TED HAGOS

KOTLIN

QUICK BITES

Kotlin Quick bites

A no fluff introduction to the Kotlin language

Ted Hagos

This book is for sale at http://leanpub.com/kotlinquickbites

This version was published on 2021-06-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2018 - 2021 Ted Hagos

Tweet This Book!

Please help Ted Hagos by spreading the word about this book on Twitter!

The suggested hashtag for this book is #kotlinquickbites.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#kotlinquickbites

Also By Ted Hagos

Java Lessons

For Steph and Adrianne

Contents

Introduction . . . . **i**
- About the author i
- Thanks i
- Who this book is for i
- How the book is organized ii
- Typography ii

Something about Kotlin **1**

The Kotlin language **2**

Variables and Constants **4**
- Type inference 4

Function main **6**

Working with Strings **7**

Arrays **8**

Functions **9**

When **12**

Exception Handling **16**

Types and Classes **18**

Extension Functions **19**

Object declarations **20**

Higher order functions and lambdas **21**

Generics **22**

Activities and UI **28**

Dealing with Threads **29**

Introduction

Hello World again. This is a test section.

About the author

Ted Hagos is the CTO and Data Protection Officer of RenditionDigital International, a software development company based out of Dublin. Before he joined RDI, he had various software development roles and also spent time as trainer at IBM Advanced Career Education, Ateneo ITI, and Asia Pacific College. He spent many years in software development dating back to Turbo C, Clipper, dBase IV, and Visual Basic. Eventually, he found Java and spent many years there. Nowadays, heâ€™s busy with full-stack Javascript and Android.

Thanks

To Stephanie and Adrianne, my endless fount of joy and happiness.

To all of our friends, for giving me reasons to smile when there isn't much to smile about.

Who this book is for

This book is for you if;

- You're already an Android developer using Java, and you want a very very quick introduction to Kotlin
- You have experience in CFOL (C family of language e.g. JavaScript, C#, C, C++ and Java) and you'r curious about the newest kid in the JVM block
- You're just generally curious about Kotlin

This book might not be for you if;

- You're already a Kotlin expert, you've been using Kotlin even before it was officially supported in Android Studio. You probably won't learn anything new in here. But the book may still be of value, if you need a quick and short reference to the language

- You're looking for an introductory material to Android Programming e.g. concept of components, Android architeucture, Activity lifecycle, event handling basics, Layout and Views etc., then this book isn't for you. You might want to check out my other books, "Learn Android Studio 3" Java edition[1] or Kotlin edition[2]
- You're a complete beginner to programming. The fundamental concepts like variables, storage, iterations, object oriented concepts etc., are not tackled in the book. You will be better served by other books that offers such introductory concepts

How the book is organized

this section left blank

This is a test

Typography

this section left blank

[1]https://www.apress.com/gp/book/9781484231555
[2]https://www.apress.com/us/book/9781484239063

Something about Kotlin

A little bit of history;

- In 2011, JetBrains (the creators of IntelliJ, WebStorm, ReSharper etc.) unveiled Kotlin. They opensourced it the following year
- In Google I/O 2017, Google announced first-class support for Kotlin on the Android platform. Kotlin, together with Java, is now also an official language for building Android apps

About the name "Kotlin", it's the name of an island near St. Petersburg, where most of the Kotlin teama are located.

Okay, that's about all trivia we're gonna do. Let's move on to more practical things.

Kotlin is a JVM language. So, like Jython, JRuby, Scala, Groovy and Clojure, you write your program file in one language (Kotlin) use a compiler that targets the JVM (kotlin compiler) and then run. Java programs are saved in `.java` files, Kotlin programs on the other hand, are saved in `.Kt` files.

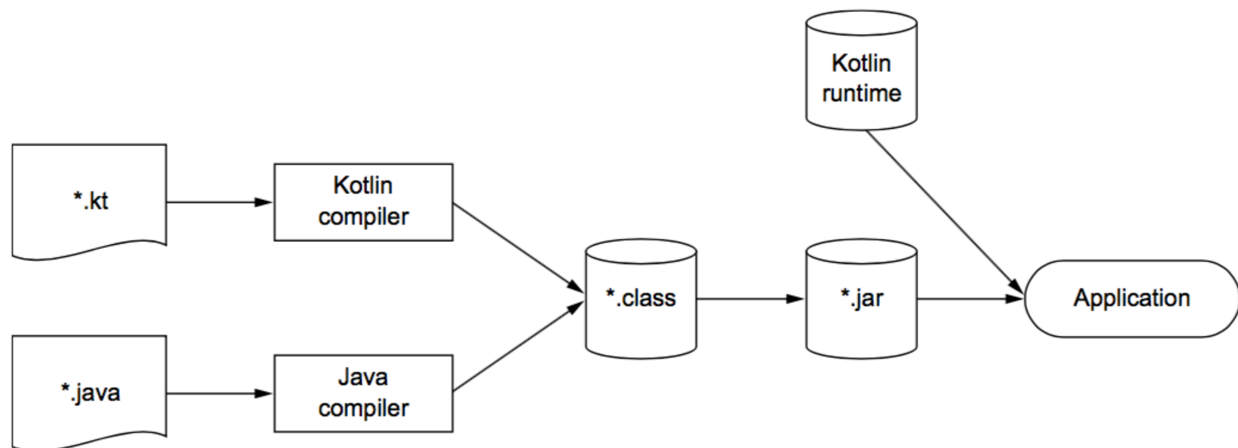


Figure 1-1.

Kotlin requires a JDK installation. JDK 8,9 should do. At the time of this writing, JDK 10 on early access, so I couldn't test it.

Oh, and one more thing. Kotlin bytecodes aren't like Java bytecodes. You cannot run Kotlin bytecodes like this

```
java HelloKotlin
```

Kotlin executables needs the Kotlin runtime libraries before they can run.

The Kotlin language

If you're a Java programmer, Kotlin program should look very familiar. The code sample in listing 4-1 shows a typical

In Kotlin, you can write codes like this (listing 4-1)

listing 4-1, Hello.Kt

```
/*
  This is a multiline comment
  It spans multiple lines
*/
class Hello {  // (1)
  fun talk() { // (2)
   println("${this.javaClass.simpleName} World") // (3)
  }
}

// You can also use the double slash for comments
// just like in Java

fun main(args: Array<String>) {        // (4)
  Hello().talk()        // (5)
}
```

Intuitively, I'm sure you know what's going on in the code, but let's inspect it a bit closer.

(1) It uses the same `class` keyword, like Java. It also uses the curly braces for class blocks, function blocks and other kinds of control structures like `if`, `while` etc.

(2) Its syntax for writing methods (functions) is a bit different from Java, but I'm sure that, even vaguely, you intuitively know what it is and what it does. Kotlin uses the reserved word **fun** to define functions

(3) `println()` is part of the package **kotlin.io**, which is automatically imported into any `.Kt` file. The other default imports for every Kotlin file are found in the docs Kotlin default imports[3]. BTW, have you noticed? We don't have to end our statements with semi-colon anymore

(4) We don't have to write `function main()` inside a class anymore. Functions in Kotlin can be written as top-level

[3]bit.ly/kotlinpackages

(5) When we create an object, we don't need the `new` keyword anymore. Just call the constructor of the class, `Hello()`, and you're good to go. You probably noticed that we didn't define any constructor in class Hello, and yet, we were able to call it a no-arg constructor. Because like Java, when a Kotlin class doesn't have an explicit constructor, it is given a no-arg constructor by default

Also, comments in Kotlin looks the same as the comments in Java. You use the same constructs for multi-line and single comments.

Variables and Constants

Kotlin, like Java, is a **statically typed** language. You still have to tell the compiler what kind of data a variable is

```
int a = 10;                 // Java
var a:Int = 10         // Kotlin
```

In Java, the type is written to left of variable. In Kotlin, the type is written to the right of the variable. You can use the keyword `var` to declare a variable

```
final int = 10;  // Java
val a:Int = 10.  // Kotlin
```

The `val` keyword (in Kotlin) means the variable is effectively *final*. While the `var` keyword makes it just a regular variable. You can assign a value to it again and again, throughout the life of your program.

```
var a:Int = 10 // OK
a = 2        // OK
a = "Hello"        // NOT OK
```

Kotlin, like Java, is *strongly typed*. When you define a variable as an `Int`, you can only assign Ints to it; not String, Double, Floats or anything other than an Int. You can only decide on the type once. You can't define a variable as Int and then change the type later in the program.

Type inference

You don't always have to write the type. Kotlin is smart enough to know what you mean. You can write codes like this.

```
var a:Int = 10        // we've seen this before
var b = 10        // this okay too
var c = "Hello"        // Compiler knows you want a String
var d = 10.F        // assigns a Float type
var e = 10.0        // assigns a Double type
```

Most of the time, Kotlin can figure out the types of your vars. It can infer the type from the RHS (right hand side of the assignment). But this isn't the case with functions.

```
fun foo() {        // (1)
  return "Hi Foo"
}
```

(1) This won't compile. If you don't specify a return type for a function, the compiler will think it's `Unit` (by default). In this situation, it can't infer that the return type of `foo()` should be String. You have to specify it, like this

```
fun foo(): String {
  return "Hi Foo"
}
```

Function main

You can write Kotlin programs without ever defining a class. You can write function at the top-level (not nested in a class).

```
1  fun main(args: Array<String>) {
2    println("Hello Kotlin")
3  }
```

This is the equivalent of Java's main function. The name `main` is special because the runtime will look for this function when you run your program. Like in Java, `main` is the entry point.

Remember Java's main?

```
1  public static void main(String []args) {
2    System.out.println("Hello Java");
3  }
```

In Kotlin's main, we didn't to write the keywords `public`, `static` and `void`.

- We don't have to write `public` because in Kotlin, classes and functions are automatically **public**, by default
- There is no `static` keyword in Kotlin. There is no concept of a method or a property that belongs to the class (like in Java). What you'll use instead, are *object declarations*
- Kotlin doesn't have a `void` keyword. The equivalent type of void is `Unit`

If you don't specify a return type for a function, it's type will be automatically `Unit`

```
1  fun foo() {
2    println("Hi foo")
3  }
```

Function main's type is `Unit`, it doesn't return anything. There is no `return` keyword anywhere in the body of the function.

Working with Strings

All the other stuff you used to do with Java Strings are still okay. There are some new things.

```
var a = "Hello"
var b = "World"
var c = 1234
println(a + " " b + " " + c.toString()) // this OK
println("$a + $b + $c") // this is better
```

You can interpolate the variables inside a String literal. Much easier to use than Java's `String.format`. String interpolation works by embedding the variable inside a String literal and prepending the variable with the dollar sign. Kotlin calls this String templates. A template is simply a String that contains template expressions (the dollar sign or the dollar sign with curly braces). The expression is evaluated at runtime and the result is concatenated into the String.

```
"Hello $<yourVarHere> World"
```

It even works with expressions

```
println("The sum of 1 + 2 + 3 + 4 is ${1+2+3+4}")
```

In a complex expression, enclose the expression inside curly braces; like in the sample code above.

Kotlin has another kind of String called *raw Strings*. Instead of enclosing the Strings in double quotes, these Strings are enclosed in triple quotes. Like this

```
var long_text = """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!
"""
```

I'm sure you can think of situations where raw String will come in handy.

Arrays

In Kotlin, Array is just a class, not a special type (like it is in Java). It' s a parameterized class (generics), so, you have to specify type arguments when declare an Array.

You can create Arrays in a couple of ways. The Array class has several constructor functions like `emptyArray()`,`arrayOfNulls()` , `Array()` or, my favorite of them all, `arrayOf()`

```
1 val a = arrayOf(1,2,3,4,5)                                   // (1)
2 val b = (0..100).toList().toTypedArray()        // (2)
3 val c = "The quick brown".split(" ").toTypedArray() // (3)
4
5 println(java.util.Arrays.toString(a))        // (4)
```

(1) Creates an Array of Ints `[1,2,3,4,5]`

(2) Uses the range operator (`..`) to create a Int series from `0` to `100`. Then converts it to an Array of Ints

(3) Uses the `split()` method of String class, then converts it to an Array<String>. It produces `["The", "quick", "fox"]`

(4) We're just printing the Array (of Ints) here. I used the `Arrays` class from `java.util` to print the Array objects. You have to use this utility class if you want to see the actual contents of the Array, otherwisem, `println` will just output something like this `[Ljava.lang.Integer;@4b6995df`

Functions

Kotlin functions are defined using `fun` keyword, followed by the name of the function, a pair of parentheses, optional parameters, the return type of function and then the body of the function, which is pair of curly braces. It's pretty much like a Java function, except that in Java, we have no `fun` (pun intended) and the type of the function is written to the left of the function (in Java), while in Kotlin, the return type of the function is to the right of function name.

When a function declares a *return type*, the type and the function's name must be separated by a colon. Here's a function that takes in a Float argument and returns a Float value (see listing 10-1).

listing 10-1, function inchesToCm

```
fun inchesToCm(inches:Float) : Float {
  return inches * 2.54
}

fun main(args:Array<String) {
  println(inchesToCm(10))
}
```

If a function has a single statement in it's body (like the previous code sample), you can write it as an expression. Like this

```
fun inchesToCm(inches:Float) = inches * 2.54
```

In a function expression, you don't have to write the return type because Kotlin can infer it from the RHS (right hand side of the assignment operator).

When a function doesn't specify it's return type, the compiler will assume that it returns the `Unit` type. Think of `Unit` as the equivalent of Java's `void`. See listing 10-2.

listing 10-2, function talk

```
fun talk() { // (1)
  println("Hello") // (2)
}
```

(1) Our function doesn't have a declared return type. Hence, it's automatically `Unit`

(2) The body of a function whose return type is `Unit` cannot have any *return* statement in it

Functions can be written in 3 places; as a top-level (which you've seen already) inside classes (which you've also seen) and inside another function.

```
fun foo() {
  fun baz() : String{  // (1)
    return "Bar"
  }
  println("Foo ${baz()}")         // (2)
}

fun main(args:Array<String>) {
  foo()
  bar()        // (3)
}
```

(1) The function `baz()` is nested inside function `foo()` . Which means, `baz` is scoped with `foo`; which means, it can only be called within the body of `foo()`

(2) This call is ok, `baz()` is callable within `foo()`

(3) Not ok, `baz()` is not callable from `main()` , you can only call `baz()` from within `foo()`

Default function arguments

You can assign default values to function parameters, like this

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

And because there are default parameters, you can call this function without passing any arguments. Like this

```
connectToDb()
```

Or you can pass only two arguments, like this

```
connectToDb("mycomputer", "ted")
```

"mycomputer" is the hostname, "ted" is the username. The two arguments that we passed corresponds to the first two parameters that were defined in the function.

Named params

In addition to default function arguments, we can also call the function by naming the arguments at the callsite.

```
fun main(args:Array<String>) {
  connectToDb(hostname="neptune",
              username="saturn")
}
```

In the example above, we only passed two arguments, but since our function has default parameters, that's okay.

When

Kotlin doesn't have a *switch* statement, but it has the *when* construct. It looks a lot like the *switch* but it packs a lot more punch. 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")
 }
```

The idea is to match the argument (the variable `day`) against the branches `1`, `2`, `3` or `4` . The test is carried out from top to bottom (1, then 2, then 3 then 4) and when a match is made the statement (or block) to right of the thin arrow `->` is executed. Unlike in *switch* statements, when a match is found, it doesn't flow through or cascade to the next branch, so, you 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 below.

```
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 if you use it 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.

You're not limited to numeric literals, you can use a wide variety of types for the branches, like this code below.

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

  print("What is the answer to life? ")

  var response:Int? = readLine()?.toInt()  // (1)                              \

  val message = when(response){
    42 -> "So long, and thanks for the all fish"
    43, 44, 45 -> "either 43,44 or 45" // (2)
    in 46 .. 100 ->  "forty six to one hundred" // (3)
    else -> "Not what I'm looking for" // (4)
  }
  println(message)
}
```

(1) `readLine()` reads an input from the console. Don't worry about the questions marks for now, we'll get to that in the coming sections

(2) The branch conditions may be combined with a comma

(3) We can check if it's a member of a range or a collection

(4) The else clause is required when when is used as an expression

You can also use *when* without an argument, like this

```
val a = 3
val b = 2
val c = 1
val str = "str"

when {
  a > b -> println("a is greater than b")
  b > c -> println("b is greater than c")
  b == 2 -> println("b is equal to 2")
  str == "str" -> println("str is actually equal to $str")
}
```

Which effectively makes it the equivalent of an *if-elseif* structure; it looks much cleaner though. But hold on a minute, read the sample code above again, and this time, examine it closely. Have you noticed that all the conditions in all the branches should evaluate to true? They do. However, if you run code above, it will only print the first branch, `a > b`. Like the *if-elseif*, when one branch evaluates to true, the rest of the succeeding branches are ignored. They will no longer be evaluated. Best to take note of that.

You can also use the *is* operator in *when* construct. If you do that, you get the benefit of smart casts as well. See the following code for an example.

```
when (arg) {
  is Int -> println("The square of $arg is ${arg * arg}")
  is String -> println("Hello" + arg)
  is IntArray -> println(arg.sum())
}
```

Another, albeit, slightly more complicated example of using the *is* operator in a *when* construction.

```
fun <T> fooBar(arg:T) : T {
  var retval:T = 0 as T
  when (arg) {
    is String -> {
      retval = "Hello world" as T
    }
    is Number -> {
      retval = 100 as T
    }
  }
  return retval
}
```

Here's how you might use it in an Android app.

```
class MainActivity : AppCompatActivity() {
  val Log = Logger.getLogger(MainActivity::class.java.name)

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
  }

  override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    Log.info("onCreateOptionsMenu")
    menu?.add("File")
    menu?.add("Exit")
    return super.onCreateOptionsMenu(menu)
  }

  override fun onOptionsItemSelected(item: MenuItem?): Boolean {
```

```
17     when (item?.toString()) {
18       "File" -> {
19         Log.info("LOG File menu")
20       }
21       "Exit" -> {
22         Log.info("LOG Exit menu")
23       }
24     }
25     return true
26   }
27 }
```

Exception Handling

Kotlin's approach to exception is similar to Java. Somewhat. It uses the *try-catch-finally*, just like in Java. So, your knowledge about how *try-catch* works commutes nicely to Kotlin. The code below should be very familiar. It shows a typical code on how to open a file

```
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)
 }
}
```

So, what's different? Well, in Kotlin, everything is an *unchecked exception*. Which means, the try-catch block is optional. It's up to the programmer if you want to use it. So, the code above, can be written like this (in Kotlin).

```
import java.io.FileReader

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

So how do you know if you need to use try-catch. You might not like the answers now, but in the long run, these are good advice to follow ;

1. You have to know the API's you're using. I know that this is not great news for beginners, but look at it this way, you won't be a beginner for long. And you really should know the API's you're using. TL;DR won't serve you well in this area
2. You have to get into the habit of unit-testing your code. This is a good habit to develop anyway

Types and Classes

this page left blank

Extension Functions

Object declarations

this page left blank

Higher order functions and lambdas

this page left blank

Use this photo https://medium.com/@cscalfani/so-you-want-to-be-a-functional-programmer-part-4-18fbe3ea9e49

Generics

Generics came to Java around 2004, when JDK 1.5 was released. Before generics, you could write codes like this

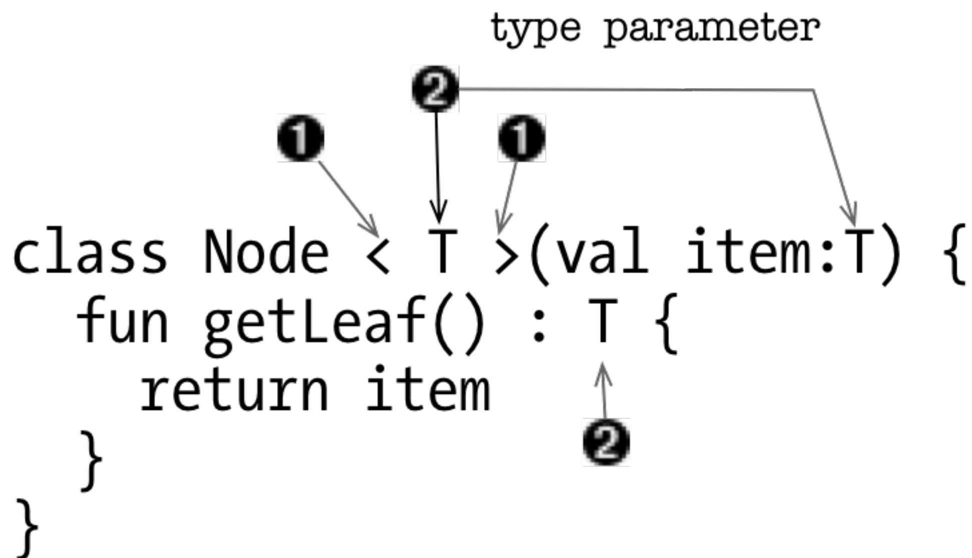
```
1  List v = new ArrayList();
2  v.add("test");
3  Integer i = (Integer) v.get(0); // Run time error
```

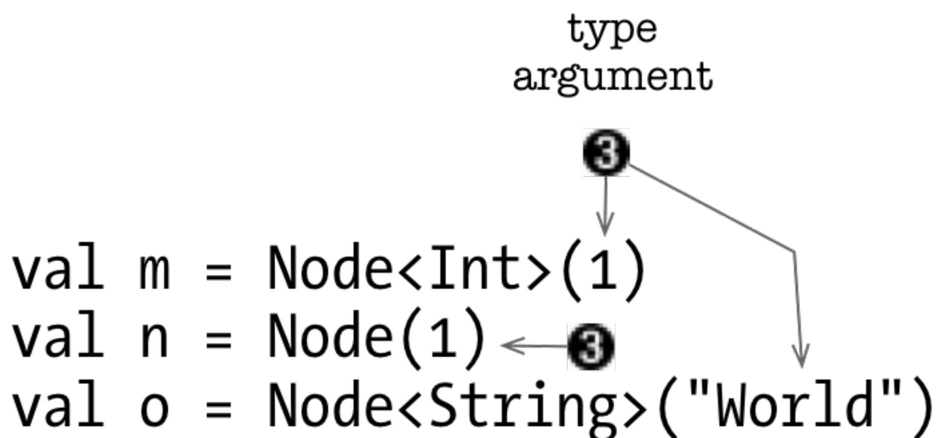
This is an idiotic thing to do, to be sure. From these three lines of code, you can clearly see that *List v* contains a String, so, it's easy to see why line 3 in the code above is a mistake. But when you're working in a project with thousand of lines of code, the problem won't be as easy to spot like this. You won't be able to tell (always) what the List contains.

The other point to notice about the sample code — and it's actually the main point — is that the code will compile without problems. You'll only discover the error at runtime. There was no way for the compiler to warn us that we're about to do something that isn't type-safe. This is the main problem that generics is trying to solve. Type-safety.

Terminologies

Generic programming is a language feature of Kotlin. With it, we can define classes, functions and interfaces that accepts type parameters. The parameterized type lets us re-use the algorithm to work with different types, it truly is, a form of *parametric polymorphism*. Diagram below shows the where the type parameters and type arguments are in a generic class.





terminologies

(1) Angle brackets. When a class, has angle brackets at the end of its name, it's called a generic class (there are also generic functions and interfaces)

(2) Type parameter. It defines the typeof data that this class can work with. You can think of it as being part of the class implementation. Right now, we're using the letter **T** to symbolize the type parameter, but this is arbitrary. You can call it anything you want, it can be any letter or a combination of letters; I'd stick to **T** if I were you, because it's the convention many developers follow. You can use **T** throughout the code inside the class as if it's a real type. It's a *placeholder* for a type. In this example, we used *T* as type for the **item** property and as return type for the **getLeaf** function

(3) **Type argument**. In order to use the generic class, you to have provide the **type argument**. Now that we're creating an instance of the Node class, **T** will be substituted by *type argument* (*Int* and

String, in this illustration)

Using generics in Functions

To create a generic function, declare the type parameter before the function name. Then, you can use the type parameter anywhere in the function.

```
fun <T> fooBar(arg:T) : String {  // (1)
  return "Heya $arg"  //   (2)
}

println(fooBar("Joe"))  // prints "Heya Joe"
println(fooBar(10))     // prints "Heya 10"
```

(1) The typeparameter **T** is used as the type of the function parameter **arg**

(2) We're just concatenating the value of arg into a String, and then returning it

That'spretty simple to follow. We just used the type param in one place and the function is returning a String, no matter what type the param is. Let's see a more complex example

```
fun <T> fooBar(arg:T) : T {   // (1)
  var retval:T = 0 as T
  when (arg) {
    is String -> {                 // (2)
      retval = "Hello world" as T  // (3)
    }
    is Number -> {
      retval = 100 as T
    }
  }
  return retval
}
```

(1) In this example, we used the *type parameter* as a type for **arg** (parameter to **fooBar** function) and as a return type of the function itself

(2) We're testing if **arg** is of String type. If it is, we're also effectively casting it to a String; smart cast, remember?

(3) We're returning "Hello world", and were casting it (forcibly) as **T**. We cannot return a "String" type right here, because fooBar expects to return type **T** to its caller, not String

Using generics in Classes

Like in Java, you can create Kotlin generic classes by putting a pair of angle brackets after the name of the class and placing the type parameter between the angle brackets. After that, you can use the type parameter anywhere in the class. The following code sample shows us how to write a generic class.

```
class Node<T>(val item:T) {   // (1)
  fun getLeaf() : T {         // (2)
    return item
  }
}

fun main(args: Array<String>) {
  val m = Node<Int>(1)         // (3)
  val n = Node(1)              // (4)
  val o = Node<String>("World")// (5)
}
```

(1) Type parameter is declared right after the name of the class, **Node<T>**. We're using the **T** as the type for parameter **item**

(2) We're also using **T** as the return value of the function **getLeaf**

(3) We're passing an Int to the constructor of Node. We can be verbose and specify Int as the as the type parameter, **Node<Int>**

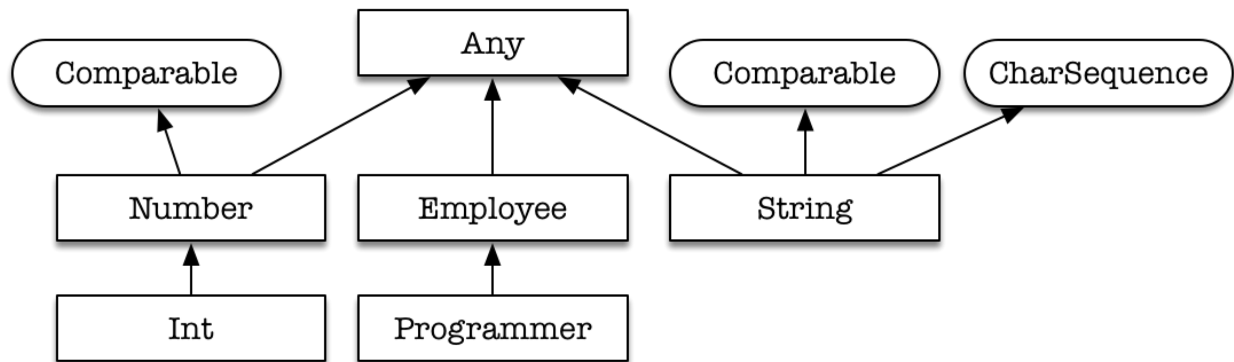
(4) Node can infer what the type parameter is, so we can skip the angle brackets. It's okay to write it this way too

(5) And because it's a generic class, it works with Strings too

Subclass vs Subtype

Most of the time we think of subtypes as being the same as a subclass; and most of the time, that's okay. So, when is the distinction between a subclass and subtype become important? When you get to generics.

Alright, let's jog down memory lane and remember what your Java teacher, mentor or favorite author told you about types; that they are "the sum total of all its public behavior, otherwise known as the object's methods or contract" — or something like that; let's just say it's the set of behavior that the object has. You can probably also remember that a class has at least one type, that of itself; and that it can have other types.



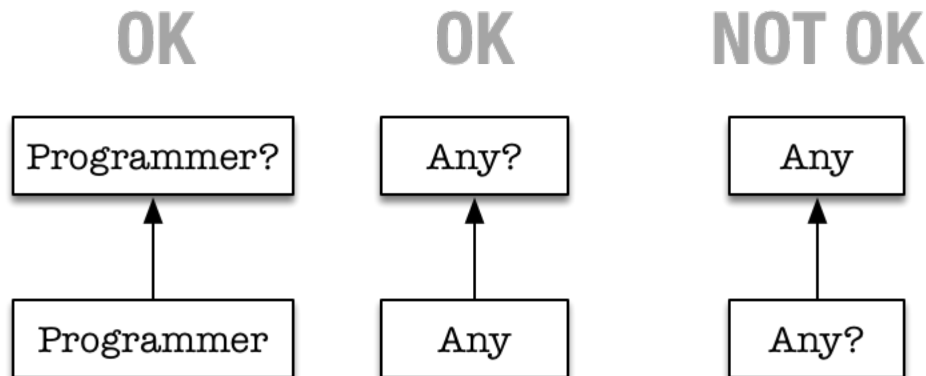
classchart

From figure above, we can say that;

- class **Any** is at the top of the class chart — class **Any** is the equivalent of java.lang.Object (in Kotlin)
- **Employee** is a subclass of **Any**. Employee has two types, the one that it inherited from Any, and itself — because the Employee class can define its own set of behavior (methods), so that counts as one type. `Employee` is user-defined class that we'll use for purposes of illustration
- **Programmer** is a subclass of **Employee** which is a subclass of **Any**, which means Programmer has 3 types. One from Any, another from Employee and another coming from the Programmer class itself. `Programmer` is also a user-defined class that we'll use for purposes of illustration, the rest of the classes and interfaces in our diagram are Kotlin classes/interfaces
- **Number** is a subtype of **Any**, but it also implements the **Comparable** interface. So, Number has 3 types, one from Any, another one from itself and another from the Comparable interface. We can say that Number is a subtype of Any and it's also a subtype of Comparable — whatever you expect the Comparable to do, the Number can do, whatever Any can do, Number can also do. This is basic OOP
- The **String** class has `4` types. One from **Any**, another from **Comparable**, another one from **CharSequence** and lastly, from its own class

From the statements and the diagram above, it's okay to use subclass and subtype interchangeably. There's not much difference between the two. Their difference will become apparent when we start considering nullable types.

When we go to nullable types, it'll become clearer why a subclass is not always the same as a subtype.



When you put a question mark after the name of a type, it becomes the nullable version of that type. In Kotlin, we can create two types from the same class — the nullable and the non-nullable version. We can't really say **Programmer** is a subclass of **Programmer?** because there is just one class definition for Programmer, but **Programmer** (the non-nullable version) is a subtype of **Programmer?** (the nullable one). Similarly, **Any** is a subtype of **Any?** but **Any?** is not a subtype of **Any** — the reverse direction isn't true.

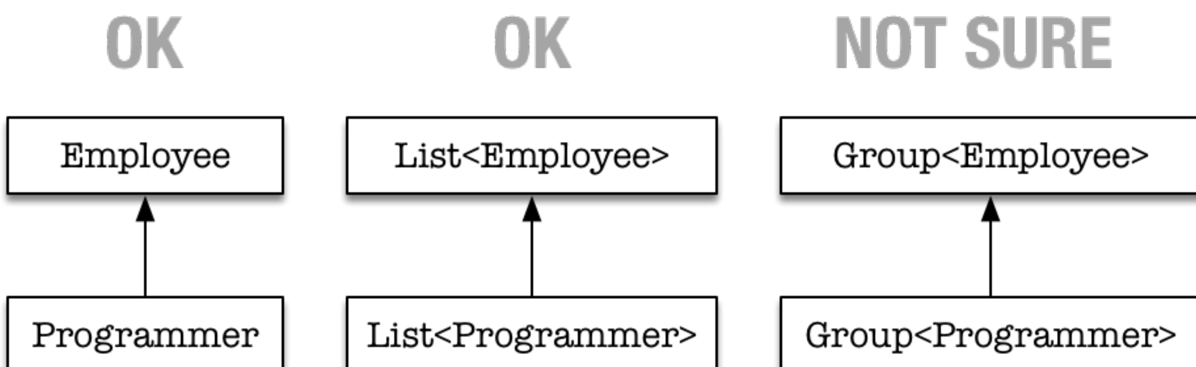
It's okay not write this

```
var j:Programmer? = Programmer("Ted") // assign non-null to nullable Programmer
j = null. // then we assign a null to j
```

But it's not okay to write this

```
var i:Programmer = j // assign j (which is null) to non-nullable Programmer
```

Next, we explore some more examples of when classes are not the same types in Kotlin. Let's look at some more diagrams.



Activities and UI

this page left blank

Dealing with Threads

this page left blank