Uniwersytet Wrocławski

Scala in Practice

Billion-dollar mistake

I call it my ***billion-dollar mistake****. It was the* ***invention of the null reference*** *in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to* ***innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years****.*

Tony Hoare, QCon London, 2019

Null

```
scala> val someValue = null
someValue: Null = null

scala> val someValue = someJavaMethodWhichCouldReturnNull()
```

Null

```
scala> val someValue = null
someValue: Null = null
```

~~scala> val someValue = someJavaMethodWhichCouldReturnNull()~~

```
scala> val someValue = Option(someJavaMethodWhichCouldReturnNull())
```

Option[A] type

- sealed abstract class Option has two subtypes
  - Some[A](x : A)
  - None

```
sealed abstract class Option[+A]
final case object None extends Option[Nothing]
final case class Some[A](a: A) extends Option[A]
```

Option - apply

- *def apply[A](x: A): Option[A]*
  - *An Option factory which creates Some(x) if the argument is not null, and None if it is null.*

Option - Some

```
scala > val greeting = Some(" Hello ")
greeting: Some[String] = Some( Hello )

scala > val greeting2 = Option(" Hello ")
greeting2: Option[String] = Some( Hello )

scala> greeting == greeting2
res3: Boolean = true
```

Option - None

```
scala > val noGreeting = None
a: None.type = None

scala> val noGreeting2 = Option(null)
greeting2: Option[Null] = None

scala> noGreeting2 == None
res3: Boolean = true
```

Database example

```
case class User(
  id: Int,
  firstName: String,
  secondName: Option[String]
  lastName: String,
  age: Int)

object UserRepository {
  def findById(id: Int): Option[User] = . . .
  . . .
}
```

Java example

```
public int toInt(String in)
```

Scala example

```
def toInt(in: String): Option[Int]
```

Scala example

```
def toInt(in: String): Option[Int] = {
    try {
        Some(Integer.parseInt(in.trim))
    } catch {
        case e: NumberFormatException => None
    }
}

scala> val x = toInt("1")
x: Option[Int] = Some(1)

scala> val x = toInt("foo")
x: Option[Int] = None
```

Processing Options - ifs

```
def myPrint(someString: String) = {
    val optInt = toInt(someString)

    if (optInt.isDefined) println(optInt.get)
    else println(s"$someString is not a number")
}
```

Processing Options – PM

```
def myPrint(someString: String) =
  toInt(someString) match {
    case Some(i) => println(i)
    case None => println(s"$someString is not a number")
  }
```

Pattern matching

```
def myPrint(someString: String) =
  toInt(someString) match {
    case Some(0) => println("zero")
    case Some(i) => println(i)
    case _ => println(s"$someString is not a number")
  }
```

Build-in functions

```
someOption match {
  case None => None
  case Some(x) => Some(foo(x))
}

someOption.map(foo(_))

someOption match {
  case None => false
  case Some(_) => true
}

someOption.isDefined
```

Build-in functions

```
someOption match {
  case None => None
  case Some(x) => x
}

someOption.flatten

someOption match {
  case None => None
  case Some(x) => foo(x)
}

someOption.flatMap(foo(_))
```

Build-in functions

```
someOption match {
  case None => true
  case Some(_) => false
}

someOption.isEmpty

someOption match {
  case None => true
  case Some(x) => foo(x)
}

someOption.forall(foo(_))
```

Build-in functions example

```
scala > val bag = List("1", "2", "foo", "3", "bar")
bag: List[String] = List(1, 2, foo, 3, bar)

scala> bag.map(toInt)
res0: List[Option[Int]] = List(Some(1), Some(2), None, Some(3), None)

scala> bag.map(toInt).flatten
res1: List[Int] = List(1, 2, 3)
```

Case classes

- Auto-generates methods:
  - *apply*
  - *accessors* for all constructor parameters
  - *copy*
  - *equals* & *hashCode*
  - *toString*
  - *unapply*

Unapply method

```
scala> case class Person(name: String, age: Int)
scala> val alice = Person("Alice", 35)

scala> alice.match {
             case Person(n, r) => println(s"$n, $r")
           }
(Alice, 35)
```

Extractor object

```
scala> class Person(val name: String, val age: Int)

scala> object Person {
          . . .
          def unapply(arg: Person): Option[(String, Int)] = Some((arg.name, arg.age))
        }

scala> def personPrint(p: Person) =
          p match {
            case Person("Alice", _) => "Its Alice"
            case _ => "Its not Alice"
        }
```

Traits

```
trait SomeName {
  ...
}
```

Traits

```
scala > trait BaseSoundPlayer {
            def play: Unit
            def close: Unit
            def pause: Unit
            def stop: Unit
            def resume: Unit
            private . . .
            protected . . .
          }
scala > defined trait BaseSoundPlayer
```

Concrete fields

```
scala > trait BaseSoundPlayer {
            val type: String ="Mp3 Player"
            def getBasicPlayer: BasicPlayer = {...}
            def play: Unit
            def close: Unit
            def pause: Unit
            def stop: Unit
            def resume: Unit
            private ...
            protected ...
        }
```

Class/Object Inheritance

```
scala > class Mp3SoundPlayer extends BaseSoundPlayer {
            def play: Unit  = {...}
            def close: Unit = {...}
            def pause: Unit = {...}
            def stop: Unit  = {...}
            def resume: Unit = {...}
        }
```

Class/Object Inheritance

```
scala > class Mp3SoundPlayer extends BaseSoundPlayer {
            def play: Unit  = {}
            def close: Unit = {}
            def pause: Unit  = {}
            def stop: Unit  = {}
          }
scala > <console>:12: error: class Mp3SoundPlayer needs to be abstract,
since method resume in trait BaseSoundPlayer of type => () is not defined
        class Mp3SoundPlayer extends BaseSoundPlayer
```

Trait Class Inheritance

```
trait Mp3BaseSoundFilePlayer extends
BaseSoundFilePlayer {
  def getBasicPlayer:BasicPlayer
  def getBasicController:BasicController
  def setGain(volume: Double)
}
```

Override

```
scala > trait Tail { def length: Int}
scala > trait BigTail extends Tail { def length = 50 }
scala > trait BigTailWithColor extends BigTail {
               def length = 3
               def color: Int
             }
<console>:13: error: overriding method length in trait BigTail of type => Int;
 method length needs `override' modifier
           def length = 3
```

Override

```
scala > trait Tail { def length: Int}
scala > trait BigTail extends Tail { def length = 50 }
scala > trait BigTailWithColor extends BigTail {
            override def length = 3
            def color: Int
          }
defined trait BigTailWithColor
```

Def can’t override val

```
scala > trait Tail { def length: Int}
scala > trait ColoredTail extends Tail { val length = 50 ;  def color: Int}
defined trait ColoredTail

scala > trait Tail { val length: Int}
scala > trait ColoredTail extends Tail { def length: Int ;  def color: Int}
<console>:13: error: overriding value length in trait Tail of type Int;
 method length needs to be a stable, immutable value
          def length: Int
```

Multiple inheritance: Mixins

```
scala > abstract class Animal { def speak(): Unit }
scala > trait Tail { def length: Int}
scala > trait Legs { def move(): Unit }

scala > class Dog(tailLen: Int) extends Animal with Tail with Legs {
            def speak(): Unit = println("Bark")
            val length: Int = tailLen
            def move(): Unit = println("diagonal walk")
          }

scala > val dog = new Dog(20)
dog: Dog = Dog@27a7ef08
```

Traits limitations

```
class BaseClass
trait A extends BaseClass
class B
class C extends B with A

<console>:13: error: illegal inheritance; superclass B
 is not a subclass of the superclass BaseClass
 of the mixin trait A
       class C extends B with A
```

Traits are types

```
scala > val dog = new Dog(20)

scala> val dogTail: Tail = new Dog(20)
dogTail: Tail = Dog@64412d34

scala> dogTail.length
res0: Int = 20

scala> dogTail.speak
<console>:13: error: value speak is not a member of Tail
       dogTail.speak
```

Traits are types

```
scala > class Human(name: String) extends Legs {
          def move() = println("Walk")
        }

scala > def moveAll(w: List[Legs]) = w.foreach(_.move)

scala > val dog = new Dog(20)
scala > val human = new Human("Bob")

scala> moveAll(List(dog, human))
Diagonal walk
Walk
```

Self-type

```
scala > trait User {
            def username: String
          }

scala > trait FacebookWrapper {
            this: User =>
            def post(text: String) = println(s"$username: $text")
          }
```

Trick - limit the extends

```
scala > trait Tail { def length: Int}
scala > trait Tail {
              this: Animal =>
                def length: Int
            }

scala> class Human extends Tail
<console>:12: error: illegal inheritance;
 self-type Human does not conform to Tail's selftype Tail with Animal
       class Human extends Tail
```

Adding a trait dynamically

```
scala > class Dog(name: String)
scala > trait Barking { def speak(): Unit = println("woof-woof") }
scala > trait AngryBarking { def speak(): Unit = println("WOOF-WOOF!!") }

scala > val reksio = new Dog("Reksio") with Barking
reksio: Dog with Barking = Dog(Reksio)
scala > reksio.speak()
woof-woof

scala > val boxer = new Dog("Boxer") with AngryBarking
boxer: Dog with AngryBarking = Dog(Boxer)
scala > boxer.speak()
WOOF-WOOF!!
```

Adding a trait dynamically

```
scala > abstract class Dog(name: String)
scala > trait Barking { def speak(): Unit = println("woof-woof") }
scala > trait AngryBarking { def speak(): Unit = println("WOOF-WOOF!!") }

scala > val reksio = new Dog("Reksio") with Barking
reksio: Dog with Barking = Dog(Reksio)
scala > reksio.speak()
woof-woof

scala > val boxer = new Dog("Boxer") with AngryBarking
boxer: Dog with AngryBarking = Dog(Boxer)
scala > boxer.speak()
WOOF-WOOF!!
```

Uniwersytet Wrocławski

Adding a trait dynamically

```
scala > abstract class Dog(name: String)
scala > trait Barking { def speak(): Unit = println("woof-woof") }
scala > trait AngryBarking { def speak(): Unit = println("WOOF-WOOF!!") }

scala > val reksio = new Dog("Reksio") with Barking with AngryBarking
<console>:14: error: <$anon: Dog with Barking with AngryBarking> inherits conflicting
members:
  method speak in trait Barking of type ()Unit  and
  method speak in trait AngryBarking of type ()Unit
(Note: this can be resolved by declaring an override in <$anon: Dog with Barking with
AngryBarking>.)
       val dog1 = new Dog("Reksio") with  Barking with AngryBarking
```

Stackable Modifications

- Multiple traits can be mixed in to stack functionality
- Methods on super are called according to linear order of with clauses (right to left) [*new SomeClass extends A with B with … Z]*

Stackable Modifications

```
scala > abstract class Operations{
            def fun(x: Int): Int = x
          }

scala> trait Doubling extends Operations {
           abstract override def fun(x: Int): Int = { super.fun(2 * x) }
         }

scala > trait Incrementing extends Operations {
            abstract override def fun(x: Int): Int = { super.fun(x + 1) }
         }
```

Stackable Modifications

```
scala > val double = new Operations with Doubling
scala > double.fun(5)
10

scala > val decoratedOpt1 = new Operations with Doubling with Incrementing
scala > decoratedOpt1.fun(5)
12

scala > val decoratedOpt2 = new Operations with Incrementing with Doubling
scala > decoratedOpt2.fun(5)
11
```

Generic trait

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}

class IntIterator(to: Int) extends Iterator[Int] {
  private var current = 0
  override def hasNext: Boolean = current < to
  override def next(): Int =  { … }
}
```

Abstract class vs Trait

- Abstract class
  - Rich interfaces
  - You want to create a base class that requires constructor arguments
  - The code will be called from Java code

- Trait
  - Thin interfaces
  - Multiple traits can be mixed into class
  - Decorator pattern