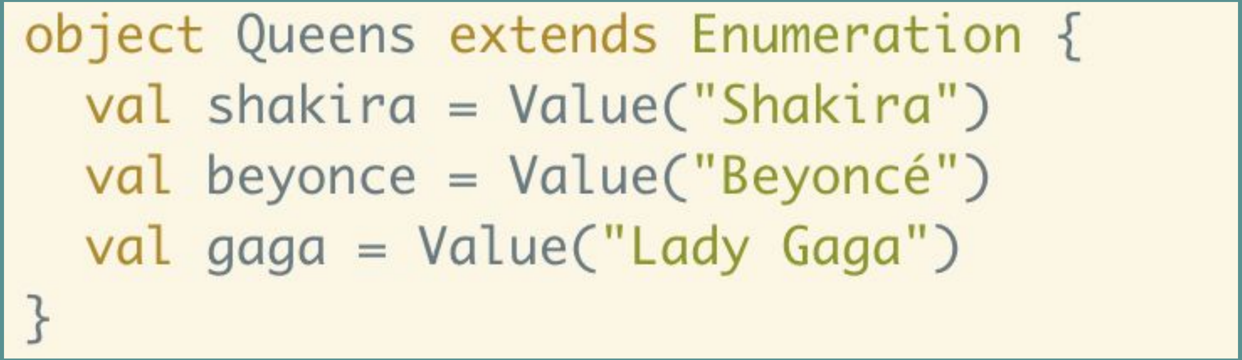
Enums vs Case Objects

Which should you use?

What are Enums?

Enum is an object that can be used to model a finite set of entities.

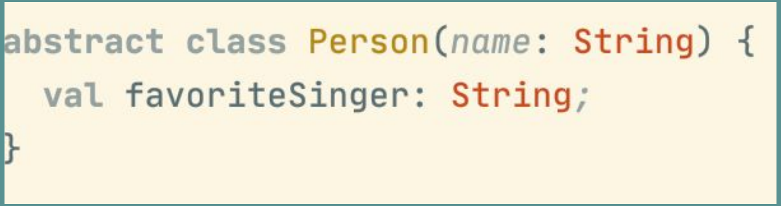
For example, for representing your favorite singers.



```
object Queens extends Enumeration {
  val shakira = Value("Shakira")
  val beyonce = Value("Beyoncé")
  val gaga = Value("Lady Gaga")
}
```

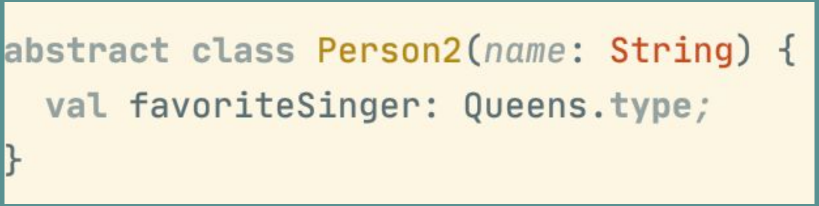
Why use Enums?

Enums give us stronger typing when we use primitive values (String, Int, etc...)
Especially in untyped languages -



```
abstract class Person(name: String) {
  val favoriteSinger: String;
}
```

VS

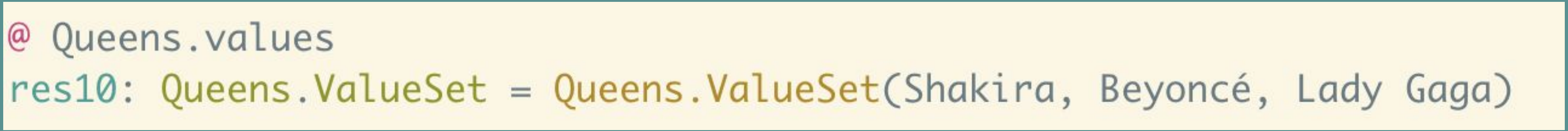


```
abstract class Person2(name: String) {
  val favoriteSinger: Queens.type;
}
```

What features do Enums have?

Enums have some great out-of-the-box features -

1. retrieving all possible values.

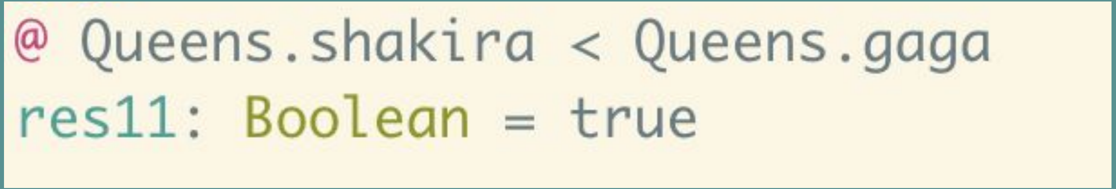


```
@ Queens.values
res10: Queens.ValueSet = Queens.ValueSet(Shakira, Beyoncé, Lady Gaga)
```

What features do Enums have?

Enums have some great out-of-the-box features -

2. Ordering

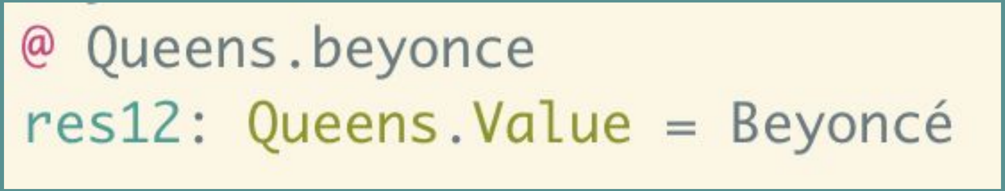


```
@ Queens.shakira < Queens.gaga
res11: Boolean = true
```

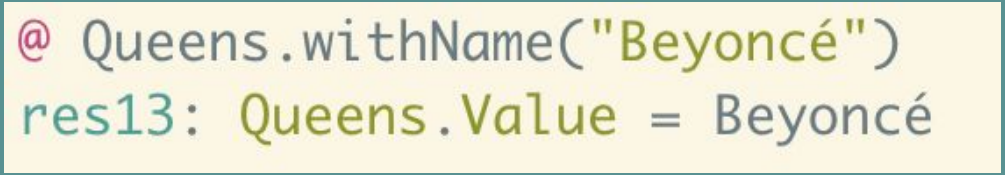
What features do Enums have?

Enums have some great out-of-the-box features -

3. Built in serializing and deserializing



```
@ Queens.beyonce
res12: Queens.Value = Beyoncé
```



```
@ Queens.withName("Beyoncé")
res13: Queens.Value = Beyoncé
```

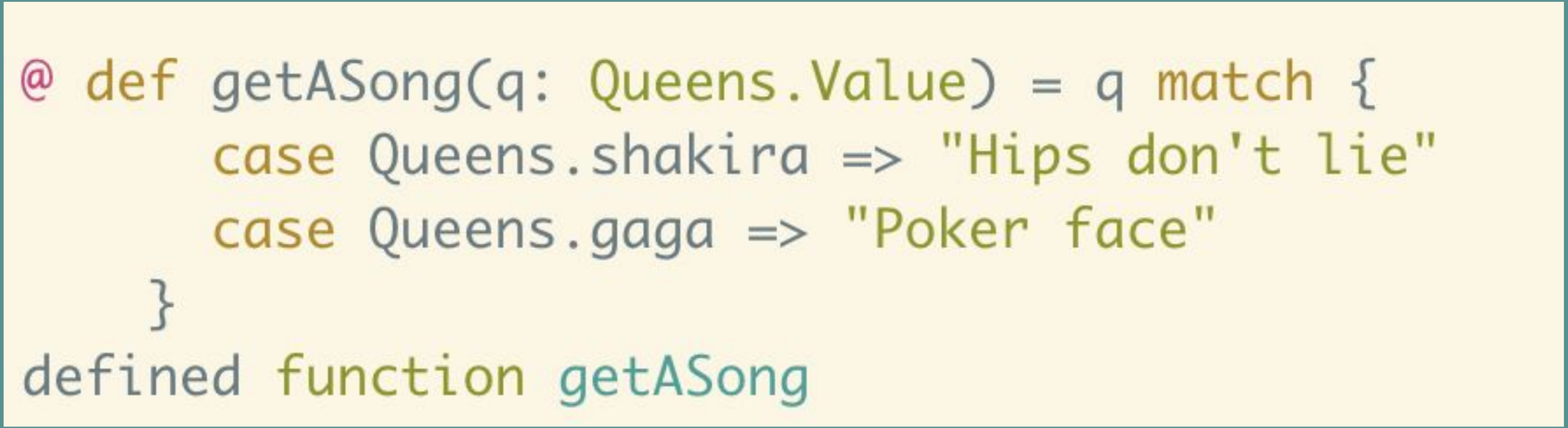
What's the problem with Enums?

The basic Enum type in Scala does not provide exhaustive matching check during compilation.

That means that enums allows us to write code that compiles and then fails at runtime!

What’s the problem with Enums?

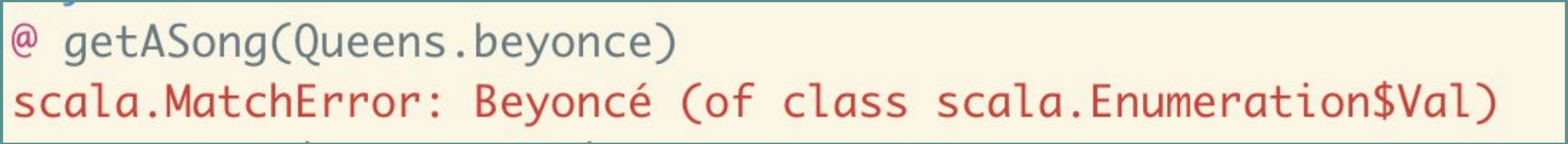
For example, this function would fly at compilation time, but then crash and burn at runtime -



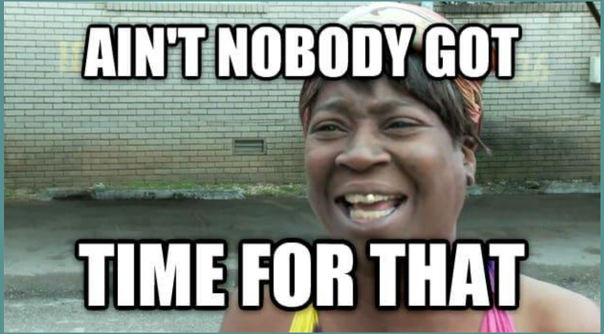
```
@ def getASong(q: Queens.Value) = q match {
       case Queens.shakira => "Hips don't lie"
       case Queens.gaga => "Poker face"
    }
defined function getASong
```

What's the problem with Enums?

For example, this function would fly at compilation time, but then crash and burn at runtime -

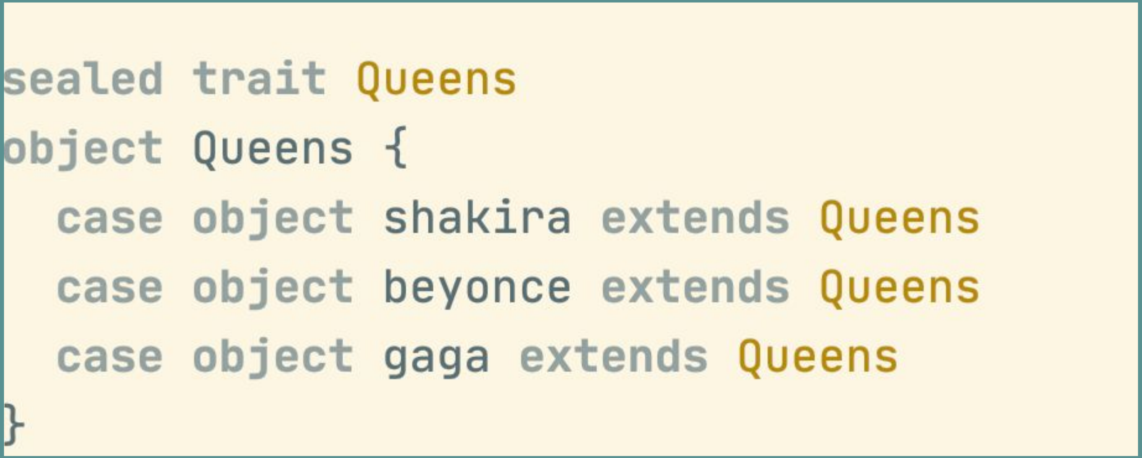


```
@ getASong(Queens.beyonce)
scala.MatchError: Beyoncé (of class scala.Enumeration$Val)
```



So, what's the solution?

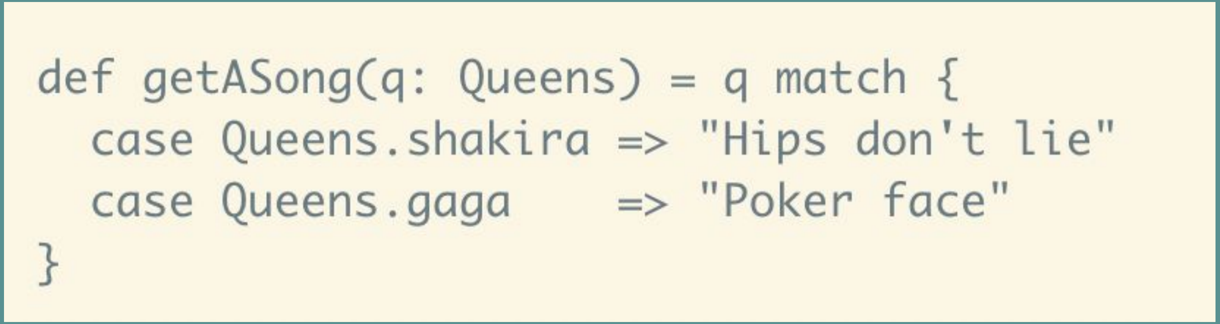
First, consider if you really need to use an enum?
If you don't need specific enum features, case objects can do the trick.



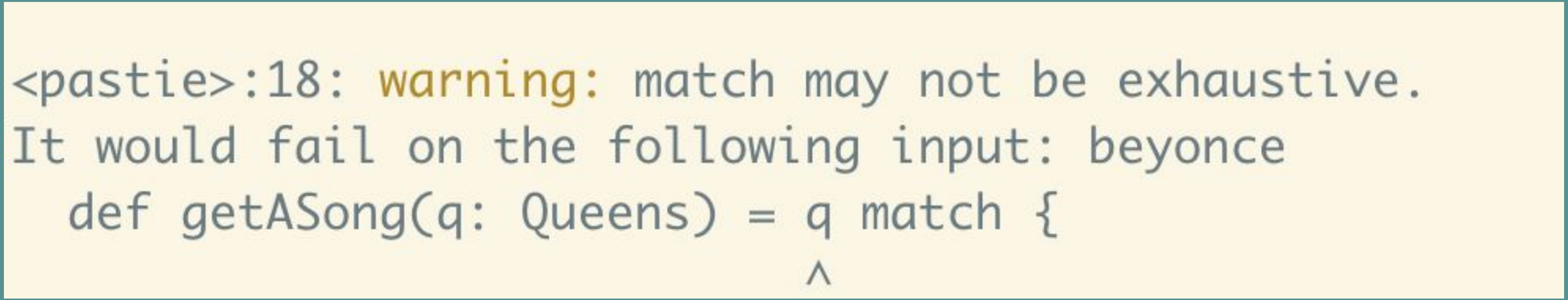
```scala
sealed trait Queens
object Queens {
  case object shakira extends Queens
  case object beyonce extends Queens
  case object gaga extends Queens
}
```

So, what's the solution?

Let's try that function again...



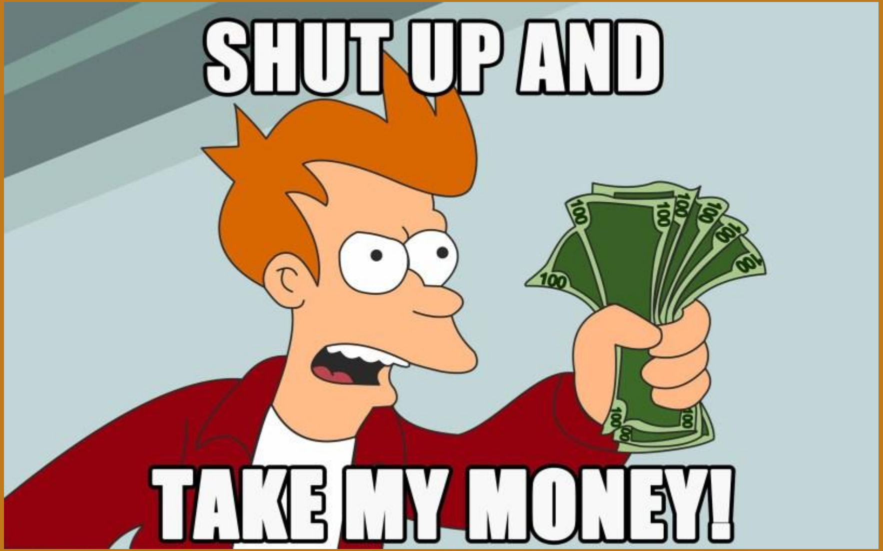
```scala
def getASong(q: Queens) = q match {
  case Queens.shakira => "Hips don't lie"
  case Queens.gaga    => "Poker face"
}
```



```
<pastie>:18: warning: match may not be exhaustive.
It would fail on the following input: beyonce
  def getASong(q: Queens) = q match {
                            ^
```

And what if I still need an Enum?

If you still want to use the Enum features, but without the drawbacks you can use enumeratum.



Enumeratum

It has -

1. Value retrieval
2. Ordering
3. Default serializing\deserializing
4. Exhaustiveness checks
5. Multiple fields (like regular case objects)

Example