Inheritance and traits

Scala training, 2016

Outline

Single Inheritance

- Scala offers class-based reference, just as in Java
- Only one class might be extended
- If *extends* is missing, AnyRef is considered to be the super class
- All non-private members are inherited
- All non-final members can be overriden

```
class Animal
class Snake extends Animal
class RattleSnake extends Snake
class Wolf extends Animal
```

Constructor Arguments

- Classes inheriting from a base class must provide the arguments during construction:
  - by calling the super class's primary constructor OR
  - by calling an auxiliary constructor

```
class Animal(val age: Int) {
  var name = ""
  def this(age: Int, name: String) = {
    this(age)
    this.name = name
  }
}
class Snake(age: Int) extends Animal(age)
class RattleSnake(age: Int) extends Animal(age,
    "rattle-snake")
```

final

- Prevents a class from being extended

```
// Console output
scala> final class Animal
defined class Animal

scala> class Snake extends Animal
console>:11: error: illegal inheritance from final
    class Animal
        class Snake extends Animal
                            ^
```

sealed

- Allows a class to be subclassed only within the same source file
- Sealed classes / traits have one advantage when used in pattern match (covered later):
  - the compiler checks that all cases have been covered
  - this does NOT happen for regular classes / traits

```
sealed class Animal
class Snake extends Animal
class Wolf extends Animal
```

override

- similar meaning to Java's @Override annotation
- defs can be overriden with vals or lazy vals

```
class Animal {
  def name = "Animal"
}

class Snake extends Animal {
  override val name = "Snake"
}
```

override

- similar meaning to Java's @Override annotation
- defs can be overriden with vals or lazy vals

```
class Animal {
  def name = "Animal"
}

class Snake extends Animal {
  override val name = "Snake"
}
```

- Can you override vals with defs?

super

- Used for calling a visible method on a superclass / trait
- It can be used with a qualifier
- Qualifiers must be imediate parent classes / traits
- The qualifier is used to select the type defining the called method

```
class Animal { def name: String = "" }
trait Lion  extends Animal { override def name = "L" }
trait Tiger extends Animal { override def name = "T" }
class Liger extends Animal {
  override def name = super[Tiger].name
}
```

Abstract Classes

- Abstract classes are just classes with optionally undefined members / fields
- They can't be instantiated
- Important note: they can receive constructor arguments (just like any other class)

```
abstract class Instrument {
  def play(): Unit
}
class Drum extends Instrument {
 override def play(): Unit = println("drum sound")
}
```

Traits: Basics

- A trait is a basic piece of functionality
- It allows you to mix-in behavior
- In terms of Java, think of interfaces that also have concrete methods

```
trait PartialFunction[A, B] {
  def isDefinedAt(x: A): Boolean
  def applyOrElse[A1, B1](x: A1, def: A1 => B1): B1 =
    if (isDefinedAt(x)) apply(x) else def(x)
}
```

Rules of playing

- Traits cannot have parameters
- Classes can inherit from exactly one class and / or mutiple traits
- First keyword must be *extends*, then use *with*
  - *with* = *implements*

```
scala> trait Animal(val name: String)
console>:1: error: traits or objects may not have
    parameters

// This is how a class testing MyClass could look like
class MyClassSpec extends WordSpecLike with Matchers
```

Self Types / The cake pattern

- A kind of dependency injection

Self Types / The cake pattern

- A kind of dependency injection
  - Q: dependency injection?

Self Types / The cake pattern

- A kind of dependency injection
  - Q: dependency injection?
- Forces classes to mix-in multiple behaviors (defined in the self type)

Self Types / The cake pattern

- A kind of dependency injection
  - Q: dependency injection?
- Forces classes to mix-in multiple behaviors (defined in the self type)
- Try to avoid self types, they easily break backwards compatibility

Self Types / The cake pattern

- A kind of dependency injection
  - Q: dependency injection?
- Forces classes to mix-in multiple behaviors (defined in the self type)
- Try to avoid self types, they easily break backwards compatibility

```
trait A { def method(): Unit }

// Basically if B depends on A you can use this self
// type to get access to A's members
trait B { self: A =>
  method()
}
```

Wrap up

- Any number of traits can be mixed in
- Traits don't allow constructor arguments. Abstract classes do.
- In classes, super calls are statically bound. In traits they are dynamic
- Inherit traits in Java code using abstract classes

What about multiple inheritance?

- The solution to the diamond problem:
  - It considers the order in which traits are inherited
  - Solves conflicting members by picking the implementation supertype appearing furthest to the right
- It's called *trait linearization*

```
class Animal { def name: String }
trait Lion  extends Animal { override def name = "L" }
trait Tiger extends Animal { override def name = "T" }
class Liger extends Lion with Tiger
val liger = new Liger // liger.name?
```

Practice!

http://workshop.rosedu.org/2016/sesiuni/scala/lab4