

# 3.0

## Containers and Exceptional Conditions

© 2019 Robin Hillyard



Northeastern  
University

# Containers

- Most real-world computing involves using **containers**:
  - Containers are wrappers around zero or more values
  - The most obvious and common container is *List[X]*, which represents zero or more elements of type *X*, in some order from *head* to *last*.
  - What exactly is *List[X]*? It's a trait defining methods such as *apply(x: Int): X* and *head: X*.
  - But is there a concrete type of List? Well, actually there are two:
    - *case object Nil* (i.e. the empty list) which is a sub-class of *List[Nothing]*;
    - *case class ::[X] (h: X, t: List[X])* which is a sub-class of *List[X]*.

# The functional programming way of dealing with abnormal situations

- In O-O, there are two constructs that were developed to allow for a deviation from the expected flow:
  - *null*
  - *Exception*.
- These are not FP-friendly. Why?
  - two reasons why *null* is not FP-friendly:
    - *null* is not an object and so does *not conform to type*;
    - if you try to dereference a *null* you get a NPE (null-pointer-exception)
  - *Exception* is not FP-friendly because throwing an exception is a *side-effect*!

# Dealing with *null*\* in O-O

- Where do nulls come from in O-O?
  - missing optional values (for example in a database).
  - Basically, nulls are for lazy programmers: but they are dangerous!!
- How do we typically deal with a *null*?

```
public class Phones {
```

```
    public static void main(String[] args) {
```

```
        Map<String,Long> phones = new HashMap<>();
        phones.put("Prof. Hillyard", 6173733972L);
        // ...
        String professor = "...";
        Long x = phones.get(professor);
        if (x != null) System.out.println(professor+"'s Phone number is: "+x);
        else System.out.println("No phone number for "+professor);
```

```
    }
```

```
}
```

Before we can do anything with *x*, we first must ensure that it is not *null*.

And yet, *null* is an expected value, not unusual, at all.

\* Tony Hoare (quicksort) called “null” his \$1B mistake. Methods that can yield null do not advertise the same (unless they are strictly using @Nullable annotation).

# Optional values—The Scala way

- If the return type of a method is optional, then why not make it *explicitly* optional?

“foreach” is a pretty basic method available on all container-types. Here it means do it once or not at all, as the case may be.



```
val phones = Map("Prof. Hillyard" -> 6173733972L)  
phones.get("Prof. Hillyard") foreach (x => println(s"found phone: $x"))
```

*found phone: 6173733972*

- In other words:
  - force the caller to deal with the possibility that there might not be a value returned...
  - ...*but* make it *easy* for the user to deal with that returned value.

# Dealing with exceptional conditions in O-O

- What if something goes wrong in O-O and we want to know what actually happened?
  - real life example: unable to get connection to remote database.
  - we catch/handle the exception (e.g. print stack trace) if we can, otherwise, we pass it up to the caller.
- What does this look like in practice?

```
Map<String,Long> phones = new HashMap<>();  
phones.put("Prof. Hillyard", 6173733972L);  
// ...  
String professor = "...";  
long x = phones.get(professor);
```

Unboxing to *long* (which the compiler has no problem with) causes a null-pointer-exception when the professor is not found.



# Exceptions—The Scala way

- Let's deal with these errors (exceptional conditions) in a calm, *referentially-transparent* way, with no loss of information:

```
def log(x: Double) = if (x > 0) Math.log(x) else throw new Exception("x must be positive")
def tryLog(x: Double) = Try(log(x))
val result = tryLog(-1)
result foreach (x => println(s"log value is $x"))
tryLog(math.E) foreach (x => println(s"log value is $x"))
log value is 1.0
```

“foreach” is used in the same way as in *Option*. There are ways to recover the actual cause of the *Failure* if we need to.



- In other words:
  - this looks just like the situation where we returned *Option[Long]*
  - the difference is that we have ways to recover the exception from the *Try* object (including “throwing” it if we really want to).

# Option

- Suppose we want to find an element in a list that satisfies a predicate?
  - What if there's no such element (the list might be empty, or the predicate simply never yields *true* for any element)?
  - In Scala, the *find* method on *List[X]* returns an *Option[X]*.
  - How should we implement *Option[X]*?
  - What should its API be?
    - *isDefined: Boolean*
    - *isEmpty: Boolean = !isDefined*
    - *get: X [will throw exception if empty]* 🗨️
    - *getOrElse[Y >: X](default: => Y): Y = if (isDefined) get else default*
    - *map[Y](f: X=>Y): Option[Y]*



## Option (2)

- So, we can define some methods we will want to call. Let's make them into a **trait**.

```
trait Option[X] {  
  def isDefined: Boolean  
  def get: X  
  ...  
}
```

- What have we got? A container based on a parametric type  $X$  and of which essentially there are two\* types: an empty container and a non-empty container that holds an  $X$  in it.

\* it's nearly always *two* types

# Option (3)

- Let's call our two containers *Some* and *None* and implement them as *case classes/objects* extending *Option*.

```
case class Some[X](x: X) extends Option[X] {  
  def isDefined: Boolean = true  
  def get: X = x  
}  
case object None extends Option[Nothing] {  
  def isDefined: Boolean = false  
  def get: Nothing = throw new NoSuchElementException("None.get")  
}
```

- Now, we can use pattern matching to figure what we've got:

```
List(1,2,3).find(_%2==0) match {  
  case Some(x) => println(x)  
  case None => println("no even number found")  
}
```

# Option (4)

- Using *Option* to handle objects returned from Java methods:

```
object Option {  
  import scala.language.implicitConversions  
  
  /** An implicit conversion that converts an option to an iterable value  
  */  
  implicit def option2Iterable[A](xo: Option[A]): Iterable[A] = xo.toList  
  
  /** An Option factory which creates Some(x) if the argument is not null,  
  * and None if it is null.  
  *  
  * @param x the value  
  * @return Some(value) if value != null, None if value == null  
  */  
  def apply[A](x: A): Option[A] = if (x == null) None else Some(x)  
}
```

- `val x = javaFunction(); // could return null`
- `val xo = Option(x)`

# Try

- Suppose we want to convert a *String* to an *Int* and know that it might throw an exception?
  - In Java we can wrap the expression in `try..catch..finally`
  - In Scala, we can actually do the same thing. But there's a much better, more functional way: *Try[X]*.
  - How should we implement *Try[X]*?
  - What should its API be?
    - *isSuccess: Boolean*
    - *isFailure: Boolean = !isSuccess*
    - *get: X [will throw exception if failure]*
    - *getOrElse[Y >: X](default: => Y): Y = if (isSuccess) get else default*
    - *map[Y](f: X=>Y): Try[Y]*
    - ...

## Try (2)

- So, we can define some methods we will want to call. Let's make them into a **trait**.

```
trait Try[X] {  
  def isSuccess: Boolean  
  def get: X  
  ...  
}
```

Sound familiar? We defined *Option* just like this.



- What have we got? A container with a parametric type *X* and of which essentially there are two types: a successful container with an *X* in it and a failure container that holds the exception.

# Try (3)

- Let's call our two containers *Success* and *Failure*. They will be case classes extending *Try*.

```
case class Success[X](x: X) extends Try[X] {  
  def isSuccess: Boolean = true  
  def get: X = x  
}  
case class Failure[X](e: Throwable) extends Try[X] {  
  def isSuccess: Boolean = false  
  def get: X = throw e  
}
```

- Now, we can use pattern matching to figure what we've got:

```
Try("a".toInt) match {  
  case Success(x) => println(x)  
  case Failure(e) => e.printStackTrace()  
}
```




Look familiar? We extracted the contents of *Option* rather like this. The big difference is that in the fail case, we actually have an exception

# Try (4)

- Using *Try* to handle methods which may throw an exception (usually but not always Java methods):

```
object Try {  
  /** Constructs a `Try` using the by-name parameter. This  
  * method will ensure any non-fatal exception is caught and a  
  * `Failure` object is returned.  
  */  
  def apply[T](r: => T): Try[T] =  
    try Success(r) catch {  
      case NonFatal(e) => Failure(e)  
    }  
}
```

Note that the parameter to the *Try.apply* method is call-by-name: this allows the actual exception to be caught *inside* the apply method.



- `val xy = Try("s".toInt)`

# Summary

- Scala defines two traits, each with two case classes/objects to represent the exceptional conditions corresponding to Java's *null* and *Exception*.
- As we shall see, these containers are very natural and easy to use and they are *referentially transparent*!
- Oh yes, and they are *monads*! (We'll talk about these later).