CMPSCI 220 Programming Methodology

Traits and Packages

Based on Head First Design Patterns

Objectives

Traits

- Learn how traits work
- Learn the difference between thin and rich interfaces
- Learn about stackable modifications and linearization
- Learn why multiple inheritance is bad

Packages and Imports

- Learn about putting code into packages
- Learn about concise access to related code
- Learn about access modifiers

Traits are Interfaces

```
trait Philosophical {
  def philosophize(): Unit
}
```

Traits are like Classes

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
```

Traits can be “Mixed In”

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}


class Frog extends Philosophical {
  override def toString = "green"
}
```

Traits can be “Mixed In” and Used

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
```

```
class Frog extends Philosophical {
  override def toString = "green"
}
```

```
scala> val frog = new Frog
frog: Frog = green

scala> frog.philosophize()
I consume memory, therefore I am!
```

Traits can be “Mixed In” and Used

```
trait Philosophical {
  def philosophize() {
    println("I consume memory, therefore I am!")
  }
}
```

```
class Frog extends Philosophical {
  override def toString = "green"
}
```

```
scala> val frog = new Frog
frog: Frog = green

scala> frog.philosophize()
I consume memory, therefore I am!

scala> val phil: Philosophical = frog
phil: Philosophical = green

scala> phil.philosophize()
I consume memory, therefore I am!
```

Traits can be used with Classes

```
class Animal

class Frog extends Animal with Philosophical {
  override def toString = "green"
}
```

Traits can be used with Classes/Traits

```
class Animal
trait HasLegs

class Frog extends Animal with Philosophical with HasLegs {
  override def toString = "green"
}
```

Thin Interfaces

A “thin” interface is one that provides fewer methods requiring less to implement.

This is good from the Java perspective as interfaces can’t provide implementation.

Rich Interfaces

A “rich” interface provides many methods requiring more to (possibly) implement.

This is bad from the Java perspective as interfaces can’t provide implementation.

For Scala, however, traits can provide implementation that can be used in terms of the class extending it.

Example

src/main/scala/cs220/graphics/Shapes.scala

Traits: Stackable Modifications

Imagine we want to implement a Queue library. This is simple enough – we all know how queues are implemented.

What if…
We wanted to be able to create similar (but different) variants that allow for queues with slightly different behaviors?

Queue Library: Java

What if we wanted to implement integer queue variants that can have one or more of the following behaviors:

- doubling: doubles the element on put
- increment: increments the element on put
- filtering: filters out negative integers on put

How might you do this in a language such as Java?

Queue Library: Scala

What if we wanted to implement integer queue variants that can have one or more of the following behaviors:

- doubling: doubles the element on put
- increment: increments the element on put
- filtering: filters out negative integers on put

How might you do this in a language such as Scala?

Example

src/main/cs220/queue/Queue.scala

Multiple Inheritance?

Traits are a way to inherit from multiple class-like constructs, but they differ from other languages that provide multiple inheritance.

- The interpretation of *super* is different
  In Scala, *super* depends on a *linearization process* which allows traits to be “stacked”

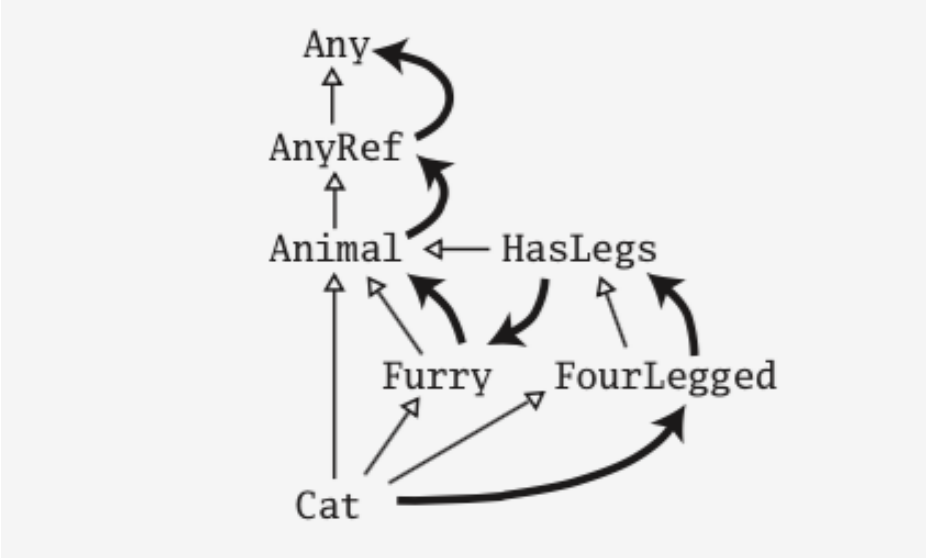
In other languages *super* is chosen for only one of the multiple classes that are inherited.

Animals

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Animals Linearization

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```



To trait, or not to trait?

- **If the behavior will not be reused** make it a concrete class.
- **If it might be reused in multiple, unrelated classes** make it a trait
- **If it is distributed as a compiled library** make it an abstract class
- **If efficiency is crucial** make it a class

Like any other programing language construct it takes practice and experience to know when it should be used or not.

When building libraries it is important to know which language construct to use to benefit clients and to simplify the use of the library.

SimUDuck

Think back to the problem we had with creating ducks that could Quack, Fly, etc. How might you use stackable traits to achieve a better design for your employer?