noob-constraints

PDXScala • 7 May 2013

What's this about?

- Type Constraints
- Several Flavors
- Some are unusual
- Some are rarely explained
- "Enthusiastic Advanced Beginner" Topic

The Plan

- A Few Slides
- Coding Exercise
- Questions
- Shooting for ~30 mins

Scala Prerequisites

- Parametric polymorphism "generics"
- Implicit Values / Conversions
- Structural Types
- `git` and `sbt` *

* if you want to do the exercise

Review: Structural Types

- Quick Review
- Used in the exercise
- Occasionally Useful
- Beware Performance Cost

Structural Type

```
type Transactional = {
  def rollback(): Unit
  def commit(): Unit
}

def tx: Transactional =
  DriverManager.getConnection(...)
```

Assignable if structure conforms.

Bounded Types

- Widely available in OO languages
- Based on subtype polymorphism
- Very similar to what you can do with Java generics

Monomorphic

```
def id(a: Int): Int = a
```

id is only defined for Int.

Universal

```
def id[A](a: A): A = a
```

id is defined for *any* type.

Upper Bound

```
def flip[A <: Shape](a: A): A =
  a.reflect(math.Pi)
```

Upper Bound Syntax

flip is defined for any type
that is assignable to Shape.

Lower Bound

```
val ss: List[Shape] = ...

def prepend[A >: Shape](a:A):List[A] =
  a :: ss
```

Lower Bound Syntax

prepend is defined for any type
to which Shape can be assigned.

Evidence Constraints

- Carried by implicit values
- Must be constructed
  - By you
  - By the compiler, via the implicit resolution process

View Bounds

```
def area(a: Shape): Int =
  a.width * a.height

def area[A](a: A)(implicit ev: A => Shape): Int =
  a.width * a.height
```

flip is defined for any type that can be *converted* [implicitly] to Shape.

View Bounds

```
def area[A <% Shape](a: A): Int =
  a.width * a.height
```

View Bound Syntax

This syntax is *exactly* equivalent.

Context Bounds

```
trait Shape[A] {
  def reflect(a:A, θ: Double): A = ...
}

def flip[A](a: A)(implicit ev: Shape[A]): A =
  ev.reflect(a, math.Pi)
```

Defined for any type that has an associated instance of Shape[A].

Context Bounds

```
trait Shape[A] {
  def reflect(a:A, θ: Double): A = ...
}

def flip[A: Shape](a: A): A =
  implicitly[Shape[A]].reflect(a, math.Pi)
```

Context Bound Syntax

These are equivalent.

This is the *typeclass* pattern.

Equality

```
class Foo[A](a:A) {

  def toOption:Option[A] = Some(a)

  def inc(implicit ev: A =:= Int) =
    new Foo(a + 1)

}
```

Equality Evidence

inc can only be called if A is known to be Int *at compile time.*

Conformance

```
class Foo[A](a:A) {

  ...

  def bytes(a: A)(implicit ev: A <:< Serializable) =
    byteBlaster.writeObject(a)

}
```

Conformance Evidence

Can only be called if A is known
to be Serializable *at compile time.*

Bonus Trivia

- You can write your own constraints!
- `=:=` and `<:<` are library code
- They are almost identical
- Compiler knows nothing about them
- Defined in `Predef.scala`

Exercise!

- Work in pairs or teams if you like.
- Clone `github.com/tpolecat/noob`
- Run `sbt` and `~compile`
- Exercises in `Evidence.scala`
- Slides are there as well.

Cheat Sheet

```
{ def a: A }                    // Structural type
[A <: B]                        // Upper Bound
[A >: B]                        // Lower Bound
[A <% B]                        // View Bound
[A: B]                          // Context Bound
(implicit ev: A =:= B)          // Equality
(implicit ev: A <:< B)          // Conformance
```