Error Handling

Effective Programming in Scala

What Can Go Wrong?

- errors in input (user can fix),
- programming errors (programmer can fix),
- hardware failure, external systems—like a database—not working, and so on (most likely, someone else's problem).

What Can We Do?

- Abort the program,
- Report the errors to the user or the programmer,
- Retry, implement some form of resilience.

What Should We Report?

To the user, we want to give informative error messages, so they can fix bad input.

To the programmer, we want to give detailed information that relates to the source code, to aid with debugging.

How Can a Program Be Interrupted?

When something unexpected happens, the execution flow of the program can be interrupted.

The low-level language mechanism that is involved is called **exceptions**.

Exceptions can be **thrown** at any point of the execution of the program. They interrupt the execution flow unless they are **caught** by a surrounding exception handler.

Throwing an Exception

In Scala, you can throw an exception by using a **throw expression**:

```
def attemptSomething(): Unit =
  println("So far, so good")
  println("Still there")
  throw RuntimeException("We can't continue")
  println("You will never see this")
```

Note: unlike Java, there is no notion of "checked" exception in Scala. Any method can throw any exception at any time without having to declare it in its signature.

Throwing an Exception (2)

Calling this method prints the following to the standard output:

```
So far, so good
Still there
```

There is a default exception handler that stops the program after printing the **stack trace** to the standard error:

```
Exception in thread "main" java.lang.RuntimeException: We can't continue
        at exceptions$package$.attemptSomething(exceptions.scala:6)
        at exceptions$package$.run(exceptions.scala:10)
        at run.main(exceptions.scala:9)
```

Stack Traces

```
Exception in thread "main" java.lang.RuntimeException: We can't continue
        at exceptions$package$.attemptSomething(exceptions.scala:6)
        at exceptions$package$.run(exceptions.scala:10)
        at run.main(exceptions.scala:9)
```

Stack traces show the call chain, starting from the position where the error was thrown.

We can see the name of the method in which the exception occurred (here, `attemptSomething`), and the source file that defines this method and the number of the line where the exception occurred (here, `exceptions.scala`, line 6).

Note that the JVM prefixes the method name with its class name. This can lead to synthetic names such as `exceptions$package$` for top-level definitions or object definitions.

Regain Control of Program Execution

Exceptions can be caught with a handler:

```
@main def run(): Unit =
  try
    attemptSomething()
  catch
    case exn: RuntimeException =>
      System.err.println(s"Something went wrong: $exn")
      println("Stopping the program")
```

Regain Control of Program Execution (2)

Now the program prints the following on the standard output:

```
So far, so good
Still there
Stopping the program
```

And the following on the standard error:

```
Something went wrong: java.lang.RuntimeException: We can't continue
```

Catching Exceptions

The syntax for catching exceptions is similar to pattern matching:

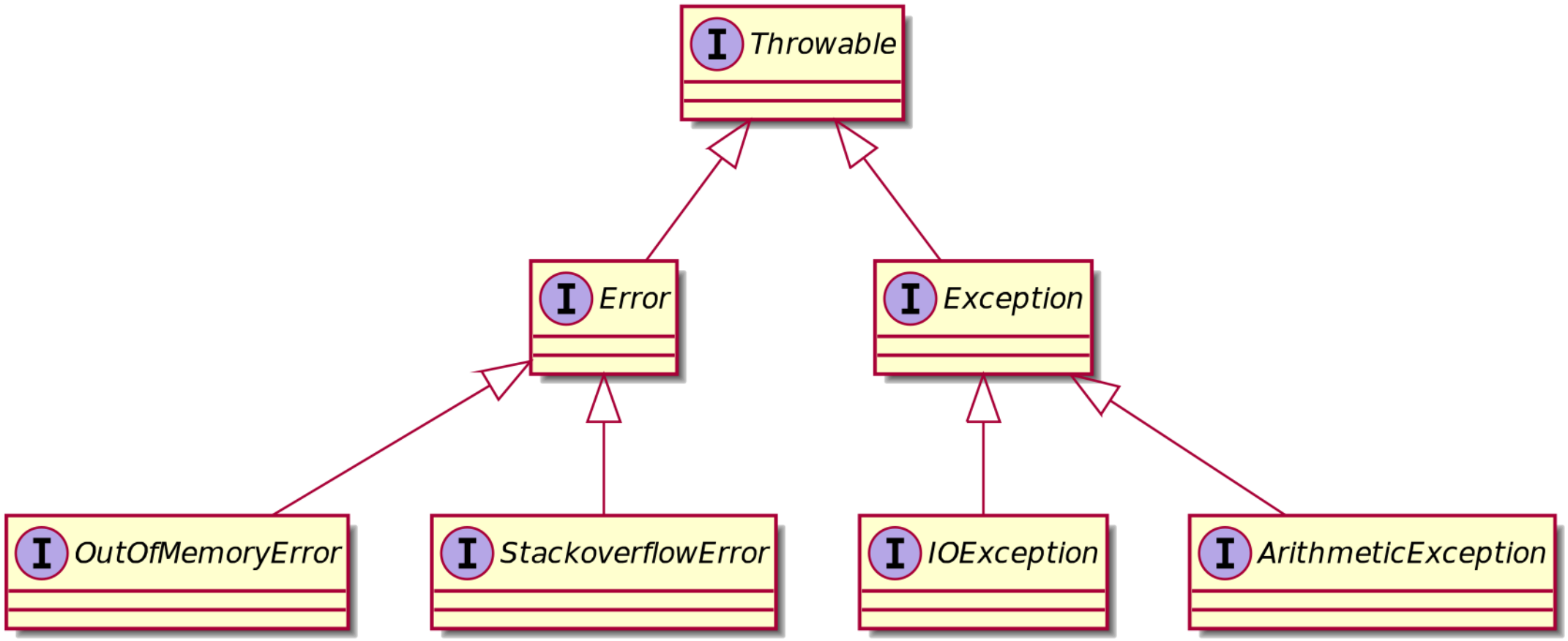
```
try
  attemptSomething()
catch
  case exn: ArithmeticException =>
    println("An arithmetic exception occurred.")
  case exn: RuntimeException =>
    println("Something unexpected happened.")
```

Exception classes are defined in the standard library and form a type hierarchy.

Each pattern of the `catch` clause checks whether the exception that happened is of some specific type or not.

`catch` clauses don't have to be exhaustive.

Throwable Hierarchy



Fatal Exceptions

There are some types of "throwables" that your programs should never try to catch, such as `OutOfMemoryError`. These are known as fatal exceptions.

In general, you should never try to catch "all the throwables":

```
catch
  case anyThrowable => // Don't do that
```

To catch all throwables except those that are fatal, Scala provides a `NonFatal` extractor.

```
import scala.util.control.NonFatal
```

And then:

```
catch
  case NonFatal(throwable) => // OK
```

try/catch/finally

A `try/catch` block is an expression that returns a value:

```
val stream = getClass.getResourceAsStream("data.txt")
val data =
  try
    parseIntData(stream)
  catch
    case NonFatal(exn) => -1
  finally
    stream.close()
```

`try/catch` expressions can be followed by a `finally` clause that is executed both if no exceptions were thrown or if an exception was thrown.

Summary

Exceptions are a low-level mechanism for dealing with unexpected problems during the program execution.

Throwing an exception interrupts the program execution unless the exception is caught by a handler.