Object-Oriented Programming in Scala

Georgia Tech

Scala OOP Overview

Mostly like Java with important differences:

- overridden methods must have `override` modifier, which is part of the language, not an annotation
- instead of interfaces Scala has *traits*, which are much richer
  - traits can have everything a class can have except constructors
  - a class can "mix-in" any number of traits (kinda like multiple inheritance, but without the "diamond inheritance problem")

Note: these slides based on examples in Cay Horstmann's excellent Scala for the Impatient, 2ed

Georgia Tech

Extending Classes

```
1  class Person(val name: String, val age: Int) {
2    override def toString = s"${getClass.getName}[name=$name]"
3  }
4  class Employee(name: String, age: Int) extends Person(name, age) {
5    var salary: Int = 0.0
6  }
```

- `Person` implicitly extends `AnyRef` (`java.lang.Object`)
- `name` and `age` are *parametric fields* – constructor parameters that define instance variables
- `Employee`'s constructor takes two parameters that are passed to `Person` constructor (equivalent to a `super()` call in a Java constructor)

Georgia Tech

Run-time Type Identification

- `obj.isInstanceOf[Cl]` like `obj instanceof Cl` in Java
- `obj.asInstanceOf[Cl]` like `(Cl) obj` in Java
- `classOf[Cl]` like `Cl.class` in Java

Georgia Tech

Overriding Fields

```
1  abstract class Person(val name: String) {
2    def id: Int
3    override def toString = s"${getClass.getName}[name=$name]"
4  }
5  class SecretAgent(val id: Int, codename: String)
6      extends Person(codename) {
7    override val name = "secret" // 'Dont want to reveal name . . .
8    override val toString = "secret" // . . . or class name
9  }
```

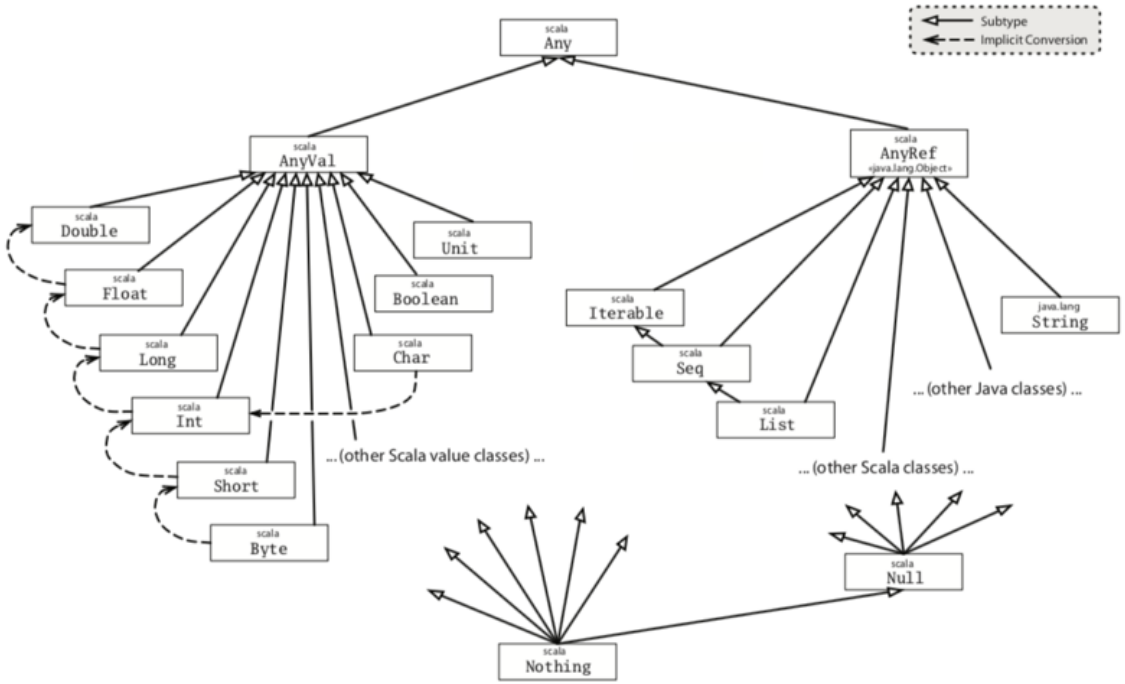
- `id` is abstract in `Person` because it is not defined (so `Person` must be declared `abstract`, just like in Java)
- `SecretAgent` overrides `id` with a `val` field (could add `override` modifier, but not required when overriding abstract member)

Rules

- A `def` can only override another `def`
- A `val` can only override another `val` or a parameterless def
- A `var` can only override an abstract `var`

Georgia Tech

Scala's Class Hierarchy



Scala has two "bottom types"

- `Null` has a single value, `null`, which is an instance of any class
- `Nothing` has no instances and is useful in a couple of places:
  - Empty list `Nil` has type `List[Nothing]`, which is a subtype of `List[T]` for any `T`
  - `AnyRef` defines a `???` method with return type `Nothing` that simply throws a `NotImplementedError` when invoked

Georgia Tech

Equality

Use `eq` for identity equality (alias test – like `==` in Java). Similar to Java, `AnyRef`'s `equals` method invokes `eq`. Override `equals` like this:

```
1  class Item(val description: String, val price: Double) {
2    final override def equals(other: Any) = other match {
3      case that: Item => description == that.description && price ==
           that.price
4      case _ => false
5    }
6    final override def hashCode = (description, price).##
7  }
```

- Recipe similar to Java's, but much more convenient
- Remember parameter type is `Any`
- Marked `final` to prevent symmetry problems in subclasses
- `##` is a convenience method on tuples which makes defining `hashCode` trivial

Georgia Tech

Companion Objects

Scala doesn't have "static" members but use cases for static members can be done with a *companion object*, which:

- has the same name as its companion class
- must be defined in the same source file as its companion class
- has access to its companion class's private members (and vice-versa)

Companion objects are most often used for factory methods:

```
1  class Item(val description: String, val price: Double)
2
3  object Item {
4    def apply(description: String, price: Double): Item =
5      new Item(description, price)
6  }
7
8  val item = Item("Key Lime", 3.14) // Calls Item.apply
```

Georgia Tech

Traits as Interfaces

```
1  trait Logger {
2    def log(msg: String)
3  }
4  class ConsoleLogger extends Logger {
5    def log(msg: String) = { println(msg) }
6  }
```

- Pretty much like a Java interface
- `extends`, not `implements`

Traits with Concrete Implementations

Traits can have concrete implementations (like default methods in Java interfaces), so our `ConsoleLogger` could be a trait:

```
1  trait ConsoleLogger extends Logger {
2    def log(msg: String) { println(msg) }
3  }
```

Then we can "mix-in" the trait without having to override any methods:

```
1  abstract class SavingsAccount(var balance: Int) extends ConsoleLogger {
2    def withdraw(amount: Int) {
3      if (amount > balance) log("Insufficient funds")
4      else balance -= amount
5     }
6  }
```

Georgia Tech

Objects with Traits

We can have `SavingsAccount` extend the abstract `Logger` instead of the concrete `ConsoleLogger`.

```
1  abstract class SavingsAccount(var balance: Int) extends Logger {
2    def withdraw(amount: Int) {
3      if (amount > balance) log("Insufficient funds")
4      else balance -= amount
5    }
6  }
```

You can mix in a trait with a concrete implementation of `log` at construction:

```
1  val acct = new SavingsAccount(1) with ConsoleLogger
```

This works because `SavingsAccount` is a subtype of `Logger` and so is `ConsoleLogger`.

Georgia Tech

Stackable Modifications

Traits can invoke methods in other traits that have a common supertype declaring the method. The supertype can be abstract, and the result is that a chain of operations takes place when the method is called.

```
1  trait Timestamping extends ConsoleLogger {
2    override def log(msg: String) =
         super.log(s"${java.time.Instant.now()} $msg")
3  }
4  trait Shortening extends ConsoleLogger {
5    override def log(msg: String) =
6      super.log( if (msg.length <= 15) msg else s"${msg.substring(0,
           12)}...")
7  }
```

Here, `super` doesn't mean "supertype", it means "trait that was mixed-in to my left."

Georgia Tech

Resolution of `super` in Stacked Traits (1/2)

For simple mixin sequences you may think of method resolution as "back to"front". (Note the `with` syntax when extending multiple traits.)

```
1 val acct1 = new SavingsAccount(1) with Timestamping with Shortening
2 acct1.withdraw(2)
```

In the code above, `Shortening` is furthest to the right, so its `log` method is called with "Insufficient funds", which, being 18 characters, is passed to the `log` method in `Timestamping` so we get something like

```
1 2019-02-17T23:28:15.747452Z Insufficient funds
```

Georgia Tech

Resolution of `super` in Stacked Traits (2/2)

Here we mix-in `Timestamping` last, so its `log` method is called with "Insufficient funds", `Timestamping.log` prepends a timestamp, then passes the result to `Shortening.log` because its to the left of `Timestamping` in the mix-in order. So

```
1 val acct2 = new SavingsAccount(1) with Shortening with Timestamping
2 acct2.withdraw(2)
```

gives us something like

```
1 2019-02-17
```

Abstract Overrides

Because `super` calls are dynamically bound, you can invoke an abstract method as long as you mark your method as `abstract override`. See the `Shouting` we've added below.

```
1  trait Timestamping extends ConsoleLogger {
2    override def log(msg: String) =
         super.log(s"${java.time.Instant.now()} $msg")
3  }
4  trait Shortening extends ConsoleLogger {
5    override def log(msg: String) =
6      super.log( if (msg.length <= 18) msg else s"${msg.substring(0,
           10)}")
7  }
8  trait Shouting extends Logger {
9    abstract override def log(msg: String) =
10     super.log(msg.toUpperCase)
11 }
```

This is saying "we assume a concrete `log` method exists." The compiler ensures that you can only mix `Shouting` into a class that somehow provides a concrete `log` method.

Compiling Traits with Abstract Overrides

The compiler ensures that the super call will succeed. So this will compile because `Shortening` provides a concrete `log` method (from `ConsoleLogger`) for the `super` call in `Shouting`

```
1 val acct3 = new SavingsAccount(1) with Shortening with Shouting
2 acct3.withdraw(2) // => INSUFFICIENT FUNDS
```

But this will not compile because `Shortening`'s `super` call is refferring to `Shouting`'s `log` method, which has no concrete `log` method for its `super` call.

```
1 // Won't compile
2 val acct4 = new SavingsAccount(1) with Shouting with Shortening
3 acct4.withdraw(2)
```

The resolution of `super` is called *linearization* and it is the (somewhat complicated) way Scala solves the diamond inheritance problem.

Georgia Tech

Packages

Like Java, Scala code that's not in a named package is in the global *unnamed* package. Put code into packages in two ways:

- Putting a package declaration at top of source code file, like in Java:

```
1 package edu.gatech.cs2340.zoo
2
3 class Animal
4 trait Mammal extends Animal
5 class Dog extends Animal with Mammal
```

and . . .

Namespace Packaging Syntax

- Explicit packaging syntax (like the namespace feature of other languages):

```
1 package edu.gatech.cs2340 {
2   package zoo {
3     class Animal
4     trait Mammal extends Animal
5     class Dog extends Animal with Mammal
6   }
7 }
```

The second approach is flexible but not used much in practice.

Note: although Scala allows you to organize your code any way you want, be a good person and follow Java's package naming (reverse domain name) and source code organization conventions (source directory tree mirrors package structure).

Georgia Tech

Imports

Scala imports are more flexible than Java's

- `import edu.gatech.cs2340.zoo.Animal` – import `Animal` into namespace as simple name (name without package).
- `import edu.gatech.cs2340.zoo.{Animal, Mammal}` – import `Animal`, `Mammal` but not `Dog` into namespace as simple names.
- `import edu.gatech.cs2340.zoo._` – import all top-level names in `zoo` into namespace as simple names.
- `import edu.gatech.cs2340.zoo.Animal._` – import all members of `Animal` into namespace as simple names.
- `import edu.gatech.cs2340.zoo.{Mammal => FurryCreature}` – import `Mammal` into namespace but rename to `FurryCreature`.
- `import edu.gatech.cs2340.zoo.{Mammal => _, _}` – import everything from `zoo` except `Mammal`.

Georgia Tech

Conclusion

- OOP in Scala is more consistent, more expressive, more flexible, and less verbose than in Java
- With great power comes great resonsibility
  - Don't get too crazy with trait mix-ins, whose linearizations can be difficult to understand
- Stick to Java's conventions for packages and source file organization

Georgia Tech