Programming in Scala

based on the book
“Programming in Scala Third Edition”

Agenda

- Chapter X - Composition and Inheritance
  - Abstract classes
  - Methods and fields
  - Extending classes
  - Inheritance
  - Polymorphism and dynamic binding
- Chapter XI - Scala’s Hierarchy
  - AnyVal
  - AnyRef
  - Custom value classes

Chapter X - Composition and Inheritance

Abstract classes

- May contain abstract methods: no implementation (unlike in Java, there is no `abstract` keyword for methods)
- Non abstract methods are called concrete
- A class with abstract members must itself be declared abstract using the abstract keyword
- Cannot be instantiated

```
abstract class Element {
    def contents: Array[String]
}
```

- Terminology: Class Element declares the abstract method contents, but currently defines no concrete methods.

Methods vs Fields

```
abstract class Element {
    def height = contents.length
}
```

vs.

```
abstract class Element {
    val height = contents.length
}
```

Methods vs Fields

- These definitions (`def` and `val`) are equivalent from a client's point of view.
  - The only difference is that field accesses might be slightly faster than method invocations because the field values are pre-computed when the class is initialized, instead of being computed on each method call.
  - On the other hand, the fields require extra memory space in each Element object.
- Guideline: it is encouraged to define methods that take no parameters and have no side effects as parameterless methods, but you should never define a method that has side-effects without parentheses because invocations of that method would then look like a field selection. For example `read` vs `.read()`
- Uniform access principle: client code should not be affected by a decision to implement an attribute as a field or method

Extending classes

```
class ArrayElement(conts: Array[String]) extends Element {
    def contents: Array[String] = conts
}
```

- Terminology: subclass/subtype, superclass
- If you leave out an `extends` clause, the Scala compiler implicitly assumes your class extends from `scala.AnyRef`, which on the Java platform is the same as class `java.lang.Object`

Inheritance

- *Inheritance* means that all members of the superclass are also members of the subclass, with two exceptions:
  - First, private members of the superclass are not inherited in a subclass
  - Second, a member of a superclass is not inherited if a member with the same name and parameters is already implemented in the subclass. In that case we say the member of the subclass *overrides* the member of the superclass. If the member in the subclass is concrete and the member of the superclass is abstract, we also say that the concrete member *implements* the abstract one.
- Subtyping means that a value of the subclass can be used wherever a value of the superclass is required.
  - For example: `val e: Element = new ArrayElement(Array("hello"))`

Overriding methods and fields

- Fields and methods belong to the same namespace. This makes it possible for a field to override a parameterless method.

```
class WontCompile {
    private var f = 0 // Won't compile, because a field
    def f = 1         // and method have the same name
}
```

Overriding methods and fields (cont'd)

- Java's four namespaces are *fields*, *methods*, *types*, and *packages*
- Scala's two namespaces are
  - *values* (fields, methods, packages, and singleton objects)
  - *types* (class and trait names)

override keyword

- The modifier is **required** for all members that override a concrete member in a parent class.
- The modifier is **optional** if a member implements an abstract member with the same name.
- The modifier is **forbidden** if a member does not override or implement some other member in a base class.

Overriding methods and fields (cont'd)

```
class Cat {
  val dangerous = false
}

class Tiger(
  override val dangerous: Boolean,
  private var age: Int
) extends Cat
```

```
class Tiger(
  param1: Boolean, param2: Int
) extends Cat {
  override val dangerous = param1
  private var age = param2
}
```

Both Tiger classes are the same.

Invoking superclass constructors

```
class LineElement(s: String) extends  ArrayElement(Array(s)) {
  override def width = s.length
  override def height = 1
}
```

Polymorphism and dynamic binding

- A variable of type Element could refer to an object of type ArrayElement. The name for this phenomenon is *polymorphism*, which means “many shapes” or “many forms.”
  - For example
    - `val e1: Element = new ArrayElement(Array("hello", "world"))`
    - `e1.m() // This calls ArrayElement’s method if it overrides m`
- Method invocations on variables and expressions are dynamically bound. This means that the actual method implementation invoked is determined at run time based on the class of the object, not the type of the variable or expression.

final keyword

```
final def fn() = {} // cannot be overridden

final class Leaf extends Element {} // cannot be subclassed
```

Composition vs Inheritance

If what you're after is primarily code reuse, you should in general prefer composition to inheritance. Only inheritance suffers from the fragile base class problem, in which you can inadvertently break subclasses by changing a superclass.

One question you can ask yourself about an inheritance relationship is whether it models an is-a relationship. For example, it would be reasonable to say that ArrayElement is-an Element. Another question you can ask is whether clients will want to use the subclass type as a superclass type. In the case of ArrayElement, we do indeed expect clients will want to use an ArrayElement as an Element.

If you ask these questions about the inheritance relationships, do any of the relationships seem suspicious? In particular, does it seem obvious to you that a LineElement is-an ArrayElement? Do you think clients would ever need to use a LineElement as an ArrayElement?

Chapter XI - Scala's Hierarchy

Scala's hierarchy

- Every class inherits from a common superclass named `Any`
- Scala also defines some interesting classes at the bottom of the hierarchy, `Null` and `Nothing`, which essentially act as common subclasses

```
def error(message: String): Nothing =
  throw new RuntimeException(message)
```

  - Because `Nothing` is a subtype of every other type, you can use methods like error like this:

```
def divide(x: Int, y: Int): Int =
  if (y != 0) x / y
  else error("can't divide by zero")
```

Methods of Any

- Methods of Any

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def ##: Int
def hashCode: Int
def toString: String
```

- == is the same as `equals`, individual classes can tailor what == or `!=` means by overriding the equals method

AnyVal and AnyRef

- The root class `Any` has two subclasses: `AnyVal` and `AnyRef`.
- `AnyVal` is the parent class of value classes in Scala.
  - Built-in value classes: `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean`, and `Unit`.
  - The instances of these classes are all written as literals in Scala.
    - For example, 42 is an instance of `Int`, 'x' is an instance of `Char`, and `false` an instance of Boolean.
  - You cannot create instances of these classes using `new`.
  - `Unit` corresponds roughly to Java's void type; it is used as the result type of a method that does not otherwise return an interesting result. Unit has a single instance value, which is written `()`.
- `AnyRef` is just an alias for class `java.lang.Object`
  - It has `eq` and `ne` methods that are the equivalent of == and != in Java

Value classes

Like the built-in value classes, an instance of your value class will usually compile to Java bytecode that does not use the wrapper class. In contexts where a wrapper is needed, such as with generic code, the value will get boxed and unboxed automatically.

Custom value classes

- It must have exactly one parameter
- It must have nothing inside it except defs
- No other class can extend a value class
- A value class cannot redefine `equals` or `hashCode`.
- Make it a subclass of `AnyVal`
- Put `val` before the one parameter

```
class Dollars(val amount: Int) extends AnyVal {
  override def toString() = "$" + amount
}
```