Exceptions

We have done this a lot when validating input

In general: throw an exception to indicate that something unusual has happened and the current function cannot proceed with its work

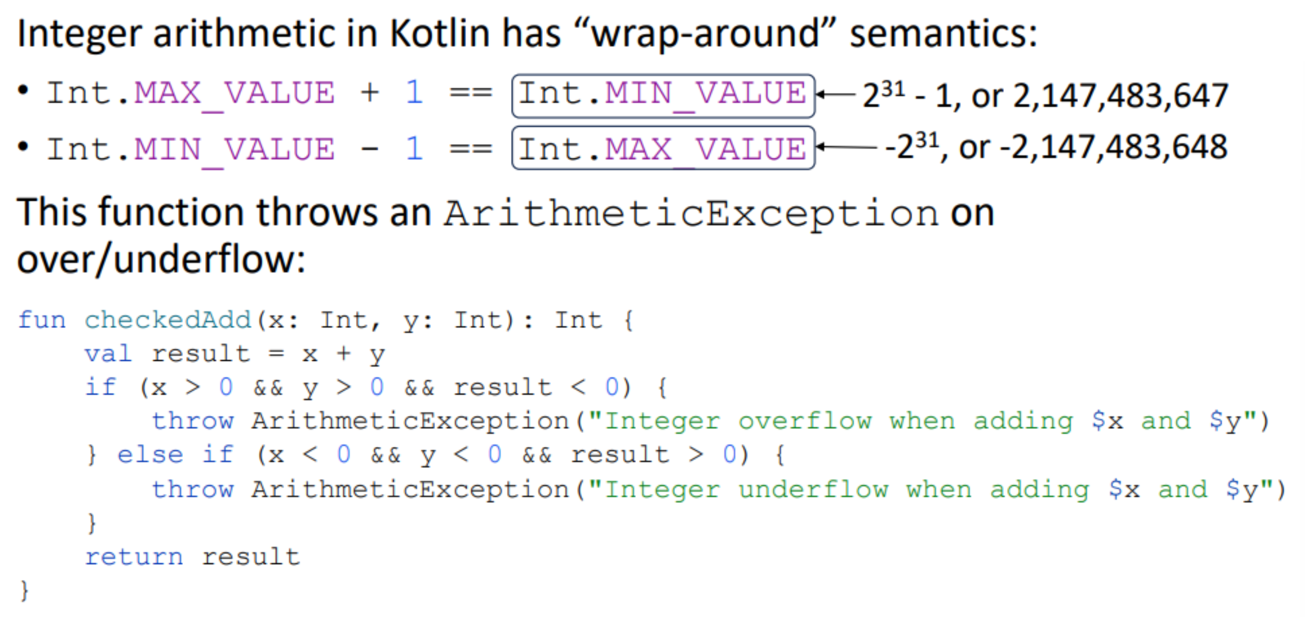
A thrown exception is propagated to the calling function

The calling function can **catch** the exception, to take remedial action

Otherwise, the exception is propagated up the stack to the next caller

If `main` propagates an exception, the program terminates

Example: checked addition



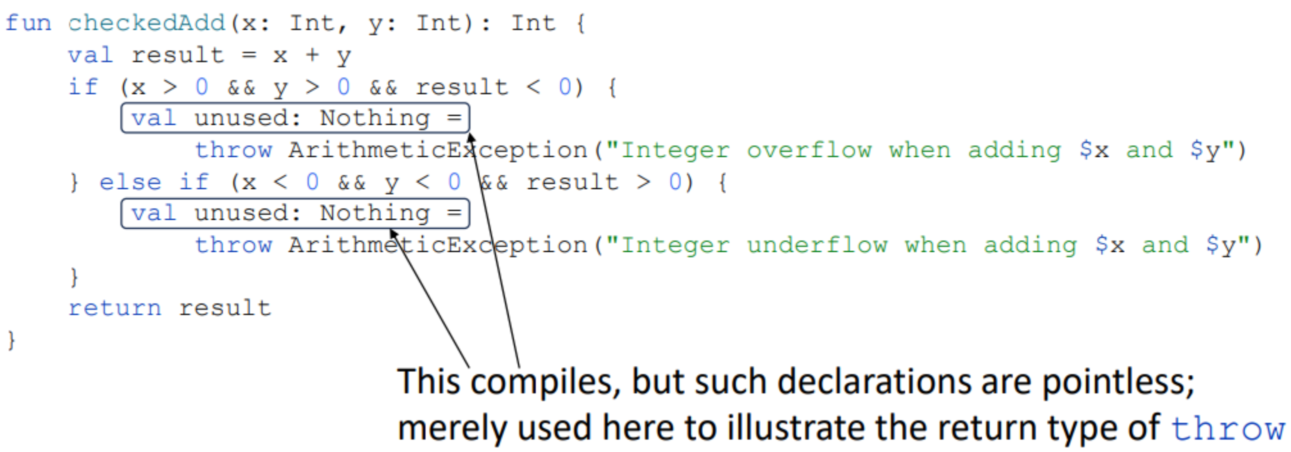
Integer arithmetic in Kotlin has "wrap-around" semantics:

- `Int.MAX_VALUE + 1 == Int.MIN_VALUE` ← 2^{31} - 1, or 2,147,483,647
- `Int.MIN_VALUE - 1 == Int.MAX_VALUE` ← -2^{31}, or -2,147,483,648

This function throws an `ArithmeticException` on over/underflow:

```kotlin
fun checkedAdd(x: Int, y: Int): Int {
    val result = x + y
    if (x > 0 && y > 0 && result < 0) {
        throw ArithmeticException("Integer overflow when adding $x and $y")
    } else if (x < 0 && y < 0 && result > 0) {
        throw ArithmeticException("Integer underflow when adding $x and $y")
    }
    return result
}
```

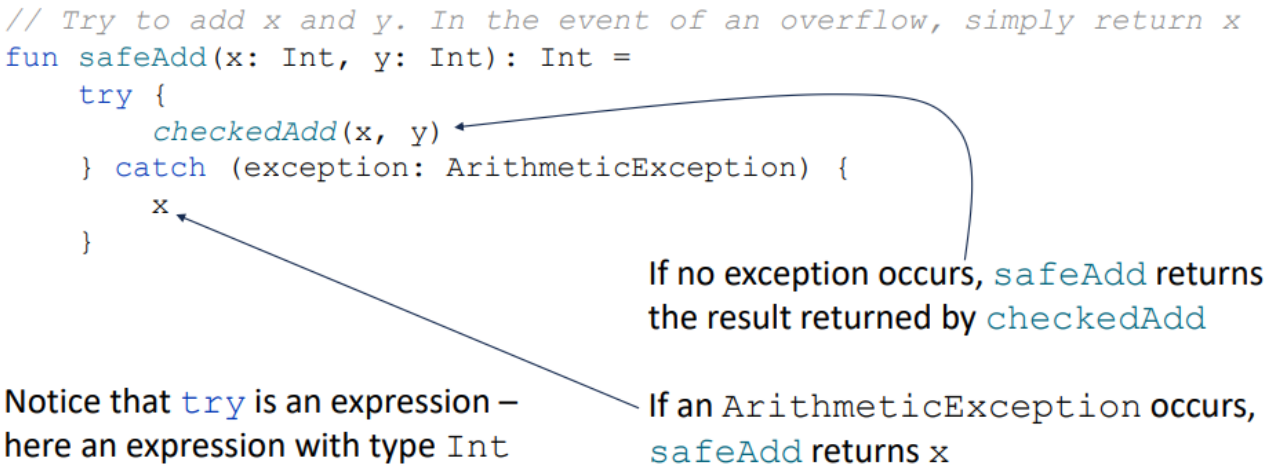
`throw` is an expression with type `Nothing`



```
fun checkedAdd(x: Int, y: Int): Int {
    val result = x + y
    if (x > 0 && y > 0 && result < 0) {
        val unused: Nothing =
            throw ArithmeticException("Integer overflow when adding $x and $y")
    } else if (x < 0 && y < 0 && result > 0) {
        val unused: Nothing =
            throw ArithmeticException("Integer underflow when adding $x and $y")
    }
    return result
}
```

This compiles, but such declarations are pointless; merely used here to illustrate the return type of throw

Catching an exception



```
// Try to add x and y. In the event of an overflow, simply return x
fun safeAdd(x: Int, y: Int): Int =
    try {
        checkedAdd(x, y)
    } catch (exception: ArithmeticException) {
        x
    }
```

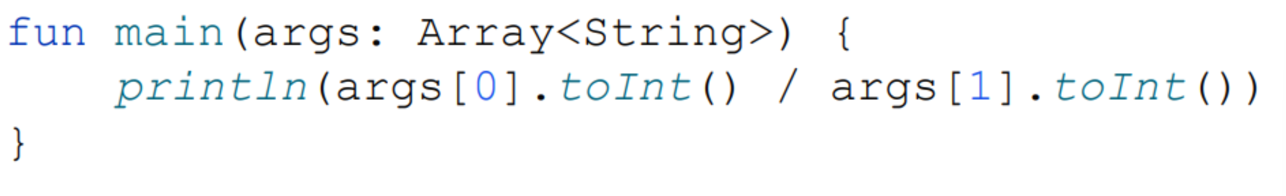
If no exception occurs, safeAdd returns the result returned by checkedAdd

Notice that try is an expression – here an expression with type Int

If an ArithmeticException occurs, safeAdd returns x

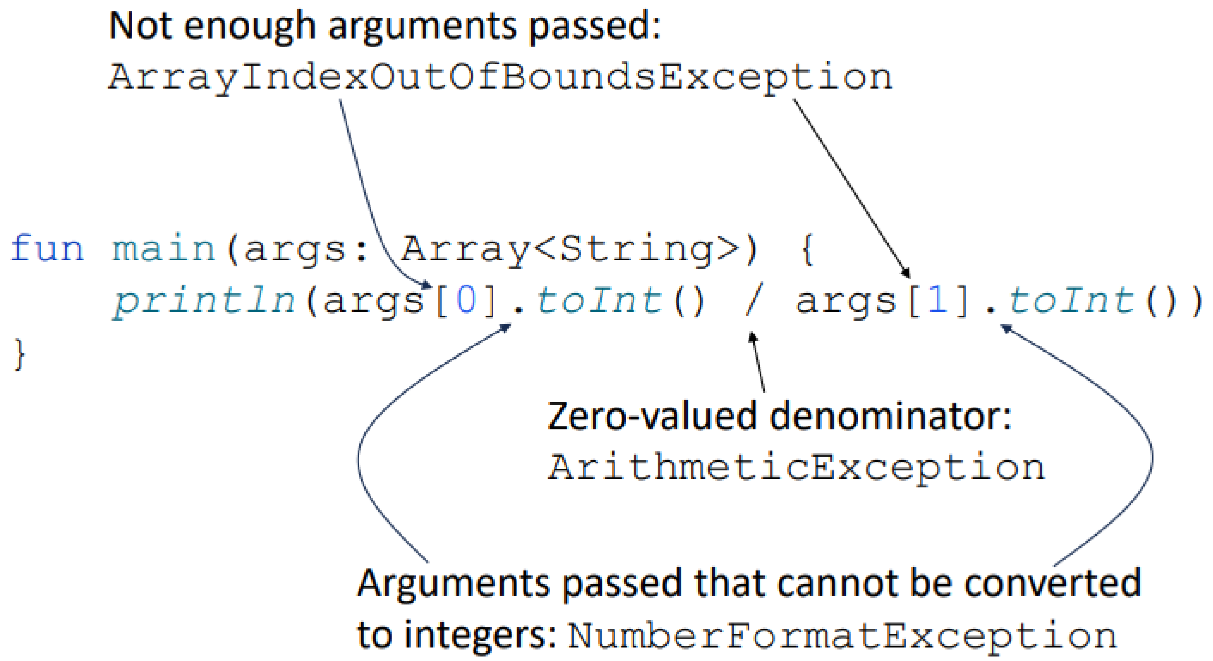
Example: a simple divider program

This simple program treats its arguments as integers and prints the result of dividing one by the other:



```
fun main(args: Array<String>) {
    println(args[0].toInt() / args[1].toInt())
}
```

What could go wrong?



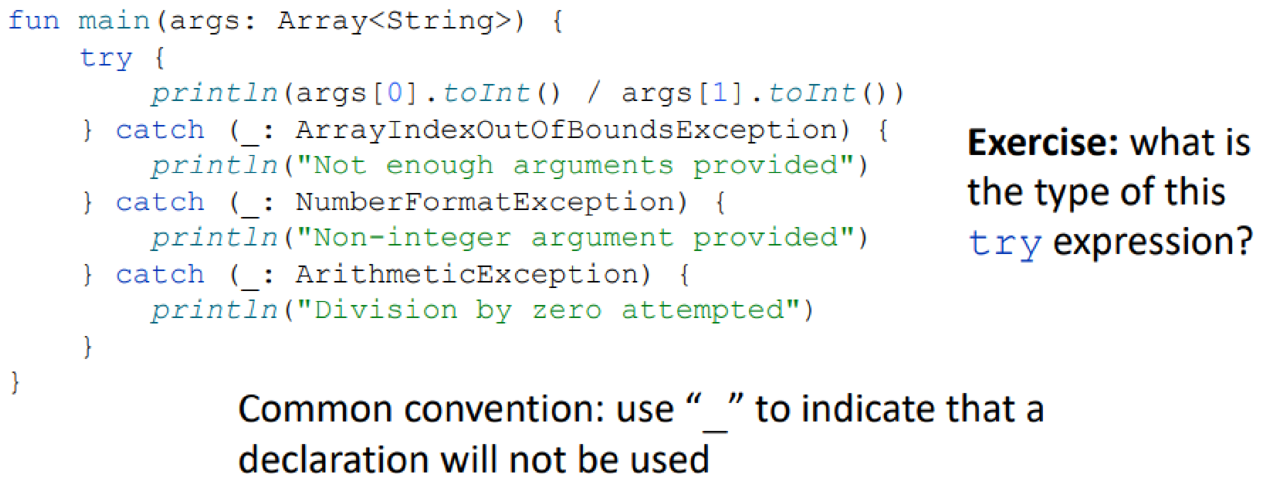
Not enough arguments passed: `ArrayIndexOutOfBoundsException`

```
fun main(args: Array<String>) {
    println(args[0].toInt() / args[1].toInt())
}
```

Zero-valued denominator: `ArithmeticException`

Arguments passed that cannot be converted to integers: `NumberFormatException`

Using multiple `catch` blocks to catch different kinds of exception

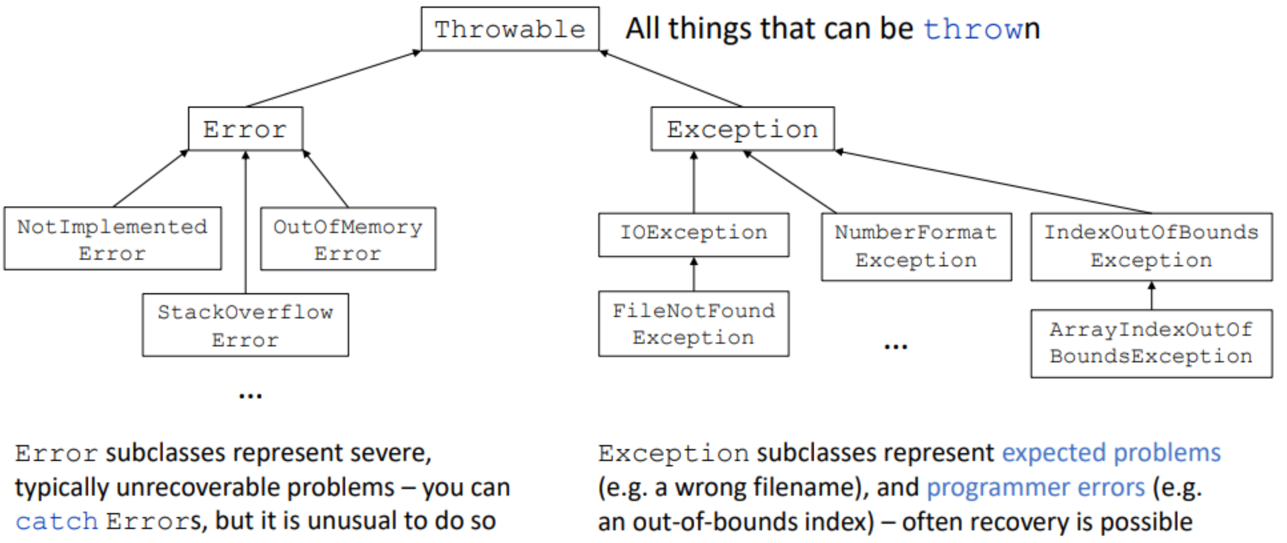


```
fun main(args: Array<String>) {
    try {
        println(args[0].toInt() / args[1].toInt())
    } catch (_: ArrayIndexOutOfBoundsException) {
        println("Not enough arguments provided")
    } catch (_: NumberFormatException) {
        println("Non-integer argument provided")
    } catch (_: ArithmeticException) {
        println("Division by zero attempted")
    }
}
```

Exercise: what is the type of this `try` expression?

Common convention: use "_" to indicate that a declaration will not be used

The `Throwable` class hierarchy

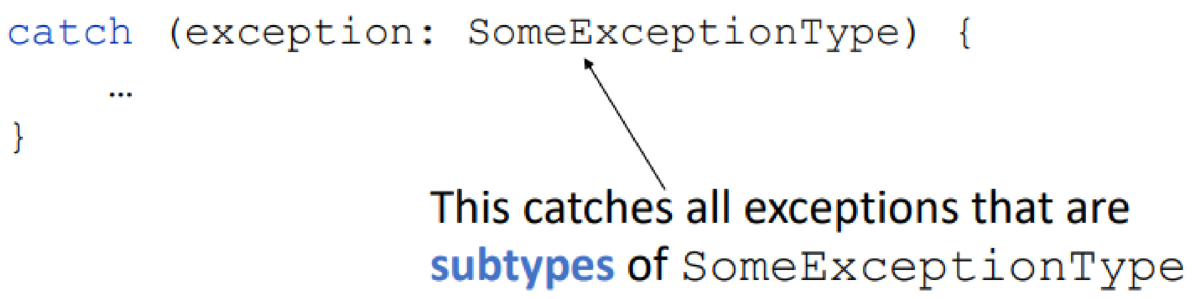


All things that can be thrown

Error subclasses represent severe, typically unrecoverable problems – you can catch Errors, but it is unusual to do so

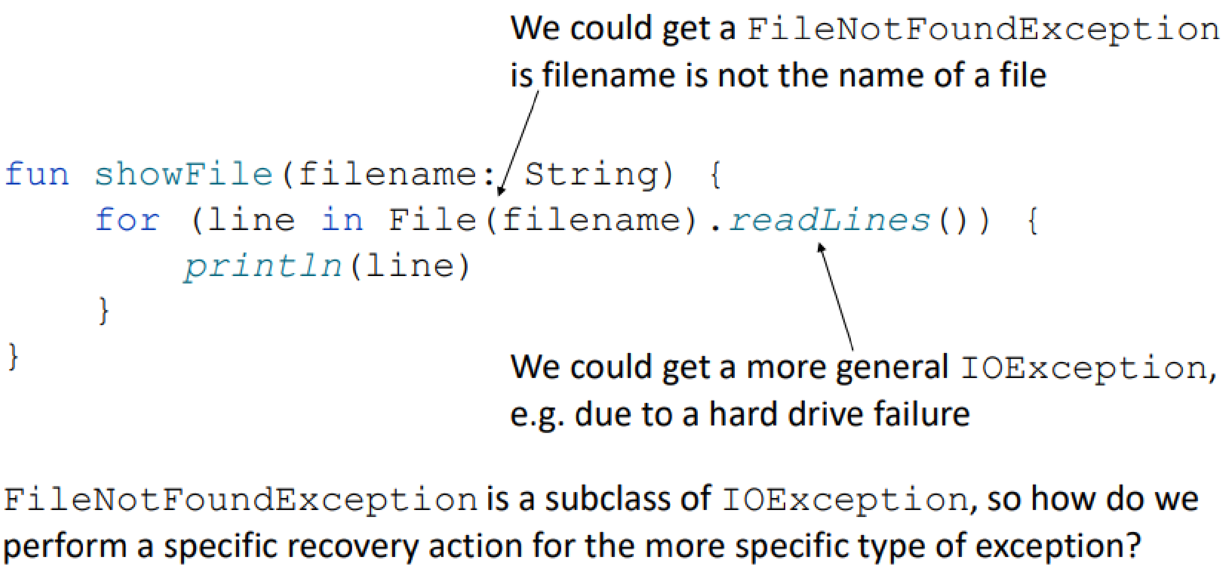
Exception subclasses represent expected problems (e.g. a wrong filename), and programmer errors (e.g. an out-of-bounds index) – often recovery is possible

Exceptions and subtyping



```
catch (exception: SomeExceptionType) {
    ...
}
```

This catches all exceptions that are subtypes of SomeExceptionType



We could get a FileNotFoundException is filename is not the name of a file

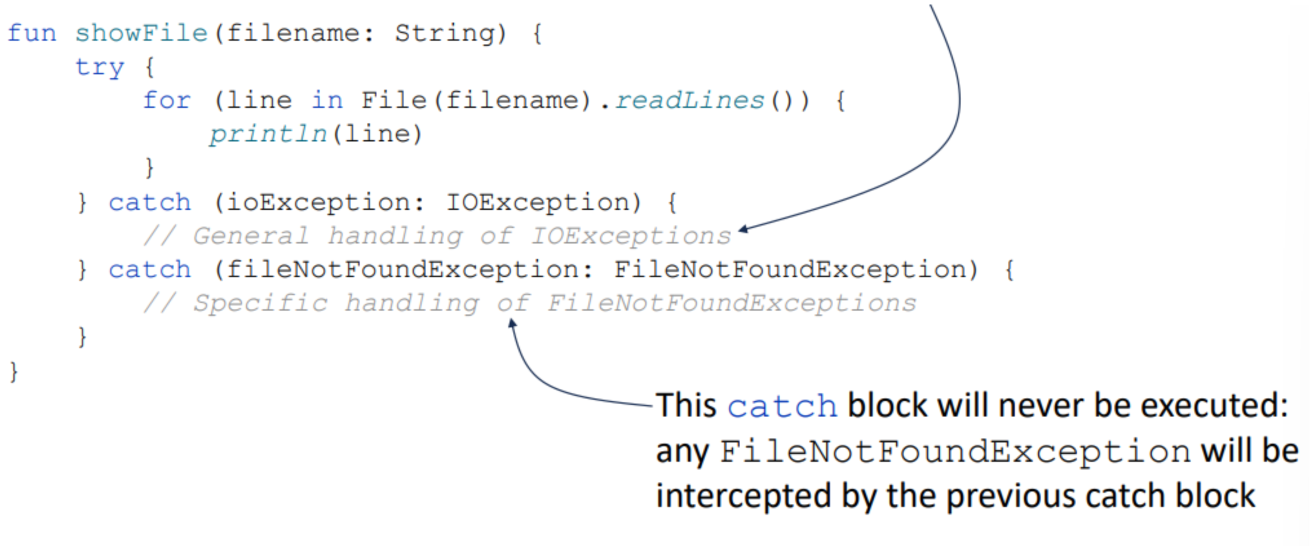
```
fun showFile(filename: String) {
    for (line in File(filename).readLines()) {
        println(line)
    }
}
```

We could get a more general IOException, e.g. due to a hard drive failure

FileNotFoundException is a subclass of IOException, so how do we perform a specific recovery action for the more specific type of exception?

Wrong solution

The first `catch` block catches any subtype of `IOException`, including `FileNotFoundException`

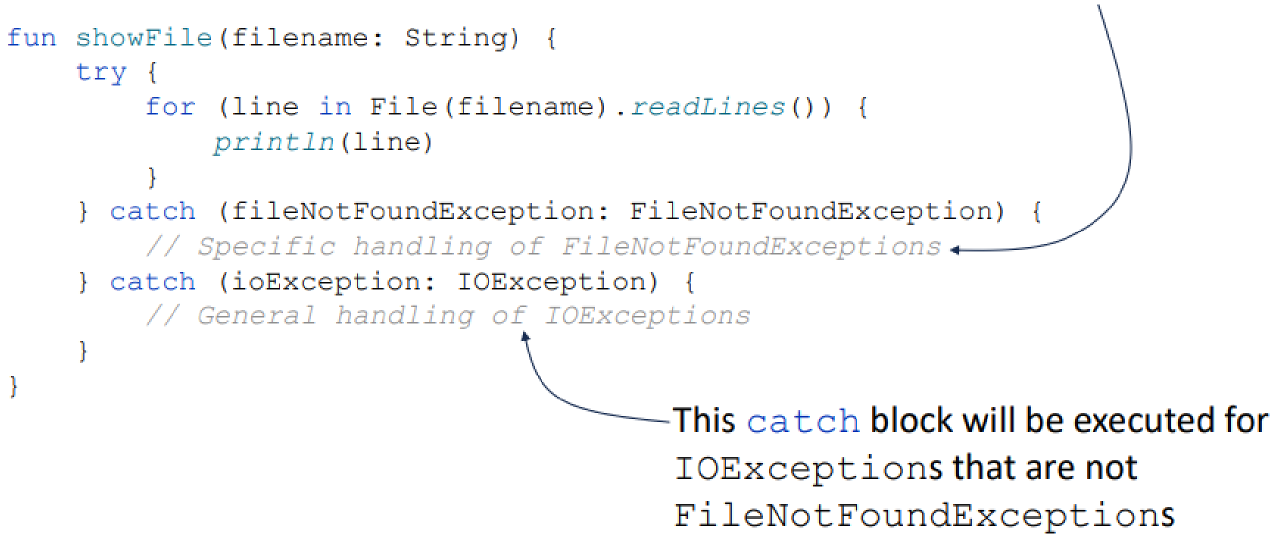


```kotlin
fun showFile(filename: String) {
    try {
        for (line in File(filename).readLines()) {
            println(line)
        }
    } catch (ioException: IOException) {
        // General handling of IOExceptions
    } catch (fileNotFoundException: FileNotFoundException) {
        // Specific handling of FileNotFoundExceptions
    }
}
```

This `catch` block will never be executed: any `FileNotFoundException` will be intercepted by the previous catch block

Correct solution

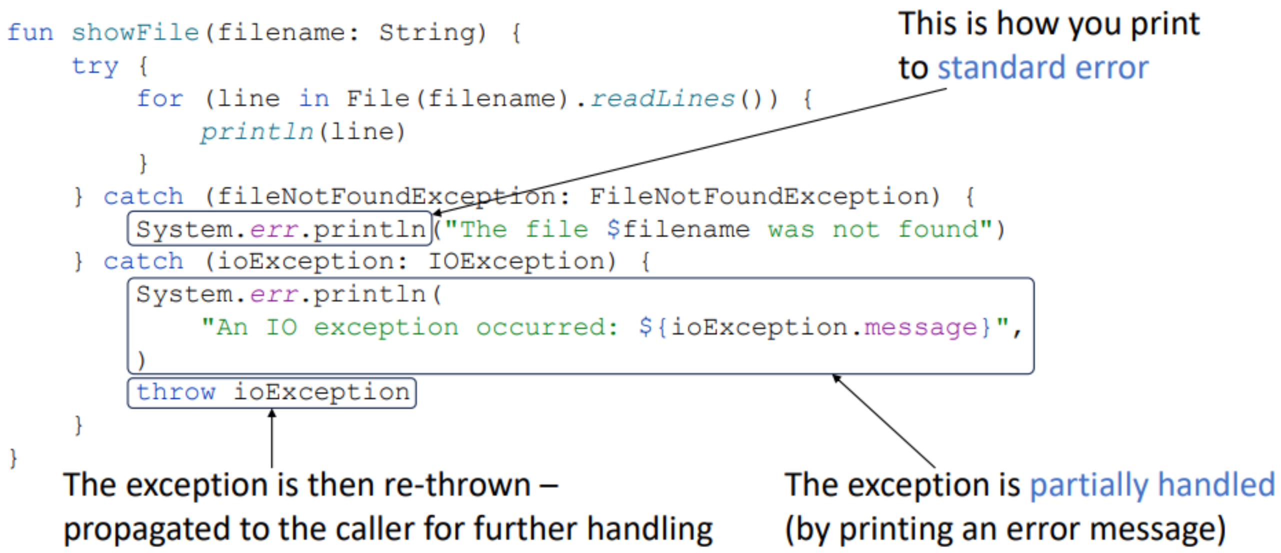
The first `catch` block catches any subtype of `FileNotFoundException`, but not more general `IOException` s



```kotlin
fun showFile(filename: String) {
    try {
        for (line in File(filename).readLines()) {
            println(line)
        }
    } catch (fileNotFoundException: FileNotFoundException) {
        // Specific handling of FileNotFoundExceptions
    } catch (ioException: IOException) {
        // General handling of IOExceptions
    }
}
```

This `catch` block will be executed for `IOExceptions` that are not `FileNotFoundExceptions`

One way these exceptions might be handled



```
fun showFile(filename: String) {
    try {
        for (line in File(filename).readLines()) {
            println(line)
        }
    } catch (fileNotFoundException: FileNotFoundException) {
        System.err.println("The file $filename was not found")
    } catch (ioException: IOException) {
        System.err.println(
            "An IO exception occurred: ${ioException.message}",
        )
        throw ioException
    }
}
```

This is how you print to standard error

The exception is then re-thrown – propagated to the caller for further handling

The exception is partially handled (by printing an error message)