EPFL

Error Handling With `Try`

Effective Programming in Scala

When Should We Catch Exceptions?

We have seen that running a program may be interrupted in case something unexpected happens and the program throws an exception.

What should we do about that?

When Should We Catch Exceptions?

We have seen that running a program may be interrupted in case something unexpected happens and the program throws an exception.

What should we do about that?

In some cases, aborting the program execution is what we want, so there is no need to handle exceptions.

However, in most cases, we don't want to abort the program execution:

- Servers need to return a response,
- Long-running programs need to keep running,
- etc.

Defensive Programming

In case we need to handle exceptions, should we add `try`/`catch` handlers around each expression?

Defensive Programming

In case we need to handle exceptions, should we add `try`/`catch` handlers around each expression?

This “defensive” way of programming would be very inconvenient.

One way to address this problem is to explicitly model the fact that (parts of) our program can fail.

This lesson and the next ones will introduce different techniques to achieve this.

Explicit Modeling of Failures

- If exceptions are used as a last resort only, there is no need to be defensive, unless we call an API that we know uses exceptions (e.g., this is a common practice in Java, or with some low-level APIs),
- Just define an exception handler at the beginning of your program, or use the default exception handler provided by the runtime,
- Explicitly indicate in the result type of a method that it can either succeed or fail.

Explicit Failures With Try

You can explicitly indicate that a method may fail by returning a value of type Try:

```
import scala.util.Try

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

Note that we also wrap the implementation of attemptSomething() into the Try constructor: `Try { ... }`.

Explicit Errors With `Try` (2)

The type `Try[A]` informs users that the program may fail.

A value of type `Try[A]` can either be a `Success[A]`, or a `Failure`.

If an exception is thrown within a `Try { ... }` block, we get a `Failure`, otherwise we get a `Success` containing the result of evaluating the block.

Explicit Errors With Try (3)

At the use site, one way to handle errors is by calling the recover method:

```
@main def run(): Unit =
  attemptSomething()   //: Try[Unit]
    .recover {
      case exn: RuntimeException =>
        System.err.println(s"Something went wrong: $exn")
        println("Stopping the program")
    }
```

Explicit Errors With `Try` (3)

At the use site, one way to handle errors is by calling the `recover` method:

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

So, we use `recover` like we would use `catch` in a `try` expression.

Aside: Partial Functions

Partial functions are functions that may not be defined on all their domain type.

For instance, a `PartialFunction[Int, String]` is a function that may not be defined for some `Int` values.

Before applying a partial function to a value, you should check that it is defined for this value by calling `isDefinedAt`:

```
if somePartialFunction.isDefinedAt(someValue) then
  somePartialFunction(someValue)
```

Aside: Syntax for Partial Functions

```
val handler: PartialFunction[Throwable, Unit] =
  case exn: RuntimeException => println("An exception was thrown")
```

Here, `handler` is defined only on `Throwable` values that are also subtypes of `RuntimeException`.

This is equivalent to the following definition:

```
val handler: PartialFunction[Throwable, Unit] =
  new PartialFunction[Throwable, Unit]:
    def apply(t: Throwable): Unit = t match
      case exn: RuntimeException => println("An exception was thrown")
    def isDefinedAt(t: Throwable): Boolean = t match
      case exn: RuntimeException => true
      case _                     => false
```

Pattern Matching on Try

In the previous example, the return type of attemptSomething was Try[Unit], which means that this program may fail but doesn't return a meaningful value (Unit).

Alternatively, consider a program

```
import java.time.LocalDate
def parseDate(str: String): Try[LocalDate]
```

which may fail to return a LocalDate. You can use it as follows:

```
import scala.util.{ Failure, Success }

parseDate("2020-02-02") match
  case Success(date)      => println("Successfuly parsed date")
  case Failure(throwable) => println("Failed to parse date")
```

Discussion: Explicit Modeling of Failures

Returning a value of type `Try[A]`, as opposed to returning a value of type `A` is a double edged sword:

- On the one hand, callers of your program won't be surprised by a possible failure,
- On the other hand, they will have to deal with failures (they can't just ignore them and let the program be aborted).

Explicit modeling of failures may not always be relevant. It is probably a good choice, though, if the failure is likely to happen (e.g., parsing data, opening files, performing remote calls).

A typical use case is to wrap a Java API that models failures by throwing exceptions.

Summary

`Try` makes it explicit that a computation may fail and lets you manipulate successful results or recover from exceptions.

A value of type `Try` can be either a `Success` or a `Failure`.

It is common practice to use `Try` blocks to wrap calls to APIs that model failures by throwing exceptions.