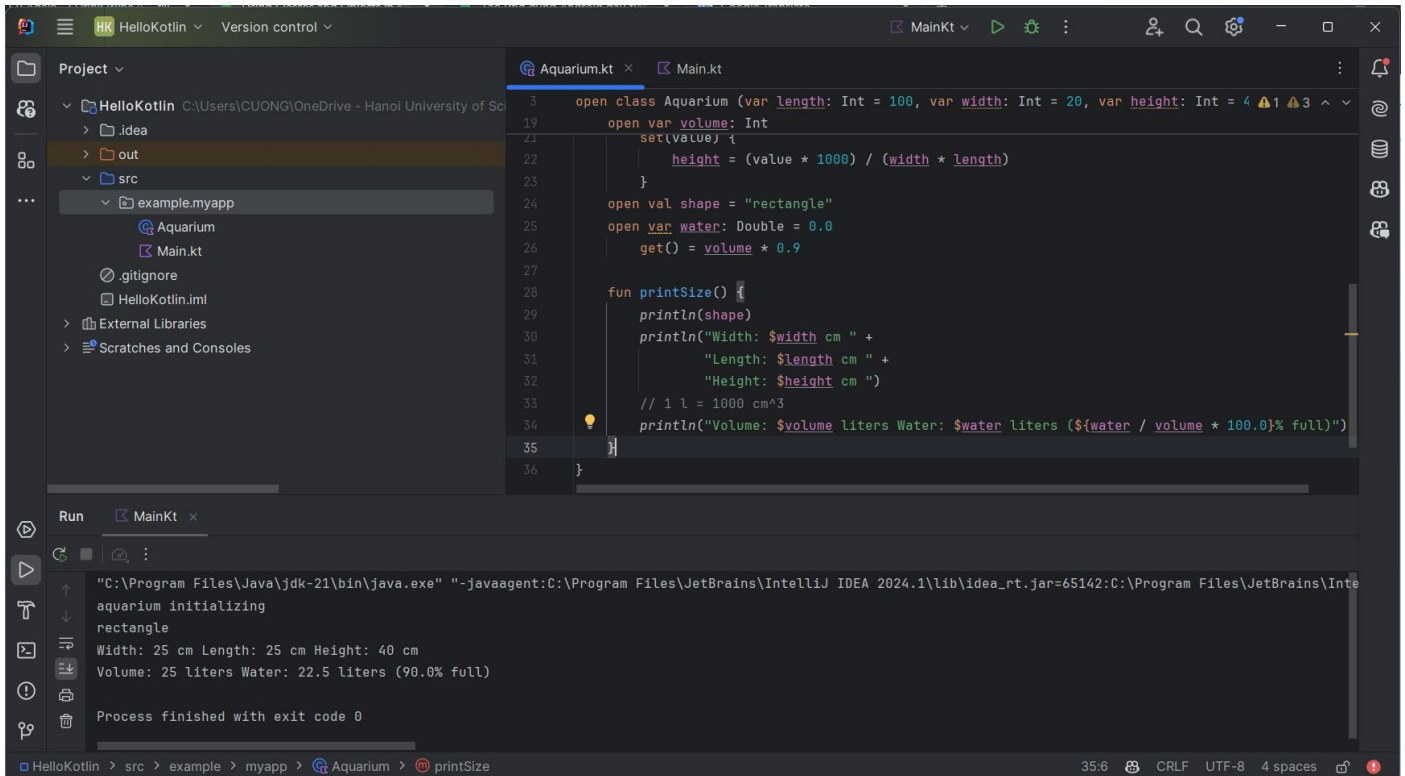
Learn about visibility modifiers

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.

Learn about subclasses and inheritance

Step 1: Make the Aquarium class open



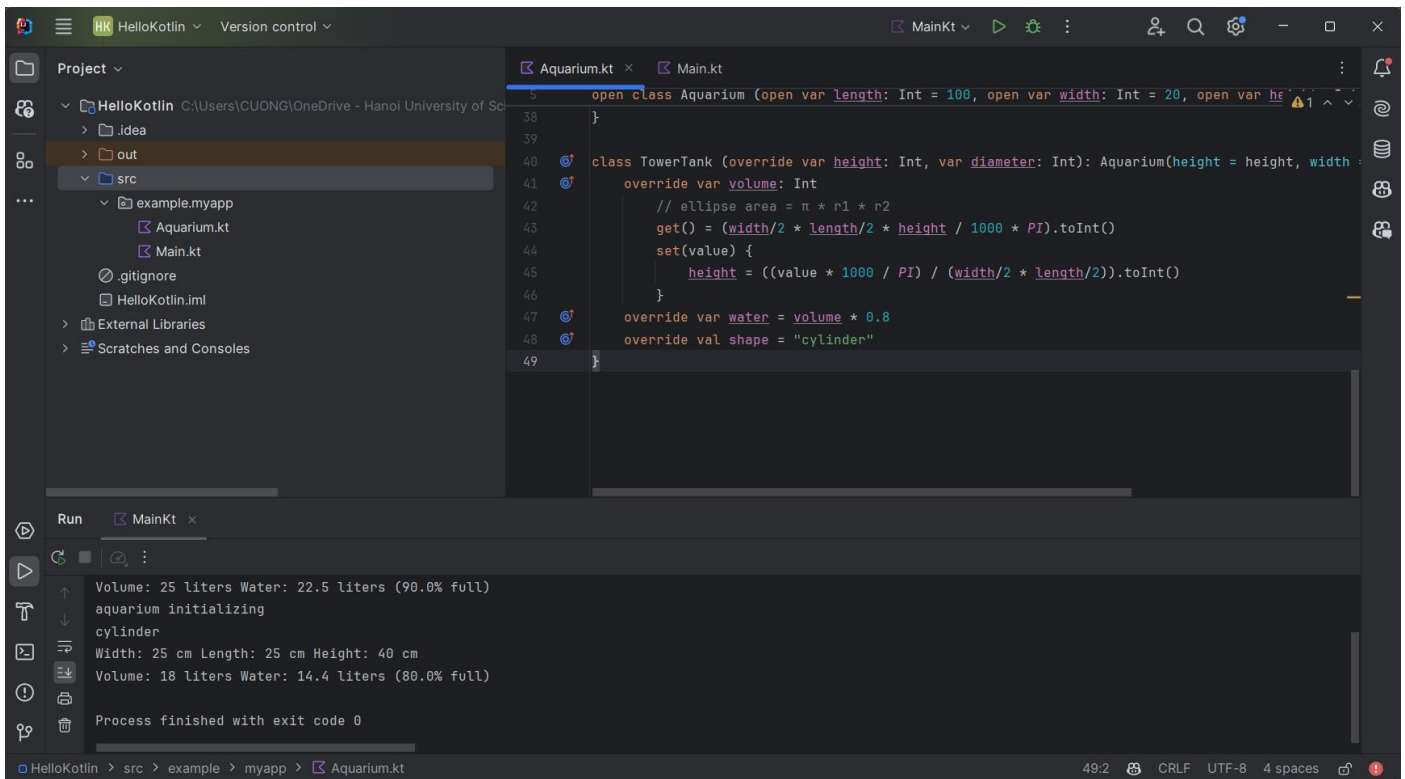
The screenshot shows the IntelliJ IDEA interface with the `Aquarium.kt` file open. The code defines an `open class Aquarium` with properties `length`, `width`, and `height`. It includes a `volume` property with a setter and a `water` property with a getter. A `printSize` function prints the dimensions and volume. The Run console shows the output of the `printSize` function.

```
3 open class Aquarium (var length: Int = 100, var width: Int = 20, var height: Int = 40) {
19     open var volume: Int
21         set(value) {
22             height = (value * 1000) / (width * length)
23         }
24     open val shape = "rectangle"
25     open var water: Double = 0.0
26     get() = volume * 0.9
27
28     fun printSize() {
29         println(shape)
30         println("Width: $width cm " +
31             "Length: $length cm " +
32             "Height: $height cm ")
33         // 1 l = 1000 cm^3
34         println("Volume: $volume liters Water: $water liters (${water / volume * 100.0}% full)")
35     }
36 }
```

Run MainKt x

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\lib\idea_rt.jar=65142:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin" -Dfile.encoding=UTF-8
aquarium initializing
rectangle
Width: 25 cm Length: 25 cm Height: 40 cm
Volume: 25 liters Water: 22.5 liters (90.0% full)
Process finished with exit code 0
```

Step 2: Create a subclass



The screenshot shows the IntelliJ IDEA interface with the `Aquarium.kt` file open. The code defines a `class TowerTank` that inherits from `Aquarium`. It overrides the `height` property, the `volume` property with a setter, and the `water` property. The `printSize` function is also overridden. The Run console shows the output of the `printSize` function.

```
38 open class Aquarium (open var length: Int = 100, open var width: Int = 20, open var height: Int = 40) {
39 }
40 class TowerTank (override var height: Int, var diameter: Int): Aquarium(height = height, width = width) {
41     override var volume: Int
42         // ellipse area = π * r1 * r2
43         get() = (width/2 * length/2 * height / 1000 * PI).toInt()
44         set(value) {
45             height = ((value * 1000 / PI) / (width/2 * length/2)).toInt()
46         }
47     override var water = volume * 0.8
48     override val shape = "cylinder"
49 }
```

Run MainKt x

```
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 liters (80.0% full)
Process finished with exit code 0
```

Compare abstract classes and interfaces

Step 1. Create an abstract class

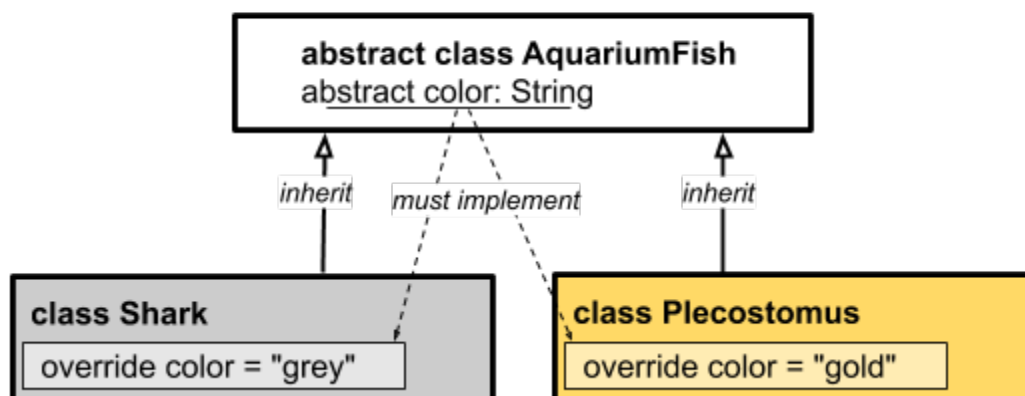
The screenshot shows the IntelliJ IDEA IDE with a project named 'HelloKotlin'. The project structure on the left includes a package 'example.myapp' containing files 'Aquarium.kt', 'AquariumFish.kt', and 'Main.kt'. The 'AquariumFish.kt' file is open in the editor, showing the following code:

```
1 package example.myapp
2
3 abstract class AquariumFish {
4     abstract val color: String
5 }
6
7 class Shark: AquariumFish() {
8     override val color = "grey"
9 }
10
11 class Plecostomus: AquariumFish() {
12     override val color = "gold"
13 }
```

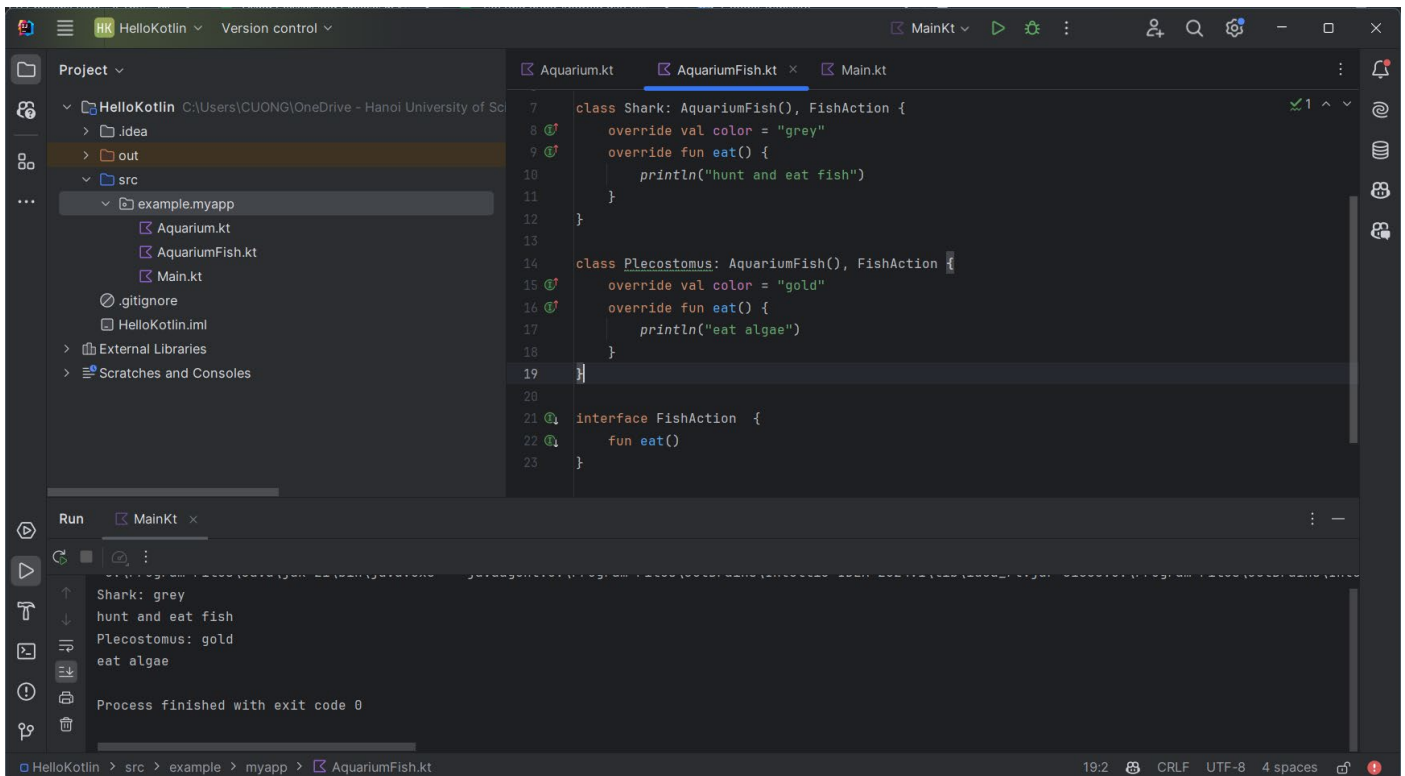
The 'Run' tab at the bottom shows the output of the program:

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\lib\idea_rt.jar=51797:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin" -jar C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin\idea_rt.jar 51797
Shark: grey
Plecostomus: gold
Process finished with exit code 0
```

One abstract class, two subclasses



Step 2: Create an interface



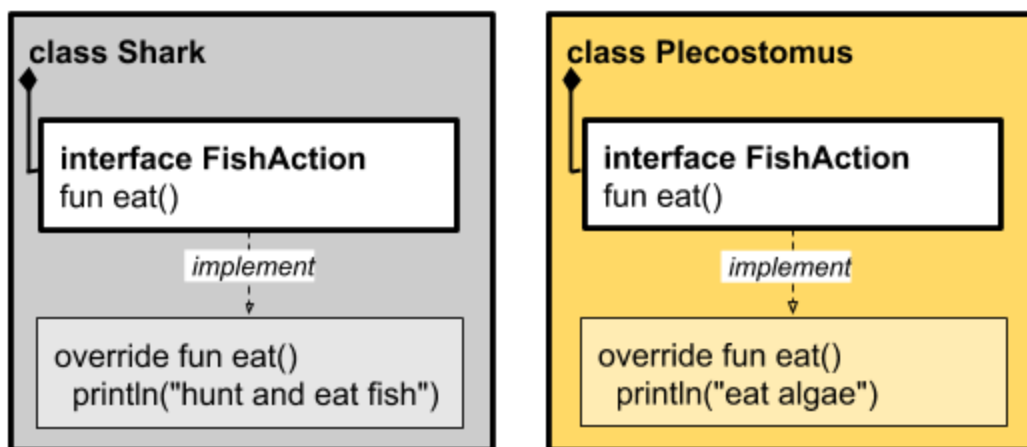
The screenshot shows an IDE with the following code:

```
7 class Shark: AquariumFish(), FishAction {  
8     override val color = "grey"  
9     override fun eat() {  
10         println("hunt and eat fish")  
11     }  
12 }  
13  
14 class Plecostomus: AquariumFish(), FishAction {  
15     override val color = "gold"  
16     override fun eat() {  
17         println("eat algae")  
18     }  
19 }  
20  
21 interface FishAction {  
22     fun eat()  
23 }
```

The Run console shows the output of the program:

```
Shark: grey  
hunt and eat fish  
Plecostomus: gold  
eat algae  
Process finished with exit code 0
```

Two classes, one interface



When to use abstract classes versus interfaces

- 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.

Use interface delegation

Step 1: Make a new interface

```
7 interface FishColor {  
8     val color: String  
9 }  
10  
11 class Plecostomus: FishAction, FishColor {  
12     override val color = "gold"  
13     override fun eat() {  
14         println("eat algae")  
15     }  
16 }  
17  
18 class Shark: FishAction, FishColor {  
19     override val color = "grey"  
20     override fun eat() {  
21         println("hunt and eat fish")  
22     }  
23 }
```

Step 2: Make a singleton class

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

```
25 object GoldColor : FishColor {  
26     override val color = "gold"  
27 }
```

Step 3: Add interface delegation for FishColor

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

Step 4: Add interface delegation for FishAction

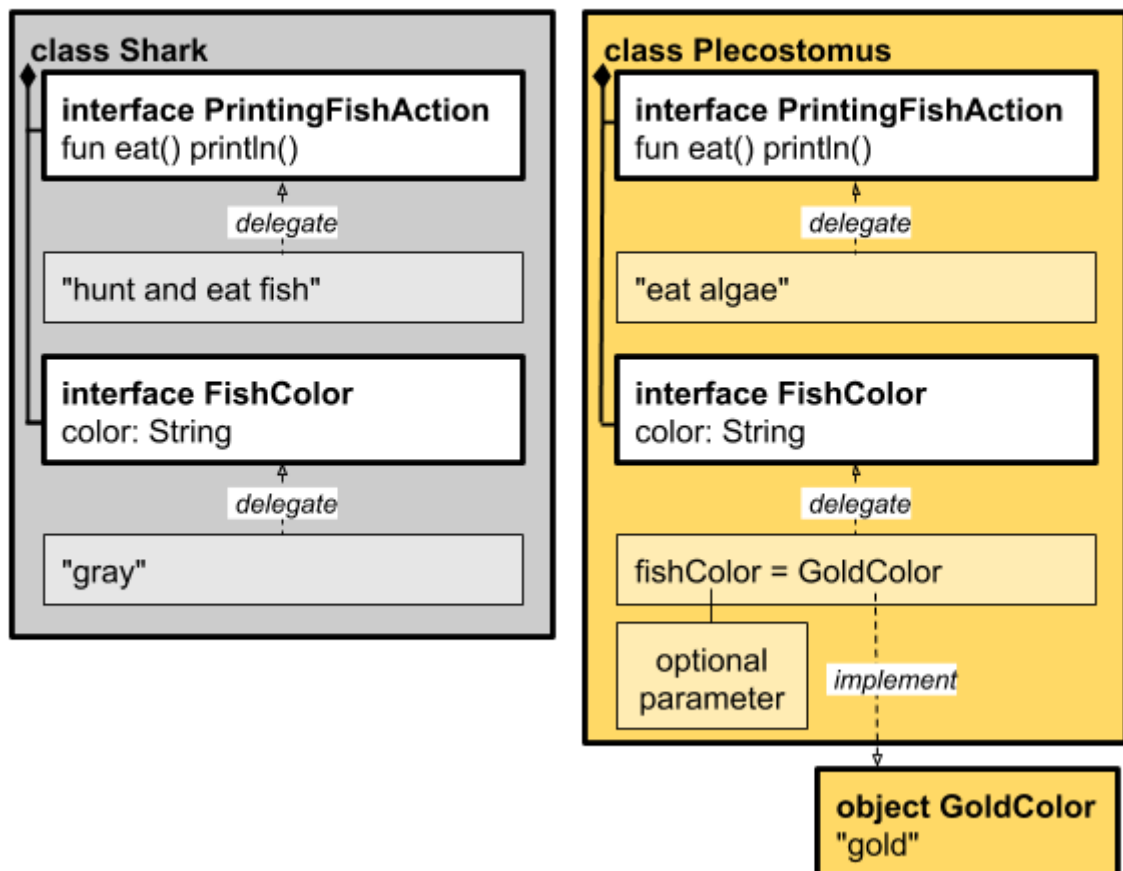
```

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

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

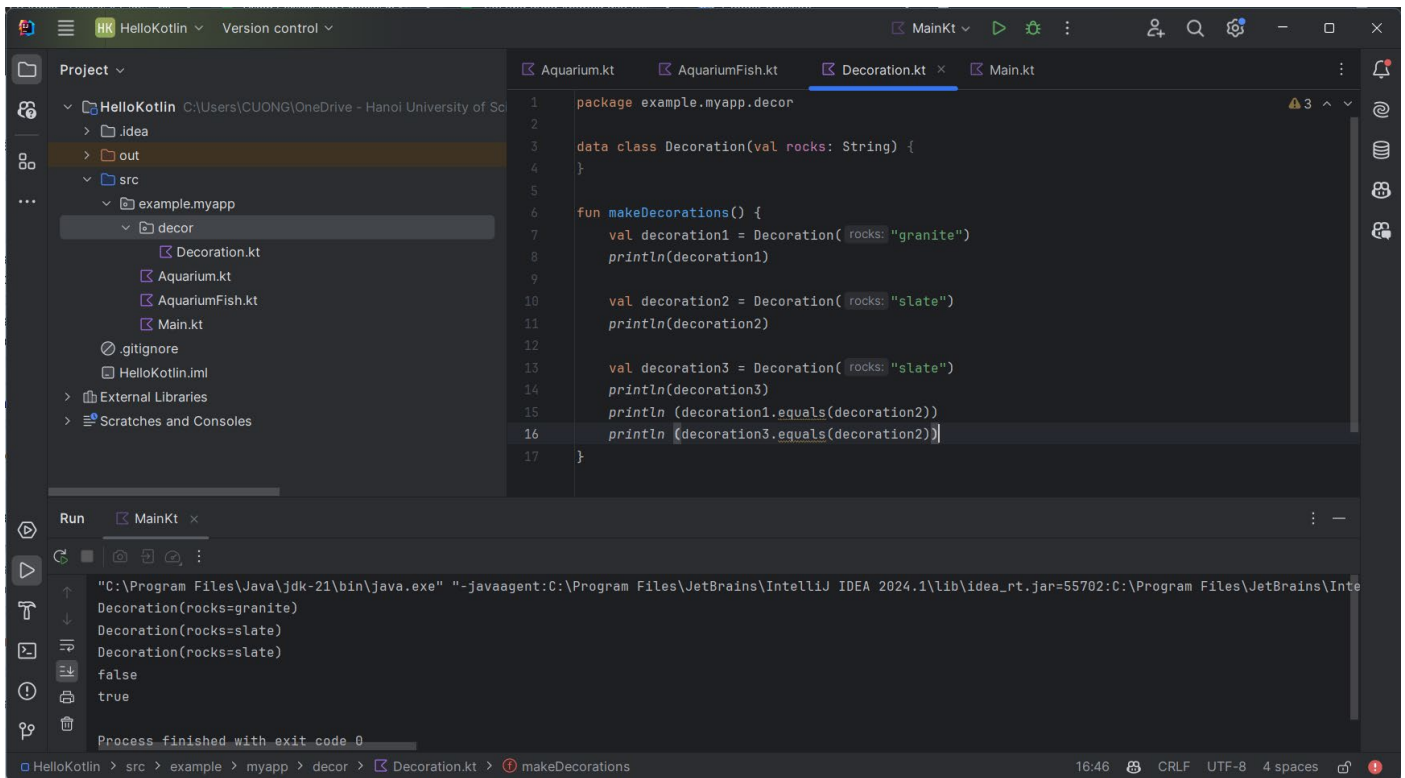
```

Two classes, two interfaces with delegation

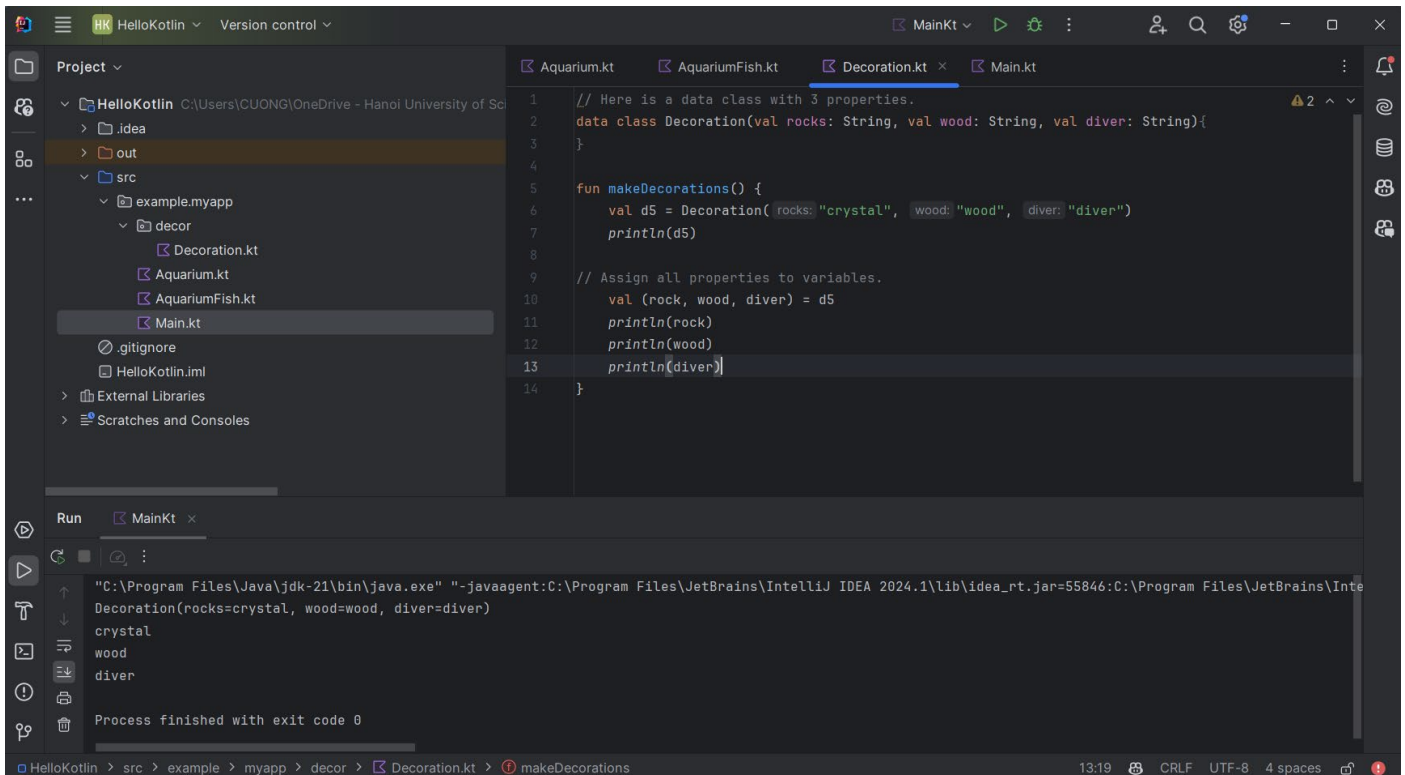


Create a data class

Step 1: Create a data class



Step 2. Use destructuring



Learn about singletons and enums

Step 1: Recall singleton classes

Step 2: Create an enum

1. Create a pair

```
val equipment = "fish net" to "catching fish"
println("${equipment.first} used for ${equipment.second}")
fish net used for catching fish
```

2. Create a triple

```
val numbers = Triple(6, 9, 42)
println(numbers.toString())
println(numbers.toList())
(6, 9, 42)[6, 9, 42]
```

3. Create a pair where the first part of the pair is itself a pair.

```
val equipment2 = ("fish net" to "catching fish") to "equipment"
println("${equipment2.first} is ${equipment2.second}\n")
println("${equipment2.first.second}")
(fish net, catching fish) is equipment
catching fish
```

Step 2: Destructure some pairs and triples

1. Destructure a pair and print the values.

```
val equipment = "fish net" to "catching fish"
val (tool, use) = equipment
println("$tool is used for $use")
fish net is used for catching fish
```

2. Destructure a triple and print the values.

```
val numbers = Triple(6, 9, 42)
val (n1, n2, n3) = numbers
println("$n1 $n2 $n3")
6 9 42
```

Learn more about collections

Step 1: Understand more about lists

1. Complete listings in the Kotlin documentation for both [List](#) and [MutableList](#)

Function	Purpose
<code>add(element: E)</code>	Add an item to the mutable list.
<code>remove(element: E)</code>	Remove an item from a mutable list.
<code>reversed()</code>	Return a copy of the list with the elements in reverse order.
<code>contains(element: E)</code>	Return <code>true</code> if the list contains the item.
<code>subList(fromIndex: Int, toIndex: Int)</code>	Return part of the list, from the first index up to but not including the second index.

2. Sums up all the elements.

```
val list = listOf(1, 5, 3, 4)
println(list.sum())
13
```

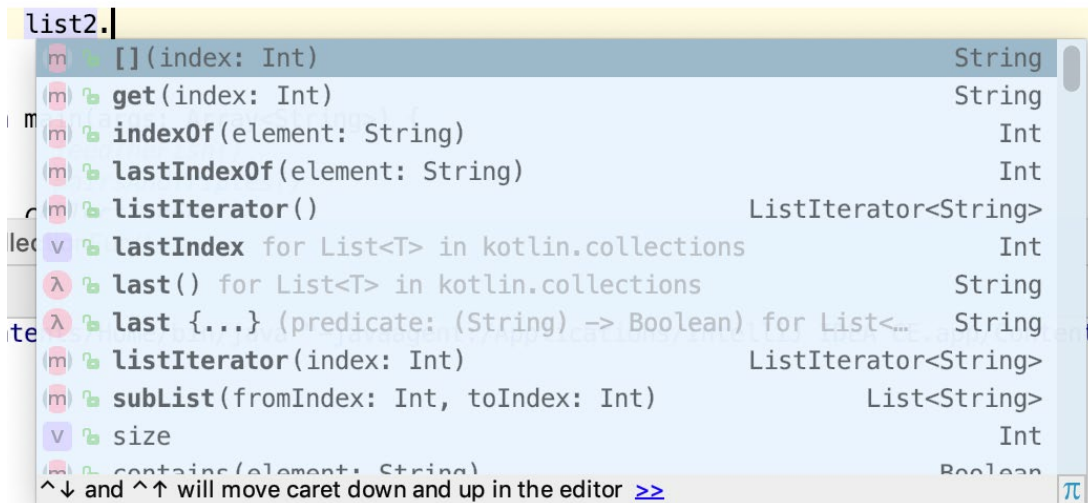
3. Create a list of strings and sum the list.

```
val list2 = listOf("a", "bbb", "cc")
println(list2.sum())
error: unresolved reference. None of the following candidates is applicable because of receiver type mismatch:
public fun Array<out Byte>.sum(): Int defined in kotlin.collections
public fun Array<out Double>.sum(): Double defined in kotlin.collections
public fun Array<out Float>.sum(): Float defined in kotlin.collections
public fun Array<out Int>.sum(): Int defined in kotlin.collections
public fun Array<out Long>.sum(): Long defined in kotlin.collections
public fun Array<out Short>.sum(): Int defined in kotlin.collections
public fun Array<out UByte>.sum(): UInt defined in kotlin.collections
public fun Array<out UInt>.sum(): UInt defined in kotlin.collections
public fun Array<out ULong>.sum(): ULong defined in kotlin.collections
public fun Array<out UShort>.sum(): UInt defined in kotlin.collections
public fun ByteArray.sum(): Int defined in kotlin.collections
public fun DoubleArray.sum(): Double defined in kotlin.collections
public fun FloatArray.sum(): Float defined in kotlin.collections
public fun IntArray.sum(): Int defined in kotlin.collections
public fun LongArray.sum(): Long defined in kotlin.collections
public fun ShortArray.sum(): Int defined in kotlin.collections
public inline fun UByteArray.sum(): UInt defined in kotlin.collections
public inline fun UIntArray.sum(): UInt defined in kotlin.collections
public inline fun ULongArray.sum(): ULong defined in kotlin.collections
public inline fun UShortArray.sum(): UInt defined in kotlin.collections
public fun Iterable<Byte>.sum(): Int defined in kotlin.collections
public fun Iterable<Double>.sum(): Double defined in kotlin.collections
public fun Iterable<Float>.sum(): Float defined in kotlin.collections
```

4. Using `.sumBy()` with a lambda function

```
val list2 = listOf("a", "bbb", "cc")
println(list2.sumBy { it.length })
6
```

5. There's a lot more you can do with lists. One way to see the functionality available is to create a list in IntelliJ IDEA, add the dot, and then look at the auto-completion list in the tooltip. This works for any object. Try it out with a list.



6. Choose listIterator() from the list, then go through the list with a for statement and print all the elements separated by spaces.

```
val list2 = listOf("a", "bbb", "cc")
for (s in list2.listIterator()) {
    println("$s ")
}
a bbb cc
```

Step 2: Try out hash maps

1. Create a hash map

```
val scientific = hashMapOf("guppy" to "poecilia reticulata", "catfish" to "corydoras", "zebra fish" to "danio rerio")
```

2. Retrieve the scientific name value based on the common fish name key, using get(), or even shorter, square brackets [].

```
println (scientific.get("guppy"))
poecilia reticulata

println(scientific.get("zebra fish"))
danio rerio
```

3. Try specifying a fish name that isn't in the map.

```
println("scientific.get("swordtail")")
error: unresolved reference: swordtail
```

4. Try looking up a key that has no match, using getOrDefault().

```
println(scientific.getOrDefault("swordtail", "sorry, I don't know"))
sorry, I don't know
```

5. Change your code to use getOrElse() instead of getOrDefault().

```
println(scientific.getOrElse("swordtail") {"sorry, I don't know"})  
sorry, I don't know
```

Organize and define constants

Step 1: Learn about const vs. val

The value for `const val` is determined at compile time, whereas the value for `val` is determined during program execution, which means, `val` can be assigned by a function at run time.

That means `val` can be assigned a value from a function, but `const val` cannot.

```
const val rocks = 3  
error: const 'val' are only allowed on top level, in nar  
const val rocks = 3  
^  
  
val value1 = complexFunctionCall() // OK  
const val CONSTANT1 = complexFunctionCall() // NOT ok  
error: unresolved reference: complexFunctionCall  
val value1 = complexFunctionCall() // OK  
^  
  
error: const 'val' are only allowed on top level, in nar  
const val CONSTANT1 = complexFunctionCall() // NOT ok  
^  
  
error: unresolved reference: complexFunctionCall  
const val CONSTANT1 = complexFunctionCall() // NOT ok  
^  
  
object Constants {  
    const val CONSTANT2 = "object constant"  
}  
val foo = Constants.CONSTANT2  
  
|<Ctrl+Enter> to execute
```

Step 2: Create a companion object

The basic difference between companion objects and regular objects is:

- Companion objects are initialized from the static constructor of the containing class, that is, they are created when the object is created.

- Regular objects are initialized lazily on the first access to that object; that is, when they are first used.

There is more, but all that you need to know for now is to wrap constants in classes in a companion object.

Understand extension functions


Step 1: Write an extension function

1. String is a valuable data type in Kotlin with many useful functions. But what if we needed some additional String functionality that wasn't directly available? For example, we might want to determine if a String has any embedded spaces.

```
fun String.hasSpaces(): Boolean {  
    val found = this.indexOf(' ')  
    // also valid: this.indexOf(" ")  
    // returns positive number index in String or -1 if not found  
    return found != -1  
}
```

2. You can simplify the hasSpaces() function. The this isn't explicitly needed, and the function can be reduced to a single expression and returned.

```
fun String.hasSpaces() = indexOf(" ") != -1
```

 *<Ctrl+Enter> to execute*

Step 2: Learn the limitations of extensions

1. Try adding extension functions that call a property marked private.

```
class AquariumPlant(val color: String, private val size: Int)  
  
fun AquariumPlant.isRed() = color == "red" // OK  
fun AquariumPlant.isBig() = size > 50 // gives error  
error: cannot access 'size': it is private in 'AquariumPlant'
```

Note: Extension functions are resolved statically, at compile time, based on the type of the variable.

2. Examine the code below and figure out what it will print.


```

open class AquariumPlant(val color: String, private val size: Int)

class GreenLeafyPlant(size: Int) : AquariumPlant("green", size)

fun AquariumPlant.print() = println("AquariumPlant")
fun GreenLeafyPlant.print() = println("GreenLeafyPlant")

val plant = GreenLeafyPlant(size = 10)
plant.print()
println("\n")
val aquariumPlant: AquariumPlant = plant
aquariumPlant.print() // what will it print?
GreenLeafyPlant
AquariumPlant

```

Step 3: Add an extension property

1. Add an extension property isGreen to AquariumPlant, which is true if the color is green.
2. Print the isGreen property for the aquariumPlant variable and observe the result.

```

val AquariumPlant.isGreen: Boolean
    get() = color == "green"

aquariumPlant.isGreen
res22: kotlin.Boolean = true

```

Step 4: Know about nullable receivers

1. Define a pull() method that takes a nullable receiver. This is indicated with a question mark ? after the type, before the dot. Inside the body, you can test if this is not null by using ?.apply.

```

fun AquariumPlant?.pull() {
    this?.apply {
        println("removing $this")
    }
}

val plant: AquariumPlant? = null
plant.pull()

```

2. In this case, there is no output when you run the program. Because plant is null, the inner println() is not called.