Exceptions

When a runtime error occurs, the program terminates with an exception message:

```
>> val a = 3
>>> a / 0
java.lang.ArithmeticException: / by zero
>>> val s = "abc"
>>> s.toInt()
java.lang.NumberFormatException:
   For input string: "abc"
>>> val s = Array<Int>(100000000) { 0 }
java.lang.OutOfMemoryError: Java heap space
>>> java.io.File("test.txt").forEachLine
   { println(it) }
java.io.FileNotFoundException: test.txt
   (No such file or directory)
```

Exceptions and Errors

```
>>> var s: String? = null
>>> s!!.length
kotlin.KotlinNullPointerException

>>> val a = Array(100000000) { 0 }
java.lang.OutOfMemoryError: Java heap space
```

Errors indicate a serious failure, where continuing the program makes no sense.

An Exception indicates an unusual (exceptional) condition, such as a mistake in input data.

Handling exceptions

Some exceptions can be handled (or caught).

- NumberFormatException: print an error message to the user and request a new input.
- FileNotFoundException: try a different file name.

Old programming languages like C do not have exceptions, and all errors or unusual conditions need to be handled by error codes.

Exceptions make function calls cleaner:

```
val n = s.toInt()
```

In C, converting a string to an integer must return both an error code and the resulting integer.

Catching exceptions

If an exception occurs inside a `try` clause, execution continues with a matching exception handler in the `catch` clause:

```
val str = readString("Enter a number> ")
try {
  val x = str.toInt()
  println("You said: $x")
}
catch (e: NumberFormatException) {
  println("’$str’ is not a number")
}
```

Exceptions are caught even if they occur inside functions called in the `try` block.

catch1.kts

Catching across function calls

```
fun test(s: String): Int =
            (s.toDouble() * 100).toInt()

fun show(s: String) {
  try {
    println(test(s))
  }
  catch (e: NumberFormatException) {
    println("Incorrect input")
  }
}

>>> show("123.456")
12345
>>> show("123a456")
Incorrect input
```

catch2.kts

Handling exceptions

If an exception occurs, the normal flow of control is interrupted. Execution continues in the innermost catch block with a matching exception handler.

```
fun f(n: Int) =  g(n)

fun g(n: Int) {
  val m = 100 / n
  println("The result is $m")
}
try {
  f(n)
}
catch (e: ArithmeticException) {
  println("I can't handle this value!")
}
```

except1.kts

Throwing exceptions

When we detect an error in the input data, we can throw an exception ourselves:

```
  if (n < 0)
    throw IllegalArgumentException()
```

Exceptions are often used to detect errors in the input data.

We can catch the exception at a suitable place in the program and print an error message, or handle the problem in some other way.

except2.kts
except3.kts

Assertions

Exceptions are used to detect errors in input data.
Assertions are used to detect errors in your program.

The statement:

```
  assert(condition)
```

throws an `AssertionError` if condition is false.

```
  ... code A computing string s ...
  // if A is correct, then s is not empty
  assert(s.isNotEmpty())
  ... code B (using s) ...
```

The assertion protects code B from errors in code A.

Without the assertion, an error in A could cause a strange problem in B. Debugging could be difficult.

require

require is a special form of assertion, that throws an IllegalArgumentException. It is used to protect a function from being used with incorrect arguments.

```
fun factorial(n: Int): Long {
  require(n >= 2)
  assert(false)
  var result = 1L
  for (i in 1 .. n)
    result *= i
  return result
}
```

Again, require makes debugging easier. We do not need to search for a bug in factorial when the problem is in the code calling factorial.