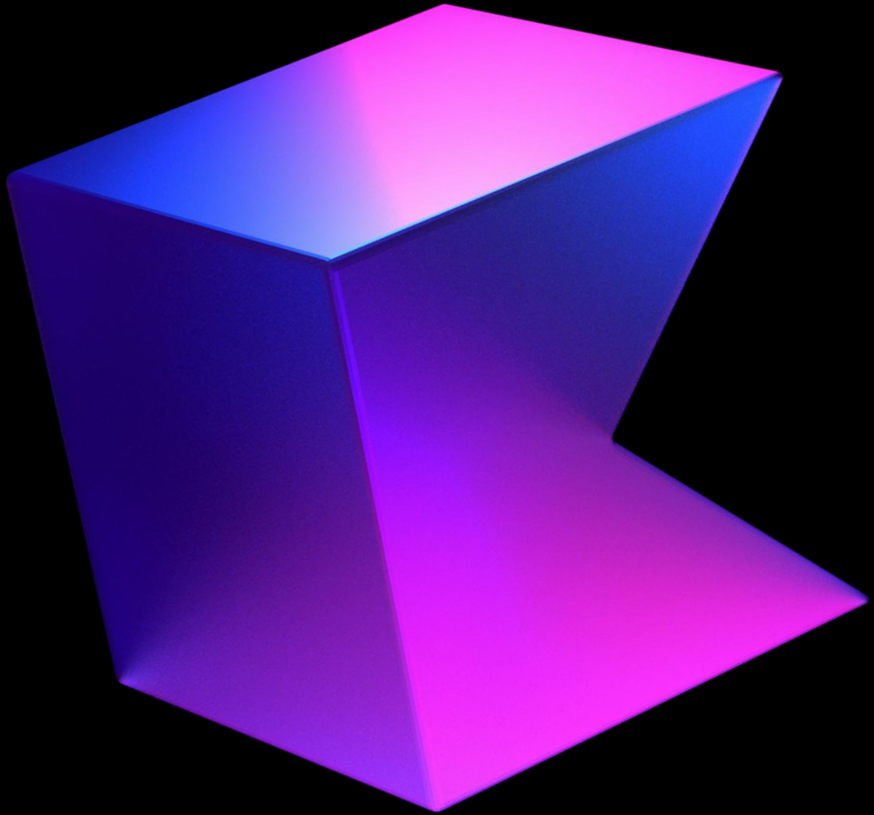
Kotlin

Functions



@kotlin | Developed by JetBrains

Introduction

- In Kotlin, functions can be defined either inside a class or without any class inside the file.
- Functions are defined using the **fun** keyword.
- If a function doesn't return anything, then it returns **Unit**.
- **Unit** doesn't have to be written within the function's header.

Defining Function

```kotlin
fun greet(): Unit {
    println("Hello!")
}
```

$\xrightarrow{\text{The same as}}$

```kotlin
fun greet() {
    println("Hello!")
}
```

```kotlin
fun greet(name: String) {
    println("Hello $name!")
}
```

```kotlin
fun sum (num1: Int, num2: Int): Int {
    return num1 + num2
}
```

Named Arguments

You can name one or more of a function's arguments when calling it.

This can be helpful when a function has many arguments and it's difficult to associate a value with an argument.

When you use named arguments in a function call,
you can freely change the order that they are listed in.

Named Arguments – Example

```
fun reformat(str: String,
             normalizeCase: Boolean,
             upperCaseFirstLetter: Boolean ,
             wordSeparator: Char) {
    /*...*/
}

// calling the function using the named parameters
reformat("String!", false, upperCaseFirstLetter = false, '_' )
```

Default Arguments

Function parameters can have default values,
which are used when you skip the corresponding argument.

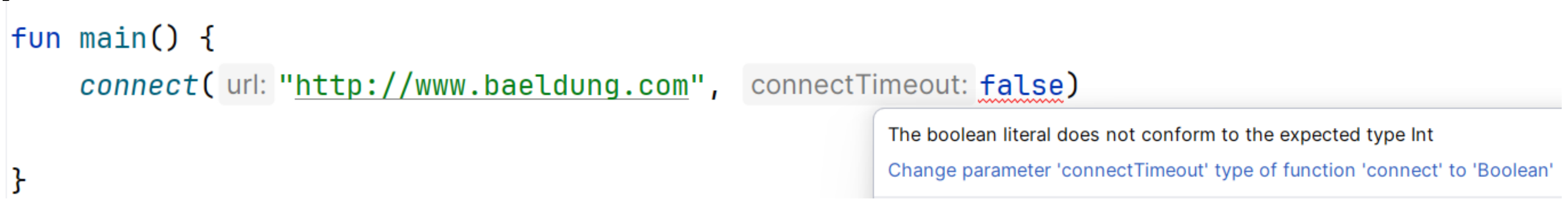
This reduces the number of **overloads**.

```
fun sum(num1: Int = 0, num2: Int = 0, num3: Int = 0): Int {
    return num1 + num2 + num3
}
//calling the function:
sum()
sum(1)
sum(1,2)
sum(1,2,3)
```

Using Named & Default Arguments

Once an argument with a default value is skipped,
all subsequent arguments must be passed as named arguments.

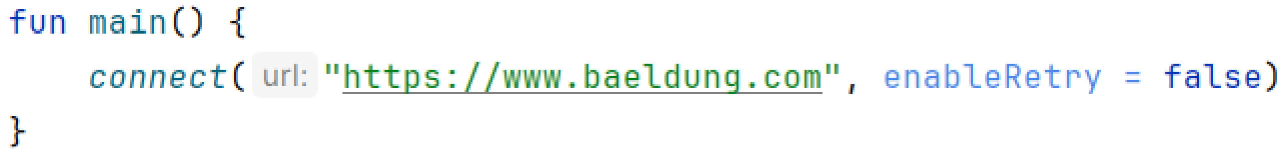
```kotlin
fun connect(url: String,
            connectTimeout: Int = 1000,
            enableRetry: Boolean = true) {
    println("The parameters are url = $url,Timeout = $connectTimeout")
}
```



```kotlin
fun main() {
    connect( url: "http://www.baeldung.com",  connectTimeout: false)

}
```

The boolean literal does not conform to the expected type Int
Change parameter 'connectTimeout' type of function 'connect' to 'Boolean'



```kotlin
fun main() {
    connect( url: "https://www.baeldung.com", enableRetry = false)
}
```

Single-Expression Functions

When the function body consists of a single expression,
the curly braces can be omitted and the body specified after an **=** symbol.

```
fun subtract(num1: Int, num2: Int ): Int{
    return num1 - num2
}

//As a single-expression function
fun subtract(num1: Int, num2: Int ): Int = num1 - num2

// Explicitly declaring the return type is optional
// when this can be inferred by the compiler
fun subtract(num1: Int, num2: Int) = num1 - num2
```

Extensions

Kotlin provides the ability to extend a class or an interface with new functionality without having to inherit from the class via **Extensions**

To declare an extension function, prefix its name with a receiver type, which refers to the type being extended.

```
fun ReceiverType.functionName(parameters) : ReturnType {
    //use this to refer to the reciever
}
```

The class that is being extended does not change at all; it is simply a new function that can be called like a method. It **cannot access private members**, for example.

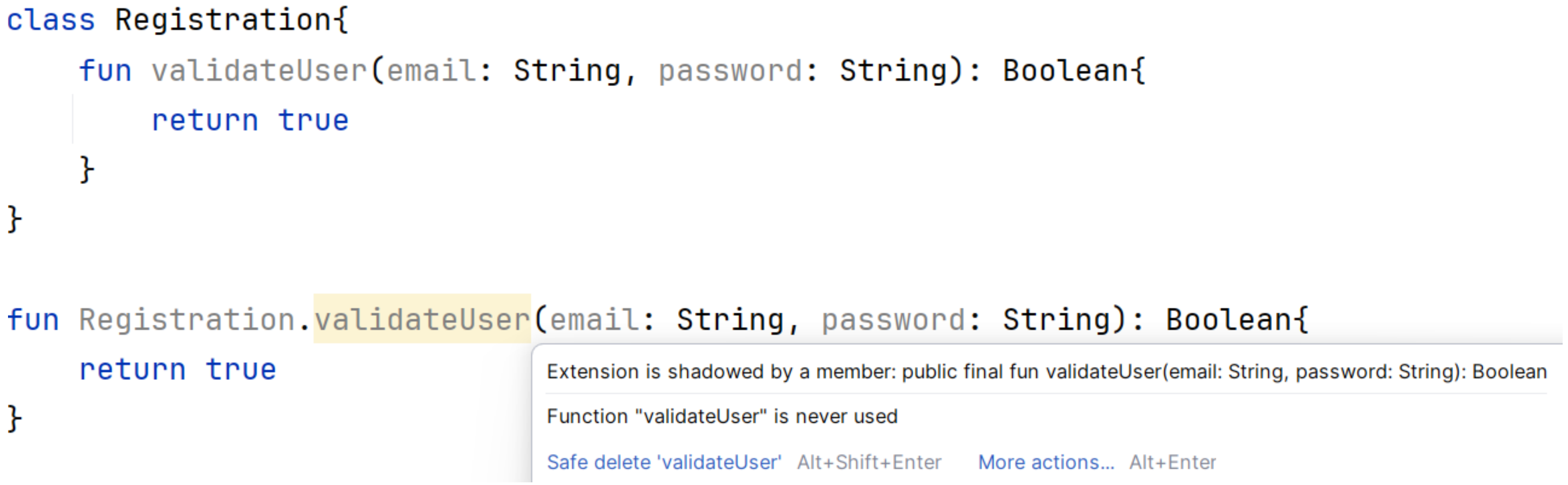
Extensions - Example

```kotlin
fun String.checkIfEven(): Boolean {
    val result : Boolean = if(this.length % 2 == 0) {
        true
    } else {
        false
    }
    return result
}

fun main(){
    val name : String = "Kotlin"
    val isEven: Boolean = name.checkIfEven() // using extension
    println("Does $name have an even length? $isEven")
}
```

Extensions

If the extended class **already has** the new method with the same name and signature, the **original** one will be used.



```kotlin
class Registration{
    fun validateUser(email: String, password: String): Boolean{
        return true
    }
}

fun Registration.validateUser(email: String, password: String): Boolean{
    return true
}
```

```
Extension is shadowed by a member: public final fun validateUser(email: String, password: String): Boolean
Function "validateUser" is never used
Safe delete 'validateUser'  Alt+Shift+Enter     More actions...  Alt+Enter
```

Extensions under the hood

Extensions have static dispatch, rather than virtual dispatch by receiver type.
An extension function being called is determined by the type of the expression on which the function is invoked, not by the type of the result from evaluating that expression at runtime.

```kotlin
open class Shape
class Rectangle: Shape()

fun Shape.getName() = "Shape"
fun Rectangle.getName() = "Rectangle"

fun printClassName(s: Shape) {
    println(s.getName())
}

printClassName(Rectangle()) // "Shape", not Rectangle
```

Infix functions

```kotlin
data class Person(val name: String, val surname: String)

infix fun String.with(other: String) = Person(this, other)

fun main() {
   val realHero = "Ryan" with "Gosling"
   val (real, bean) = realHero
}
```

What is an Infix Function?

An infix function is a special type of function in Kotlin that:

- Is called using **infix** notation: a with b instead of a.with(b)
- Must be a **member** or **extension** function
- Must have **only one** parameter
- Must be marked with the **infix** keyword

Key Concepts

✅ **Readable Syntax**

Makes the code more expressive and closer to natural language.

✅ **`this` in Extension Function**

- Refers to the string on the left (`"Ryan"`)
- `other` refers to the string on the right (`"Gosling"`)

✅ **Destructuring**

`Person` is a `data class`, so we can unpack it like this:

```
val (real, bean) = realHero
```

ComponentN operator

```kotlin
class SomeData(val list: List<Int>) {
   operator fun component1() = list.first()
   operator fun component2() = SomeData(list.subList(1, list.size))
   operator fun component3() = "This is weird"
}

fun main() {
   val sd = SomeData(listOf(1, 2, 3))
   val (head, tail, msg) = sd
   val (h, t) = sd
   val (onlyComponent1) = sd
}
```

Any class can overload any number of `componentN` methods that can be used in **destructive declarations**.

Data classes have these methods by default.

Type Check

In Kotlin, you can perform type checks to check the type of an object at **runtime**.

Use the `is` operator or its negated form `!is` to perform a **runtime check** that identifies whether an object conforms to a given type.

```
fun checkMessageType(msg: Any){
    if (msg is String){
        println("String Message says: $msg")
    } else{
        println("Unknown Message Type")
    }
}
```

Casting

Type casts convert objects to a different type. Casting can be:

1. **Unsafe:** using the `as` operator.

   Usually, the cast operator **throws an exception** <u>if the cast isn't possible</u>. Thus, it's called **unsafe**.

2. **Safe:** using the `as?` Operator.

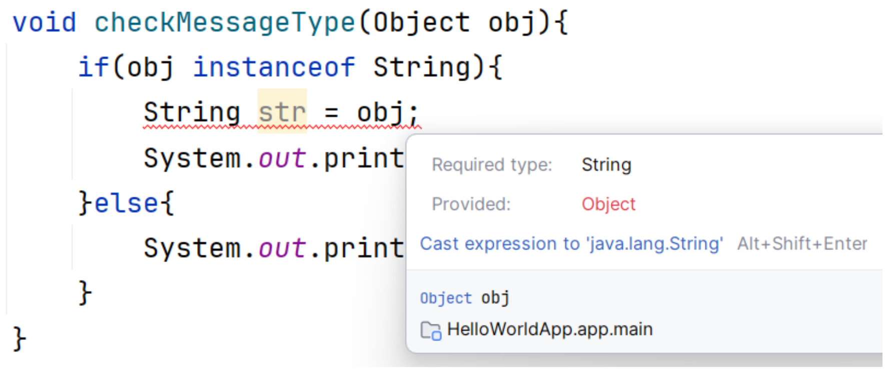
   To avoid exceptions, use the safe cast operator `as?`, which returns **null** on failure.

   - Despite the fact that the right-hand side of `as?` is a non-nullable type String,

     the result of the cast is nullable.

Smart Cast (is)

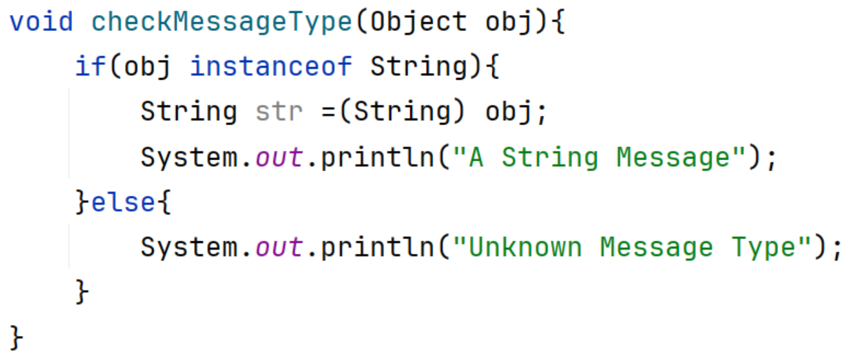
In most cases, you don't need to use explicit cast operators in Kotlin because the compiler tracks the **is-checks** and explicit casts for immutable values and inserts (safe) casts automatically when necessary



```
void checkMessageType(Object obj){
    if(obj instanceof String){
        String str = obj;
        System.out.print
    }else{
        System.out.print
    }
}
```

Required type: String
Provided: Object
Cast expression to 'java.lang.String' Alt+Shift+Enter

Object obj
HelloWorldApp.app.main



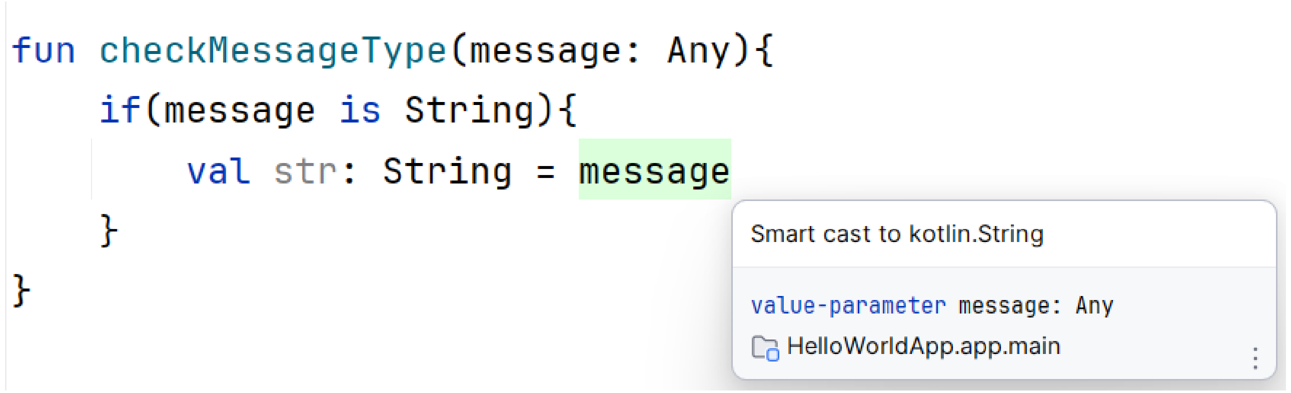
```
void checkMessageType(Object obj){
    if(obj instanceof String){
        String str =(String) obj;
        System.out.println("A String Message");
    }else{
        System.out.println("Unknown Message Type");
    }
}
```

Even though the object proved it’s an instance of String.
There is a **compile–time error,** and Java asks you to make an **explicit casting**.

Smart Casting (is)

In Kotlin, Since the object has proved to be a String.

Kotlin automatically converts the object to a String



```
fun checkMessageType(message: Any){
    if(message is String){
        val str: String = message
    }
}
```

Smart cast to kotlin.String

```
value-parameter message: Any
```

HelloWorldApp.app.main

Today's Challenge(s)