**Find the Length of a String**

You can find the length of a String value by writing .length after the string variable or string literal.

console.log("Alan Peter".length);

The value 10 would be displayed in the console. Note that the space character between "Alan" and "Peter" is also counted.

For example, if we created a variable const firstName = "Ada", we could find out how long the string Ada is by using the firstName.length property.

Use the .length property to set lastNameLength to the number of characters in lastName.

1 // Setup

2 let lastNameLength = 0;

3 const lastName = "Lovelace";

4

5 // Only change code below this line

6 lastNameLength = lastName;//lastNameLength = 8;

7 console.log(lastNameLength.length);

**Console**

8

Answer:

// Only change code below this line

lastNameLength = lastName.length;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Use Bracket Notation to Find the First Character in a String**

*Bracket notation* is a way to get a character at a specific index within a string.

Most modern programming languages, like JavaScript, don't start counting at 1 like humans do. They start at 0. This is referred to as *Zero-based* indexing.

For example, the character at index 0 in the word Charles is C. So if const firstName = "Charles", you can get the value of the first letter of the string by using firstName[0].

Example:

const firstName = "Charles";

const firstLetter = firstName[0];

firstLetter would have a value of the string C.

Use bracket notation to find the first character in the lastName variable and assign it to firstLetterOfLastName.

**Hint:** Try looking at the example above if you get stuck.

## Tests

* Waiting:The firstLetterOfLastName variable should have the value of L.
* Waiting:You should use bracket notation.
* // Setup
* let firstLetterOfLastName = "";
* const lastName = "Lovelace";
* // Only change code below this line
* firstLetterOfLastName = lastName; // Change this line

Answer

firstLetterOfLastName = lastName[0]; // Change this line

# Understand String Immutability

In JavaScript, String values are immutable, which means that they cannot be altered once created.

For example, the following code will produce an error because the letter B in the string Bob cannot be changed to the letter J:

let myStr = "Bob";

myStr[0] = "J";

Note that this does not mean that myStr could not be re-assigned. The only way to change myStr would be to assign it with a new value, like this:

let myStr = "Bob";

myStr = "Job";

## Tests

Correct the assignment to myStr so it contains the string value of Hello World using the approach shown in the example above.

* Waiting:myStr should have a value of the string Hello World.
* Waiting:You should not change the code above the specified comment.

// Setup

let myStr = "Jello World";

// Only change code below this line

myStr[0] = "H"; // Change this line

// Only change code above this line

Answer:

myStr = "Hello World";

**Use Bracket Notation to Find the Nth Character in a String**

You can also use *bracket notation* to get the character at other positions within a string.

Remember that computers start counting at 0, so the first character is actually the zeroth character.

Example:

const firstName = "Ada";

const secondLetterOfFirstName = firstName[1];

secondLetterOfFirstName would have a value of the string d.

Let's try to set thirdLetterOfLastName to equal the third letter of the lastName variable using bracket notation.

**Hint:** Try looking at the example above if you get stuck.

## Tests

* Failed:The thirdLetterOfLastName variable should have the value of v.
* Failed:You should use bracket notation.
* // Setup
* const lastName = "Lovelace";
* // Only change code below this line
* const thirdLetterOfLastName = lastName; // Change this line

const thirdLetterOfLastName = lastName[2]

# Use Bracket Notation to Find the Last Character in a String

In order to get the last letter of a string, you can subtract one from the string's length.

For example, if const firstName = "Ada", you can get the value of the last letter of the string by using firstName[firstName.length - 1].

Example:

const firstName = "Ada";

const lastLetter = firstName[firstName.length - 1];

lastLetter would have a value of the string a.

Use bracket notation to find the last character in the lastName variable.

**Hint:** Try looking at the example above if you get stuck.

Run the Tests (Ctrl + Enter)

Reset this lesson

Get Help

## Tests

* Waiting:lastLetterOfLastName should be the letter e.
* Waiting:You should use .length to get the last letter.

// Setup

const lastName = "Lovelace";

// Only change code below this line

const lastLetterOfLastName = lastName; // Change this line

Answer:

const lastLetterOfLastName = lastName[lastName.length-1];

# Use Bracket Notation to Find the Nth-to-Last Character in a String

You can use the same principle we just used to retrieve the last character in a string to retrieve the Nth-to-last character.

For example, you can get the value of the third-to-last letter of the const firstName = "Augusta" string by using firstName[firstName.length - 3]

Example:

const firstName = "Augusta";

const thirdToLastLetter = firstName[firstName.length - 3];

thirdToLastLetter would have a value of the string s.

## Tests

* Waiting:secondToLastLetterOfLastName should be the letter c.
* Waiting:You should use .length to get the second last letter.

// Setup

const lastName = "Lovelace";

// Only change code below this line

const secondToLastLetterOfLastName = lastName; // Change this line

Answer:

const secondToLastLetterOfLastName = lastName[lastName.length-2];

**Word Blanks**

You are provided sentences with some missing words, like nouns, verbs, adjectives and adverbs. You then fill in the missing pieces with words of your choice in a way that the completed sentence makes sense.

Consider this sentence:

It was really \_\_\_\_, and we \_\_\_\_ ourselves \_\_\_\_.

This sentence has three missing pieces- an adjective, a verb and an adverb, and we can add words of our choice to complete it. We can then assign the completed sentence to a variable as follows:

const sentence = "It was really " + "hot" + ", and we " + "laughed" + " ourselves " + "silly" + ".";

In this challenge, we provide you with a noun, a verb, an adjective and an adverb. You need to form a complete sentence using words of your choice, along with the words we provide.

You will need to use the string concatenation operator + to build a new string, using the provided variables: myNoun, myAdjective, myVerb, and myAdverb. You will then assign the formed string to the wordBlanks variable. You should not change the words assigned to the variables.

You will also need to account for spaces in your string, so that the final sentence has spaces between all the words. The result should be a complete sentence.

## Tests

* Waiting:wordBlanks should be a string.
* Waiting:You should not change the values assigned to myNoun, myVerb, myAdjective or myAdverb.
* Waiting:You should not directly use the values dog, ran, big, or quickly to create wordBlanks.
* Waiting:wordBlanks should contain all of the words assigned to the variables myNoun, myVerb, myAdjective and myAdverb separated by non-word characters (and any additional words of your choice).

const myNoun = "dog";

const myAdjective = "big";

const myVerb = "ran";

const myAdverb = "quickly";

// Only change code below this line

const wordBlanks = ""; // Change this line

// Only change code above this line

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Answer:

const wordBlanks = "This" + myNoun + "is vety" + myAdjective + ", however it" + myVerb + "race" + myAdverb+"."; // Change this line

### ES6

ECMAScript, or ES, is a standardized version of JavaScript. Because all major browsers follow this specification, the terms ECMAScript and JavaScript are interchangeable.

Most of the JavaScript you've learned up to this point was in ES5 (ECMAScript 5), which was finalized in 2009. While you can still write programs in ES5, JavaScript is constantly evolving, and new features are released every year.

ES6, released in 2015, added many powerful new features to the language. In this course, you'll learn these new features, including arrow functions, destructuring, classes, promises, and modules.

[Edit this page](https://github.com/Umuzi-org/ACN-syllabustopics/kotlin/basic-syntax-types/_index.md)

[African Coding Network Syllabus](http://syllabus.africacode.net/) > [Topics](http://syllabus.africacode.net/topics/) > [Android-Kotlin](http://syllabus.africacode.net/topics/kotlin/) > Basic Syntax & Types

# TOPIC: BASIC SYNTAX & TYPES

|  |
| --- |
| Hard Prerequisites |
| IMPORTANT: Please review these prerequisites, they include important information that will help you with this content. |
|  [TOPICS: Set Up](http://syllabus.africacode.net/topics/kotlin/set-up) |

### [**var us val:**](https://www.youtube.com/watch?v=Nz-lMqxfUUs)

* **val and var** both are used to declare a variable.
* **var** is like general variable and it's known as a **mutable** variable in **Kotlin** and can be assigned multiple times.
* **val** is like Final variable and it's known as **immutable** in **Kotlin** and can be initialized only single time., after it become read only. The IllegalAccessorError will occur when you try to reassign the value.
* You can enforce a type called [strongtyping](https://whatis.techtarget.com/definition/strongly-typed). This is the opposite of statically typed.

Syntaxt : var book:String = "Maths" // This should only be used when necessary.

### **Numbers:**

* Kotlin handles numbers in a way close to Java, but not exactly the same.
* Kotlin provides the following built in types representing numbers (this is close to Java):

| **Type** | **Bit**[**Width**](https://www.youtube.com/watch?v=_SkpnG571z8) |
| --- | --- |
| Double | 64 |
| Float | 32 |
| Long | 64 |
| Int(Default data type in Kotlin) | 32 |
| Short | 16 |
| Byte | 8 |

### **Characters:**

* Note that characters are not numbers in Kotlin.
* Characters are represented by the type Char.
* They cannot be treated directly as number.
* [In Java](https://www.youtube.com/watch?v=LBQrD2nkKQg) they are stored as numbers internally.
  + <https://www.youtube.com/watch?v=LBQrD2nkKQg>

Fun check(c: Char){

If (c == 1) {

// Error : incomaptable types will occur

}

}

### **Booleans:**

The type Boolean represents booleans, has a **true** or **false** value.

**Arrays:**

Arrays in Kotlin are represented by the array class, that has get and set functions (that turn into [] by operator overloading conventions), and size property, along with a few other useful member functions:

Class Array private constructor() {

val size: int

operator fun get (index : Int) T

operator fun set(index : Int, value: T ) : Unit

}

### **Strings**

Strings represented by the type String. Strings are immutable.

They are immutable in nature.

Should be written in double quotes.

Elements of a string are characters that can be accessed by the indexing operation:

s[I]

A string can be iterated over with a for loop:

for (c in str){

prinln(c)

}

### **Defining packages**

Package specification should be at the top of the source file:

package my.demo

import java.util.\*

// ...

It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

### **Defining functions**

Function having two Int parameters with Int return type:

//sampleStart

fun sum(a: Int, b: Int): Int {

return a + b

}

//sampleEnd

fun main(args: Array<String>) {

print("sum of 3 and 5 is ")

println(sum(3, 5))

}

Function with an expression body and inferred return type:

//sampleStart

fun sum(a: Int, b: Int) = a + b

//sampleEnd

fun main(args: Array<String>) {

println("sum of 19 and 23 is ${sum(19, 23)}")

}

Function returning no meaningful value:

//sampleStart

fun printSum(a: Int, b: Int): Unit {

println("sum of $a and $b is ${a + b}")

}

//sampleEnd

fun main(args: Array<String>) {

printSum(-1, 8)

}

Unit return type can be omitted:

//sampleStart

fun printSum(a: Int, b: Int) {

println("sum of $a and $b is ${a + b}")

}

//sampleEnd

fun main(args: Array<String>) {

printSum(-1, 8)

}

### **Defining local variables**

Assign-once (read-only) local variable:

fun main(args: Array<String>) {

//sampleStart

val a: Int = 1 // immediate assignment

val b = 2 // `Int` type is inferred

val c: Int // Type required when no initializer is provided

c = 3 // deferred assignment

//sampleEnd

println("a = $a, b = $b, c = $c")

}

Mutable variable:

fun main(args: Array<String>) {

//sampleStart

var x = 5 // `Int` type is inferred

x += 1

//sampleEnd

println("x = $x")

}

### **Comments**

Just like Java and JavaScript, Kotlin supports end-of-line and block comments.

// This is an end-of-line comment

/\* This is a block comment

on multiple lines. \*/

Unlike Java, block comments in Kotlin can be nested.

### **Using string templates**

fun main(args: Array<String>) {

//sampleStart

var a = 1

// simple name in template:

val s1 = "a is $a"

a = 2

// arbitrary expression in template:

val s2 = "${s1.replace("is", "was")}, but now is $a"

//sampleEnd

println(s2)

}

### **Using conditional expressions**

//sampleStart

fun maxOf(a: Int, b: Int): Int {

if (a > b) {

return a

} else {

return b

}

}

//sampleEnd

fun main(args: Array<String>) {

println("max of 0 and 42 is ${maxOf(0, 42)}")

}

Using if as an expression:

//sampleStart

fun maxOf(a: Int, b: Int) = if (a > b) a else b

//sampleEnd

fun main(args: Array<String>) {

println("max of 0 and 42 is ${maxOf(0, 42)}")

}

### **Using nullable values and checking for null**

A reference must be explicitly marked as nullable when null value is possible.

Return null if str does not hold an integer:

fun parseInt(str: String): Int? {

// ...

}

Use a function returning nullable value:

fun parseInt(str: String): Int? {

return str.toIntOrNull()

}

//sampleStart

fun printProduct(arg1: String, arg2: String) {

val x = parseInt(arg1)

val y = parseInt(arg2)

// Using `x \* y` yields error because they may hold nulls.

if (x != null && y != null) {

// x and y are automatically cast to non-nullable after null check

println(x \* y)

}

else {

println("either '$arg1' or '$arg2' is not a number")

}

}

//sampleEnd

fun main(args: Array<String>) {

printProduct("6", "7")

printProduct("a", "7")

printProduct("a", "b")

}

or

fun parseInt(str: String): Int? {

return str.toIntOrNull()

}

fun printProduct(arg1: String, arg2: String) {

val x = parseInt(arg1)

val y = parseInt(arg2)

//sampleStart

// ...

if (x == null) {

println("Wrong number format in arg1: '${arg1}'")

return

}

if (y == null) {

println("Wrong number format in arg2: '${arg2}'")

return

}

// x and y are automatically cast to non-nullable after null check

println(x \* y)

//sampleEnd

}

fun main(args: Array<String>) {

printProduct("6", "7")

printProduct("a", "7")

printProduct("99", "b")

}

### **Using type checks and automatic casts**

The is operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there’s no need to cast it explicitly:

//sampleStart

fun getStringLength(obj: Any): Int? {

if (obj is String) {

// `obj` is automatically cast to `String` in this branch

return obj.length

}

// `obj` is still of type `Any` outside of the type-checked branch

return null

}

//sampleEnd

fun main(args: Array<String>) {

fun printLength(obj: Any) {

println("'$obj' string length is ${getStringLength(obj) ?: "... err, not a string"} ")

}

printLength("Incomprehensibilities")

printLength(1000)

printLength(listOf(Any()))

}

or

//sampleStart

fun getStringLength(obj: Any): Int? {

if (obj !is String) return null

// `obj` is automatically cast to `String` in this branch

return obj.length

}

//sampleEnd

fun main(args: Array<String>) {

fun printLength(obj: Any) {

println("'$obj' string length is ${getStringLength(obj) ?: "... err, not a string"} ")

}

printLength("Incomprehensibilities")

printLength(1000)

printLength(listOf(Any()))

}

or even

//sampleStart

fun getStringLength(obj: Any): Int? {

// `obj` is automatically cast to `String` on the right-hand side of `&&`

if (obj is String && obj.length > 0) {

return obj.length

}

return null

}

//sampleEnd

fun main(args: Array<String>) {

fun printLength(obj: Any) {

println("'$obj' string length is ${getStringLength(obj) ?: "... err, is empty or not a string at all"} ")

}

printLength("Incomprehensibilities")

printLength("")

printLength(1000)

}

### **Using a for loop**

fun main(args: Array<String>) {

//sampleStart

val items = listOf("apple", "banana", "kiwi")

for (item in items) {

println(item)

}

//sampleEnd

}

or

fun main(args: Array<String>) {

//sampleStart

val items = listOf("apple", "banana", "kiwi")

for (index in items.indices) {

println("item at $index is ${items[index]}")

}

//sampleEnd

}

### **Using a while loop**

fun main(args: Array<String>) {

//sampleStart

val items = listOf("apple", "banana", "kiwi")

var index = 0

while (index < items.size) {

println("item at $index is ${items[index]}")

index++

}

//sampleEnd

}

### **Using when expression**

//sampleStart

fun describe(obj: Any): String =

when (obj) {

1 -> "One"

"Hello" -> "Greeting"

is Long -> "Long"

!is String -> "Not a string"

else -> "Unknown"

}

//sampleEnd

fun main(args: Array<String>) {

println(describe(1))

println(describe("Hello"))

println(describe(1000L))

println(describe(2))

println(describe("other"))

}

### **Using ranges**

Check if a number is within a range using in operator:

fun main(args: Array<String>) {

//sampleStart

val x = 10

val y = 9

if (x in 1..y+1) {

println("fits in range")

}

//sampleEnd

}

Check if a number is out of range:

fun main(args: Array<String>) {

//sampleStart

val list = listOf("a", "b", "c")

if (-1 !in 0..list.lastIndex) {

println("-1 is out of range")

}

if (list.size !in list.indices) {

println("list size is out of valid list indices range too")

}

//sampleEnd

}

Iterating over a range:

fun main(args: Array<String>) {

//sampleStart

for (x in 1..5) {

print(x)

}

//sampleEnd

}

or over a progression:

fun main(args: Array<String>) {

//sampleStart

for (x in 1..10 step 2) {

print(x)

}

for (x in 9 downTo 0 step 3) {

print(x)

}

//sampleEnd

}

### **Using collections**

Iterating over a collection:

fun main(args: Array<String>) {

val items = listOf("apple", "banana", "kiwi")

//sampleStart

for (item in items) {

println(item)

}

//sampleEnd

}

Checking if a collection contains an object using in operator:

fun main(args: Array<String>) {

val items = setOf("apple", "banana", "kiwi")

//sampleStart

when {

"orange" in items -> println("juicy")

"apple" in items -> println("apple is fine too")

}

//sampleEnd

}

Using lambda expressions to filter and map collections:

fun main(args: Array<String>) {

val fruits = listOf("banana", "avocado", "apple", "kiwi")

//sampleStart

fruits

.filter { it.startsWith("a") }

.sortedBy { it }

.map { it.toUpperCase() }

.forEach { println(it) }

//sampleEnd

}

# TOPIC: CLASSES

|  |
| --- |
| Hard Prerequisites |
| IMPORTANT: Please review these prerequisites, they include important information that will help you with this content. |
|  [TOPICS: Basic Control Flow](http://syllabus.africacode.net/topics/kotlin/basic-control-flow) |

Classes in Kotlin are declared using the keyword class:

class Invoice {

}

### **Constructors**

A class in Kotlin can have a primary constructor and one or more secondary constructors.

class Person constructor(firstName: String) {

}

### **Secondary Constructors**

The class can also declare secondary constructors, which are prefixed with constructor:

class Person {

constructor(parent: Person) {

parent.children.add(this)

}

}

### **Creating instances of classes**

To create an instance of a class, we call the constructor as if it were a regular function:

val invoice = Invoice()

val customer = Customer("Joe Smith")

Note: that Kotlin does not have a new keyword.

### **Inheritance**

All classes in Kotlin have a common superclass Any, that is a default super for a class with no supertypes declared:

class Example // Implicitly inherits from Any

Any is not java.lang.Object; in particular, it does not have any members other than equals(), hashCode() and toString().

To declare an explicit supertype, we place the type after a colon in the class header:

open class Base(p: Int)

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

### **Overriding Methods**

Kotlin requires explicit annotations for overridable members (we call them open) and for overrides:

open class Base {

open fun v() {}

fun nv() {}

}

class Derived() : Base() {

override fun v() {}

}

The override annotation is required for Derived.v(). If it were missing, the compiler would complain.

### **Abstract Classes**

A class and some of its members may be declared abstract. An abstract member does not have an implementation in its class.

We can override a non-abstract open member with an abstract one

open class Base {

open fun f() {}

}

abstract class Derived : Base() {

override abstract fun f()

}

[R](https://github.com/Umuzi-org/ACN-syllabus/blob/develop/content/topics/kotlin/classes/_index.md)

# TOPIC: CONSTRUCTOR DEFAULT VALUES

|  |
| --- |
| Hard Prerequisites |
| IMPORTANT: Please review these prerequisites, they include important information that will help you with this content. |
|  [TOPICS: Null Safety](http://syllabus.africacode.net/topics/kotlin/null-safety) |

### **Kotlin has two nice features that you’ll also find in Scala:**

You can supply default values for constructor parameters You can use named arguments when calling a constructor

### **Default values for constructor parameters**

A convenient Kotlin feature is that you can supply default values for constructor parameters. For example, you could define a Socket class like this:

class Socket(var timeout: Int, var linger: Int) {

override def toString = s"timeout: $timeout, linger: $linger"

}

That’s nice, but you can make this class even better by supplying default values for the timeout and linger parameters:

class Socket(var timeout: Int = 2000, var linger: Int = 3000) {

override fun toString(): String = "timeout: ${timeout}, linger: ${linger}"

}

By supplying default values for the parameters, you can now create a new Socket in a variety of different ways:

Socket()

Socket(1000)

Socket(4000, 6000)

This is what those examples look like in the REPL:

>>> Socket()

timeout: 2000, linger: 3000

>>> Socket(1000)

timeout: 1000, linger: 3000

>>> Socket(4000, 6000)

timeout: 4000, linger: 6000

As those examples shows:

When all parameters have default values, you don’t have to provide any values when creating a new instance If you supply one value, it’s used for the first named parameter You can override the default values with your own values An important implication of this is that default values have the effect of letting consumers consumers create instances of your class in a variety of ways — in a sense they work just as though you had created multiple, different constructors for your class.

### **When you don’t provide defaults for all parameters**

As a word of caution, it generally doesn’t make any sense to provide a default value for an early parameter without providing a default for subsequent parameters.

// don't do this

class Socket(var timeout: Int = 5000, var linger: Int) {

override fun toString(): String = "timeout: ${timeout}, linger: ${linger}"

}

If you do that, you’ll get errors when trying to construct an instance with zero or one parameters:

>>> val s = Socket()

error: no value passed for parameter 'linger'

val s = Socket()

^

>>> val s = Socket(2)

error: no value passed for parameter 'linger'

val s = Socket(2)

^

If you’re not going to provide default values for all parameters, you should only provide default values for the last parameters in the constructor:

// this is a little better

class Socket(var timeout: Int, var linger: Int = 5000) {

override fun toString(): String = "timeout: ${timeout}, linger: ${linger}"

}

With this approach the zero-argument constructor still fails, but you can use the one-argument constructor:

>>> val s = Socket()

error: no value passed for parameter 'timeout'

val s = Socket()

^

>>> val s = Socket(10)

>>> s

timeout: 10, linger: 5000

Named arguments

You can use named arguments when creating new class instances. For example, given this class:

class Socket(var timeout: Int, var linger: Int) {

override fun toString(): String = "timeout: ${timeout}, linger: ${linger}"

}

you can create a new Socket like this:

val s = Socket(timeout=2000, linger=3000)

This feature is not used too often, but it comes in handy in certain situations, such as when all of the class constructor parameters have the same type (such as Int in this example). For example, some people find that this code:

val s = new Socket(timeout=2000, linger=3000)

is more readable than this code:

val s = new Socket(2000, 3000)

# TOPIC: DATA BINDING

|  |
| --- |
| Hard Prerequisites |
| IMPORTANT: Please review these prerequisites, they include important information that will help you with this content. |
|  [TOPICS: In Line Functions](http://syllabus.africacode.net/topics/kotlin/in-line-functions) |

First of all, after having a created Android project in Android Studio, we need to add the Data Binding dependency and the ones of Kotlin to the build.gradle file of our app.

apply plugin: 'kotlin-android'

apply plugin: 'kotlin-kapt'

android {

....

dataBinding {

enabled = true

}

}

dependencies {

...

// notice that the compiler version must be the same than our gradle version

kapt 'com.android.databinding:compiler:2.3.1'

}

That’s all the configuration we need to start using Data Binding with Kotlin. Thank you very much for reading me… Now lets continue with the fun starting to see the code.

First we need to create a model. In this case a basic one like User.kt data class User(val name: String, val age: Int)

In our activity\_main.xml we can do something like this:

<?xml version="1.0" encoding="utf-8"?>

<layout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

xmlns:tools="http://schemas.android.com/tools"

>

<!-- Inside the layout tag it is possible to set the data tag in order to set one or many variables. For this example we are having an User property-->

<data>

<variable

name="user"

type="com.kuma.sample.User"

/>

</data>

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical"

tools:context="com.kuma.sample.MainActivity"

>

<TextView

android:id="@+id/user\_name\_text\_view"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:padding="16dp"

android:text="@{user.name}"

tools:text="Name"

/>

<TextView

android:id="@+id/user\_age\_text\_view"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:padding="16dp"

android:text="@{Integer.toString(user.age)}"

tools:text="XX"

/>

</LinearLayout>

</layout>

Remember to always set your usual xml view inside the tag and attach all the “xmlns:” properties to it. Otherwise it will throw a compilation error, since the generated files will have duplicated properties.

### **And here comes the Binding:**

package com.kuma.sample

import android.databinding.DataBindingUtil

import android.os.Bundle

import android.support.v7.app.AppCompatActivity

import com.kuma.kotlinsteps.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

val binding: ActivityMainBinding = DataBindingUtil.setContentView(this, R.layout.activity\_main)

val user = User("Kuma", 23)

binding.setVariable(BR.user, user)

binding.executePendingBindings()

}

}

In that code snippet there are somethings to be noticed:

* Now exists a class called ActivityMainBinding, which is autogenerated from the activity\_main.xml and this contains all the references to use the views that the xml contains.
* The way of making an instance of ActivityMainBinding is a little variation of the way of setting the xml layout to an activity.
* There is also a new BR class which is some kind of secondary R class used to store the variables declared on the data tag of the xml.
* After setting the variable to the binding object, it is necessary to call the executePendingBindings() in order to set the user variable attributes to the marked views.

After compiling this you’ll be able to see that the Data has been set to your view without the necessity of writing any

textView.text = user.name

### **Two-way data binding**

Using one-way data binding, you can set a value on an attribute and set a listener that reacts to a change in that attribute:

<CheckBox

android:id="@+id/rememberMeCheckBox"

android:checked="@{viewmodel.rememberMe}"

android:onCheckedChanged="@{viewmodel.rememberMeChanged}"

/>

Two-way data binding provides a shortcut to this process:

<CheckBox

android:id="@+id/rememberMeCheckBox"

android:checked="@={viewmodel.rememberMe}"

/>

The @={} notation, which importantly includes the “=” sign, receives data changes to the property and listen to user updates at the same time.

In order to react to changes in the backing data, you can make your layout variable an implementation of Observable, usually BaseObservable, and use a @Bindable annotation, as shown in the following code snippet:

class LoginViewModel : BaseObservable {

// val data = ...

@Bindable

fun getRememberMe(): Boolean {

return data.rememberMe

}

fun setRememberMe(value: Boolean) {

// Avoids infinite loops.

if (data.rememberMe != value) {

data.rememberMe = value

// React to the change.

saveData()

// Notify observers of a new value.

notifyPropertyChanged(BR.remember\_me)

}

}

}

Because the bindable property’s getter method is called getRememberMe(), the property’s corresponding setter method automatically uses the name setRememberMe().

### **Two-way data binding using custom attributes**

The platform provides two-way data binding implementations for the most common two-way attributes and change listeners, which you can use as part of your app. If you want to use two-way data binding with custom attributes, you need to work with the @InverseBindingAdapter and @InverseBindingMethod annotations.

For example, if you want to enable two-way data binding on a “time” attribute in a custom view called MyView, complete the following steps:

1.Annotate the method that sets the initial value and updates when the value changes using @BindingAdapter:

@BindingAdapter("time")

@JvmStatic fun setTime(view: MyView, newValue: Time) {

// Important to break potential infinite loops.

if (view.time != newValue) {

view.time = newValue

}

}

2.Annotate the method that reads the value from the view using @InverseBindingAdapter:

@InverseBindingAdapter("time")

@JvmStatic fun getTime(view: MyView) : Time {

return view.getTime()

}

At this point, data binding knows what to do when the data changes (it calls the method annotated with @BindingAdapter) and what to call when the view attribute changes (it calls the InverseBindingListener). However, it doesn’t know when or how the attribute changes.

For that, you need to set a listener on the view. It can be a custom listener associated with your custom view, or it can be a generic event, such as a loss of focus or a text change. Add the @BindingAdapter annotation to the method that sets the listener for changes on the property:

@BindingAdapter("app:timeAttrChanged")

@JvmStatic fun setListeners(

view: MyView,

attrChange: InverseBindingListener

) {

// Set a listener for click, focus, touch, etc.

}

The listener includes an InverseBindingListener as a parameter. You use the InverseBindingListener to tell the data binding system that the attribute has changed. The system can then start calling the method annotated using @InverseBindingAdapter, and so on.

Note: Every two-way binding generates a synthetic event attribute. This attribute has the same name as the base attribute, but it includes the suffix “AttrChanged”. The synthetic event attribute allows the library to create a method annotated using @BindingAdapter to associate the event listener to the appropriate instance of View. In practice, this listener includes some non-trivial logic, including listeners for one-way data binding. For an example, see the adapter for the text attribute change, TextViewBindingAdapter.

### **Converters**

If the variable that’s bound to a View object needs to be formatted, translated, or changed somehow before being displayed, it’s possible to use a Converter object.

For example, take an EditText object that shows a date:

<EditText

android:id="@+id/birth\_date"

android:text="@={Converter.dateToString(viewmodel.birthDate)}"

/>

The viewmodel.birthDate attribute contains a value of type Long, so it needs to be formatted by using a converter.

Because a two-way expression is being used, there also needs to be an inverse converter to let the library know how to convert the user-provided string back to the backing data type, in this case Long. This process is done by adding the @InverseMethod annotation to one of the converters and have this annotation reference the inverse converter. An example of this configuration appears in the following code snippet:

object Converter {

@InverseMethod("stringToDate")

@JvmStatic fun dateToString(

view: EditText, oldValue: Long,

value: Long

): String {

// Converts long to String.

}

@JvmStatic fun stringToDate(

view: EditText, oldValue: String,

value: String

): Long {

// Converts String to long.

}

}

### **Infinite loops using two-way data binding**

Be careful not to introduce infinite loops when using two-way data binding. When the user changes an attribute, the method annotated using @InverseBindingAdapter is called, and the value is assigned to the backing property. This, in turn, would call the method annotated using @BindingAdapter, which would trigger another call to the method annotated using @InverseBindingAdapter, and so on.

For this reason, it’s important to break possible infinite loops by comparing new and old values in the methods annotated using @BindingAdapter.

### **Two-way attributes**

The platform provides built-in support for two-way data binding when you use the attributes in the following table. For details on how the platform provides this support, see the implementations for the corresponding binding adapters:

* <https://developer.android.com/topic/libraries/data-binding/two-way>

### **Bind layout views to Architecture Components**

The AndroidX library includes the **Architecture Components**, which you can use to design robust, testable, and maintainable apps. The Data Binding Library works seamlessly with the Architecture Components to further simplify the development of your UI. The layouts in your app can bind to the data in the Architecture Components, which already help you manage the UI controllers lifecycle and notify about changes in the data.

This page shows how to incorporate the Architecture Components to your app to further enhance the benefits of using the Data Binding Library.

### **Use LiveData to notify the UI about data changes**

You can use LiveData objects as the data binding source to automatically notify the UI about changes in the data. For more information about this Architecture Component, see LiveData Overview.

Unlike objects that implement Observable—such as observable fields—LiveData objects know about the lifecycle of the observers subscribed to the data changes. This knowledge enables many benefits, which are explained in The advantages of using LiveData. In Android Studio version 3.1 and higher, you can replace observable fields with LiveData objects in your data binding code.

To use a LiveData object with your binding class, you need to specify a lifecycle owner to define the scope of the LiveData object. The following example specifies the activity as the lifecycle owner after the binding class has been instantiated:

class ViewModelActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

// Inflate view and obtain an instance of the binding class.

val binding: UserBinding = DataBindingUtil.setContentView(this, R.layout.user)

// Specify the current activity as the lifecycle owner.

binding.setLifecycleOwner(this)

}

}

You can use a ViewModel component, as explained in Use ViewModel to manage UI-related data, to bind the data to the layout. In the ViewModel component, you can use the LiveData object to transform the data or merge multiple data sources. The following example shows how to transform the data in the ViewModel:

class ScheduleViewModel : ViewModel() {

val userName: LiveData

init {

val result = Repository.userName

userName = Transformations.map(result) { result -> result.value }

}

}

### **Use ViewModel to manage UI-related data**

The Data Binding Library works seamlessly with ViewModel components, which expose the data that the layout observes and reacts to its changes. Using ViewModel components with the Data Binding Library allows you to move UI logic out of the layouts and into the components, which are easier to test. The Data Binding Library ensures that the views are bound and unbound from the data source when needed. Most of the remaining work consists in making sure that you’re exposing the correct data. For more information about this Architecture Component, see ViewModel Overview.

To use the ViewModel component with the Data Binding Library, you must instantiate your component, which inherits from the ViewModel class, obtain an instance of your binding class, and assign your ViewModel component to a property in the binding class. The following example shows how to use the component with the library:

class ViewModelActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

// Obtain the ViewModel component.

val userModel: UserModel by viewModels()

// Inflate view and obtain an instance of the binding class.

val binding: UserBinding = DataBindingUtil.setContentView(this, R.layout.user)

// Assign the component to a property in the binding class.

binding.viewmodel = userModel

}

}

In your layout, assign the properties and methods of your ViewModel component to the corresponding views using binding expressions, as shown in the following example:

<CheckBox

android:id="@+id/rememberMeCheckBox"

android:checked="@{viewmodel.rememberMe}"

android:onCheckedChanged="@{() -> viewmodel.rememberMeChanged()}" />

### **Use an Observable ViewModel for more control over binding adapters**

You can use a ViewModel component that implements the Observable to notify other app components about changes in the data, similar to how you would use a LiveData object.

There are situations where you might prefer to use a ViewModel component that implements the Observable interface over using LiveData objects, even if you lose the lifecycle management capabilities of LiveData. Using a ViewModel component that implements Observable gives you more control over the binding adapters in your app. For example, this pattern gives you more control over the notifications when data changes, it also allows you to specify a custom method to set the value of an attribute in two-way data binding.

To implement an observable ViewModel component, you must create a class that inherits from the ViewModel class and implements the Observable interface. You can provide your custom logic when an observer subscribes or unsubscribes to notifications using the addOnPropertyChangedCallback() and removeOnPropertyChangedCallback() methods. You can also provide custom logic that runs when properties change in the notifyPropertyChanged() method. The following code example shows how to implement an observable ViewModel:

/\*\*

\* A ViewModel that is also an Observable,

\* to be used with the Data Binding Library.

\*/

open class ObservableViewModel : ViewModel(), Observable {

private val callbacks: PropertyChangeRegistry = PropertyChangeRegistry()

override fun addOnPropertyChangedCallback(

callback: Observable.OnPropertyChangedCallback) {

callbacks.add(callback)

}

override fun removeOnPropertyChangedCallback(

callback: Observable.OnPropertyChangedCallback) {

callbacks.remove(callback)

}

/\*\*

\* Notifies observers that all properties of this instance have changed.

\*/

fun notifyChange() {

callbacks.notifyCallbacks(this, 0, null)

}

/\*\*

\* Notifies observers that a specific property has changed. The getter for the

\* property that changes should be marked with the @Bindable annotation to

\* generate a field in the BR class to be used as the fieldId parameter.

\*

\* @param fieldId The generated BR id for the Bindable field.

\*/

fun notifyPropertyChanged(fieldId: Int) {

callbacks.notifyCallbacks(this, fieldId, null)

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# TOPIC: DATA CLASSES

|  |
| --- |
| Hard Prerequisites |
| IMPORTANT: Please review these prerequisites, they include important information that will help you with this content. |
|  [TOPICS: Interface](http://syllabus.africacode.net/topics/kotlin/interface) |

## **Data Classes**

We frequently create a class to do nothing but hold data. In such a class some standard functionality is often mechanically derivable from the data. In Kotlin, this is called a data class and is marked as data:

data class User(val name: String, val age: Int)

The compiler automatically derives the following members from all properties declared in the primary constructor:

* equals()/hashCode() pair,
* toString() of the form “User(name=John, age=42)”,
* componentN() functions corresponding to the properties in their order of declaration,
* copy() function (see below).

If any of these functions is explicitly defined in the class body or inherited from the base types, it will not be generated.

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfil the following requirements:

* The primary constructor needs to have at least one parameter;
* All primary constructor parameters need to be marked as val or var;
* Data classes cannot be abstract, open, sealed or inner;
* (before 1.1) Data classes may only implement interfaces. Since 1.1, data classes may extend other classes .

On the JVM, if the generated class needs to have a parameterless constructor, default values for all properties have to be specified.

data class User(val name: String = "", val age: Int = 0)

### **Copying**

It’s often the case that we need to copy an object altering some of its properties, but keeping the rest unchanged. This is what copy() function is generated for. For the User class above, its implementation would be as follows:

fun copy(name: String = this.name, age: Int = this.age) = User(name, age)

This allows us to write

val jack = User(name = "Jack", age = 1)

val olderJack = jack.copy(age = 2)

### **Data Classes and Destructuring Declarations**

Component functions generated for data classes enable their use in destructuring declarations:

val jane = User("Jane", 35)

val (name, age) = jane

println("$name, $age years of age") // prints "Jane, 35 years of age"

### **Standard Data Classes**

The standard library provides Pair and Triple. In most cases, though, named data classes are a better design choice, because they make the code more readable by providing meaningful names for properties.

# TOPIC: DELEGATED PROPERTIES

|  |
| --- |
| Hard Prerequisites |
| IMPORTANT: Please review these prerequisites, they include important information that will help you with this content. |
|  [TOPICS: Delegation](http://syllabus.africacode.net/topics/kotlin/delegation) |

There are certain common kinds of properties, that, though we can implement them manually every time we need them, would be very nice to implement once and for all, and put into a library. Examples include

* lazy properties: the value gets computed only upon first access,
* observable properties: listeners get notified about changes to this property,
* storing properties in a map, instead of a separate field for each property.

To cover these (and other) cases, Kotlin supports delegated properties:

class Example {

var p: String by Delegate()

}

The syntax is: val/var : by . The expression after by is the delegate, because get() (and set()) corresponding to the property will be delegated to its getValue() and setValue() methods. Property delegates don’t have to implement any interface, but they have to provide a getValue() function (and setValue() — for var’s). For example:

class Delegate {

operator fun getValue(thisRef: Any?, property: KProperty<\*>): String {

return "$thisRef, thank you for delegating '${property.name}' to me!"

}

operator fun setValue(thisRef: Any?, property: KProperty<\*>, value: String) {

println("$value has been assigned to '${property.name} in $thisRef.'")

}

}

When we read from p that delegates to an instance of Delegate, the getValue() function from Delegate is called, so that its first parameter is the object we read p from and the second parameter holds a description of p itself (e.g. you can take its name). For example:

val e = Example()

println(e.p)

This prints

Example@33a17727, thank you for delegating ‘p’ to me!

Similarly, when we assign to p, the setValue() function is called. The first two parameters are the same, and the third holds the value being assigned:

e.p = "NEW"

This prints

NEW has been assigned to ‘p’ in Example@33a17727. Note that since Kotlin 1.1 you can declare a delegated property inside a function or code block, it shouldn’t necessarily be a member of a class.

### **Standard Delegates**

The Kotlin standard library provides factory methods for several useful kinds of delegates.

### **Lazy**

lazy() is a function that takes a lambda and returns an instance of Lazy which can serve as a delegate for implementing a lazy property: the first call to get() executes the lambda passed to lazy() and remembers the result, subsequent calls to get() simply return the remembered result.

val lazyValue: String by lazy {

println("computed!")

"Hello"

}

fun main(args: Array<String>) {

println(lazyValue)

println(lazyValue)

}

This example prints:

computed!

Hello

Hello

By default, the evaluation of lazy properties is synchronized: the value is computed only in one thread, and all threads will see the same value.

If the synchronization of initialization delegate is not required, so that multiple threads can execute it simultaneously, pass LazyThreadSafetyMode.PUBLICATION as a parameter to the lazy() function.

And if you’re sure that the initialization will always happen on a single thread, you can use LazyThreadSafetyMode.NONE mode, which doesn’t incur any thread-safety guarantees and the related overhead.

### **Observable**

Delegates.observable() takes two arguments: the initial value and a handler for modifications. The handler gets called every time we assign to the property (after the assignment has been performed). It has three parameters: a property being assigned to, the old value and the new one:

import kotlin.properties.Delegates

class User {

var name: String by Delegates.observable("<no name>") {

prop, old, new ->

println("$old -> $new")

}

}

fun main(args: Array<String>) {

val user = User()

user.name = "first"

user.name = "second"

}

This example prints:

<no name> -> first

first -> second

If you want to be able to intercept an assignment and “veto” it, use vetoable() instead of observable(). The handler passed to the vetoable is called before the assignment of a new property value has been performed.

### **Storing Properties in a Map**

One common use case is storing the values of properties in a map. This comes up often in applications like parsing JSON or doing other “dynamic” things. In this case, you can use the map instance itself as the delegate for a delegated property.

class User(val map: Map<String, Any?>) {

val name: String by map

val age: Int by map

}

In this example, the constructor takes a map:

val user = User(mapOf(

"name" to "John Doe",

"age" to 25

))

Delegated properties take values from this map (by the string keys –– names of properties):

println(user.name) // Prints "John Doe"

println(user.age) // Prints 25

This works also for var’s properties if you use a MutableMap instead of read-only Map:

class MutableUser(val map: MutableMap<String, Any?>) {

var name: String by map

var age: Int by map

}

### **Local Delegated Properties (since 1.1)**

You can declare local variables as delegated properties. For instance, you can make a local variable lazy:

fun example(computeFoo: () -> Foo) {

val memoizedFoo by lazy(computeFoo)

if (someCondition && memoizedFoo.isValid()) {

memoizedFoo.doSomething()

}

}

The memoizedFoo variable will be computed on the first access only. If someCondition fails, the variable won’t be computed at all.

### **Property Delegate Requirements**

Here we summarize requirements to delegate objects.

For a read-only property (i.e. a val), a delegate has to provide a function named getValue that takes the following parameters:

* thisRef — must be the same or a supertype of the property owner (for extension properties — the type being extended),
* property — must be of type KProperty<\*> or its supertype, this function must return the same type as property (or its subtype).

For a mutable property (a var), a delegate has to additionally provide a function named setValue that takes the following parameters:

* thisRef — same as for getValue(),
* property — same as for getValue(),
* new value — must be of the same type as a property or its supertype.

getValue() and/or setValue() functions may be provided either as member functions of the delegate class or extension functions. The latter is handy when you need to delegate property to an object which doesn’t originally provide these functions. Both of the functions need to be marked with the operator keyword.

The delegate class may implement one of the interfaces ReadOnlyProperty and ReadWriteProperty containing the required operator methods. These interfaces are declared in the Kotlin standard library:

interface ReadOnlyProperty<in R, out T> {

operator fun getValue(thisRef: R, property: KProperty<\*>): T

}

interface ReadWriteProperty<in R, T> {

operator fun getValue(thisRef: R, property: KProperty<\*>): T

operator fun setValue(thisRef: R, property: KProperty<\*>, value: T)

}

Translation Rules Under the hood for every delegated property the Kotlin compiler generates an auxiliary property and delegates to it. For instance, for the property prop the hidden property prop$delegate is generated, and the code of the accessors simply delegates to this additional property:

class C {

var prop: Type by MyDelegate()

}

// this code is generated by the compiler instead:

class C {

private val prop$delegate = MyDelegate()

var prop: Type

get() = prop$delegate.getValue(this, this::prop)

set(value: Type) = prop$delegate.setValue(this, this::prop, value)

}

The Kotlin compiler provides all the necessary information about prop in the arguments: the first argument this refers to an instance of the outer class C and this::prop is a reflection object of the KProperty type describing prop itself.

Note that the syntax this::prop to refer a bound callable reference in the code directly is available only since Kotlin 1.1.

### **Providing a delegate (since 1.1)**

By defining the provideDelegate operator you can extend the logic of creating the object to which the property implementation is delegated. If the object used on the right hand side of by defines provideDelegate as a member or extension function, that function will be called to create the property delegate instance.

One of the possible use cases of provideDelegate is to check property consistency when the property is created, not only in its getter or setter.

For example, if you want to check the property name before binding, you can write something like this:

class ResourceLoader<T>(id: ResourceID<T>) {

operator fun provideDelegate(

thisRef: MyUI,

prop: KProperty<\*>

): ReadOnlyProperty<MyUI, T> {

checkProperty(thisRef, prop.name)

// create delegate

}

private fun checkProperty(thisRef: MyUI, name: String) { ... }

}

fun <T> bindResource(id: ResourceID<T>): ResourceLoader<T> { ... }

class MyUI {

val image by bindResource(ResourceID.image\_id)

val text by bindResource(ResourceID.text\_id)

}

The parameters of provideDelegate are the same as for getValue:

* thisRef — must be the same or a supertype of the property owner (for extension properties — the type being extended),
* property — must be of type KProperty<\*> or its supertype.

The provideDelegate method is called for each property during the creation of the MyUI instance, and it performs the necessary validation right away.

Without this ability to intercept the binding between the property and its delegate, to achieve the same functionality you’d have to pass the property name explicitly, which isn’t very convenient:

// Checking the property name without "provideDelegate" functionality

class MyUI {

val image by bindResource(ResourceID.image\_id, "image")

val text by bindResource(ResourceID.text\_id, "text")

}

fun <T> MyUI.bindResource(

id: ResourceID<T>,

propertyName: String

): ReadOnlyProperty<MyUI, T> {

checkProperty(this, propertyName)

// create delegate

}

In the generated code, the provideDelegate method is called to initialize the auxiliary prop$delegate property. Compare the generated code for the property declaration val prop: Type by MyDelegate() with the generated code above (when the provideDelegate method is not present):

class C {

var prop: Type by MyDelegate()

}

// this code is generated by the compiler

// when the 'provideDelegate' function is available:

class C {

// calling "provideDelegate" to create the additional "delegate" property

private val prop$delegate = MyDelegate().provideDelegate(this, this::prop)

val prop: Type

get() = prop$delegate.getValue(this, this::prop)

}

Note that the provideDelegate method affects only the creation of the auxiliary property and doesn’t affect the code generated for getter or setter.