The University Of Sheffield.

COM4510/6510
Software Development for Mobile Devices

Lecture 2: Intro to Kotlin

Dr Po Yang
The University of Sheffield
po.yang@sheffield.ac.uk

Lecture Overview

- **Part 1: What is Kotlin?**
  - Kotlin vs Java
  - Design goals of Kotlin
- **Part 2: Kotlin basics**
  - Variables and functions
  - Classes and properties
  - Defining and calling Functions
  - Programming with lambdas
- **Lab tutorial :**
  - Layout of your App

https://kotlinlang.org/docs/basic-types.html

What is Kotlin ?

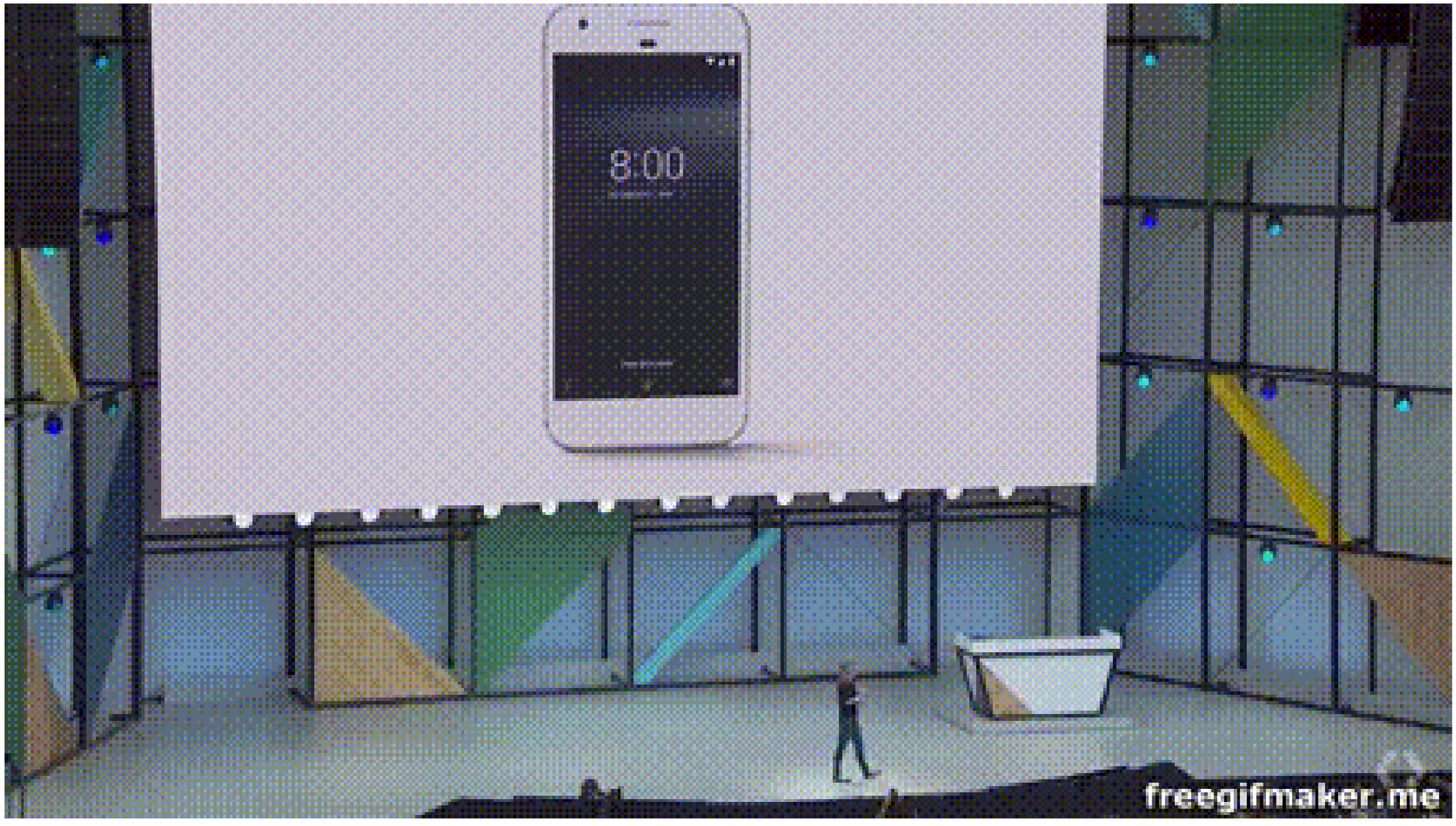
- A (fairly) new **statically-typed language from JetBrains** (creators of IntelliJ Idea, ReSharper, PyCharm and other IDEs and IDE extensions)
- Tries to fix many of **Java's shortcomings**
- Compiles to **JVM bytecode**, **JavaScript (!) and Kotlin native** (no VM)
- Created with focus on Java **interoperability** – Kotlin and Java classes can be used together in a project (but compilation is much faster than Scala's)

What is Kotlin ?

- Works seamlessly with **IntelliJ Idea**, just a bit less seamlessly with Eclipse
- The **Java interoperability** is real – even complex applications, relying on **annotation processing**
- It's gaining more and more traction and is being adopted increasingly often, especially in the recent year
- Has very good suport from **JetBrains**, an established company, who use Kotlin to develop their own products

Google and Android

- At Google I/O 2017, Google announces that Kotlin would receive **first-class suport for Android development**



Google and Android

- At Google I/O 2019,

*Today we're announcing another big step: Android development will become **increasingly Kotlin-first**. Many new Jetpack APIs and features will be offered first in Kotlin. If you're starting a new project, you should write it in Kotlin; code written in Kotlin often means much less code for you–less code to type, test, and maintain. And we're continuing to invest in tooling, docs, training and events to make Kotlin even easier to learn and use.*

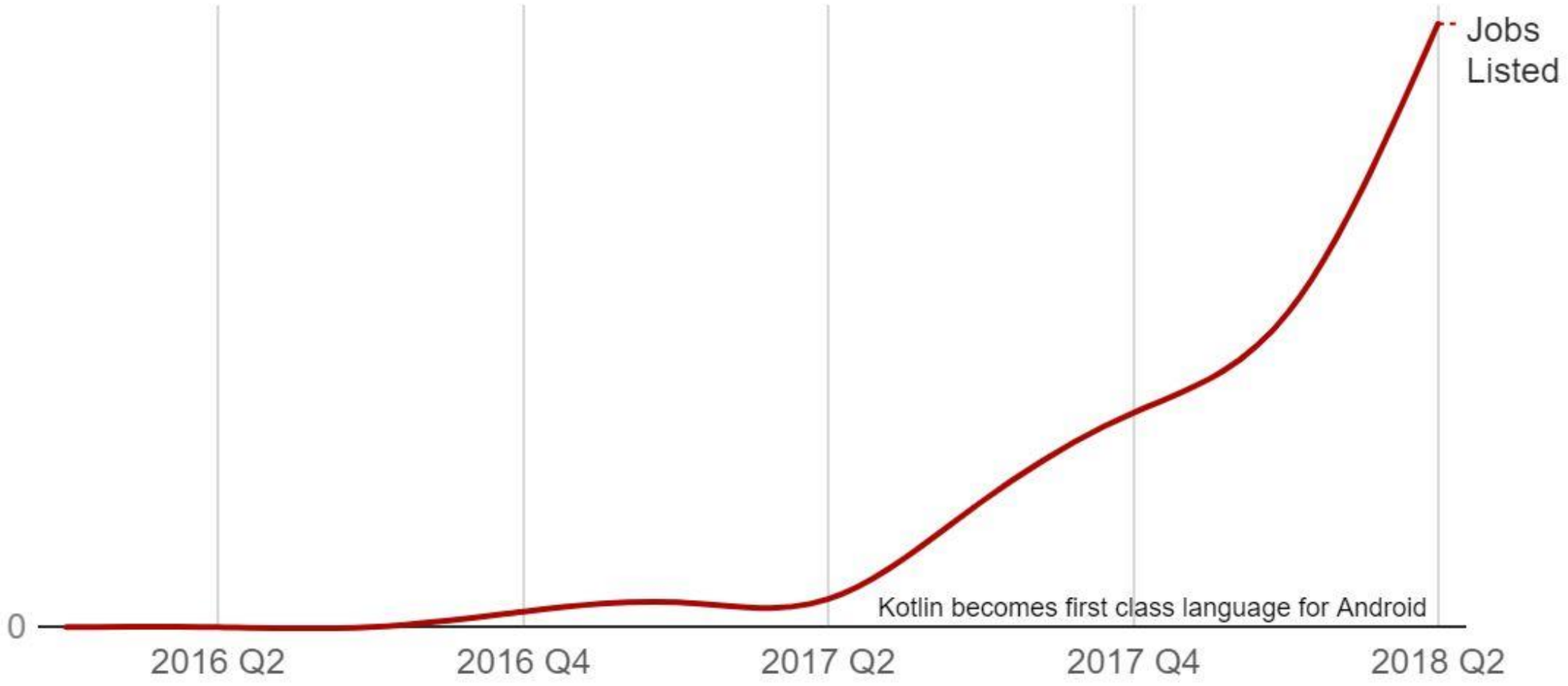
Posted by Chet Haase

The University Of Sheffield.

Kotlin developers

The Rise of Kotlin

Now that it's a first-class language for Android, Kotlin job postings have increased over 15x.



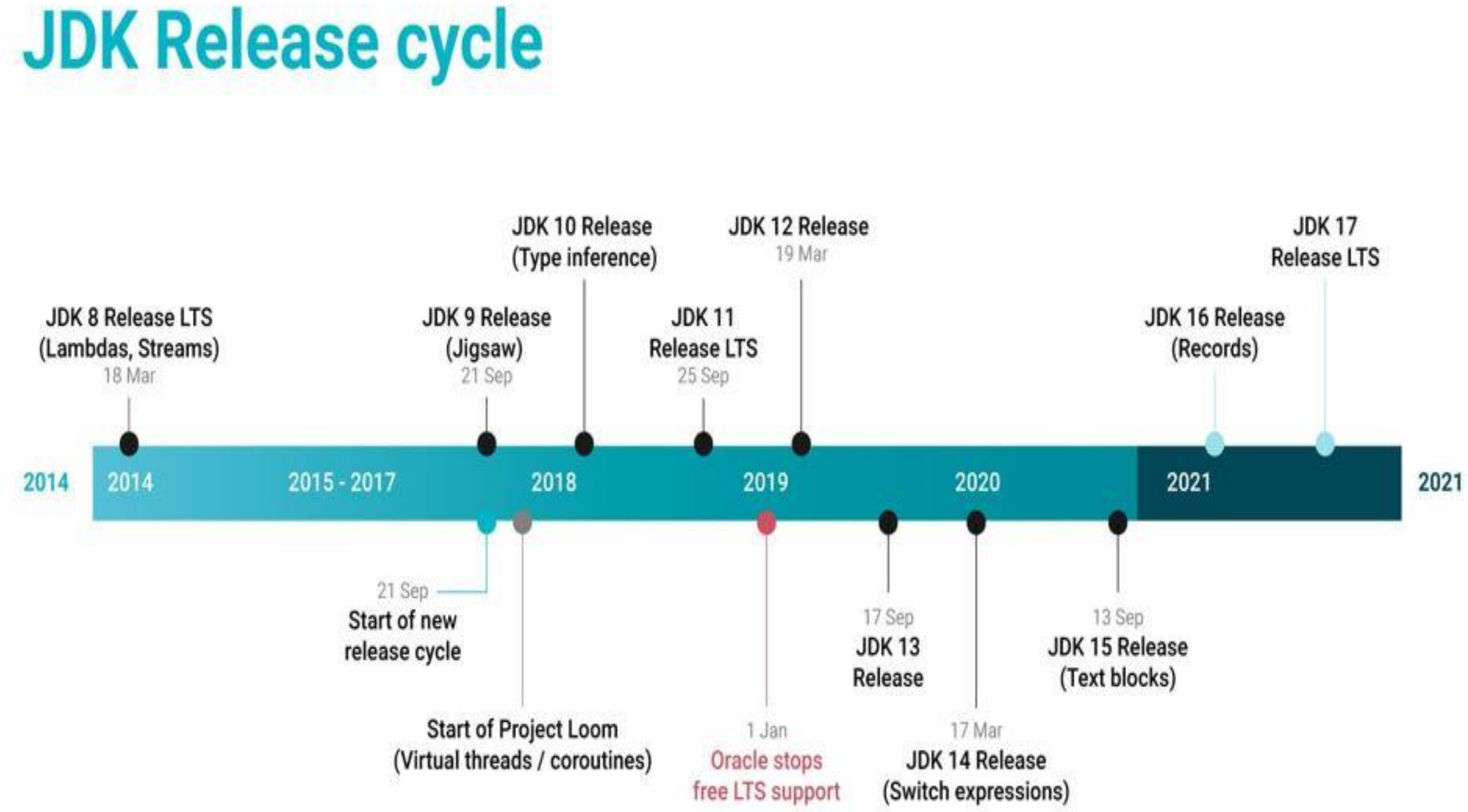
All data pulled from the Dice jobs database

Source: Dice

https://adtmag.com/articles/2019/06/25/kotlin-climb.aspx

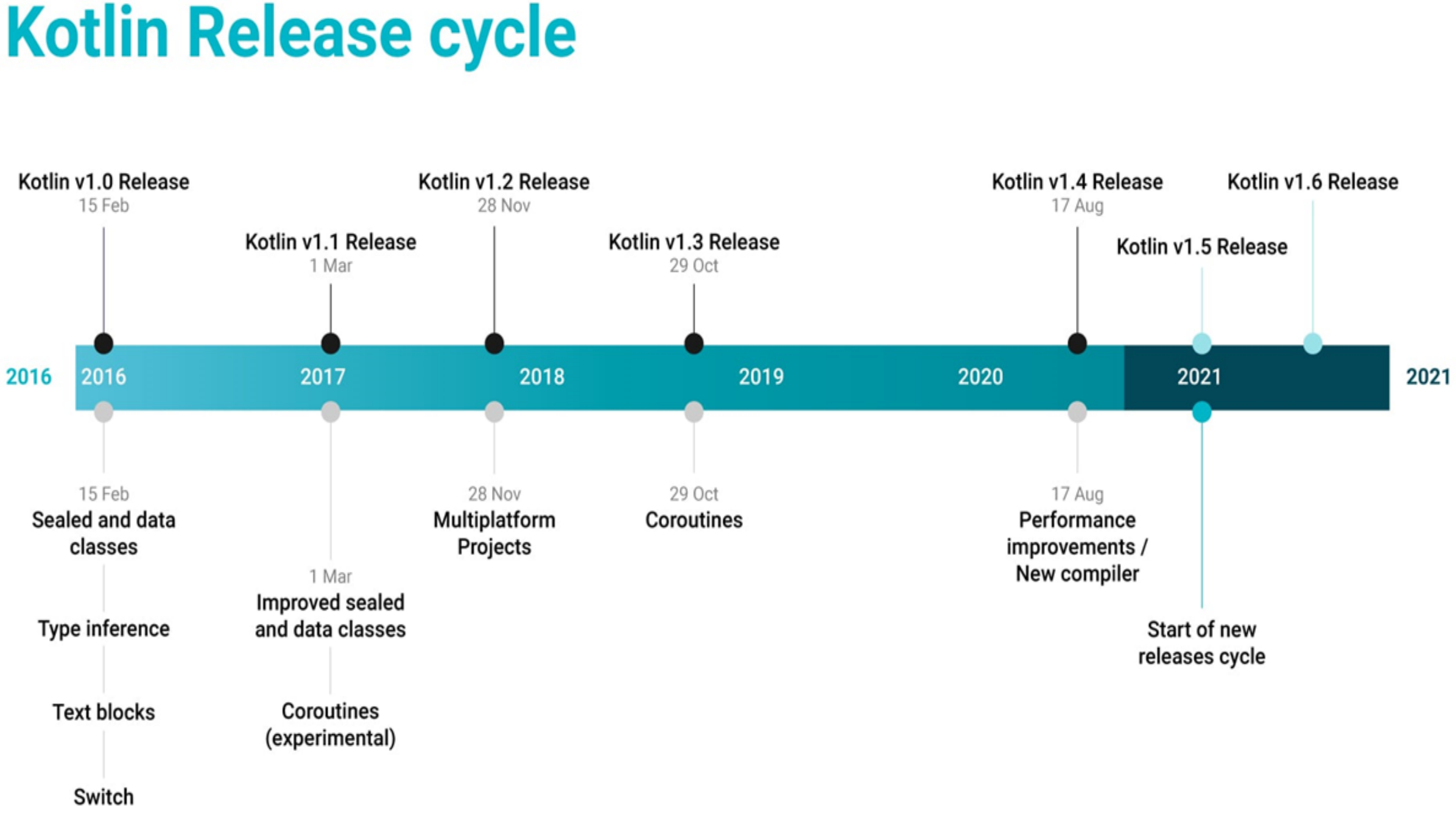
The University Of Sheffield.

Timeline: Kotlin v Java



https://bell-sw.com/announcements/2020/12/04/An-intelligent-bet-for-Kotlin-part-two/

Timeline: Kotlin v Java



https://bell-sw.com/announcements/2020/12/04/An-intelligent-bet-for-Kotlin-part-two/

Kotlin vs Java

Parameter	Java	Kotlin
Static memebers	Almost the same. Thinking of the solution is the most time-consuming part	Almost the same. Thinking of the solution is the most time-consuming part
Perfomance	Almost the same. Both compile to ByteCode	Almost the same. Both compile to ByteCode
Stability	Has stable versions with long-term maintenance	Almost the same. Both compile to ByteCode
Documentation	Good, easy to find	Good, a little bit harder to find
Popularity	Extremely popular worldwide	Not so popular worldwide
Community	Mostly Indian, very broad	Mostly Russian, comparatively little
Talent pool	Not in the top-list	In the list of the most popular technologies 2020 according to StackOverflow Dev Survey
Easiness to learn	Easy to learn	Can be tricky to learn if you are not a good abstract thinker

https://nix-united.com/blog/kotlin-vs-java-which-is-the-winner/

Why Kotlin is better for Android Development ?

- The ability to use **the latest version of the programming language** for Android development
  - Both compile their code in bytecode and run in JVM
  - But JVM in Android is consistent with **Java 7 and 8 only, not Java 16**.
- Kotlin is support for **true multiplatform programming**, can be complied in JavaScript and iOS native code.
- but Java is limited to a range of systems containing JVM, i.e., backend, desktop,.....
- Company can use **single-business logic** for desktop and web solutions.

Design goals of Kotlin

- **Pragmatic**
  - **Tool-friendly**
  - **Interoperable**
    - Use existing libraries for JVM (like Android API)
    - Compile to JVM or JavaScript

- modern
- pragmatic

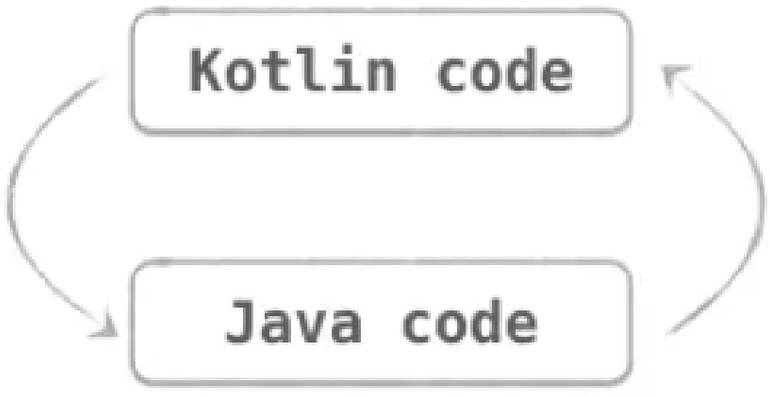
- **Modern**
  - **Concise**
    - Reduce the amount of code
  - **Safe**
    - No more **NullPointerExceptions**
    - Val name = data?.getStringExtra(Name)
      - Example: KotlinPersonalInfo->MainActivity

Design goals

- **Tool-friendly**
  - **From JETBRAINS**
  - **Has good tooling**
    - -completion
    - -navigation
    - -refactorings
    - -inspections
- …

Design goals

- **Interoperable**
  - Use existing libraries for JVM (like Android API)
  - Compile to JVM or JavaScript
- **Can be easily mixed with Java Code**
- **You can have Java & Kotlin code in one project**



- **You can gradually add Kotlin to your existing app**

Design goals

- **Modern**
  - **Concise & readable**
    - Reduce the amount of code

```
public class Person {
  private final String name;
  private final int age;

  public Person(String name, int age) {
    this.n
    this.a
  }

  public S
    return name;
  }

  public int getAge() {
    return age;
  }
}
```

Enter action or option name:
convert Java to Kotlin
Convert Java File to Kotlin File (⌥⇧⌘K) Code
Press ^↑ or ^↓ to navigate through the history

```
data class Person(val name: String, val age: Int)
```

- equals
- hashCode
- toString

```
public void updateWeather(int degrees) {
    String description;
    Colour colour;
    if (degrees < 5) {
        description = "cold";
        colour = BLUE;
    } else {
        description = "hot";
        colour = RED;
    }
}
```

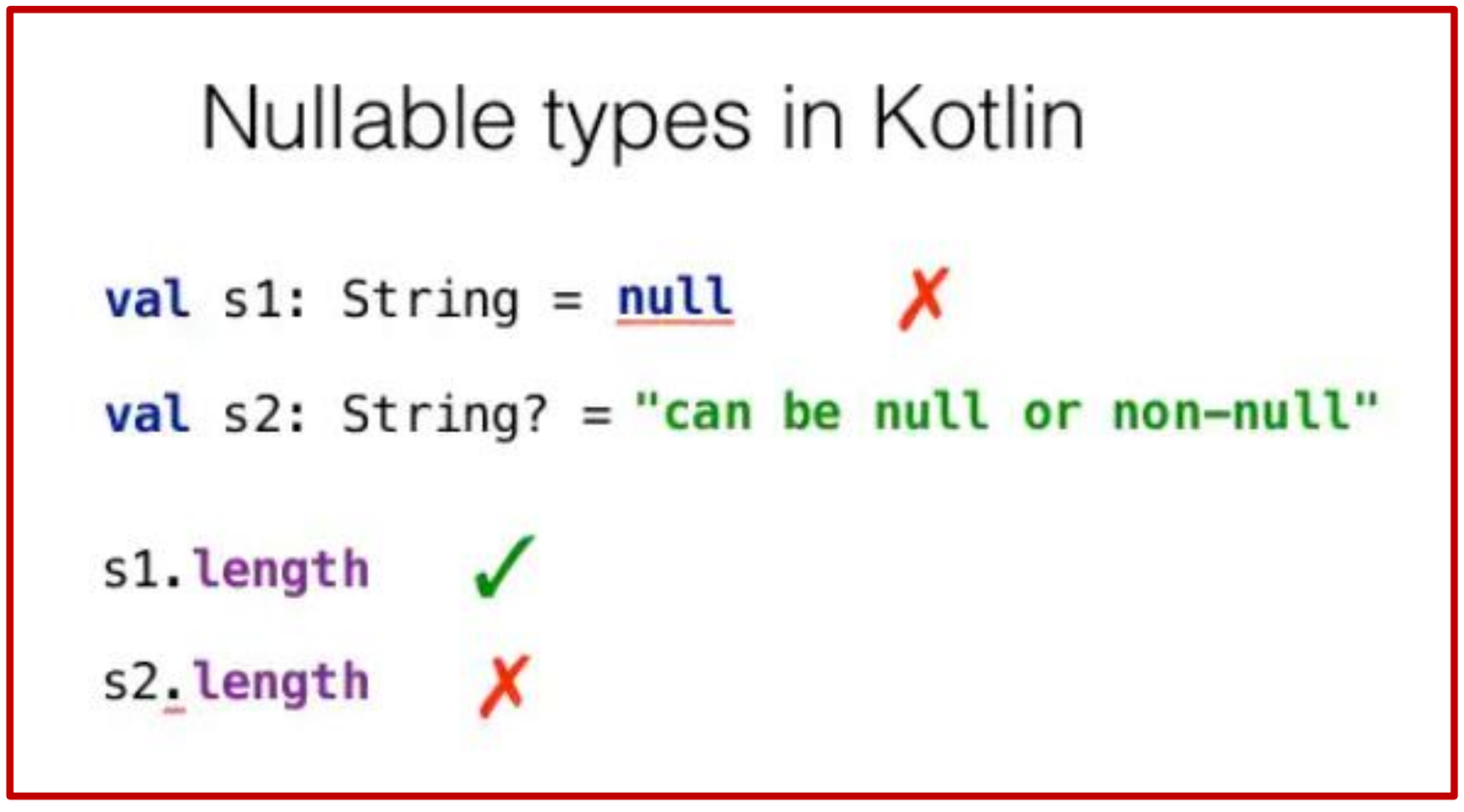
Convert Code From Java
Clipboard content copied from Java file. Do you want to convert it to Kotlin code?
Don't show this dialog next time
No Yes

```
String description;
Colour colour;
if (degrees < 5) {
    description = "cold";
    colour = BLUE;
} else if (degrees < 23) {
    description = "mild";
    colour = ORANGE;
} else {
    description = "hot";
    colour = RED;
}
```

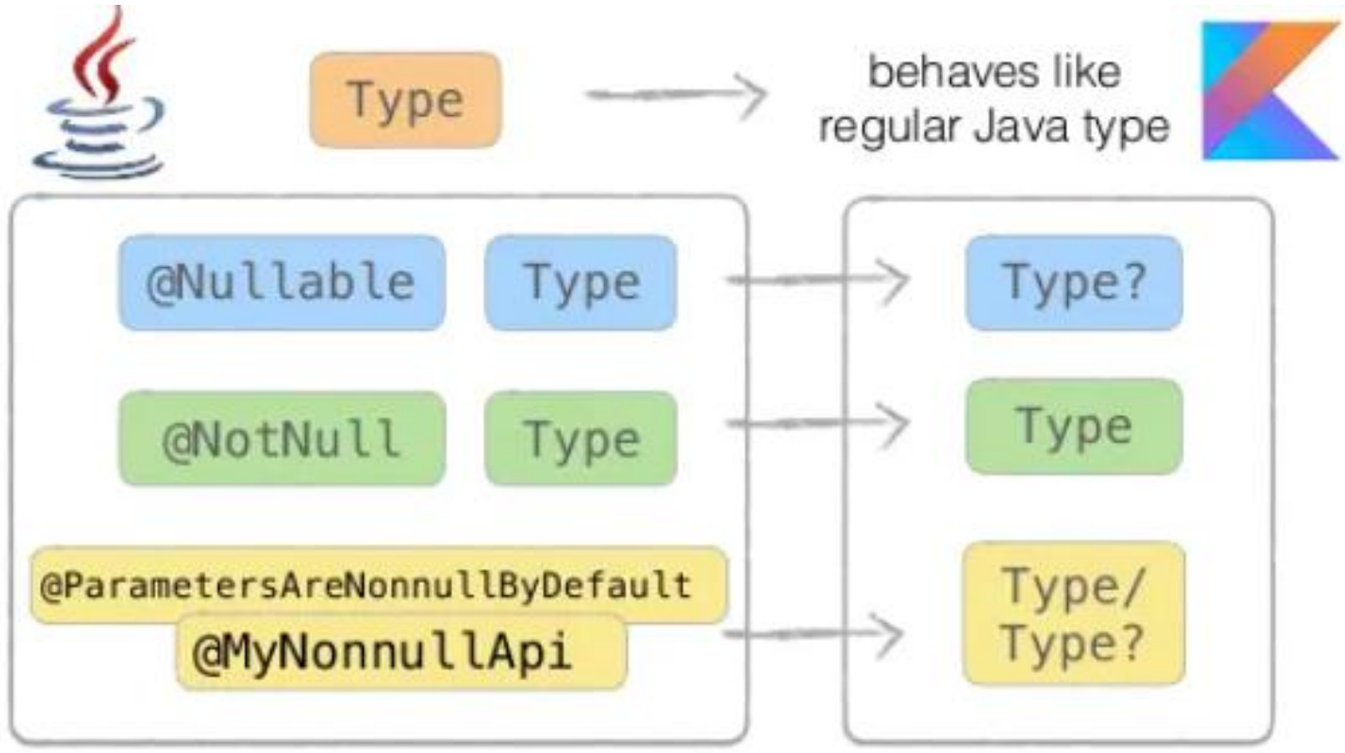
```
val (description, colour) = when {
    degrees < 5 -> Pair("cold", BLUE)
    degrees < 23 -> Pair("mild", ORANGE)
    else -> Pair("hot" to RED)
}
```

Design goals

- **Safe**
  - Modern approach to make **NPE (NullPointerException)**
  - Compile-time error, not run-time error



Using annotated Java Types from Kotlin



- Expressive
- More code reuse!
- You can avoid any repetition
- You can make the code look nicer
- You can create API look like DSL (Domain-Specific Language)

Lecture Overview

- **Part 1: What is Kotlin?**
  - Kotlin vs Java
  - Design goals of Kotlin
- **Part 2: Kotlin basics**
  - Variables and data types
  - Defining and calling Functions
  - Type-safe builders and null safety
  - Classes
- **Lab tutorial :**
  - Layout of your App

https://kotlinlang.org/docs/basic-types.html

Packages and imports

- **A source file may start with a package declaration**

```
package org.example

fun printMessage() { /*...*/ }
class Message { /*...*/ }
```

- **Default imports**
- **Imports**

```
import org.example.Message

import org.example.*

import org.example.Message // Message is accessible
import org.test.Message as testMessage // testMessage
```

- kotlin.* ↗
- kotlin.annotation.* ↗
- kotlin.collections.* ↗
- kotlin.comparisons.* ↗
- kotlin.io.* ↗
- kotlin.ranges.* ↗
- kotlin.sequences.* ↗
- kotlin.text.* ↗

https://kotlinlang.org/docs/basic-types.html

Variables

- **Support two kinds of variables**
  - **Immutable**: the value cannot be changed once the value is assigned, similar to the final variable in Java by **val**

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

    val welcome = "Welcome to Kotlin Tutorial" // we can not reassign
    print(welcome)

}
```

- **Mutable**: the value can be changed later in the programmes, mutable variable donated by **var**

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

    var welcome = "Welcome to Kotlin Tutorial"
    welcome = "Variables and Data Type in Kotlin" // reassign the value
    print(welcome)

}
```

Variables

- **Type inference:** Not specified the type of the variable, the compiler can understand the type of the variable by looking at the value

```
fun main(args: Array<String>) {
    // case 1
    var myNumber = 10 // type inferred as `Int`
    // case 2
    var myDecimal = 1.0;  // type inferred as `Float`
    // case 3
    var myString = "Kotlin Tutorial"     // type inferred as `String`


    print(myString)

}
```

- **Lazy Initialisation:** Do not want to initilise variable the time of the declaration, you must have to specify the type of variable during declaration

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

    var helloWorld: String  // It is mandatory to specify the type in this case

    helloWorld = "Hello World "

    print(helloWorld)

}
```

Data types

- **Data types in Kotlin**
  - Numbers, Booleans, Characters, Strings, Arrays

	Size	Default Value	Example
Boolean	1 bit	All variable must be initialised	true, false
Byte	8 bit		-127 to 128
Char	16 bit		'a', '\n', '2', '\101'
Short	16 bit		none
Int	32 bit		-2, -1, 0, 1, 2
Long	64 bit		-2L, -1L, 0L, 1L, 2L
Float	32 bit		3.4f, 3.7F
Double	64 bit		3.4d, 3.7D

String templates

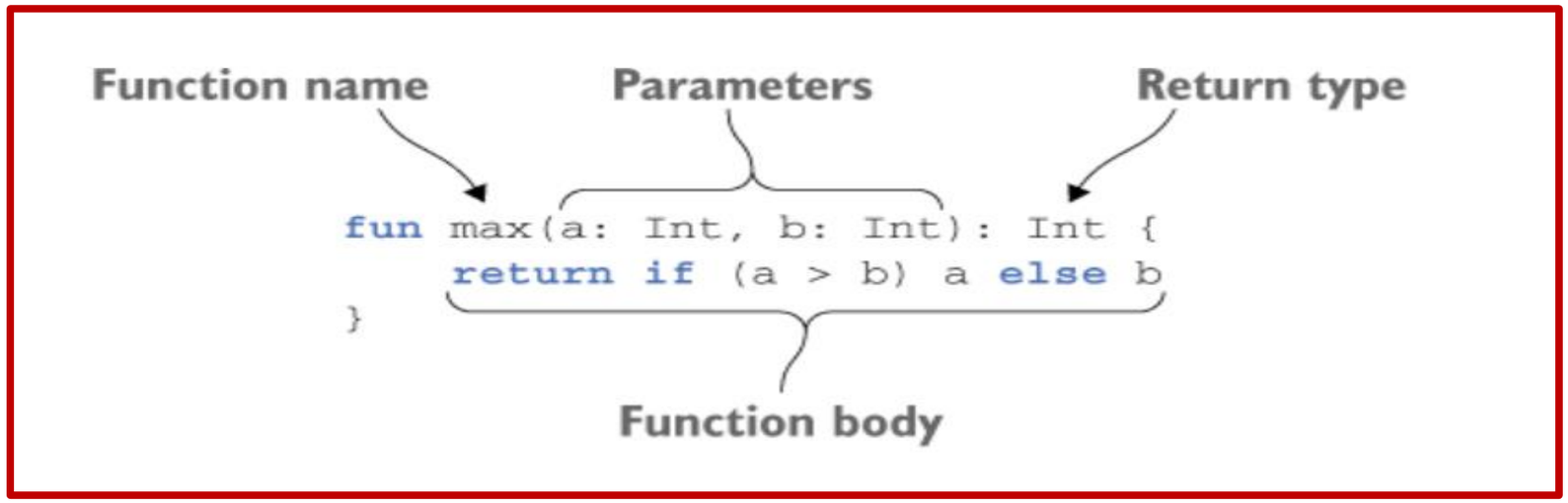
- **Easier string formatting: String templates**

```
fun main() {
    val myString = "KOTLIN"
    println(myString[0])
    println(myString[1])
    println(myString.isEmpty())
    println(myString.length)
    println(myString.substring(2, 4))
    myString.
}
```

```
[](index: Int)                    Char
plus(other: Any?)                 String
length                            Int
get(index: Int)                   Char
chars()                           IntStream!
codePoints()                      IntStream!
compareTo(other: String)          Int
equals(other: Any?)               Boolean
hashCode()                        Int
subSequence(startInd…             CharSequence
toString()                        String
```

Functions

- **Function declaration**



- **Statements and expressions**
  - In Kotlin, ***if*** **is an expression, not a statement**
  - An **expression has a value**, which can be used as part of another expression,
  - whereas **a statement is always a top-level element** in its enclosing block and doesn't have its own value.
  - Most control structures, except for **the loops** (**for, do, and do/while**) **are expressions**.

Functions

- **Expression bodies**

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

  - If a function is written with its body in curly braces, we say that this function has a **block body**.
  - If it returns an expression directly, it has **expression body**.

- **Function scope**
  - **Local functions**: Kotlin supports local functions, which are functions inside other functions
  - A local function can access **local variables** of outer functions (the closure).
  - In the case right, **visited** can be a local variable

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

Functions

- **Generic functions**
  - Functions can be **generic parameters**, which are specified using **angle brackets** before the function name

```
fun <T> singletonList(item: T): List<T> { /*...*/ }
```

  - To **call a generic function**, specify the time arguments at the all site after the name of the function

```
val l = singletonList<Int>(1)
```

  - A example:

```
fun main(args: Array<String>) {
    val age = 23
    val name = "runoob"
    val bool = true

    doPrintln(age)    // Int
    doPrintln(name)   // String
    doPrintln(bool)   // Bool
}
```

```
fun <T> doPrintln(content: T) {

    when (content) {
        is Int -> println("Integer $content")
        is String -> println("String: ${content.toUpperCase()}")
        else -> println("T is neither an integer nor an string ")
    }
}
```

Functions

- **Tail recursive functions**
  - For some algorithms using **loops**, you can use a **recursive function** instead without the risk of stack overflow.
  - A function is marked with the **tailrec** modifier and meets the required formal conditions, the compiler optimises out the recursion, leaving behind a fast and efficient loop based version instead.

```
val eps = 1E-10 // "good enough", could be 10^-15

tailrec fun findFixPoint(x: Double = 1.0): Double =
    if (Math.abs(x - Math.cos(x)) < eps) x else findFixPoint(Math.cos(x))
```

- It calls **Math.cos** repeatedly starting

At 1.0 until the result **no longer changes**

Yielding a result of 0.739…. for the

Specified eps precision.

```
val eps = 1E-10 // "good enough", could be 10^-15

private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (Math.abs(x - y) < eps) return x
        x = Math.cos(x)
    }
}
```

Lambda expressions

Lambda expressions are functions (function literals) that are not declared but are passed immediately as an expression

- **Lambda expression syntax**

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

  - A lambda expression is **always surrounded by curly braces**
  - Parameter declarations in the full syntactic form go inside curly braces and have optional type annotations.
  - **The body goes after the ->**
  - If the inferred return type of the lambda is not Unit, the last expression inside the lambda body is treated as the return value.
  - If you leave all the optional annotations out,

```
val sum = { x: Int, y: Int -> x + y }
```

Lambda expressions

- **Passing trailing lambdas**
  - If the last parameter of a function is a function, then ***a lambda expression*** passed at the corresponding argument can be placed outside the parentheses

```
val product = items.fold(1) { acc, e -> acc * e }
```

  - Name as ***trailing lambda***
  - If the lambda is the only argument in that call, the parentheses can be omitted entirely.
- **It is very common for a lambda expression to have only one parameter**

Lambda expressions

- **Returning a value from a lambda expression**
  - You can explicitly return a value from the lambda using the qualified return syntax. Otherwise, the value of the last expression is implicitly returned.
  - Two right snippets are equivalent

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

Anonymous functions

- **An anonymous function looks very much like a regular function declaration**, except its name is omitted. Its body can be either an expression or a block:

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

- The parameters and the return type are specified in the same way as for regular functions, except the parameter types can be omitted if they can inferred from the context

```
ints.filter(fun(item) = item > 0)
```

When passing anonymous functions as parameters, place them inside the parentheses. The shorthand syntax that allows you to leave the function outside the parentheses works only for lambda expressions.

Type-safe builders

- **Type-safe builders** allow creating **Kotlin-based domain-specific language (DSLs)** suitable for building complex hierarchical data structures in a semi-declarative way.
- **Examples:**
  - Generating markup with Kotlin code, such as HTML or XML
  - Programmatically laying out UI components: Anko
  - Configuring routes for a web server: Ktor

Null safety

- **Nullable types and non-null types**
- Kotlin's type system is aimed at eliminating the danger of null references, know as The Billion Dollar Mistake
- The only possible causes of an **NPE (NullPointerException)** in Kotlin
- An explicit call to throw **NullPointerException()**
- Usage of the **!!** Operator
- Data inconsistency with regard to initilisation

Equality

- **Structural equality**
  - Checked by the == operation and its **negated counterpart** !=. By convention, an expression like a == b is translated to:

```
a?.equals(b) ?: (b === null)
```

- **Referential equality**
  - Is checked by == operation and its negated counterpart !==, a = b evaluates to true if and only if a and b point to the same object.
  - For values represented by primitive types at runtime, the **=== equality check** is equivalent to the == check.

Classes and instances

- **Define a class**

```
class Shape
```

  - Properties of a class can be listed in its declaration or body

```
class Rectangle(var height: Double, var length: Double)
    var perimeter = (height + length) * 2
}
```

  - Default constructor with parameters listed in the class declaration is available automatically

```
val rectangle = Rectangle(5.0, 2.0)
println("The perimeter is ${rectangle.perimeter}")
```

- Inheritance between classes is declared by a colon (:), to make a class inheritable, mark it as Open

```
open class Shape

class Rectangle(var height: Double, var length: Double)
    var perimeter = (height + length) * 2
}
```

Conditional Expressions

- **If-Else expression: it returns something it's supposed to:**

```
val highestMarks = if (marksInMaths > marksInScience) marksInMaths else marksInScience
```

- If you want to execute some code along with returning value using if-else expression,

```
val highestMarksAgain =
        if (marksInMaths > marksInScience) {
            println("The last line is returned and you can put any code here")
            marksInMaths
        } else {
            println("The last line is returned and you can put any code here")
            marksInScience
        }
```

Conditional Expressions

- **For loop**

```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

- **While loop**

```
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}
```

Expressions

- **When expression:**
  - The switch statement of Java is replaced by when expression in Kotlin. It is more concise and elegant way to match the case and perform an appropriate action

```
val totalMarks = marksInMaths + marksInScience

when (totalMarks) {
    100 -> println("Good score but I'm not impressed")
    120 -> println("That's a great score")
    140 -> { // For multiline code, use braces {}
        println("Well, you are really intelligent")
        println("You have a bright future")
    }

    else -> println("You're out of the world :)") //else is optional
}
```

Ranges

- **Check if a number is with a range using In operator**

```
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```

- **Check if a number of out of range using !in**
- **Iterate over a range**

```
for (x in 1..5) {
    print(x)
}
```

- **Over a progression**

```
for (x in 1..10 step 2) {
    print(x)
}
println()
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

Collections

- Iterate over a collection

```
for (item in items) {
    println(item)
}
```

- Check if a collection contains an object using In operator

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

- Using Lambda expression to filleter and map collections

```
val fruits = listOf("banana", "avocado", "apple",
fruits
    .filter { it.startsWith("a") }
    .sortedBy { it }
    .map { it.uppercase() }
    .forEach { println(it) }
```

Nullable values and null check

- **A reference must be explicitly marked as nullable** when null value is possible, nullable type names have ? at the end.
- Return null if str does not hold an integer:

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

- Use a function returning nullable value

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullab
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
```

Type checks and casts

- **is and !is operators**
  - To perform a runtime check that ide to a given type

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // same as !(obj is String)
    print("Not a String")
} else {
    print(obj.length)
}
```

- **Smart casts**
  - In most cases, you do not need to use explicit cast operators, is checks and explicit casts for immutable values and inserts casts automatically
  - The compiler is smart enough to know that a case is safe or not
  - Smart casts work for when expressions and while loops

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to S
    }
}
```

https://kotlinlang.org/docs/basic-types.html

```
when (x) {
    is Int -> print(x + 1)
    is String -> print(x.length + 1)
    is IntArray -> print(x.sum())
}
```

The University Of Sheffield.

Type checks and casts

- **Unsafe cast operator**
  - Usually, the cast operator throws an exception if the cast is not possible. The unsafe cast is done by the infix operator as
- **Safe (nullable) cast operator**
  - To avoid exceptions, use the safe cast operator as?, when return null on failure.
- **Type erasure and generic type checks**
  - To avoid exceptions, use the safe cast operator as?, when return null on failure.
- **Unchecked casts**
  - To avoid exceptions, use the safe cast operator as?, when return null on failure.

https://kotlinlang.org/docs/basic-types.html

Exceptions

- All exception classes in Kotlin **inherit the Throwable class**. Every exception has **a message, a stack trace, and an optional cause**.
- To throw an exception object, use the **throw** expression

```
throw Exception("Hi There!")
```

- To catch an exception, use the **try…catch** expression

```
try {
    // some code
} catch (e: SomeException) {
    // handler
} finally {
    // optional finally block
}
```

- There may be zero or more **catch** blocks, and then **finally** block may be omitted. But, at least one **catch** or **finally** block is required

Exceptions

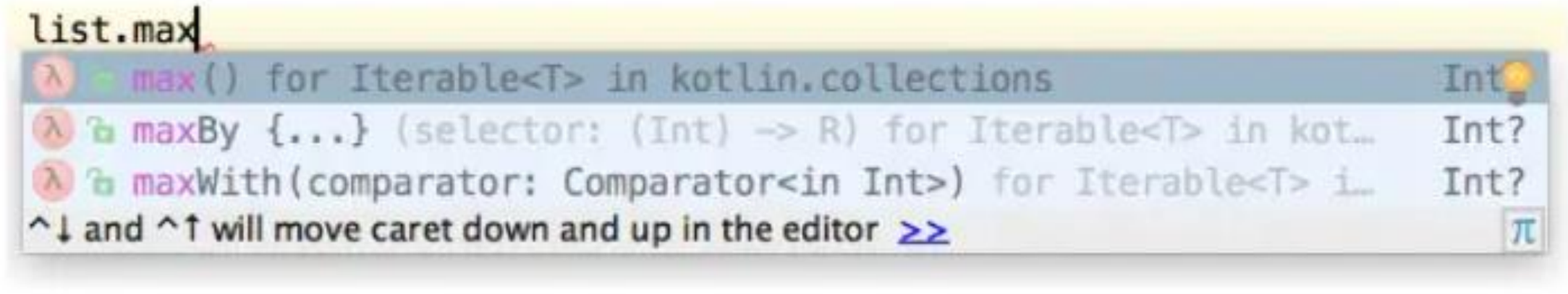
- Try is an expression, which means it can have a return value:

```
val a: Int? = try { input.toInt() } catch (e: NumberFormatException) { null }
```

- The returned value of a try expression is either **the last expression in the try block** or **the last expression in the catch block.**
- The contents of the finally block do not affect the result of the expression.

Kotlin Library:

- Extensions on collections



- filter
- map
- reduce
- count
- find
- any
- flatMap
- groupBy
- …

Kotlin, a little syntax

- **Semicolons are optional**
  - Adding a new-line is generally enough
- **Variables and constants**
  - var name = “Anders” // variable
    - Var means “variable”
  - val name2 = “Anders” // read-only (constant)
    - Val means “value”
  - Types are not specified explicitly by the programmer
    - Types are inferred by the compiler
- **Examples**
  - KotlinCollectWords, Calculator, Personal Information

The University Of Sheffield.

What does Kotlin feature?

- **Null safety** (drastically limiting the number of NPEs);
- If, try-catch, when **expressions**;
- **Extension functions**;
- **Var and val** keywords;
- **Inline functions**;
- **Named function parameters**;
- **Multi-value return functions**;
- Semi-colons are **optional**

More things…

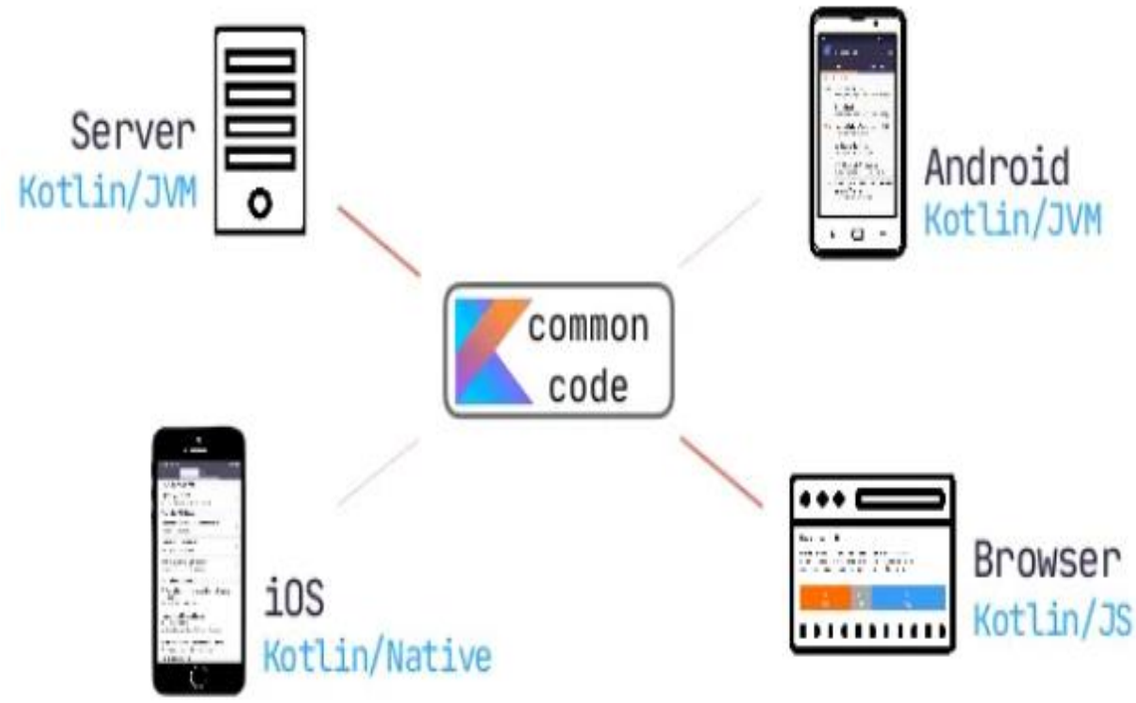
- **Functional programming constructs** (like higher order functions)
- **Smart casting**
- Async/await
- **Data classes**
- Properties (C#- like)
- **No checked exceptions**
- **Operator overloading**
- … this is just scratching the surface

Kotlin support in Android Studio

- **Kotlin + Android Studio** both made by **JetBrains**
- Great support for Kotlin in Android Studio
- **New project**
  - Choose if you want to add support for Kotlin
- **New Activity**
  - Choose if you want Java or Kotlin (scroll down)
- **Convert existing Jave code to Kotlin**
  - Code->Convert Jave File to Kotlin
- **Add Kotlin to an existing project** (without support for Kotlin)

Multi-platform projects

- **Sharing business logic**
- **Keeping UI platform-dependent**
  - The shared part might vary
- **Common code**
  - You define expect declarations in the common code and use them
  - You provide **different actual implementations** for different platforms



Summary

- **Part 1: What is Kotlin?**
  - Kotlin vs Java
  - Design goals of Kotlin
- **Part 2: Kotlin basics**
  - Variables and functions
  - Classes and properties
  - Defining and calling Functions
  - Programming with lambdas
- **Lab tutorial :**
  - Layout of your App

https://kotlinlang.org/docs/basic-types.html