Mastering the type system

Scala training, 2016
Outline

A little more about the type hierarchy
   Adding value classes

Type parameters and variance
   Type parameters
   Variance
   Bounded types

More about types
   Type members
   "Inner" types
   Type refinements
   Static duck typing
Recap

- most of our custom types extend `AnyVal`
- since Scala 2.10 we can define value types
Outline

A little more about the type hierarchy
  Adding value classes

Type parameters and variance
  Type parameters
  Variance
  Bounded types

More about types
  Type members
  "Inner" types
  Type refinements
  Static duck typing
Adding value classes

```scala
class MyInteger(val underlying: Int) extends AnyVal {
  def +(that: MyInteger): MyInteger =
    new MyInteger(this.underlying + that.underlying)
}
```

Value types:

- are wrappers over a single value
- extend AnyVal
- have only one class parameter
- the class parameter is a `val`
Outline

A little more about the type hierarchy
  Adding value classes

Type parameters and variance
  Type parameters
  Variance
  Bounded types

More about types
  Type members
  "Inner" types
  Type refinements
  Static duck typing
The Scala "generics"

```scala
// parameterized trait
trait Set[A] {  
  // parameterized method
  def map[B](f: A => B): Set[B]
}
```

You may

- declare (one or more) type params in square brackets
- add type parameters to classes, traits or methods
Outline

A little more about the type hierarchy
  Adding value classes

Type parameters and variance
  Type parameters
  Variance
  Bounded types

More about types
  Type members
  "Inner" types
  Type refinements
  Static duck typing
"Generics" and subtyping

Given the Animal and Cat (extends Animal) types:

- should a list of Cats also be a list of Animals...
  - if I just read it?
  - if I want to add to it?

- if I can feed an Animal, should I also be able to feed a Cat?
"Generics" and subtyping

Given the **Animal** and **Cat (extends Animal)** types:

- should a list of Cats also be a list of Animals...
  - if I just *read* it?
    (yes - *covariant*)
  - if I want to *add* to it?

- if I can feed an Animal, should I also be able to feed a Cat?
"Generics" and subtyping

Given the **Animal** and **Cat** (extends **Animal**) types:

▶ should a list of Cats also be a list of Animals...
  ▶ if I just *read* it?
    (yes - *covariant*)
  ▶ if I want to *add* to it?
    (no - *invariant*: what if I add a Horse to it?)

▶ if I can feed an Animal, should I also be able to feed a Cat?
"Generics" and subtyping

Given the Animal and Cat (extends Animal) types:

- should a list of Cats also be a list of Animals...
  - if I just read it?
    (yes - covariant)
  - if I want to add to it?
    (no - invariant: what if I add a Horse to it?)

- if I can feed an Animal, should I also be able to feed a Cat?
  (yes - contravariant; we’ll explain shortly)
Variance

Example (the question of variance in the List type)

Let B extend A.

Should List[B] extend List[A]?

We have 3 options:

- covariance = List[B] extends List[A]
- invariance = no relationship and no substitution
- contravariance = List[A] extends List[B]

Variance = relationship between parameterized types, given the relationship of their components
Variance syntax

▶ invariant is default

```java
class Container[A]
```

▶ + is covariant

```java
class CovariantContainer[+A]
```

▶ - is contravariant

```java
class ContravariantContainer[-A]
```
A contravariance example

```scala
class FeedAction[-T] {
  def apply(t : T): Unit = {
    println(t.toString + " eating")
  }
}

// this feeds the Cat
def feedCat(action: FeedAction[Cat], c: Cat): Unit = {
  action(c)
}

val f = new FeedAction[Animal]
// I can feed an Animal, so I can also feed a Cat
feedCat(f, new Cat)
```
Assume we have types Animal, Cat, Dog. Cat and Dog inherit from Animal.

```scala
class Cage[-A] {
  val underlying: A
}

val w: Cage[Cat] = new Cage[Animal]
println(w.underlying)
```

Everything OK with this code?
Variance positions

Assume we have types Animal, Cat, Dog. Cat and Dog inherit from Animal.

```scala
class Cage[-A] {
  val underlying: A
}
val w: Cage[Cat] = new Cage[Animal]
println(w.underlying)
```

Everything OK with this code?
(no - what type is w.underlying?!)

Rule
Types of val s are in covariant position.
(same for var s)
Variance positions

Assume we have types Animal, Cat, Dog. Cat and Dog inherit from Animal.

```scala
class Cage[-A] {
    val underlying: A
}
val w: Cage[Cat] = new Cage[Animal]
println(w.underlying)
```

Everything OK with this code? (no - what type is w.underlying?!)  

- vals can’t have contravariant types

**Rule**

Types of vals are in covariant position. (same for vars)
Variance positions (2)

Same hierarchy with types Animal, Cat, Dog.

```scala
class Cage[+A] {
  var underlying: A
}

def process(w: Cage[Animal]) {
  w.underlying = new Dog
}
process(new Cage[Cat])
```

Everything OK with this code?
Same hierarchy with types Animal, Cat, Dog.

```java
class Cage[+A] {
    var underlying: A
}

def process(w: Cage[Animal]) {
    w.underlying = new Dog
}
process(new Cage[Cat])
```

Everything OK with this code?
(no - we’re polluting a Cage[Cat] with a Dog)
Variance positions (2)

Same hierarchy with types Animal, Cat, Dog.

```java
class Cage[A] {
    var underlying: A
}

def process(w: Cage[Animal]) {
    w.underlying = new Dog
}
process(new Cage[Cat])
```

Everything OK with this code?
(no - we’re polluting a Cage[Cat] with a Dog)

Rule

Types of `vars` are in *covariant AND contravariant* position.
Variance positions in methods

Same hierarchy with types Animal, Cat, Dog.

```scala
class Cage[+A] {
  def method(a: A): Unit = ...
}

val w: Cage[Animal] = new Cage[Cat]
w.method(new Dog)
```

Everything OK with this code?
Variance positions in methods

Same hierarchy with types Animal, Cat, Dog.

```scala
class Cage[+A] {
  def method(a: A): Unit = ...
}
val w: Cage[Animal] = new Cage[Cat]
w.method(new Dog)
```

Everything OK with this code?
(no - we’re messing up declared and used types)

Rule

Types of method parameters are in contravariant position.
Variance positions in methods (2)

class Cage[-A] {
    def get: A = ...
}

val w: Cage[Cat] = new Cage[Animal]
val getResult = w.get

Everything OK with this code?
class Cage[-A] {
    def get: A = ...
}

val w: Cage[Cat] = new Cage[Animal]
val getResult = w.get

Everything OK with this code?
(no - w.get must be a Cat, is originally an Animal, might be a Dog)

Rule

*Method return types are in covariant position.*
Big rule.
Methods are
- contravariant in argument types and
- covariant in return types.
Big rule.
Methods are
- contravariant in argument types and
- covariant in return types.

Actually, methods have their own types (remember?) declared as:

```scala
trait Function1[-T1, +R] {
  def apply(t : T1) : R
}
```

Now you know why...
Outline

A little more about the type hierarchy
  Adding value classes

Type parameters and variance
  Type parameters
  Variance
  Bounded types

More about types
  Type members
  "Inner" types
  Type refinements
  Static duck typing
Variance is good, but...

```scala
class List[+A] {
  def add(a: A): List[A]
}
```

Error: Covariant type A occurs in contravariant position
▶ covariance makes sense
▶ but we cannot add covariant methods
Variance is good, but...

class List[+A] {
    def add(a: A): List[A]
}

Error: Covariant type A occurs in contravariant position

- covariance makes sense
- but we cannot add covariant methods
Solution: lower bounded types

```java
class List[+A] {
    def add[B <: A](b: B): List[B]
}
```

Notice the `[B <: A]` syntax:
- B is a type parameter to the method
- B is a supertype of A
- the result type `List[B]` is thus widened
Upper bounded types

```scala
class Car
class SuperCar extends Car

class Garage[A <: Car] (car: A)

// OK
new Garage(new SuperCar)
// error: inferred type arguments [String]
// do not conform to type parameter bounds [A <: Car]
new Garage("supercar")
```

- A is a subtype of Car (*is a Car*)
- A *has to be* a (subtype of) Car
- otherwise it doesn’t compile
Outline

A little more about the type hierarchy
   Adding value classes

Type parameters and variance
   Type parameters
   Variance
   Bounded types

More about types
   Type members
   "Inner" types
   Type refinements
   Static duck typing
Declaring type members

Classes, traits and singleton objects can have type members:

```java
class MyClass {
    // an abstract type
    type MyType
    // a bounded abstract type
    type MagicList <: List
    // a type alias
    type LL = java.util.LinkedList

    //...
}
```
Outline

A little more about the type hierarchy
  Adding value classes

Type parameters and variance
  Type parameters
  Variance
  Bounded types

More about types
  Type members
  "Inner" types
  Type refinements
  Static duck typing
Inner types

Classes, traits and singleton objects can have *inner types*:

```rust
class OuterClass {
    class InnerClass
    trait InnerTrait
    type InnerType

    //...
}
```
Referring and using inner types

```scala
class Outer {
    class Inner
}
val o = new Outer

// notice the type and the instantiation syntax
val i1: o.Inner = new o.Inner

// error: value Outer not found
val i2: Outer.Inner = new o.Inner

To use an inner type, you must use an outer instance
```
Referring and using inner types (2)

class Outer {
  class Inner
    def print(i: Inner): Unit = println(i)
}

val o1 = new Outer
val o2 = new Outer

val i2: o2.Inner = new o2.Inner

// error: type mismatch:
// found o2.Inner, required o1.Inner
o1.print(i2)

Different outer instances ⇒ different inner types
All inner types have a common supertype - Outer#Inner

class Outer {
  class Inner
    def print(i: Outer#Inner): Unit = println(i)
}

val o1 = new Outer
val o2 = new Outer

val i2: o2.Inner = new o2.Inner

// OK
o1.print(i2)
Outline

A little more about the type hierarchy
  Adding value classes

Type parameters and variance
  Type parameters
  Variance
  Bounded types

More about types
  Type members
  "Inner" types
  Type refinements
  Static duck typing
Enhancing existing types

We can add definitions/declarations to existing types

```scala
// assume this is in some library
class Printer {
  def print(o: Any): Unit
}

// in our code
// this is called structural definition
// we don't override any existing member
type VerbosePrinter = Printer {
  def printVerbose(o: Any): Unit
}

// structurally defined members are found via reflection
```
Outline

A little more about the type hierarchy
   Adding value classes

Type parameters and variance
   Type parameters
   Variance
   Bounded types

More about types
   Type members
   "Inner" types
   Type refinements
   Static duck typing
On duck typing

Q: What is duck typing?
Q: What is duck typing?
Q: What is the duck test?
On duck typing

Q: What is duck typing?
Q: What is the duck test?

The duck test
If something looks like a duck, swims like a duck and quacks like a duck, then it’s probably a duck.

We want Scala to do the duck test at compile time.
Static duck typing in Scala

Omit the type to be refined/enhanced:

```scala
// a purely structural type declaration
type SoundMaker = { def makeSound(): Unit }

class Dog {
  def makeSound(): Unit = {...}
}

class Car {
  def makeSound(): Unit = {...}
}

val d: SoundMaker = new Dog
val c: SoundMaker = new Car
```

- no relationship between e.g. Dog and Soundmaker...
- ...but Scala is OK
- ...and will use dog as a SoundMaker
Recap

A little more about the type hierarchy
  Adding value classes

Type parameters and variance
  Type parameters
  Variance
  Bounded types

More about types
  Type members
  "Inner" types
  Type refinements
  Static duck typing