Kotlin: Datos Básicos

Variables

All programs need to be able to store data, and variables help you to do just that. In Kotlin, you can declare:

- Read-only variables with `val`
- Mutable variables with `var`

To assign a value, use the assignment operator =

```
val popcorn = 5    // There are 5 boxes of popcorn
val hotdog = 7     // There are 7 hotdogs
var customers = 10 // There are 10 customers in
the queue

// Some customers leave the queue
customers = 8
println(customers)
// 8
```

As `customers` is a mutable variable, its value can be reassigned after declaration.

We recommend that you declare all variables as read-only (`val`) by default. Declare mutable variables (`var`) only if necessary.

String templates

It's useful to know how to print the contents of variables to standard output. You can do this with string templates. You can use template expressions to access data stored in variables and other objects, and convert them into strings. A string value is a sequence of characters in double quotes ". Template expressions always start with a dollar sign $

To evaluate a piece of code in a template expression, place the code within curly braces {} after the dollar sign $.

```
val customers = 10
println("There are $customers customers")
// There are 10 customers

println("There are ${customers + 1} customers")
// There are 11 customers
```

Basic types

Every variable and data structure in Kotlin has a type. Types are important because they tell the compiler what you are allowed to do with that variable or data structure. In other words, what functions and properties it has.

Kotlin was able to tell in the previous example that `customers` has type `Int`. Kotlin's ability to infer the type is called type inference. `customers` is assigned an integer value. From this, Kotlin infers that `customers` has a numerical type `Int`. As a result, the compiler knows that you can perform arithmetic operations with `customers`:

```
var customers = 10

// Some customers leave the queue
customers = 8

customers = customers + 3 // Example of addition: 11
customers += 7            // Example of addition: 18
customers -= 3            // Example of subtraction: 15
customers *= 2            // Example of multiplication: 30
customers /= 3            // Example of division: 10

println(customers) // 10
```

`+=`, `-=`, `*=`, `/=`, and `%=` are augmented assignment operators. For more information, see Augmented assignments.

Basic types

Category	Basic types	Example code
Integers	`Byte`, `Short`, `Int`, `Long`	`val year: Int = 2020`
Unsigned integers	`UByte`, `UShort`, `UInt`, `ULong`	`val score: UInt = 100u`
Floating-point numbers	`Float`, `Double`	`val currentTemp: Float = 24.5f`, `val price: Double = 19.99`
Booleans	`Boolean`	`val isEnabled: Boolean = true`
Characters	`Char`	`val separator: Char = ','`
Strings	`String`	`val message: String = "Hello, world!"`

Basic Types

To declare a variable without initializing it, specify its type with `:`. For example:

```
// Variable declared without initialization
val d: Int
// Variable initialized
d = 3

// Variable explicitly typed and initialized
val e: String = "hello"

// Variables can be read because they have been initialized
println(d) // 3
println(e) // hello
```

Collections

Collection type	Description
Lists	Ordered collections of items
Sets	Unique unordered collections of items
Maps	Sets of key-value pairs where keys are unique and map to only one value

Each collection type can be mutable or read only.

Collections: List

Lists store items in the order that they are added, and allow for duplicate items.

To create a read-only list (`List`), use the `listOf()` function.
To create a mutable list (`MutableList`), use the `mutableListOf()` function.

When creating lists, Kotlin can infer the type of items stored. To declare the type explicitly, add the type within angled brackets <> after the list declaration:

```
// Read only list
val readOnlyShapes = listOf("triangle", "square", "circle")
println(readOnlyShapes)
// [triangle, square, circle]

// Mutable list with explicit type declaration
val shapes: MutableList<String> = mutableListOf("triangle", "square", "circle")
println(shapes)
// [triangle, square, circle]
```

Collections:List

Lists are ordered so to access an item in a list, use the indexed access operator [ ]:

```
val readOnlyShapes = listOf("triangle", "square", "circle")

println("The first item in the list is: ${readOnlyShapes[0]}")

// The first item in the list is: triangle
```

To get the first or last item in a list, use `.first()` and `.last()` functions respectively:

```
val readOnlyShapes = listOf("triangle", "square", "circle")
println("The first item in the list is: ${readOnlyShapes.first()}")
// The first item in the list is: triangle
```

`.first()` and `.last()` functions are examples of extension functions. To call an extension function on an object, write the function name after the object appended with a period .

For more information about extension functions, see Extension functions. For the purposes of this tour, you only need to know how to call them.

Collections:List

To get the number of items in a list, use the `.count()` function:

```
val readOnlyShapes = listOf("triangle", "square", "circle")

println("This list has ${readOnlyShapes.count()} items")

// This list has 3 items
```

To check that an item is in a list, use the `in` operator:

```
val readOnlyShapes = listOf("triangle", "square", "circle")

println("circle" in readOnlyShapes)

// true
```

Collections:List

To add or remove items from a mutable list, use `.add()` and `.remove()` functions respectively:

```
val shapes: MutableList<String> = mutableListOf("triangle", "square", "circle")

// Add "pentagon" to the list

shapes.add("pentagon")

println(shapes)

// [triangle, square, circle, pentagon]

// Remove the first "pentagon" from the list

shapes.remove("pentagon")

println(shapes)

// [triangle, square, circle]
```

Collections: Set

Whereas lists are ordered and allow duplicate items, sets are unordered and only store unique items.

- To create a read-only set (`Set`), use the `setOf()` function.
- To create a mutable set (`MutableSet`), use the `mutableSetOf()` function.

When creating sets, Kotlin can infer the type of items stored. To declare the type explicitly, add the type within angled brackets `<>` after the set declaration:

```
// Read-only set
val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")
// Mutable set with explicit type declaration
val fruit: MutableSet<String> = mutableSetOf("apple", "banana", "cherry", "cherry")
println(readOnlyFruit)
// [apple, banana, cherry]
```

Collections:Set

To get the number of items in a set, use the `.count()` function:

```
val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")

println("This set has ${readOnlyFruit.count()} items")

// This set has 3 items
```

To check that an item is in a set, use the `in` operator:

```
val readOnlyFruit = setOf("apple", "banana", "cherry", "cherry")

println("banana" in readOnlyFruit)

// true
```

To add or remove items from a mutable set, use `.add()` and `.remove()` functions respectively:

Collections:Set

To add or remove items from a mutable set, use `.add()` and `.remove()` functions respectively:

```
val fruit: MutableSet<String> = mutableSetOf("apple", "banana", "cherry", "cherry")
fruit.add("dragonfruit")    // Add "dragonfruit" to the set
println(fruit)              // [apple, banana, cherry, dragonfruit]
fruit.remove("dragonfruit") // Remove "dragonfruit" from the set
println(fruit)              // [apple, banana, cherry]
```

Collections:Map

Maps store items as key-value pairs. You access the value by referencing the key. You can imagine a map like a food menu. You can find the price (value), by finding the food (key) you want to eat. Maps are useful if you want to look up a value without using a numbered index, like in a list.

- To create a read-only map (`Map`), use the `mapOf()` function.
- To create a mutable map (`MutableMap`), use the `mutableMapOf()` function.

When creating maps, Kotlin can infer the type of items stored. To declare the type explicitly, add the types of the keys and values within angled brackets `<>` after the map declaration. For example: `MutableMap<String, Int>`. The keys have type `String` and the values have type `Int`.

Collections:Map

You can also use the indexed access operator `[ ]` to add items to a mutable map:

```
val juiceMenu: MutableMap<String, Int> = mutableMapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
juiceMenu["coconut"] = 150 // Add key "coconut" with value 150 to the map
println(juiceMenu)
// {apple=100, kiwi=190, orange=100, coconut=150}
```

To remove items from a mutable map, use the `.remove()` function:

```
val juiceMenu: MutableMap<String, Int> = mutableMapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
juiceMenu.remove("orange")    // Remove key "orange" from the map
println(juiceMenu)
// {apple=100, kiwi=190}
```

Collections:Map

The easiest way to create maps is to use `to` between each key and its related value:

```
// Read-only map
val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
println(readOnlyJuiceMenu)
// {apple=100, kiwi=190, orange=100}

// Mutable map with explicit type declaration
val juiceMenu: MutableMap<String, Int> = mutableMapOf("apple" to 100, "kiwi" to
190, "orange" to 100)
println(juiceMenu)
// {apple=100, kiwi=190, orange=100}
```

Collections:Map

To get the number of items in a map, use the `.count()` function:

```
// Read-only map
val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
println("This map has ${readOnlyJuiceMenu.count()} key-value pairs")
// This map has 3 key-value pairs
```

To check if a specific key is already included in a map, use the `.containsKey()` function:

```
val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
println(readOnlyJuiceMenu.containsKey("kiwi"))
// true
```

Collections:Map

To obtain a collection of the keys or values of a map, use the `keys` and `values` properties respectively:

```
val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
println(readOnlyJuiceMenu.keys)
// [apple, kiwi, orange]
println(readOnlyJuiceMenu.values)
// [100, 190, 100]
```

To check that a key or value is in a map, use the `in` operator:

```
val readOnlyJuiceMenu = mapOf("apple" to 100, "kiwi" to 190, "orange" to 100)
println("orange" in readOnlyJuiceMenu.keys)// true
// Alternatively, you don't need to use the keys property
println("orange" in readOnlyJuiceMenu)// true
println(200 in readOnlyJuiceMenu.values)
// false
```

Conditional expressions

Kotlin provides `if` and `when` for checking conditional expressions.

If you have to choose between `if` and `when`, we recommend using `when` because it:

- Makes your code easier to read.
- Makes it easier to add another branch.
- Leads to fewer mistakes in your code.

If

To use `if`, add the conditional expression within parentheses `( )` and the action to take if the result is true within curly braces `{}`.
There is no ternary operator `condition ? then : else` in Kotlin. Instead, `if` can be used as an expression. If there is only one line of code per action, the curly braces `{}` are optional:

```
val a = 1
val b = 2
println(if (a > b) a else b) // Returns a value: 2
```

When

Use `when` when you have a conditional expression with multiple branches.

To use `when`:

- Place the value you want to evaluate within parentheses `()`.
- Place the branches within curly braces `{}`.
- Use `->` in each branch to separate each check from the action to take if the check is successful.

`when` can be used either as a statement or as an expression. A statement doesn't return anything but performs actions instead.

```
val obj = "Hello"
when (obj) {
    // Checks whether obj equals to "1"
    "1" -> println("One")
    // Checks whether obj equals to "Hello"
    "Hello" -> println("Greeting")
    // Default statement
    else -> println("Unknown")
}
// Greeting
```

Note that all branch conditions are checked sequentially until one of them is satisfied. So only the first suitable branch is executed.

An expression returns a value that can be used later in your code. Here is an example of using `when` as an expression. The `when` expression is assigned immediately to a variable which is later used with the `println()` function:

```
fun main() {
    val trafficLightState = "Red" // This can be
"Green", "Yellow", or "Red"
    val trafficAction = when {
        trafficLightState == "Green" -> "Go"
        trafficLightState == "Yellow" -> "Slow down"
        trafficLightState == "Red" -> "Stop"
        else -> "Malfunction"
    }
    println(trafficAction)
    // Stop
```

The examples of `when` that you've seen so far both had a subject: `obj`. But `when` can also be used without a subject.

This example uses a `when` expression without a subject to check a chain of Boolean expressions:

if and when

```
fun main() {
    val trafficLightState = "Red" // This can be "Green", "Yellow", or "Red"

    val trafficAction = when (trafficLightState) {
        "Green" -> "Go"
        "Yellow" -> "Slow down"
        "Red" -> "Stop"
        else -> "Malfunction"
    }

    println(trafficAction)
    // Stop
}
```

However, you can have the same code but with `trafficLightState` as the subject

Using `when` with a subject makes your code easier to read and maintain. When you use a subject with a `when` expression, it also helps Kotlin check that all possible cases are covered. Otherwise, if you don't use a subject with a `when` expression, you need to provide an else branch.

RANGES

Before talking about loops, it's useful to know how to construct ranges for loops to iterate over.

The most common way to create a range in Kotlin is to use the `..` operator. For example, `1..4` is equivalent to `1, 2, 3, 4`.

To declare a range that doesn't include the end value, use the `..<` operator. For example, `1..<4` is equivalent to `1, 2, 3`.

To declare a range in reverse order, use `downTo`. For example, `4 downTo 1` is equivalent to `4, 3, 2, 1`.

To declare a range that increments in a step that isn't 1, use `step` and your desired increment value. For example, `1..5 step 2` is equivalent to `1, 3, 5`.

You can also do the same with `Char` ranges:

`'a'..'d'` is equivalent to `'a', 'b', 'c', 'd'`
`'z' downTo 's' step 2` is equivalent to `'z', 'x', 'v', 't'`

Loop

The two most common loop structures in programming are `for` and `while`. Use `for` to iterate over a range of values and perform an action. Use `while` to continue an action until a particular condition is satisfied.

For
Using your new knowledge of ranges, you can create a `for` loop that iterates over numbers 1 to 5 and prints the number each time.
Place the iterator and range within parentheses `( )` with keyword `in`. Add the action you want to complete within curly braces `{ }`:

```
for (number in 1..5) {
    // number is the iterator and 1..5 is the range
    print(number)
}
// 12345


val cakes = listOf("carrot", "cheese", "chocolate")

for (cake in cakes) {
    println("Yummy, it's a $cake cake!")
}
```

Loop

While

`while` can be used in two ways:

To execute a code block while a conditional expression is true. (`while`)

To execute the code block first and then check the conditional expression. (`do-while`)

In the first use case (`while`):

Declare the conditional expression for your while loop to continue within parentheses `( )`.

Add the action you want to complete within curly braces `{}`.

```
var cakesEaten = 0

while (cakesEaten < 3) {

    println("Eat a cake")

    cakesEaten++

}
```

In the second use case (`do-while`):

- Declare the conditional expression for your while loop to continue within parentheses `( )`.
- Define the action you want to complete within curly braces `{}` with the keyword `do`.
-

```
var cakesEaten = 0
var cakesBaked = 0
while (cakesEaten < 3) {
    println("Eat a cake")
    cakesEaten++
}
do {
    println("Bake a cake")
    cakesBaked++
} while (cakesBaked < cakesEaten)
```

Functions

In Kotlin:

- Function parameters are written within parentheses `()`.
- Each parameter must have a type, and multiple parameters must be separated by commas `,`.
- The return type is written after the function's parentheses `()`, separated by a colon `:`.
- The body of a function is written within curly braces `{}`.
- The `return` keyword is used to exit or return something from a function.

Functions

```
fun sum(x: Int, y: Int): Int {
    return x + y
}
fun main() {
    println(sum(1, 2))
    // 3
}
```

- `x` and `y` are function parameters.
- `x` and `y` have type `Int`.
- The function's return type is `Int`.
- The function returns a sum of `x` and `y` when called.

Functions

Named arguments
For concise code, when calling your function, you don't have to include parameter names.

However, including parameter names does make your code easier to read. This is called using named arguments.

If you do include parameter names, then you can write the parameters in any order.

```
fun printMessageWithPrefix(message:
String, prefix: String) {
    println("[$prefix] $message")
}
fun main() {
    // Uses named arguments with
swapped parameter order
    printMessageWithPrefix(prefix = "Log",
message = "Hello")
    // [Log] Hello
}
```

Functions

Default parameter values
You can define default values for your function parameters.

Any parameter with a default value can be omitted when calling your function.

To declare a default value, use the assignment operator = after the type:

```
fun printMessageWithPrefix(message: String,
prefix: String = "Info") {
    println("[$prefix] $message")
}
fun main() {
    // Function called with both parameters
    printMessageWithPrefix("Hello", "Log")
    // [Log] Hello
    // Function called only with message
parameter    printMessageWithPrefix("Hello")
    // [Info] Hello
    printMessageWithPrefix(prefix = "Log",
message = "Hello")
    // [Log] Hello
```

Functions

Functions without return

If your function doesn't return a useful value then its return type is `Unit`. `Unit` is a type with only one value – `Unit`.

You don't have to declare that `Unit` is returned explicitly in your function body. This means that you don't have to use the `return` keyword or declare a return type:

```
fun printMessage(message: String) {
    println(message)
    // `return Unit` or `return` is optional
}

fun main() {
    printMessage("Hello")
    // Hello
}
```

Functions

Single-expression functions

To make your code more concise, you can use single-expression functions. For example, the `sum()` function can be shortened:

You can remove the curly braces `{}` and declare the function body using the assignment operator =. When you use the assignment operator =, Kotlin uses type inference, so you can also omit the return type. The `sum()` function then becomes one line:

```
fun sum(x: Int, y: Int): Int {
    return x + y
}

fun sum(x;Int, y:Int) = x + y

fun main() {
    println(sum(1, 2))
    // 3
}
```

Functions

Early returns in functions

To stop the code in your function from being processed further than a certain point, use the `return` keyword.

This example uses `if` to return from a function early if the conditional expression is found to be true:

```
// A list of registered usernames
val registeredUsernames =
mutableListOf("john_doe", "jane_smith")

// A list of registered emails
val registeredEmails =
mutableListOf("john@example.com",
"jane@example.com")

fun registerUser(username: String, email:
String): String {
    // Early return if the username is already
taken
    if (username in registeredUsernames) {
        return "Username already taken.
Please choose a different username."
    }
```

Functions: Lambda

```
fun main() {

    val upperCaseString = {
text: String -> text.uppercase()
}


println(upperCaseString("hello
"))

    // HELLO

}
```

Lambda expressions can be hard to understand at first glance so let's break it down. Lambda expressions are written within curly braces `{}`.

Within the lambda expression, you write:

- The parameters followed by an `->`.
- The function body after the `->`.

In the previous example:

- `text` is a function parameter.
- `text` has type `String`.
- The function returns the result of the `.uppercase()` function called on `text`.
- The entire lambda expression is assigned to the `upperCaseString` variable with the assignment operator `=`.
- The lambda expression is called by using the variable `upperCaseString` like a function and the string `"hello"` as a parameter.
- The `println()` function prints the result.

Functions:Lambda

Lambda expressions can be used in a number of ways. You can:

- Pass a lambda expression as a parameter to another function
- Return a lambda expression from a function
- Invoke a lambda expression on its own

```
val numbers = listOf(1, -2, 3, -4, 5, -6)


val positives = numbers.filter ({ x -> x > 0 })

val isNegative = { x: Int -> x < 0 }
val negatives = numbers.filter(isNegative)

println(positives)
// [1, 3, 5]
println(negatives)
// [-2, -4, -6]
```

Functions: Lambda

A great example of when it is useful to pass a lambda expression to a function, is using the `.filter()` function on collections:

```
val numbers = listOf(1, -2, 3, -4, 5, -6)


val positives = numbers.filter ({ x -> x > 0 })

val isNegative = { x: Int -> x < 0 }
val negatives = numbers.filter(isNegative)

println(positives)
// [1, 3, 5]
println(negatives)
// [-2, -4, -6]
```

Functions: Lambda

In the following example, the `toSeconds()` function has function type `(Int) -> Int` because it always returns a lambda expression that takes a parameter of type `Int` and returns an `Int` value.

This example uses a when expression to determine which lambda expression is returned when `toSeconds()` is called:

```
fun toSeconds(time: String): (Int) -> Int = when (time) {
    "hour" -> { value -> value * 60 * 60 }
    "minute" -> { value -> value * 60 }
    "second" -> { value -> value }
    else -> { value -> value }
}

fun main() {
    val timesInMinutes = listOf(2, 10, 15, 1)
    val min2sec = toSeconds("minute")
    val totalTimeInSeconds = timesInMinutes.map(min2sec).sum()
    println("Total time is $totalTimeInSeconds secs")
    // Total time is 1680 secs
}
```

Functions: Lambda

Lambda expressions can be invoked on their own by adding parentheses ( ) after the curly braces { } and including any parameters within the parentheses

```
println({ text: String -> text.uppercase() }("hello"))
// HELLO
```

Classes

To declare a class, use the `class` keyword: Class Customer

Properties

Characteristics of a class's object can be declared in properties. You can declare properties for a class:

- Within parentheses ( ) after the class name.

```
class Contact(val id: Int, var email: String)
```

- Within the class body defined by curly braces { }.

```
class Contact(val id: Int, var email: String) {
    val category: String = ""
}
```

Classes

We recommend that you declare properties as read-only (`val`) unless they need to be changed after an instance of the class is created.
You can declare properties without `val` or `var` within parentheses but these properties are not accessible after an instance has been created.

- The content contained within parentheses `()` is called the **class header**.
- You can use a trailing comma when declaring class properties.

Just like with function parameters, class properties can have default values:

```
class Contact(val id: Int, var email: String = "example@gmail.com") {
    val category: String = "work"
}
```

Classes

Create instance

To create an object from a class, you declare a class instance using a constructor.
By default, Kotlin automatically creates a constructor with the parameters declared in the class header.

```
class Contact(val id: Int, var email: String)

fun main() {
    val contact = Contact(1, "mary@gmail.com")
}
```

Access properties

To access a property of an instance, write the name of the property after the instance name appended with a period `.`:

```
class Contact(val id: Int, var email: String)

fun main() {
    val contact = Contact(1, "mary@gmail.com")

    // Prints the value of the property: email
    println(contact.email)
    // mary@gmail.com

    // Updates the value of the property: email
    contact.email = "jane@gmail.com"

    // Prints the new value of the property: email
    println(contact.email)
    // jane@gmail.com
}
```

To concatenate the value of a property as part of a string, you can use string templates ($). For example:

```
println("Their email
address is:
${contact.email}")
```

Classes

Data classes

Kotlin has data classes which are particularly useful for storing data. Data classes have the same functionality as classes, but they come automatically with additional member functions. These member functions allow you to easily print the instance to readable output, compare instances of a class, copy instances, and more.

As these functions are automatically available, you don't have to spend time writing the same boilerplate code for each of your classes.
To declare a data class, use the keyword `data`:

```
data class User(val name:
String, val id: Int)
```

The most useful predefined member functions of data classes are:

Function	Description
`toString()`	Prints a readable string of the class instance and its properties.
`equals()` or `==`	Compares instances of a class.
`copy()`	Creates a class instance by copying another, potentially with some different properties.

Classes

Print as string
To print a readable string of a class instance, you can explicitly call the `toString()` function, or use print functions (`println()` and `print()`) which automatically call `toString()` for you:

Compare instances
To compare data class instances, use the equality operator ==:

Copy instance
To create an exact copy of a data class instance, call the `copy()` function on the instance.

To create a copy of a data class instance and change some properties, call the `copy()` function on the instance and add replacement values for properties as function parameters.

```
val user = User("Alex", 1)

// Automatically uses toString() function so that output is easy
println(user)
// User(name=Alex, id=1)
```

```
val user = User("Alex", 1)
val secondUser = User("Alex", 1)
val thirdUser = User("Max", 2)

// Compares user to second user
println("user == secondUser: ${user == secondUser}")
// user == secondUser: true

// Compares user to third user
println("user == thirdUser: ${user == thirdUser}")
// user == thirdUser: false
```

```
val user = User("Alex", 1)

// Creates an exact copy of user
println(user.copy())
// User(name=Alex, id=1)

// Creates a copy of user with name: "Max"
println(user.copy("Max"))
// User(name=Max, id=1)

// Creates a copy of user with id: 3
println(user.copy(id = 3))
// User(name=Alex, id=3)
```