Basics of Kotlin

What is Kotlin

Kotlin is a general-purpose, statically typed, and open-source programming language. It runs on JVM and can be used anywhere Java is used today. It can be used to develop Android apps, server-side apps and much more.

History of Kotlin

Kotlin was developed by the JetBrains team. A project was started in 2010 to develop the language and officially, first released in February 2016. Kotlin was developed under the Apache 2.0 license.

Features of Kotlin

Concise: Kotlin reduces writing the extra codes. This makes Kotlin more concise.

Null safety: Kotlin is null safety language. Kotlin aimed to eliminate the NullPointerException (null reference) from the code. Interoperable.

Interoperable: Kotlin easily calls the Java code in a natural way as well as Kotlin code can be used by Java.

Smart cast: It explicitly typecasts the immutable values and inserts the value in its safe cast automatically.

Compilation Time: It has better performance and fast compilation time.

Tool-friendly: Kotlin programs are build using the command line as well as any Java IDE.

Extension function: Kotlin supports extension functions and extension properties which means it helps to extend the functionality of classes without touching their code.

Kotlin Environment Setup

Prerequisite

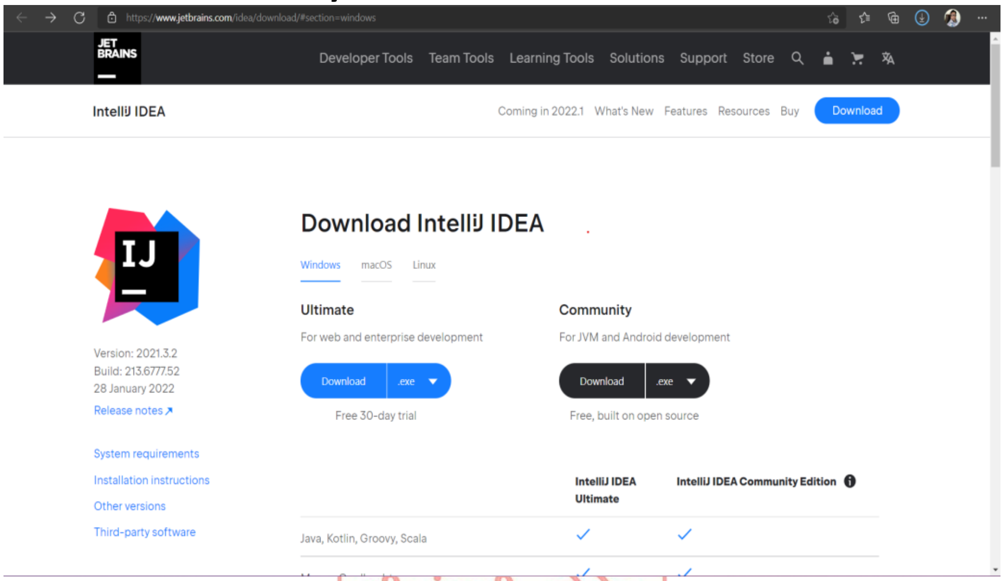
Since Kotlin runs on JVM, it is necessary to install JDK and setup the JDK and JRE path in the local system environment variable.

Kotlin Environment Setup (IDE)

Install IDE for Kotlin

There are various Java IDE available which support Kotlin project development. We can choose these IDE according to our compatibility.

1. **Download Community Version IDE.**



Download IntelliJ IDEA

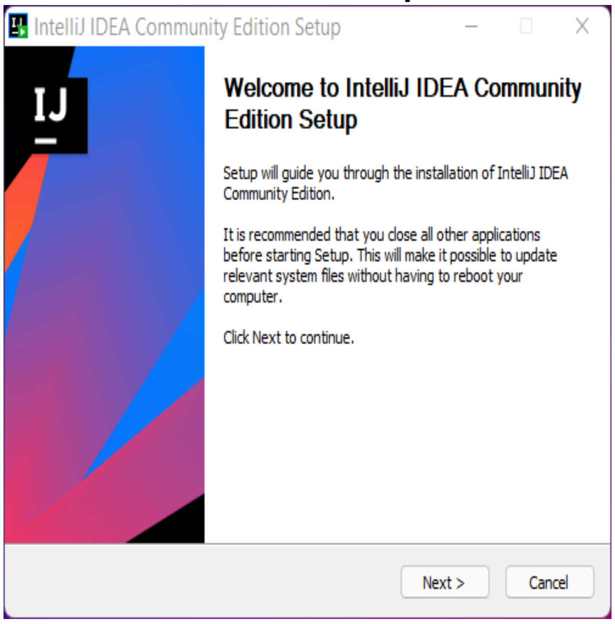
Version: 2021.3.2
Build: 213.6777.52
28 January 2022

Ultimate
For web and enterprise development
Free 30-day trial

Community
For JVM and Android development
Free, built on open source

	IntelliJ IDEA Ultimate	IntelliJ IDEA Community Edition
Java, Kotlin, Groovy, Scala	✓	✓

2. **Run the downloaded setup and click next.**



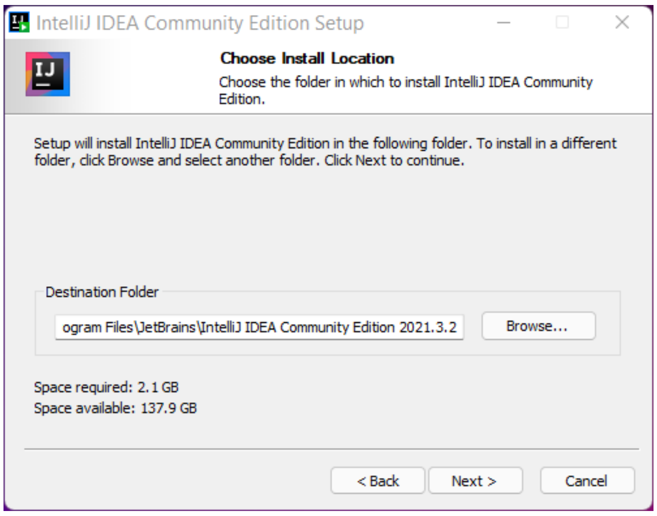
Welcome to IntelliJ IDEA Community Edition Setup

Setup will guide you through the installation of IntelliJ IDEA Community Edition.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

3. Click Next again


IntelliJ IDEA Community Edition Setup

Choose Install Location

Choose the folder in which to install IntelliJ IDEA Community Edition.

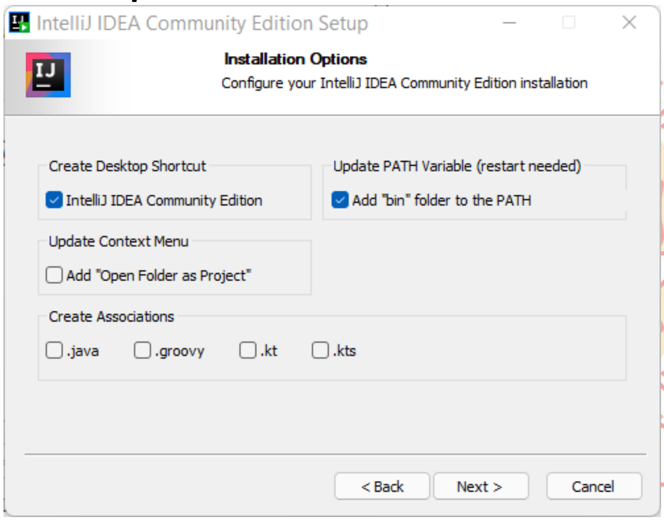
Setup will install IntelliJ IDEA Community Edition in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder

ogram Files\JetBrains\IntelliJ IDEA Community Edition 2021.3.2

Space required: 2.1 GB
Space available: 137.9 GB

4. Select Update Path and click next.


IntelliJ IDEA Community Edition Setup

Installation Options

Configure your IntelliJ IDEA Community Edition installation

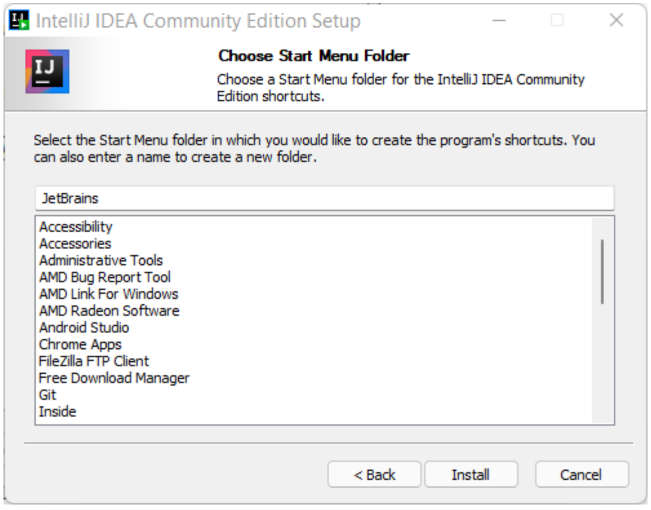
Create Desktop Shortcut
- [✓] IntelliJ IDEA Community Edition

Update PATH Variable (restart needed)
- [✓] Add "bin" folder to the PATH

Update Context Menu
- [ ] Add "Open Folder as Project"

Create Associations
- [ ] .java
- [ ] .groovy
- [ ] .kt
- [ ] .kts

5. Click Install and then Finish.


IntelliJ IDEA Community Edition Setup

Choose Start Menu Folder

Choose a Start Menu folder for the IntelliJ IDEA Community Edition shortcuts.

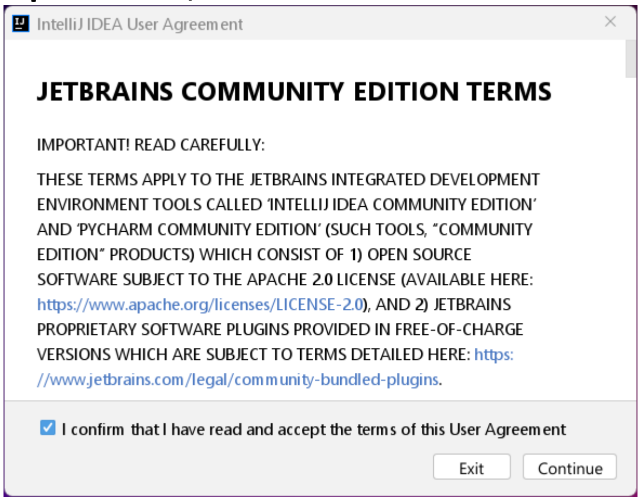
Select the Start Menu folder in which you would like to create the program's shortcuts. You can also enter a name to create a new folder.

JetBrains

Accessibility
Accessories
Administrative Tools
AMD Bug Report Tool
AMD Link For Windows
AMD Radeon Software
Android Studio
Chrome Apps
FileZilla FTP Client
Free Download Manager
Git
Inside

Running First Program

1. **Open Kotlin, Select Confirm and Continue**



IntelliJ IDEA User Agreement

JETBRAINS COMMUNITY EDITION TERMS

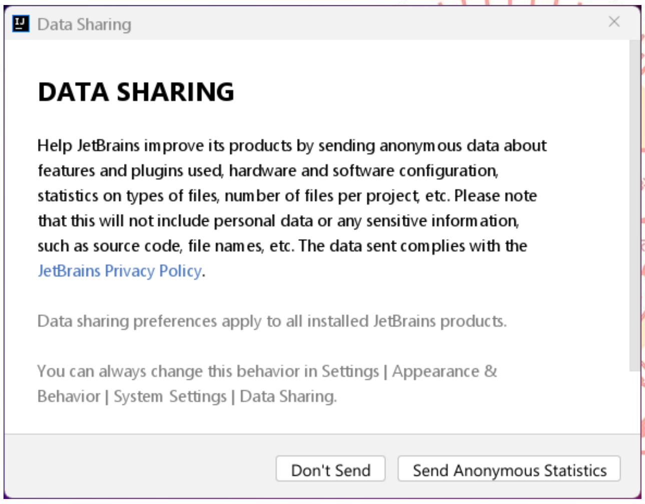
IMPORTANT! READ CAREFULLY:

THESE TERMS APPLY TO THE JETBRAINS INTEGRATED DEVELOPMENT ENVIRONMENT TOOLS CALLED 'INTELLIJ IDEA COMMUNITY EDITION' AND 'PYCHARM COMMUNITY EDITION' (SUCH TOOLS, "COMMUNITY EDITION" PRODUCTS) WHICH CONSIST OF 1) OPEN SOURCE SOFTWARE SUBJECT TO THE APACHE 2.0 LICENSE (AVAILABLE HERE: https://www.apache.org/licenses/LICENSE-2.0), AND 2) JETBRAINS PROPRIETARY SOFTWARE PLUGINS PROVIDED IN FREE-OF-CHARGE VERSIONS WHICH ARE SUBJECT TO TERMS DETAILED HERE: https://www.jetbrains.com/legal/community-bundled-plugins.

[✓] I confirm that I have read and accept the terms of this User Agreement

Exit Continue

2. **Press Don't Send.**



Data Sharing

DATA SHARING

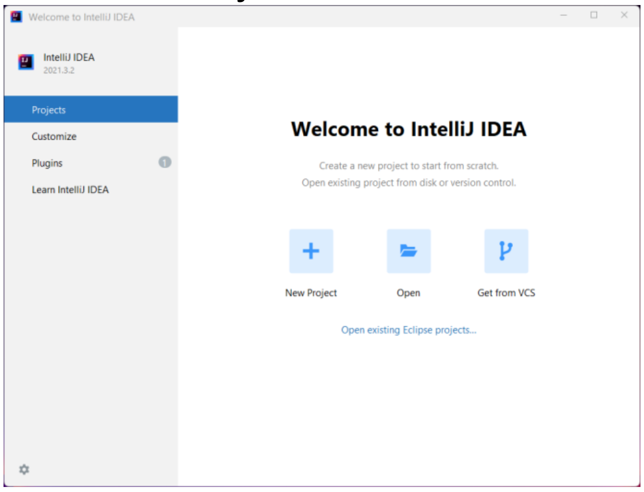
Help JetBrains improve its products by sending anonymous data about features and plugins used, hardware and software configuration, statistics on types of files, number of files per project, etc. Please note that this will not include personal data or any sensitive information, such as source code, file names, etc. The data sent complies with the JetBrains Privacy Policy.

Data sharing preferences apply to all installed JetBrains products.

You can always change this behavior in Settings | Appearance & Behavior | System Settings | Data Sharing.

Don't Send Send Anonymous Statistics

3. **Select New Project**



Welcome to IntelliJ IDEA

IntelliJ IDEA 2021.3.2

Projects
Customize
Plugins
Learn IntelliJ IDEA

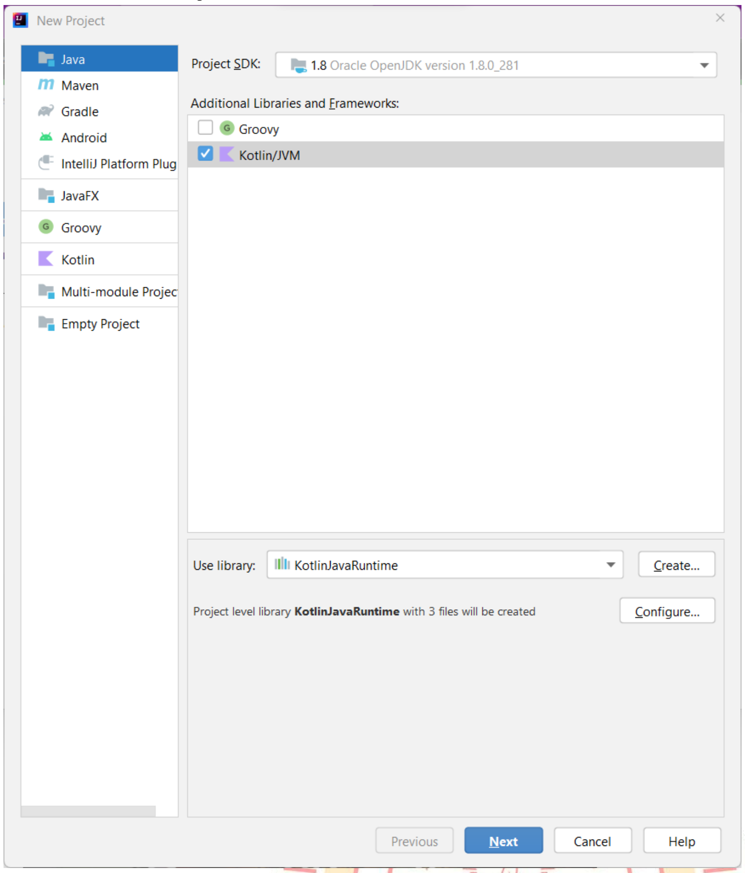
Welcome to IntelliJ IDEA

Create a new project to start from scratch.
Open existing project from disk or version control.

New Project Open Get from VCS

Open existing Eclipse projects...

4. Select Kotlin/JVM and Press Next



Project SDK: 1.8 Oracle OpenJDK version 1.8.0_281

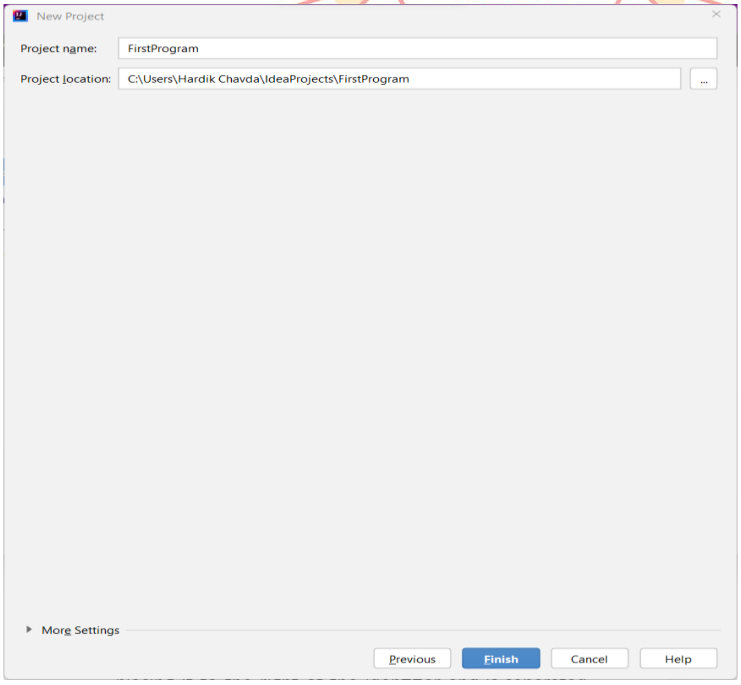
Additional Libraries and Frameworks:

- [ ] Groovy
- [x] Kotlin/JVM

Use library: KotlinJavaRuntime

Project level library **KotlinJavaRuntime** with 3 files will be created

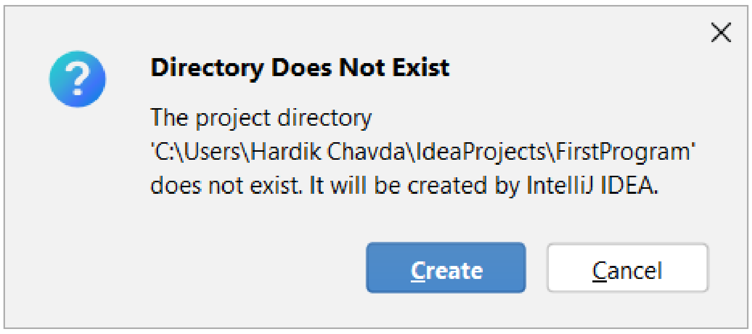
5. Press Finish



Project name: FirstProgram

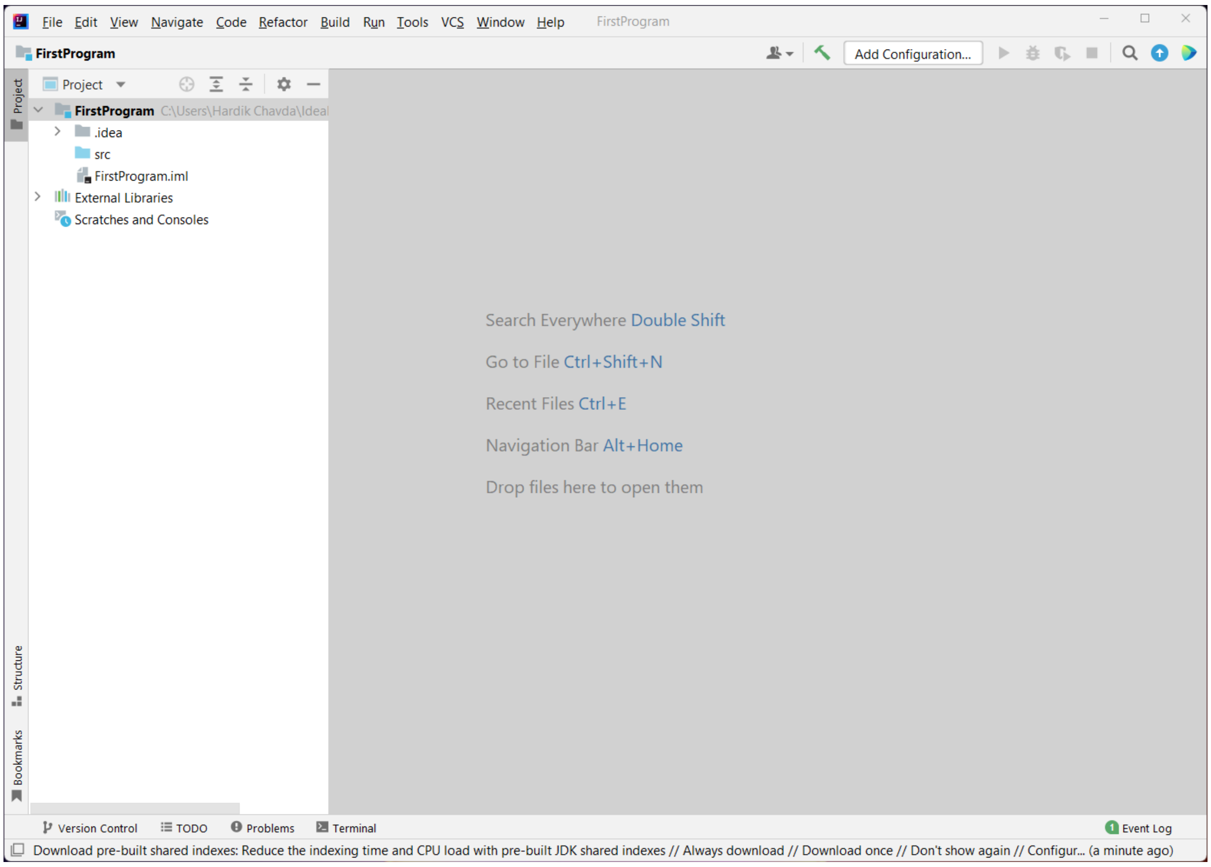
Project location: C:\Users\Hardik Chavda\IdeaProjects\FirstProgram

6. Press Create


Directory Does Not Exist

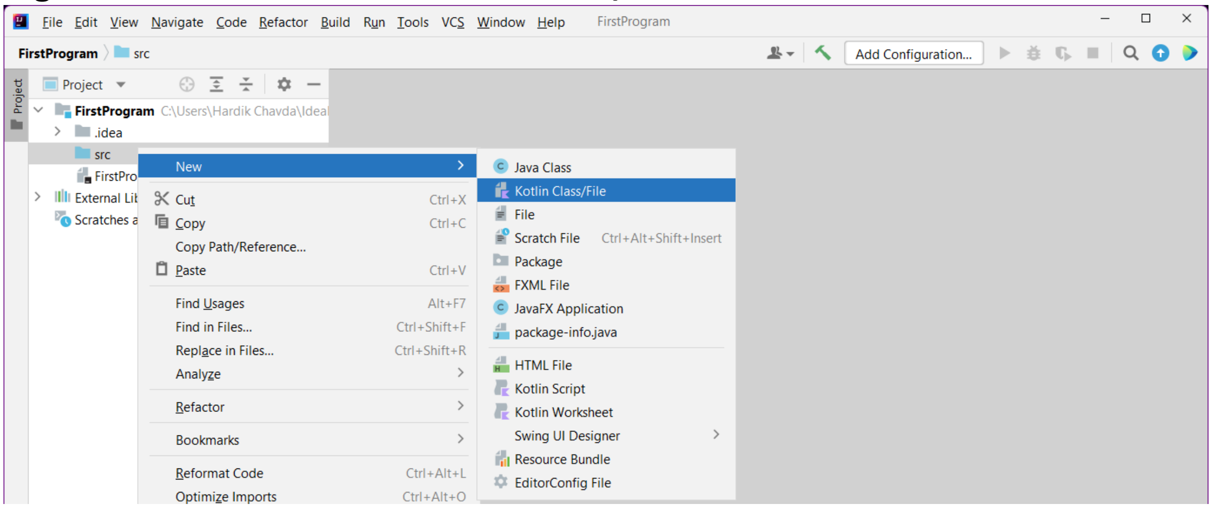
The project directory 'C:\Users\Hardik Chavda\IdeaProjects\FirstProgram' does not exist. It will be created by IntelliJ IDEA.

7. Final Screen

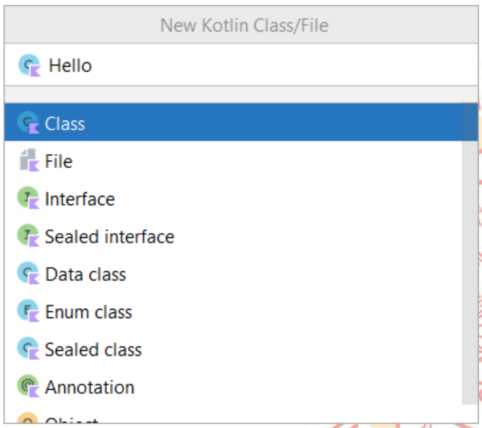


Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built JDK shared indexes // Always download // Download once // Don't show again // Configur... (a minute ago)

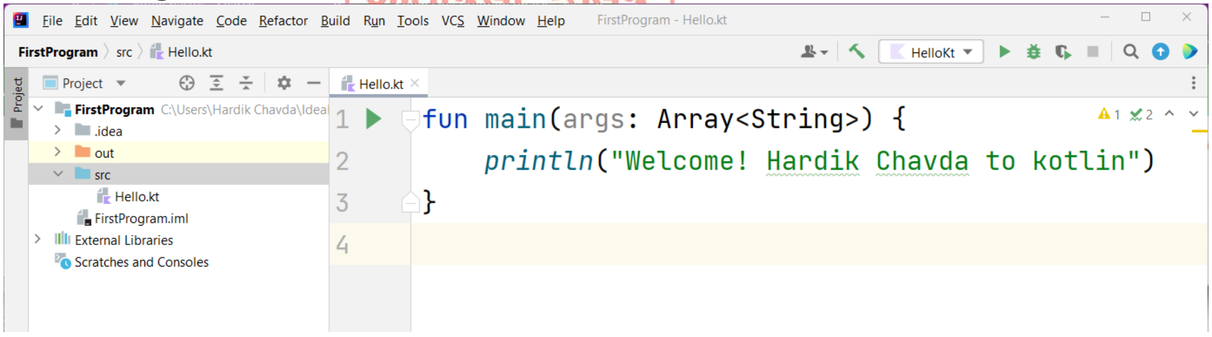
8. Right click on src and select Kotlin Class/File



9. Select Class and Write FileName

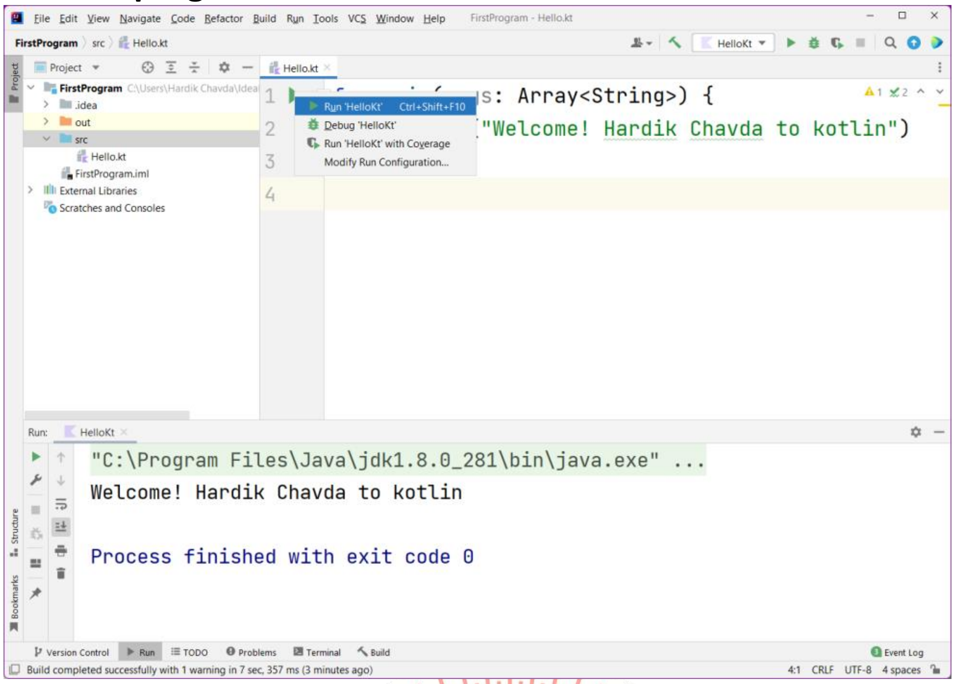


10.Write Program


File Edit View Navigate Code Refactor Build Run Tools VCS Window Help FirstProgram - Hello.kt
FirstProgram src Hello.kt
HelloKt
Project
FirstProgram C:\Users\Hardik Chavda\Idea
.idea
out
src
Hello.kt
FirstProgram.iml
External Libraries
Scratches and Consoles
Hello.kt

```kotlin
fun main(args: Array<String>) {
    println("Welcome! Hardik Chavda to kotlin")
}
```

11.Run the program


File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
FirstProgram - Hello.kt
Project
FirstProgram
.idea
out
src
Hello.kt
FirstProgram.iml
External Libraries
Scratches and Consoles
Run 'HelloKt' Ctrl+Shift+F10
Debug 'HelloKt'
Run 'HelloKt' with Coverage
Modify Run Configuration...
Run: HelloKt
Version Control
Run
TODO
Problems
Terminal
Build
Event Log
Build completed successfully with 1 warning in 7 sec, 357 ms (3 minutes ago)

```
s: Array<String>) {
"Welcome! Hardik Chavda to kotlin")
```

```
"C:\Program Files\Java\jdk1.8.0_281\bin\java.exe" ...
Welcome! Hardik Chavda to kotlin

Process finished with exit code 0
```

The **fun** keyword is used to declare a function. A function is a block of code designed to perform a particular task. In the example above, it declares the **main()** function.

The **main()** function is something you will see in every Kotlin program. This function is used to execute code. Any code inside the **main()** function's curly brackets **{}** will be executed.

For example, the **println()** function is inside the **main()** function, meaning that this will be executed. The **println()** function is used to output/print text, and in our example it will output "Hello World".

NOTE: Before Kotlin version 1.3, it was required to use the main() function with parameters, like: fun main(args : Array<String>).

Operations and Priorities

Program elements:

- **Literals**

Kotlin provides literals for the basic types (numbers, character, Boolean, String).

```
var intLiteral = 5
var doubleLiteral = .02
var stringLiteral = "Hello"
var charLiteral = '1'
var boolLiteral = true
```

- **Variables**

A variable is something that we use to manipulate data or, more precisely, a value. Values are things that you can store, manipulate, print, push, or pull from the network. For us to be able to work with values, we need to put them inside variables. A variable in Kotlin is created by declaring an identifier using the var keyword followed by the type, like in the statement.

```
var foo: Int
```

In this statement, foo is the identifier and Int is the type. Kotlin specifies types by placing it to the right of the identifier and is separated from it by a colon.

Now that the variable is declared, we can assign a value to it, like so:

```
foo = 10 and then, use it in a function, like the following:
println(foo)
```

We can declare and define variables on the same line, like in Java.
Here's the var foo example again.

```
var foo: Int = 10
println(foo)
```

We can still shorten the assignment statement above by omitting the type (Int). See the sample code:

```
var foo = 10
println(foo)
```

Kotlin uses another keyword to declare variables, the **val** keyword. Variables declared with this keyword can be initialized only once within the execution block where they were defined. That makes them effectively constants; think of **val** as the equivalent of the final keyword in Java—once you initialize it to a value, you can't change it anymore, they're immutable. While variables that were created using var are mutable, they can be changed as many times as you want.

val variables are declared and initialized just like var variables:

```
val a = 10 // declaration and initialization on the same line
```

They can also be declared and initialized at a later time, like the statements here:

```
val a: Int
a = 10
```

Just remember that variables that are declared with the val keyword are final and cannot be re-assigned once you've initialized them to a value. The code snippet here will not work:

```
val boo = "Hello"
boo = "World" // boo already has a value
```

If you think you need to change the value of the variable boo at a later time, change the declaration from val to var.

- **Expressions and Statements**

An expression is a combination of operators, functions, literal values, variables, or constants and always resolves to a value. It also can be part of a more complex expression. A statement can contain expressions, but in itself, a statement doesn't resolve to a value. It cannot be part of other statements. It's always a top-level element in its enclosing block.

For the most part, what you learned in Java about expressions and statements holds true in Kotlin, but there are slight differences. As we go further along, I'll point out the differences between Java and Kotlin when it comes to statements and expressions. Some of these differences are:

Assignments are expressions in Java, but they are statements in Kotlin. That means you cannot pass assignment operations as argument to loop statements like while

In Java

```
while ((rem = a % b) != 0) {
            a = b
            b = rem
}
println(b)
```

In Kotlin

```
var foundGcf = false
while(!foundGcf) {
 rem = a % b
 if (rem != 0) {
            a = b
            b = rem
  }
  else {
            foundGcf = true
           }
}
println(b)
```

- **Keywords**

Keywords are reserved terms that have special meaning to the compiler, and as such, they cannot be used as identifiers for any program elements such as classes, variable names, function names, and interfaces.

Kotlin has **hard**, **soft**, and **modifier** keywords. The **hard** keywords are always interpreted as keywords and cannot really be used as identifiers. Some examples of these are:
as, break, class, continue, do, else, false, while, this, throw, try, super, and when.

Soft keywords act as reserved words in certain context where they are applicable; otherwise, they can be used as a regular identifier. Some examples of soft keywords are the following:
file, finally, get, import, receiver, set, constructor, delegate, get, by, and where.

Finally, there are **modifier** keywords. These things act as reserved words in modifier lists of declarations; otherwise, they can be used as identifiers. Some examples of these things are the following:
abstract, actual, annotation, companion, enum, final, infix, inline, lateinit, operator, and open.

- **Whitespace**

Like Java, Kotlin is also a tokenized language; whitespace is not significant and can be safely ignored. You can write your codes with extravagant use of whitespace, like

```
fun main(args: Array<String>) {
 println( "Hello")
}
```

or you can write it with very little of it, like the following example:

```
fun main(args: Array<String>) {println("Hello")
```

- **Operators**

Operators are the special symbols that perform different operation on operands. For example + and – are operators that perform addition and subtraction respectively. Like Java, Kotlin contains different kinds of operators.

Arithmetic Operators –

Operators	Meaning	Expression	Translate to
+	Addition	a + b	a.plus(b)
–	Subtraction	a – b	a.minus(b)
*	Multiplication	a * b	a.times(b)
/	Division	a / b	a.div(b)
%	Modulus	a % b	a.rem(b)

Relational Operators –

Operators	Meaning	Expression	Translate to
>	greater than	a > b	a.compareTo(b) > 0
<	less than	a < b	a.compareTo(b) < 0
>=	greater than	a >= b	a.compareTo(b) >= 0 or equal to
<=	less than or	a <= b	a.compareTo(b) <= 0 equal to
==	is equal to	a == b	a?.equals(b) ?: (b === null)
!=	not equal to	a != b	!(a?.equals(b) ?: (b === null)) > 0

Assignment Operators –

Operators	Expression	Translate to
+=	a = a + b	a.plusAssign(b) > 0
-=	a = a – b	a.minusAssign(b) < 0
*=	a = a * b	a.timesAssign(b)>= 0
/=	a = a / b	a.divAssign(b) <= 0
%=	a = a % b	a.remAssign(b)

Unary Operators –

Operators	Expression	Translate to
++	++a or a++	a.inc()
—	–a or a–	a.dec()

Logical Operators –

Operators	Meaning	Expression
&&	return true if all expressions are true	(a>b) && (a>c)
||	return true if any of expression is true	(a>b) || (a>c)
!	return complement of the expression	a.not()

- **Comments**

Comments are useless to the compiler; it ignores them. But they are useful to other people (and you) who will read the codes.

```
// This statement will be ignored
```

```
/*
 Everything inside the pair of these slashes
 and asterisks will be ignored by the compiler
*/
```

Basic types

Kotlin has some basic types, but they are not the same as Java's primitive types because all types in Kotlin are objects.

- **Numbers and Literal Constants**
  Kotlin's Number Built-In Type

Type Bit	Width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Kotlin handles numbers very close to how Java handles them but with some notable differences. For example, widening conversions are not implicit anymore; you will need to perform the conversions deliberately.

```
var a = 10L // a is a Long literal, note the L postfix
var b = 20
var a = b // this won't work
var a = b.toLong() // this will work
```

When whole numbers are used as literal constants, they are automatically Ints. To declare a Long literal, use the L postfix, like

```
var a = 100 // Int literal
var b = 10L // Long literal
```

You can use underscores in numeric literals to make them more readable. This feature was introduced in Java 7 and its later versions.

```
var oneMillion = 1_000_000
var creditCardNumber = 1234_5678_9012_3456
```

Literals with decimal positions are automatically Doubles. To declare a float literal, use the F postfix, like

```
var a = 3.1416 // Double literal
var b = 2.54 // Float literal
```

Every number type can be converted to any of the number types. That means all Double, Float, Int, Long, Byte, and Short types support the following member functions:

```
• toByte() : Byte
• toShort() : Short
• toInt() : Int
• toLong() : Long
• toFloat() : Float
• toDouble() : Double
• toChar() : Char
```

- **Characters**

Character literals are created by using single quotes, like

```
var enterKey = 'a'
```

- **Booleans**

Booleans are represented by the literals true and false. Kotlin doesn't have the notion of truthy and falsy values, like in other languages such as Python or JavaScript. It means that for constructs that expect a Boolean type, you have to supply either a Boolean literal, variable, or expression that will resolve to either true or false.

```
var count = 0
if (count) println("zero") // won't work
if ("") println("empty") // won't work either
```

- **Strings**

Much of what we've learned about Java Strings are still applicable in Kotlin; hence, this section will be short. The easiest way to create a String is to use the escaped string literal—escaped strings are actually the kind of strings we know from Java. These strings may contain escape characters like \n, \t, \b, etc. See the code snippet below.

```
var str: String = "Hello World\n"
```

Kotlin has another kind of string that is called a raw string. A raw string is created by using triple quote delimiter. They may not contain escape sequences, but they can contain new lines, like

```
var rawStr = """Amy Pond, there's something you'd better understand about me 'cause it's
important, and one day your life may depend on it:
 I am definitely a mad man with a box! """
```

A couple more things we need to know about Kotlin strings are as follows:

1. They have iterators, so we can walk through the characters using a for loop:

```
val str = "The quick brown fox"
for (i in str) println(i)
```

2. Its elements can be accessed by the indexing operator (str[elem]), pretty much like Arrays

```
println(str[2]) // returns 'e'
```

3. We can no longer convert numbers (or anything else for that matter) to a String by simply adding an empty String literal to it:

```
var strNum = 10 + "" // this won't work anymore
var strNum = 10.toString() // we have to explicitly convert now.
```

We can still use String.format and System.out.printf in Kotlin; after all, we can use Java codes from within Kotlin.

Decision Making

if (expression) statement where expression resolves to Boolean. If the expression is true, the statement will be executed; otherwise, the statement will be ignored and program control will flow to the next executable statement. When you need to execute more than one statement, you can use a block with the if construct, like

If else

```
val d = Date()
val c = Calendar.getInstance()
val day = c.get(Calendar.DAY_OF_WEEK)
if (day == 1) {
 println("Today is Sunday")
}
else if (day == 2) {
 println("Today is Monday")
}
else if ( day == 3) {
 println("Today is Tuesday")
}
```

The new thing about Kotlin's if is that it's an expression, which means we can do things like

```
val theQuestion = "Doctor who"
val answer = "Theta Sigma"
val correctAnswer = ""
var message = if (answer == correctAnswer) {
 statement
}
else{
 statement
}
```

When

Kotlin doesn’t have a switch statement, but it has the when construct. Its form and structure is strikingly similar to the switch statement. In its simplest form, it can be implemented like this:

```
val d = Date()
val c = Calendar.getInstance()
val day = c.get(Calendar.DAY_OF_WEEK)
when (day) {
 1 -> println("Sunday")
 2 -> println("Monday")
 3 -> println("Tuesday")
 4 -> println("Wednesday")
}
```

when matches the argument (the variable day) against all branches sequentially until it encounters a match; note that unlike in switch statements, when a match is found, it doesn’t flow through or cascade to the next branch—hence, we don’t need to put a break statement.

The when construct can also be used as an expression, and when it’s used as such, each branch becomes the returned value of the expression. See the code example:

```
val d = Date()
val c = Calendar.getInstance()
val day = c.get(Calendar.DAY_OF_WEEK)
var dayOfweek = when (day) {
 1 -> "Sunday"
 2 -> "Monday"
 3 -> "Tuesday"
 4 -> "Wednesday"
 else -> "Unknown"
}
```

Just remember to include the else clause when it is used as an expression. The compiler thoroughly checks all possible pathways and it needs to be exhaustive, which is why the else clause becomes a requirement.

Loop Control

While

The while and do . . while statements work exactly as they do in Java—and like in Java, these are also statements and not expressions. We won't spend too much time on while and do . . while loops here.

A basic usage of the while loop is shown here, just as a refresher.

```
fun main(args: Array<String>) {
 var count = 0
 val finish = 5
 while (count++ < finish) {
 println("counter = $count")
 }
}
```

For

Kotlin's for loop, instead, works on things that have an iterator. If you've seen the for each loop in JavaScript, C#, or Java 8, Kotlin's is a lot closer to that. A basic example is shown

Basic for Loop

```
fun main(args: Array<String>) {
 val words = "The quick brown fox".split(" ")
 for(word in words) {
 println(word)
 }
}
```

If you need to work with numbers on the for loop, you can use Ranges. A range is a type that represents an arithmetic progression of integers. Ranges are created with the rangeTo() function, but we usually use it in its operator form (. .). To create a range of integers from 1 to 10, we write like this:

var zeroToTen = 0..10
We can use the in keyword to perform a test of membership.

if (9 in zeroToTen) println("9 is in zeroToTen")

Exception Handling

Kotlin’s exception handling is very similar to Java: it also uses the try-catch-finally construct. Whatever we’ve learned about Java’s exception handling commutes nicely to Kotlin. However, Kotlin simplifies exception handling by simply using unchecked exceptions. What that means is writing try-catch blocks is now optional. You may or may not do it.

```
import java.io.FileReader ➊
fun main(args: Array<String>) {
 var fileReader = FileReader("README.txt") ➋
 var content = fileReader.read() ➌
 println(content)
}
```

➊ We can use Java’s standard library in Kotlin.

➋ This one may throw the "FileNotFoundException".

➌ And this could throw the "IOException", but Kotlin happily lets us code without handling the possible Exceptions that may be thrown.

Although Kotlin lets us get away with not having to handle exceptions, we still can do that, and for some situations, we may really have to. When that happens, just write the exception handling code the way you did in Java

Kotlin’s Try-Catch Block

```
import java.io.FileNotFoundException
import java.io.FileReader
import java.io.IOException
fun main(args: Array<String>) {
var fileReader: FileReader
try {
 fileReader = FileReader("README.txt")
 var content = fileReader.read()
 println(content)
 }
 catch (ffe: FileNotFoundException) {
 println(ffe.message)
 }
 catch(ioe: IOException) {
 println(ioe.message)
 }
}
```

Data Structures (Collections)

Arrays

Coming from Java, you'll need to step back a bit before working with Kotlin arrays. In Java, these are special types; they have first-class support on the language level. In Kotlin, arrays are just types; more specifically, they are parameterized types. If you wanted to create an array of Strings, you might think that the following snippet might work:

```
var arr = {"1", "2", "3", "4", "5"}
```

This code wouldn't make sense to Kotlin—it doesn't treat arrays as a special type. If we wanted to create an array of Strings like the example, we can do it in a couple of ways. Kotlin has some library functions like arrayOf, emptyArray, and arrayOfNulls that we can use to facilitate array creation.

Using the emptyArray Function

```
var arr = emptyArray<String>();
arr += "1"
arr += "2"
arr += "3"
arr += "4"
arr += "5"
```

Using the arrayOfNulls Function

```
var arr2 = arrayOfNulls<String>(2)
arr2.set(0, "1")
arr2.set(1, "2")
println(arr2[0]) // same as arr2.get(0)
println(arr2[1])
```

Using the arrayOf Function

```
var arr4 = arrayOf("1", "2", "3")
```

This function is probably the closest syntax we can get to the Java array literal, which is probably why it is used by programmers more commonly. You can pass a comma separated list of values to the function, and that automatically populates the newly created array.

Special Array Types

```
var z = intArrayOf(1,2,3)
var y = longArrayOf(1,2,3)
var x = byteArrayOf(1,2,3)
var w = shortArrayOf(1,2,3)

println(Arrays.toString(z))
println(Arrays.toString(y))
println(Arrays.toString(x))
println(Arrays.toString(w))
```

Lists

A list is a type of collection that has a specific iteration order. It means that if we added a couple of elements to the list, and then we stepped through it, the elements would come out in a very specific order—it's the order by which they were added or inserted. They won't come out in a random order or reverse chronology, but precisely in the sequence they were added. It implies that each element in the list has a placement order, an index number that indicates its ordinal position.

Basic Usage of Lists

```
fun main(args: Array<String>) {
 val fruits = mutableListOf<String>("Apple") ❶
 fruits.add("Orange") ❷
 fruits.add(1, "Banana") ❸
 fruits.add("Guava")
 println(fruits) // prints [Apple, Banana, Orange, Guava]
 fruits.remove("Guava") ❹
 fruits.removeAt(2) ❺
 println(fruits.first() == "Strawberries") ❻
 println(fruits.last() == "Banana") ❼
 println(fruits) // prints [Apple, Banana]
}
```

❶ Creates a mutable list, the constructor function allows us to pass a variable argument that will be used to populate the list. In this case, we only passed one argument—we could have passed more.

❷ Adds an element to the list; "Orange" will come right after "Apple" since we did not specify the ordinal position for the insertion.

❸ Adds another element to the list, but this time, we told it where exactly to put the element. This one bumps down the "Orange" element and then inserts itself.

Naturally, the ordinal position or the index of all the elements that come after it will change.

❹ You can remove elements by name. When an element is removed, the element next to it will take its place. The ordinal position of all the elements that comes after it will change accordingly.

❺ You can also remove elements by specifying its position on the list.

❻ You can ask if the first() element is equal to “Strawberries”.

❼ You can also test if the last() element is equal to “Banana”.

Sets

Sets are very similar to lists, both in operation and in structure, so all of the things we’ve learned about lists apply to sets as well. Sets differ from lists in the way they put constraints on the uniqueness of elements. They don't allow duplicate elements or the same elements within a set. It may seem obvious to many what the “same” means, but Kotlin, like Java, has a specific meaning for “sameness.” When we say that two objects are the same, it means that we’ve subjected the objects to a test for structural equality.

Basic Usage for Sets

```
val nums = mutableSetOf("one", "two") ❶
nums.add("two") ❷
nums.add("two") ❸
nums.add("three") ❹
println(nums) // prints [one, two, three]
val numbers = (1..1000).toMutableSet() ❺
numbers.add(6)
numbers.removeIf { i -> i % 2 == 0 } ❻
println(numbers)
```

❶ Creates a mutable set and initializes it by passing a variable argument to the creator function.

❷ This doesn’t do anything. It won’t add “two” to the set because the element “two” is already in the set.

❸ No matter how many times you try to add “two,” the set will reject it because it already exists.

❹ This, on the other hand, will be added because “three” doesn’t exist in the elements yet.

❺ Creates a mutable set from a range. This is a handy way of creating a set (or a list) with many numeric elements.

❻ This demonstrates how to use a lambda to remove all the even numbers in the set.

Maps

Unlike lists or sets, maps aren't a collection of individual values; rather, they are a collection of pairs of values. Think of a map like a dictionary or a phone book. Its contents are organized using a key-value pair. For each key in a map, there is one and only one corresponding value. In a dictionary example, the key would be the term, and its value would be the meaning or the definition of the term.

```
val dict = hashMapOf("foo" to 1) ❶
dict["bar"] = 2 ❷
val snapshot: MutableMap<String, Int> = dict ❸
snapshot["baz"] = 3 ❹
println(snapshot) ❺
println(dict) ❻
println(snapshot["bar"]) // prints 2 ❼
```

❶ Ca mutable map

❷ Adds a new key and value to the map

❸ Assigns the dict map to a new variable. This doesn't create a new map. It only adds an object reference to the existing map.

❹ Adds another key-value pair to the map

❺ Prints {bar = 2, baz = 3, foo=1}

❻ Also prints {bar = 2, baz = 3, foo=1}, because both snapshot and dict points to the same map.

❼ Gets the value from the map using the key

Functions

Declaring functions

Functions can be written in three places. You can write them (1) inside a class, like methods in Java—these are called member functions; (2) outside classes—these are called top-level functions; and (3) they can be written inside other functions—theseare called local functions. Regardless of where you put the function, the mechanics of declaring it doesn't change much. The basic form a function is as follows:

```
fun functionName([parameters]) [:type] {
 statements
}
```

The function is declared using the reserved word fun followed by an identifier, which is the function name. The function name includes the parentheses where you can declare optional parameters. You may also declare the type of data the function will return, but this is optional since Kotlin can infer the function's return type by simply looking at the function's body declaration. What follows is the pair of curly braces with some statements inside the function's body.

You should name your functions following the same guidelines as if you are writing Java methods—namely, the function name (1) shouldn't be a reserved word; (2) mustn't start with a number; and (3) shouldn't have special characters in them. And finally, from a stylistic perspective, its name should contain a verb or something signifying an action—as opposed to when you are naming a variable where the name contains a noun.

Function Example

```
fun displayMessage(msg: String, count: Int) {
 var counter = 1
 while(counter++ <= count ) {
 println(msg)
 }
}
```

The displayMessage() is a non-productive function; it doesn't return anything—notice the absence of a return keyword in the body of the function. In Java, when a function doesn't return anything, we still indicate that the return type is void.

In Kotlin, however, we don't really have to do that since Kotlin is capable of type inference—it can figure it out for itself. But as an academic exercise, let's rewrite again verbosely to completely tell the compiler what kind of return type displayMessage() has. See the code example in below displayMessage With an

Explicit Return Type

```
fun displayMessage(msg: String, count: Int) : Unit {
 var counter = 1
 while(counter++ <= count ) {
 println(msg)
 }
}
```

Default parameters

Function parameters can have default values in Kotlin, which allows the caller (of the function) to omit some arguments on the call site. A default value can be added to a function's signature by assigning a value to a function's parameter.

connectToDb

```
fun connectToDb(hostname: String = "localhost",
 username: String = "mysql",
 password:String = "secret") {
}
```

Notice that "localhost", "mysql", and "secret" were assigned to hostname, username, and password, respectively.connectToDb("mycomputer","root") Any and all arguments to call the connectToDb() function can be omitted because all of its parameters have default values. But in this case, we omitted only the third one.

We can even call the function without passing any arguments to it, like so: connectToDb() Kotlin's ability to provide default arguments to functions allows us to avoid creating function overloads. We couldn't do this in Java, which is why we had to resort to method overloading. Overloading functions is still possible in Kotlin, but we'll probably have fewer reasons to do that now, all thanks to default parameters.

Named parameters

```
connectToDb("neptune", jupiter", "saturn")
```

This is a valid call because all of the parameters of connectToDb() are Strings, and we passed three String arguments. Can you spot the problem? It isn't clear from the call site which one is the username, the hostname, or the password. In Java, this problem of ambiguity was solved by a variety of workarounds, including commenting on the call site.

```
connectoToDb
(/* hostname*/, "neptune, /* username*/ "jupiter", /*password*/ "saturn")
```

We don't have to do this in Kotlin because we can name the argument at the call site.

```
connecToDb
(hostname = "neptune", username="jupiter", password = "saturn")
```

It's important to remember that when we start specifying the argument name, we need to specify the names of all the arguments after that in order to avoid confusion. Besides, Kotlin wouldn't let us compile if we did that. For example, a call like this

```
connectToDb(hostname = "neptune", username = "jupiter", "saturn")
```

isn't allowed because once we name the second argument (username), we need to provide the name of all the arguments that come after it. And in the example call above, the second argument is named but not the third one. On the other hand, a call like this

```
connectToDb("neptune", username = "jupiter", password = "saturn")
```

is allowed. It's okay that we didn't name the first argument, because Kotlin would have treated this as a regular function call and use the positional value of the argument to resolve the parameter. And then we named all the remaining arguments.

Extension functions

In Java, if we needed to add functionality to a class, we could either add methods to the class itself or extend it by inheritance. An extension function in Kotlin allows us to add behavior to an existing class, including the ones written in Java, without using inheritance. It essentially lets us define a function that can be invoked as a member of the class, but the function is implemented outside the class.

```
fun main() {
   var str = "Hello"
   var str2 = "welcome"
   var str3 = "hey"

   println(str3.add(str2,str))

}
//Extension Function
fun String.add(str2: String, str: String):String {
   return this + str2 + str
}
```

Infix functions

“Infix” notation is one of the notations used in math and logical expressions. It’s the placement of operator between operands (e.g., a + b; the plus symbol is “infixed” because it’s between the operands a and b). In contrast, operations can follow “post fixed” notation where the expression is written like so (+ a b) or they can be “post fixed,” in which our expression is written like this (a b +).

In Kotlin, member functions can be “infixed,” which allow us to write codes like the following:

```
fun main() {

   var str = "Hello"
   var str2 = "welcome"
   println(str2 add str) // Infix Function

}
//Extension Function
infix fun String.add(str: String):String {
   return this  + str
}
```

The only thing you need to do in order to use the say() function in an "infixed" way is to add the infix keyword in the beginning of the function, as shown in above. Having said that, you cannot convert every function to become an infix.

A function can be converted to infix, only if

- it's a member function (part of a class) or an extension function, and
- it accepts exactly one parameter (only).

If you're thinking of a loophole like, "I could probably define a single parameter in my function and use vararg," that won't work. Variable arguments are not allowed to be converted to infix functions.

By the way, you cannot call an infix function using named parameters, like this

```
john say msg = "Hello World" // won't work
```

Infix operators

The topic of operator overloading may seem a bit out of place in a chapter that is all about functions. But in Kotlin, this topic dovetails nicely into a discussion of infix functions because of their shared mechanics in implementation, as we will see shortly.

Operator overloading allows us to appropriate the use of some standard operators, like the math operators' addition, subtraction, division, multiplication, and modulo.

Object Oriented Programming:

Inheritance

Kotlin classes are final by default, as opposed to Java classes that are “open” or non-final.

Person and Employee class

```
class Person {
}
class Employee : Person() {
}
```

In order for our code sample to compile, we have to explicitly tell Kotlin that class Person is open, which signifies that we intend for it to be extended or inherited. This default behavior of Kotlin classes is considered to be the correct behavior and good practice. This effectively means that all classes and methods that you don’t intend to be extended or overridden ought to be declared as final. In Kotlin, this is the automatic behavior.

Below shows the Person class again, but this time, it has the open modifier, which signifies that class Person can be extended.

Person and Employee class

```
open class Person {
}
class Employee : Person() {
}
```

The behavior of being final as a default behavior isn’t just for classes; member functions are like that too in Kotlin. When a function is written without the open modifier, it is final.

abstract

The abstract keyword has the same meaning in Kotlin as it does in Java. It’s applicable to classes and functions. When you mark a class as abstract, it becomes implicitly open as well, so you don’t need to use the open modifier, it becomes redundant. Interfaces don’t need to be declared as abstract and open, since they are implicitly, already, abstract and open.

interface

It still uses the interface keyword, and it also contains abstract function(s). What's remarkable about Kotlin interfaces are that they can (1) contain properties and (2) have functions with implementations—in other words, concrete functions. Although, Java 8 did allow for default implementations in Java, so that last one is no longer unique to Kotlin, but still pretty useful, as we shall see later.

```
class MultiFunction Implementing Fax
class MultiFunction : Fax { ❶
 override fun answer () { ❷
 }
}
```

❶ The colon operator is used, instead of Java's implements keyword. The colon is used for inheriting classes as well.

❷ We have to provide an implementation for the answer() function because it didn't have an implementation in the interface definition. On the other hand, we don't have to provide implementation for call() and print() because they have an implementation in the interface definition.

You may also note that we are using the override keyword in this function. Its use is necessary in order to clarify to the compiler that we don't intend to hide or overshadow the answer() function in the interface definition. Rather, we intend to replace it, so it can be polymorphic. We want to provide our own behavior for the answer() function in this class.

super

Like Java, Kotlin's functions can call the functions of its supertype if it has an implementation. Also, like in Java, Kotlin uses the super keyword to accomplish this. The super keyword in Kotlin means the same as it did in Java—it's a reference to the instance of the supertype. To invoke a function on a supertype, you'll need three things:

(1) the super keyword;

(2) name of the supertype enclosed in a pair of angle brackets; and

(3) the name of function you want to invoke on the supertype. It looks something like the code snippet here:

```
super<NameOfSuperType>.functionName()
```

this

The “this” keyword in Kotlin is the same as in Java, it refers to an instance of yourself—no surprises there.

visibility modifiers

Kotlin uses almost the same keywords as Java for controlling visibility. The keywords **public**, **private**, and **protected** mean exactly the same in Kotlin as they do in Java. But the default visibility is where the difference lies.

In Kotlin, whenever you omit the visibility modifier, the **default visibility is public**

```
class Foo
class Foo {
 var bar:String = ""
 fun doSomething() {
 }
}
```

class Foo and its members are visible publicly. In contrast, Java’s default visibility is package-private, meaning it’s only available to classes that are on the same package. Kotlin doesn’t have a package-private equivalent because Kotlin doesn’t use packages as a way to manage visibility.

Packages in Kotlin are simply a way to organize files and prevent name clashes. In place of Java’s package-private, **Kotlin introduces the internal keyword**, which means it is visible in a module.

Demonstrating Visibility Modifiers

```
internal open class Foo { ❶
 private fun boo() = println("boo")
 protected fun doo() = println("doo")
}
fun Foo.bar() { ❷
 333
}

fun main(args: Array<String>) {
 var fu = Foo()
 fu.bar()
}
```

❶ Class Foo is marked as internal, which makes it visible only in classes and top-level functions that are within the same module and whose visibility are also marked internal.

❷ This is an error. The extension function is marked as public, but the receiver of the function (Foo) is marked as internal. Class Foo is less visible than the extension function; hence, Kotlin doesn't allow us.

❸ boo() is private to the class, so we can't reach it from here.

❹ doo() is protected, we can't reach it from here

Corrected Visibility Errors

```
internal open class Foo {
 internal fun boo() = println("boo")
 internal fun doo() = println("doo")
}
internal fun Foo.bar() {
 boo()
 doo()
}
fun main(args: Array<String>) {
 var fu = Foo()
 fu.bar()
}
```

Access Modifiers

The access modifiers of Kotlin are final, open, abstract, and override. They affect inheritance.