

Kotlin Beginners Notes

These are all personal notes taken from the Udacity Course (ud9011) of Kotlin Bootcamp for Programmers by Google as well as other resources. You can use it to learn Kotlin if you are a **beginner**, I have taken most of the things mentioned in the all sections/videos of this course including some brief pieces from official documentation and official video of JetBrains as well. I also cared about the order of the topics, so it starts from the basics from top to bottom goes to the more advanced ones. This is not an official documentation. You will be probably finding more personal notes and human mistakes :)



Table of Contents

- [Lesson 1 & 2 Introduction | Kotlin Basics](#)
- [Lesson 3 | Functions](#)
- [Lesson 4 | Classes](#)
- [Lesson 5 | Kotlin Essentials: Beyond The Basics](#)

Lesson 1 & 2 Introduction | Kotlin Basics

Package Definition and Imports

```
// Package specification should be at the top of the source file.
package my.demo

import kotlin.text.*

// Main function
fun main(args: Array<String>) {
    printHello ()
}

// Alternative main, no need to write parameters!
fun main() {
    printHello ()
}
```

```

}

// Prints "Hello Kotlin", println() puts newline, print()
fun printHello () {
    println("Hello Kotlin!")
}

// A function that returns "OK" string
fun start(): String = "OK"

// Function returns "Genesis", no need to write "return"
fun returnString () : String = "Genesis"

// Function returns "Genesis2" Alternative Way
fun returnString2 () : String {
    return "Genesis2"
}

```

Operators, -, +, /, *

```

// Returns integer
println(1 + 1) // Prints: 2
println(53 - 3) // Prints: 50
println(50 / 10) // Prints: 5
println(1 / 2) // Prints: 0
println(6 * 50) // Prints: 300

// Returns double
println(1.0 / 2.0) // Prints: 0.5
println(1.0 / 2) // Prints: 0.5
println(2 / 2.0) // Prints: 1.0

// Kotlin let's you overwrite the basic operators
// You can call methods on variables
val fish = 2
println(fish.times(6)) // Prints: 12
println(fish.div(10.0)) // Prints: 0.2
println(fish.plus(3)) // Prints: 5
println(fish.minus(3)) // Prints: -1

// You can use numbers(basic types) as if they were objects
// Use primitive 'int' as an object
1.toLong() // 1
println(false.not()) // true

```

Boxing

```

// Boxing describes the process of converting a primitive value to an object
// All numerical types in Kotlin have a supertype called Number
// Store value one in a variable of type Number
// It'll need to be placed in an object wrapper
// This is called boxing
val boxed: Number = 1
    ^      ^      ^
    name   type   value

val num: Int = 2
val dob: Double = 2.0

// Both lines do the exact same thing internally
Integer x = 42;
Integer y = Integer.valueOf(42);

// Eventhough this is very handy, it unfortunately leads to a decrease in performance
// We can avoid creating these objects wrappers by not storing numbers in objects
// There are two types of variables in Kotlin
// Changeable & Unchangeable
//   var           val

// With "val" you can assign value only once
val aquarium = 1
aquarium = 2 // -> ERROR! cannot be reassigned

// You can assign vals;
val str = "string"
val numInt = 1
val numDouble = 1.0
val bool = false

// With "var" you can assign a value, and then you can change it
var fish = 2
fish = 50
// Type is inferred meaning that compiler can figure out the type from the context
// Even so the type is inferred, it becomes fixed at compile time,
// So you cannot change a type of a variable in kotlin once it's type has been inferred
fish = "Bubbles" // ERROR

// We can use variables in operations and there is no punctuation at the end of the line
var str = 8
var a = 5
a + str
print(a + str)

// Number types won't implicitly convert to other types, so you can't assign a
// A short value to a long variable or a byte to an int
val b: Byte = 1

```

```

val i: Int = b // ERROR Type Mismatch

// But you can always assign them by casting like this;
val i: Int = b.toInt()

// Kotlin supports underscores in numbers
val oneMillion = 1_000_000
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_100100010

// You can specify long constants in a format that makes sense to you
// The type is inferred by Kotlin

```

Nullability

```

// Kotlin helps avoid null pointer exceptions
// When you declare a variable's type explicitly, by default its value cannot
var rocks: Int = null

// Use the question mark operator to indicate that a variable can be null
var rocks: Int? = null

// When you have complex data types such as a list,
var lotsOfFish: List<String> = listOf(null, null)

// You can allow for the list to be null, but if it is not null its elements
var evenMoreFish: List<String>? = null
var definitelyFish: List<String?>? = null

// Or you can allow both the list or the elements to be null
definitelyFish = listOf(null, null)

// Samples
// Creating List
var names5: List<String> = listOf("asd", "adsad3")

// Allow list to be null
var names: List<String>? = null

// Allow list items to be null
// But list cannot be null
var names2: List<String?> = listOf()

// ERROR, List cannot be null
var names3: List<String?> = null

// Allow both list items and list itself to be null

```

```

// But list cannot be null
var names4: List<String?>? = null

var a: String = "abc" // Regular initialization means non-null by default
a = null // compilation error
// it's guaranteed not to cause an NPE, so you can safely say:
val l = a.length

// To allow nulls, you can declare a variable as nullable string, written String?
var b: String? = "abc" // can be set null
b = null // ok
val l = b.length // error: variable 'b' can be null
print(b)

// List with some null items
var listWithNulls: List<String?> = listOf("Kot", null, "melo", null)

// Checking for null in conditions
// Option 1: First, you can explicitly check if b is null, and handle the two cases
    val l = if (b != null) b.length else -1

// Option 2: Safe calls, Your second option is the safe call operator, written ?. 
    val a = "Kotlin"
    val b: String? = null
    println(b?.length)
    println(a?.length) // Unnecessary safe call

// This returns "b.length" if "b is not null", and "null" otherwise. The type of the result is String?
// "b?.length" is equal to "if (b != null) b.length else -1"

    var b: String? = "abc"
    val l = if (b != null) b.length else -1
    val l2 = b?.length

    // Such a chain returns null if any of the properties in it is null.
    bob?.department?.head?.name

// If you want to do an operation on the non-null items
// To perform a certain operation only for non-null values, you can use the Elvis operator ?:
val listWithNulls: List<String?> = listOf("Kotlin", null)
for (item in listWithNulls) {
    item?.let { println(it) } // prints Kotlin and ignores null
}

// Print also nulls
var listWithNulls: List<String?> = listOf("Genesis", null, "melo", null)
print(listWithNulls)

```

Elvis operator "?: "

```

/*If the expression to the left of ?: is not null, the elvis operator returns the value of the expression to the right of ?:, otherwise it returns null*/

val l: Int = if (b != null) b.length else -1
val l = b?.length ?: -1

/*If the expression to the left of ?: is not null, the elvis operator returns the value of the expression to the right of ?:, otherwise it returns null*/

var b: String? = null
var l = b?.length ?: -1
print(l)

/*Since throw and return are expressions in Kotlin, they can also be used or */
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}

```

The "!" Operator & Not-null Assertion Operator "!!"

```

/*This is unsafe nullable type (T?) conversion to a non-nullable type (T), ! */

/*The third option is for NPE-lovers: the not-null assertion operator (!!) converts a nullable type (T?) to a non-nullable type (T)*/

    val l = b!!.length

/*Thus, if you want an NPE, you can have it, but you have to ask for it explicitly*/

```

Safe casts

```

/*Regular casts may result into a ClassCastException if the object is not of the expected type*/

// Safe Casts
var a = "1"
var b = 5
var aInt: Int? = a as? Int
var bInt: Int? = b as? Int
print(aInt) // null
print(bInt) // 5

```

Collections of a nullable type

```

/*If you have a collection of elements of a nullable type and want to filter out the null elements, you can use the filterNotNull() method*/

```

```

val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()

var nullableList: List<Int?> = listOf(1, null, 2, null, null, 5)
var nonNullList = nullableList.filterNotNull()
print(nonNullList) // Prints: [1, 2, 5]

// You can do some cool null testing with the question mark operator saving
// You can check if an object or variable is non null before accessing one c

val fishFoodTreats: Int? = null
return fishFoodTreats?.dec() ?: 0

val fishFoodTreats = 5
return fishFoodTreats?.dec() ?: 0
// You can also chain null tests in an expression
/*If "fishFoodTreats" is not null use a treat and return a new value and oth

```

Practice Time: Basic Operations

```

/*
Solve the following using the operator methods in one line of code.
If you start with 2 fish, and they breed twice, producing 71 offspring the 1

Hint: You can chain method calls.
Hint: You can call the methods on numbers, and Kotlin will convert them to c
Bonus question: What is special about all the numbers of fish?*/

// Solution Code
2.plus(71).plus(233).minus(13).div(30).plus(1)
// Bonus question: If you've noticed, all fish numbers above are prime.

// My One Line Solution
println(((2.times(71).plus(2.times(233))).minus(13)).div(30).plus((if (595.n

// To find how many fishes left
println((2.times(71).plus(2.times(233))).minus(13))

// To find Aquariums Needed
println(595.div(30) + if (595.mod(30) > 0) 1 else 0)

```

Practice Time: Variables

```

/*Create a String variable rainbowColor, set its color value, then change it

```

Create a variable blackColor whose value cannot be changed once assigned. Try

```
var rainbowColor: String = "green"
rainbowColor = "blue"
```

```
val blackColor: String = "you cannot change me// I am pure Black!"
blackColor = "White!"
```

//Alternative

```
var rainbowColor = "green"
rainbowColor = "blue"
val blackColor = "black"
blackColor = "white" // Error
```

Practice Time: Nullability

// Try to set rainbowColor to null. Declare two variables, greenColor and blueColor

```
var rainbowColor = "red"
rainbowColor = null // Error
```

```
var greenColor = null
var blueColor: Int? = null
```

Practice Time: Nullability/Lists

*// Create a list with two elements that are null; do it in two different ways
// Next, create a list where the list is null.*

```
// list with two null items
var list = listOf(null, null)
var list1: List<Int?> = listOf(null, null)
```

```
// The list2 itself is null
var list2: List<Int>? = null
```

Practice Time: Null Checks

*// Create a nullable integer variable called nullTest, and set it to null. Use the Elvis operator.
// Hint: Use the Elvis operator.*

```
var nullable: Int? = null
println(nullable?.inc() ?: 0)
println(nullTest?.inc() ?: 0)
```

Strings


```
"Hello Fish" // Hello Fish

// Concatenation
"hello" + "fish" // hello fish

val numberOfFish = 5
val numberOfPlants = 12

"I have $numberOfFish fish and $numberOfPlants plants" // I have 5 fish and

// Here two numbers get added first then the result will be printed
"I have ${numberOfFish + numberOfPlants} fish and plants" // I have 17 fish

val fish = "fish"
val plant = "plant"
println(fish == plant) // false
println(fish != plant) // true

val A = "A"
val B = "Z"
println(A < B) // true
println(A > B) // false
```

If-Else Blocks

```
val numberOfFish = 50
val numberOfPlants = 23
if (numberOfFish > numberOfPlants) {
    println("Good Ratio!")
} else {
    println("unhealthy ratio")
}
```

Ranges

```

val fish = 50
// .. -> inclusively 1 <= fish <= 50
if (fish in 1..50) {
    println(fish.toString() + " is in the range 1 <= fish <= 50!")
}
// until -> exclusively 1 <= fish < 50
if (fish in 1 until 50) {
    println(fish)
} else {
    println(fish.toString() + " is not in the range 1 <= fish < 50!")
}

```

When

```

// "when" is the way of switching in Kotlin
val numberOfFish = 50
when (numberOfFish) {
    0 -> println("Empty tank")
    50 -> println("Full tank")
    else -> println("Perfect!")
} // Output: Full tank

val numberOfFish = 50
when (numberOfFish) {
    in 1..50 -> println("Full tank")
} // Output: Full tank

// Create a string which would contain a * symbol n times.
val str: String = "*".repeat(100)

```

Practice Time

```

// Create three String variables for trout, haddock, and snapper.
// Use a String template to print whether you do or don't like to eat these
var trout: String = "trout"
var haddock: String = "haddock"
var snappe: String = "snappe"
var currentFish = trout
when (currentFish) {
    "trout" -> println("I love it!")
    "haddock" -> println("I like it")
    "snappe" -> println("I hate it")
    else -> println("That's enough fish")
}

val trout1 = "trout"

```

```

var haddock1 = "haddock"
var snapper1 = "snapper"
println("I like to eat $trout1 and $snapper1, but not a big fan of $haddock1

```

Practice Time

*/*when statements in Kotlin are like case or switch statements in other lang
Create a when statement with three comparisons:*

*If the length of the fishName is 0, print an error message.
If the length is in the range of 3...12, print "Good fish name".
If it's anything else, print "OK fish name".*/*

```

var fishName = "Salmon"
when (fishName.length) {
    0 -> println("Fish name cannot be empty!")
    in 3..12 -> println("Good fish name")
    else -> println("OK fish name")
}

```

Arrays & Loops

// If val variable value is a reference, then you cannot assign it a different

```

val myList = mutableListOf("tuna", "salmon", "shark");
myList = mutableListOf("Koi"); // ERROR// Cannot be re-assigned

```

// If you're referencing something that's not immutable(değişmez), it can still

// val only applies to the reference and it doesn't make the object it points to

// Here we cannot assign a different list in myList but we can manipulate the

```

val myList = mutableListOf("tuna", "salmon", "shark");
myList.remove("shark") // True
myList.add("fish") // True

```

For/While Loop Examples

```

val myList = mutableListOf("tuna", "salmon", "shark");
// Loop through an array
for (item in myList) {
    print(item + " ") // tuna salmon shark
}
// Loop through an array With index

```

```

for (index in myList.indices) {
    print(myList[index] + " ") // tuna salmon shark
}
for ((index, value) in myList.withIndex()) {
    println("the element at $index is $value") // the element at 0 is tuna..
}
//     for (i in array.indices) {
//         println(array[i])
//     }
// To iterate over a range of numbers, use a range expression
for (i in 1..5) {
    //     print(i.toString() + " ") /// Alternate
    print("$i ") // 1 2 3 4 5
}
for (c in 'a'..'z') {
    print("$c ") // a b c d e f g h i j k l m n o p q r s t u v w x y z
}
for (c in 'z' downTo 'a') {
    print("$c ") // z y x w v u t s r q p o n m l k j i h g f e d c b a
}
for (c in 10 downTo 0) {
    print("$c ") // 10 9 8 7 6 5 4 3 2 1 0
}
for (c in 10 downTo 0 step 2) {
    print("$c ") // 10 8 6 4 2 0
}
for (c in 1..10 step 2) {
    print("$c ") // 1 3 5 7 9
}

```

While loop

```

var x = 5
while (x > 0) {
    print("$x ") // 5 4 3 2 1
    x--
}
// Arrays work pretty much as you'd expect with some cool additions
// Good practice is to prefer using "lists" over "arrays" everywhere except

// It is pretty similar to Java
val l1 = listOf("a")
val l2 = listOf("a")
var x = (l1 == l2) // => true

val a1 = arrayOf("a")
val a2 = arrayOf("a")
var y = (a1 == a2) // => false

```

listOf vs mutableListOf

```

/*
List: READ-ONLY
MutableList: READ/WRITE
You can modify a MutableList: change, remove, add... its elements.
In a List you can only read them.

// Prefer MutableList over Array
// The major difference from usage side is that;
-> Arrays have a fixed size (like int [] in C++)
-> MutableList can adjust their size dynamically (like vectors in C++, a.k.a
-> Moreover Array is MUTABLE whereas List is not. (List is read-only, Array

// Difference between ArrayList<String>() and mutableListOf<String>() in Kot
-> The only difference between the two is communicating your intent :)
-> So, there is no difference, just a convenience method.*/

// Create an array
val school = arrayOf("fish", "tuna", "salmon")

// Create Typed array (e.g. integers)
val numbers = intArrayOf(1, 2, 3)

// Error, Type Mismatch
val test = intArrayOf(2, "foo")

// But you can mix types in Untyped arrays
val mixedArray = arrayOf("fish", 2, 's', 0.0)
for (element in mixedArray) {

```

```

    println(element) // fish 2 s 0.0
    // print(element.toString() + " ")
}

// This does not prints the all elements, it prints the array address instead
val mixedArray = arrayOf("fish", 2, 's', 0.0)
print(mixedArray) // [Ljava.lang.Object;@66d3c617

// You can use joinToString or forEach, forEachIndexed, Arrays.toString( arr
val mixedArray = arrayOf("fish", 2, 's', 0.0)
print(mixedArray.joinToString()) // fish, 2, s, 0.0
mixedArray.forEach { print("$it ") } // fish, 2, s, 0.0
mixedArray.forEachIndexed { index, any -> println("$any at $index") }
// fish at 0 ...
println(Arrays.toString(mixedArray)) // [fish, 2, s, 0.0]

```

Nesting Arrays

```

// You can nest arrays
val swarm = listOf(5, 12)
// When you put an array within an array, you have an array of arrays
// !Not a flattened array of the contents of the two
val bigSwarm = arrayOf(swarm, arrayOf("A", "B", "C"))
println(Arrays.toString(bigSwarm))
println(bigSwarm.asList()) // Shorter Printing Array Alternative
// Prints: [[5, 12], [Ljava.lang.String;@452b3a41]

// You can nest arrays
val intList = listOf(5, 12)
val stringList = mutableListOf("A", "B", "C")
// OR this -> val stringList = listOf("A", "B", "C")
// When you put "LIST or MUTABLELIST" within an array, you have an array of
val bigList = listOf(intList, stringList)
println(bigList.joinToString())
// [5, 12], [A, B, C]

```

Create Typed Lists, Mutablelists and Arrays

```

val intList = listOf<Int>(5, 12)
val stringList = listOf<String>("1", "2", "3", "4")

// Mutablelists
val intList = mutableListOf<Int>(5, 12)
val stringList = mutableListOf<String>("1", "2", "3", "4")

// Array

```

```

val intList = arrayOf<Int>(5, 12)
val stringList = arrayOf<String>("1","2","3","4")

// Sized array
var table = Array<String>(words.size) {""}
val literals = arrayOf<String>("January", "February", "March")

// Create 2D Array
val grid = Array(rows) { Array(cols) { Any() } }

//String[] in Java equivalent Array<String> in Kotlin
//eg.
var array1 : Array<String?> = emptyArray()
var array2: Array<String?> = arrayOfNulls(4)
var array3: Array<String> = arrayOf("Mashroom", "Kitkat", "Oreo", "Lolipop")

val num = arrayOf(1, 2, 3, 4) //implicit type declaration
val num = arrayOf<Int>(1, 2, 3) //explicit type declaration

// Or you can also create typed lists, arrays, mutable lists
val intList = listOf<Int>(5, 12)
val listSample: List<Int> = listOf(1,2,3)
val mutableListSample: MutableList<Int> = mutableListOf(1,2,3)
val stringList = listOf<String>("1","2","3","4")
val stringListSample: List<String> = listOf<String>("1","2","3","4")
val initList = List(4){"s"} // {"s", "s", "s", "s"}

// Arrays
val intArray = arrayOf(1,2,3)
val intArray2: Array<Int> = arrayOf(1,2,3)
val intArray3 = intArrayOf(1,2,3)
val charArray = charArrayOf('a', 'b', 'c')
val intArray = arrayOf(1,2,3)
val intArray2: Array<Int> = arrayOf(1,2,3)
val intArray3 = intArrayOf(1,2,3)
val charArray = charArrayOf('a', 'b', 'c')
val stringArray = arrayOf("genesis", "melo")
val stringArray2: Array<String> = arrayOf("genesis", "melo")
val stringOrNulls = arrayOfNulls<String>(5) // returns Array<String?>
val stringOrNulls2: Array<String?> = arrayOf("", null)// returns Array<Strir
var emptyStringArray: Array<String> = emptyArray()
var emptyStringArray2: Array<String> = arrayOf()
var sizedEmptyArray = Array(4){"s"} // {"s", "s", "s", "s"}

// In this line, we create an array from a range of numbers.
val nums3 = IntArray(5, { i -> i * 2 + 3})
// This line creates an array with IntArray. It takes the number of elements
// This is the output.
/*
[1, 2, 3, 4, 5]

```

```
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
[3, 5, 7, 9, 11]
```

```
*/
```

```
// You can read this as initialize an array of 5 elements, assign each item
```

```
val array = Array(5) {it * 2}
```

```
// OR -> val array = List(5) {it * 2}
```

```
println(array.joinToString()) // 0, 2, 4, 6, 8
```

```
val list = List(5){ it.times(2) } // Creates List: [0, 2, 4, 6, 8]
```

```
val array = Array<Int>(5) { e -> if (e % 2 == 0) 2 else 1 }
```

```
println(array.asList())
```

```
val array = List<String>(5) { e -> if (e % 2 == 0) "even" else "$e" }
```

```
println(array.joinToString()) // Prints: even, 1, even, 3, even
```

```
// Loop array with indices
```

```
val swarm = listOf(5, 12, 15, 17)
```

```
for (i in 0 until swarm.size) {
```

```
    print("$i ") // 0 1 2 3
```

```
}
```

```
for (i in 0..swarm.size - 1) {
```

```
    print("$i ") // 0 1 2 3
```

```
}
```

```
for (i in swarm.indices) {
```

```
    print("$i ") // 0 1 2 3
```

```
}
```

```
for (indexValuePair in swarm.withIndex()) {
```

```
    print("index: ${indexValuePair.index}, value: ${indexValuePair.value}\n")
```

```
} // Prints: index: 0, value: 5
```

Quiz

```
// Read the code below, and then select the correct array initialization that
```

```
val array = // initialize array here
```

```
val sizes = arrayOf("byte", "kilobyte", "megabyte", "gigabyte",  
    "terabyte", "petabyte", "exabyte")
```

```
for ((i, value) in array.withIndex()) {
```

```
    println("1 ${sizes[i]} = ${value.toLong()} bytes")
```

```
}
```

```
// Output:
```

```
1 byte = 1 bytes
```



```

1 kilobyte = 1000 bytes
1 megabyte = 1000000 bytes
1 gigabyte = 1000000000 bytes
1 terabyte = 1000000000000 bytes
1 petabyte = 1000000000000000 bytes
1 exabyte = 1000000000000000000 bytes

```

// Answer / Solution Code:

```
val array = Array(7){ 1000.0.pow(it) }
```

// Notice how we had to use the double value 1000.0 and not just 1000 to be

Quiz

/ Which of these options are good reasons to explicitly make a list immutable
-> It reduces errors in general.
-> Prevents accidental changing of objects that were meant to be unchangeable
-> In a multi-threaded environment, makes the variable thread safe, because*

// Answer: Immutable variables are the safest option when you know that a variable

Practice Time

*/*Looping over arrays and lists is a fundamental technique that has a lot of*

Basic example

Create an integer array of numbers called numbers, from 11 to 15.

Create an empty mutable list for Strings.

Write a for loop that loops over the array and adds the string representation

Challenge example

How can you use a for loop to create (a list of) the numbers between 0 and 100

// Solution Code

```
var list3 : MutableList<Int> = mutableListOf()
```

```
for (i in 0..100 step 7) list3.add(i)
```

```
print(list3)
```

```
[0, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
```

// OR

```
for (i in 0..100 step 7) println(i.toString() + " - ")
```

// My Solution

```
val numbers = Array<Int>(5) { it + 11 }
```

```
println(numbers.asList())
```

```
var mutableList = mutableListOf<String>()
```

```

for (number in numbers) {
    mutableList.add(number.toString())
}
println(mutableList)
// Challenge Example
for (number in 7..100 step 7) {
    println("$number ")
}

```

Kotlin Lists (from JetBrains Official Video)

```

listOf(1, 2, 3)
// [1, 2, 3]

val l2 = List(5){ "No. $it" }
val l3 = List(5){ idx -> "No. $idx" }
// [No. 0, No. 1, No. 2, No. 3, No. 4]

val l4 = "word-salad".toList()
// [w, o, r, d, -, s, a, l, a, d]

val m1 = mapOf(
    1 to "Gold",
    2 to "Silver",
    3 to "Bronze"
).toList()
// [(1, Gold), (2, Silver), (3, Bronze)]

generateSequence {
    Random.nextInt(100).takeIf { it > 30 }
}.toList()
// [45, 75, 74, 31, 54, 36, 63]

(0..5).toList()
// [0, 1, 2, 3, 4, 5]

val mutList = mutableListOf(1, 2, 3)
val otherList = mutList.toList()

mutList[0] = 5

mutList
// [5, 2, 3]

otherList
// [1, 2, 3]

val myList = listOf("A", "B", "C")

```

```

myList.get(0)
myList[0]

myList.getOrElse(3)
val test = myList.getOrElse(3) {
    println("There is no index $it")
    ":"
}
// There is no index 3
// :(

val listOfNullableItems = listOf(1, 2, null, 4)
// Elvis operator checks if item at index 2 null, if it is null it returns 0
// returns the element at index 2 otherwise
val item = listOfNullableItems[2] ?: 0

val myList1 = listOf("a", "b", "c", "d", "e")
myList1.slice(listOf(0, 2, 4))
// [a, c, e]

myList1.slice(0..3)
// [a, b, c, d]

myList1.slice(0..myList1.lastIndex step 2)
// [a, c, e]

myList1.slice(2 downTo 0)
// [c, b, a]

mutableListOf(1, 2, 3)
// [1, 2, 3]

(0..5).toMutableList()
// [0, 1, 2, 3, 4, 5]

listOf(1, 2, 3).toMutableList()
// [1, 2, 3]

val m = mutableListOf(1, 2, 3)
m.add(4)
m += 4
// [1, 2, 3, 4, 4]

m.add(2, 10)
// [1, 2, 10, 3, 4, 4]

// Append a list to another list
m += listOf(5, 6, 7)
// [1, 2, 10, 3, 4, 4, 5, 6, 7]

```

```

val mList = mutableListOf(1, 2, 3, 3, 3, 4)
mList -= 3
mList.remove(3)
// [1, 2, 3, 4]

// Removes all instances of given elements from the mList
mList -= listOf(1, 4)
// [2, 3]

mList.removeAt(1)
// [2]

mList[0] = 5
// [5]

val fruits = mutableListOf("Apple", "Apricot", "Cherry")
fruits.fill("sugar")
// [sugar, sugar, sugar]
fruits.clear()
// []

// The following will create a new list and return it
val list = listOf(3, 1, 4, 1, 5, 9)
list.shuffled()
// [5, 3, 4, 1, 1, 9]

list.sorted()
// [1, 1, 3, 4, 5, 9]

list.reversed()
// [9, 5, 1, 4, 1, 3]

// The following will do the operations "in-place" without creating a new li
val mm = list.toMutableList()
mm.shuffle()
// [5, 3, 4, 1, 1, 9]

mm.sort()
// [1, 1, 3, 4, 5, 9]

mm.reverse()
// [9, 5, 1, 4, 1, 3]

val numbers = mutableListOf(3, 1, 4, 1, 5 ,9)
numbers.removeAll { it < 5 }
// [5, 9]

val letters = mutableListOf('a', 'b', '3', 'd', '5')
letters.retainAll { it.isLetter() }
// [a, b, d]

```

```

val letters2 = mutableListOf("A", "B", "C", "D")
val sub = letters2.subList(1, 4) // [ inclusive, exclusive )
// [B, C, D]

// There is only one list here, sub list is just a reference, a view of lett
// And they are reflecting each other
letters2[1] = "Z"
println(sub)
// [Z, C, D]

sub[2] = "MM"
println(letters2)
// [A, Z, C, MM]

sub.fill("FF")
println(letters2)
// [A, FF, FF, FF]

sub.clear()
println(letters2)
// [A]

letters2.clear()
//      println(sub) // ERROR// Because there is no more original list

val nums = mutableListOf(1, 2 ,3, 4)
// A reversed view of nums
val smun = nums.asReversed()

println(smun)
// [4, 3, 2, 1]

nums[1] = 99
println(smun)
// [4, 3, 99, 1]

smun[2] = -1
println(nums)
// [1, -1, 3, 4]

```

Lesson 3 | Functions

```

// A function like main returns a type "UNIT" which is Kotlin's way of sayir
fun main(args: Array<String>) {
    println("Hello, world!")
    println(test()) // kotlin.Unit
}

```

```
fun test() {  
}
```

Practice Time

*/*Basic Task*

Create a new Kotlin file.

Copy and paste the main() function from Hello World into the file.

Create a new function, dayOfWeek().

In the body of the function, print "What day is it today?"

Call dayOfWeek() from main().

Run your program.

Task List

Extended Task

In the body of the dayOfWeek() function, answer the question by printing the

Hints

*You can use Java libraries (java.util) from Kotlin. For example, to get the
Calendar.getInstance().get(Calendar.DAY_OF_WEEK)*

Type in the code, then press Option + Enter in Mac, or Alt + Enter in Windows

Use a when statement/*

// Answer:

```
import java.util.*  
fun main(args: Array<String>) {  
    dayOfWeek()  
}  
  
fun dayOfWeek() {  
    println("What day is it today?")  
    val day = Calendar.DAY_OF_WEEK  
    println(when(day) {  
        1 -> "Monday"  
        2 -> "Tuesday"  
        3 -> "Wednesday"  
        4 -> "Thursday"  
        5 -> "Friday"  
        6 -> "Saturday"  
        7 -> "Sunday"  
        else -> "Time has stopped"  
    })  
}
```

```

// Run -> Edit COnfigurations -> Program Args: Kotlin
fun main(args: Array<String>) {
    println("Hello, ${ args[0] }") // Hello, Kotlin
}

// Fetching the first element of an array is EXPRESSION
// not a value, that why we used ${ args[0] }

// In Kotlin almost everthing has a value, even if that value is unit
// Everything in Kotlin is an expression
// You can use the value of an "if" expression right away

val isUnit = println("This is an expression")
println(isUnit)
// This is an expression
// kotlin.Unit

val temperature = 10
val isHot = if (temperature > 50) true else false
println(isHot) // false

val message = "You are ${ if (temperature > 50) "fried" else "safe" } fish"
println(message) // You are safe fish

```

Exercise: Greetings, Kotlin

*/*Create a main() function that takes an argument representing the time in 2 (values between and including 0 -> 23).*

In the main() function, check if the time is before midday (<12), then print

Notes:

Remember that all main() function arguments are Strings, so you will have to

Advanced

Try to use Kotlin's string templates to do this in 1 line./*

// Your reflection

```

fun main(args: Array<String>) {
    println(if (args[0].toInt() < 12) "Good morning, Kotlin" else "Good nigh
}

```

*/*Things to think about*

There are multiple ways you can do this in Kotlin. Make sure you test your c

Here's one way to do it:/*

```

if (args[0].toInt() < 12) println("Good morning, Kotlin")
else println("Good night, Kotlin" )

// OR

println("${if (args[0].toInt() < 12) "Good morning, Kotlin" else "Good night"

// CTRL + ALT + L -> Indent File

// Repeat an action x times
repeat(10) { i ->
    println("This line will be printed 10 times")
    println("We are on the ${i + 1}. loop iteration")
}

// Greets three times
repeat(3) {
    println("Hello")
}

// Greets with an index
repeat(3) { index ->
    println("Hello with index $index")
}

```

Practice Time

```

import kotlin.random.Random
fun main(args: Array<String>) {
    val str = "*".repeat(10) // *****
    println(str)
    // Repeat an action 10 times
    repeat (10) { index ->
        println("${Random.nextInt(7)} index: $index")
    }
    feedTheFish()
}

fun feedTheFish() {
    val day = randomDay()
    val food = "pellets"
    println("Today is $day and the fish eat $food")
}

fun randomDay(): String {
    val week = listOf(
        "Monday",
        "Tuesday",

```



```

        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    )
    return week[Random.nextInt(7)]
}

```

Practice Time

*/*Create a program with a function that returns a fortune cookie message that*

Create a main() function.

From the main() function, call a function, getFortuneCookie(), that returns

Create a getFortuneCookie() function that takes no arguments and returns a String

In the body of getFortuneCookie(), create a list of fortunes. Here are some

"You will have a great day!"

"Things will go well for you today."

"Enjoy a wonderful day of success."

"Be humble and all will turn out well."

"Today is a good day for exercising restraint."

"Take it easy and enjoy life!"

"Treasure your friends because they are your greatest fortune."

Below the list, print: "Enter your birthday: "

Hint: Use print(), not println()

Create a variable, birthday.

Read the user's input from the standard input and assign it to birthday. If

Hint: Use readLine() to read a line of input (completed with Enter) as a String

Hint: In Kotlin, you can use toIntOrNull() to convert a number as a String to an Int

Hint: Check for null using the ? operator and use the ?: operator to handle

Divide the birthday by the number of fortunes, and use the remainder as the

Return the fortune.

In main(), print: "Your fortune is: ", followed by the fortune string.

Extra practice:

Use a for loop to run the program 10 times, or until the "Take it easy" fortune

// Solution Code

```

fun main(args: Array<String>) {
    println("\nYour fortune is: ${getFortuneCookie()}")
}

```

```

fun getFortuneCookie() : String {
    val fortunes = listOf( "You will have a great day!",
        "Things will go well for you today.",
        "Enjoy a wonderful day of success.",

```

```

        "Be humble and all will turn out well.",
        "Today is a good day for exercising restraint.",
        "Take it easy and enjoy life!",
        "Treasure your friends, because they are your greatest fortune.")
print("\nEnter your birthday: ")
val birthday = readLine()?.toIntOrNull() ?: 1
return fortunes[birthday.rem(fortunes.size)]
}

// Extra Practice
fun main(args: Array<String>) {
    var fortune: String
    for (i in 1..10) {
        fortune = getFortuneCookie()
        println("\nYour fortune is: $fortune")
        if (fortune.contains("Take it easy")) break
    }
}

// My Solution
import kotlin.random.Random
fun main(args: Array<String>) {
    // OR -> for (i in 1..10) { ... }
    repeat (10) {
        val fortune = getFortuneCookie()
        println("Your fortune is: ${fortune.first}")
        if (fortune.second == 5) {
            return
        }
    }
}

fun getFortuneCookie(): Pair<String, Int> {
    val fortunes = listOf(
        "You will have a great day!",
        "Things will go well for you today.",
        "Enjoy a wonderful day of success.",
        "Be humble and all will turn out well.",
        "Today is a good day for exercising restraint.",
        "Take it easy and enjoy life!",
        "Treasure your friends because they are your greatest fortune."
    )
    print("Enter your birthday: ")
    var birthday: String = readLine() ?: "1"
    val selectedFortuneIndex = birthday.toInt().rem(fortunes.size)
    return Pair(fortunes[ selectedFortuneIndex ], selectedFortuneIndex)
}

```

```

import java.time.MonthDay
import kotlin.random.Random

fun main(args: Array<String>) {
    repeat (10) { index ->
        feedTheFish()
    }
}

fun fishFood(day: String): String {
    return when (day) {
        "Monday" -> "flakes"
        "Tuesday"-> "redworms"
        "Wednesday" -> "granules"
        "Thursday" -> "mosquitoes"
        "Friday" -> "plankton"
        "Saturday" -> "lettuce"
        else -> "fasting"
    }
}

fun feedTheFish() {
    val day = randomDay()
    val food = fishFood(day)
    println("Today is $day and the fish eat $food")
}

fun randomDay(): String {
    val week = listOf(
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    )
    return week[Random.nextInt(7)]
}

```

Practice Time

```

// Solution Code
fun getBirthday(): Int {
    print("\nEnter your birthday: ")
    return readLine()?.toIntOrNull() ?: 1
}

```

```

fun getFortune(birthday: Int): String {
    val fortunes = listOf("You will have a great day!",
        "Things will go well for you today.",
        "Enjoy a wonderful day of success.",
        "Be humble and all will turn out well.",
        "Today is a good day for exercising restraint.",
        "Take it easy and enjoy life!",
        "Treasure your friends, because they are your greatest fortune."
    )
    val index = when (birthday) {
        in 1..7 -> 4
        28, 31 -> 2
        else -> birthday.rem(fortunes.size)
    }
    return fortunes[index]
}

```

// My Code

```

import kotlin.random.Random
fun main() {
    for (i in 1..10) {
        val birthday = getBirthday()
        val fortune: Pair<String, Int> = getFortuneCookie(birthday)
        println("Your fortune is: ${fortune.first}")
        if (fortune.second == 5) {
            break
        }
    }
}

fun getFortuneCookie(birthday: Int): Pair<String, Int> {
    val fortunes = listOf(
        "You will have a great day!",
        "Things will go well for you today.",
        "Enjoy a wonderful day of success.",
        "Be humble and all will turn out well.",
        "Today is a good day for exercising restraint.",
        "Take it easy and enjoy life!",
        "Treasure your friends because they are your greatest fortune."
    )
    val index = when (birthday) {
        in 1..10 -> Random.nextInt(3)
        28, 34 -> Random.nextInt(3, 6) // Means if it is 28 or 34
        else -> birthday.rem(fortunes.size)
    }
    return Pair(fortunes[index], index)
}

```

```

fun getBirthday(): Int {
    print("Enter your birthday: ")
}

```

```
        return readLine()?.toIntOrNull() ?: 1  
    }
```

Parameters

```

// Parameters kotlin can have a default value, this means when you call a fu
// If the value is missing, the default value is used
fun main(args: Array<String>) {
    swim()
    swim("Slow") // Specify the default argument positionally
    swim(speed = "Slow") // Or Specify the argument by name
}

fun swim(speed: String = "fast") {
    println("swimming $speed")
}

// You can mix default and positional arguments
fun main(args: Array<String>) {
    swim(5)
    swim(5, "Slow")
    swim(5, speed = "Slow")
    swim(time = 5, speed = "Slow")
}

fun swim(time: Int, speed: String = "fast") {
    println("swimming $speed")
}

// It is the best practice to put arguments without defaults first
// And then the ones with the defaults afterwards
shouldChangeWater(day, 20, 50)
shouldChangeWater(day)
shouldChangeWater(day, dirty = 50)

// Wrong example
shouldChangeWaterWRONG("Monday") // Error!
// We have to specify that Monday is the day
shouldChangeWaterWRONG(day = "Monday")

fun shouldChangeWater(day: String, temperature: Int = 22, dirty: Int = 20) {
}

// You can define a function where the default variables are listed first or
// mixed in others, but this easily leads to mistakes
// If you forget to list all arguments by name
fun shouldChangeWaterWRONG(temperature: Int = 22, dirty: Int = 20, day: Stri
}

```

Practice Time | Fit More Fish

```

/*Create a function that checks if we can add another fish into a tank that

```

How many fish in a tank?

The most widely known rule for stocking a tank is the one-inch-per-fish-per-

Typically, a tank with decorations can contain a total length of fish (in in

For example:

A 10 gallon tank with decorations can hold up to 8 inches of fish, for exampl

A 20 gallon tank without decorations can hold up to 20 inches of fish, for e
fitMoreFish function

Create a function that takes these arguments:

tankSize (in gallons)

currentFish (a list of Ints representing the length of each fish currently i

fishSize (the length of the new fish we want to add to the tank)

hasDecorations (true if the the tank has decorations, false if not)

You can assume that typically a tank has decorations, and that a typical fis

Output

Make sure you test your code against the following calls, and that you get t

canAddFish(10.0, listOf(3,3,3)) ---> false

canAddFish(8.0, listOf(2,2,2), hasDecorations = false) ---> true

canAddFish(9.0, listOf(1,1,3), 3) ---> false

canAddFish(10.0, listOf(), 7, true) ---> true

Things to think about

Again, there are so many ways you can do this, this is one of them:

```
fun canAddFish(tankSize: Double, currentFish: List<Int>, fishSize: Int = 2,  
return (tankSize * if (hasDecorations) 0.8 else 1.0) >= (currentFish.sum() +  
})
```

Notice how you can use the .sum() function in the list? This is a way to add

// Solution Code

```
fun main(args: Array<String>) {  
    println(canAddFish(10, listOf(3,3,3))) // ---> false  
    println(canAddFish(8, listOf(2,2,2), hasDecorations = false)) // ---> tr  
    println(canAddFish(9, listOf(1,1,3), 3)) // ---> false  
    println(canAddFish(10, listOf(), 7, true)) // ---> true  
}
```

// Alternative Shorter Solution

```
fun canAddFishAlternative(tankSize: Double, currentFish: List<Int>, fishSize  
    return (tankSize * if (hasDecorations) 0.8 else 1.0) >= (currentFish.sum  
})
```

```
fun canAddFish(tankSize: Int, currentFish: List<Int>, fishSize: Int = 2, has  
    var availableTankSize = tankSize
```

```

    // Without decorations, Total length of fish <= 100% of the tank size
    if (!hasDecorations) {
        availableTankSize -= currentFish.sum()
    }
    // With decorations, Total length of fish <= 80% of the tank size
    else {
        availableTankSize = availableTankSize.times(4).div(5)
        availableTankSize -= currentFish.sum()
    }
    println("tankSize: $tankSize, sum: ${currentFish.sum()}, availableTankSi
    return fishSize <= availableTankSize
}

```

Practice Time

*/*Create a program that suggests an activity based on various parameters.*

Start in a new file with a main function.

From main(), create a function, whatShouldIDoToday().

Let the function have three parameters.

mood: a required string parameter

weather: a string parameter that defaults to "sunny"

temperature: an Integer parameter that defaults to 24 (Celsius).

Use a when construct to return some activities based on combinations of conc

mood == "happy" && weather == "Sunny" -> "go for a walk"

else -> "Stay home and read."

Copy/paste your finished function into REPL, and call it with combinations c

whatShouldIDoToday("sad")

> Stay home and read.

Note: Keep your work as you will do more with this code in the next practice

```

fun main(args: Array<String>) {
    println(whatShouldIDoToday("happy"))
}

```

```

fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature:
    return when {
        mood == "happy" && weather == "Sunny" -> "go for a walk"
        else -> "Stay home and read."
    }
}

```

// My Code

```

fun main(args: Array<String>) {
    println(whatShouldIDoToday("sad"))
}

```

```

fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature:

```



```

        return when {
            mood == "happy" && weather == "Sunny" -> "go for a walk"
            else -> "Stay home and read."
        }
    }
}

```

```

// Return type can be inferred from the function
// The name of the function give a hint to the reader about the expected val
fun isTooHot(temperature: Int) = temperature > 30
fun isDirty(dirty: Int) = dirty > 30
fun isSunday(day: String): Boolean = day == "Sunday"

```

```

fun shouldChangeWater(day: String, temperature: Int = 22, dirty: Int = 20):

```

```

// Way 3 with one line function syntax
return when {
    isTooHot(temperature) -> true
    isDirty(dirty) -> true
    isSunday(day) -> true
    else -> false
}

```

```

// Way 2
//     val isTooHot = temperature > 30
//     val isDirty = dirty > 30
//     val isSunday = day == "Sunday"
//     return when {
//         isTooHot -> true
//         isDirty -> true
//         isSunday -> true
//         else -> false
//     }

```

```

// Way 1
//     return when {
//         temperature > 30 -> true
//         dirty > 30 -> true
//         day == "Sunday" -> true
//         else -> false
//     }

```

```

// Sometimes you might be tempted to use expensive functions to initialize c
// Examples of expensive operations include reading files or allocating a lc
// BE CAREFUL WITH THIS, They can affect the performance of your code quite
// Because Default parameters are evaluated at call time by Kotlin

```

```

fun getDirtySensorReading() = 20
fun shouldChangeWater(dirty: Int = getDirtySensorReading()): Boolean {
    // ...
}

```

```

////////////////////////////////////
fun main(args: Array<String>) {
    aquariumStatusReport()
    aquariumStatusReport("sfg")
}
/*
 * Every time you call aquariumStatusReport() without passing a value for the
 * a new aquarium will be made which is costly
 * */
fun makeNewAquarium() = println("Building a new aquarium.....")
fun aquariumStatusReport(aquarium: Any = makeNewAquarium()) {
    // Any can hold any type of object
}
////////////////////////////////////

// In kotlin, for and while loops are not expressions
val noValue = for (x in 1..2) {} // For is not an expression, and only expressions
val notThisEither = while (false) {} // For is not an expression, and only expressions

```

Practice Time

```

fun main(args: Array<String>) {
    println(whatShouldIDoToday("happy", "sunny"))
    println(whatShouldIDoToday("sad"))
    print("How do you feel?")
    println(whatShouldIDoToday(readLine()!!))
}

fun isVeryHot (temperature: Int) = temperature > 35

fun isSadRainyCold (mood: String, weather: String, temperature: Int) =
    mood == "sad" && weather == "rainy" && temperature == 0

fun isHappySunny (mood: String, weather: String) = mood == "happy" && weather == "sunny"

fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature: Int) =
    return when {
        isVeryHot(temperature) -> "go swimming"
        isSadRainyCold(mood, weather, temperature) -> "stay in bed"
        isHappySunny(mood, weather) -> "go for a walk"
        else -> "Stay home and read."
    }
}

// My Code
fun main() {
    print("Enter mood: ")
    val mood = readLine().orEmpty() // OR -> readLine() ?: ""
}

```

```

        // Double Bang operator does the following line;
        // if (readLine() != null) readLine() else throw    NullPointerException
        println(whatShouldIDoToday(mood))
    }

    fun shouldWalk(mood: String, weather: String) = mood == "happy" && weather =
    fun shouldSleep(mood: String, weather: String, temperature: Int) = mood == "
    fun shouldSwim(mood: String, weather: String, temperature: Int) = temperatur

    fun whatShouldIDoToday(mood: String, weather: String = "sunny", temperature:
        return when {
            shouldWalk(mood, weather) -> "go for a walk"
            shouldSleep(mood, weather, temperature) -> "stay in bed"
            shouldSwim(mood, weather, temperature) -> "go swimming"
            else -> "Stay home and read."
        }
    }
}

```

Repeat and While

```

// Using repeat:
fun main(args: Array<String>) {
    var fortune: String = ""
    repeat (10) {
        fortune = getFortune(getBirthday())
        println("\nYour fortune is: $fortune")
        if (fortune.contains("Take it easy")) break;
    }
}

// Using a while loop:
fun main(args: Array<String>) {
    var fortune: String = ""
    while (!fortune.contains("Take it easy")) {
        fortune = getFortune(getBirthday())
        println("\nYour fortune is: $fortune")
    }
}

```

Filters

```

val list = listOf("abc", "ghf", "aaa", "tur")
println(list.filter { it[0] == 'a' }) // Outputs: [abc, aaa]
// Returns the elements that satisfy the condition it[0] == 'a'
// In Kotlin, 'c' -> characters, "string" -> string
// Strings and chars are not interchangeable// They are different things!

```

```

// For example list.filter { it[0] == "a" } // ERROR// because we do char

// Filter is a standard library function on list in kotlin

// The difference between EAGER and LAZY
// -> AN EAGER algorithm executes immediately and returns a result.
// -> A LAZY algorithm defers computation until it is necessary to execute a

// By default, filter analyst is EAGER that means everytime you call filter,

// EAGER example
fun main() {
    val list = listOf("rock", "pagoda", "plastic", "tur")
    // Decoration EAGER here// We'll hold a new list
    // containing strings that starts with 'p'
    val decorations = list.filter { it[0] == 'p' }
    println(decorations)
}

// If you want LAZY Behaviour, you can use SEQUENCES// A sequence is a colle

// When you return the filter results as a sequence, our filtered variable w

// Ehenever you access elements of the sequence, the filter is applied and t

// Of course, if we want to turn our sequence back into the list, we can cal
fun main() {
    val list = listOf("rock", "pagoda", "plastic", "tur")
    // Apply filter LAZILY
    val decorations = list.asSequence().filter { it[0] == 'p' }
    println(decorations) // kotlin.sequences.FilteringSequence@1fb3ebeb
    println(decorations.toList()) // Ignite the filter!: [pagoda, plastic]
}

// Let's use the function map and tell it to print every item, since it's la

// Let's use the function map and tell it to print every item
val lazyMap = decorations.asSequence().map {
    println("map $it")
    it
}
println(lazyMap) // kotlin.sequences.TransformingSequence@53d8d10a

// When I take the first element however, you can see that the map operator
fun main() {
    val list = listOf("rock", "pagoda", "plastic", "tur")
    // Apply filter LAZILY
    val decorations = list.asSequence().filter { it[0] == 'p' }
    // Let's use the function map and tell it to print every item
    val lazyMap = decorations.asSequence().map {

```

```

        println("map $it") // map pagoda
        it
    }
    println(lazyMap) // kotlin.sequences.TransformingSequence@53d8d10a
    println("first: ${lazyMap.first()}") // first: pagoda
}

// Of course taking the full list will iterate over all the values
fun main() {
    val list = listOf("rock", "pagoda", "plastic", "tur")
    // Apply filter LAZILY
    val decorations = list.asSequence().filter { it[0] == 'p' }
    // Let's use the function map and tell it to print every item
    val lazyMap = decorations.asSequence().map {
        println("map $it")
        it
    }
    println("all: ${lazyMap.toList()}")
}

// map pagoda
// map plastic
// all: [pagoda, plastic]

```

Practice Time

*/*You can do the following filter exercise in REPL.*

Create a list of spices, as follows:

```
val spices = listOf("curry", "pepper", "cayenne", "ginger", "red curry", "gr
```

Create a filter that gets all the curries and sorts them by string length.

Hint: After you type the dot (.), IntelliJ will give you a list of functions

Filter the list of spices to return all the spices that start with 'c' and e

Take the first three elements of the list and return the ones that start wit

Note: We will be able to do a lot more interesting stuff with filters after

```

fun main() {
    val spices = listOf("curry", "pepper", "cayenne", "ginger", "red curry",
    // 2
    println(spices.filter { s -> s.contains("curry") }.sortedBy { s -> s.ler
    // 3
    println(spices.filter { s -> s.startsWith('c') && s.endsWith('e') })
    println(spices.filter { s -> s.first() == 'c' && s.last() == 'e' })
    println(spices.filter { s -> s[0] == 'c' && s[s.length - 1] == 'e' })
}

```

```

// 4
println(spices.take(3).filter { s -> s.first() == 'c' })

// OR WE CAN USE "it"
val spices = listOf("curry", "pepper", "cayenne", "ginger", "red curry",
// 2
println(spices.filter { it.contains("curry") }.sortedBy { it.length })
// 3
println(spices.filter { it.startsWith('c') && it.endsWith('e') })
println(spices.filter { it.first() == 'c' && it.last() == 'e' })
println(spices.filter { it[0] == 'c' && it[it.length - 1] == 'e' })
// 4
println(spices.take(3).filter { it.first() == 'c' })
}

// Solution Code
// Sorting curries by string length
spices.filter { it.contains("curry") }.sortedBy { it.length }

// Filtering by those that start with 'c' and end with 'e'
spices.filter{it.startsWith('c')}.filter{it.endsWith('e')}
> [cayenne]
// OR
spices.filter { {it.startsWith('c') && it.endsWith('e') }
> [cayenne]

// Filtering the first 3 items by 'c'
spices.take(3).filter{it.startsWith('c')}
> [curry, cayenne]

```

Kotlin Labmdas

```

// Lambda functions are used when you need a function FOR A SHORT PERIOD OF
// A LAMBDA is an expression that makes a function, instead of declaring a r

fun main() {
    // Lambda function
    { println("Hello") }()
}

// We can declera a variable called swim and assign it to a lambda
// Lambda function, If we put "()" this runs/calls the lambda function
{ println("Hello") }() // Hello

// We can also say: run { println("Hello") } // Hello
// We can declera a variable called swim and assign it to a lambda
var swimDontRun = { println("swim") } // swim
var swimRunDirectly = { println("swim") }() // swim

```

```

var swimRunDirectly2 = run { println("swim") } // swim

// We can call variable just like a regular function
swimDontRun() // Output: swim

// Lambdas can take arguments just like named functions
// Lambda arguments go on the left hand side of what's called a function arrow
// The body of the lambda goes after the function arrow
fun main() {
    var dirty = 20
    val waterFilter = { dirty: Int -> dirty / 2 }
    println(waterFilter(dirty)) // 10
    // waterFilter can be any function that takes an int and returns an int
    val waterFilter2: (Int) -> Int = { abc: Int -> abc + 2 }
    // We don't have to specify the type of the lambda argument anymore
    val waterFilter3: (Int) -> Int = { abc -> abc + 2 }
}

```

Higher-Order Functions

```

// The real power of lambda happens when we make higher-order functions
// A higher-order function is just any function that takes a function as the
// Kotlin prefers function arguments to be the last parameter
// Higher-order function that takes function as an argument
fun updateDirty(dirty: Int, operation: (Int) -> Int): Int {
    return operation(dirty)
}

/*
 * When you combine higher-order functions with lambdas
 * Kotlin has a special syntax
 * it's called the last parameter called syntax
 * */
fun dirtyProcessor() {
    dirty = updateDirty(dirty, waterFilter)
    println("1: $dirty")
    // since feedFish is a named function and not a lambda
    // you'll need to use a double colon to pass it
    // This way Kotlin know you're not trying to call feedFish
    // and it will let you pass a REFERENCE
    // So here we don't call the function but we pass it to another function
    // that function will run the function passed it to
    dirty = updateDirty(dirty, ::feedFish)
    // " :: " means, it creates a member reference or a class reference.
    println("2: $dirty")
    // Above method is similar as the following
    dirty = feedFish(dirty)
    println("22: $dirty")
}

```

```

// Here we call updateDirty again, but this time
// we pass a lambda as an argument for the parameter operation
/*
 * What's really interesting here, a lambda is an argument to updateDirty
 * but since we're passing it as the last parameter
 * we don't have to put it inside the function parentheses
 * */
dirty = updateDirty(dirty) { dirty ->
    dirty + 50
}
/*
 * To really show you what is going on,
 * you can put the parentheses back in, here you can see we're just
 * passing the lambda as an argument updateDirty
 * */
dirty = updateDirty(dirty, { dirty ->
    dirty + 50
})
/*
 * Using this syntax we can define functions that look like they're built
 * Actually, we've already used a few higher-order functions from the sta
 * */
val list = listOf(1, 2, 3)
list.filter {
    it == 2
}
/*
 * The filter function we used in the last section, takes a lambda and
 * uses it to filter a list,
 * repeat is also just a function that takes a repeat count and a lambda
 * */
}

```

Practice Time

```

// What is the difference between?
val random1 = random()
val random2 = { random() }
// Try it out in REPL or a file:
> The second will generate a random number every time random2 is accessed.
// ANSWER
// random1 has a value assigned at compile time, and the value never changes
// random2 has a lambda assigned at compile time, and the lambda is executed

```

Practice Time | Lambdas


```
// Create a lambda and assign it to rollDice, which returns a dice roll (num  
// Extend the lambda to take an argument indicating the number of sides of 1  
// If you haven't done so, fix the lambda to return 0 if the number of sides  
// Create a new variable, rollDice2, for this same lambda using the functor
```

```
// Solution Code
```

```
val rollDice = { Random().nextInt(12) + 1 }  
val rollDice = { sides: Int ->  
    Random().nextInt(sides) + 1  
}  
val rollDice0 = { sides: Int ->  
    if (sides == 0) 0  
    else Random().nextInt(sides) + 1  
}  
val rollDice2: (Int) -> Int = { sides ->  
    if (sides == 0) 0  
    else Random().nextInt(sides) + 1  
}
```

```
// My Code
```

```
import kotlin.random.Random  
fun main() {  
    val rollDice6Sides = { Random.nextInt(12) + 1 }  
    val rollDice = { sides: Int ->  
        if (sides == 0) 0  
        else Random.nextInt(sides) + 1  
    }  
    repeat(10) {  
        println("${rollDice(0)}")  
    }  
    val rollDice2: (Int) -> Int = { sides: Int ->  
        if (sides == 0) 0  
        else Random.nextInt(sides) + 1  
    }  
}
```

Practice Time | Extra Questions

```
// Why would you want to use the function type notation instead of just the  
  
// Create a function gamePlay() that takes a roll of the dice as an argument  
  
// Pass your rollDice2 function as an argument to gamePlay() to generate a c  
  
// Solution Explanation  
// Function type notation is MORE READABLE, which REDUCES ERRORS, clearly st
```

```

// Solution Code
gamePlay(rollDice2(4))
    fun gamePlay(diceRoll: Int){
        // do something with the dice roll
        println(diceRoll)
    }

// My Code
import kotlin.random.Random
fun main() {
    // Why would you want to use the function type notation instead of just
    // -> We might want to know what type we pass in the function
    // this will reduce errors related to parameters
    val dice = { sides: Int ->
        Random.nextInt(sides) + 1
    }
    val rollDice2: (Int) -> Int = { sides: Int ->
        if (sides == 0) 0
        else Random.nextInt(sides) + 1
    }
    gamePlay(dice, rollDice2)
}

fun gamePlay(dice: (Int) -> Int, dice2: (Int) -> Int) {
    println(dice2(6))
    println(dice(6))
}

```

Lesson 4 | Classes

```

/*
Classes are blue prints for objects

-> Class - Object Blueprint
(like an Aquarium Plan)

-> Objects are instances of classes that is the actual aquarium
(Actual Aquarium)

-> Properties are characteristics of classes such as the length
(Aquarium width, height)

-> Methods are the functionality of the class, class function, what the object
( fillWithWater() )

-> Interfaces are a specification that a class can implement
(Specification a class can implement ( Clean )), for example, cleaning is common
*/

```

Practice Time

```

/*Earlier, we created and filtered a list of spices. Spices are much better
To recap, let's make a simple Spice class. It doesn't do much, but it will serve
Create class, SimpleSpice.
Let the class be a property with a String for the name of the spice, and a Set
Set the name to curry and the spiciness to mild.
Using a string for spiciness is nice for users, but not useful for calculations
Create an instance of SimpleSpice and print out its name and heat.*/

```

```

// My Code
class Spice {
    var name: String = "curry"
    var spiciness: String = "mild"
    fun heat(): Int {
        return when (spiciness) {
            "mild" -> 5
            else -> 6
        }
    }
}

fun main() {
    // Create spice class instance
    val mySpice = Spice()
    println("name: ${mySpice.name}, heat: ${mySpice.heat()}")
}

```

```
}
```

```
// Solution Code  
class SimpleSpice() {  
    val name = "curry"  
    val spiciness = "mild"  
    val heat: Int  
        get() {return 5 }  
}  
// In main  
val simpleSpice = SimpleSpice()  
println("${simpleSpice.name} ${simpleSpice.heat}")
```

Package Visibility

```
// In kotlin everything is public by default, that means all of your variabl  
  
// Visibility modifiers in Kotlin  
-> public - Default. Everywhere  
-> private - File  
-> internal - Module  
  
// At the package level, if you don't specify any visibility modifier,  
// " public " is used by default  
// Which means that your devlarations will be visible everywhere  
// A module is a set of Kotlin files compiled together, when it's internal w  
  
// For members declared inside the class, again by default they are public
```

Class Visibility

```
// Public means that any client who sees the class can also see it's public  
// Private means members are only visible inside the class, importantly subc  
// Protected means the same as private but members are also visible to subcl  
// Class members can have a visibility of internal as well
```

Class Examples

```
// Sample Code  
// FILE: Main  
package Aquarium  
fun main() {  
    // Create spice class instance  
    // val mySpice = Spice()
```

```

        // println("name: ${mySpice.name}, heat: ${mySpice.heat}")
        buildAquarium()
    }

    // If you mark a declaration private,
    // it will only be visible the inside the file containing declaration
    // Since we're only going to use buildAquarium inside this file
    // We can make it private
    // If you mark buildAquarium " internal " it is visible anywhere in the same
    private fun buildAquarium() {
        // Creates new instance of Aquarium by calling its constructor
        val myAquarium = Aquarium()
        // Under the hood, Kotlin actually made a getter for all three properties
        // Even though we did not write any code
        println("Length: ${myAquarium.length}" +
            " Width: ${myAquarium.width}" +
            " Height: ${myAquarium.height}")

        // We don't have to change " myAquarium " to a var because, we're not changing it
        // It's the same object we're modifying its properties
        myAquarium.height = 80
        println("New Height: ${myAquarium.height} cm")
        println("Volume: ${myAquarium.volume} liter")
    }

```

```

// FILE: Aquarium
package Aquarium
class Aquarium {
    var length = 100
    var width = 20
    var height = 40

    // Sample Getter/Setter Syntax
    var volume: Int
        get() {
            return width * height * length / 1000
        }
        set(value) {
            height = (value * 1000) / (width * length)
        }

    // Alternative one liner Getter/Setter Syntax
    var volume2: Int
        get() = width * height * length / 1000
        // By convention, the name of the setter parameter is " value "
        private set(value) { height = (value * 1000) / (width * length) }
        // If we didn't want anyone outside the class to be able to use
        // the setter, we could make it private
        // private set(value) { height = (value * 1000) / (width * length) }
        // In kotlin everything is public by default

```

```

    // fun volume(): Int {
    //         return width * height * length / 1000
    //     }
    //
    // // Alternative, one liner
    // fun volume1() = width * height * length / 1000
}

// Sample Code For Classes
package Aquarium

// Constructor
class Test(id: Int, name: String, val testVal: String) {
    /*
     * The primary constructor cannot contain any code.
     * Initialization code can be placed in initializer blocks,
     * which are prefixed with the init keyword.
     * */
    // As the name says,
    // "also" expressions does some additional processing on the object it w
    // Unlike let, it returns the original object instead of any new return
    var firstProperty = "First property: $name".also(::println)
    // Alternative of " also "
    // val b = "SDads: $name".also { println(it) }

    init {
        // ....
        println("First this block will be executed")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)
    init {
        println("Second initializer block that prints ${name.length}")
    }
}

// The class can also declare secondary constructors, which are prefixed wit
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    // Secondary constructor
    constructor(owner: Person) {
        // Add this pet to he list of its owner's
        owner.pets.add(this)
    }
}

/*
 * A class in Kotlin can have a primary constructor and one or more secondary

```

** The primary constructor is part of the class header:
* it goes after the class name (and optional type parameters).
* */*

```
class Person constructor(firstName: String) {  
  
}
```

*// If the primary constructor does not have any annotations or visibility modifiers
// the constructor keyword can be omitted:*

```
class Person2 (firstName: String) {  
  
}
```

```
fun main() {  
    val test = Test(24, "Melo")  
    println("Age: ${test.firstProperty}, Name: ${test.secondProperty}")  
    test.firstProperty = "123"  
    println("${test.firstProperty}")  
    test.testVal  
}
```

// Kotlin has a concise syntax for declaring properties and initializing the class

```
class Person(val firstName: String, val lastName: String, var age: Int)
```

// Such declarations can also include default values of the class properties

```
class Person(val firstName: String, val lastName: String, var isEmployed: Boolean)
```

// You can use a trailing comma when you declare class properties:

```
class Person(  
    val firstName: String,  
    val lastName: String,  
    var age: Int, // trailing comma  
) { /*...*/ }
```

/
VISIBILITY*

*PACKAGE:
public - default. Everywhere
private - file
internal - module*

*CLASS:
sealed - only subclass in same file*

*INSIDE CLASS:
public - default. Everywhere.
private - inside class, not subclasses
protected - inside class and subclasses*

```
internal - module
*/
```

Practice Time

Sample Code

```
package Aquarium

// Constructor
class Test(id: Int, name: String) {
  /*
   * The primary constructor cannot contain any code.
   * Initialization code can be placed in initializer blocks,
   * which are prefixed with the init keyword.
   */
  // As the name says,
  // "also" expressions does some additional processing on the object it was i
  // Unlike let, it returns the original object instead of any new return data
  var firstProperty = "First property: $name".also(::println)
  // Alternative of " also "
  // val b = "SDads: $name".also { println(it) }

  init {
    // ....
    println("First this block will be executed")
  }

  val secondProperty = "Second property: ${name.length}".also(::println)

  init {
    println("Second initializer block that prints ${name.length}")
  }

  /*
   * Accessing the Backing Field
   * Every property we define is backed by a field
   * that can only be accessed within its get() and set() methods
   * using the special field keyword.
   * The field keyword is used to access or modify the property's value.
   * This allows us to define custom logic within the get() and set() methods
   */
  var rating: Int = 5
  get() {
    if (field < 5) {
```



```

        println("Warning This is a Terrible Book!")
    }
    return field
}
set(value) {
    field = when {
        value > 10 -> 10
        value < 0 -> 0
        else -> value
    }
}
}

// Getters and setters are auto-generated in Kotlin.
// In Kotlin, a property doesn't require explicit getter or setter methods:
var author: String = "Frank Herbert"
    // Redundant getter !
    get() {
        return field
    }

    // Redundant setter !
    set(value) {
        field = value
    }

/*
* Defining a custom getter or setter allows us
* to perform any number of useful operations like input validation,
* logging, or data transformations.
* By adding this business logic directly to the getter or setter,
* we ensure that it's always performed when the property is accessed.
* Try to avoid or minimize side-effects
* in the getter and setter methods as much as possible.
* It makes our code harder to understand.
* */

/*
* If we want to be able to modify a property's value,
* we mark it with the var keyword.
* If we want an immutable property, we mark it with a val keyword.
* The main difference is that val properties can't have setters.
* */
val isWorthReading: Boolean get() = this.rating > 5
// set(value) { // A 'val'-property cannot have a setter!
// // ERROR
// }
// In this sense, the property acts as a method when using a custom getter.

/*
* Now any consumers of the book class can read the inventory property
* but only the Book class can modify it.
* */

```

```

var inventory: Int = 0
private set
/*
* Note that the default visibility for properties is public.
* The getter will always have the same visibility as the property itself.
* For example, if the property is private, the getter is private.
* */

// Backing Fields
/*
In Kotlin, a field is only used as a part of a property
to hold its value in memory. Fields can not be declared directly.
However, when a property needs a backing field, Kotlin provides it automatic
This backing field can be referenced in the accessors using the "field" identifier
* */
var counter = 0 // the initializer assigns the backing field directly
    set(value) {
        if (value >= 0)
            field = value
        // counter = value // ERROR StackOverflow: Using actual name 'counter'
    }

// For example, in the following case there will be no backing field:
// val isEmpty: Boolean
//     get() = this.size == 0
}

/*
* A class in Kotlin can have a primary constructor and one or more secondary
* The primary constructor is part of the class header:
* it goes after the class name (and optional type parameters).
* */
class PersonTest constructor(firstName: String) {

}

// If the primary constructor does not have any annotations or visibility modifiers
// the constructor keyword can be omitted:
class Person2 (firstName: String) {

}

// The class can also declare secondary constructors, which are prefixed with 'init'
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    // Secondary constructor
    constructor(owner: Person) {
        // Add this pet to the list of its owner's
        owner.pets.add(this)
    }
}

```

```

    }
}

fun main() {
    val test = Test(24, "Melo")
    println("Age: ${test.firstProperty}, Name: ${test.secondProperty}")
    test.firstProperty = "123"
    println("First Author: ${test.author}")
    test.author = "Melo Genesis"
    println("Second Author: ${test.author}")
}

```

Practice Time

*/*Earlier, we created and filtered a list of spices. Spices are much better
To recap, let's make a simple Spice class. It doesn't do much, but it will s
Create class, SimpleSpice.
Let the class be a property with a String for the name of the spice, and a S
Set the name to curry and the spiciness to mild.
Using a string for spiciness is nice for users, but not useful for calculati
Create an instance of SimpleSpice and print out its name and heat.*/*

```

class SimpleSpice(){
    val name = "curry"
    val spiciness = "mild"
    val heat: Int
        get() {return 5 }
}
val simpleSpice = SimpleSpice()
println("${simpleSpice.name} ${simpleSpice.heat}")

```

Sample Codes Continued

```

//////////////////////////////////// Sample Code //////////////////////////////////////
package Aquarium
class Aquarium {
    var length = 100
    var width = 20
    var height = 40

    // Sample Getter/Setter Syntax
    var volume: Int
        get() {
            return width * height * length / 1000
        }
}

```

```

    }
    set(value) {
        height = (value * 1000) / (width * length)
    }

// Alternative one liner Getter/Setter Syntax
var volume2: Int
    get() = width * height * length / 1000
    // By convention, the name of the setter parameter is " value "
    private set(value) { height = (value * 1000) / (width * length) }
    // If we didn't want anyone outside the class to be able to use
    // the setter, we could make it private
    // private set(value) { height = (value * 1000) / (width * length) }
    // In kotlin everything is public by default

// fun volume(): Int {
//     return width * height * length / 1000
// }
//
// // Alternative, one liner
// fun volume1() = width * height * length / 1000
}

////////////////////////////////////

// With default parameters constructor overloading is not needed

////////////////////////////////////
// Solution Code
class Spice(val name: String, val spiciness: String = "mild") {

    private val heat: Int
    get() {
        return when (spiciness) {
            "mild" -> 1
            "medium" -> 3
            "spicy" -> 5
            "very spicy" -> 7
            "extremely spicy" -> 10
            else -> 0
        }
    }
}

val spices1 = listOf(
    Spice("curry", "mild"),
    Spice("pepper", "medium"),
    Spice("cayenne", "spicy"),
    Spice("ginger", "mild"),
    Spice("red curry", "medium"),

```

```

        Spice("green curry", "mild"),
        Spice("hot pepper", "extremely spicy")
    )

    val spice = Spice("cayenne", spiciness = "spicy")

    val spicelist = spices1.filter {it.heat < 5}

    fun makeSalt() = Spice("Salt")
    ///////////////////////////////////////////////////

    // Kotlin does not have a new keyword

    ///////////////////////////////////////////////////
    // My Code
    package Aquarium
    class Spice(val name: String, val spiciness: String = "mild") {
        val heat: Int
        get() {
            return when (spiciness) {
                "mild" -> 1
                "medium" -> 3
                "spicy" -> 5
                "very spicy" -> 7
                "extremely spicy" -> 10
                else -> 0
            }
        }
        init {
            println("Name. $name, Spiciness: $spiciness, Heat: $heat")
        }
    }

    fun makeSalt(): Spice {
        return Spice("Salt")
    }

    fun main() {
        val spices = listOf<Spice>(
            Spice("curry", "mild"),
            Spice("pepper", "medium"),
            Spice("cayenne", "spicy"),
            Spice("ginger", "mild"),
            Spice("red curry", "medium"),
            Spice("green curry", "mild"),
            Spice("hot pepper", "extremely spicy")
        )
        val spice = spices.filter {
            it.heat < 5
        }
    }

```

```

    fun makeSalt() = Spice("Salt")
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
package Aquarium
class Aquarium (var length: Int = 100, var width: Int = 20, var height: Int
    var volume: Int
        get() = width * height * length / 1000
        // By convention, the name of the setter parameter is " value "
        private set(value) { height = (value * 1000) / (width * length) }

    // The inferred data type is double
    var water = volume * 0.9

    // If we need to have another constructor than the default one for our class
    // For example, instead of specifying the dimensions when we create the class
    // we might want to specify the number of fish when we create an aquarium
    constructor(numberOfFish: Int): this() {
        val water = numberOfFish * 2000 // cm3
        val tank = water + water * 0.1
        height = (tank / (length * width)).toInt()
    }
    // Note that we can't mix constructor arguments, so we cannot create an
    // The arguments have to match exactly with one of the available constructors
}

```

Inheritance

```

package Aquarium
import kotlin.math.PI
// It doesn't say explicitly but this class actually inherits from the top level
/*
 * The first thing we have to do to be able to inherit from a class
 * is make the class " open ", by default classes are not subclassable
 * We have to explicitly allow it
 * */
open class Aquarium (var length: Int = 100, var width: Int = 20, var height: Int
// We could add "Any()" but it's not required and doesn't give anything extra
    open var volume: Int
        get() = width * height * length / 1000
        set(value) { height = (value * 1000) / (width * length) }
    // Private setters are not allowed for open properties!
    open var water = volume * 0.9

    constructor(numberOfFish: Int): this() {
        val water = numberOfFish * 2000 // cm3
        val tank = water + water * 0.1
    }
}

```

```

        height = (tank / (length * width)).toInt()
    }
}

```

// All classes in Kotlin have a common superclass Any, that is the default s

```
class Example // Implicitly inherits from Any
```

// Any has three methods: equals(), hashCode() and toString(). Thus, they ar

// By default, Kotlin classes are final: they can't be inherited. To make a

```
open class Base //Class is open for inheritance
```

// To declare an explicit supertype, place the type after a colon in the cla

```
open class Base(p: Int)
```

```
class Derived(p: Int) : Base(p)
```

// Overriding Methods

// Kotlin requires explicit modifiers for overridable members and overrides:

```

open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}

```

// Inheritance Explanation

*/**

** Let's say we want to have a different type of aquarium such as cylindrical*

** Tower tanks are a lot like regular aquariums*

** But they are also different in some ways*

** So we couldn't inherit a lot of stuff from our basic aquarium*

** and change the things that are different*

** Now, in the same file is okay, we can create a tower tank that inherits 1*

** We specify the inheritance or the parent class, after the colon*

** */*

```

class TowerTank(): Aquarium() {
    // We need to change how the volume is calculated
    // And we don't want to fill as much water into the tall tank
    // We are doing this by overriding the water property in tower tank
    override var water = volume * 0.8

```

// Members are not available for subclassing by default

// This is so we don't accidentally leak implementation details without

```

        override var volume: Int
        get() = (width * height * length / 1000 * PI).toInt()
        set(value) {
            height = (value * 1000) / (width*length)
        }
    }

    //////////////////////////////////////
open class Book(val title: String, val author: String) {
    private var currentPage = 1
    open fun readPage() {
        currentPage++
    }
}

class eBook(title: String, author: String, var format: String = "text") : Book {
    private var wordsRead = 0
    override fun readPage() {
        wordsRead = wordsRead + 250
    }
}

    //////////////////////////////////////

    //////////////////////////////////////
// My Code
package Aquarium
open class Book(val title: String, val author: String) {
    private var currentPage = 1
    open fun readPage() {
        currentPage++
    }
}

// Subclass
class eBook(title: String, author: String, var format: String = "text"): Book {
    var wordCount = 0
    override fun readPage() {
        wordCount += 250
    }
}
    //////////////////////////////////////

```

Interfaces

```

/* Different types of fish have lots in common, and they do similar things 1

For example;
All fish have a color and all fish have to eat

```


*So we want to make sure that all our fish that we create do that
Kotlin offers two ways of doing that*

- 1) Abstract Classes*
- 2) Interfaces*

Both are classes that cannot be instantiated on their own which means you can't

The difference is that ABSTRACT CLASSES HAVE CONSTRUCTORS while Interfaceses

A final thing you can do in Kotlin, when using classes that implement inter

Interface Examples & Explanations

```
////////// AquariumFish.kt Class//////////  
package Aquarium  
// Simple Abstract Class  
/*  
 * Because AquariumFish is abstract we can't make instances of AquariumFish c  
 * We need to provide sub classes that implement its missing functionality  
 * */  
abstract class AquariumFish {  
    abstract val color: String  
}  
  
// Two subclasses, we have to implement the abstract property color// otherw  
// it will leave us with errors// as following  
/*  
 * ERROR: Class 'Shark' is not abstract and does not implement  
 * abstract base class member public abstract val color: String  
 * defined in Aquarium.AquariumFish  
 * */  
// Now we can use it like any other class  
class Shark: AquariumFish(), FishAction {  
    override val color = "gray"  
    override fun eat() {  
        println("hunt and eat fish")  
    }  
}  
  
// Add a comma and then the FishAction interface without "()" and implement  
// You have to implement interface methods!  
class Plecostomus: AquariumFish(), FishAction {  
    override val color = "gold"  
    override fun eat() {  
        println("much on algae")  
    }  
}
```

```

    }
}

// Interface example, FishAction that defines an eat function
interface FishAction {
    fun eat()
}

//////////////////////////////// Main //////////////////////////////////
package Aquarium
fun main() {
    // Create Aquarium class instance
    buildAquarium()
    makeFish()
}

// If you mark a declaration private,
// it will only be visible inside the file containing declaration
// Since we're only going to use buildAquarium inside this file
// We can make it private
// If you mark buildAquarium " internal " it is visible anywhere in the same
private fun buildAquarium() {
    // Creates new instance of Aquarium by calling its constructor
    val myAquarium = Aquarium()
    // Under the hood, Kotlin actually made a getter for all three properties
    // Even though we did not write any code
    println(
        "Length: ${myAquarium.length}" +
        " Width: ${myAquarium.width}" +
        " Height: ${myAquarium.height}"
    )

    // We don't have to change " myAquarium " to a var because, we're not changing it
    // It's the same object we're modifying its properties
    myAquarium.height = 80
    println("New Height: ${myAquarium.height} cm")
    println("Volume: ${myAquarium.volume} liters")
    // To make this more readable, let's pass in name parameters
    val smallAquarium = Aquarium(length = 20, width = 15, height = 30)
    val smallAquarium2 = Aquarium(numberOfFish = 9)
    println("Small Aquarium: Volume: ${smallAquarium2.volume} " +
        "liters with length ${smallAquarium2.length} " +
        "width ${smallAquarium2.width} " +
        "height ${smallAquarium2.height}")
}

/*
* This function creates a shark and a pleco and prints out their colors
* */
fun makeFish() {

```

```

    val shark = Shark()
    val pleco = Plecostomus()

    println("Shark: ${shark.color} \n Pleco: ${pleco.color}")

    shark.eat()
    pleco.eat()
}

/*
 * When a fish gets the food, it eats it, we don't care what kind of fish it
 * as long as it can eat the food. "Eat" is defined in fish action, So every
 * to feed fish needs to implement fish action, we don't care about any other
 * As long as it implements fish action, we can use it.
 * Only fish that implement fish action can be passed into "feedFish"
 * This is a simplistic example but when you have a lot of classes
 * this can help you keep clearer and more organized
 * */
fun feedFish(fish: FishAction) {
    fish.eat()
}

////////////////////////////////////

////////////////////////////////////
package Aquarium
// Simple Abstract Class
/*
 * Because AquariumFish is abstract we can't make instances of AquariumFish c
 * We need to provide sub classes that implement its missing functionality
 * */
abstract class AquariumFish {
    abstract val color: String
}

// Two subclasses, we have to implement the abstract property color// otherwise
// it will leave us with errors// as following
/*
 * ERROR: Class 'Shark' is not abstract and does not implement
 * abstract base class member public abstract val color: String
 * defined in Aquarium.AquariumFish
 * */
// Now we can use it like any other class
class Shark: AquariumFish(), FishAction {
    override val color = "gray"
    override fun eat() {
        println("hunt and eat fish")
    }
}

// Add a comma and then the FishAction interface without "()" and implement

```

```

// You have to implement Interface methods!
class Plecostomus: AquariumFish(), FishAction {
    override val color = "gold"
    override fun eat() {
        println("much on algae")
    }
}

// Interface example, FishAction that defines an eat function
interface FishAction {
    fun eat()
}

/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////

Difference Between Abstract Classes And Interfaces
// There is really only one syntax difference in Kotlin between abstract cla
-> Abstract classes can have constructors and interfaces cannot

// Both abstract classes and interfaces can contain implementations of metho
// On interfaces we call them default implementations
// The big difference really is in when and how you use them

// Use an interface if you have a lot of methods and one or two default imple
interface AquariumAction {
    fun eat()
    fun jump()
    fun clean()
    fun catchFish()
    fun swim() {
        println("swim")
    }
}

// Use an abstract class anytime you can't complete a class
interface FishActionTest {
    fun eat()
}

abstract class AquariumFishTest: FishActionTest {
    abstract val color: String
    override fun eat() {
        println("yum")
    }
}

// Making all aquarium fish implement "FishActionTest", we can provide a def

// But really Kotlin provides us a better tool for this than abstract classe

```

```

// INTERFACE DELEGATION let's you add features to a class via composition
// Composition is when you use an instance of another class as opposed to inheritance

// Instead of requiring the caller's subclass' giant abstract class, give the caller
// How do we do composition ?

////////////////////////////////////
package Aquarium
/*
 * Interface delegation is really powerful
 * and you should generally consider how to use it whenever you
 * might use an abstract class in another language
 * It let's you use composition to plug-in behaviours
 * instead of requiring a lot of sub classes each specialized in a different way
 * */

fun main() {
    delegate()
}

fun delegate() {
    val pleco = Plecostomus2()
    println("Fish has color ${pleco.color}")
    pleco.eat()
}

// Let's start breaking up aquarium fish into interfaces
interface FishAction2 {
    fun eat()
}

//
interface FishColor {
    val color: String
}

/*
 * We can remove inheritance from aquarium fish
 * because we get all the functionality from the interfaces
 * and we don't even have to change the code in the body of plecostomus
 * */
// Fish color could have been implemented by a class instead of object
// But this time, we would have had to create many unnecessary same class objects
// Here, FishColor interface is implemented by GoldColor object which will take care of
// object at all
/* This means implement the interface fish color,
by deferring all calls to the object, gold color
So everytime you call the color property on this class, it will actually
call the color property on gold color
*/

```

```

* */
/*
* Of course there are different colors of plecostomi in the world
* So we can make the fish color object a constructor parameter
* with a default of gold color and defer calls to the color property whatever
* fish color we get passed in
* */
// Now Plecostomus2 doesn't have a body, all its overrides are handled by
// interface delegation
class Plecostomus2(fishColor: FishColor = GoldColor):
    FishAction2 by PrintingFishAction("a lot of algae"),
    FishColor by GoldColor

/*
* It doesn't really make sense to make multiple instances of
* gold color as they would all do the exact same thing
* Kotlin let's us declare a class where we can only have one instance by using
* the keyword "object" instead of "class"
* */
// This will declare a class and make exactly one instance of it
// The instance will be called gold color and there's no way
// to make another instance of this class but that's okay we don't need to
// If you're familiar with the Singleton Pattern this is how to implement it
/*
* In software engineering, the singleton pattern is a software design pattern
* that restricts the instantiation of a class to one "single" instance.
* This is useful when exactly one object is needed to coordinate actions across
* the system. The term comes from the mathematical concept of a singleton.
* */
object GoldColor : FishColor {
    override val color = "gold"
}

// If we were passed in a red color, then fish color would be by red color as
object RedColor : FishColor {
    override val color = "red"
}

// Instead of printing a fixed string, we print out whatever food we were passed
// Since we have a member variable food, we can't make PrintingFishAction an object
// We want a different instance for each food that we passed in
// Constructors are not allowed for "object"
class PrintingFishAction(val food: String) : FishAction2 {
    override fun eat() {
        println(food)
    }
}
////////////////////////////////////

```

Delegation Design Pattern

```
package Delegation
/*
 * KOTLIN DELEGATION
 *
 * Delegation is an object oriented design pattern
 * And Kotlin supports it natively
 * Delegation Pattern means delegating the responsibilities
 * to other objects.
 * */

class FilePlayer(private val file: String): Player {
    override fun play() {
        println("$file is playing...")
    }
}

class FileDownloader(private val file: String): Downloader {
    override fun download() {
        println("$file downloaded")
    }
}

/*
 * Here, we will be delegating the responsibility of
 * "download()" and "play()" interfaces to
 * "Downloader" and "Player" objects that we pass in
 * So the class is just forwarding the responsibility
 * */
class MediaFile(
    private val downloader: Downloader,
    private val player: Player
) : Downloader by downloader, Player by player {
    /*
     * We don't need to write following two methods
     * because Kotlin already supports delegation natively
     * This is boilerplate code
     * */
    // override fun download() {
    //     downloader.download()
    // }
    //
    // override fun play() {
    //     player.play()
    // }
}

fun main() {
```

```

    val file = "FileGenesis1.mp4"
    val mediaFile = MediaFile(FileDownloader(file), FilePlayer(file))
    mediaFile.download()
    mediaFile.play()
}

interface Downloader {
    fun download()
}

interface Player {
    fun play()
}

```

Difference Between "Open Class" and "Abstract Class"

// Imagine you have 2 classes

```

Class Person [parent class]
Class Coder [sub/child class]

```

*/*When you want to inherit Coder from Person you have to make Person open, so you can create objects from parent class(in our case it's Person).
When you don't need to make objects from parent class(in our case it's Person), you can make it abstract.
It works the same way as open does. But the main difference is that you cannot create objects from abstract class.
Abstract class cannot be instantiated and must be inherited, abstract classes are used when you want to inherit a class and you don't need to create objects from it.
Open modifier on the class allows inheriting it. If the class has not open modifier, it cannot be inherited.*/*

Practice Time | Abstract & Interface

*/*Let's go back to your spices. Make Spice an abstract class, and then create a subclass, Curry. Curry can have varying levels of spiciness, so we can create a method grind(). Spices are processed in different ways before they can be used. Add an abstract method grind() to the Spice class. Curry is ground into a powder, so let's call a method grind(). However, grinder is used to grind the spices. Then add the Grinder interface to the Curry class.*/*

// Delegation

// Using the provided code from the lesson for guidance, add a yellow color


```

fun main (args: Array<String>) {
    delegate()
}

fun delegate() {
    val pleco = Plecostomus()
    println("Fish has has color ${pleco.color}")
    pleco.eat()
}

interface FishAction {
    fun eat()
}

interface FishColor {
    val color: String
}

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

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

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

```

// Interface

*/*Create an interface, SpiceColor, that has a color property. You can use a
Create a singleton subclass, YellowSpiceColor, using the object keyword, bec
Add a color property to Curry of type SpiceColor, and set the default value
Add SpiceColor as an interface, and let it be by color.
Create an instance of Curry, and print its color. However, color is actually
Change your code so that the SpiceColor interface is added to the Spice clas*

// Solution Code

```

abstract class Spice(val name: String, val spiciness: String = "mild", color
    abstract fun prepareSpice()
}

class Curry(name: String, spiciness: String, color: SpiceColor = YellowSpice
    override fun grind() {
    }

    override fun prepareSpice() {

```

```

        grind()
    }
}

interface Grinder {
    fun grind()
}

interface SpiceColor {
    val color: String
}

object YellowSpiceColor : SpiceColor {
    override val color = "Yellow"
}

```

Data Classes

```

package Decorations
/*
 * DATA CLASSES
 * Often, we have classes that mostly act as data containers
 * In Kotlin, for classes that mostly hold data,
 * there is a class with benefits
 * */
fun main() {
    makeDecoration()
}

fun makeDecoration() {
    // Create instance of Decorations class
    val d1 = Decorations("granite")

    /*
     * With a data class printing the object
     * prints the values of properties
     * instead of just an address of the object
     * that is the object pointer
     * basically it creates toString for us to print the properties
     * */
    println(d1) // Decorations(rocks=granite)

    /*
     * Data class also provides an equals method to compare two
     * instances of a data class
     * */
    val d2 = Decorations("slate")
    println(d2) // Decorations(rocks=slate)
}

```

```

val d3 = Decorations("slate")
println(d3) // Decorations(rocks=slate)

// Comparison
println(d1 == d2) // false
println(d3 == d2) // true

// We can copy data objects using the copy method
// This creates a new object with the same
// property values
val d4 = d3.copy()
println(d3)
println(d4)

// Another Decoration
val d5 = Decorations2("crystal", "wood", "diver")
println(d5)

/*
 * DECOMPOSITION
 * To get at the properties and assign them to variables
 * Kotlin let's us use a process called decomposition
 * */

// We can make three variables, one for each property
// and assign the object to it
// Kotlin puts the property values in each variable and
// we can then use it
// We do need to put parentheses around the variables for decomposition
// The number of variables must match the number of properties
// or we get compiler error
// The variables are assigned in the order in which
// they are declared in the class
val (rock, wood, diver) = d5
// Or we can also do this alternatively
val (rock2, wood2, diver2) = Decorations2("crystal", "wood", "diver")
println(rock)
println(wood)
println(diver)
}

// Data classes must have
// at least one primary constructor parameter
data class Decorations(val rocks: String) {
}

data class Decorations2(
    val rocks: String,
    val wood: String,

```

```

        val diver: String) {
    }

```

Practice Time

*/*Create a simple data class, SpiceContainer, that holds one spice.
Give SpiceContainer a property, label, that is derived from the name of the
Create some containers with spices and print out their labels.*/*

// Solution Code

```

data class SpiceContainer(var spice: Spice) {
    val label = spice.name
}

```

```

val spiceCabinet = listOf(SpiceContainer(Curry("Yellow Curry", "mild")),
    SpiceContainer(Curry("Red Curry", "medium")),
    SpiceContainer(Curry("Green Curry", "spicy")))

```

```

for(element in spiceCabinet) println(element.label)

```

// My Code

```

package DataClasses

```

```

abstract class Spice(val name: String, val spiciness: String = "mild", ) {
}

```

```

class Curry(name: String, spice: String): Spice(name, spice) {
}

```

```

data class SpiceContainer(
    val spice: Spice,
    val label: String = spice.name,
    val spiciness: String = spice.spiciness
)

```

```

fun main() {
    val spiceCabinet = listOf(SpiceContainer(Curry("Yellow Curry", "mild")),
        SpiceContainer(Curry("Red Curry", "medium")),
        SpiceContainer(Curry("Green Curry", "spicy")))

    for (element in spiceCabinet) {
        println("${element.label}, ${element.spiciness}")
    }
}

```

Special Purpose Classes | Singletons, Enums, Sealed Classes

Singeltons | Objects

```
/*
 * SINGLETONS - "Object"
 *
 * To create singleton, use the "object" keyword
 * when you declare you class
 *
 * Anytime you're defining a class that
 * shouldn't be instantiated multiple times
 * you can use the "object" keyword in place of class
 *
 * Kotlin will instantiate exactly one instance of the class
 *
 * Since there can be only one MobyDick, we declare it as an object
 * instead of a class
 * */
object MobyDickWhale {
    val author = "Herman Melville"
    fun jump () {
        // ...
    }
}
```

Enums

```
/*
 * ENUMS
 *
 * which lets you enumerate items
 * enums actually define a class
 * and you can give them properties or even methods
 *
 * Enums are like singletons, Kotlin will make
 * exactly one red, exactly one green and exactly one blue
 * there is no way to create more than one color object
 * And, there is not any way to define more colors
 * other then where the enum is declared
 * */
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}

// The most basic usage of enum classes is implementing type-safe enums:
enum class Direction {
```

```

    NORTH, SOUTH, WEST, EAST
}

// Each enum constant is an object. Enum constants are separated with commas
// Since each enum is an instance of the enum class, it can be initialized a

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

```

Sealed Classes

```

/*
* SEALED CLASS
*
* It's a class that can be subclassed
* but only inside the file which it's declared
* If you try to subclass it in a different file, you'll get an error
* This makes sealed classes a safe way to represent a fixed number of types
*
* They're great for returning success or error from a network API
*
* */
sealed class Seal {

}

// If we want to create more Seals we have to put them
// In this file, since the Seal class is in this file!
// I can't subclass Seal in any other file
// Since They're all in the same file
// Kotlin knows statically(at compile time) about all of the subclasses
class Sealion: Seal()
class Walrus: Seal()

/*
* I can use a "when" statement to check
* what type of seal I have
* And If I don't match all of the types of seal
* Kotlin will give me a compiler error!
* */
fun matchSeal(seal: Seal): String {
    return when (seal) {
        is Walrus -> "walrus"
        is Sealion -> "sealion"
    }
}

```

```
}
```

Practice Time

```
// You used object in the previous lesson and quiz.  
// And now that you know about enums, here's the code for Color as an enum:  
  
enum class Color(val rgb: Int) {  
    RED(0xFF0000), GREEN(0x00FF00), BLUE(0x0000FF);  
}  
  
// In SpiceColor, change the type of color from String to the Color class, a  
// Hint: The color code for yellow is YELLOW(0xFFFF00)  
  
// Make Spice a sealed class.  
// What is the effect of doing this?  
// Why is this useful?  
  
// Solution Code  
interface SpiceColor {  
    val color: Color  
}  
  
object YellowSpiceColor : SpiceColor {  
    override val color = Color.YELLOW  
}  
  
// Answer Explanation:  
// Making Spice a sealed class helps keep all the spices together in one file
```

Lesson 5 | Kotlin Essentials: Beyond The Basics

Pairs

```
package Collections  
fun main() {  
    // Sample Generic Pair  
    val equipment = "fishnet" to "catching"  
    println(equipment.first)  
    println(equipment.second)  
  
    // You can also chain the pairs  
    val chain = "A" to "B" to "C" to "D"  
    println(chain.first) // Output: (A, B), C)  
    println(chain.first.first.first) // Output: A  
}
```

```

val testChain = ("A" to "B" to "C") to "D"
println(testChain)

// You can also create triplets
val triple: Triple<Int, Int, Int> = Triple(1, 2, 3)

// deconstructing
val fishnet = "fishnet" to "catching fish"
val (tool, use) = fishnet
val (first, second, third) = Triple(1, 2 ,3)

val fishnetString = fishnet.toString()
val fishnetList = fishnet.toList()

// We can use them to return more than one variable from a function
// and we can destruct it and use
val (tool2, use2) = giveMeATool()
}

fun giveMeATool(): Pair<String, String> {
    return Pair("fishnet", "catching")
    //    return ("fishnet" to "catching") // Alternative
}

```

Practice Time

```

/*
Let's go through an example of getting information about a book in the form

Let's create a basic book class, with a title, author, and year. Of course,
Create a method that returns both the title and the author as a Pair.
Create a method that returns the title, author and year as a Triple. Use the
Create a book instance.
Print out the information about the book in a sentence, such as: "Here is yo
*/

// My Code
package lesson_5
class Book(
    val title: String,
    val author: String,
    val year: Int
) {
    fun getTitleAuthor(): Pair<String, String> {
        return Pair(title, author)
    }

    fun getTitleAuthorYear(): Triple<String, String, Int> {

```



```

        return Triple(title, author, year)
    }
}

fun main() {
    val book = Book("Elon Musk", "Ashlee Vance", 2012)

    val (title, author) = book.getTitleAuthor()
    val (title1, author1, year) = book.getTitleAuthorYear()

    println("Here is your book ${book.title} written by ${book.author} in ${
}

// Solution Code
class Book(val title: String, val author: String, val year: Int) {

    fun getTitleAuthor(): Pair<String, String> {
        return (title to author)
    }

    fun getTitleAuthorYear(): Triple<String, String, Int> {
        return Triple(title, author, year)
    }
}

fun main(args: Array<String>) {
    val book = Book("Romeon and Juliet", "William Shakespeare", 1597)
    val bookTitleAuthor = book.getTitleAuthor()
    val bookTitleAuthorYear = book.getTitleAuthorYear()

    println("Here is your book ${bookTitleAuthor.first} by ${bookTitleAuthor

    println("Here is your book ${bookTitleAuthorYear.first} " +
        "by ${bookTitleAuthorYear.second} written in ${bookTitleAuthorYe
}

```

Practice Time

```

/*
One book is rarely alone, and one author rarely writes just one book.
Create a Set of book titles called allBooks, for example, by William Sha
Create a Map called library that associates the set of books, allBooks,
Use the collections function any() on library to see if any of the books
Create a MutableMap called moreBooks, and add one title/author to it.

Use getOrElsePut() to see whether a title is in the map, and if the title is

Hints:

```

any() is applied to a collection and takes a lambda as its argument,

```
myList.any {it.contains("name")}
```

getOrPut() is a handy function that will check whether a key is in a mapOf() may come in handy.

```
*/
```

// Solution Code

```
val allBooks = setOf("Macbeth", "Romeo and Juliet", "Hamlet", "A Midsummer N
val library = mapOf("Shakespeare" to allBooks)
println(library.any { it.value.contains("Hamlet") })

val moreBooks = mutableMapOf<String, String>("Wilhelm Tell" to "Schiller")
moreBooks.getOrPut("Jungle Book") { "Kipling" }
moreBooks.getOrPut("Hamlet") { "Shakespeare" }
println(moreBooks)

// My Code
val allBooksOfOneAuthor = setOf("A", "B", "C")
// { set of books, author }
val library = mapOf(
    "by William Shakespeare" to allBooksOfOneAuthor,
    "Genesis" to setOf("M", "H", "N")
)

val hamletFound = library.any() { it.value.contains("Hamlet") }
println("Hamlet is " + (if (hamletFound) " " else "not ") + "in the Library")

val moreBooks = mutableMapOf<String, Set<String>>()
moreBooks["Melo"] = setOf("istanbul", "ankara", "izmir")
moreBooks.getOrPut("Seno") { setOf("Senosis") }
```

Constants

// We can make top level constants and assign them a value at compile time

// We have "val" and "const val" now, What is the difference?

// Top Level, Compile Time Constant Variable

// The value is always determined at compile time

// const value set it compile time so we cannot call and execute a function

// to get its value set

```
const val num = 5
```

// However, const val only works at the top level and in classes declared wi

// Not with regular classes declared with class

```
/*
* So we can use this to create a file or object that
* contains only constants and import them one-by-one
```

```

* */
const val CONSTANT = "top-level constant"

object Constants {
    const val CONSTANT2 = "top-level constant"
}

val foo = Constants.CONSTANT2
// Kotlin does not have a concept of class level constants
// To define constants inside a class
// You have to wrap them into a companion object
class MyClass {
    /*
    * The differences between
    * "Regular Objects" and "Companion Objects" are as follows;
    *
    * -> Companion objects are initialized from the static constructor
    * of the containing class, that is, they are created when the object is
    *
    * -> Plain objects are initialized lazily on the first access to that of
    * that is, when they are first used
    * */
    companion object {
        const val CONSTANT3 = "constant inside companion"
    }
    // So you need to wrap constants in classes inside a companion object
}

fun main() {
    // Difference between val and const val
    // With val, the value that is assigned can be determined during program
    val number = 5

    // For example we can assign the return value from a function
    fun complexFunctionCall() {}
    // Because we can set it during execution
    val result = complexFunctionCall()

    // ERROR!
    // Modifier 'const' is not applicable to 'local variable'
    // This should be Top Level!
    const val num = 5
}

// There are 3 different ways in which you can create constants in Kotlin. 1

// For each situation, decide when you would use a CONSTANT, an ENUM, and a

```

Quiz Question

*/*Let's continue with our books setup to practice creating constants in Kotlin
 For each situation, decide when you would use a constant, an enum, and a data class
 Situation -> Data Type*

- 1) Storing simple values without any functionality. For example, a URL or a constant*
- 2) They are objects that store groups of values that are related. They offer functionality*
- 3) Creating objects that only have properties without additional functionality
 -> Use data classes
 /

Practice Time

Practice Time

*/*Create a top-level constant for the maximum number of books a person could borrow
 Inside the Book class, create a method canBorrow() that returns true or false
 Create a Constants object that provides constants to the book. For this example
 The base URL is really of interest to the Book class. As such, it makes sense to have it as a constant*

```
// Solution Code
const val MAX_NUMBER_BOOKS = 20

fun canBorrow(hasBooks: Int): Boolean {
    return (hasBooks < MAX_NUMBER_BOOKS)
}

object Constants {
    const val BASE_URL = "http://www.turtlecare.net/"
}

fun printUrl() {
    println(Constants.BASE_URL + title + ".html")
}

companion object {
    val BASE_URL = "http://www.turtlecare.net/"
}

// My Code
package lesson_5
const val maximumBooks = 100
```

```

object Constants {
    const val BASE_URL = "www.library.com/"
}

class Book(
    val title: String,
    val author: String,
    val year: Int,
    val maxBooks: Int = 0
) {

    companion object BookURL {
        const val BASE_URL = "www.library.com/"
    }

    fun printUrl() {
        println(
            Constants.BASE_URL + "/"
            + title + "/"
            + author + "/"
            + year + "/"
            + ".html"
        )
    }

    fun getTitleAuthor(): Pair<String, String> {
        return Pair(title, author)
    }

    fun getTitleAuthorYear(): Triple<String, String, Int> {
        return Triple(title, author, year)
    }

    fun canBorrow(): Boolean {
        return maxBooks < maximumBooks
    }
}

fun main() {
    val book = Book("Elon Musk", "Ashlee Vance", 2012)

    val (title, author) = book.getTitleAuthor()
    val (title1, author1, year) = book.getTitleAuthorYear()

    println("Here is your book ${book.title} written by ${book.author} in ${
}

```

Extension Functions

```

/*
Extension Functions allow you to add functions to an existing class without
*/

// Extension Functions are great way to add helpful functionality to classes

// We merely make a new function callable
// with the dot-notation on variables of this type
fun String.hasSpaces(): Boolean {
    val found = this.find { it == ' ' }
    return found != null
}

// We can also write one line
fun String.hasSpacesShorterWay() = this.find { it == ' ' } != null

fun Int.hasZero(): Boolean {
    val found = this.toString().find { it == '0' }
    return found != null
}

fun main() {
    println("Does it have spaces?".hasSpaces())
    println(1232.hasZero())
}

// You can also use it to separate the core API from helper methods on class
class AquariumPlant(val color: String, private val size: Int)
// This extension function is just a helper
// Extension functions are defined outside of the class they extend
fun AquariumPlant.isRed() = color == "Red"
// So, they cannot access to private variables!
fun AquariumPlant.isBig() = size > 50
// ERROR// Cannot access 'size': it is private in 'AquariumPlant'

// You should think of them as helper functions that rely only on the public
// xtension functions are always resolved statically based on the variable t

// We can define extension properties too, "isGreen" is the property name
// We can use "isGreen" just like a regular property
val AquariumPlant.isGreen: Boolean
    get() = color == "Green"

fun propertyExample() {
    val plant = AquariumPlant("Green", 50)
    println(plant.isGreen) // true
}

```

Extension Function Examples

```
package lesson_5
// We merely make a new function callable
// with the dot-notation on variables of this type
fun String.hasSpaces(): Boolean {
    val found = this.find { it == ' ' }
    return found != null
}

// We can also write one line
fun String.hasSpacesShorterWay() = this.find { it == ' ' } != null

fun Int.hasZero(): Boolean {
    val found = this.toString().find { it == '0' }
    return found != null
}

// This extension function is just a helper
// Extension functions are defined outside of the class they extend
fun AquariumPlant.isRed() = color == "Red"
// So, they cannot access to private variables!
//fun AquariumPlant.isBig() = size > 50
// ERROR// Cannot access 'size': it is private in 'AquariumPlant'

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")

// We can define extension properties too, "isGreen" is the property name
// We can use "isGreen" just like a regular property
val AquariumPlant.isGreen: Boolean
    get() = color == "Green"

fun propertyExample() {
    val plant = AquariumPlant("Green", 50)
    println(plant.isGreen) // true
}

// We can also make the class we are extending which is sometimes called the
// If we do that then the "this" variable used in the body can be null
// The object on which the extension function is called can be null
// We indicate this with a question mark after "AquariumPlant?" but before t
fun AquariumPlant?.pull() {
    // Inside the body we can test for null by using "?.apply"
    // If object is not null, the apply body will be executed
    this?.apply {
```

```

        println("removing $this")
    }
}

/*
 * You would want to take a nullable receiver if you expect
 * the callers will want to call your extension function on nullable variable
 * */
fun nullableExample() {
    val plantNull: AquariumPlant? = null
    plantNull.pull() // ok

    val plantNotNull = AquariumPlant("Black", 15)
    plantNotNull.pull() // ok
}

fun main() {
    val plant = GreenLeafyPlant(50)
    plant.print() // GreenLeafyPlant
    /*
     * Compiler just looks at the type of the variable
     * So at compile time, AquariumPlant is an AquariumPlant
     * So it will print "AquariumPlant"
     * */
    val aquariumPlant: AquariumPlant = plant // Type is not "GreenLeafyPlant"
    aquariumPlant.print() // AquariumPlant
    propertyExample() // true
    nullableExample()
}

```

Practice Time

```

/*
It can be useful to know the weight of a book, for example, for shipping. The
weight of a book is calculated as follows:
    Add a mutable property pages to Book.
    Create an extension function on Book that returns the weight of a book as a Double.
    Create another extension, tornPages(), that takes the number of torn pages and returns the
    weight of the torn pages.
    Write a class Puppy with a method playWithBook() that takes a book as an argument and
    returns the weight of the book after playing with it.
    Create a puppy and give it a book to play with, until there are no more pages left.

Note: If you don't want to give your puppy a book, then create a puzzle toy instead.

*/

// Solution Code
fun Book.weight() : Double { return (pages * 1.5) }

fun Book.tornPages(torn: Int) = if (pages >= torn) pages -= torn else pages

```



```

class Puppy() {
    fun playWithBook(book: Book) {
        book.tornPages(Random().nextInt(12))
    }
}

val puppy = Puppy()
val book = Book("Oliver Twist", "Charles Dickens", 1837, 540)

while (book.pages > 0) {
    puppy.playWithBook(book)
    println("${book.pages} left in ${book.title}")
}
println("Sad puppy, no more pages in ${book.title}. ")

////////////////////////////////////
// My Code
package lesson_5
import kotlin.random.Random

const val maximumBooks = 100

object Constants {
    const val BASE_URL = "www.library.com/"
}

// Extension function
fun Book.getWeight(): Double {
    return 1.5 * pages
}

// Extension function
fun Book.tornPages(tornPages: Int) {
    pages -= tornPages
}

// Extension function
fun Book.printNumOfPages() {
    println("Number of pages: $pages")
}

class Puppy() {
    fun playWithBook(book: Book) {
        val randPagesToTorn = Random.nextInt(1, book.pages + 1)
        book.tornPages(randPagesToTorn)
    }
}

class Book(

```

```

    val title: String,
    val author: String,
    val year: Int,
    val maxBooks: Int = 0,
    var pages: Int = 0
) {

    companion object BookURL {
        const val BASE_URL = "www.library.com/"
    }

    fun printUrl() {
        println(
            Constants.BASE_URL + "/"
            + title + "/"
            + author + "/"
            + year + "/"
            + ".html"
        )
    }

    fun getTitleAuthor(): Pair<String, String> {
        return Pair(title, author)
    }

    fun getTitleAuthorYear(): Triple<String, String, Int> {
        return Triple(title, author, year)
    }

    fun canBorrow(): Boolean {
        return maxBooks < maximumBooks
    }
}

fun main() {
    val book = Book("Elon Musk", "Ashlee Vance", 2012)

    val (title, author) = book.getTitleAuthor()
    val (title1, author1, year) = book.getTitleAuthorYear()

    println("Here is your book ${book.title} written by ${book.author} in ${

// 9. Quiz: Practice Time
book.pages = 100
val pupy = Puppy()
while (book.pages > 0) {
    pupy.playWithBook(book)
    book.printNumOfPages()
}
}
```

Generic Classes

```
// With generics we can make the list generic so it can hold any type of obj
// It's like you make the type a wildcard(Joker) that will fit many types
package Aquarium.generics

// GENERICS
// How to declare a generic class with an upper bound and use it
fun main() {
    genericExample()
}

open class WaterSupply(var needsProcessed: Boolean)

class TapWater : WaterSupply(true) {
    fun addChemicalCleaners() {
        needsProcessed = false
    }
}

class FishStoreWater : WaterSupply(false)

class LakeWater : WaterSupply(true) {
    fun filter() {
        needsProcessed = false
    }
}

// To ensure that our parameter must be nonnull but can still be any type
// We remove the question mark "Aquarium<T: Any?>" and just say "Aquarium<T:"
// This makes it impossible to pass null
class Aquarium<T: WaterSupply>(val waterSupply: T) {
    fun addWater() {
        // Check throws an error if condition is not true, continues otherwise
        check(!waterSupply.needsProcessed) { "water supply needs processed" }
        println("Adding water from $waterSupply")
    }
}

// But we really want to ensure our type is a water supply
// We can be as specific as we want with the generic constraint and replace
// any with the top of any type hierarchy we want to use

fun genericExample() {
    // val aquarium = Aquarium(TapWater()) // Type inference
    val aquarium = Aquarium<TapWater>(TapWater())
    aquarium.waterSupply.addChemicalCleaners()

    // We are able to pass a string in as a water supply
}
```

```

    // This is because type T doesn't have any bounds
    // So it can actually be set to any type, that could be a problem
    // val aquarium2 = Aquarium("string")
    // println(aquarium2.waterSupply)

    // Another unexpected example is passing in nulls this also works
    // I didn't really want to let WaterSupply be null
    // Because T can be any type including nullable
    // val aquarium3 = Aquarium(null)
    // println(aquarium3.waterSupply)

    val aquarium4 = Aquarium(LakeWater())
    aquarium4.waterSupply.filter()
    aquarium4.addWater()
}

```

Practice Time | Generics

```

/*
Using type hierarchies with generic classes follows a pretty basic pattern 1

    Create a type/class hierarchy. The parent is non-specific and the sub-ty
    There is at least one shared property between the classes/types, and it
    We then have a class that uses all the subtypes, and performs different

Let's put this into practice using building materials and a building that ne

    Create a new package and file and call them Buildings.
    Create a class BaseBuildingMaterial with a property numberNeeded that is
    Create two subclasses, Wood and Brick. For BaseBuildingMaterial you need
    Create a generic class Building that can take any building material as a
    A building always requires 100 base materials. Add a property baseMaterial
    Add another property, actualMaterialsNeeded and use a one-line function

    Add a method build() that prints the type and number of materials needed
    Hint: Use reflection to get the class and simple name: instance::class

    Create a main function and make a building using Wood.
    If you did this correctly, running main() will print something like "400

*/

// Solution Code
open class BaseBuildingMaterial() {
    open val numberNeeded = 1
}

class Wood : BaseBuildingMaterial() {
    override val numberNeeded = 4
}

```

```

}

class Brick : BaseBuildingMaterial() {
    override val numberNeeded = 8
}

class Building<T: BaseBuildingMaterial>(val buildingMaterial: T) {
    val baseMaterialsNeeded = 100
    val actualMaterialsNeeded = buildingMaterial.numberNeeded * baseMaterial

    fun build() {
        println(" $actualMaterialsNeeded ${buildingMaterial::class.simpleName}")
    }
}

fun main(args: Array<String>) {
    Building(Wood()).build()
}

// Output: 400 Wood required

```

Generics in & out

```

// Out types are type parameters that only ever occur in return values of functions
// In types can be used anytime the generic is only used as an argument to a function

/* More specifically
    -> IN TYPES CAN ONLY BE PASSED INTO AN OBJECT (Can be used as parameter)
    -> OUT TYPES CAN ONLY BE PASS OUT OF AN OBJECT OR RETURNED (Can be used

Constructors can take out types as arguments but functions never can
*/

package Aquarium.generics
// GENERICS
// How to declare a generic class with an upper bound and use it
fun main() {
    genericExample()
}

fun addItemTo(aquarium: Aquarium<WaterSupply>) = println("item added")

open class WaterSupply(var needsProcessed: Boolean)

class TapWater : WaterSupply(true) {
    fun addChemicalCleaners() {
        needsProcessed = false
    }
}

```

```

}

class FishStoreWater : WaterSupply(false)

class LakeWater : WaterSupply(true) {
    fun filter() {
        needsProcessed = false
    }
}

// To ensure that our parameter must be nonnull but can still be any type
// We remove the question mark "Aquarium<T: Any?>" and just say "Aquarium<T:
// This makes it impossible to pass null
class Aquarium<out T: WaterSupply>(val waterSupply: T) {
    fun addWater(cleaner: Cleaner<T>) {
        // Check throws an error if condition is not true, continues otherwise
        if (waterSupply.needsProcessed) {
            cleaner.clean(waterSupply)
        }

        println("Adding water from $waterSupply")
    }
}

// But we really want to ensure our type is a water supply
// We can be as specific as we want with the generic constraint and replace
// any with the top of any type hierarchy we want to use

interface Cleaner<in T: WaterSupply> {
    fun clean(waterSupply: T)
}

class TapWaterCleaner: Cleaner<TapWater> {
    override fun clean(waterSupply: TapWater) {
        waterSupply.addChemicalCleaners()
    }
}

fun genericExample() {
    // val aquarium = Aquarium(TapWater()) // Type inference
    val aquarium = Aquarium<TapWater>(TapWater())
    aquarium.waterSupply.addChemicalCleaners()

    // We are able to pass a string in as a water supply
    // This is because type T doesn't have any bounds
    // So it can actually be set to any type, that could be a problem
    // val aquarium2 = Aquarium("string")
    // println(aquarium2.waterSupply)

    // Another unexpected example is passing in nulls this also works

```

```

        // I didn't really want to let WaterSupply be null
        // Because T can be any type including nullable
//     val aquarium3 = Aquarium(null)
//     println(aquarium3.waterSupply)

    val cleaner = TapWaterCleaner()
    val aquarium4 = Aquarium(TapWater())
    aquarium4.addWater(cleaner)

    // If we did not put this "out" -> "class Aquarium<out T: WaterSupply>",
    addItemTo(aquarium)
}

```

Practice Time

```

/*
That was a lot of explanations. Fortunately, IntelliJ gives you hints as to

Look at the code from the previous practice and consider whether it can be a
Notice that the parameter is underlined gray, and if you hover over T, Intel
*/

class Building<out T: BaseBuildingMaterial>(val buildingMaterial: T)

```

Practice Time | Generic Functions

```

// We can use generic functions for methods too
package Aquarium.generics

// GENERICS
// How to declare a generic class with an upper bound and use it
fun main() {
    genericExample()
}

// Generic Function Example
fun <T: WaterSupply> isWaterClean(aquarium: Aquarium<T>) {
    println("Aquarium water is clean ${aquarium.waterSupply.needsProcessed}")
}

fun addItemTo(aquarium: Aquarium<WaterSupply>) = println("item added")

open class WaterSupply(var needsProcessed: Boolean)

class TapWater : WaterSupply(true) {
    fun addChemicalCleaners() {

```

```

        needsProcessed = false
    }
}

class FishStoreWater : WaterSupply(false)

class LakeWater : WaterSupply(true) {
    fun filter() {
        needsProcessed = false
    }
}

// To ensure that our parameter must be nonnull but can still be any type
// We remove the question mark "Aquarium<T: Any?>" and just say "Aquarium<T:
// This makes it impossible to pass null
class Aquarium<out T: WaterSupply>(val waterSupply: T) {
    fun addWater(cleaner: Cleaner<T>) {
        // Check throws an error if condition is not true, continues otherwi
        if (waterSupply.needsProcessed) {
            cleaner.clean(waterSupply)
        }
        println("Adding water from $waterSupply")
    }
    // Declare a parameter type parameter R, but make it a real type
    inline fun <reified R: WaterSupply> hasWaterSupplyOfType() = waterSupply
}

// But we really want to ensure our type is a water supply
// We can be as specific as we want with the generic constraint and replace
// any with the top of any type hierarchy we want to use

interface Cleaner<in T: WaterSupply> {
    fun clean(waterSupply: T)
}

class TapWaterCleaner: Cleaner<TapWater> {
    override fun clean(waterSupply: TapWater) {
        waterSupply.addChemicalCleaners()
    }
}

fun genericExample() {
    // val aquarium = Aquarium(TapWater()) // Type inference
    val aquarium = Aquarium<TapWater>(TapWater())
    aquarium.waterSupply.addChemicalCleaners()

    // We are able to pass a string in as a water supply
    // This is because type T doesn't have any bounds
    // So it can actually be set to any type, that could be a problem
    // val aquarium2 = Aquarium("string")
    // println(aquarium2.waterSupply)
}

```



```

    // Another unexpected example is passing in nulls this also works
    // I didn't really want to let WaterSupply be null
    // Because T can be any type including nullable
    //     val aquarium3 = Aquarium(null)
    //     println(aquarium3.waterSupply)

    val cleaner = TapWaterCleaner()
    val aquarium4 = Aquarium(TapWater())
    aquarium4.addWater(cleaner)

    // If we did not put this "out" -> "class Aquarium<out T: WaterSupply>",
    addItemTo(aquarium)

    isWaterClean<TapWater>(aquarium)

    aquarium4.hasWaterSupplyOfType<TapWater>() // True
    aquarium4.waterSupply.isType<LakeWater>() // False
}

inline fun <reified T: WaterSupply> WaterSupply.isType() = this is T

```

Practice Time

```

/*
Create a generic function for type BaseBuildingMaterial and call it isSmallBuilding
Note: For this function, IntelliJ recommends not to inline the function. Get
*/

fun <T : BaseBuildingMaterial> isSmallBuilding(building: Building<T>) {
    if (building.actualMaterialsNeeded < 500) println("Small building")
    else println("large building")
}

isSmallBuilding(Building(Brick()))

```

Annotations

```

/*
Annotations are a means of attaching metadata to code, that is, the Annotation
*/

// Annotations go right before the thing that is Annotated, and most things
// Some annotations can even take arguments
// They're really useful if you are exporting Kotlin to Java, but otherwise
*/

annotation class ImAPlant

```

```

@Target(AnnotationTarget.PROPERTY_GETTER)
annotation class OnGet

@Target(AnnotationTarget.PROPERTY_SETTER)
annotation class OnSet

@ImAPlant class Plant {
    fun trim() {}
    fun fertilize() {}

    @get:OnGet
    val isGrowing: Boolean = true

    @set:OnSet
    var needsFood: Boolean = false
}

fun reflections() {
    val classObj = Plant::class

    // print all annotations
    for (annotation in classObj.annotations) {
        println(annotation.annotationClass.simpleName)
    }

    // find one annotation, or null
    val annotated = classObj

    annotated?.apply {
        println("Found a plant annotation!")
    }
}

```

Labeled Breaks

```

// Kotlin has several ways of controlling the flow
// Kotlin gives you additional control over loops with what's called a label
// Any expression in Kotlin may be marked with a label
// Labeled break can be used (labeled form) to terminate the desired loop (c

fun main() {
    for (i in 1..10) {
        for (j in 1..10) {
            if (i > 5) {
                break
            }
        }
    }
}

```

```

// Labeled Breaks
loop@ for (i in 1..10) {
    for (j in 1..10) {
        if (i > 5) {
            println()
            break@loop
        }
    }
} // break@loop will come here
// then ends the first loop
// different than just "break" keyword
}

```

Lambdas Recap

```

// A lambda is an anonymous function, a function without a name
data class Fish(val name: String)
fun main() {
    // Lambda function
    { println("Hello Lambda!") }()

    // We can assign lambda to a variable
    val waterFilter = { dirty: Int -> dirty / 2 }
    // Run lambda function
    println(waterFilter(30))

    val myFish = listOf(Fish("Flipper"), Fish("Moby Dick"), Fish("Dory"))

    // "joinToString" creates a string from all the names of the element
    // in the list separated using this applied separator
    val list = myFish.filter { it.name.contains('i') }.joinToString(" ") { i
    println(list)
}

```

Practice Time | Game

```

/*
In this practice, you are going to write the the first part of a higher-order

Create a new file.
Create an enum class, Directions, that has the directions NORTH, SOUTH,
Create a class Game.
Inside Game, declare a var, path, that is a mutable list of Direction.
Create 4 lambdas, north, south, east, and west, that add the respective
Add another lambda, end, that:
    Adds END to path

```

Prints "Game Over"
Prints the path
Clears the path
Returns false
Create a main function.
Inside main(), create an instance of Game.
To test your code so far, in main() print the path, then invoke north, €

You should see this output:

```
> [START]
Game Over: [START, NORTH, SOUTH, EAST, WEST, END]
[]
```

You will finish your game as the last practice in this course.

**/*

```
////////////////////////////////////
// Solution Code
enum class Direction {
    NORTH, EAST, WEST, SOUTH, START, END
}

class Game {
    var path = mutableListOf<Direction>(Direction.START)
    val north = { path.add(Direction.NORTH) }
    val south = { path.add(Direction.SOUTH) }
    val east = { path.add(Direction.EAST) }
    val west = { path.add(Direction.WEST) }
    val end = { path.add(Direction.END); println("Game Over: $path"); path.c
}

fun main(args: Array<String>) {
    val game = Game()
    println(game.path)
    game.north()
    game.south()
    game.east()
    game.west()
    game.end()
    println(game.path)
}

////////////////////////////////////
// My Code
package lesson_6
enum class Directions {
    START, END,
    NORTH, SOUTH, EAST, WEST
}
```

```

fun main() {
    val game = Game()
    println(game.path)
    game.east() // Don't forget the add parentheses "( )" at the end of Lamk
    game.north()
    game.south()
    game.west()
    game.end()
    println(game.path)
}

class Game {
    var path: MutableList<Directions> = mutableListOf(Directions.START)
    val north = { path.add(Directions.NORTH) }
    val south = { path.add(Directions.SOUTH) }
    val east = { this.path.add(Directions.EAST) }
    val west = { path.add(Directions.WEST) }
    val end = {
        path.add(Directions.END)
        println("Game Over: $path")
        path.clear()
        false
    }
}

```

Higher-order Functions

```

/*
* WRITING HIGHER ORDER FUNCTIONS WITH EXTENSIONS LAMBDAS
* is the most advanced part of the Kotlin Language
* */

// There are tons of built in functions in the Kotlin standard library that
// A higher-order function is a function that takes another function as para
data class Fish(var name: String)

fun main() {
    fishExamples()
}

fun fishExamples() {
    val fish = Fish("splashy")

    // "run" is an extension that works with all data types
    // It takes one lambda as its argument and returns the result of executi
    println(fish.run { "$name:" })
}

```

```

// "apply" is similar to run and can also be used on all data types
// but unlike "run" which returns the result of the block function
// "apply" returns the object it's applied to, so if we applied it to a
// It will return the fish object
// It turns out that "apply" can be really useful for calling functions
// on a newly created object
println(fish.apply{})

val fish2 = Fish("Melo").apply { name = "Genesis" }
println(fish2.name)

/*
 * So the difference is that "run" returns the result of executing the block
 * while "apply" returns the object after the lambda has been applied
 * This is a really common pattern for initializing objects
 */

// There is also "let"
// "let" returns a copy of the changed objects
// Let is particularly useful for chaining manipulations together
println(fish.let { it.name.capitalize() }
    .let { it + "fish" }
    .let { it.length }
    .let { it + 35 })

// Here we're saying
// With fish.name call this.uppercase()
// Under the hood, with is a higher order function
with (fish.name) {
    // We don't actually need this, because it's implicit
    // this.uppercase()
    println(uppercase()) // SPLASHY
    // capitalize returns a copy of the passed in string
    // It does not change the original string
}

// We can replace "with" with "myWith"
// fish.name is our named argument and
// uppercase() is our block function
myWith(fish.name) {
    // block function
    println(uppercase()) // SPLASHY
    // uppercase returns a copy of the passed in string
    // It does not change the original string
}
println("Original fish name: ${fish.name}")
}

// "block" is now an extension function on a string object
// And we can apply it to a string

```

```

fun myWith(name: String, block: String.() -> Unit) {
    // We take name and call block on it
    name.block()
}

```

Practice Time

```

/*
Create an extension on List using a higher order function that returns all t

val numbers = listOf<Int>(1,2,3,4,5,6,7,8,9,0)

Should return
> [3, 6, 9, 0]
*/

// Solution Code
fun main() {
    val numbers = listOf(1,2,3,4,5,6,7,8,9,0)
    println(numbers.divisibleBy3())
    println(numbers.divisibleBy { it.rem(3) })
}

fun List<Int>.divisibleBy(block: (Int) -> Int): List<Int> {
    val result = mutableListOf<Int>()
    for (item in this) {
        if (block(item) == 0) {
            result.add(item)
        }
    }
    return result
}

// My Code
fun List<Int>.divisibleBy3(): List<Int> {
    return this.filter { it % 3 == 0 }
}

```

Inline

```

package Aquarium5
fun main() {
    fishExamples2()
}

data class Fish2(var name: String)

```

```

fun fishExamples2() {
    val fish = Fish("splashy")

    // PROBLEM HERE!
    // Every time we call myWith, Kotlin will make a new lambda object!
    // Which takes CPU time and memory!
    // Lambdas are objects
    // A Lambda expression is an instance of a function interface
    // which is itself a subtype of object
    myWith(fish.name) {
        println(uppercase()) // SPLASHY
    }

    // To help understand, we can write it out longhand like this
    // Without inline an object is created every call to myWith
    myWith(fish.name, object : Function1<String, Unit> {
        override fun invoke(name: String) {
            name.capitalize()
        }
    })
    // When the inline transform is applied,
    // the call to the lambda is replaced with the contents of the
    // function body of the lambda
    // In our myWith example when we apply the transform
    // capitalize is called directly on fish.name
    // This is really important
    // Kotlin let's us define Lambda-based APIs with zero overhead
    // It won't even pay the cost of calling the function myWith
    // since it gets inlined
    // Inlining large functions does increase your code size
    // so it's best used for simple functions like myWith

    // with inline no object is created and lambda body is inlined here
    fish.name.capitalize()
}

// To fix this problem, Kotlin let's us define myWith as inline
// That is a promise that every time myWith is called
// it will actually transform the source code to inline, the function
// That is, the compiler will change the code to replace the Lambda
// with the instructions inside the Lambda, that means zero overhead
inline fun myWith2(name: String, block: String.() -> Unit) {
    name.block()
}

```

Practice Time

/*

In this practice, you will finish your simple game using higher-order functions.

In the game class, create a function move() that takes an argument called where.

Hint: Declaring a function that takes a lambda as its argument:

```
fun move(where: () -> Boolean )
```

Inside move(), invoke the passed-in lambda.

In the Game class, create a function makeMove() that takes a nullable String as its argument.

Inside makeMove, test whether the String is any of the 4 directions and invoke move() if so.

Hint: You can call the function like this:

```
move(north)
```

In main() add a while loop that is always true.

Inside the loop, print instructions to the player:

```
print("Enter a direction: n/s/e/w:")
```

Call makeMove() with the contents of the input from the user via readLine().

Remove the code for testing the first version of your game.

Run your program.

Challenge:

Create a simple "map" for your game, and when the user moves, show a description of the current location.

Use a Location class that takes a default width and height to track location.

You can create a matrix like this:

```
val map = Array(width) { arrayOfNulls<String>(height) }
```

Use an init block to initialize your map with descriptions for each location.

When you move() also updateLocation(). There is some math involved to print the map.

When you are done, zip up the code and send it to a friend to try your program.

*/

// Solution Code

```
fun move(where: () -> Boolean ) {  
    where.invoke()  
}
```

```
fun makeMove(command:String?) {  
    if (command.equals("n")) move(north)  
    else if (command.equals("s")) move(south)
```

```

        else if (command.equals("e")) move(east)
        else if (command.equals("w")) move(west)
        else move(end)
    }

while (true) {
    print("Enter a direction: n/s/e/w: ")
    game.makeMove(readLine())
}

////////////////////////////////////

////////////////////////////////////
// My Solution
enum class Directions {
    START, END,
    NORTH, SOUTH, EAST, WEST
}

class Map(val width: Int = 5, val height: Int = 5) {
    // 1 2 3 4 5
    // 1 . . . .
    // 2 . . . .
    // 3 . . C .
    // 4 . . . .
    // 5 . . . .
    private var location = mutableListOf(3, 3) // Center of the map { x, y }
    fun updateLocation(direction: String?): Boolean {
        val newLocation = location
        when (direction) {
            "n" -> newLocation[0] += 1
            "e" -> newLocation[1] += 1
            "s" -> newLocation[0] -= 1
            "w" -> newLocation[1] -= 1
        }
        if (isInside(newLocation[0], newLocation[1])) {
            location = newLocation
            printLocation()
            return true
        }
        else {
            println("Oops// You cannot move outside the map!")
            return false
        }
    }
}

private fun printLocation() {
    println("X: ${location[0]}, Y: ${location[1]}")
}

private fun isInside(x: Int, y: Int): Boolean {

```

```

        if (x < 1 || x > 5 || y < 1 || y > 5) {
            return false
        }
        return true
    }
}

fun main() {
    val game = Game()
    val map = Map()
    var validMove = true
    while (validMove) {
        print("Enter a direction: n/s/e/w: ")
        val direction = readLine()
        validMove = map.updateLocation(direction)
        if (validMove)
            game.makeMove(direction)
    }
    game.end()
}

class Game {
    var path: MutableList<Directions> = mutableListOf(Directions.START)
    val north = { path.add(Directions.NORTH) }
    val south = { path.add(Directions.SOUTH) }
    val east = { path.add(Directions.EAST) }
    val west = { path.add(Directions.WEST) }
    val end = {
        path.add(Directions.END)
        println("Game Over: $path")
        path.clear()
        false
    }

    private fun move(where: () -> Boolean) {
//        where.invoke() // Alternative
        where()
    }

    fun makeMove(direction: String?) {
        when (direction) {
            "n" -> move(north)
            "s" -> move(south)
            "e" -> move(east)
            "w" -> move(west)
            else -> move(end)
        }
    }
}

```

SAM - Single Abstract Method

```
// You'll run into SAM all the time in APIs written in the Java

//////////////////////////////////// JAVA //////////////////////////////////////
package SAM;

// Java Code
class JavaRun {
    public static void runNow(Runnable runnable) {
        runnable.run();
    }
}

////////////////////////////////////

//////////////////////////////////// KOTLIN //////////////////////////////////////
package SAM
/*
 * SAM - Single Abstract Method
 * You'll run into SAM all the time in APIs written in the Java
 * */

/*
 * Runnable and callable are two examples
 * Basically, SAM just means an interface with one method on it, That's it
 * In Kotlin, we have to call functions that take SAM
 * as parameters all the time
 * */
//interface Runnable {
//    fun run()
//}
//
//interface Callable<T> {
//    fun call(): T
//}
//interface Runnable {
//    fun run()
//}
//
//interface Callable<T> {
//    fun call(): T
//}

// Int Kotlin, we can pass a lambda in place of a SAM
fun example2() {
    JavaRun.runNow {
        println("Passing a lambda as a runnable")
    }
}
```

```
}
```

```
fun example() {  
    val runnable = object: Runnable {  
        override fun run() {  
            println("I'm a runnable")  
        }  
    }  
    JavaRun.runNow(runnable)  
}
```