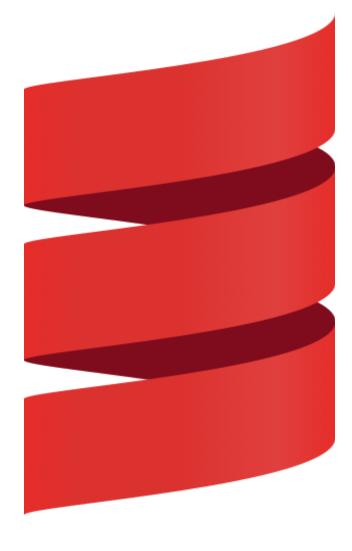
# Get rid of the boilerplate with Scala

by Michał Pawlik





# What's boilerplate?



#### What's boilerplate?

Something you have to write but you don't want to



#### Like what?



#### Like this



# How to identify boilerplate?



## **Boilerplate characteristics**

- repetitive
- unnecessary
- burdensome



# Is it a real problem?

What's your opinion?



#### Seems it is

2015 study on popular Java projects shows that 60% of methods can be uniquely identified by the occurrence of 4.6% of its tokens, making the remaining 95.4% boilerplate irrelevant to logic

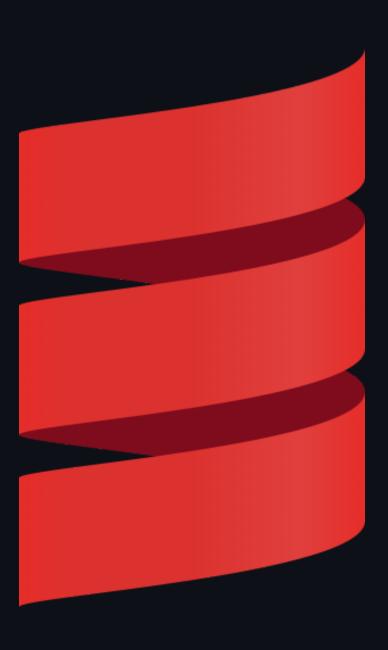


# Can we really avoid it?



# Let's give it a try!





# Get rid of boilerplate with Scala



#### What's scala anyway

- Functional and Object oriented
- concise, high-level language
- statically typed
- runs on JVM, JS and Native with LVM



## How does it help?

Let's see some examples,

Take the good old hello world for starters



#### Java



## **Python**

```
def main():
    print("hello world")

if __name__ == '__main__':
    main()
```



#### **Python**

```
def main():
    print("hello world")

# this might be omitted depending how Pythonic you feel
if __name__ == '__main__':
    main()
```



#### Scala

```
@main
def main() =
  println("hello world")
```



# Which one did you like the most?



## Let's move on to something more serious



#### Do you have pets at home?

Because we're gonna model one



#### Let's model a Pet

Nothing sophisticated, any pet, it has a name and an owner



#### Java

```
public class Pet {
        private String name;
        private String owner;
        public Pet(String name, String owner) {
                this.name = name;
                this.owner = owner;
        public String getName() {
                return name;
        public void setName(String name) {
                this.name = name;
        public String getOwner() {
                return owner;
        public void setOwner(String owner) {
                this.owner = owner;
```

#### **Python**

```
class Pet:

def __init__(self, name, owner):
    self.__name = name
    self.__owner = owner
```



#### Scala

case class Pet(owner: String, name: String)



# Maybe it's the problem with the types then?

Python seems to be doing well in those comparisons



#### Not exactly

```
from dataclasses import dataclass
@dataclass()
class Pet:
  name: str
  owner: str
doge = Pet("Doge", "Adam")
doge.name = 128
print(doge)
```

#### Output

```
Pet(name=128, owner='Adam')
```



#### Types are not a boilerplate

They make the compiler help you verify the correctness of the program



## Speaking of the type system

Scala can do a lot in the compile time



# **Refined types**



# Let's model part of the order



#### **Order Line**

Even with types, the correctness is not verified



#### Order Line with validation

```
case class UnsafeOrderLine(product: String, quantity: Int)
object UnsafeOrderLine {
  def safeApply(product: String, quantity: Int): UnsafeOrderLine =
    if (product.isEmpty())
      throw new RuntimeException("Product is empty")
    else if (quantity <= 0)</pre>
      throw new RuntimeException("Quantity lower than 1")
    else
      UnsafeOrderLine(product, quantity)
// Works fine!
UnsafeOrderLine.safeApply("123", 10)
// Throws runtime exception 📍
UnsafeOrderLine.safeApply("", 10)
```

#### Let's try with refined types

```
import eu.timepit.refined.auto._
import eu.timepit.refined.types.string._
import eu.timepit.refined.types.numeric._

case class OrderLine(product: NonEmptyString, quantity: PosInt)
OrderLine("123", 10) // Returns OrderLine
OrderLine("", 10) // Doesn't compile
```



#### Even the complete order model

```
import cats.data.NonEmptyList
import eu.timepit.refined.api.Refined
import eu.timepit.refined.auto._
import eu.timepit.refined.types.string._
import eu.timepit.refined.types.numeric._

case class OrderLine(product: NonEmptyString, quantity: PosInt)
case class Order(orderId: String Refined Uuid, lines: NonEmptyList[OrderLine])
```

- The compiler verifies the correctness
- No need to write code for validation
- Less code 
   ← Less tests
- Interoperability with serialization libs for JSON, XML, Databases etc.



#### What makes Scala concise

Let's see a few more techniques that make the code type safe and yet not bloated



# Type inference

To make the statically typed language feel like a dynamic one, it needs to be able to guess the types for you



# Type inference

```
val list = List(1, 2, 3)
// is an equivalent to
val list: List[Int] = List(1, 2, 3)
```

Notice how we didn't even need to do List[Int](1, 2, 3), the compiler has deduced it



# Type inference

```
val coordinates = Map(
   "Wrocław" -> (51.107883, 17.038538),
   "Kraków" -> (50.049683, 19.944544)
)
// is an equivalent to
val coordinates: Map[String, (Double, Double)] = Map(
   "Wrocław" -> (51.107883, 17.038538),
   "Kraków" -> (50.049683, 19.944544)
)
```

And so on with even more complex types



Do you still remember constructs like switch / case?



Imagine a flow control structure so powerful it can inspect the value, runtime type and even the internal structure of a type



The simplest example

```
def matchInt(x: Int) = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "other"
}
```



#### Decomposing objects



## One more exercise

Remember the Pet model?

It was too generic, let's see how we could model an extensible enum in Scala



## **Animals**

```
enum Animal(name: String):
   case Dog(name: String) extends Animal(name)
   case Burek extends Animal("Burek")
   case Cat(name: String) extends Animal(name)
   case Horse(name: String, weight: Double) extends Animal(name)
   case Snake(name: String, length: Double) extends Animal(name)
```

The enum as you know them, but each implementation can have it's own properties



# Let's play with it a bit

Write a function that takes a sequence of animals, and only returns:

- Cats whose name starts with an "A"
- Snakes longer than 1.5m



## The filter

Here comes the power of pattern matching

```
def filterAnimals(animals: Seq[Animal]) =
  animals.collect {
    case cat @ Cat(name) if name.startsWith("A") => cat
    case snake @ Snake(_, length) if length > 1.5 => snake
  }
```



## Let's test it

```
import Animal.*

@main
def main() =
  val testData = List(
    Dog("Doge"), Burek, Cat("A"), Cat("B"),
    Snake("Python", 2.0), Snake("Snek", 0.5)
  )
  println(filterAnimals(testData))
```

#### Output

```
List(Cat(A), Snake(Python, 2.0))
```



# One more case for boilerplate

Extending someone else's API



## One more case for boilerplate

Extending someone else's API

Say you want to be able to find odd numbers on any List[Int]



In this case List is a class that comes from library, but let's extend it

```
extension (x: List[Int])
def odds = x.filter(_ % 2 == 1)
```



```
extension (x: List[Int])
  def odds = x.filter(_ % 2 == 1)

@main
def main() =
  val testData = (1 to 10).toList
  println(testData.odds)
```

#### Output

```
List(1, 3, 5, 7, 9)
```



Notice how selectively we can extend the imported APIs. If we switch to List[String] the compiler will prevent us from making a mistake

```
extension (x: List[Int])
  def odds = x.filter(_ % 2 == 1)

@main
def main() =
  val testData = List("123", "456")
  println(testData.odds)
```



#### Compiler error:



### There's a lot more

- No more null s with Option
- Union types and match types
- Async with Future, IO or ZIO
- Typeclass derivation
- Type safe metaprogramming



# Try it for yourself



# Thank you!

