Ganpat University
॥ विद्यया समाजोत्कर्षः ॥
U.V. Patel College of Engineering

Mobile Application Development

Kotlin

U. V. Patel College of Engineering, Ganpat University

Program entry point

An entry point of a Kotlin application is the `main` function:

```
fun main() {
    println("Hello world!")
}
```

Another form of `main` accepts a variable number of `String` arguments:

```
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

Print to the standard output

`print` prints its argument to the standard output:

```
print("Hello ")
print("world!")
```

Open in Playground → Target: JVM Running on v.2.0.10

`println` prints its arguments and adds a line break, so that the next thing you print appears on the next line:

```
println("Hello world!")
println(42)
```

Open in Playground → Target: JVM Running on v.2.0.10

Read from the standard input

The `readln()` function reads from the standard input. This function reads the entire line the user enters as a string.

You can use the `println()`, `readln()`, and `print()` functions together to print messages requesting and showing user input:

```
// Prints a message to request input
println("Enter any word: ")

// Reads and stores the user input. For example: Happiness
val yourWord = readln()

// Prints a message with the input
print("You entered the word: ")
print(yourWord)
// You entered the word: Happiness
```

Variables

In Kotlin, you declare a variable starting with a keyword, `val` or `var`, followed by the name of the variable.

Use the `val` keyword to declare variables that are assigned a value only once. These are immutable, read-only local variables that can't be reassigned a different value after initialization:

```
// Declares the variable x and initializes it with the value of 5
val x: Int = 5
```

Use the `var` keyword to declare variables that can be reassigned. These are mutable variables, and you can change their values after initialization:

```
// Declares the variable x and initializes it with the value of 5
var x: Int = 5
// Reassigns a new value of 6 to the variable x
x += 1
// 6
```

Type Conversion

To work with data types other than strings, you can convert the input using conversion functions like `.toInt()`, `.toLong()`, `.toDouble()`, `.toFloat()`, or `.toBoolean()`. It is possible to read multiple inputs of different data types and store each input in a variable:

```
// Converts the input from a string to an integer value. For example: 12
val myNumber = readln().toInt()
println(myNumber)
// 12

// Converts the input from a string to a double value. For example: 345
val myDouble = readln().toDouble()
println(myDouble)
// 345.0

// Converts the input from a string to a boolean value. For example: true
val myBoolean = readln().toBoolean()
println(myBoolean)
// true
```

Functions

A function with two `Int` parameters and `Int` return type:

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

A function body can be an expression. Its return type is inferred:

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

A function that returns no meaningful value:

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

`Unit` return type can be omitted:

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

Function usage

Kotlin functions are declared using the `fun` keyword:

```
fun double(x: Int): Int {
    return 2 * x
}
```

Functions are called using the standard approach:

```
val result = double(2)
```

Function usage

Parameters

Function parameters are defined using Pascal notation - **name: type**. Parameters are separated using commas, and each parameter must be explicitly typed:

```
fun powerOf(number: Int, exponent: Int): Int { /*...*/ }
```

You can use a trailing comma when you declare function parameters:

```
fun powerOf(
    number: Int,
    exponent: Int, // trailing comma
) { /*...*/ }
```

Function usage

Default arguments

Function parameters can have default values, which are used when you skip the corresponding argument. This reduces the number of overloads:

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

A default value is set by appending `=` to the type.

String Templates

Ganpat University
॥ विद्यया समाजोत्कर्षः ॥
U.V. Patel College of Engineering

```
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"
```

```
val s = "abc"
println("$s.length is ${s.length}")
// abc.length is 3
```

String Templates


```
// Formats an integer, adding leading zeroes to reach a length of seven char
val integerNumber = String.format("%07d", 31416)
println(integerNumber)
// 0031416

// Formats a floating-point number to display with a + sign and four decimal
val floatNumber = String.format("%+.4f", 3.141592)
println(floatNumber)
// +3.1416

// Formats two strings to uppercase, each taking one placeholder
val helloString = String.format("%S %S", "hello", "world")
println(helloString)
// HELLO WORLD

// Formats a negative number to be enclosed in parentheses, then repeats the
val negativeNumberInParentheses = String.format("%(d means %1\$d", -31416)
println(negativeNumberInParentheses)
//(31416) means -31416
```

Conditional expressions

In Kotlin, `if` is an expression: it returns a value. Therefore, there is no ternary operator (`condition ? then : else`) because ordinary `if` works fine in this role.

```
var max = a
if (a < b) max = b

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

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

// You can also use `else if` in expressions:
val maxLimit = 1
val maxOrLimit = if (maxLimit > a) maxLimit else if (a > b) a else b
```

Conditional expressions


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

In Kotlin, `if` can also be used as an expression:

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

When expression

`when` defines a conditional expression with multiple branches. It is similar to the `switch` statement in C-like languages. Its simple form looks like this.

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

`when` matches its argument against all branches sequentially until some branch condition is satisfied.

When expression

`when` can be used either as an expression or as a statement. If it is used as an expression, the value of the first matching branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored. Just like with `if`, each branch can be a block, and its value is the value of the last expression in the block.

The `else` branch is evaluated if none of the other branch conditions are satisfied.

If `when` is used as an **expression**, the `else` branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions, for example, with `enum` class entries and `sealed` class subtypes).

When expression


```
enum class Bit {
    ZERO, ONE
}

val numericValue = when (getRandomBit()) {
    Bit.ZERO -> 0
    Bit.ONE -> 1
    // 'else' is not required because all cases are covered
}
```

In `when` **statements**, the `else` branch is mandatory in the following conditions:

- `when` has a subject of a `Boolean`, `enum`, or `sealed` type, or their nullable counterparts.
- branches of `when` don't cover all possible cases for this subject.

When expression


```
enum class Color {
    RED, GREEN, BLUE
}

when (getColor()) {
    Color.RED -> println("red")
    Color.GREEN -> println("green")
    Color.BLUE -> println("blue")
    // 'else' is not required because all cases are covered
}

when (getColor()) {
    Color.RED -> println("red") // no branches for GREEN and BLUE
    else -> println("not red") // 'else' is required
}
```

When expression

To define a common behavior for multiple cases, combine their conditions in a single line with a comma:

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

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

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

You can also check a value for being `in` or `!in` a range or a collection:

For loops

The `for` loop iterates through anything that provides an iterator. This is equivalent to the `foreach` loop in languages like C#. The syntax of `for` is the following:

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

The body of `for` can be a block.

```
for (item: Int in ints) {
    // ...
}
```

For loops

```
for (i in 1..3) {
    println(i)
}
for (i in 6 downTo 0 step 2) {
    println(i)
}
```

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

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

While loops

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

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

Break and continue in loops

Kotlin supports traditional `break` and `continue` operators in loops.

Arrays


```
var riversArray = arrayOf("Nile", "Amazon", "Yangtze")

// Using the += assignment operation creates a new riversArray,
// copies over the original elements and adds "Mississippi"
riversArray += "Mississippi"
println(riversArray.joinToString())
// Nile, Amazon, Yangtze, Mississippi
```

Create arrays

To create arrays in Kotlin, you can use:

- functions, such as `arrayOf()` ↗, `arrayOfNulls()` ↗ or `emptyArray()` ↗.
- the `Array` constructor.

Arrays


```
// Creates an array with values [1, 2, 3]
val simpleArray = arrayOf(1, 2, 3)
println(simpleArray.joinToString())
// 1, 2, 3
```

```
var exampleArray = emptyArray<String>()
```

```
// Creates an array with values [null, null, null]
val nullArray: Array<Int?> = arrayOfNulls(3)
println(nullArray.joinToString())
// null, null, null
```

```
// Creates an Array<Int> that initializes with zeros [0, 0, 0]
val initArray = Array<Int>(3) { 0 }
println(initArray.joinToString())
// 0, 0, 0

// Creates an Array<String> with values ["0", "1", "4", "9", "16"]
val asc = Array(5) { i -> (i * i).toString() }
asc.forEach { print(it) }
// 014916
```

Arrays


```
// Creates a two-dimensional array
val twoDArray = Array(2) { Array<Int>(2) { 0 } }
println(twoDArray.contentDeepToString())
// [[0, 0], [0, 0]]

// Creates a three-dimensional array
val threeDArray = Array(3) { Array(3) { Array<Int>(3) { 0 } } }
println(threeDArray.contentDeepToString())
// [[[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[
```

```
val simpleArray = arrayOf(1, 2, 3)
val twoDArray = Array(2) { Array<Int>(2) { 0 } }

// Accesses the element and modifies it
simpleArray[0] = 10
twoDArray[0][0] = 2

// Prints the modified element
println(simpleArray[0].toString()) // 10
println(twoDArray[0][0].toString()) // 2
```

Compare Arrays


```
val simpleArray = arrayOf(1, 2, 3)
val anotherArray = arrayOf(1, 2, 3)

// Compares contents of arrays
println(simpleArray.contentEquals(anotherArray))
// true

// Using infix notation, compares contents of arrays after an element
// is changed
simpleArray[0] = 10
println(simpleArray contentEquals anotherArray)
// false
```

Arrays


```
val sumArray = arrayOf(1, 2, 3)

// Sums array elements
println(sumArray.sum())
// 6
```

```
val simpleArray = arrayOf(1, 2, 3)

// Shuffles elements [3, 2, 1]
simpleArray.shuffle()
println(simpleArray.joinToString())

// Shuffles elements again [2, 3, 1]
simpleArray.shuffle()
println(simpleArray.joinToString())
```

```
val simpleArray = arrayOf("a", "b", "c", "c")

// Converts to a Set
println(simpleArray.toSet())
// [a, b, c]

// Converts to a List
println(simpleArray.toList())
// [a, b, c, c]
```

```
val pairArray = arrayOf("apple" to 120, "banana" to 150, "cherry" to 90,

// Converts to a Map
// The keys are fruits and the values are their number of calories
// Note how keys must be unique, so the latest value of "apple"
// overwrites the first
println(pairArray.toMap())
// {apple=140, banana=150, cherry=90}
```

Arrays

Ganpat University
॥ विद्यया समाजोत्कर्षः ॥
U.V. Patel College of Engineering

```
// Creates an array of Int of size 5 with the values initialized to zero
val exampleArray = IntArray(5)
println(exampleArray.joinToString())
// 0, 0, 0, 0, 0
```

Classes

Ganpat University
॥ विद्यया समाजोत्कर्षः ॥
U.V. Patel College of Engineering

Classes in Kotlin are declared using the keyword `class`:

```
class Person { /*...*/ }
```

The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor, and some other things), and the class body surrounded by curly braces. Both the header and the body are optional; if the class has no body, the curly braces can be omitted.

```
class Empty
```

Constructors

A class in Kotlin has a **primary constructor** and possibly one or more **secondary constructors**. The primary constructor is declared in the class header, and it goes after the class name and optional type parameters.

```
class Person constructor(firstName: String) { /*...*/ }
```

If the primary constructor does not have any annotations or visibility modifiers, the `constructor` keyword can be omitted:

```
class Person(firstName: String) { /*...*/ }
```

Constructors

The primary constructor initializes a class instance and its properties in the class header. The class header can't contain any runnable code. If you want to run some code during object creation, use **initializer blocks** inside the class body. Initializer blocks are declared with the `init` keyword followed by curly braces. Write any code that you want to run within the curly braces.

During the initialization of an instance, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

Constructors


```
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints $name")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
```

Primary Constructors

Primary constructor parameters can be used in the initializer blocks. They can also be used in property initializers declared in the class body:

```
class Customer(name: String) {
    val customerKey = name.uppercase()
}
```

Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor:

```
class Person(val firstName: String, val lastName: String, var age: Int)
```

Secondary Constructors

A class can also declare **secondary constructors**, which are prefixed with `constructor`:

```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
}
```

Constructors

If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s). Delegation to another constructor of the same class is done using the `this` keyword:

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Code in initializer blocks effectively becomes part of the primary constructor. Delegation to the primary constructor happens at the moment of access to the first statement of a secondary constructor, so the code in all initializer blocks and property initializers is executed before the body of the secondary constructor.

Constructors

Code in initializer blocks effectively becomes part of the primary constructor. Delegation to the primary constructor happens at the moment of access to the first statement of a secondary constructor, so the code in all initializer blocks and property initializers is executed before the body of the secondary constructor.

Even if the class has no primary constructor, the delegation still happens implicitly, and the initializer blocks are still executed:

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}
```

Creating instances of classes

To create an instance of a class, call the constructor as if it were a regular function. You can assign the created instance to a variable:

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

Kotlin does not have a `new` keyword.

Class members

- Classes can contain:
- Constructors and initializer blocks
- Functions
- Properties
- Nested and inner classes
- Object declarations

Inheritance

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

Inheritance

Overriding methods

Kotlin requires explicit modifiers for overridable members and overrides:

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}
```

Inheritance

Overriding properties

The overriding mechanism works on properties in the same way that it does on methods. Properties declared on a superclass that are then redeclared on a derived class must be prefaced with `override`, and they must have a compatible type. Each declared property can be overridden by a property with an initializer or by a property with a `get` method:

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle : Shape() {
    override val vertexCount = 4
}
```

inner class

Inside an inner class, accessing the superclass of the outer class is done using the `super` keyword qualified with the outer class name: `super@Outer`:

```
class FilledRectangle: Rectangle() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle.draw() // Calls Rectangle's implementation of
            fill()
            println("Drawn a filled rectangle with color ${super@FilledRectangle
        }
    }
}
```

Overriding rules

To denote the supertype from which the inherited implementation is taken, use `super` qualified by the supertype name in angle brackets, such as `super<Base>`:

```
open class Rectangle {
    open fun draw() { /* ... */ }
}

interface Polygon {
    fun draw() { /* ... */ } // interface members are 'open' by default
}

class Square() : Rectangle(), Polygon {
    // The compiler requires draw() to be overridden:
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw()
        super<Polygon>.draw() // call to Polygon.draw()
    }
}
```

Null safety

In Kotlin, the type system distinguishes between references that can hold `null` (nullable references) and those that cannot (non-nullable references). For example, a regular variable of type `String` cannot hold `null`:

```
var a: String = "abc" // Regular initialization means non-nullable by default
a = null // compilation error
```

To allow nulls, you can declare a variable as a nullable string by writing `String?`:

```
var b: String? = "abc" // can be set to null
b = null // ok
print(b)
```

Null safety

```kotlin
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length) // Unnecessary safe call
```

```kotlin
val listWithNulls: List<String?> = listOf("Kotlin", null)
for (item in listWithNulls) {
    item?.let { println(it) } // prints Kotlin and ignores null
}
```

Null safety

The !! operator

The third option is for NPE-lovers: the not-null assertion operator (`!!`) converts any value to a non-nullable type and throws an exception if the value is `null` . You can write `b!!` , and this will return a non-null value of `b` (for example, a `String` in our example) or throw an NPE if `b` is `null` :

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

Thus, if you want an NPE, you can have it, but you have to ask for it explicitly and it won't appear out of the blue.

Null safety

Safe casts

Regular casts may result in a `ClassCastException` if the object is not of the target type. Another option is to use safe casts that return `null` if the attempt was not successful:

```
val aInt: Int? = a as? Int
```

Collections of a nullable type

If you have a collection of elements of a nullable type and want to filter non-nullable elements, you can do so by using `filterNotNull` :

```
val nullableList: List<Int?> = listOf(1, 2, null, 4)
val intList: List<Int> = nullableList.filterNotNull()
```

Asynchronous programming techniques

- There have been many approaches to solving this problem, including:
- Threading
- Callbacks
- Futures, promises, and others
- Reactive Extensions
- Coroutines

Threading

Threads are by far probably the most well-known approach to avoid applications from blocking.

```
fun postItem(item: Item) {
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
}

fun preparePost(): Token {
    // makes a request and consequently blocks the main thread
    return token
}
```

Callbacks

With callbacks, the idea is to pass one function as a parameter to another function, and have this one invoked once the process has completed.

```
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}

fun preparePostAsync(callback: (Token) -> Unit) {
    // make request and return immediately
    // arrange callback to be invoked later
}
```

Futures, promises, and others

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token ->
            submitPostAsync(token, item)
        }
        .thenAccept { post ->
            processPost(post)
        }

}

fun preparePostAsync(): Promise<Token> {
    // makes request and returns a promise that is completed later
    return promise
}
```

Coroutines


```
fun postItem(item: Item) {
    launch {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}

suspend fun preparePost(): Token {
    // makes a request and suspends the coroutine
    return suspendCoroutine { /* ... */ }
}
```

Coroutines are not a new concept, let alone invented by Kotlin. They've been around for decades and are popular in some other programming languages such as Go. What is important to note though is that the way they're implemented in Kotlin, most of the functionality is delegated to libraries. In fact, beyond the `suspend` keyword, no other keywords are added to the language. This is somewhat different from languages such as C# that have `async` and `await` as part of the syntax. With Kotlin, these are just library functions.

Coroutine scope


```
class Activity {
    private val mainScope = MainScope()

    fun destroy() {
        mainScope.cancel()
    }
```

```
// class Activity continues
    fun doSomething() {
        // launch ten coroutines for a demo, each working for a different time
        repeat(10) { i ->
            mainScope.launch {
                delay((i + 1) * 200L) // variable delay 200ms, 400ms, ... etc
                println("Coroutine $i is done")
            }
        }
    }
} // class Activity ends
```

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

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

  public String getName() {
    return name;
  }

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

Enter action or option name:

convert Java to Kotlin

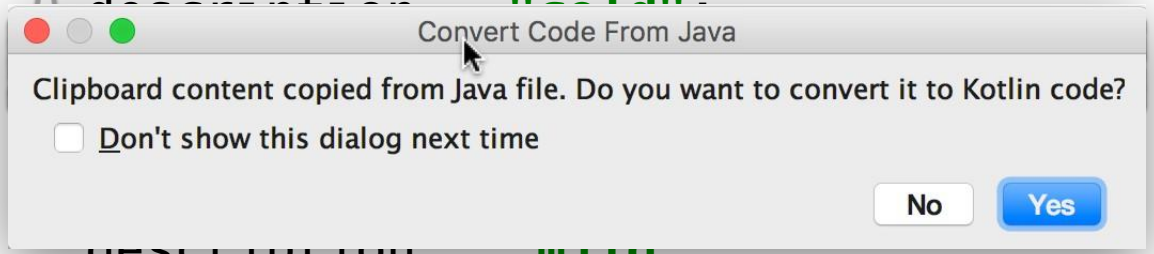
Convert Java File to Kotlin File (⌥⇧⌘K) Code

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 if (degrees < 15) {
        description = "mild";
        colour = ORANGE;
    } else {
        description = "hot";
        colour = RED;
    }
    // ...
}
```



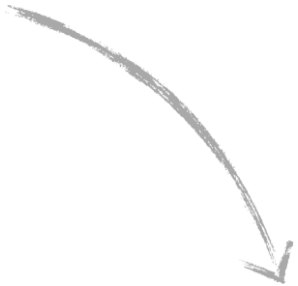
```
fun updateWeather(degrees: Int) {
    val description: String
    val colour: Colour
    if (degrees < 5) {
        description = "cold"
        colour = BLUE
    } else if (degrees < 23) {
        description = "mild"
        colour = ORANGE
    } else {
        description = "hot"
        colour = RED
    }
    // ...
}
```

```
fun updateWeather(degrees: Int) {
    val (description: String, colour: Colour) =
        if (degrees < 5) {
            Pair("cold", BLUE)
        } else if (degrees < 23) {
            Pair("mild", ORANGE)
        } else {
            Pair("hot", RED)
        }
    // ...
}
```

```
fun updateWeather(degrees: Int) {
    val (description, colour) =
        if (degrees < 5) {
            Replace 'if' with 'when'
        } else if (degrees < 23) {
            Pair("mild", ORANGE)
        } else {
            Pair("hot", RED)
        }
    // ...
}
```

```
fun updateWeather(degrees: Int) {
    val (description, colour) = when {
        degrees < 5 -> Pair("cold", BLUE)
        degrees < 23 -> Pair("mild", ORANGE)
        else -> Pair("hot", RED)
    }
    // ...
}
```

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

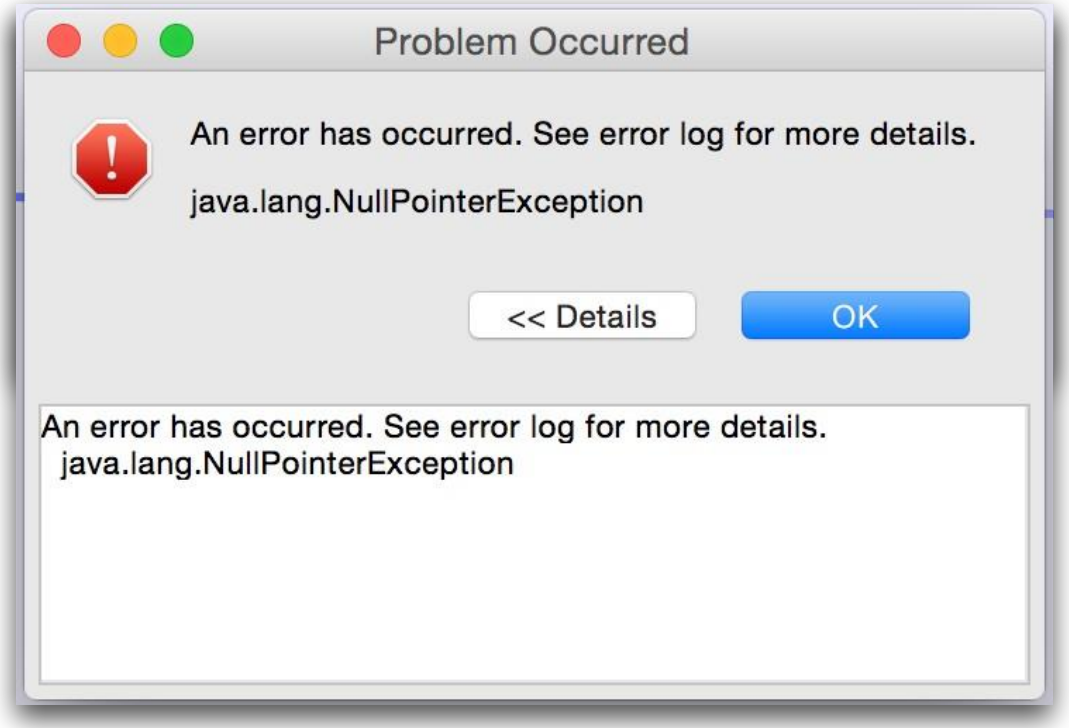


```
val (description, colour) = when {
    degrees < 5 -> Pair("cold", BLUE)
    degrees < 23 -> Pair("mild", ORANGE)
    else -> Pair("hot" to RED)
}
```

Ganpat University
॥ विद्यया समाजोत्कर्षः ॥
U.V. Patel College of Engineering

safe

Billion Dollar Mistake



Modern approach:
to make NPE
compile-time error,
not run-time error

Nullable types in Kotlin

```
val s1: String = null
```

✗

```
val s2: String? = "can be null or non-null"
```

```
s1.length
```

✓

```
s2.length
```

✗

Dealing with Nullable Types


```
val s: String?

if (s != null) {
    s.
}
```

Replace 'if' expression with safe access expression

Dealing with Nullable Types


```
val s: String?

s?.length
```

Nullability operators


```
val s: String?

val length = if (s != null) s.length else null
```

```
val length = s?.length
```

Nullability operators


```
val s: String?

val length = if (s != null) s.length else null
```

```
val length: Int? = s?.length
```

Nullability operators


```
val s: String?

val length = if (s != null) s.length else 0
```

```
val length: Int = s?.length ?: 0
```

Making NPE explicit


```
val s: String?

s!!

s!!.length
```

throws NPE if s is null

Nullable Types Under the Hood

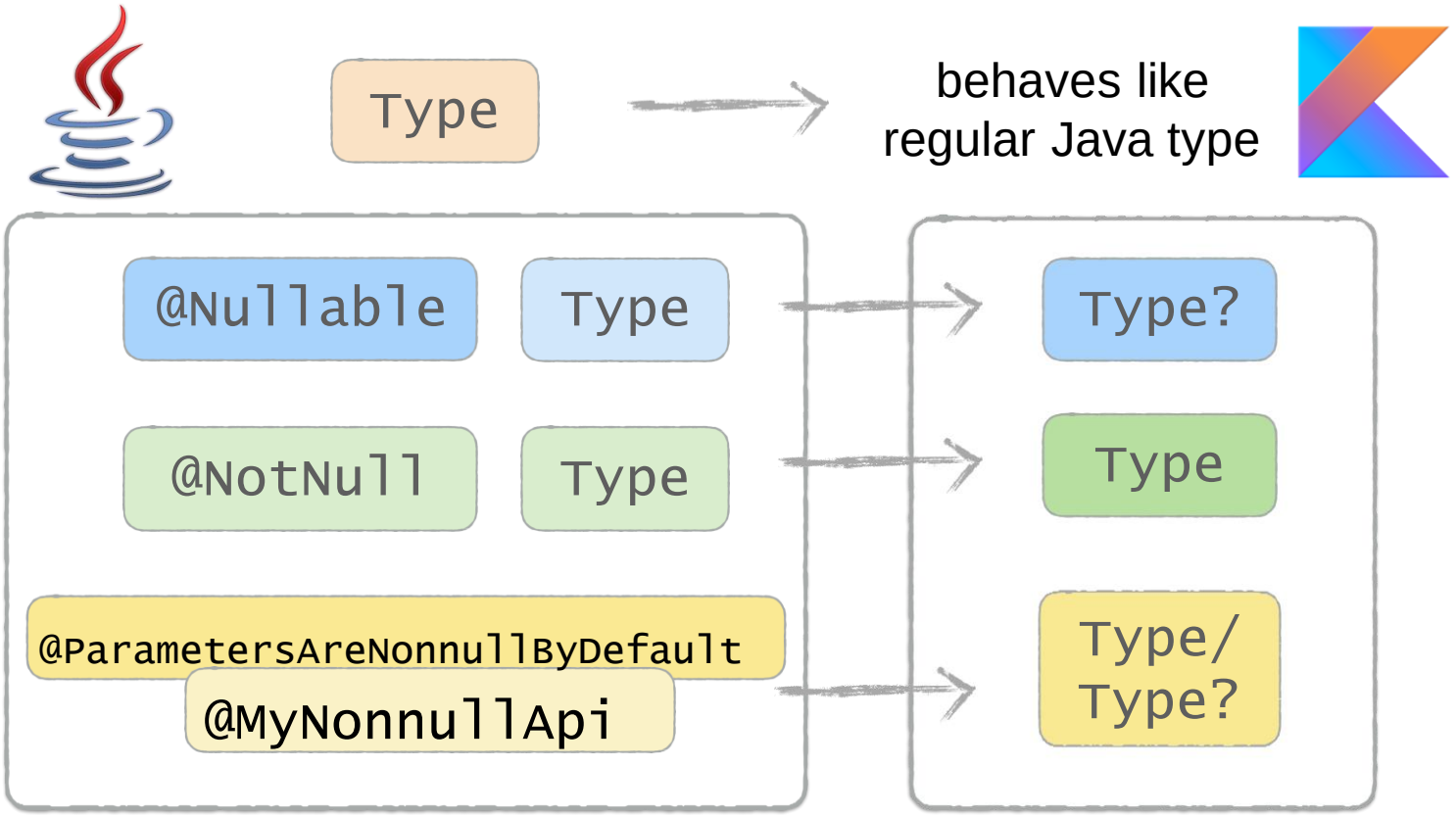
`@Nullable, @NotNull` annotations

No performance overhead

Nullable types ≠ Optional


```
class Optional<T>(val value: T) {

    fun isPresent() = value != null

    fun get() = value ?:
            throw NoSuchElementException("No value present")
}
```

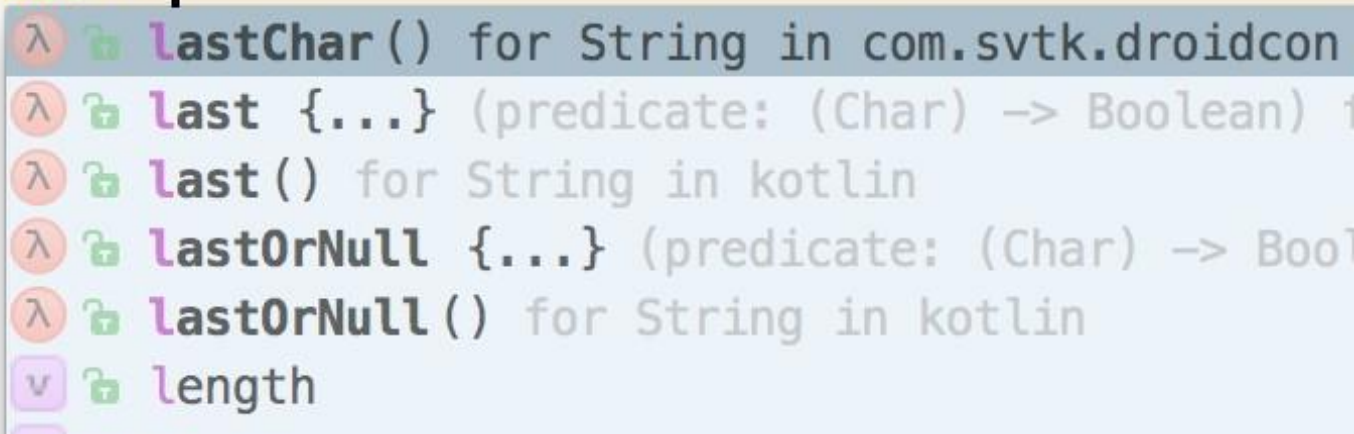
Using annotated Java types from Kotlin



Extension Functions


```
fun String.lastChar() = get(length - 1)
```

```
val c: Char = "abc".l
```



Extension Functions


```
fun String.lastChar() = this.get(this.length - 1)
```

`this` can be omitted

```
fun String.lastChar() = get(length - 1)
```

Calling Extension Functions from Java code

StringExtensions.kt

```kotlin
fun String.lastChar() = get(length - 1)
```

JavaClass.java

```java
import static StringExtensionsKt.lastChar;
char c = lastChar("abc");
```

Extension Functions


```
fun String.lastChar() = get(length - 1)
```

Is it possible to call a private member of String here?

No. Because it's a regular static method under the hood.

Lambdas

Lambdas

Ganpat University
॥ विद्यया समाजोत्कर्षः ॥

U.V. Patel College of Engineering

```
button.addActionListener { println("Hi") }
```

Working with collections with Lambdas

```
val employees: List<Employee>

data class Employee(
    val city: City, val age:
)   Int
```

What's an average age of employees working in Prague?

```
employees.filter { it.city == City.PRAGUE }.map { it.age
}.average()
```

```
{ employee: Employee -> employee.city == City.PRAGUE
}
```

Working with collections with Lambdas

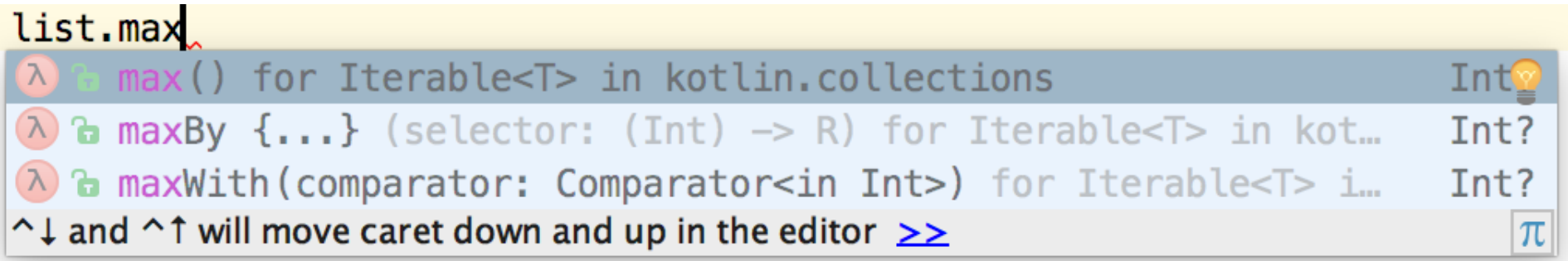
```
val employees: List<Employee>

data class Employee(
    val city: City, val age:
)   Int
```

What's an average age of employees working in Prague?

```
employees.filter { it.city == City.PRAGUE }.map { it.age
}.average()
```

extension functions

Kotlin library: extensions on collections



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

Extension Function & Lambda

Lambda with receiver

The with function


```
val sb = StringBuilder()
sb.appendln("Alphabet: ")
for (c in 'a'..'z') {
    sb.append(c)
}
sb.toString()
```

with is a function

```
val sb = StringBuilder()
with (sb) {
    appendln("Alphabet: ")
    for (c in 'a'..'z') {
        append(c)
    }
    toString()
}
```

Lambda with receiver

`with` is a function

```
val sb = StringBuilder()
with (sb, { ->
this.appendln("Alphabet: ")
for (c in 'a'..'z') {
    this.append(c)
}
this.toString()
})
```

lambda is its second argument

`this` is an implicit receiver in the lambda

Lambda with receiver

lambda with implicit `this`

```
val sb = StringBuilder()
with (sb) {
    appendln("Alphabet: ")
    for (c in 'a'..'z') {
        this.append(c)
    }
}
```

Under the Hood

Lambdas can be inlined

No performance overhead

The with function

```
class Window(
    var width: Int,
    var height: Int,
    var isVisible: Boolean
)
```

```
with (window) {
    width = 300
    height = 200
    isVisible = true
}
```

Inline functions

is declared as **`inline`** function

```
with (window) {
    width = 300
    height = 200
    isVisible = true
}
```

generated bytecode is similar to:

```
window.width = 300
window.height = 200
window.isVisible = true
```

No performance overhead for creating a lambda!

The apply function

Applies the given actions
and returns receiver as a result:

```
val mainWindow =
    windowById["main"]?.apply {
        width = 300
        height = 200
        isVisible = true
    } ?: return
```