Updated: 2021-02-26

3.5 Type Declarations

What are types?

- A type defines the range of values (domain) and the set of operations that may operate on the given values.
- What does a program do at run-time?
  - It uses machine instructions such as + to operate on *values*.
- What does a compiler* do at compile-time?
  - It uses grammar and inference to operate on *types*;
  - Then it generates machine instructions to execute during run-time.

* of a typed-language

Defining Types (1)

- 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

- define *behavior;*
- are basic to type hierarchies and can be “mixed in” to provide multiple inheritance (with rules on exactly how methods are overridden if necessary);
- 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);

Defining Types (3)

Traits (continued)

- 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
}
```

- 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 (not so important now): traits with concrete code cannot be extended by Java-7 classes*

Defining Types (4)

Value classes

- *Int, Boolean, Double,* etc.
- Programmers can define Value Classes (since 2.10):
  - 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 (5)

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.

Defining Types (6)

Object (companion object or ordinary singleton)

- companion object is where we put the Scala equivalent of “class” (static) methods
- singleton objects are just that, e.g. a message type (“Status”) that we sent to our Ticket Agency actor in week 1.

Case classes/objects (already discussed in detail??)

- extend *Product* (the trait extended by Tuples) but cannot normally be sub-classed (they are leaves of the class hierarchy);
- case *objects* have no parameters (that’s why they’re objects, not classes).

Defining Types (7): Case classes continued

When you use a case class, the compiler provides a lot of "boiler-plate" code for you:

- It provides **val** for each "element"—this essentially exposes the elements to the public scope
- You can override this **val** in several ways:
  - Add the **private** keyword (or **private**[*scope*]);
  - You can make it a variable by using the **var** keyword;
  - You can put the element in a second parameter set—but be careful as that parameter is really not an element—but one time you might want to do that is if you want the parameter to be "call-by-name."

Defining Types (8): Case classes continued

Continuing with what the compiler provides:

- It provides *equals, hashCode, and toString* as instance methods;
- It provides *apply* and *unapply* in the companion object (even though you may not have defined a companion object, the compiler provides an invisible one).
  - *apply*: takes the exact same parameters as the elements and essentially exists so that you don't have to keep using the **new** keyword;
  - *unapply*: takes an object and returns *Some(tuple)* where the tuple corresponds to the elements. If the object isn't of the same type as the case class, then *None* will be returned. *Unapply* is primarily used by the **match...case** construction.
- It provides a default implementation of *Serializable* (true also for a case object);

Defining Types (9): Case classes continued

There are one or two other aspects to case classes:

- Case classes support the *copy* method;

```
case class Complex(real: Double, imag: Double) {
  def moveHorizontal(dx: Double): Complex = copy(real = real + dx)
  def moveVertical(dy: Double): Complex = copy(imag = imag + dy)
}
```

- The *apply* method is often used for the purpose of type inference, rather than being invoked. For example:

```
implicit val formatFormat: RootJsonFormat[Format] = jsonFormat4(Format.apply)
```

- In this case, *Format.apply* will only be invoked directory when reading from Json. When writing to Json, the *apply* method is used solely for type inference.

Defining Types (10): Tuples

Tuples are case classes (and so extend *Product*) but with names chosen for their position.

- A case class allows you to name the elements—a TupleN has element names like _1, _2, etc.;
- Note that *(T,R)* is syntactic sugar for *Tuple2[T,R]*.

Defining Types (11): Extending functions

What about extending function types? Functions aren't classes, are they?

- No, but they are types and types can be extended;
- See the *lab-sorted* code for some examples.
- Note that *T=>R* is syntactic sugar for *Function1[T,R]*.

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.