```
// Single-line comments start with //
/*
Multi-line comments look like this.
*/

// The "package" keyword works in the same way as in Java.
package com.learnxinyminutes.kotlin

/*
The entry point to a Kotlin program is a function named "main".
The function is passed an array containing any command-line arguments.
Since Kotlin 1.3 the "main" function can also be defined without
any parameters.
*/
fun main(args: Array<String>) {
    /*
    Declaring values is done using either "var" or "val".
    "val" declarations cannot be reassigned, whereas "vars" can.
    */
    val fooVal = 10 // we cannot later reassign fooVal to something else
    var fooVar = 10
    fooVar = 20 // fooVar can be reassigned

    /*
    In most cases, Kotlin can determine what the type of a variable is,
    so we don't have to explicitly specify it every time.
    We can explicitly declare the type of a variable like so:
    */
    val foo: Int = 7

    /*
    Strings can be represented in a similar way as in Java.
    Escaping is done with a backslash.
    */
    val fooString = "My String Is Here!"
    val barString = "Printing on a new line?\nNo Problem!"
    val bazString = "Do you want to add a tab?\tNo Problem!"
    println(fooString)
    println(barString)
    println(bazString)

    /*
    A raw string is delimited by a triple quote (""").
    Raw strings can contain newlines and any other characters.
    */
    val fooRawString = """
fun helloWorld(val name : String) {
   println("Hello, world!")
}
"""
    println(fooRawString)

    /*
    Strings can contain template expressions.
    A template expression starts with a dollar sign ($).
    */
    val fooTemplateString = "$fooString has ${fooString.length} characters"
    println(fooTemplateString) // => My String Is Here! has 18 characters

    /*
    For a variable to hold null it must be explicitly specified as nullable.
    A variable can be specified as nullable by appending a ? to its type.
    We can access a nullable variable by using the ?. operator.
    We can use the ?: operator to specify an alternative value to use
    if a variable is null.
    */
    var fooNullable: String? = "abc"
    println(fooNullable?.length) // => 3
    println(fooNullable?.length ?: -1) // => 3
```



```
    fooNullable = null
    println(fooNullable?.length) // => null
    println(fooNullable?.length ?: -1) // => -1

    /*
    Functions can be declared using the "fun" keyword.
    Function arguments are specified in brackets after the function name.
    Function arguments can optionally have a default value.
    The function return type, if required, is specified after the arguments.
    */
    fun hello(name: String = "world"): String {
        return "Hello, $name!"
    }
    println(hello("foo")) // => Hello, foo!
    println(hello(name = "bar")) // => Hello, bar!
    println(hello()) // => Hello, world!

    /*
    A function parameter may be marked with the "vararg" keyword
    to allow a variable number of arguments to be passed to the function.
    */
    fun varargExample(vararg names: Int) {
        println("Argument has ${names.size} elements")
    }
    varargExample() // => Argument has 0 elements
    varargExample(1) // => Argument has 1 elements
    varargExample(1, 2, 3) // => Argument has 3 elements

    /*
    When a function consists of a single expression then the curly brackets can
    be omitted. The body is specified after the = symbol.
    */
    fun odd(x: Int): Boolean = x % 2 == 1
    println(odd(6)) // => false
    println(odd(7)) // => true

    // If the return type can be inferred then we don't need to specify it.
    fun even(x: Int) = x % 2 == 0
    println(even(6)) // => true
    println(even(7)) // => false

    // Functions can take functions as arguments and return functions.
    fun not(f: (Int) -> Boolean): (Int) -> Boolean {
        return {n -> !f.invoke(n)}
    }
    // Named functions can be specified as arguments using the :: operator.
    val notOdd = not(::odd)
    val notEven = not(::even)
    // Lambda expressions can be specified as arguments.
    val notZero = not {n -> n == 0}
    /*
    If a lambda has only one parameter
    then its declaration can be omitted (along with the ->).
    The name of the single parameter will be "it".
    */
    val notPositive = not {it > 0}
    for (i in 0..4) {
        println("${notOdd(i)} ${notEven(i)} ${notZero(i)} ${notPositive(i)}")
    }

    // The "class" keyword is used to declare classes.
    class ExampleClass(val x: Int) {
        fun memberFunction(y: Int): Int {
            return x + y
        }

        infix fun infixMemberFunction(y: Int): Int {
            return x * y
        }
```

```
}
/*
To create a new instance we call the constructor.
Note that Kotlin does not have a "new" keyword.
*/
val fooExampleClass = ExampleClass(7)
// Member functions can be called using dot notation.
println(fooExampleClass.memberFunction(4)) // => 11
/*
If a function has been marked with the "infix" keyword then it can be
called using infix notation.
*/
println(fooExampleClass infixMemberFunction 4) // => 28

/*
Data classes are a concise way to create classes that just hold data.
The "hashCode"/"equals" and "toString" methods are automatically generated.
*/
data class DataClassExample (val x: Int, val y: Int, val z: Int)
val fooData = DataClassExample(1, 2, 4)
println(fooData) // => DataClassExample(x=1, y=2, z=4)

// Data classes have a "copy" function.
val fooCopy = fooData.copy(y = 100)
println(fooCopy) // => DataClassExample(x=1, y=100, z=4)

// Objects can be destructured into multiple variables.
val (a, b, c) = fooCopy
println("$a $b $c") // => 1 100 4

// destructuring in "for" loop
for ((a, b, c) in listOf(fooData)) {
    println("$a $b $c") // => 1 100 4
}

val mapData = mapOf("a" to 1, "b" to 2)
// Map.Entry is destructurable as well
for ((key, value) in mapData) {
    println("$key -> $value")
}

// The "with" function is similar to the JavaScript "with" statement.
data class MutableDataClassExample (var x: Int, var y: Int, var z: Int)
val fooMutableData = MutableDataClassExample(7, 4, 9)
with (fooMutableData) {
    x -= 2
    y += 2
    z--
}
println(fooMutableData) // => MutableDataClassExample(x=5, y=6, z=8)

/*
We can create a list using the "listOf" function.
The list will be immutable - elements cannot be added or removed.
*/
val fooList = listOf("a", "b", "c")
println(fooList.size) // => 3
println(fooList.first()) // => a
println(fooList.last()) // => c
// Elements of a list can be accessed by their index.
println(fooList[1]) // => b

// A mutable list can be created using the "mutableListOf" function.
val fooMutableList = mutableListOf("a", "b", "c")
fooMutableList.add("d")
println(fooMutableList.last()) // => d
println(fooMutableList.size) // => 4

// We can create a set using the "setOf" function.
```



```
val fooSet = setOf("a", "b", "c")
println(fooSet.contains("a")) // => true
println(fooSet.contains("z")) // => false

// We can create a map using the "mapOf" function.
val fooMap = mapOf("a" to 8, "b" to 7, "c" to 9)
// Map values can be accessed by their key.
println(fooMap["a"]) // => 8

/*
Sequences represent lazily-evaluated collections.
We can create a sequence using the "generateSequence" function.
*/
val fooSequence = generateSequence(1, { it + 1 })
val x = fooSequence.take(10).toList()
println(x) // => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// An example of using a sequence to generate Fibonacci numbers:
fun fibonacciSequence(): Sequence<Long> {
    var a = 0L
    var b = 1L

    fun next(): Long {
        val result = a + b
        a = b
        b = result
        return a
    }

    return generateSequence(::next)
}
val y = fibonacciSequence().take(10).toList()
println(y) // => [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

// Kotlin provides higher-order functions for working with collections.
val z = (1..9).map {it * 3}
              .filter {it < 20}
              .groupBy {it % 2 == 0}
              .mapKeys {if (it.key) "even" else "odd"}
println(z) // => {odd=[3, 9, 15], even=[6, 12, 18]}

// A "for" loop can be used with anything that provides an iterator.
for (c in "hello") {
    println(c)
}

// "while" loops work in the same way as other languages.
var ctr = 0
while (ctr < 5) {
    println(ctr)
    ctr++
}
do {
    println(ctr)
    ctr++
} while (ctr < 10)

/*
"if" can be used as an expression that returns a value.
For this reason the ternary ?: operator is not needed in Kotlin.
*/
val num = 5
val message = if (num % 2 == 0) "even" else "odd"
println("$num is $message") // => 5 is odd

// "when" can be used as an alternative to "if-else if" chains.
val i = 10
when {
    i < 7 -> println("first block")
```

```
        fooString.startsWith("hello") -> println("second block")
        else -> println("else block")
    }

    // "when" can be used with an argument.
    when (i) {
        0, 21 -> println("0 or 21")
        in 1..20 -> println("in the range 1 to 20")
        else -> println("none of the above")
    }

    // "when" can be used as a function that returns a value.
    var result = when (i) {
        0, 21 -> "0 or 21"
        in 1..20 -> "in the range 1 to 20"
        else -> "none of the above"
    }
    println(result)

    /*
    We can check if an object is of a particular type by using the "is" operator
.
    If an object passes a type check then it can be used as that type without
    explicitly casting it.
    */
    fun smartCastExample(x: Any) : Boolean {
        if (x is Boolean) {
            // x is automatically cast to Boolean
            return x
        } else if (x is Int) {
            // x is automatically cast to Int
            return x > 0
        } else if (x is String) {
            // x is automatically cast to String
            return x.isNotEmpty()
        } else {
            return false
        }
    }
    println(smartCastExample("Hello, world!")) // => true
    println(smartCastExample("")) // => false
    println(smartCastExample(5)) // => true
    println(smartCastExample(0)) // => false
    println(smartCastExample(true)) // => true

    // Smartcast also works with when block
    fun smartCastWhenExample(x: Any) = when (x) {
        is Boolean -> x
        is Int -> x > 0
        is String -> x.isNotEmpty()
        else -> false
    }

    /*
    Extensions are a way to add new functionality to a class.
    This is similar to C# extension methods.
    */
    fun String.remove(c: Char): String {
        return this.filter {it != c}
    }
    println("Hello, world!".remove('l')) // => Heo, word!
}

// Enum classes are similar to Java enum types.
enum class EnumExample {
    A, B, C // Enum constants are separated with commas.
}
fun printEnum() = println(EnumExample.A) // => A
```



```
// Since each enum is an instance of the enum class, they can be initialized as:
enum class EnumExample(val value: Int) {
    A(value = 1),
    B(value = 2),
    C(value = 3)
}
fun printProperty() = println(EnumExample.A.value) // => 1

// Every enum has properties to obtain its name and ordinal(position) in the enu
m class declaration:
fun printName() = println(EnumExample.A.name) // => A
fun printPosition() = println(EnumExample.A.ordinal) // => 0

/*
The "object" keyword can be used to create singleton objects.
We cannot instantiate it but we can refer to its unique instance by its name.
This is similar to Scala singleton objects.
*/
object ObjectExample {
    fun hello(): String {
        return "hello"
    }

    override fun toString(): String {
        return "Hello, it's me, ${ObjectExample::class.simpleName}"
    }
}


fun useSingletonObject() {
    println(ObjectExample.hello()) // => hello
    // In Kotlin, "Any" is the root of the class hierarchy, just like "Object" i
s in Java
    val someRef: Any = ObjectExample
    println(someRef) // => Hello, it's me, ObjectExample
}


/* The not-null assertion operator (!!) converts any value to a non-null type an
d
throws an exception if the value is null.
*/
var b: String? = "abc"
val l = b!!.length

data class Counter(var value: Int) {
    // overload Counter += Int
    operator fun plusAssign(increment: Int) {
        this.value += increment
    }

    // overload Counter++ and ++Counter
    operator fun inc() = Counter(value + 1)

    // overload Counter + Counter
    operator fun plus(other: Counter) = Counter(this.value + other.value)

    // overload Counter * Counter
    operator fun times(other: Counter) = Counter(this.value * other.value)

    // overload Counter * Int
    operator fun times(value: Int) = Counter(this.value * value)

    // overload Counter in Counter
    operator fun contains(other: Counter) = other.value == this.value

    // overload Counter[Int] = Int
    operator fun set(index: Int, value: Int) {
        this.value = index + value
```

```
    }

    // overload Counter instance invocation
    operator fun invoke() = println("The value of the counter is $value")

}
/* You can also overload operators through an extension methods */
// overload -Counter
operator fun Counter.unaryMinus() = Counter(-this.value)

fun operatorOverloadingDemo() {
    var counter1 = Counter(0)
    var counter2 = Counter(5)
    counter1 += 7
    println(counter1) // => Counter(value=7)
    println(counter1 + counter2) // => Counter(value=12)
    println(counter1 * counter2) // => Counter(value=35)
    println(counter2 * 2) // => Counter(value=10)
    println(counter1 in Counter(5)) // => false
    println(counter1 in Counter(7)) // => true
    counter1[26] = 10
    println(counter1) // => Counter(value=36)
    counter1() // => The value of the counter is 36
    println(-counter2) // => Counter(value=-5)
}
```