CS 2340 Objects and Design - Scala

Functional Objects

Christopher Simpkins
chris.simpkins@gatech.edu

Functional Objects

Functional objects have immutable state. Immutable objects:

- are asier to reason about,
- can be passed around without making defensive copies
- are not susceptible to concurrent modification hazard in multi-threaded code
- can safely be used as hash table keys

Primary disadvantage: sometimes require many copies. Libraries often provide mutable versions, e.g. `StringBuilder`, collection classes.

Primary Constructors

- Class parameters passed to primary constructor
- Code between class's { and } that's not a method or field definition is executed as the primary constructor

```
class User1(n: String, d: String) {

  println("Created "+n+"@"+d)
}
```

Now we can construct User1 objects using operator `new`

```
scala> val user1 = new User1("user1", "gatech.edu")
Created user1@gatech.edu
user1: User1 = User1@339d035e
```

Overriding toString

- Override `toString` to make a nicer printed representation
- `toString` defined in `java.lang.Object` superclass, so must use `override modifier`

```
class User2(n: String, d: String) {

  override def toString = n+"@"+d
}
```

Now we don't need the println in the primary constructor

```
scala> val user2 = new User2("user2", "gatech.edu")
user2: User2 = user2@gatech.edu
```

Checking Preconditions

- Use `require`, defined in Predef to check preconditions
- Throws `IllegalArgumentException` if condition fails

```
class User3(n: String, d: String) {

  require(d == "gatech.edu")

  override def toString() = n+"@"+d
}
```

Now Scala won't let us construct invalid objects

```
scala> val user2 = new User2("user2", "gatech.edu")
user2: User2 = user2@gatech.edu

scala> val user3 = new User3("user3", "gatech.edu")
java.lang.IllegalArgumentException: requirement failed
        at scala.Predef$.require(Predef.scala:145)
        at User3.<init>(<console>:9)
         ...
```

Self References

- User `this` to refer to self

Let's define a `hasSameDomainAs` method

```
class User4(n: String, d: String) {

  require(d == "gatech.edu")

  override def toString() = n+"@"+d

  def hasSameDomainAs(other: User4): Boolean = {
    this.d == other.d
  }
}
```

Oops! What's wrong?

```
<console>:14: error: value d is not a member of User4
           this.d == other.d
```

Defining Fields

```
class User5(n: String, d: String) {
  require(d == "gatech.edu")
  val username = n // Field definition
  val domain = d   // Field definition

  override def toString() = n+"@"+d

  def hasSameDomainAs(other: User5): Boolean = {
    this.domain == other.domain
  }
}
```

Now it works

```
scala> val user5 = new User5("user5", "gatech.edu")
user5: User5 = user5@gatech.edu

scala> val user51 = new User5("user51", "gatech.edu")
user51: User5 = user51@gatech.edu

scala> user5 hasSameDomainAs user51
res10: Boolean = true
```

Caution: Methods capture class parameters

```
class User6(n: String, d: String) {
  require(d == "gatech.edu")
  var username = n // Notice the ‘‘var’’
  val domain = d

  override def toString() = n+"@"+d

  def hasSameDomainAs(other: User6): Boolean = {
    this.domain == other.domain
  }
}
```

Probably not the behavior you intended ...

```
scala> val user6 = new User6("user6", "gatech.edu")
user6: User6 = user6@gatech.edu

scala> user6.username = "new"

scala> user6
res5: User6 = user6@gatech.edu
```

Use Field Names Within Methods

```
class User7(n: String, d: String) {
  require(d == "gatech.edu")
  var username = n
  val domain = d

  override def toString() = this.username+"@"+domain

  def hasSameDomainAs(other: User7): Boolean = {
    this.domain == other.domain
  }
}
```

Now it works as expected (but is not functional!)

```
scala> val user7 = new User7("user7", "gatech.edu")
user7: User7 = user7@gatech.edu

scala> user7.username = "new"

scala> user7
res9: User7 = new@gatech.edu
```

Auxilliary Constructors

- Define auxilliary constructors with `this()`
- First action must be invocation of primary constructor or textually preceding auxilliary constructor
- First thing in primary constructor is (possibly implicit) `super()` call

```
class User8(n: String, d: String) {
  require(d == "gatech.edu")
  val username = n
  val domain = d

  def this(n: String) = this(n, "gatech.edu")

  ...
}
```

Now have short cut, but in 2.8/2.9 we'd just use default params

```
scala> val user8 = new User8("user8")
user8: User8 = user8@gatech.edu
```

Defining Operators

```
class User(val username: String, val domain: String = "gatech.edu") {
  require(domain == "gatech.edu")

  override def toString() = this.username+"@"+domain

  def + (other: User): Group = {
    new Group(this, other)
  }
}
```

Now we can “add” users

```
scala> val user1 = new User("user1"); val user2 = new User("user2")
user1: User = user1@gatech.edu
user2: User = user2@gatech.edu

scala> user1 + user2
res1: Group = user1@gatech.edu, user2@gatech.edu
```

Overloaded Methods

Methods with same but different parameter lists are *overloaded*

```
  def + (user: User): Group = new Group(List(this, user))

  def + (group: Group): Group = new Group(this::group.users)
```

With these two overloaded + methods, we can add users or groups

```
scala> val user1 = new User("foo", "gatech.edu")
user1: User = foo@gatech.edu

scala> val user2 = new User("bar", "gatech.edu")
user2: User = bar@gatech.edu

scala> val g = user1 + user2
g: Group = foo@gatech.edu, bar@gatech.edu

scala> val user3 = new User("boo", "gatech.edu")
user3: User = boo@gatech.edu

scala> val g2 = user3 + g
g2: Group = boo@gatech.edu, foo@gatech.edu, bar@gatech.edu
```

Private Members

- Private visibility enforces encapsulation and “hides” inforamtion

`isValid` method only visible within `User` objects

```
require(isValid(username, domain), "Invalid username or domain")

private def isValid(name: String, domain: String): Boolean = {
  domain match {
    case "gatech.edu" => true
    case "gatech.edu" => true
    case _ => false // All other domains invalid
  }
}
```

With new `isValid` method, we can easily add validity checks.

Conclusion

- State of an object is defined by its instance variables, a.ka. fields
- Functional objects have immutable state
- Methods define behavior of objects, i.e., the “messages” an object responsds to
- Operators are methods that can be defined to make object usage more natrual - use judiciously!
- Methods and operators can be overloaded to work with arguments of different types
- Can use private members to enforce encapsulation and information hiding