Week 2: OOP

What is OOP?

- OOP (Object Oriented Programming) is a programing paradigm revolving around "encapsulating" behavior and state together, as well as some template system.

- Most modern languages have some degree of OO, including Java, C++, and Python.

- Commonly, OO revolves around "classes" which are templates for objects that contain state and behavior.

OOP in Scala

- Java, the language Scala is partially based on, is strongly-OO.
- Scala's OOP is similar to Java's with several important improvements, such as "mixin inheritance".
- We will mostly cover on Scala's differences from Java in OO, since prior knowledge is assumed.

A Simple Java Class

```java
package com.xorinc.scalatour;

/**
  * A simple class that contains a mutable int and
  * an immutable string, and an extra function.
  */
public class SimpleClass {

    private int foo;
    private String bar;

    public SimpleClass(int foo, String bar){
        this.foo = foo % 10;
        this.bar = bar;
        System.out.println("Hello from SimpleClass");
    }

    public int getFoo() { return foo; }
    public String getBar() { return bar; }

    public void setFoo(int newFoo) {
        this.foo = newFoo % 10;
    }

    public String something(String pre) {
        return pre + " " + foo + " " + bar;
    }
}
```

The same class in Scala

```scala
package com.xorinc.scalatour

/**
  * A simple class that contains a mutable int and
  * an immutable string, and an extra function.
  */
class SimpleClass(newFoo: Int, newBar: String) {

    private var _foo = newFoo % 10
    val bar: String = newBar

    println("Hello from SimpleClass")

    // getter for _foo
    def foo: Int = _foo

    // setter for _foo
    def foo_=(i: Int): Unit = _foo = i % 10

    def something(pre: String): String = s"${pre.toUpperCase} $foo $bar"

}

// example usage
val simple = new SimpleClass(11, "Hi")
println(simple.foo) // prints "1"
simple.foo = 24
println(simple.something("Foo")) // prints "FOO 4 Hi"
```

Other Class Syntax

- Constructors other than the primary one are declared as `def this(args) = {...}`. The body syntax is equivalent to Java's.
- `this` as a variable behaves similarly to Java.
- Some class member names have special shorthands:

```
val foo = ...

// equivalent
foo(bar) -> foo.apply(bar)

// equivalent
foo.baz = beep -> foo.baz_=(beep)

//equivalent
foo(bar) = baz -> foo.update(bar, baz)
```

Special Members

- All user-defined classes have methods such as `toString`, `getClass`, `equals`, and `hashCode` (which is aliased as `##`).
- `==` and `!=` are null-safe versions of `equals`.
- Unlike in Java, `==` is always value equality. To compare reference, use `eq` and `ne`.

Non-mixin Inheritance

- Like in Java, it is possible to inherit classes not declared `final`.

- Instead of a **`super`**`(args)` call in the first line of the constructor, the superclass constructor is called at the `extends` clause.

- When overriding a method from the superclass, the `override` keyword is required. Java's optional `@Override` annotation is similar.

Non-mixin Inheritance

- In addition to `final` classes, there is a weaker form, `sealed` classes, that cannot be inherited except by classes in the same source file.

- `super` can be used to refer to the superclass's members like in Java.

- Note that `val`s can be overriden, and parametrless `def`s can be overriden by `val`s with the same name and return type.

Type Tests and Casts

```
// Java
anVariable instanceof AnClass // type test
(AnClass) anVariable // type case
```

```
// Scala
anVariable.isInstanceOf[AnClass] // type test
anVariable.asInstanceOf[AnClass] // type cast
```

Abstract Classes

- Abstract classes are similar to Java's, denoted with the `abstract` keyword.
- Any `val` or `def` without an assignment is automatically `abstract`; the keyword is not required.

Member Visibility

- Unlike in Java, `public` is the default visibility in Scala.

- `private` and `protected` behave just like in Java.

- `private[name]` is like regular `private`, but public to the package/class `name` and everything inside it.

- `private[this]` is like `private`, except objects of the same class can't see each others' members marked `private[this]`

Packages and Imports

```
package com.xorinc.scalatour // pretty much the same as Java

// normal import
import java.util.Date
// all of package `scala`
import scala._
// Option and Predef
import scala.{Option, Predef}
// Option as Optional
import scala.{Option => Optional}
// all of `scala` exception Option
import scala.{Option => _, _}
// all members of the object myVar
import myVar._
// addition from aNumber
import aNumber.+
```

Singletons

- Scala is pure OO, and thus everything is an object. Hence, Java's `static` notation does not make sense.

- Scala has a special notation for singleton objects, which are classes with exactly one instance. They are declared like classes, except without a constructor and with the keyword `object`.

- A singleton object with the same name as a normal class placed in the same source file is a companion object.

- Companion objects' members behave just like Java `static` members, except singletons can extend classes and can be passed as values themselves.

Special Types

- `scala.Any` is the superclass of *every* single type, no exceptions. `foo.isInstanceOf[Any]` is *always* true.
- `scala.AnyRef` (alias for `java.lang.Object`) is a subtype of `Any` and the parent of all user-defined classes.
- `scala.AnyVal` is the other subtype of `Any` and the parent of the value types (including numbers, `Boolean`, `Char` and `Unit`)
- `scala.Null` is a subtype of every single `AnyRef` subtype, and its only member is `null`.
- `scala.Nothing` is a subtype of *every* single type, but no object belongs to it. `foo.isInstanceOf[Nothing]` is *always* false.

Case Classes

- Case classes provide an easy syntax for creating immutable data containers.

```
case class AnCaseClass(i: Int, s: String)
// becomes
class AnCaseClass(val i: Int, val s: String) extends scala.Product {

    override def toString() = s"AnCaseClass($i,$s)"

    // plus some other compiler-generated members
}
object AnCaseClass {

    def apply(i: Int, s: String): AnCaseClass = new AnCaseClass(i, s)

    // this is for pattern matching, more on this later
    def unapply(x: AnCaseClass): Option[(Int, String)] = Some(x.i, x.s)
}
```

Value Classes

- Value classes are user-defined subclasses of `scala.AnyVal`
- Value classes are used to create wrappers that do not create objects in the underlying platform.
- Value classes are important in the extension method pattern we will see later.

Traits

- One of the biggest selling points for Scala's OO is `trait`s.
- A `trait` is like a Java 8 `interface`, with some bonuses.
- Traits are the key to Scala's mixin inheritance.

Traits

- Traits can contain almost anything a class can, except constructors.
- Traits can extend other traits, or a class, which means any class extending the trait has to extend that class, too.
- Traits can be mixed into a class to create a new type combining features of all the types mixed together.

Mixins

- `AnClass with AnTrait with YourOtherTrait` is a mixin.
- Instead of extending a class or trait, it is possible to extend a mixin.
- Triaits using `abstract override` can modify the behavior of existing classes by simply mixing them in.

Type Refinement

- Like Java, scala has anonymous classes, which have similar syntax.
- Unlike Java, Scala can infer the anon class's refined type, allowing its methods to be called.

```
val anonClass = new SomeClass {
    override def foo: Int = compute();
    def aNewFunction(s: String): String =
        s + "hello from anon class"
}
println(aNewFunction("foo")) // prints "foohello from anon class"
```

`type`

- Similarly to C's and C++'s `typedef`, Scala has `type`, which among other uses can be used to alias types.
- `foo.type` refers to a type that uniquely identifies `foo`, which can be used in function signatures to ensure it returns the same object.
- `AnSingleton.type` additionally refers to the type of an `object`-declared singleton.