CS 2340 Objects and Design - Scala

Composition and Inheritance

Christopher Simpkins
chris.simpkins@gatech.edu

OOP 1: Composition and Inheritance

Two ways to define one class in terms of another:

- Composition: one class is composed of other classes
- Inheritance: one class is a subclass of another class

Consider this example:

```
class Agent(val id: Int, val name: String)

class ModularAgent(id: Int,
                   name: String,
                   modules: Seq[Module])
  extends Agent(id, name)
```

- `AfablAgent` is a subclass of `Agent`, or, `AfablAgent` *inherits* from `Agent`.
- `Agent` is *composed* of an `Int` and a `String`
- `AfablAgent` is composed of a `Seq` of `Module`s, and by inheritance also an `Int` and a `String`.

Defining Subclasses

When defining a class with a superclass whose primary constructor takes parameters, must pass those parameters in the `extends` clause.

```
class Agent(val id: Int, val name: String)

class ModularAgent(id: Int,
                   name: String,
                   val modules: Seq[Module])
  extends Agent(id, name)
```

- Becuase they're declared with `val`, `id` and `name` are fields of `Agent` (Parameters declared with `val` or `var` are called *parametric fields* – they define constructor parameters and corresponding fields at the same time.)
- Cannot make `id` and `name` fields in `AfablAgent` – must be parameters to `AfablAgent`'s primary constructor that are passed to `Agent`'s primary constructor in `extends` clause.

Abstract Classes

- A member of a class that has no implementation is called an *abstract* member.
- A class with abstract members is itself abstract and cannot be instantiated directly with `new`

Here, `Agent` is abstract because it has an abstract method:

```
abstract class Agent(val id: Int, val name: String) {

  def getAction(state: State): Action
}
```

Abstract members are recognizable by their lack of implementation (they are *declared* but not *defined*
Abstract classes must have `abstract` modifier in their declaration.

Parameterless Methods

- A *parameterless method* is a method without parameters
- An *empty-paren* method also takes no parameters

The recommended convention is to use a parameterless method whenever there are no parameters and the method accesses mutable state only by reading fields of the containing object (in particular, it does not change mutable state). (Odersky, Spoon, Venners, 2010)

```
abstract class Agent(val name: String, val title: String = "Dr.") {

  def fullName = title+" "+name

  def getAction(state: String): Symbol
}
```

Defined this way, `fullName` could be changed to a field without requiring client code to change because it supports the *uniform access principle*.

Inheritance: Extending Classes

- You extend a class with the `extends` keyword.
- Subclasses inherit all the non-private members of superclasses.

```
abstract class Agent(val id: Int, val name: String, val title: String)
    {

  def fullName = title+" "+name

  def getAction(state: String): Symbol
}
class NormalAgent(name: String)
  extends Agent(name) {

  def getAction(state: String): Symbol = {
    state match {
      case "hungry" => 'eat
      case "tired" => 'sleep
    }
  }
}
```

Now `NormalAgent` is a concrete subclass of `Agent` because it

Overriding Methods and Fields

- If you override a non-abstract member, you need an `override` modifier
- You can override methods with fields because they're in the same namespace
- You can't give methods and fields the same name (as you can with Java)

```
class NormalAgent(name: String)
  extends Agent(name) {

  override def fullName = name

  def getAction(state: String): Symbol = {
    state match {
      case "hungry" => 'eat
      case "tired" => 'sleep
    }
  }
}
```

Polymorphism and Dynamic Binding

- Polymorphism: one concept, many forms
- A superclass represents a concept that has many forms, i.e., subclasses

```
class NormalAgent(name: String) extends Agent(name) {

  def getAction(state: String): Symbol = {
    state match {
      case "hungry" => 'eat
      case "tired" => 'sleep
    }
  }
}
class CrazyAgent(name: String) extends Agent(name) {

  def getAction(state: String): Symbol = {
    state match {
      case "hungry" => 'run
      case "tired" => 'caffeine
    }
  }
}
```

Polymorphism and Dynamic Binding

Method invocations are dynamically bound, meaning they are resolved at run-time.
Here, each variable has static type `Agent`, but the dynamic types are used in method invocations

```
scala> def act(state: String, agent: Agent) = agent.getAction(state)
act: (state: String, agent: Agent)Symbol

scala> act("tired", new NormalAgent("Fred"))
res0: Symbol = 'sleep

scala> act("tired", new CrazyAgent("Barney"))
res1: Symbol = 'caffeine
```

Composition vs. Inheritance

Prefer composition to inheritance

- Inheritance should model an *is-a* relationship. For example, `NormalAgent` is-a `Agent`
- There should be cases where instances of a subclass would be supplied when an instance of a superclasses is expected. For example, in the `act` example from previous slide, `NormalAgent` and `CrazyAgent` subclasses were supplied where an `Agent` was expected.
- Inheritance should be used to specialize behavior, not simply to reuse data structures. For example, if we only wanted to re-use `Agent`'s `name` field, would be better to include our own `name` field, which would be an example of composition

Read the book's discussion of composition and inheritance, and factory methods (which we'll discuss next)

The Factory Pattern

A *factory* makes objects. Here's a factory for Agents:

```
object Agent {
  def apply(name: String) = {
    if (name.trim.toLowerCase == "simpkins")
      new CrazyAgent(name)
    else
      new NormalAgent(name)
  }
}
```

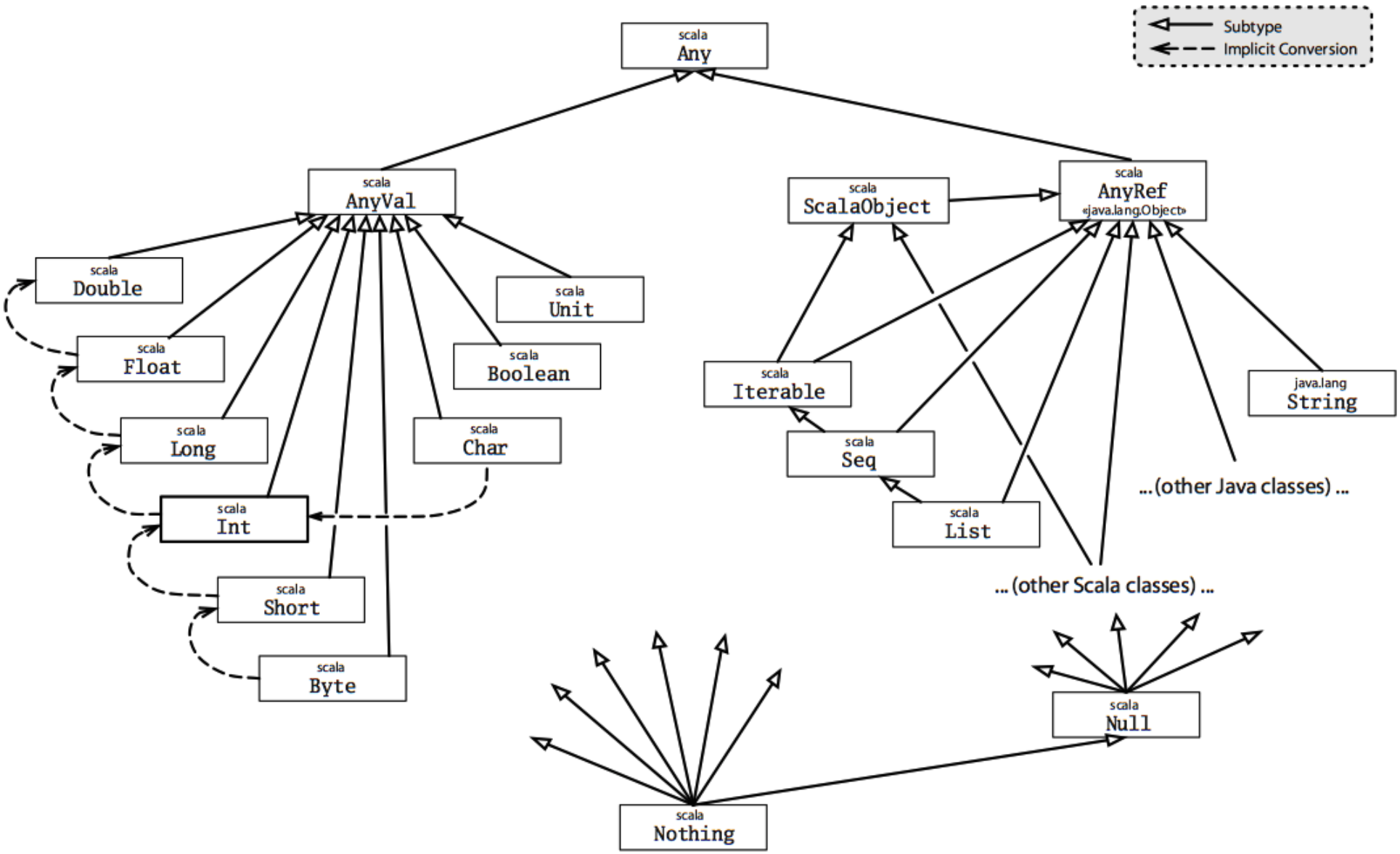
After loading this into the REPL with the other definitions, you can make new Agent objects like this, hiding the concrete implementation of `Agent` subclasses:

```
scala> val a = Agent("simpkins")
a: Agent = CrazyAgent@3a8978c7

scala> val b = Agent("fred")
b: Agent = NormalAgent@44fc63be
```

Note that you have to load this into the REPL in `:paste` mode. Why? (We'll demo at the end of class.)

Scala's Hierarchy



Any and AnyRef

`Any` is root of all Scala objects. `Any` defines:

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

- `==` is implemented in terms of `equals`, which you can override in your classes.
- `AnyRef` is parent of all *reference* classes in Scala. On JVM, `AnyRef` is alias for `java.lang.Object`.
- All Scala classes also inherit from `ScalaObject`

AnyVal

- `AnyVal` is parent of *value* classes: `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Boolean`, and `Unit`
- Can't create instances of value types with `new`; value types are represented as literals
- Primitives are represented as value types at runtime and implicitly converted to "rich" wrappers as needed

For example, `Int` value objects are converted to `RichInt` on-demand to support operations like these:

```
scala> 42 max 43
res4: Int = 43

scala> 1 until 5
res6: Range = Range(1, 2, 3, 4)

scala> 1 to 5
res7: Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> (-3).abs
res9: Int = 3
```

Bottom Types

- Handle's corner cases in Scala type system to make OO and the powerful type system work
- `Null` is a subtype of all `AnyRef`s (but not `AnyVal`s)
- `Nothing` is a aubtype of all Scala types.
- Note that there are no values of type `Nothing` - it's just a marker class.

Consider:

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

The type of the if-branch is `Int` and the type of the else-branch is `Nothing`, which type-checks becuase `Nothing` is a subtype of everything, including `Int`