LMU

LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

Praktikum Mobile und Verteilte Systeme

Kotlin 101

Prof. Dr. Claudia Linnhoff-Popien
Steffen Illium, Stefan Langer,
André Ebert
http://www.mobile.ifi.lmu.de

SoSe 2019

Structure and Content

- What and Why is Kotlin?
- Variables & DataTypes
- Operators
- Control Flow
- Nullable Types & Null Safety
- Kotlin Functions
- OOP: Classes & Objects
- Inheritanace & Overriding
- Abstract Classes & Data Classes
- Type Checking

Try and play with Kotlin at:

play.kotlinlang.org

Slides based on:

callicoder.com/categories/kotlin

Why and What is Kotlin

What is Kotlin?

- Kotlin is an OSS statically typed programming language that targets the JVM, Android, JavaScript and Native
- The first official 1.0 release was in February 2016
- The currently released version is 1.3.31, published on April 25, 2019.
- Developed by JetBrains
- Heavy support by Google as of I/O 2017

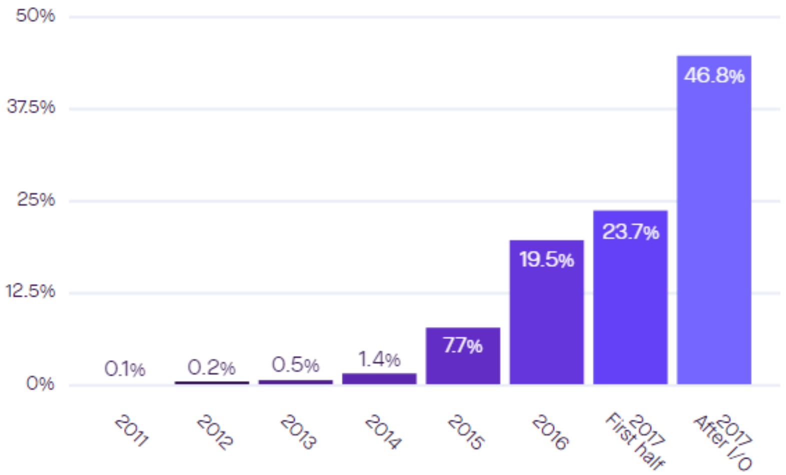
Why and What is Kotlin
Objectification

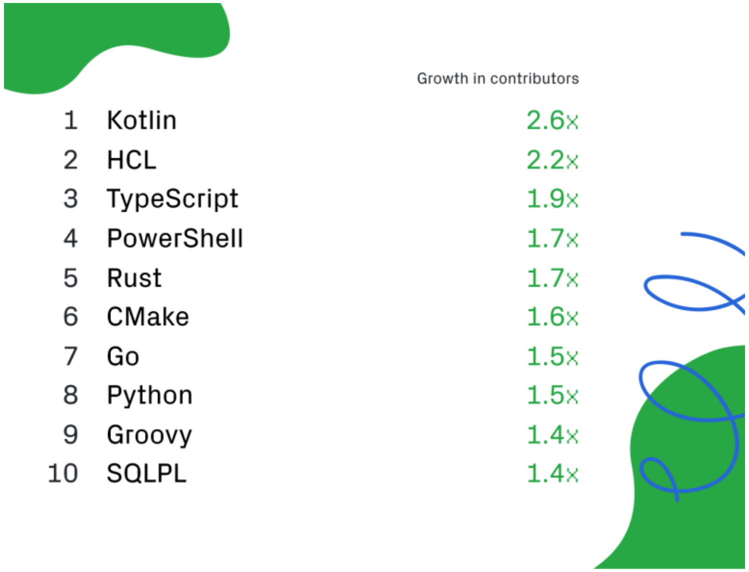
- Everything in Kotlin is an Objects
- There are no primitive Types like Int, Char, Double, Boolean
- Operations on those “Types” (Objects are represented as function calls)
- Kotlin supports functional style programming
- … as well as object oriented programming paradigms
- … has Null-Safety build in
- .. Is an compiled language

Why and What is Kotlin

Why is Kotlin?

- Fastest growing language at time
- Doubling its user base every year
- Popularity boosted at Google I/O

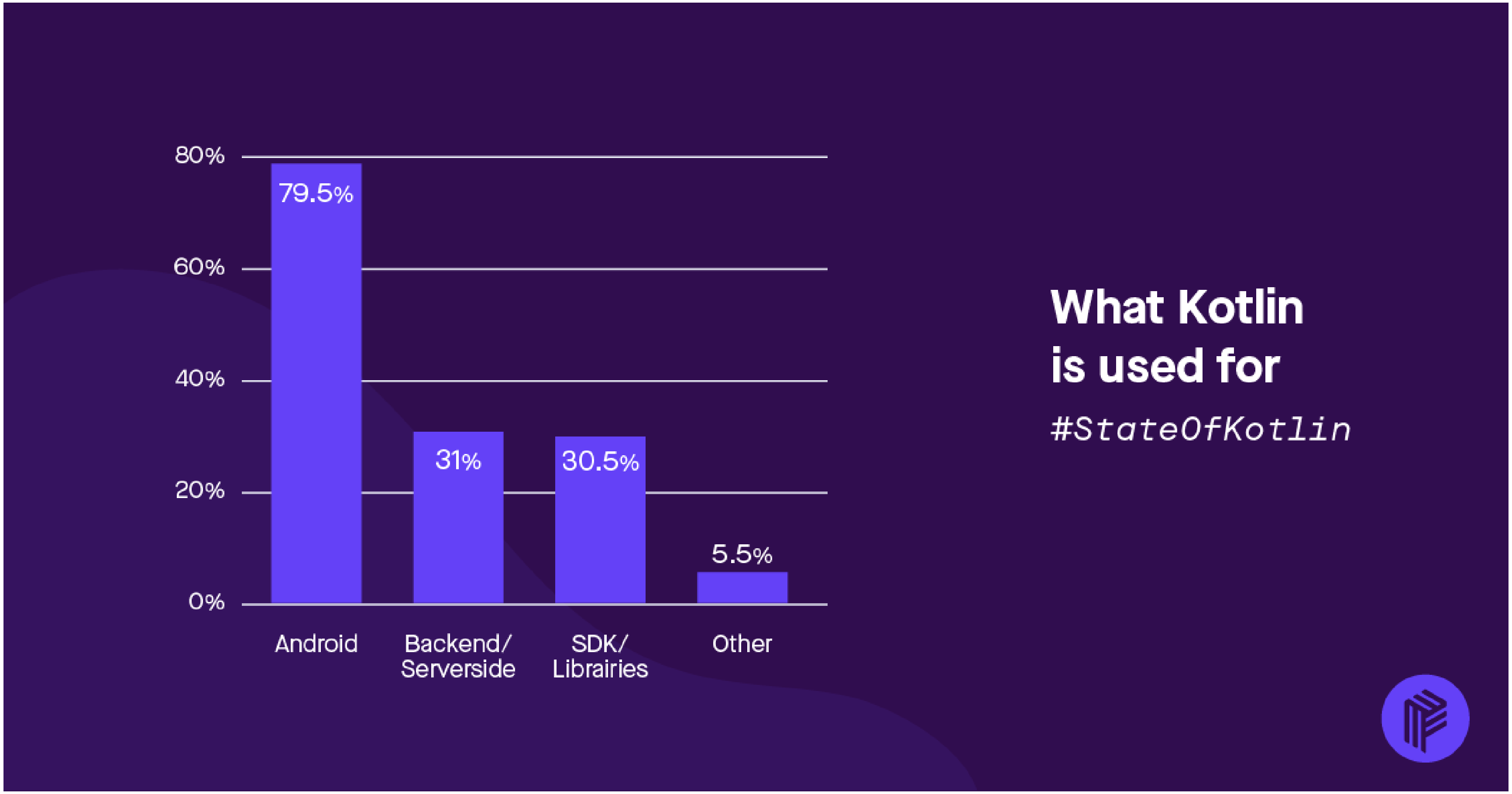




		Growth in contributors
1	Kotlin	2.6x
2	HCL	2.2x
3	TypeScript	1.9x
4	PowerShell	1.7x
5	Rust	1.7x
6	CMake	1.6x
7	Go	1.5x
8	Python	1.5x
9	Groovy	1.4x
10	SQLPL	1.4x

Why and What is Kotlin

Why is Kotlin?



Section 1: Variables and Data Types

```
Def:
   A data item that may take on more than one value
during the runtime of a program.
```

- Initialization and Assignment (var, val, =)
- Constants (val)
- Variables (var)
- Data Types (int, float, string, array)
- Type Conversion (toSomething)

Variables and DataTypes
Initialization

- Two Keywords:
  - val for fixed, immutable Constants (read-only)

```kotlin
val name = "Bill Gates"
name = "Satoshi Nakamoto"
// Error: Val cannot be reassigned
```

  - var for variable, mutable Variables (read-write)

```kotlin
var country = "USA"
country = "India" // Works
```

Variables and DataTypes
Type Inference

- Data Types of Variables are inferred from initializer expression.

```
val greeting = "Hello, World"
val year = 2018
```

- Data Types can be specified explicitly

```
val greeting: String = "Hello, World"
val year: Int = 2018
```

- Variables can be declared but need a Data Type

```
var language // Error: The variable must either have
a Type annotation or be initialized

var language: String // Works
language = "French"
```

Variables and DataTypes

Integers

```
•Byte           - 8 bit
•Short          - 16 bit
•Int            - 32 bit
•Long           - 64 bit
```

```kotlin
val myByte: Byte = 10
val myShort: Short = 125

// The suffix 'L' is used to specify a long value
val myInt = 1000
val myLong = 1000L
```

Variables and DataTypes

Floats

```
•Float  - 32 bit single-precision floating point value.
•Double - 64 bit double-precision floating point value.
```

```kotlin
// The suffix 'f' or 'F' represents a Float
val myFloat = 126.78f
val myDouble = 325.49
```

- Underscore to make numbers readable

```kotlin
val hundredThousand = 100_000
val oneMillion = 1_000_000
```

Variables and DataTypes

Booleans & Characters

- Booleans

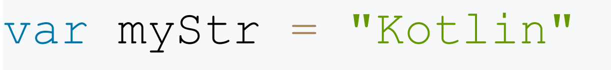
```kotlin
val myBoolean = true
val anotherBoolean = false
```

- Characters

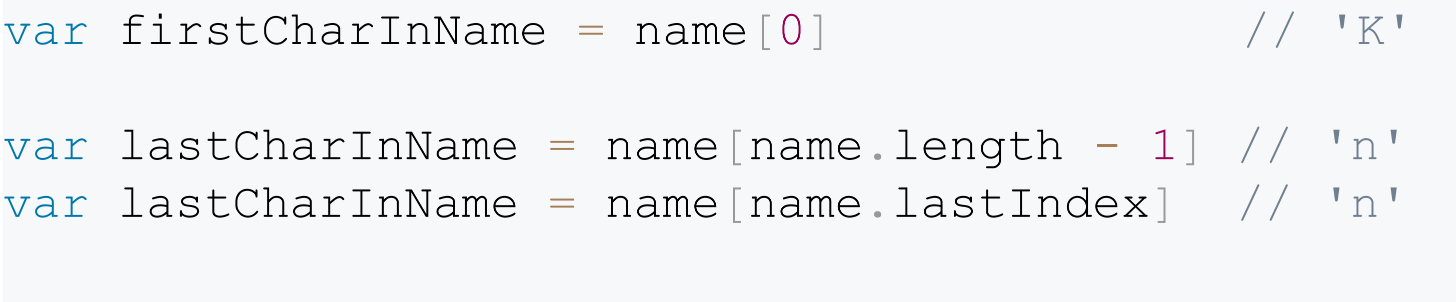
```kotlin
val letterChar = 'A'
val digitChar = '9'


// Special  Characters
// \n (newline), \t (tab),
// \r (carriage return), \b (backspace)
```

Variables and DataTypes
Strings and its variants

- Strings

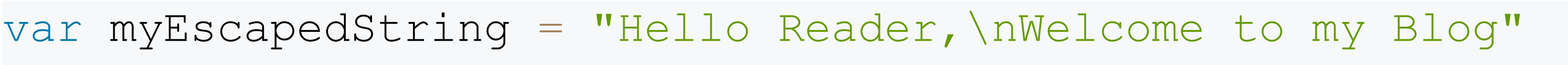


```
var myStr = "Kotlin"
```

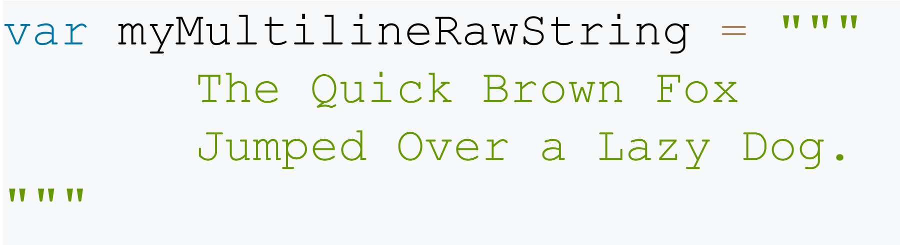


```
var firstCharInName = name[0]               // 'K'

var lastCharInName = name[name.length - 1]  // 'n'
var lastCharInName = name[name.lastIndex]   // 'n'
```

- Escaped Strings and Raw Strings



```
var myEscapedString = "Hello Reader,\nWelcome to my Blog"
```



```
var myMultilineRawString = """
      The Quick Brown Fox
      Jumped Over a Lazy Dog.
"""
```

Variables and DataTypes

Arrays

- Arrays in Kotlin are created by either arrayOf()

```kotlin
var numbers = arrayOf(1, 2, 3, 4, 5)
var animals = arrayOf("Cat", "Dog", "Lion", "Tiger")
```

```kotlin
// Works and creates an array of Objects
var mixedArray = arrayOf(1, true, 3, "Hello", 'A')
```

- … or the Array() constructor

```kotlin
// Paramters 1. Length; 2. Function that takes the Index
var mySquareArray = Array(5, {i -> i * i})
// [0, 1, 4, 9, 16]
```

Variables and DataTypes
Typing

- Kotlin doesn’t support every implicit type conversion

```
var myInt = 1000

// Compiler Error
var myLong: Long = myInt
```

- Helper Functions:

```
•toByte()
•toShort()
•toInt()
•toLong()
```

```
•toFloat()
•toDouble()
•toChar()
```

```
var myDouble = myInt.toLong()
var myStr = myLong.toString()     //"1000"
```

Variables and DataTypes

Array Typing

- Type Enforcement in general Arrays

```kotlin
var numArray = arrayOf<Int>(1, 2, 3, 4, 5)
var strArray = arrayOf<String>("Cat", "Dog", "Tiger")

// Compiler Error
var numArray = arrayOf<Int>(1, 2, 3, 4, "Hello")
```

- Primitive Arrays for better performance at runtime

```kotlin
val myCharArray = charArrayOf('K', 'O', 'T')

val myIntArray = intArrayOf(1, 3, 5, 7)
```

Variables and DataTypes

Array Indexing

- Accessing elements by array[index] notation
- Every array has a size property that return the number of elements
- Similar to String Manipulations

```
val myDoubleArray = arrayOf(4.0, 6.9, 1.7, 12.3, 5.4)
val firstElement = myDoubleArray[0]

val lastElement = myDoubleArray[myDoubleArray.size - 1]
val lastElement = myDoubleArray.last()
```

- Array modification by index

```
val a = arrayOf(4, 5, 7)              // [4, 5, 7]
a[1] = 10                             // [4, 10, 7]
```

Section 2: Operators

Def:

An Operator is a symbol or function that is denoting an operation (e.g. ×, +).

- Arithmetic operators (+, -, *, /, %)
- Comparison operators (==, !=, <, >, <=, >=)
- Assignment operators (+=, -=, *=, /=, %=)
- Increment & Decrement operators (++, --)

Operators
Examples

Just like any other language, operators are the atomic functions to manipulate values and variables.

```
var a = 10
var b = 20

var c = ((a + b) * (a + b))/2    // 450
var isALessThanB = a < b         // true

a++                              // a now becomes 11

b += 5                           // b equals to 25 now
```

Operators

Understanding Operators

Since everything in Kotlin is an object; Operators represent functions on Objects.

```
var a = 4
var b = 5


println(a + b)


// equivalent to
println(a.plus(b))
```

Expression	Translates to
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a++	a.inc()
a––	a.dec()
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a += b	a.plusAssign(b)
…	…

Operators

Boolean Type

- Kotlin supports following logical operators for performing operations on boolean types

```
•||    - Logical OR
•&&    - Logical AND
•!     - Logical NOT
```

- Examples:

```
2 == 2 && 4 != 5              // true
4 > 5 && 2 < 7                // false
!(7 > 12 || 14 < 18)          // false
```

Operators

String Operations

- The + operator is overloaded for String Types to perform string concatenations

```
var firstName = "Kot"
var lastName = "Lin"
var fullName = firstName +" "+ lastName    // "Kot Lin"
```

- Variable names can be used to insert a template expression inside a String
- Even evaluations of code fragments can be inserted this way

```
var a = 12
var b = 18

// Prints - Avg of 12 and 18 is equal to 15
println("Avg of $a and $b is equal to ${ (a + b)/2 }")
```

Section 3: Control Flow

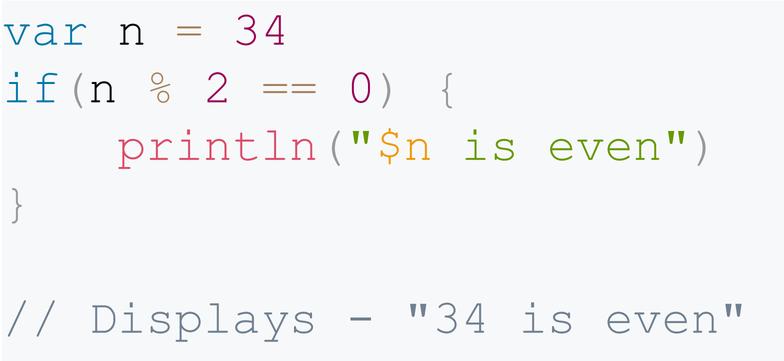
Content

> Def:
>
> In computer science, control flow (or flow of control) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.

- Conditional Expressions (if, if-else, when)
- The “in”-operator and ranges (1..10, in)
- The “is”-operator and types (Int, String, is)
- Looping Statements (for, while, and do-while)

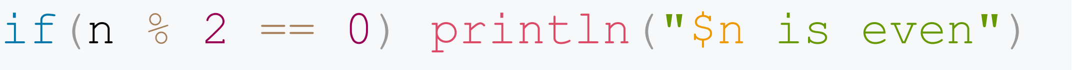
Control Flow
If - Statement

- The `If` statement allows you to specify a section of code that is executed only if a given condition is true



```
var n = 34
if(n % 2 == 0) {
    println("$n is even")
}

// Displays - "34 is even"
```

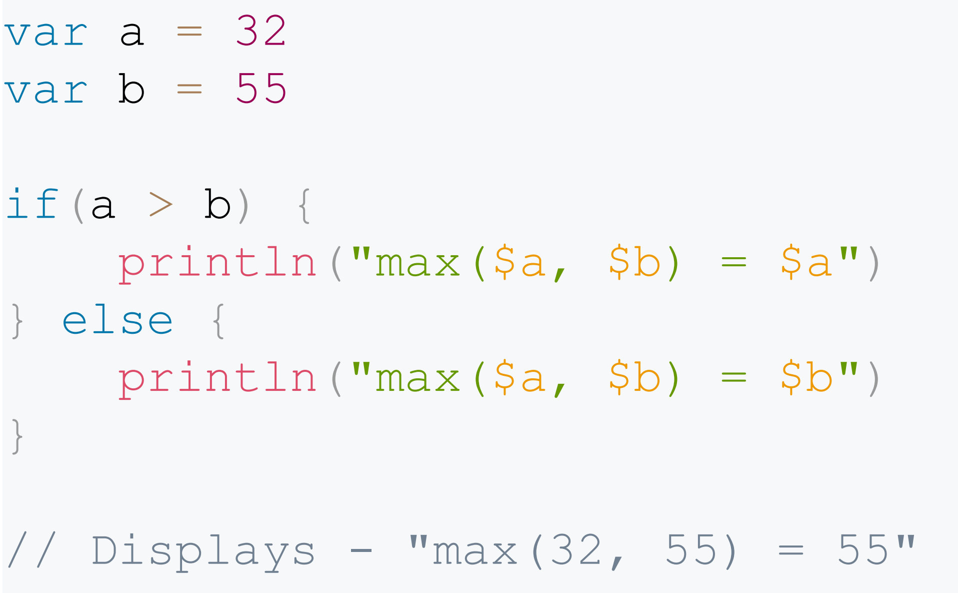
- Curly braces are optional if body is on a single line



```
if(n % 2 == 0) println("$n is even")
```

Control Flow
If-Else – Statement

- The `If` statement executes one section of code if the condition is true and the other if the condition is false

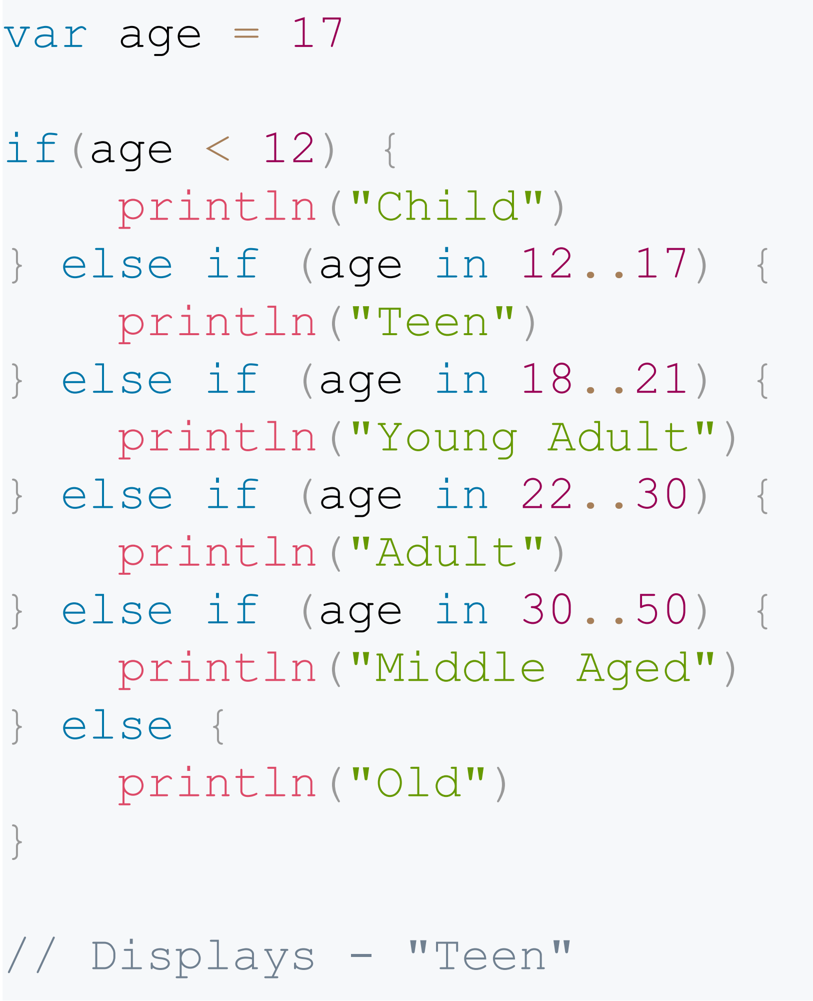


```
var a = 32
var b = 55

if(a > b) {
    println("max($a, $b) = $a")
} else {
    println("max($a, $b) = $b")
}

// Displays - "max(32, 55) = 55"
```

Control Flow

If-Else-Chaining

- Chaining `If-else-if` like usual; First true condition triggers



```
var age = 17

if(age < 12) {
    println("Child")
} else if (age in 12..17) {
    println("Teen")
} else if (age in 18..21) {
    println("Young Adult")
} else if (age in 22..30) {
    println("Adult")
} else if (age in 30..50) {
    println("Middle Aged")
} else {
    println("Old")
}

// Displays - "Teen"
```

Control Flow

If – As Expression

- Using the `If` statement as an expression instead of a statement
- Assignment of `If-Else` expression to a variable

```
var a = 32
var b = 55


var max = if(a > b) a else b

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


// Displays - "max(32, 55) = 55"
```

- `If` as an expression always needs an `else` branch
- Kotlin has <u>no</u> ternary operator like in java

Control Flow

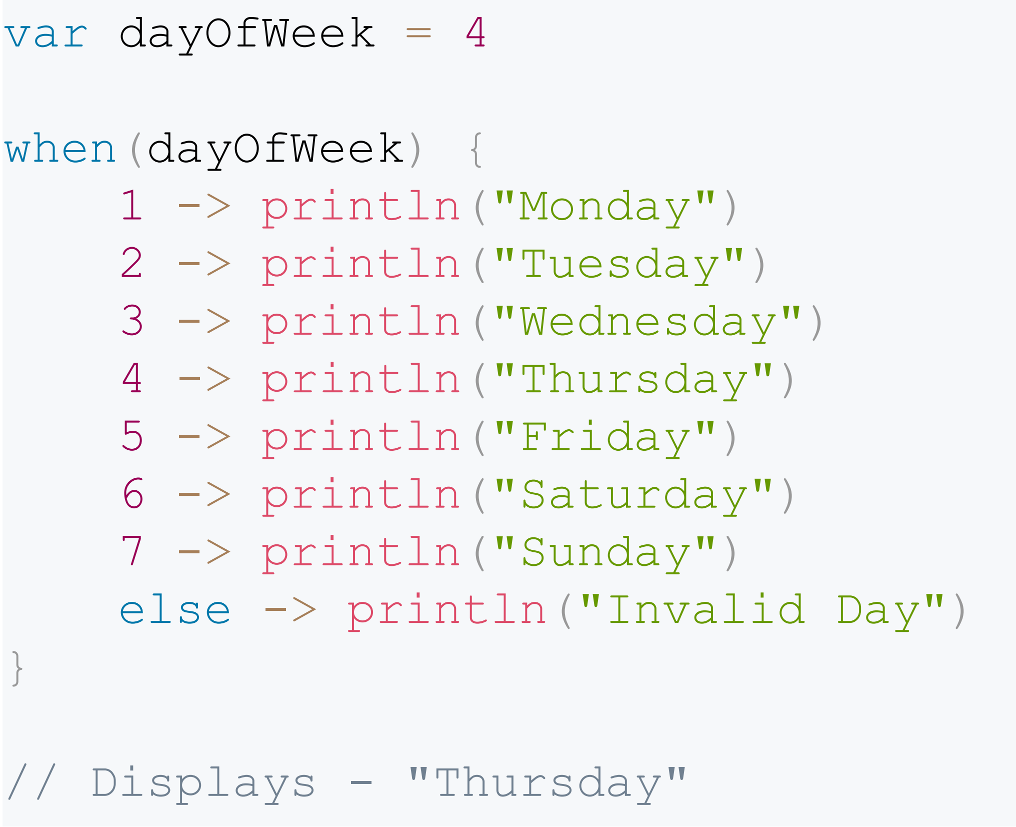
If – As Expression II

- The `If-Else` branches can also have block bodies.
- The last expression is the value of the block

```
var a = 32
var b = 55

var max = if(a > b) {
    println("$a is greater than $b")
    a
} else {
    println("$a is less than or equal to $b")
    b
}

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

// 32 is less than or equal to 55
```

Control Flow
When

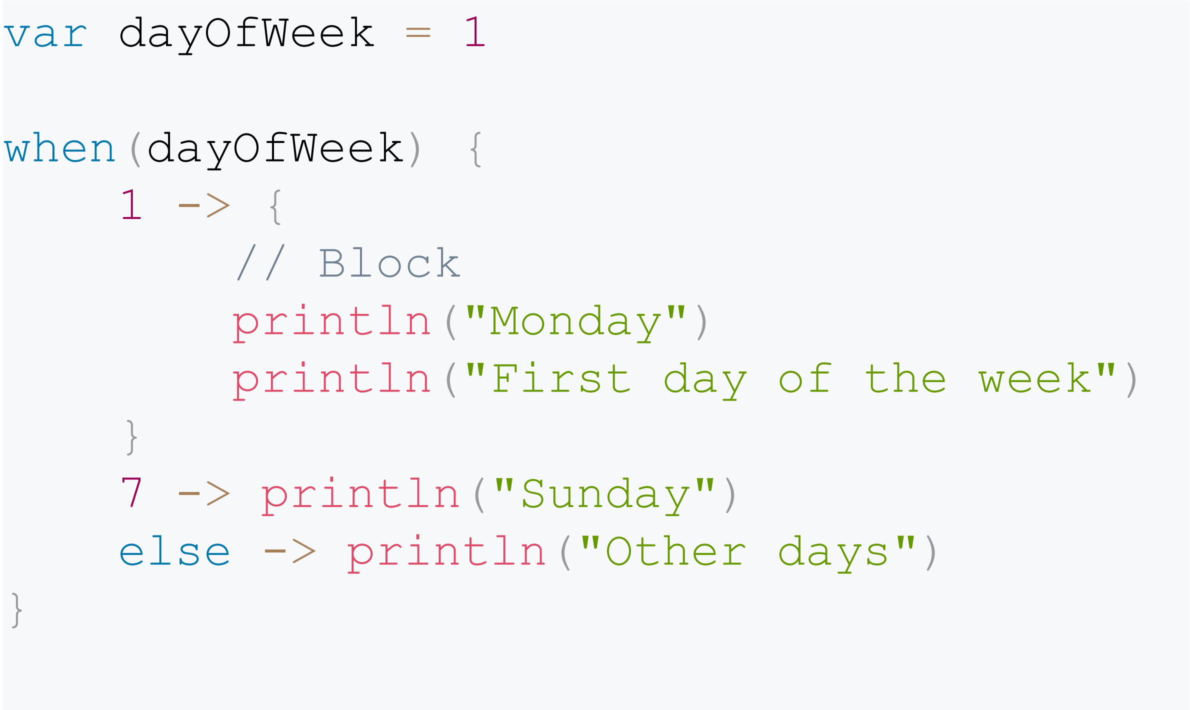
- Like a switch statement, matches the argument with all branches one by one
- If no match is found, the `else` branch is executed



```
var dayOfWeek = 4

when(dayOfWeek) {
    1 -> println("Monday")
    2 -> println("Tuesday")
    3 -> println("Wednesday")
    4 -> println("Thursday")
    5 -> println("Friday")
    6 -> println("Saturday")
    7 -> println("Sunday")
    else -> println("Invalid Day")
}

// Displays - "Thursday"
```

Control Flow
When Blocks

- Can also execute a block of multiple statements
- If no match is found, the `else` branch is executed



```
var dayOfWeek = 1

when(dayOfWeek) {
    1 -> {
        // Block
        println("Monday")
        println("First day of the week")
    }
    7 -> println("Sunday")
    else -> println("Other days")
}
```

Control Flow
Multiple when Expressions

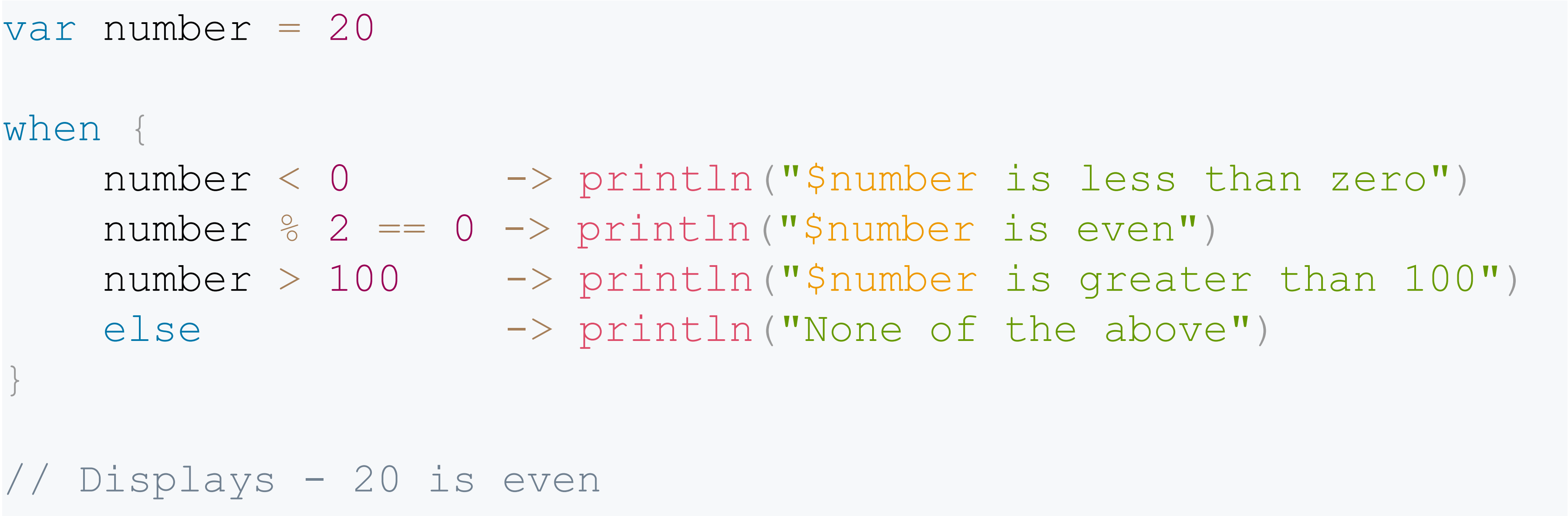
- Just like if, when can be used as an expression
- An `else` branch is required when used as an expression
- Additionally, branches can be combined using comma

```
var dayOfWeek = 1

var dayType = when(dayOfWeek) {
    1, 2, 3, 4, 5    -> „a Weekday"
    6, 7             -> "Weekend"
    else             -> "Invalid Day")
}

println("Today is $dayType")
// Today is a Weekday
```

- Helpful when running a common logic for multiple cases

Control Flow
When as if-else-if replacement

- When can replace if-else-if statements and assignments
- Conditions can replace supplied arguments



```kotlin
var number = 20

when {
    number < 0      -> println("$number is less than zero")
    number % 2 == 0 -> println("$number is even")
    number > 100    -> println("$number is greater than 100")
    else            -> println("None of the above")
}

// Displays - 20 is even
```

Control Flow
Is it in range?

- The `in` operator allows to check if a value belongs to a range or collection
- A `range` is defined by using the `..` Operator

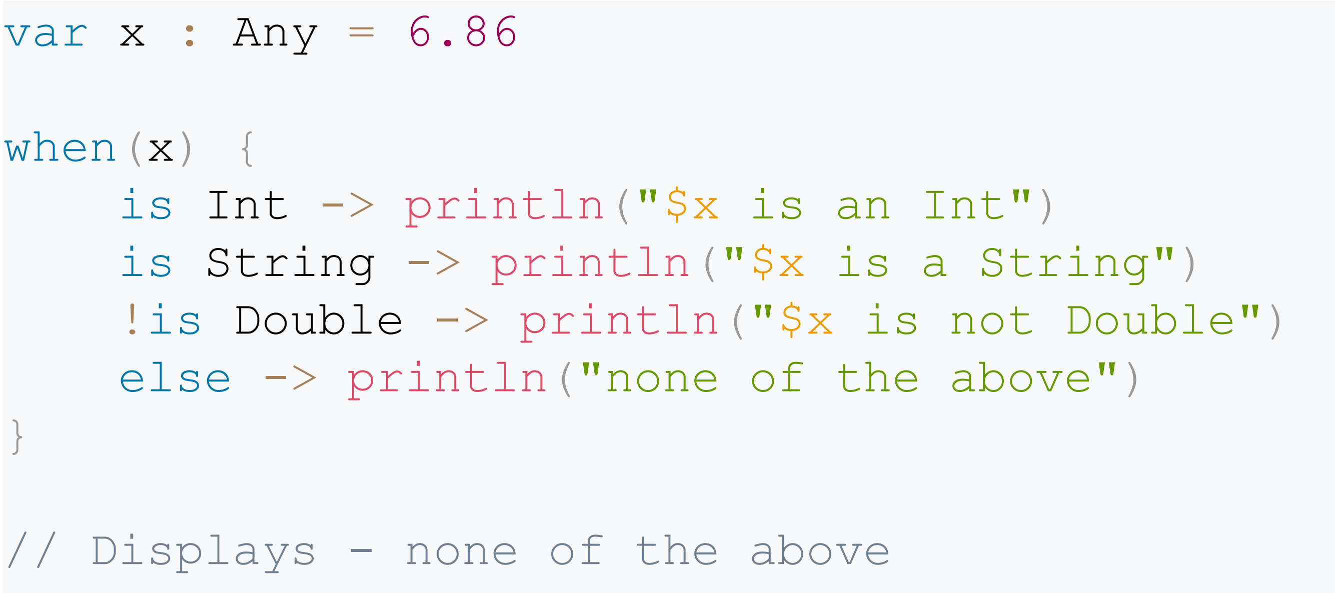
```
var dayOfMonth = 5

when(dayOfMonth) {
    in 1..7 -> println(„It's the first week")
    !in 15..21 -> println(„It's not the third week")
    else -> println("none of the above")
}

// Displays – It‛s the first week
```

- Reversed or spaced ranges are defined using `downTo` & the `step` keyword

```
when(dayOfMonth) {
    in 1 downTo 7 step 2 -> println(„It's the first week")
else -> println("none of the above")
}
```

Control Flow
Is it of type?

- The `is` operator allows to check if a value is of a certain `type`



```kotlin
var x : Any = 6.86

when(x) {
    is Int -> println("$x is an Int")
    is String -> println("$x is a String")
    !is Double -> println("$x is not Double")
    else -> println("none of the above")
}

// Displays - none of the above
```

Control Flow

Loops - While

```
Def:
    A Loop is a programmed sequence of instructions
that is repeated until or while a particular condition
is satisfied.
```

- While loop executes a block of code repeatedly as long as a given condition is `true`

```
while(condition) {
    // code to be executed
}
```

- Example:

```
var x = 1

while(x <= 5) {
    println("$x ")
    x++
}

// Displays – 1 2 3 4 5
```

Control Flow

Loops – do-while

- The `do-while` loop is similar to while loop except that it tests the condition at the end of the loop.
- Since `do-while` loop tests the condition at the end of the loop. It is executed <u>at least once</u>

```
do {
    // code to be executed
} while(condition)
```

- Example:

```
var x = 6

do {
    print("$x ")
    x--
} while(x == 5)

// Displays - 6 5
```

Control Flow

Loops – for

- The `for` is used to iterate over objects that provide an iterator; e.g. ranges, arrays, collections, …

```
for (name in itarator) {
    // code to be executed
}
```

- Example:

```
var primeNumbers = intArrayOf(2, 3, 5, 7, 11)

for(number in primeNumbers) {
    print("$number ")
}

// Displays - 2, 3, 5, 7, 11
```

Control Flow

Loops – iterating over arrays

- Every array in Kotlin has a property called indices which returns a range of valid indices of that array

```kotlin
var pNums = intArrayOf(2, 3, 5, 7, 11)

for(index in pNums.indices) {
    println("Prime#(${index+1}): ${pNums[index]}")
}
```

- withIndex() provides access to both, index and corresponding array element at the same time

```kotlin
var pNums = intArrayOf(2, 3, 5, 7, 11)

for((index, number) in pNums.withIndex()) {
    println("Prime#(${index+1}): $number")
}
```

Section 4: Null Safety

```
Citation:
    I call it my billion-dollar mistake. It was the
invention of the null reference in 1965. At that time,
I was designing the first comprehensive type system for
references in an object oriented language (ALGOL W). My
goal was to ensure that all use of references should be
absolutely safe, with checking performed automatically
by the compiler. But I couldn’t resist the temptation
to put in a null reference, simply because it was so
easy to implement. This has led to innumerable errors,
vulnerabilities, and system crashes, which have
probably caused a billion dollars of pain and damage in
the last forty years.
(Excerpt from Wikipedia)

-Sir Tony Hoare, The inventor of null reference
```

Null Safety

Content

- Nullability & Nullable Types
- How to Work with Nullable Types
  - Null Checks
  - Safe Call Operator
  - Elvis Operator
  - Not Null Assertions

Null Safety
Nullable Types

- Kotlin supports null ability as part of its type System
- Which means: Declaration whether a variable can hold a null value or not

- Compiler can detect NullPointerException and reduce them at runtime

- All variables in Kotlin are non-nullable by default…

```
var greeting: String = "Hello, World"
greeting = null // Compilation Error
```

- …but can be declared as nullable in its type declaration

```
var nullableGreeting: String? = "Hello, World"
nullableGreeting = null // Works
```

Null Safety

Nullable Types II

- It is known that “greeting” can never be null therefore the function call compilesas expected

```
val len = greeting.length
val upper = greeting.toUpperCase()
```

- Function calls to nullable variables are disallowed in compile time

```
val len = nullableGreeting.length
// Compilation Error
val upper = nullableGreeting.toUpperCase()
// Compilation Error
```

- This prevents the code from breaking at runtime

Null Safety
Safe ways to work with nulltype variables

- How is it then possible to call functions on nullable variables?

1. Null Checking
2. Safe call operator: `?.`
3. Elvis operator: `?:`
4. Not null Assertion: `!!`

Null Safety
1 Null Checking

- Null-checks are performed before working with a variable

```kotlin
val nullName: String? = "John"

if(nullName != null) {
    println("Hello, ${nullName.toUpperCase()}.")
    println(„Your length is ${nullName.length}.")
} else {
    println("Hello, Guest")
}
```

- The compiler “remembers” null-checks for the whole block

- Call to null-able variables are allowed inside the if branch

Null Safety
2 The Safe Call Operator

- ?. allows to combine a null-check and a method call in a single expression
- This reduces unnecessary *verbose* code blocks; e.g.:

```
nullableName?.toUpperCase()
```

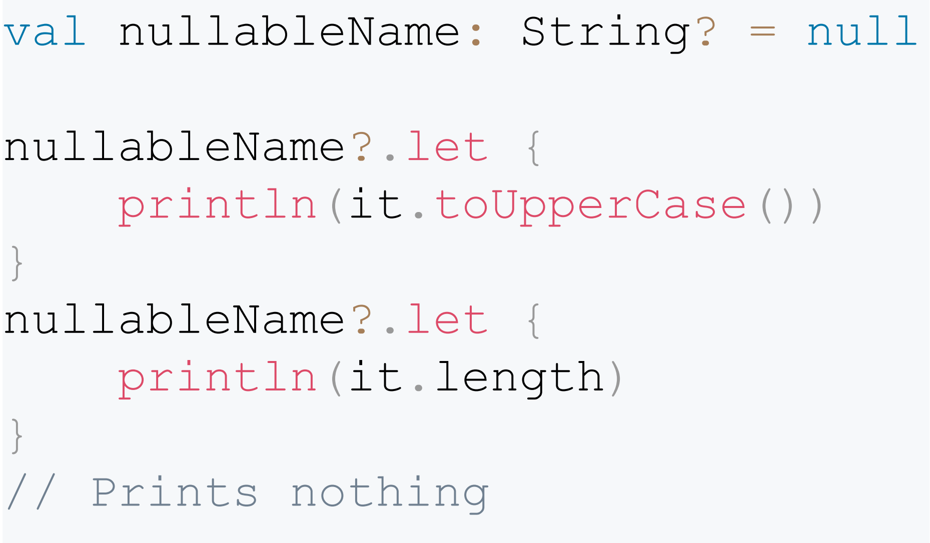
…is equivalent to:

```
if(nullableName != null)
    nullableName.toUpperCase()
else
    null
```

- Code doesn’t break anymore at runtime, but the value of your function call is still null.

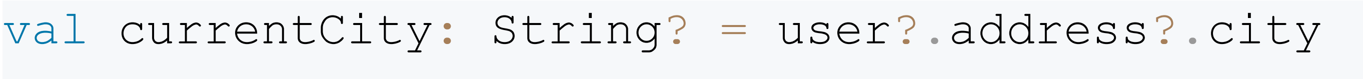
Null Safety
2 The Safe Call Operator II

- Introducing the `let` operator
- Performs an operation only if the variable is not null



```kotlin
val nullableName: String? = null

nullableName?.let {
    println(it.toUpperCase())
}
nullableName?.let {
    println(it.length)
}
// Prints nothing
```

- “it” is a reference to the object let has been called on
- Safe calls can be chained



```kotlin
val currentCity: String? = user?.address?.city
```

Null Safety
3 The Elvis operator

- Elvis Operator is used to provide a default value when the original variable is null

```
val name = if(nullName != null) nullName else "Guest"
```

- … is equivalent to:

```
val name = nullName ?: "Guest"
```

- Often used in combination with the safe call operator
- Can be combined with `throw` and `return` expressions

```
val len = nullableName?.length ?: -1
val currentCity = user?.address?.city ?: "Unknown"

val name = nullName ?: {
    throw IllegalArgumentException("Name can not be null")
}
```

Null Safety

4 Not null assertions

- Not Null assertions converts a nullable type to a non-null type
- Explicitly throwing a NullPointerException

```
val nullName: String? = null
nullName!!.toUpperCase()

// Results in NullPointerException
```

- Best practice to not use Not Null Assertions !!

Null Safety
Array Safety & Nullable Collections

- There are two possible occurrences of `null`-values in arrays

1. Collection of nullable types

```
val regularList: List<Int> = listOf(1, 2, null, 3)
// Compiler Error

val listWithNull: List<Int?> = listOf(1, 2, null, 3)
// Works

val notNullList: List<Int> = listWithNull.filterNotNull()
```

2. Nullable collections

```
listWithNull = null
// Compilation Error

var nullableList: List<Int>? = listOf(1, 2, 3)

var nullListOfNullTypes: List<Int?>? = listOf(1, null, 2)
```