Kotlin Syntax

Produced by:

Dr. Siobhán Drohan (sdrohan@wit.ie)

Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

Kotlin Syntax

Sources: http://kotlinlang.org/docs/reference/basic-syntax.html
http://petersommerhoff.com/dev/kotlin/kotlin-for-java-devs/

Agenda

- Basic Types
- Local Variables (val & var)
- Functions
- Control Flow (if, when, for, while)
- Strings & String Templates
- Ranges (and the ***in*** operator)
- Type Checks & Casts
- Null Safety
- Comments

Basic Types

Numbers, characters and booleans.

Basic Types

In Kotlin, everything is an object in the sense that we can call member functions and properties on any variable.

Basic Types - Numbers

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

Basic Types - Numbers

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

```
val doubleNumber: Double = 100.45
val floatNumber: Float = 100.45f
val longNumber: Long = 100
val intNumber: Int = 100
val shortNumber: Short = 100
val byteNumber: Byte = 100
```

Explicitly defining a numeric type

Basic Types - Numbers

Type inference

```
val doubleNumber = 100.45
val floatNumber = 100.45f
val longNumber = 100L
val intNumber = 100
val shortNumber = 100
val byteNumber = 100
```

Basic Types - Numbers

Type inference

```
val doubleNumber = 100.45
val floatNumber = 100.45f
val longNumber = 100L
val intNumber = 100
val shortNumber = 100
val byteNumber = 100
```

```
println("doubleNumber type is: " + doubleNumber.javaClass)
println("floatNumber type is:  " + floatNumber.javaClass)
println("longNumber type is:   " + longNumber.javaClass)
println("intNumber type is:    " + intNumber.javaClass)
println("shortNumber type is:  " + shortNumber.javaClass)
println("byteNumber type is:   " + byteNumber.javaClass)
```

Console

```
<terminated> Config - Main.kt [Java Appl
doubleNumber type is: double
floatNumber type is:  float
longNumber type is:   long
intNumber type is:    int
shortNumber type is:  int
byteNumber type is:   int
```

Basic Types - Numbers

```
val oneMillion = 1_000_000
val threeThousand = 3_000
val creditCardNumber = 1234_4321_5678_8765

fun main(args : Array<String>)
{
  println("" + oneMillion + " - the type is: " + oneMillion.javaClass)
  println("" + threeThousand + " - the type is: " + threeThousand.javaClass)
  println("" + creditCardNumber + " - the type is: " + creditCardNumber.javaClass)
}
```

You can use underscores to make number constants more readable.

Console

```
<terminated> Config - Main.kt [Java Application] C:\Progra
1000000 - the type is: int
3000 - the type is: int
1234432156788765 - the type is: long
```

Basic Types – Numbers: Explicit Conversions

In Kotlin, there are no implicit widening conversions for numbers i.e. smaller types (e.g. Byte) are not subtypes of bigger ones (e.g. Int)

→ smaller types are NOT implicitly converted to bigger types.

Basic Types – Numbers: Explicit Conversions

In Kotlin, there are no implicit widening conversions for numbers i.e. smaller types (e.g. Byte) are not subtypes of bigger ones (e.g. Int)

→ smaller types are NOT implicitly converted to bigger types.

```
val byteNumber: Byte = 10          //static type check: OK
val intNumber: Int = byteNumber //syntax error
```

BUT, we can use explicit conversions to widen numbers

```
val byteNumber: Byte = 10          //static type check: OK
val intNumber: Int = byteNumber.toInt()   //OK
```

Basic Types – Numbers: Explicit Conversions

Every number type supports the following conversions:

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

```
//Explicit Conversion
val intNumber: Int = byteNumber.toInt()
val floatNumber: Float = byteNumber.toFloat()
```

Basic Types – Characters

```
val aChar = 'a'
val bChar: Char = 'b'

fun main(args : Array<String>)
{
  println("" + aChar + " - the type is: " + aChar.javaClass)
  println("" + bChar + " - the type is: " + bChar.javaClass)
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0
a - the type is: char
b - the type is: char
```

Basic Types – Booleans

```
val aFlag = true
val bFlag: Boolean = false

fun main(args : Array<String>)
{
  println("" + aFlag + " - the type is: " + aFlag.javaClass)
  println("" + bFlag + " - the type is: " + bFlag.javaClass)
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0_77\bin\
true - the type is: boolean
false - the type is: boolean
```

Basic Types – Escape Characters

Special characters can be escaped using a backslash:

\t \b \n \r \’ \“ \\ \$

```
val aFlag= true
val bFlag: Boolean = false

fun main(args : Array<String>)
{
  println("" + aFlag + " - the type is: \n\t\t" + aFlag.javaClass)
  println("" + bFlag + " - the type is: \n\t\t" + bFlag.javaClass)
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0
true - the type is:
                boolean
false - the type is:
                boolean
```

Local Variables

val (read-only) and var (mutable)

Local Variables – val (read-only)

Assign-once (read-only) local variable:

```
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
```

```
a = 1, b = 2, c = 3
```

Target platform: JVM Running on kotlin v. 1.1.50

Local Variables – val (read-only)

Assign-once (read-only) local variable:

```
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
c = 4           // Syntax error: cannot reassign value
```

Target platform: JVM Running on kotlin v. 1.1.50

Local Variables – var (mutable)

Mutable variable:

```
var x = 5 // `Int` type is inferred
x += 1    // Mathematical operations permitted on Int
```

```
x = 6
```

Target platform: JVM Running on kotlin v. 1.1.50

Local Variables – var (mutable)

Mutable variable:

```
var x = 5 // `Int` type is inferred
x += 1    // Mathematical operations permitted on Int
x = 5     // Allowed to reassign value
```

```
x = 5
```

Target platform: JVM Running on kotlin v. 1.1.50

Functions

Parameters, return types, expression body, inferred return type

Functions – parameters and return types

Function having two `Int` parameters with `Int` return type:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}

fun main(args: Array<String>) {
    print("sum of 3 and 5 is ")
    println(sum(3, 5))
}
```

```
sum of 3 and 5 is 8
```

Functions – expression body, inferred return type

Function “sum” with an expression body and inferred return type

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

fun main(args: Array<String>) {
    println("sum of 19 and 23 is ${sum(19, 23)}")
    println("sum of 19 and 23 is " + sum(19, 23))
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0_77\b
sum of 19 and 23 is 42
sum of 19 and 23 is 42
```

Functions – no return data

Function returning no meaningful value:

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}

fun main(args: Array<String>) {
    printSum(-1, 8)
}
```

```
sum of -1 and 8 is 7
```

`Unit` return type can be omitted:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}

fun main(args: Array<String>) {
    printSum(-1, 8)
}
```

```
sum of -1 and 8 is 7
```

Control Flow

if, when, for, while

Control Flow – if

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

fun main(args: Array<String>) {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

```
max of 0 and 42 is 42
```

The traditional way to write if's

Control Flow – if

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

fun main(args: Array<String>) {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

```
max of 0 and 42 is 42
```

The traditional way to write if's

HOWEVER….in Kotlin, *if* is an expression, i.e. it returns a value. Therefore there is <u>no ternary operator</u> (condition ? then : else), because ordinary *if* works fine in this role.

Control Flow – if

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

fun main(args: Array<String>) {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

```
max of 0 and 42 is 42
```

Using if as an expression

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

fun main(args: Array<String>) {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

```
max of 0 and 42 is 42
```

Control Flow – if

```
// Traditional usage
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

// As expression
val max = if (a > b) a else b
```

Some examples without using functions.

The first two examples use *if* as a statement.

The last example uses *if* as an expression.

Control Flow – if

if branches can also be blocks.
The last expression is the value of a block:

```
val a = 10;
val b = 5;
val max = if (a > b) {
              print ("Choose a")
              a
            }
            else {
              print ("Choose b")
              b
            }
```

Control Flow – if

if branches can also be blocks.
The last expression is the value of a block:

```
val a = 10;
val b = 5;
val max = if (a > b) {
                print ("Choose a")
                a
            }
            else {
                print ("Choose b")
                b
            }
```

When *if* is used as an expression, the *else* part is mandatory.

Console

```
<terminated> Config - Main.kt [Java Application] C:\Progra
Choose a
```

Control Flow – when

Replaces switch in Java

```
val x = 10;
when (x){
   1 -> print("x is 1")
   2 -> print("x is 2")
   in 3..10 -> print ("x is between 3 and 10")
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java
x is between 3 and 10
```

Control Flow – when

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // Note the block
        print("x is neither 1 nor 2")
    }
}
```

Control Flow – when

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

Branch conditions may be combined with a comma.

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

We can use arbitrary expressions (not only constants) as branch conditions.

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is outside the range")
    else -> print("none of the above")
}
```

We can also check a value for being *in* or *!in* a range or a collection.

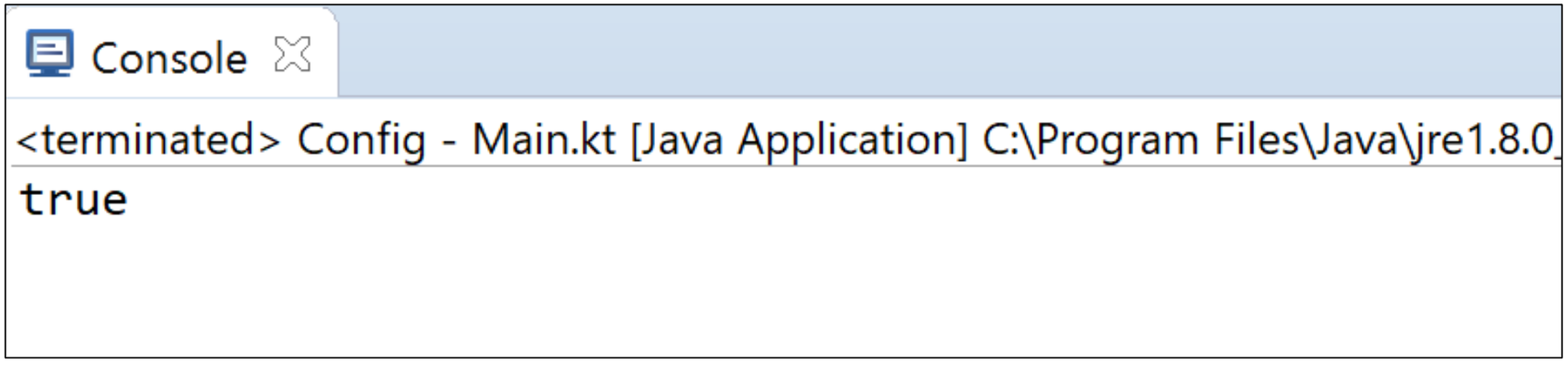
Control Flow – when

```
val x = "I am a String"

val contains = when (x){
    is String  -> x.contains("I am a")
    else -> false
}

println(contains)
```

Another possibility is to check that a value *is* or *!is* of a particular type.



Control Flow – when

when can also be used as a replacement for an *if-else if* chain.

If no argument is supplied, the branch conditions are simply boolean expressions, and a branch is executed when its condition is true.

```
val aString = "I am a String"

when {
  aString.equals("I am a String") -> println("Equal");
  else -> println("Not Equal")
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0_77\bin\javav
Equal
```

Control Flow – when

```
fun describe(obj: Any): String =
when (obj) {
    1           -> "One"
    "Hello"     -> "Greeting"
    is Long     -> "Long"
    !is String  -> "Not a string"
    else        -> "Unknown"
}

fun main(args: Array<String>) {
    println(describe(1))
    println(describe("Hello"))
    println(describe(1000L))
    println(describe(2))
    println(describe("other"))
}
```

```
One
Greeting
Long
Not a string
Unknown
```

Control Flow – for

The *for* loop iterates through anything that provides an iterator. It is similar to the *for-each* loop in Java.

```
for (item in collection) print(item)
```

```
fun main(args: Array<String>) {
    val items = listOf("apple", "banana", "kiwi")
    for (item in items) {
        println(item)
    }
}
```

```
apple
banana
kiwi
```

Control Flow – for

If you want to iterate through an array or a list with an index, you can do it this way:

```
for (i in array.indices) {
    print(array[i])
}
```

```
fun main(args: Array<String>) {
    val items = listOf("apple", "banana", "kiwi")
    for (index in items.indices) {
        println("item at $index is ${items[index]}")
    }
}
```

```
item at 0 is apple
item at 1 is banana
item at 2 is kiwi
```

Control Flow – for

Alternatively, you can use the withIndex library function:

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

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

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

  for ((index, value) in items.withIndex()) {
    println("the element at ${index} is ${value}")
  }
}
```

Console Gradle Tasks Gradle Executions

```
<terminated> Config - Main.kt [Java Application] C:\Program File
the element at 0 is apple
the element at 1 is banana
the element at 2 is kiwi
```

Control Flow – while

The *while* and *do-while* work as usual:

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

Note: Kotlin also supports traditional *break* and *continue* operators in loops.

Control Flow – while

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

var index = 0
while (index < items.size){
   println("item at $index is ${items[index]}")
   index++
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0_77\bin\
item at 0 is apple
item at 1 is banana
item at 2 is kiwi
```

Strings and String Templates

Escaped strings, raw strings, literals, templates

Strings

- Strings are represented by the type String.
- Strings are immutable.
- 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) {
    println(c)
}
```

String Literals

- Kotlin has two types of string literals:
  escaped strings that may have escaped characters in them and
  raw strings that can contain newlines and arbitrary text.

String Literals

- Kotlin has two types of string literals:
  escaped strings that may have escaped characters in them and
  raw strings that can contain newlines and arbitrary text.

An escaped string is very much like a Java string

```
val s = "Hello, world!\n"
```

A raw string is delimited by a triple quote (“”“”), contains no escaping and can contain new lines and any other characters.

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

String Literals

```
Console
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.
Tell me and I forget.
Teach me and I remember.
Involve me and I learn.
(Benjamin Franlkin)
```

You can remove leading whitespace with trimMargin()

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

String Literals

By default | is used as margin prefix, but you can choose another character and pass it as a parameter like trimMargin(">").

```
val text = """
        >   Tell me and I forget.
        >   Teach me and I remember.
        >   Involve me and I learn.
        >   (Benjamin Franlkin)
         """.trimMargin(">")

print(text)
```

Note the impact of the two spaces we put between > and the text

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Ja
  Tell me and I forget.
  Teach me and I remember.
  Involve me and I learn.
  (Benjamin Franlkin)
```

String Templates

- Strings may contain template expressions, i.e. pieces of code that are evaluated and whose results are concatenated into the string.
- A template expression starts with a dollar sign ($) and consists of either a simple name:

```
val i = 10
val s = "i = $i" // evaluates to "i = 10"
```

- or an arbitrary expression in curly braces:

```
val s = "abc"
val str = "$s.length is ${s.length}" // evaluates to "abc.length is 3"
```

String Templates

Templates are supported both inside **raw strings** and inside **escaped strings**.

```
val anInt = 10
val aString = "Value of anInt is ${anInt}\n"

val text = """
        >   Tell me and I forget.
        >   Teach me and I remember.
        >   Involve me and I learn.
        >   (Benjamin Franlkin)
         """.trimMargin(">")

print(aString)
print(text)
```

Console

```
<terminated> Config - Main.kt [Java Applicati
Value of anInt is 10
  Tell me and I forget.
  Teach me and I remember.
  Involve me and I learn.
  (Benjamin Franlkin)
```

String Templates

```
fun main(args: Array<String>) {
    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"
    println(s2)
}
```

s1

"a is 1"

```
a was 1, but now is 2
```

Ranges

The in operator

Range

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

```
fun main(args: Array<String>) {
    val x = 10
    val y = 9
    if (x in 1..y+1) {
        println("fits in range")
    }
}
```

```
fits in range
```

Check if a number is out of range:

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

```
-1 is out of range
list size is out of valid list indices range too
```

Range

Iterating over a range:

```
fun main(args: Array<String>) {
    for (x in 1..5) {
        print(x)
    }
}
```

```
12345
```

Iterating over a progression:

```
fun main(args: Array<String>) {
    for (x in 1..10 step 2) {
        print(x)
    }
    for (x in 9 downTo 0 step 3) {
        print(x)
    }
}
```

```
135799630
```

Type Checks & Casts

is and !is operators

```
fun main(args: Array<String>) {
   val aString = "I am a String"

   if (aString is String) {
      println("String length is: ${aString.length}")
   }

   if (aString !is String) { // same as !(aString is String)
      print("Not a String")
   }
   else {
      println("String length is: ${aString.length}")
   }
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0_77\
String length is: 13
String length is: 13
```

Smart Casts (an example using if)

```
fun main(args: Array<String>) {
     demo ("I am a String")
     demo (12)
}

fun demo(x: Any) {
    if (x is String) {
        println(x.length) // x is automatically cast to String
    }
    else{
        println(x.javaClass)
    }
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jre1.8.0_77\bin\javav
13
class java.lang.Integer
```

Smart Casts (an example using when)

```
fun main(args: Array<String>) {
    demo (12)
    demo ("I am a String")
    demo (intArrayOf(1,2,3,4))
}

fun demo(x: Any) {
    when (x) {
        is Int -> println(x + 1)
        is String -> println(x.length + 1)
        is IntArray -> println(x.sum())
    }
}
```

Console

```
<terminated> Config - Main.kt [Java Application] C:\P
13
14
10
```

Null

Using nullable values and checking for null

Null Safety

In Kotlin, the type system distinguishes between references that can hold ***null (nullable references)*** *and those that* ***cannot (non-null references).***

The Kotlin compiler makes sure you don't, by accident, operate on a variable that is null.

Null Safety – a non-null reference

A regular variable of type String can not hold null

```
var a: String = "abc"
a = null // syntax error
```

Calling a method / accessing a property on variable a, is guaranteed not to cause an NullPointerException

```
val l = a.length
```

Null Safety – a nullable reference

To allow nulls, we can declare a variable as nullable string, written String?

```
var b: String? = "abc"
b = null // ok
```

```
val l = b.length    // syntax error: variable
                    //'b' can be null

                    // …many ways around this…
```

Null Safety – a nullable reference

To allow nulls, we can declare a variable as nullable string, written String?

```
var b: String? = "abc"
b = null // ok
```

Option 1: you can explicitly check if b is null, and handle the two options separately:

```
val l = if (b != null) b.length else -1
```

Null Safety – a nullable reference

To allow nulls, we can declare a variable as nullable string, written String?

```
var b: String? = "abc"
b = null // ok
```

Option 2: you can use the safe call operator ?.
This returns **b.length** if **b** is not null, and null otherwise.

```
b?.length
```

Null Safety – a nullable reference

To allow nulls, we can declare a variable as nullable string, written String?

```
var b: String? = "abc"
b = null // ok
```

Option 3: you can use the **!!** Operator. This force a call to our method and will return a non-null value of **b** or throw an NPE if **b** is null. Use sparingly!

```
val l = b!!.length
```

Null Safety – The Elvis Operator, ?:

When we have a nullable reference **r**, we can say:

"if **r** is not null, use it, otherwise use some non-null value **x**"

```
val l: Int = if (b != null) b.length else -1
```

Null Safety – The Elvis Operator, ?:

When we have a nullable reference **r**, we can say:

"if **r** is not null, use it, otherwise use some non-null value **x**"

```
val l: Int = if (b != null) b.length else -1
```

Along with the complete if-expression, this can be expressed with the Elvis operator, written **?:**

```
val l = b?.length ?: -1
```

If the expression to the left of **?:** is not null, the elvis operator returns it, otherwise it returns the expression to the right.

Nullable – nullable returns

A reference must be explicitly marked as nullable (i.e. ?) when null value is possible.

Return null if the return value does not hold an integer:

```
fun parseInt(str: String): Int? {
    // ...
}
```

Comments

Single line, block, KDoc

Comments – single line and block 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.

Comments – KDoc (equivalent to JavaDoc)

```
/**
 * A group of *members*.
 *
 * This class has no useful logic; it's just a documentation example.
 *
 * @param T the type of a member in this group.
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
class Group<T>(val name: String) {
    /**
     * Adds a [member] to this group.
     * @return the new size of the group.
     */
    fun add(member: T): Int { ... }
}
```

Comments – KDoc

Block tags	Currently supported KDoc block tags
@param <name>	Documents a value parameter of a function or a type parameter of a class, property or function.
@return	Documents the return value of a function.
@constructor	Documents the primary constructor of a class.
@receiver	Documents the receiver of an extension function.
@property <name>	Documents the property of a class which has the specified name.
@throws <class>, @exception <class>	Documents an exception which can be thrown by a method.
@sample <identifier>	Embeds the body of the function with the specified qualified name into the documentation for the current element, in order to show an example of how the element could be used.
@see <identifier>	Adds a link to the specified class or method to the **See Also** block of the documentation.
@author	Specifies the author of the element being documented.
@since	Specifies the version of the software in which the element being documented was introduced.
@suppress	Excludes the element from the generated documentation. Can be used for elements which are not part of the official API of a module but still have to be visible externally.

For more info: http://kotlinlang.org/docs/reference/kotlin-doc.html