Scala Generics

Type Parameterization

Scala Generics

Consider:

```scala
class Pair[T, S](val first: T, val second: S)
```

- `Pair` is not a type
- `Pair[T, S]` is a generic type, or type constructor
- `Pair[Int, String]` is a type because arguments for `T` and `S` are provided

THanks to type inference, these are equivalent:

```scala
new Pair(42, "String")
new Pair[Int, String](42, "String")
```

Generic Functions

Functions can also be generic. Given:

```
def getMiddle[T](a: Array[T]) = a(a.length / 2)
```

These two function calls are equivalent:

```
getMiddle(Array("Mary", "had", "a", "little", "lamb"))
getMiddle[String](Array("Mary", "had", "a", "little", "lamb"))
```

What is the type of `f` in:

```
val f = getMiddle[String] _
```

Exercise: write a verbose version of `f` above using `Function1`

FunctionN Classes

Like every value in Scala, a function value is an instance of a classes. In particular, a function value is an instance of one of Scala's "FuntionN" classes, where N is the number of parameters to the function. Here is `Function1` (with some details elided for brevity):

```scala
trait Function1[-T1, +R] extends AnyRef {
  def apply(v1: T1): R
}
```

So, if `getMiddle[String] _` has the type `Array[String] => String`, it's a value of type `Function1[Array[String], String]` which we could create directly as:

```scala
val f2 = new Function1[Array[String], String] {
  def apply(a: Array[String]): String = a(a.length / 2)
}
```

Variance Annotations on FunctionN Classes

Notice the - and + on the type parameters in `Function1`. A function is *contravariant* in its parameter types and *covariant* in its return type.

```
trait Function1[-T1, +R] extends AnyRef {
  def apply(v1: T1): R
}
```

These *variance annotations* signal to the compiler how the supertpye-subtype relationships of type arguments relate to the supertype-subtype relationship of the types these arguments parameterize.

- + means covariant – if `t` is a subtype of `u`, then `C1[t]` can be a subtype of `C2[u]`
- - means contravariant – if `t` is a subtype of `u`, then `C[u]` can be a subtype of `C[t]`
- No annotation, the default, is invariant – only `C1[t]` can be a subtype of `C2[t]`

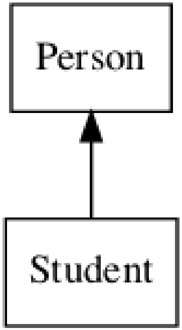
  Note: supertype and subtype are reflexive – every type is both a subtype and supertype of itself.

Variance of Function Values

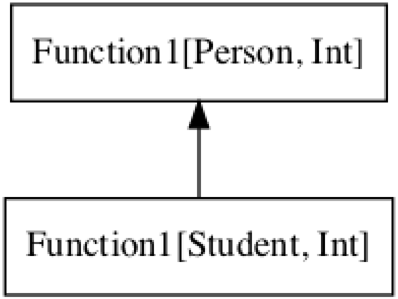
Given:

```scala
class Person(val name: String)
class Student(name: String, val id: Int) extends Person(name)
```

This relationship holds.



Should this relationship also hold?



Variance of Function Values

For example, given:

```
class Person(val name: String)
class Student(name: String, val id: Int) extends Person(name)

class NameLength extends Function1[Person, Int] {
  def apply(p: Person) = p.name.length
}
class GetId extends Function1[Student, Int] {
  def apply(s: Student) = s.id
}
class GetHashCode extends Function1[AnyRef, Int] {
  def apply(o: AnyRef) = o.hashCode
}
```

Should we be able to do this?

```
var getter1: Function1[Person, Int] = new GetId
```

How about this?

```
var getter2: Function1[Person, Int] = new GetHashCode
```

The LSP and Variance of Function/Method Parameters

Remember the Liskov Substitution Principle?

Subtypes should be substitutable for their supertypes.

Without getting into the exhaustive details of constraints and invariances, we can think of the LSP informally as

Require no more, promise no less.

A `Function1[Student, Int]` requires more of the parameter to the apply method than a `Function1[Person, Int]`, namely, that the parameter have an `id` member. So by the LSP,

- `Function1[Student, Int]` is not a proper subtype of `Function1[Person, Int]` because it requires more, and
- `Function1[AnyRef, Int]` is a proper subtype because it requires less.

Scala Enforces the LSP

So this does not compile:

```scala
var getter1: Function1[Person, Int] = new GetId
```

but this does:

```scala
var getter2: Function1[Person, Int] = new GetHashCode
```

The LSP and the Variance of Return Types

Functions are covariant in their return types, meaning return values of subclass methods can promise more, but cannot promise less.

```
val studentCreator = new Function1[String, Person] {
  def apply(name: String) = new Student(name, 1)
}
```

Variance of Scala Arrays (and Collections in General)

Scala arrays are invariant in their type parameter.

```
scala> val a1 = Array(1,2,3)
a1: Array[Int] = Array(1, 2, 3)

scala> val a2: Array[Any] = a1
<console>:12: error: type mismatch;
 found   : Array[Int]
 required: Array[Any]
```

The reason is that if the assignment to `a2` succeeded we could do something unsafe like:

```
a2(0) = "boom!'
```

So collections in Scala are invariant. In Java, collections are also invariant, but arrays aren't . . .

Java Arrays

For historical reasons, Java arrays are covariant. This compiles:

```java
String[] a1 = { "abc" };
Object[] a2 = a1;
a2[0] = new Integer(17);
String s = a1[0];
```

But the line:

```java
a2[0] = new Integer(17);
```

throws an `ArrayStoreException`. The reason for this odd behavior is that in the first versions of Java, before generics were added, the designers wanted to be able to write code like:

```java
void sort(Object[] a, Comparator cmp) { ... }
```

that would work with any array.

Lower Bounds

Say you have an immutable `Queue` class and you want to make it covariant in its type parameter, which is safe for immutable collections (becuase “modifying” them actually creates new collections taht can have a different type. Scala won’t allow this because method arguments are in contravariant position:

```scala
// Not the real scala.immutable.Queue
class Queue[+A] {
  def enqueue(elem: A) = new Queue( ... )
}
```

The way around this is make `enqueue` itself polymorphic and use a *lower bound* for its type parameter:

```scala
class Queue[+A] {
  def enqueue[B >: A](elem: B): Queue[B]
}
```

This is what Scala’s immutable Queue class does.

Flexible Polymorphic Immutable Collections

With the covariant type parameter and lower bound shown on the previous slide we can do this:

```scala
import scala.collection.immutable._

class Fruit
class Apple extends Fruit
class Orange extends Fruit

val appleQ1: Queue[Fruit] = Queue(new Apple, new Apple)
val fruitQ1: Queue[Fruit] = appleQ1.enqueue(new Orange)

val appleQ2: Queue[Apple] = Queue(new Apple, new Apple)
val fruitQ2: Queue[Fruit] = appleQ2.enqueue(new Orange)
```

Upper Bounds

Returning to our Pair example, consider this modification:

```scala
class Pair2[T <: Comparable[T]](val first: T, val second: T) {
  def smaller = if (first.compareTo(second) < 0) first else second
}
```

`T` must be a subtype of `Comparable[T]`. Without the type bound on `T`, the call to `compareTo` would not compile. So we can create a `Pair2` of any type `T` that is a subtype of `Comparable[T]`.

```
scala> new Pair2("Martin", "Odersky").smaller
res8: String = Martin
```

Try `new Pair2(1, 2).smaller`.

Context Bounds

Int does not implement Comparable[Int] but RichInt does, and there's an implicit conversion from Int to RichInt in scala.Predef:

```scala
implicit def intWrapper(x: Int): RichInt
```

Recall that we can provide a context bound to explicitly retrieve an implicit value or apply an implicit conversion:

```scala
class Pair3[T : Ordering](val first: T, val second: T) {

  def smaller = if (implicitly[Ordering[T]].lt(first, second)) first
       else second
}
```

Remember, we're not creating a subclass of Int, we're creating RichInt values from the Int values. So context bounds are different from upper or lower bounds, and far more flexible.