# Day 2

# Options

# The Billion-Dollar Mistake

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] But I couldn't resist the temptation to put in a null reference, simply because it was too easy to implement."

(Tony Hoare)

```
val string: String = null
println(string.length)
```

function calls on null objects result in NPEs and app crashes

```
Exception in thread "main" java.lang.NullPointerException
    at org.rtjvm.fundb.Main$.delayedEndpoint$org$rtjvm$Main$1(Main.scala:5)
    at org.rtjvm.fundb.Main$delayedInit$body.apply(Main.scala:3)
    …
```

```
val string: String = null
if (string != null) {
  println(string.length)
}
```

working with null values leads to spaghetti code

# Takeawyas

Use Options to stay away from the monster named Null:

- avoid runtime crashes due to NPEs
- avoid an endless amount of null-related assertions

A functional way of dealing with absence

- map, flatMap, filter
- orElse
- others: fold, collect, toList

If you design a method to return a (some type) but may return null, return an Option[that type] instead.

# Exceptions & Try

# Let's Try[T]

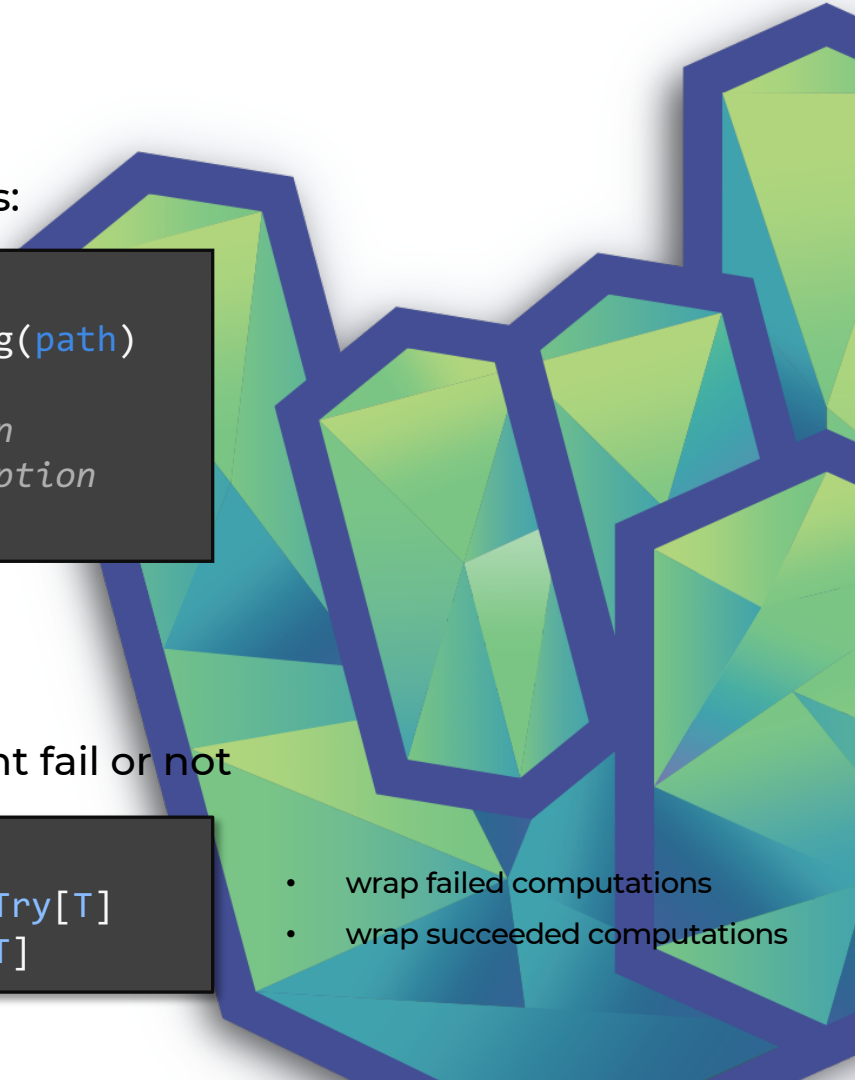
Exceptions are handled inside try-catch blocks:

```
try {
  val config: Map[String, String] = loadConfig(path)
} catch {
  case _: IOException => // handle IOException
  case _: Exception   => // handle other Exception
}
```

- multiple / nested try's make the code hard to follow
- we can't chain multiple operations prone to failure

A Try is a wrapper for a computation that might fail or not

```
sealed abstract class Try[+T]
case class Failure[+T](t: Throwable) extends Try[T]
case class Success[+T](value: T) extends Try[T]
```



- wrap failed computations
- wrap succeeded computations

# Takeaways

Use Try to handle exceptions gracefully:

- avoid runtime crashes due to uncaught exceptions
- avoid an endless amount of try-catches

A functional way of dealing with failure

- map, flatMap, filter
- orElse
- others: fold, collect, toList, conversion to Options

If you design a method to return a (some type) but may throw an exception, return a Try[that type] instead.

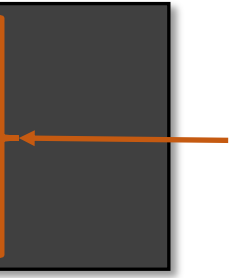
# Partial Functions

# Takeaways

```
trait PartialFunction[-A, +B] extends (A => B) {
  def apply(x: A): B
  def isDefinedAt(x: A): Boolean
}
```

How to use:

```
val simplePF: PartialFunction[Int, Int] = {
  case 1 => 42
  case 2 => 65
  case 3 => 999
}
```




Utilities:

- isDefinedAt
- lift
- orElse

Used on other types:

- map, collect on collections
- recover

# Curries & PAFs

# Takeaways

Curried functions

```
val superAdder: Int => Int => Int =
    x => y => x + y
```

Curried methods

```
def curriedAdded(x: Int)(y: Int) = x + y
```

Eta-expansion

- auto-transforming methods to functions

```
val add5 = curriedAdder(5) _
```

Underscores are powerful

- reducing function *arity*

```
val add7 = simpleAdder(7, _: Int)
```

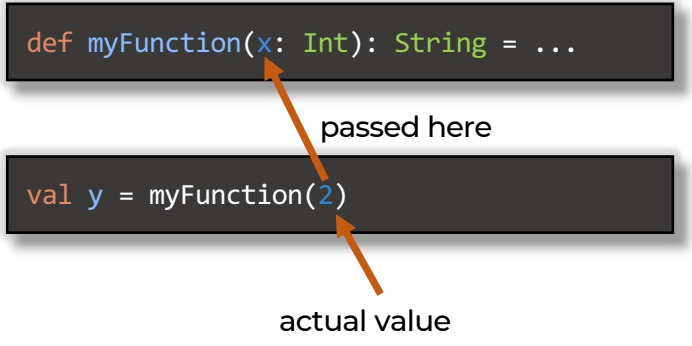
# Bonus:

# Call-By-Name

# Takeaways

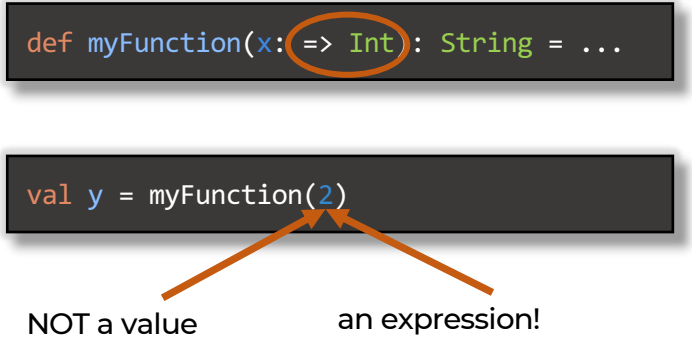
Call by value:

- value is computed before call
- same value used everywhere




Call by name:

- expression is passed literally
- expression is evaluated at every use within



# Futures & Promises

# Questions

- What's a thread?
- How can we communicate with a thread on the JVM?
- How do we synchronize threads on the JVM?
- Why do we need that?
- How are threads painful in real life?

# Back to the Future[T]

Future[T] is a computation which will finish at some point

```
import ExecutionContext.Implicits.global

val recipesFuture: Future[List[Recipe]] = Future {
  // some code that takes a long time to run
  jamieOliverDb.getAll("chicken")
}
```

a default ExecutionContext already implemented

ec is passed implicitly*

non-blocking processing

```
future.onComplete { case Success(recipes) => ... }
```

map, flatMap, filter, for-comprehensions

falling back

```
future.recover { case NotFoundException => ... }
```

blocking if need be

```
val txStatus = Await.result(transaction, 1 seconds)
```

# Promises

Futures are immutable, "read-only" objects.

Promises are "writable-once" containers over a future.

thread 1:

- creates an empty promise
- knows how to handle the result

```
val p = Promise[Int]()
val future = p.future

future.onComplete {
  case Success(value) => …
  case Failure(ex) => …
}
```

promise wraps future
future is "undefined"

triggers completion

thread 2:

- holds the promise
- fulfills or fails the promise

```
val result = doComputation()
p.success(result)
```

OR

```
p.failure(new BadException(...))
```

OR

```
p.complete(Try {...})
```

# Scala rocks