KOTLIN - BASICS

TODAY'S AGENDA

- Kotlin Basics – Basic Syntax, Idioms, Kotlin By Example, Coding Conventions
- Kotlin Concepts - Basic types, Type checks and casts
- Kotlin Concepts - Control flow - Conditions and loops, Returns and jumps Exceptions
-

Idioms

- A collection of random and frequently used idioms in Kotlin

Create DTOs (POJOs/POCOs)

- data class Customer(val name: String, val email: String)
- provides a Customer class with the following functionality:
  - getters (and setters in case of vars) for all properties
  - equals()
  - hashCode()
  - toString()
  - copy()
  - component1(), component2(), ..., for all properties

Idioms

- **Default values for function parameters**
  - fun foo(a: Int = 0, b: String = "") { ... }
- **Filter a list**
  - val positives = list.filter { x -> x > 0 }
- Or alternatively, even shorter:
  - val positives = list.filter { it > 0 }

Java and Kotlin filtering.

- In Java, to filter elements from a collection, you need to use the Stream API. The Stream API has intermediate and terminal operations. filter() is an intermediate operation, which returns a stream. To receive a collection as the output, you need to use a terminal operation, like collect(). For example, to leave only those pairs whose keys end with 1 and whose values are greater than 10:
- In Kotlin, filtering is built into collections, and filter() returns the same collection type that was filtered. So, all you need to write is the filter() and its predicate:

```java
// Java
public void filterEndsWith() {
    var numbers = Map.of("key1", 1, "key2", 2, "key3", 3, "key11", 11);
    var filteredNumbers = numbers.entrySet().stream()
        .filter(entry -> entry.getKey().endsWith("1") && entry.getValue() > 10)
        .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
    System.out.println(filteredNumbers);
}
```

```kotlin
// Kotlin
val numbers = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)
val filteredNumbers = numbers.filter { (key, value) -> key.endsWith("1") && value
println(filteredNumbers)
```

- **Check the presence of an element in a collection**
  - if ("john@example.com" in emailsList) { ... }
  - if ("jane@example.com" !in emailsList) { ... }
- **String interpolation**
  - println("Name $name")

Concatenate strings Java Vs Kotlin

In Java, you can do this in the following way:

```java
// Java
String name = "Joe";
System.out.println("Hello, " + name);
System.out.println("Your name is " + name.length() + " characters long");
```

In Kotlin, use the dollar sign (`$`) before the variable name to interpolate the value of this variable into your string:

```kotlin
// Kotlin
val name = "Joe"
println("Hello, $name")
println("Your name is ${name.length} characters long")
```

- **Instance checks**
  - when (x) {
  - is Foo -> ...
  - is Bar -> ...
  - else -> ...
  - }
- **Read-only list**
  - val list = listOf("a", "b", "c")
- **Read-only map**
  - val map = mapOf("a" to 1, "b" to 2, "c" to 3)

- **Access a map entry**
  - println(map["key"])
  - map["key"] = value
- **Traverse a map or a list of pairs**
  - for ((k, v) in map) {
  - println("$k -> $v")
  - }
  - k and v can be any convenient names, such as name and age.
- **Iterate over a range**
  - for (i in 1..100) { ... } // closed range: includes 100
  - for (i in 1 until 100) { ... } // half-open range: does not include 100
  - for (x in 2..10 step 2) { ... }
  - for (x in 10 downTo 1) { ... }
  - (1..10).forEach { ... }

- **Lazy property**
  - val p: String by lazy { // the value is computed only on first access // compute the string }
- **Extension functions**
  - fun String.spaceToCamelCase() { ... }
  - "Convert this to camelcase".spaceToCamelCase()
- **Create a singleton**
  - object Resource { val name = "Name" }

Instantiate an abstract class

```
abstract class MyAbstractClass {
    abstract fun doSomething()
    abstract fun sleep()
}

fun main() {
    val myObject = object : MyAbstractClass() {
        override fun doSomething() {
            // ...
        }

        override fun sleep() { // ...
        }
    }
    myObject.doSomething()
}
```

- **If-not-null shorthand**
  - val files = File("Test").listFiles()
  - println(files?.size) // size is printed if files is not null
- **If-not-null-else shorthand**
  - val files = File("Test").listFiles()
  - println(files?.size ?: "empty") // if files is null, this prints //"empty“
  - // To calculate the fallback value in a code block, use //`run`
  - val filesSize = files?.size ?: run {
    - return someSize }
- println(filesSize)

Get the first and the last items of a possibly empty collection

In Java, you can safely get the first and the last items by checking the size of the collection and using indices:

```java
// Java
var list = new ArrayList<>();
//...
if (list.size() > 0) {
    System.out.println(list.get(0));
    System.out.println(list.get(list.size() - 1));
}
```

In Kotlin, there are the special functions `firstOrNull()` ↗ and `lastOrNull()` ↗. Using the `Elvis operator`, you can perform further actions right away depending on the result of a function. For example, `firstOrNull()`:

```kotlin
// Kotlin
val emails = listOf<String>() // Might be empty
val theOldestEmail = emails.firstOrNull() ?: ""
val theFreshestEmail = emails.lastOrNull() ?: ""
```

- **Execute if not null**
  - val value = ...
  - value?.let { ...
  - // execute this block if not null
  - }
- **Map nullable value if not null**
  - val value = ...
  - val mapped = value?.let { transformValue(it) } ?: defaultValue // defaultValue is returned if the value or the transform result //is null.

- **Return on when statement**
  - fun transform(color: String): Int {
  - return when (color) {
  - "Red" -> 0
  - "Green" -> 1
  - "Blue" -> 2
  - else -> throw IllegalArgumentException("Invalid color param value")
  - } }

try-catch expression

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // Working with result
}
```

if expression

```
val y = if (x == 1) {
    "one"
} else if (x == 2) {
    "two"
} else {
    "other"
}
```

- **Swap two variables**
  - var a = 1
  - var b = 2
  - a = b.also { b = a }
- **Mark code as incomplete (TODO)**
  - Kotlin's standard library has a TODO() function that will always throw a NotImplementedError. Its return type is Nothing so it can be used regardless of expected type. There's also an overload that accepts a reason parameter:
  - fun calcTaxes(): BigDecimal = TODO("Waiting for feedback from accounting")
  - IntelliJ IDEA's kotlin plugin understands the semantics of TODO() and automatically adds a code pointer in the TODO tool window.

Basic types

- In Kotlin, everything is an object in the sense that we can call member functions and properties on any variable.
- Some types can have a special internal representation - for example, numbers, characters and booleans can be represented as primitive values at runtime - but to the user they look like ordinary classes.

Numbers

- **Integer types**
  - Kotlin provides a set of built-in types that represent numbers.
    For integer numbers, there are four types with different sizes and, hence, value ranges.
  - All variables initialized with integer values not exceeding the maximum value of Int have the inferred type Int. If the initial value exceeds this value, then the type is Long. To specify the Long value explicitly, append the suffix L to the value.
    - val one = 1 // Int
    - val threeBillion = 3000000000 // Long
    - val oneLong = 1L // Long
    - val oneByte: Byte = 1

Type	Size (bits)	Min value	Max value
`Byte`	8	-128	127
`Short`	16	-32768	32767
`Int`	32	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
`Long`	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63} - 1$)

- **Floating-point types**
- For real numbers, Kotlin provides floating-point types Float and Double. According to the IEEE 754 standard, floating point types differ by their *decimal place*, that is, how many decimal digits they can store. Float reflects the IEEE 754 *single precision*, while Double provides *double precision*.
- You can initialize Double and Float variables with numbers having a fractional part. It's separated from the integer part by a period (.) For variables initialized with fractional numbers, the compiler infers the Double type.
- val pi = 3.14 // Double
- // val one: Double = 1
- // Error: type mismatch val oneDouble = 1.0 // Double

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
`Float`	32	24	8	6-7
`Double`	64	53	11	15-16

- To explicitly specify the Float type for a value, add the suffix f or F. If such a value contains more than 6-7 decimal digits, it will be rounded.
- val e = 2.7182818284 // Double
- val eFloat = 2.7182818284f // Float, actual value is 2.7182817

- Note that unlike some other languages, there are no implicit widening conversions for numbers in Kotlin. For example, a function with a Double parameter can be called only on Double values, but not Float, Int, or other numeric values.
- fun main() {
- fun printDouble(d: Double) { print(d) }
- val i = 1
- val d = 1.0
- val f = 1.0f
- printDouble(d)
- // printDouble(i) // Error: Type mismatch
- // printDouble(f) // Error: Type mismatch }

Explicit conversions

- Due to different representations, smaller types *are not subtypes* of bigger ones. If they were, we would have troubles of the following sort:
  - // Hypothetical code, does not actually compile:
  - val a: Int? = 1 // A boxed Int (java.lang.Integer)
  - val b: Long? = a // implicit conversion yields a boxed Long (java.lang.Long)
  - print(b == a) // Surprise! This prints "false" as Long's equals() checks whether the other is Long as well
- So equality would have been lost silently, not to mention identity.

- As a consequence, smaller types *are NOT implicitly converted* to bigger types. This means that assigning a value of type Byte to an Int variable requires an explicit conversion.
  - val b: Byte = 1 // OK, literals are checked statically
  - // val i: Int = b // ERROR
  - val i1: Int = b.toInt()

- All number types support conversions to other types:
  - toByte(): Byte
  - toShort(): Short
  - toInt(): Int
  - toLong(): Long
  - toFloat(): Float
  - toDouble(): Double
  - toChar(): Char
- In many cases, there is no need for explicit conversions because the type is inferred from the context, and arithmetical operations are overloaded for appropriate conversions, for example:
- val l = 1L + 3 // Long + Int => Long

Operations

- Kotlin supports the standard set of arithmetical operations over numbers: +, -, *, /, %. They are declared as members of appropriate classes.
  - println(1 + 2)
  - println(2_500_000_000L - 1L)
  - println(3.14 * 2.71)
  - println(10.0 / 3)
- You can also override these operators for custom classes.

Booleans

- The type Boolean represents boolean objects that can have two values: true and false.
- Boolean has a nullable counterpart Boolean? that also has the null value.
- Built-in operations on booleans include:
  - || – disjunction (logical *OR*)
  - && – conjunction (logical *AND*)
  - ! - negation (logical *NOT*)
- || and && work lazily.
  - val myTrue: Boolean = true
  - val myFalse: Boolean = false
  - val boolNull: Boolean? = null

  - println(myTrue || myFalse)
  - println(myTrue && myFalse)
  - println(!myTrue)

Characters

- Characters are represented by the type Char. Character literals go in single quotes: '1'.
- Special characters start from an escaping backslash \. The following escape sequences are supported: \t, \b, \n, \r, \', \", \\ and \$.
- To encode any other character, use the Unicode escape sequence syntax: '\uFF00'.

```
val aChar: Char = 'a'

println(aChar)
println('\n') //prints an extra newline character
println('\uFF00‘)
```

- If a value of character variable is a digit, you can explicitly convert it to an Int number using the digitToInt() function.

Strings

- Strings in Kotlin are represented by the type String. Generally, a string value is a sequence of characters in double quotes (").
  - val str = "abcd 123"
- Elements of a string are characters that you can access via the indexing operation: s[i]. You can iterate over these characters with a for loop:
  - for (c in str) {
  - println(c)
  - }

- Strings are immutable. Once you initialize a string, you can't change its value or assign a new value to it. All operations that transform strings return their results in a new String object, leaving the original string unchanged.
  - **val** str = **"abcd“**
  - println(str.uppercase()) // Create and print a new String objectprintln(str) // the original string remains the same
- To concatenate strings, use the + operator. This also works for concatenating strings with values of other types, as long as the first element in the expression is a string:
  - val s = "abc" + 1
  - println(s + "def")

String literals

- Kotlin has two types of string literals:
  - *escaped* strings that may contain escaped characters
  - *raw* strings that can contain newlines and arbitrary text
- Here's an example of an escaped string:
  - val s = "Hello, world!\n"
- Escaping is done in the conventional way, with a backslash (\).
- A raw string is delimited by a triple quote (""""), contains no escaping and can contain newlines and any other characters:

- val text = """
  - for (c in "foo")
  - print(c)
  - """
- To remove leading whitespace from raw strings, use the trimMargin() function:
  - val text = """
  - |Tell me and I forget.
  - |Teach me and I remember.
  - |Involve me and I learn.
  - |(Benjamin Franklin)
  - """.trimMargin()

String templates

- String literals may contain *template* expressions - pieces of code that are evaluated and whose results are concatenated into the string. A template expression starts with a dollar sign ($) and consists of either a name:
  - val i = 10
  - println("i = $i") // prints "i = 10"
- or an expression in curly braces:

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

Arrays

- Arrays in Kotlin are represented by the Array class. It has get and set functions that turn into [] by operator overloading conventions, and the size property, along with other useful member functions:
  - class Array<T> private constructor() {
  - val size: Int
  - operator fun get(index: Int): T
  - operator fun set(index: Int, value: T): Unit
  - operator fun iterator(): Iterator<T>
  - // ...
  - }
- To create an array, use the function arrayOf() and pass the item values to it, so that arrayOf(1, 2, 3) creates an array [1, 2, 3]. Alternatively, the arrayOfNulls() function can be used to create an array of a given size filled with null elements.
- Another option is to use the Array constructor that takes the array size and the function that returns values of array elements given its index:

- // Creates an Array<String> with values ["0", "1", "4", "9", "16"]
- val asc = Array(5) { i -> (i * i).toString() }
- asc.forEach { println(it) }
- As we said above, the [] operation stands for calls to member functions get() and set().
- Arrays in Kotlin are *invariant*. This means that Kotlin does not let us assign an Array<String> to an Array<Any>, which prevents a possible runtime failure.

Primitive type arrays

- Kotlin also has classes that represent arrays of primitive types without boxing overhead: ByteArray, ShortArray, IntArray, and so on. These classes have no inheritance relation to the Array class, but they have the same set of methods and properties. Each of them also has a corresponding factory function:

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

```
// Array of int of size 5 with values [0, 0, 0, 0, 0]
val arr = IntArray(5)

// e.g. initialise the values in the array with a constant
// Array of int of size 5 with values [42, 42, 42, 42, 42]
val arr = IntArray(5) { 42 }

// e.g. initialise the values in the array using a lambda
// Array of int of size 5 with values [0, 1, 2, 3, 4] (values initialised to their
var arr = IntArray(5) { it * 1 }
```

Type checks and casts

- **is and !is operators**
- Use the is operator or its negated form !is to perform a runtime check that identifies whether an object conforms 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 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:
- fun demo(x: Any) {
- if (x is String) {
  - print(x.length) // x is automatically cast to String
  - }
  - }

- The compiler is smart enough to know that a cast is safe if a negative check leads to a return:
  - if (x !is String) return
  - print(x.length) // x is automatically cast to String
- or if it is on the right-hand side of && or || and the proper check (regular or negative) is on the left-hand side:
- // x is automatically cast to String on the right-hand side of `||`
- if (x !is String || x.length == 0) return
- // x is automatically cast to String on the right-hand side of `&&`
- if (x is String && x.length > 0) {
- print(x.length) // x is automatically cast to String
- }

- Smart casts work
  for when expressions and while loops as well:
- when (x) {
- is Int -> print(x + 1)
- is String -> print(x.length + 1)
- is IntArray -> print(x.sum())
- }
-

- Note that smart casts work only when the compiler can guarantee that the variable won't change between the check and the usage. More specifically, smart casts can be used under the following conditions:
  - val local variables - always, with the exception of local delegated properties.
  - val properties - if the property is private or internal or if the check is performed in the same module where the property is declared. Smart casts cannot be used on open properties or properties that have custom getters.
  - var local variables - if the variable is not modified between the check and the usage, is not captured in a lambda that modifies it, and is not a local delegated property.
  - var properties - never, because the variable can be modified at any time by other code.

“Unsafe" cast operator

- Usually, the cast operator throws an exception if the cast isn't possible. And so, it's called *unsafe*. The unsafe cast in Kotlin is done by the infix operator as.
  - val x: String = y as String
- Note that null cannot be cast to String, as this type is not nullable. If y is null, the code above throws an exception. To make code like this correct for null values, use the nullable type on the right-hand side of the cast:
  - val x: String? = y as String?

- **"Safe" (nullable) cast operator**
- To avoid exceptions, use the *safe* cast operator as?, which returns null on failure.
  - val x: String? = y as? String
- Note that despite the fact that the right-hand side of as? is a non-null type String, the result of the cast is nullable.
- T

Conditions and loops – if

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

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

if

Branches of an `if` expression can be blocks. In this case, the last expression is the value of a block:

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

If you're using `if` as an expression, for example, for returning its value or assigning it to a variable, the `else` branch is mandatory.

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

- 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).

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

```
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
}
```

- 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:
- when (x) {
  - in 1..10 -> print("x is in the range")
  - in validNumbers -> print("x is valid")
  - !in 10..20 -> print("x is outside the range")
  - else -> print("none of the above") }
- Another option is checking that a value is or !is of a particular type. Note that, due to smart casts, you can access the methods and properties of the type without any extra checks.

- fun hasPrefix(x: Any) = when(x) {
- is String -> x.startsWith("prefix")
- else -> false
- }

- when can also be used as a replacement for an if-else if chain. If no argument is supplied, the branch conditions are simply boolean expressions, and a branch is executed when its condition is true:
  - when {
  - x.isOdd() -> print("x is odd")
  - y.isEven() -> print("y is even")
  - else -> print("x+y is odd")
  - }

- You can capture *when* subject in a variable using following syntax:
  - fun Request.getBody() =
  - when (val response = executeRequest()) {
  - is Success -> response.body
  - is HttpError -> throw HttpException(response.status)
  - }
- The scope of variable introduced in *when* subject is restricted to the body of this *when*.

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) {
  - // ... }

- As mentioned before, for iterates through anything that provides an iterator. This means that it:
  - has a member or an extension function iterator() that returns Iterator<>:
    - has a member or an extension function next()
    - has a member or an extension function hasNext() that returns Boolean.
- All of these three functions need to be marked as operator.

- To iterate over a range of numbers, use a range expression:
  - for (i in 1..3) {
  - println(i)
  - }
  - for (i in 6 downTo 0 step 2) {
  - println(i)
  - }

- for loop over a range or an array is compiled to an index-based loop that does not create an iterator object.
- If you want to iterate through an array or a list with an index, you can do it this way:
  - **for** (i **in** array.indices) {
    - println(array[i])
    - }
- Alternatively, you can use the withIndex library function:
  - for ((index, value) in array.withIndex()) {
  - println("the element at $index is $value")
  - }

While loops

- while and do-while loops execute their body continuously while their condition is satisfied. The difference between them is the condition checking time:
  - while checks the condition and, if it's satisfied, executes the body and then returns to the condition check.
  - do-while executes the body and then checks the condition. If it's satisfied, the loop repeats. So, the body of do-while executes at least once regardless of the condition.
-

- while (x > 0) {
- x—
- }

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

Returns and jumps

- Kotlin has three structural jump expressions:
  - return by default returns from the nearest enclosing function or anonymous function.
  - break terminates the nearest enclosing loop.
  - continue proceeds to the next step of the nearest enclosing loop.
- All of these expressions can be used as part of larger expressions:
  - val s = person.name ?: return
- The type of these expressions is the Nothing type.

Break and continue labels

- Any expression in Kotlin may be marked with a *label*. Labels have the form of an identifier followed by the @ sign, such as abc@ or fooBar@. To label an expression, just add a label in front of it.
  - loop@ for (i in 1..100) {
  - // ...
  - }

- Now, we can qualify a break or a continue with a label:
  - loop@ for (i in 1..100) {
    - for (j in 1..100) {
    - if (...) break@loop
    - }
    - }
- A break qualified with a label jumps to the execution point right after the loop marked with that label. A continue proceeds to the next iteration of that loop.

Return to labels

- In Kotlin, functions can be nested using function literals, local functions, and object expressions. Qualified returns allow us to return from an outer function. The most important use case is returning from a lambda expression. Recall that when we write the following, the return-expression returns from the nearest enclosing function - foo:
-

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach {
        if (it == 3) return // non-local return directly to the caller of foo()
        print(it)
    }
    println("this point is unreachable")
}
```

Open in Playground → Target: JVM Running on v.1.7.0

Note that such non-local returns are supported only for lambda expressions passed to inline functions. To return from a lambda expression, label it and qualify the `return`:

```
fun foo() {
    listOf(1, 2, 3, 4, 5).forEach lit@{
        if (it == 3) return@lit // local return to the caller of the lambda - the
        print(it)
    }
    print(" done with explicit label")
}
```

Exceptions

- **Exception classes**
  - 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 the finally block may be omitted. However, at least one catch or finally block is required.

- **Try is an expression**
- 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 (or blocks). The contents of the finally block don't affect the result of the expression.

Checked exceptions

- Kotlin does not have checked exceptions. There are many reasons for this, but we will provide a simple example that illustrates why it is the case.
- The following is an example interface from the JDK implemented by the StringBuilder class:
  - Appendable append(CharSequence csq) throws IOException;

The Nothing type

- throw is an expression in Kotlin, so you can use it, for example, as part of an Elvis expression:
  - val s = person.name ?: throw IllegalArgumentException("Name required")
- The throw expression has the type Nothing. This type has no values and is used to mark code locations that can never be reached. In your own code, you can use Nothing to mark a function that never returns:
  - fun fail(message: String): Nothing {
  - throw IllegalArgumentException(message)
  - }
- When you call this function, the compiler will know that the execution doesn't continue beyond the call:
  - val s = person.name ?: fail("Name required")
  - println(s) // 's' is known to be initialized at this point