## 1. Welcome

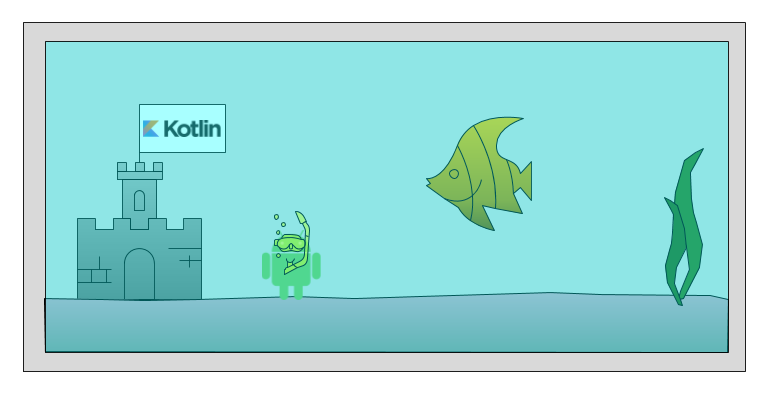
This codelab is part of the [Kotlin Bootcamp for Programmers course](https://developer.android.com/courses/kotlin-bootcamp/overview). You'll get the most value out of this course if you work through the codelabs in sequence. Depending on your knowledge, you may be able to skim some sections. This course is geared towards programmers who know an object-oriented language, and want to learn [Kotlin](https://kotlinlang.org/).



## Introduction

In this codelab, you create a Kotlin program and learn about classes and objects in Kotlin. Much of this content will be familiar to you if you know another object-oriented language, but Kotlin has some important differences to reduce the amount of code you need to write. You also learn about abstract classes and interface delegation.

Rather than build a single sample app, the lessons in this course are designed to build your knowledge, but be semi-independent of each other so you can skim sections you're familiar with. To tie them together, many of the examples use an aquarium theme. And if you want to see the full aquarium story, check out the [Kotlin Bootcamp for Programmers](https://classroom.udacity.com/courses/ud9011) Udacity course.



## What you should already know

* The basics of Kotlin, including types, operators, and looping
* Kotlin's function syntax
* The basics of object-oriented programming
* The basics of an IDE such as IntelliJ IDEA or Android Studio

## What you'll learn

* How to create classes and access properties in Kotlin
* How to create and use class constructors in Kotlin
* How to create a subclass, and how inheritance works
* About abstract classes, interfaces, and interface delegation
* How to create and use data classes
* How to use singletons, enums, and sealed classes

## What you'll do

* Create a class with properties
* Create a constructor for a class
* Create a subclass
* Examine examples of abstract classes and interfaces
* Create a simple data class
* Learn about singletons, enums, and sealed classes

2. 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.
* *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 import the package's contents into another file and reuse the code and classes in it.

## 3. Task: 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** 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](https://kotlinlang.org/docs/reference/coding-conventions.html#immutability) 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 new 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 ")

}

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

fun buildAquarium() {

val myAquarium = Aquarium()

myAquarium.printSize()

}

1. 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

1. 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()

}

1. 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

## 4. Task: 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, 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

...

}

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.

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

...

}

1. 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()

}

1. 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 block to print the volume in liters.

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} l")

}

}

1. Run the program and observe the output.

aquarium initializing

Volume: 80 l

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

aquarium initializing

Volume: 100 l

Width: 25 cm Length: 100 cm Height: 40 cm

aquarium initializing

Volume: 77 l

Width: 20 cm Length: 110 cm Height: 35 cm

aquarium initializing

Volume: 96 l

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 to allow constructor overloading, that is, constructors with different arguments.

**Note:** Kotlin coding style says each class should have only one constructor, using default values and named parameters. This is because using multiple constructors leads to more code paths, and the likelihood that one or more paths will go untested. Before writing a secondary constructor, consider whether a [factory function](https://kotlinlang.org/docs/reference/coding-conventions.html#factory-functions) would work instead, 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 called 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

}

1. 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()

1. 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} l")

}

1. Run your program and observe the output.

⇒ aquarium initializing

Volume: 80 l

Width: 20 cm Length: 100 cm Height: 31 cm

Volume: 62 l

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.

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 l*

1. Remove the init block that prints the volume.
2. Remove the code in buildAquarium() that prints the volume.
3. In the printSize() method, add a line to print the volume.

fun printSize() {

println("Width: $width cm " +

"Length: $length cm " +

"Height: $height cm "

)

*// 1 l = 1000 cm^3*

println("Volume: $volume l")

}

1. Run your program and observe the output.

⇒ aquarium initializing

Width: 20 cm Length: 100 cm Height: 31 cm

Volume: 62 l

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)

}

1. 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()

}

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

⇒ aquarium initialized

Width: 20 cm Length: 100 cm Height: 31 cm

Volume: 62 l

Width: 20 cm Length: 100 cm Height: 35 cm

Volume: 70 l