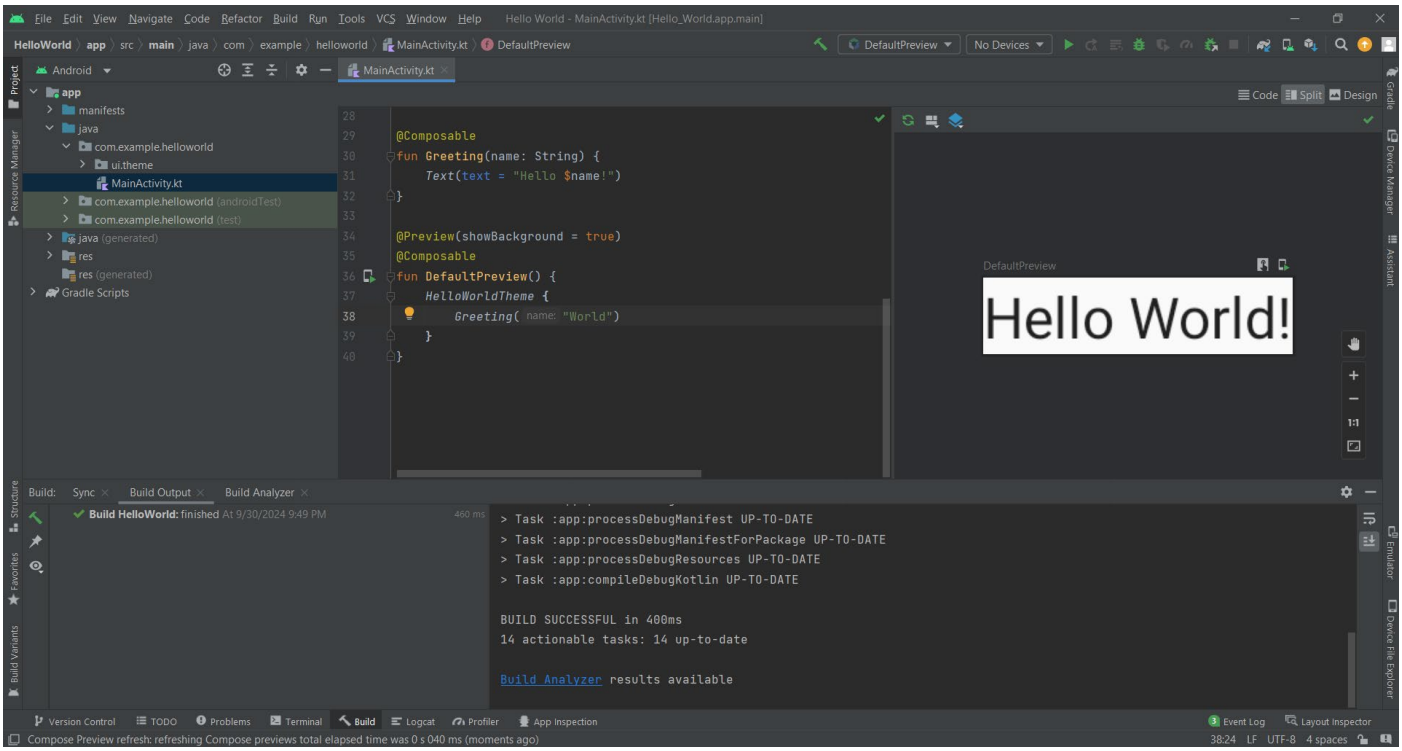
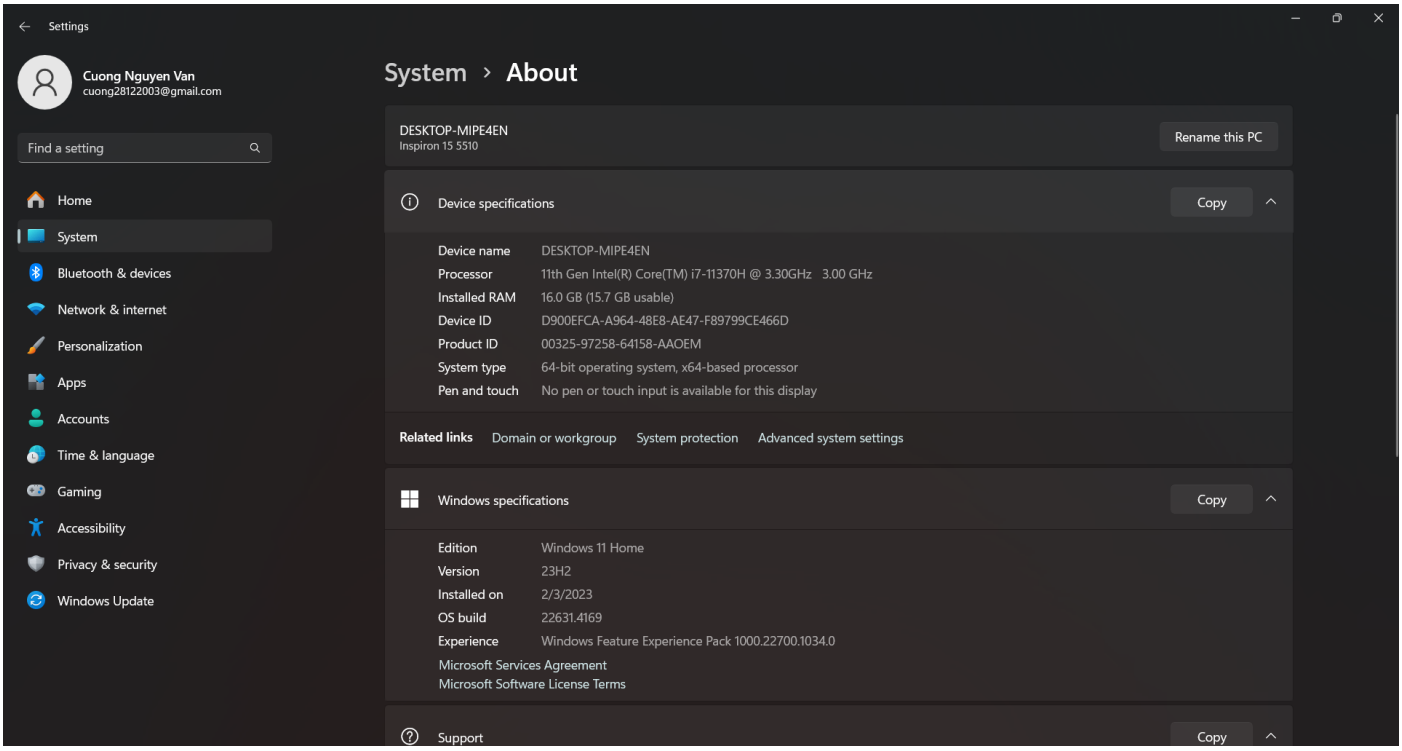
Name: Nguyễn Văn Cường – Student ID: 20215006

Report – Lesson 3

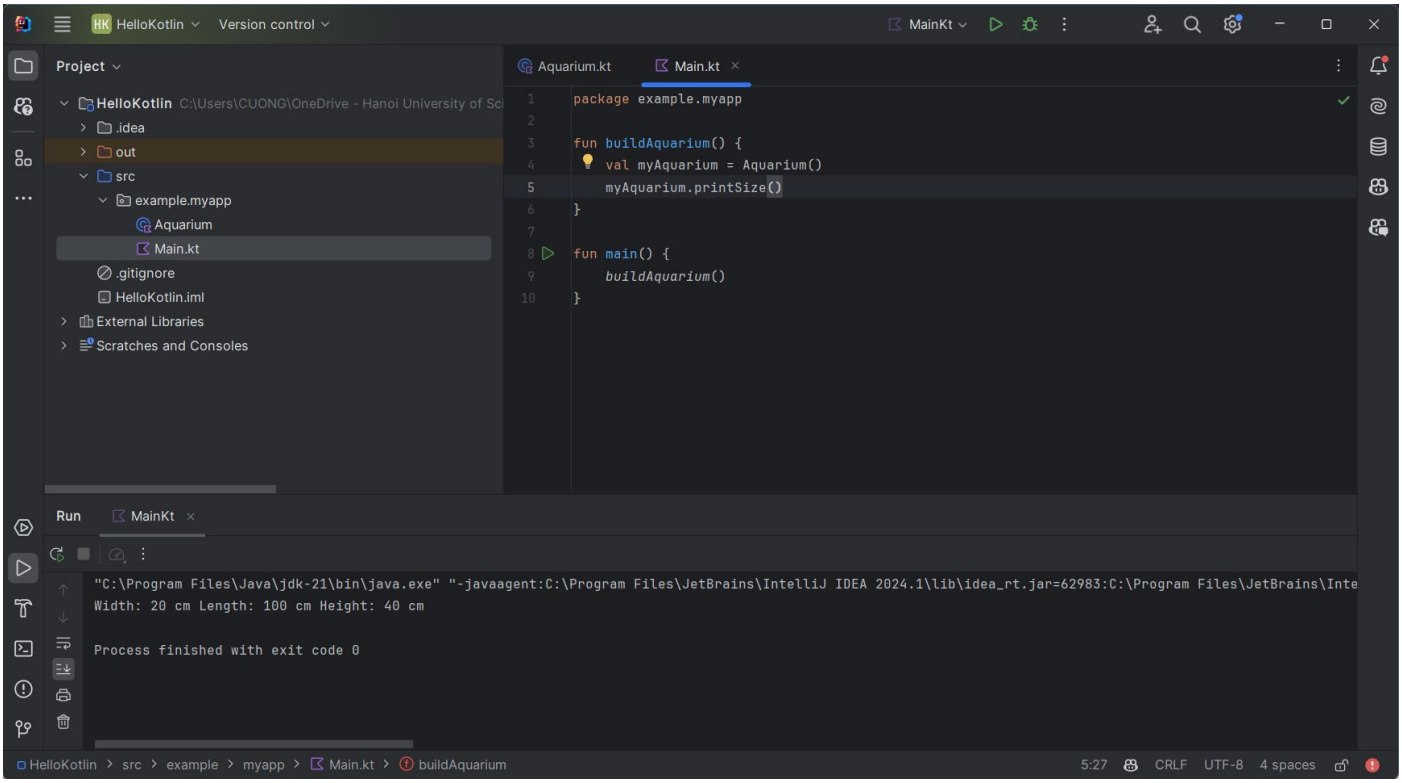
Demo Hello World

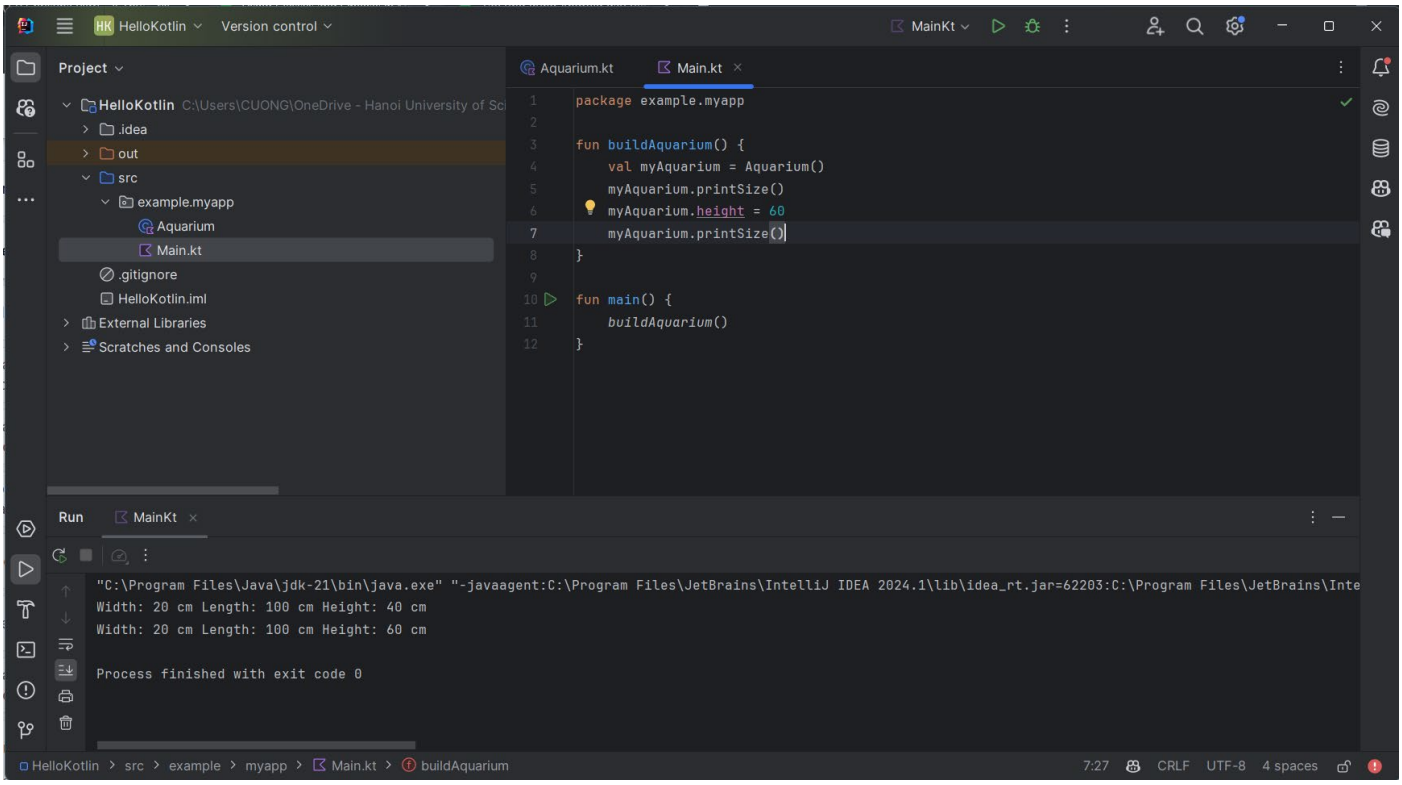




3.1 Using Classes and Objects in Kotlin

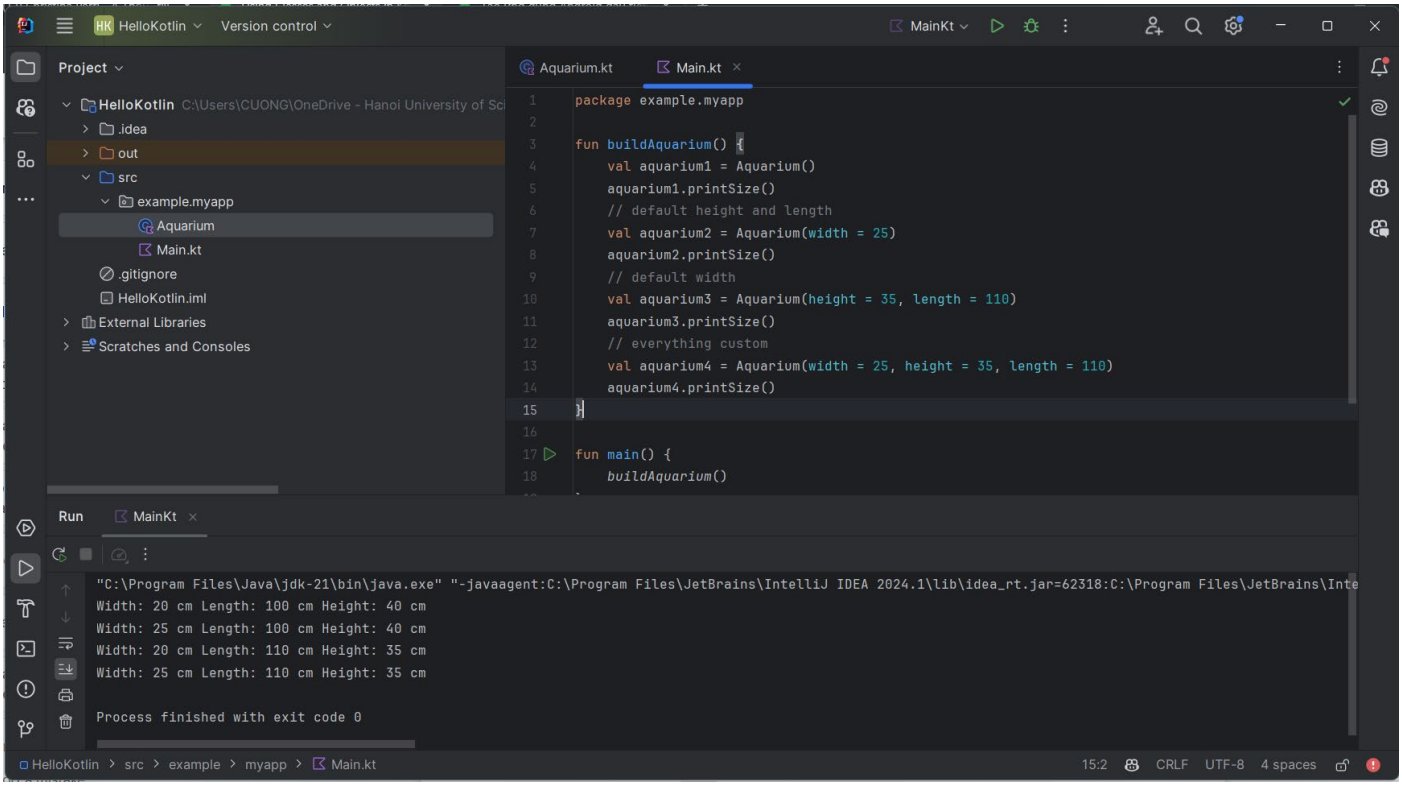
Create a class



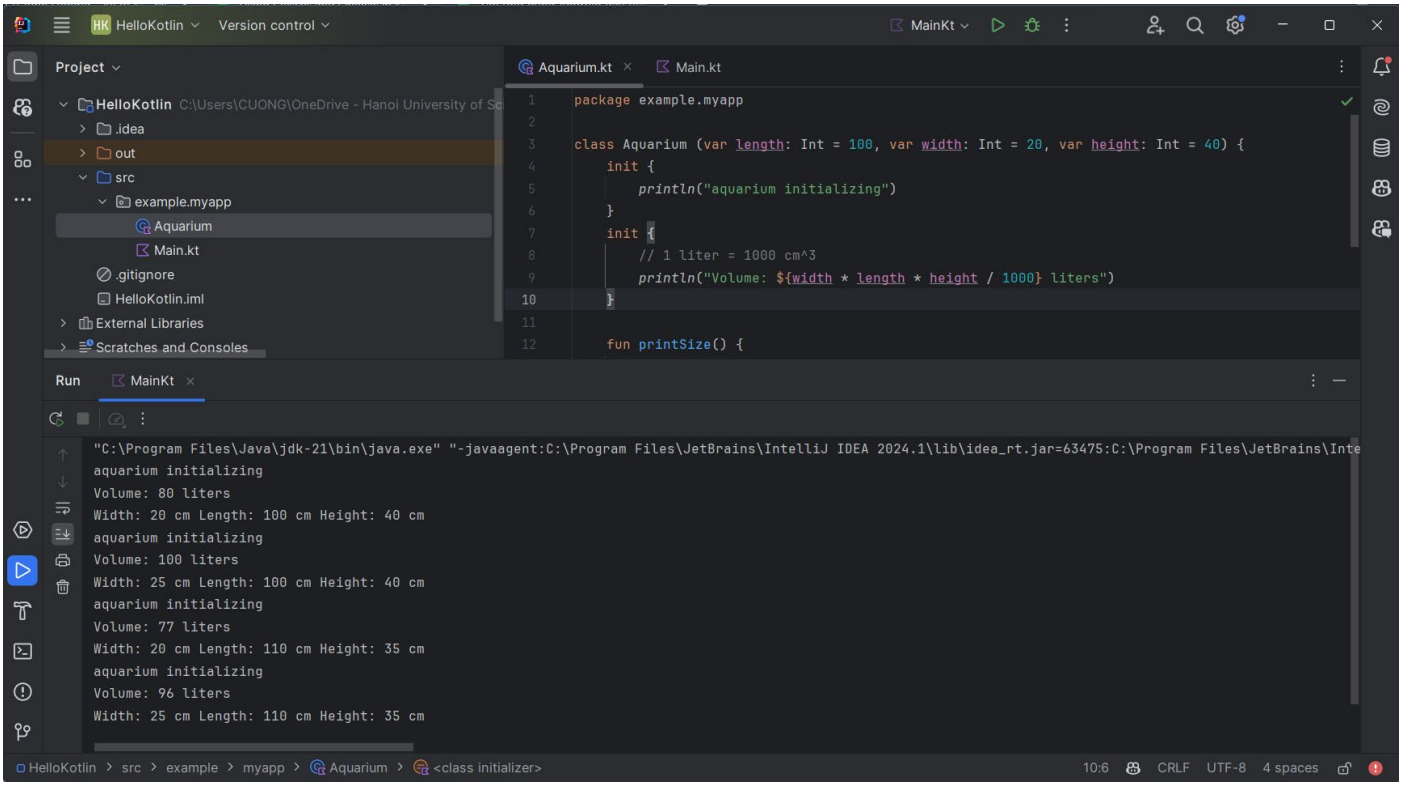


Add class constructors

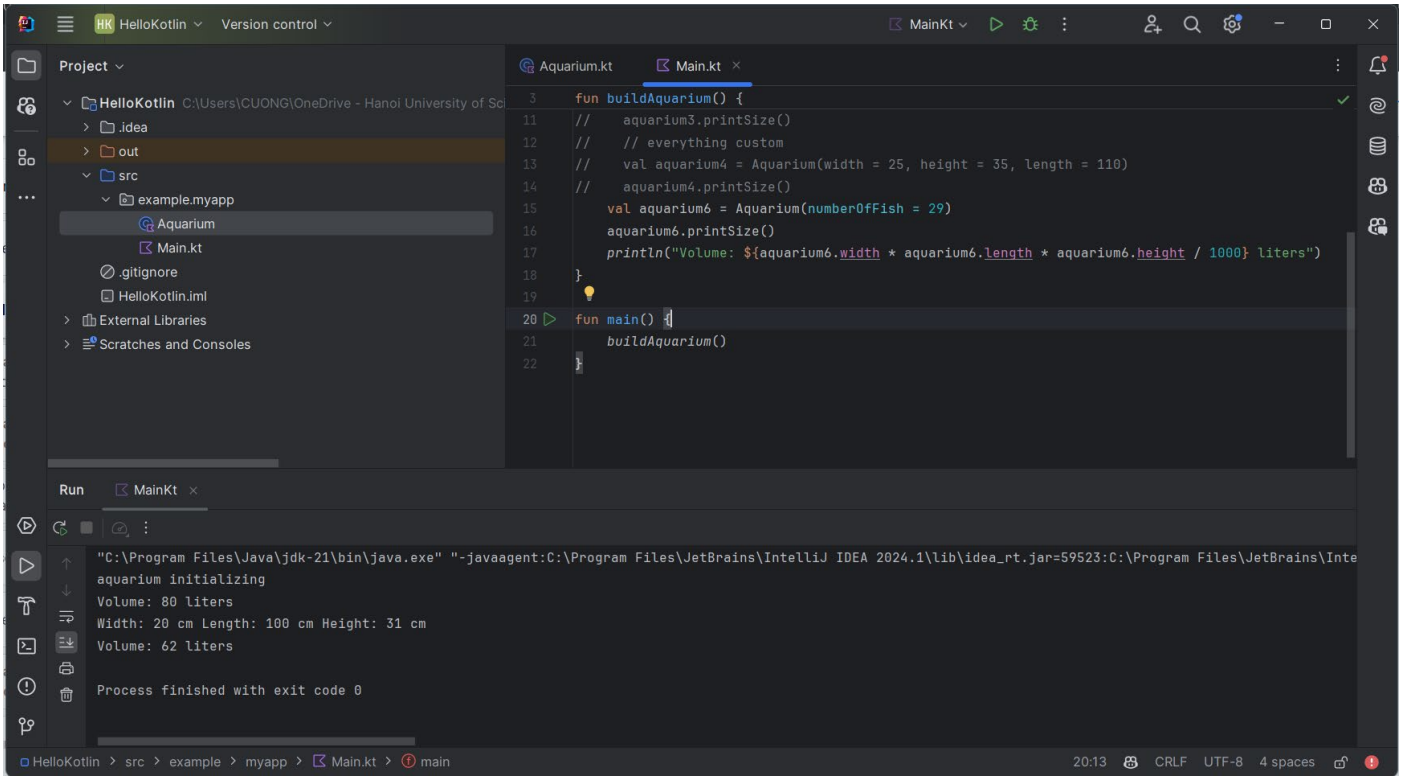
Step 1: Create a constructor



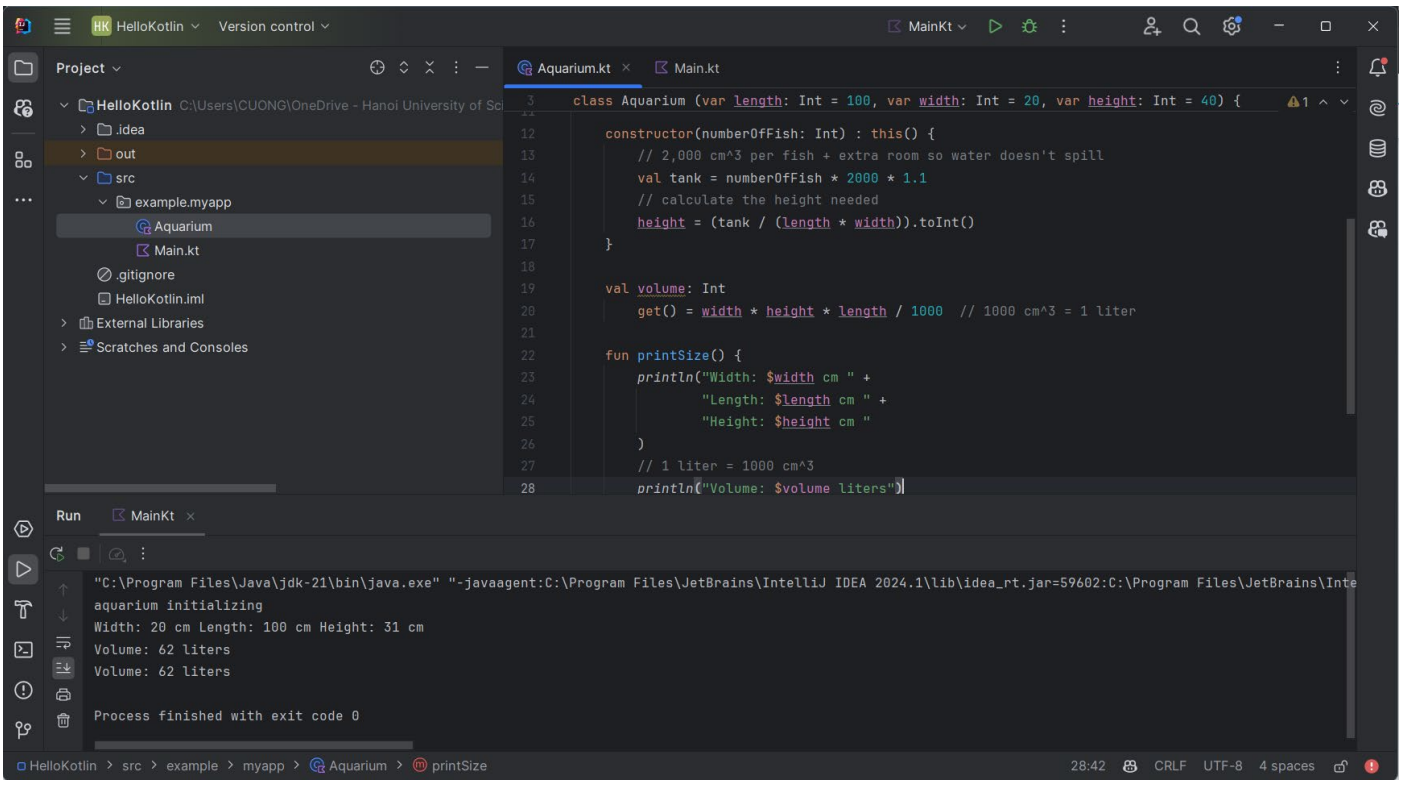
Step 2: Add init blocks



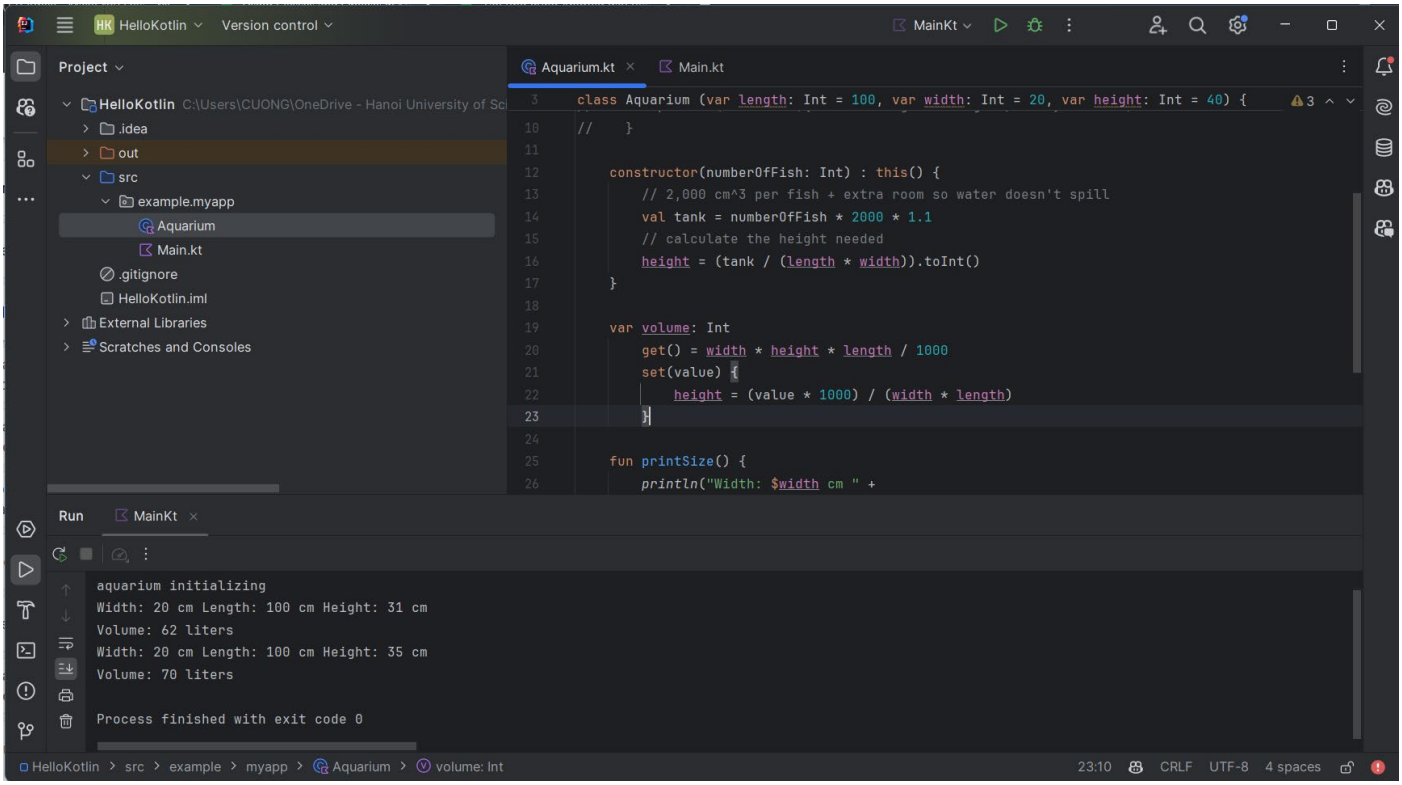
Step 3: Learn about secondary constructors



Step 4: Add a new property getter



Step 5: Add a property setter



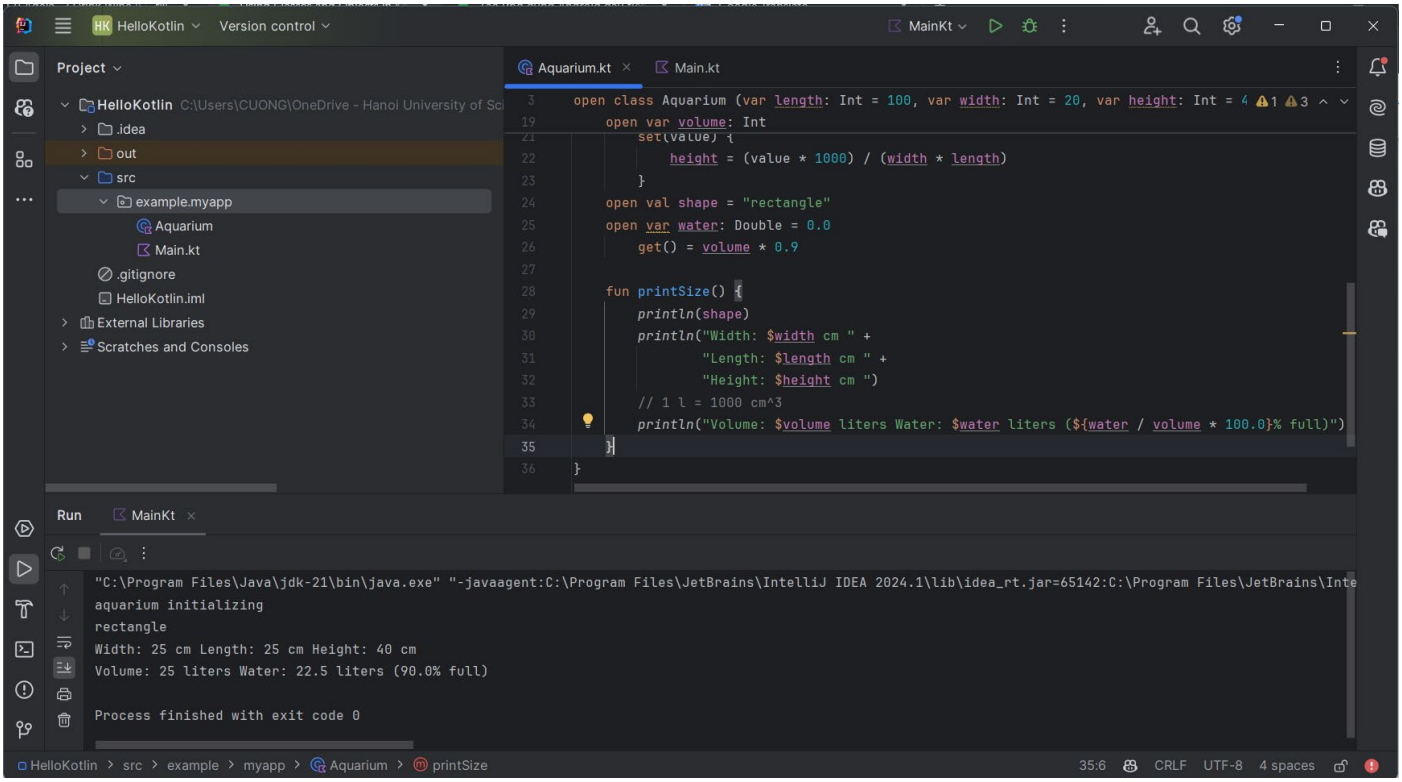
Learn about visibility modifiers

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

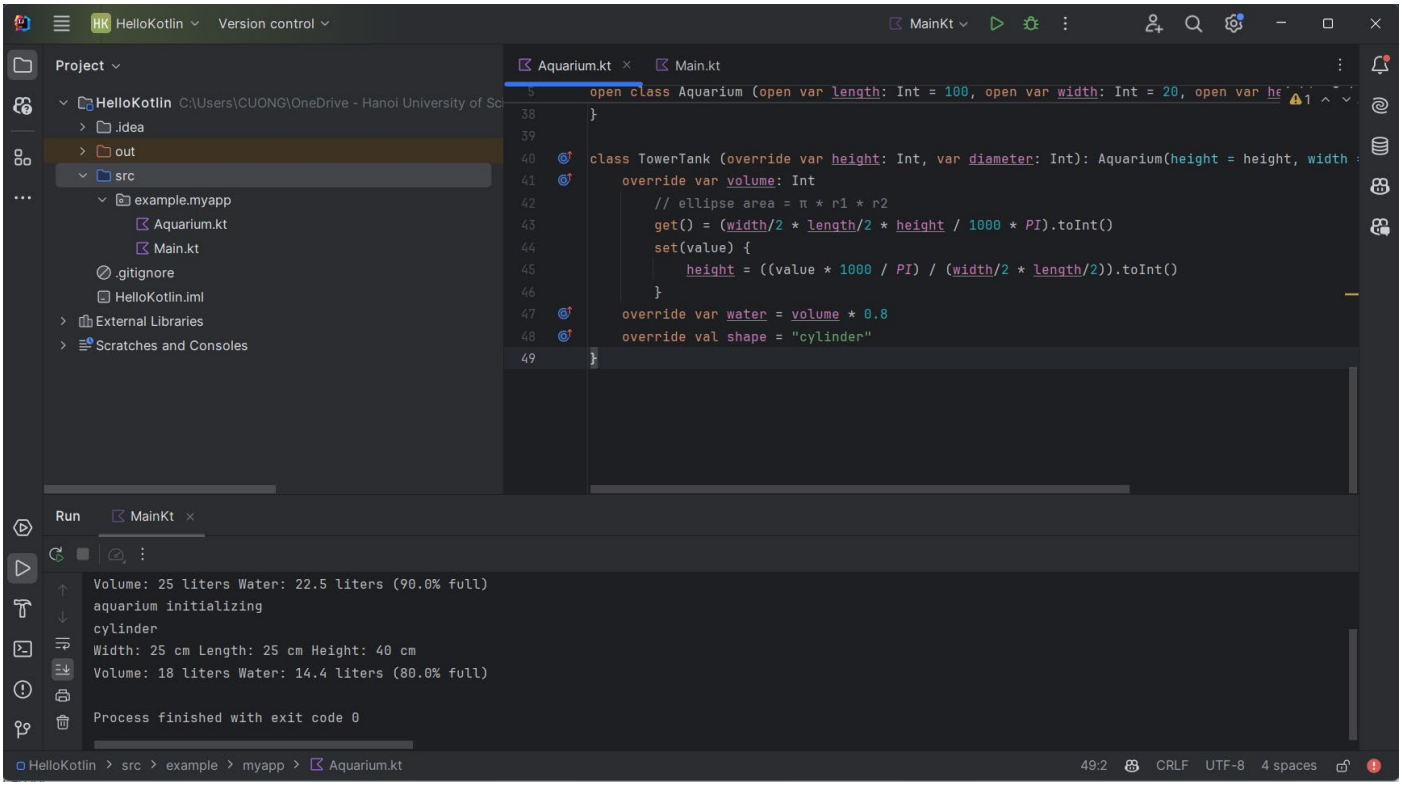
- private means it will only be visible in that class (or source file if you are working with functions).
- protected is the same as , but it will also be visible to any subclasses.private
- internal means it will only be visible within that module. A module is a set of Kotlin files compiled together, 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

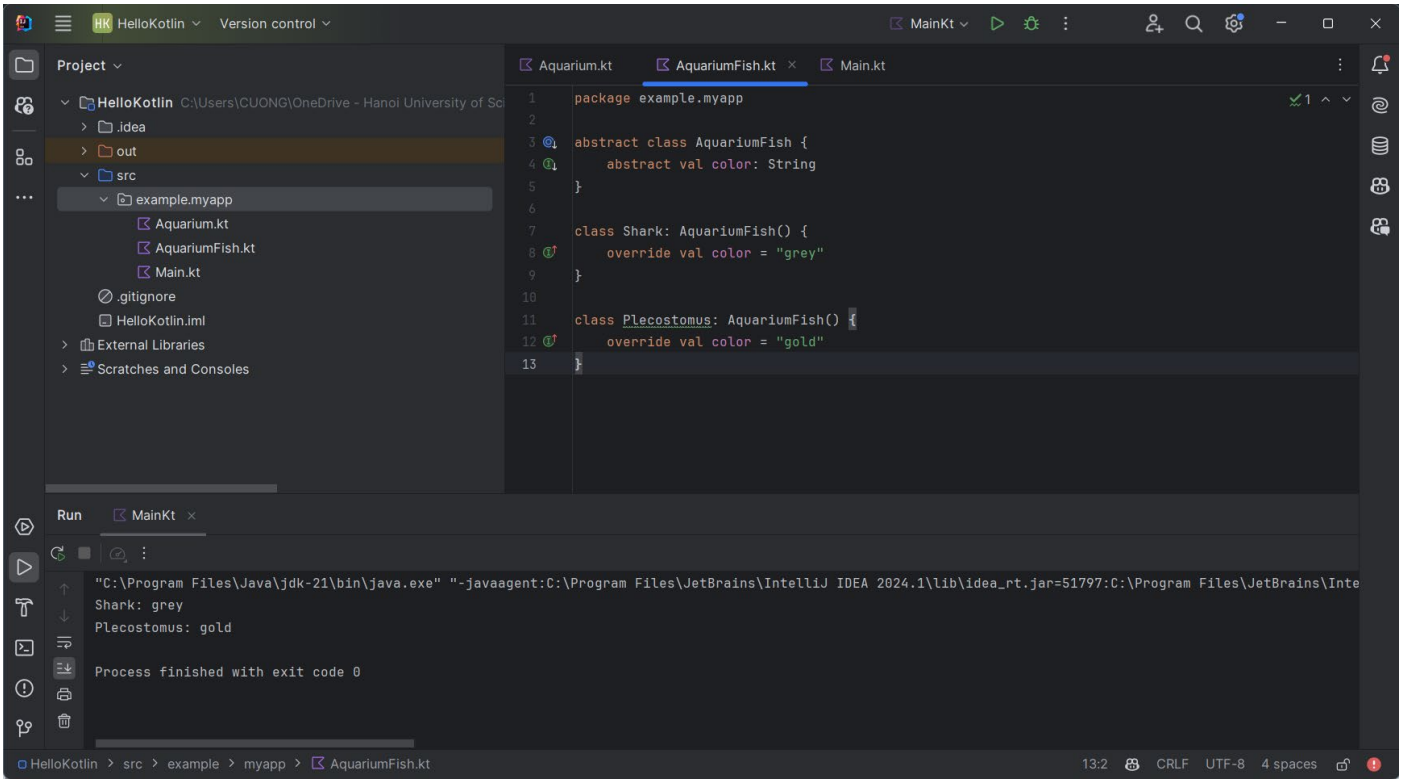


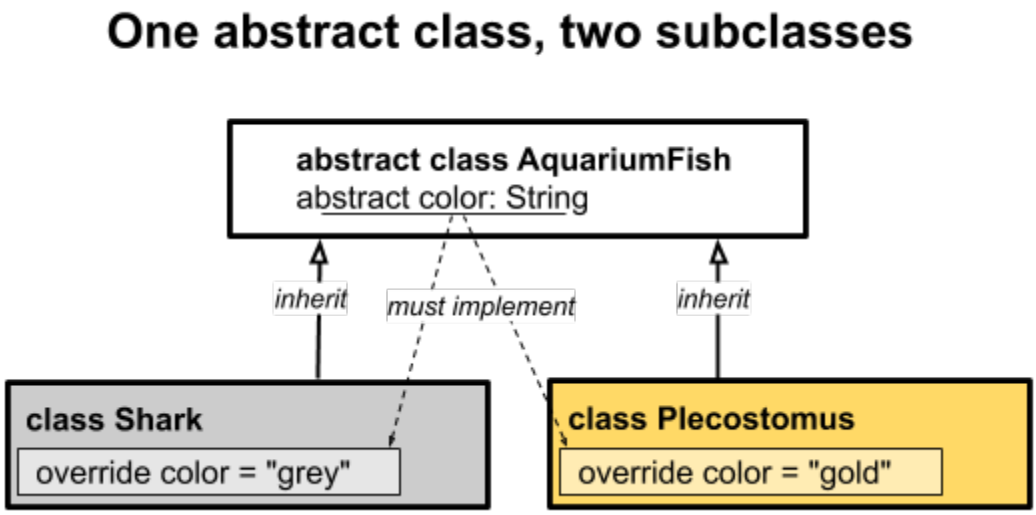
Step 2: Create a subclass



Compare abstract classes and interfaces

Step 1. Create an abstract class





Step 2: Create an interface

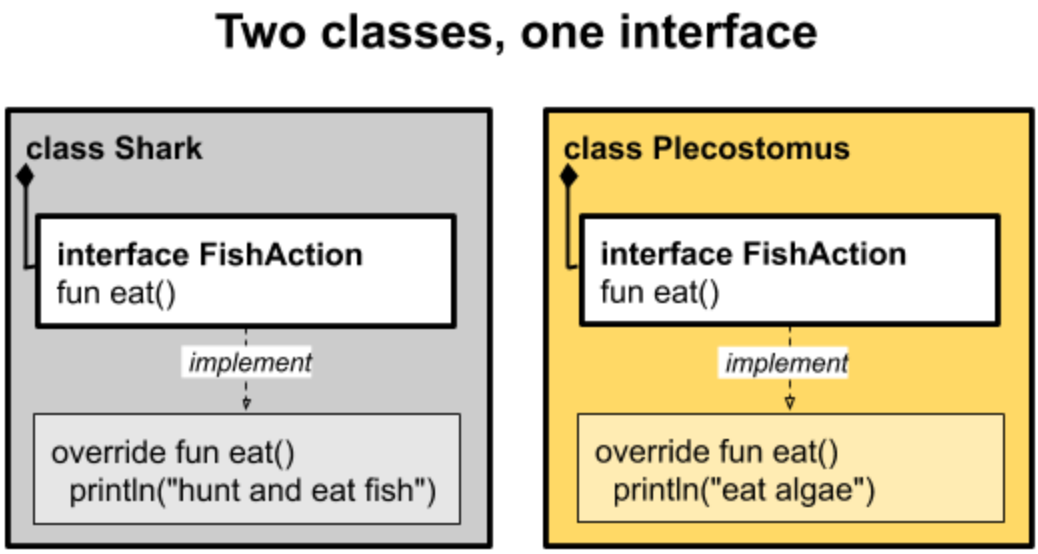
```
class Shark: AquariumFish(), FishAction {
    override val color = "grey"
    override fun eat() {
        println("hunt and eat fish")
    }
}

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

interface FishAction  {
    fun eat()
}
```

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

Process finished with exit code 0
```



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

```
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

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

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

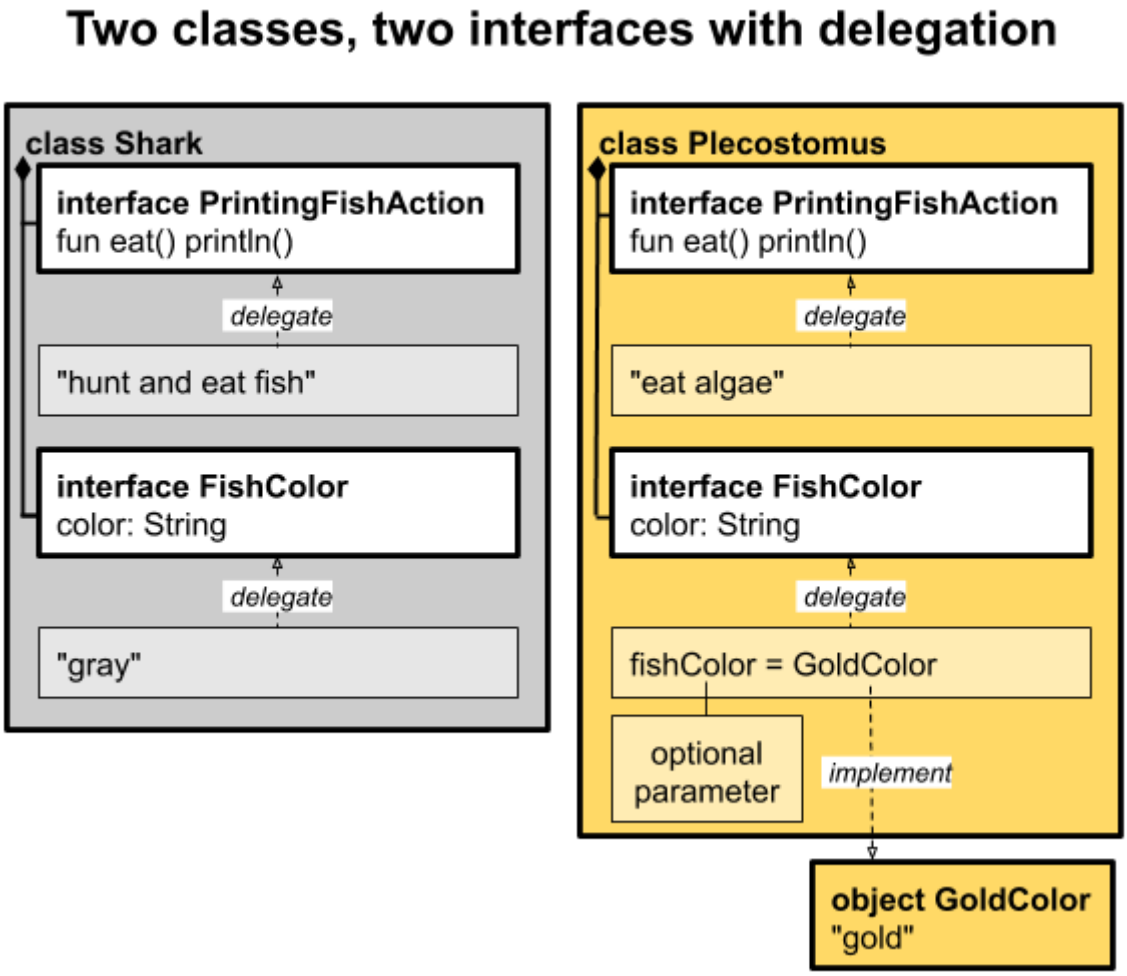
Step 3: Add interface delegation for FishColor

```
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
```



Create a data class

Step 1: Create a data class

```
package example.myapp.decor

data class Decoration(val rocks: String) {
}

fun makeDecorations() {
    val decoration1 = Decoration( rocks: "granite")
    println(decoration1)

    val decoration2 = Decoration( rocks: "slate")
    println(decoration2)

    val decoration3 = Decoration( rocks: "slate")
    println(decoration3)
    println (decoration1.equals(decoration2))
    println (decoration3.equals(decoration2))
}
```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\lib\idea_rt.jar=55702:C:\Program Files\JetBrains\Inte
Decoration(rocks=granite)
Decoration(rocks=slate)
Decoration(rocks=slate)
false
true

Process finished with exit code 0
```

Step 2. Use destructuring

```
// Here is a data class with 3 properties.
data class Decoration(val rocks: String, val wood: String, val diver: String){
}

fun makeDecorations() {
    val d5 = Decoration( rocks: "crystal",  wood: "wood",  diver: "diver")
    println(d5)

// Assign all properties to variables.
    val (rock, wood, diver) = d5
    println(rock)
    println(wood)
    println(diver)
}
```

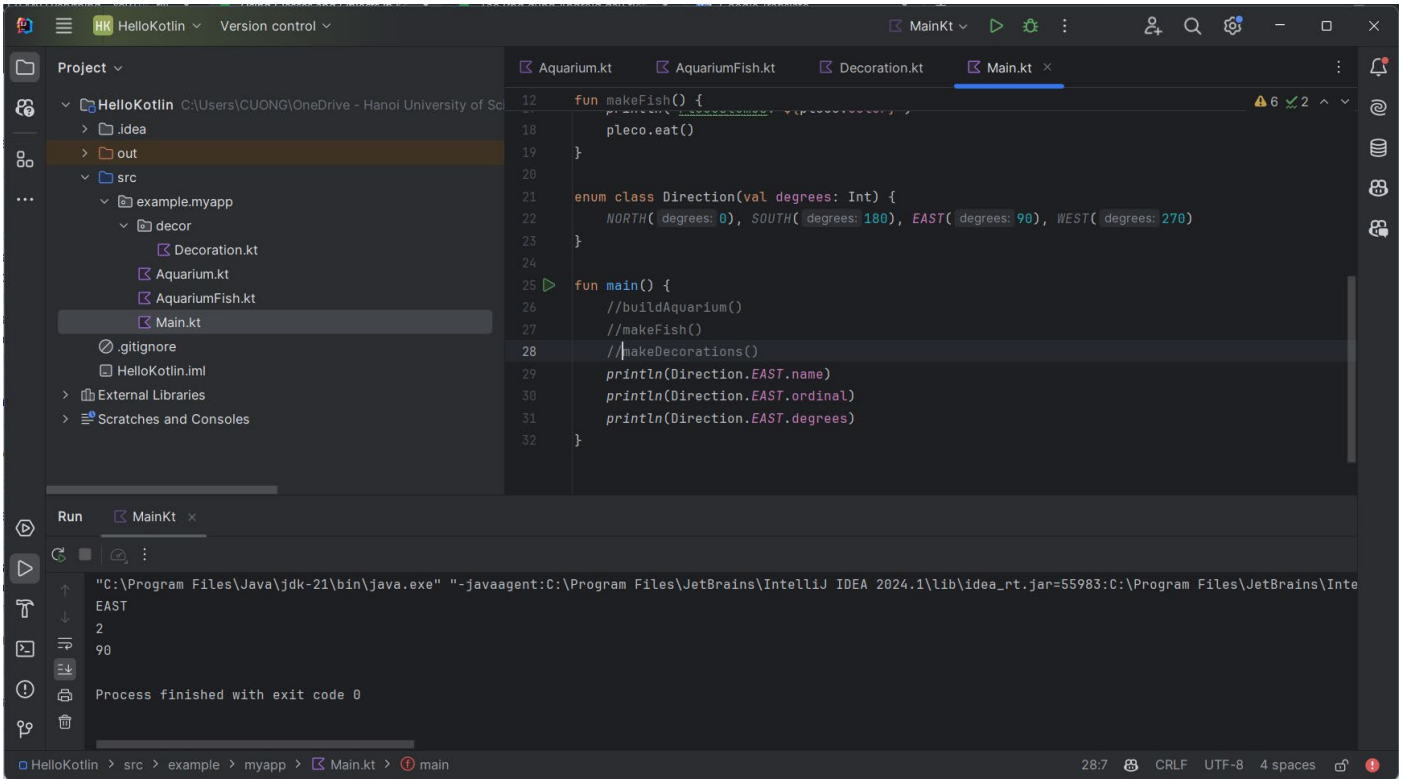
```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\lib\idea_rt.jar=55846:C:\Program Files\JetBrains\Inte
Decoration(rocks=crystal, wood=wood, diver=diver)
crystal
wood
diver

Process finished with exit code 0
```

Learn about singletons and enums

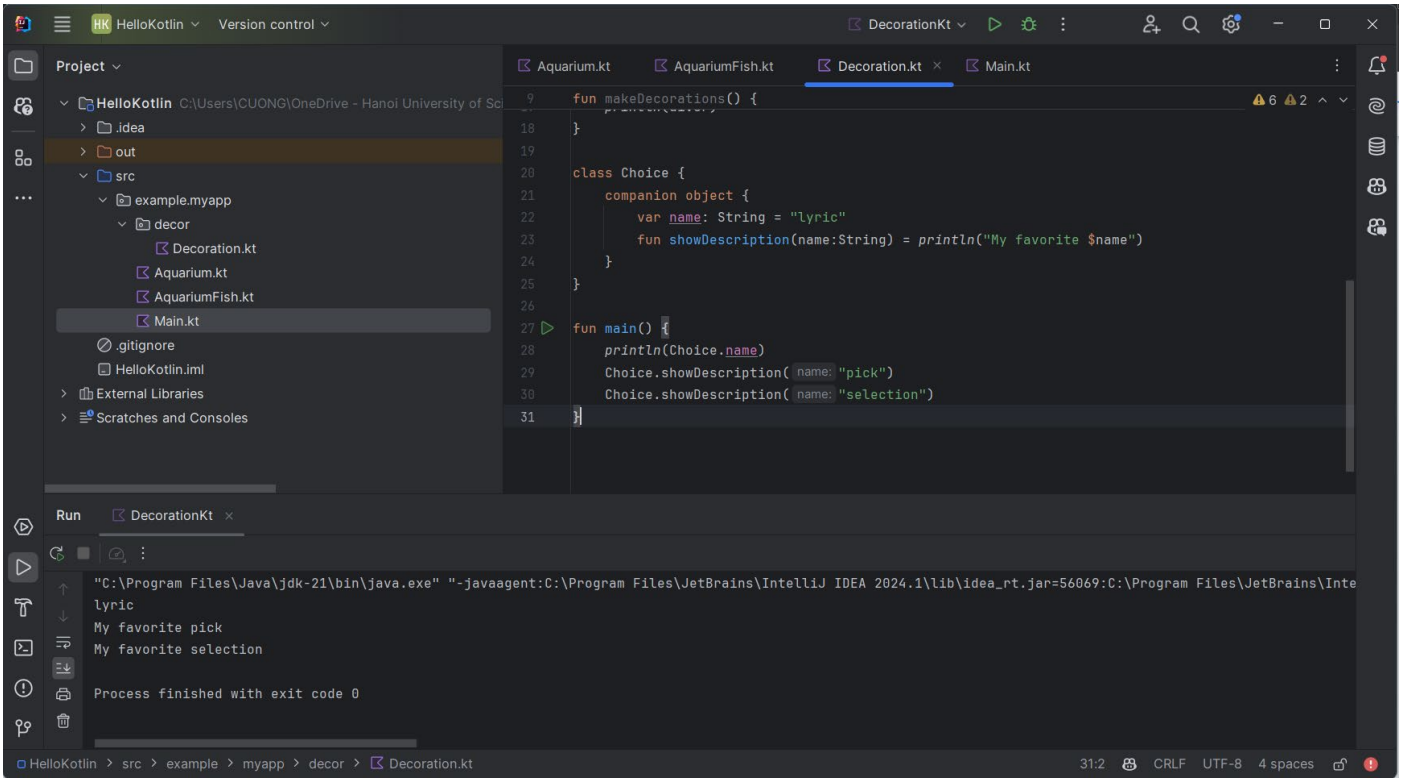
Step 1: Recall singleton classes

Step 2: Create an enum



3.2 Pairs/triples, collections, constants, and writing extension functions

Create a Companion Object



Learn about pairs and triples

Step 1: Make some pairs and triples

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
`add(element: E)`	Add an item to the mutable list.
`remove(element: E)`	Remove an item from a mutable list.
`reversed()`	Return a copy of the list with the elements in reverse order.
`contains(element: E)`	Return `true` if the list contains the item.
`subList(fromIndex: Int, toIndex: Int)`	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.

```
list2.
[](index: Int)                                          String
get(index: Int)                                         String
indexOf(element: String)                                Int
lastIndexOf(element: String)                            Int
listIterator()                                          ListIterator<String>
lastIndex for List<T> in kotlin.collections             Int
last() for List<T> in kotlin.collections                String
last {...} (predicate: (String) -> Boolean) for List<…  String
listIterator(index: Int)                                ListIterator<String>
subList(fromIndex: Int, toIndex: Int)                   List<String>
size                                                    Int
contains(element: String)                               Boolean
^↓ and ^↑ will move caret down and up in the editor >>
```

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.