

44 Type Declarations

Defining Types, etc.

- Type, trait, class, object, case class, abstract class, value class:
 - what do they all mean? what's the difference?
 - “type” is to an object, class or trait what “val” is to value (i.e. an alias)

```
object List {  
  type IntList = List[Int]  
  def sum(ints: IntList): Int = ints match {  
    case Nil => 0  
    case Cons(x, xs) => x + sum(xs)  
  }  
}
```

- “type” can also be used to refer to a singleton object, as in `List.type`.
- a trait (or object) can also define a “type” member

```
trait Base {  
  type T  
  def method: T  
}
```


- where `T` must be defined as a concrete type wherever `Base` is subclassed concretely, e.g.

```
class Dog extends Base {  
  type T = String  
  def method: String = "woof!"  
}
```

Defining types (2)

Traits

Traits are somewhat similar to interfaces in Java, but they also allow method definitions (as does Java 8) so multiple inheritance really works!



- define *behavior*;
- are basic to type hierarchies and can be “mixed in” to provide multiple inheritance;
- allow the definition of abstract/concrete types/properties/methods (features are abstract if they lack concreteness);
- traits can define both concrete and abstract methods/values (example: abstract compare function);
- can have type parameters (“parametric polymorphism”), e.g. *A* in following example (not exactly like in Scala package):

```
trait Monad[+A] {  
  def map[B, C](f: A => B): C  
  def flatMap[B, C](f: A => Seq[B]): C  
  def foreach[U](f: A => U): Unit  
}
```

We will be talking a LOT about monads in the future!



- But *cannot* have value parameters (you need an abstract class for that) [Note that this implies that you can't define a parametric type of a trait to be a member of a *type class*];
- Note: traits with concrete code cannot be extended by Java-7 classes

Defining types (3)

Value classes

- *Int*, *Boolean*, *Double*, etc.
- Since 2.10, programmers can define Value Classes
 - some efficiency improvements
 - extend *AnyVal* (rather than *AnyRef*)
 - May not contain references to non-Value objects
 - various other restrictions
 - only one actual value
 - no “require”
- Example: *UniformDouble* in an obsolete assignment:

```
case class UniformDouble(x: Double) extends AnyVal with Ordered[UniformDouble] {  
  def + (y: Double) = x + y  
  def compare(that: UniformDouble): Int = x.compare(that.x)  
}
```

Defining types (4)

- **Abstract classes**
 - similar to traits but with minor differences:
 - can define constructor parameters (in addition to type parameters as in traits)
 - cannot be mixed in like traits (a sub-class can have only one super-class but many super-traits);
 - polymorphism is slightly more efficient (because *super* is known and is statically bound);
 - can be sub-classed by Java-7 classes;
 - is binary-compatible: if members are added to an abstract class that is used by other modules, those modules do not need to be recompiled (assuming they don't reference the new member) —but if new member was added to trait, modules would need to be recompiled.
- **Case classes/objects** (already discussed in detail)
 - extend *Product* (the trait extended by Tuples) but cannot be sub-classed;
 - case *objects* have no parameters (that's why they're objects, not classes).
- **Object** (companion object or ordinary singleton)
 - companion object is where we put the Scala equivalent of “class” methods
 - singleton objects are just that, e.g. a message type (“Status”) that we sent to our Ticket Agency actor in week 1 (or 2?).

Modules/Packages

- A Scala file is called a “module”
 - A module corresponds more to a Java package than a Java class (whereas Java packages are directories of files)
 - Thus, a module in Scala can define multiple traits, classes, objects, etc.
 - You can use the keyword *package* to define a more specific package.
 - *However*, if you want to execute the main method of a module, or run tests defined in a module, you must ensure that the (first) package statement corresponds to the position in the directory structure. Otherwise, things get confused because Java is quite strict about the class packaging.

Package object

- As in Java, there is also a package “object” which can contain definitions that are common to the whole package.
- I find this particularly useful for defining types as in *type Ints = Seq[Int]* because you can't do that inside a module.