OO Support

OO Paradigm

- Encapsulation and Polymorphism are the most important features
- Abstraction is fundamental
- Most languages support Inheritance
- Inheritance is not needed for polymorphism, though that is the impression you’ll get if you are used to languages like C++, Java, C#
How OO is the Language?

- Depends on who you ask!
- C++ is hybrid—primitives, global functions
- Java is adamant about not allowing global functions, but still has primitives and static methods
- Scala does not require methods to belong to classes, but wraps them up behind the scenes. In Scala, like Smalltalk, everything is an object

Polymorphism

- Send a message to an object, the appropriate message gets called based on the actual type of the object, not the type of the reference
Polymorphism in Static Langs

- Often realized using inheritance
- Though class or interface inheritance

Multi-methods

- Polymorphism on steroids
- The method is based also on the actual type of parameters at runtime!
Try this in Java & Groovy

```java
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(1); list.add(2); list.add(3);
System.out.println(list.size());
list.remove(0);
System.out.println(list.size());
```

```java
Collection<Integer> list = new ArrayList<Integer>();
list.add(1); list.add(2); list.add(3);
System.out.println(list.size());
list.remove(0);
System.out.println(list.size());
```

Multiple Inheritance

- Often associated with Pain
- In languages like C++ leads to method collision
- Related complexity of Virtual Base Class to eliminate duplication of object appearance
- Java/C# completely avoid it
- Languages like Scala take a different approach
Method Collision in Interfaces

- What if method names in interfaces collide?
- Java has no good answer—you can have only one implementation
- C# provides explicit interfaces

Traits in Scala

- Scala Traits are like interfaces with partial implementation
- You can mixin the traits into your classes
Traits—Cross Cutting Concerns

```
class Human(val name: String)
{
  def listen =
    println("I'm " + name + " your friend. I'm listening...")
}

class Man(override val name: String) extends Human(name)

val sam = new Man("Sam")
sam.listen

//Friend is not modeled well
//Not clear
//Hard to reuse

  Traits can help here
  Think of them as interfaces with partial implementations

trait Friend
{
  val name : String //abstract
  def listen =
    println("I'm " + name + " your friend. I'm listening...")
}

class Human(val name: String)

class Man(override val name: String)
    extends Human(name)
    with Friend

class Dog(val name: String) extends Friend
{
  override def listen =
    println("Your friend " + name + " listening...")
}

def help(friend: Friend) { friend.listen }

def help(friend: Friend) { friend.listen }
help(new Man("Sam"))
help(new Dog("Casper"))
```
Traits—Cross Cutting Concerns

Not just at class level

class Cat(val name: String)

help(new Cat("Sally") with Friend)

Traits and Method Chaining

Traits avoid method conflicts by using method chaining
Class methods

- Java (C++, C#, Ruby, ...) supports class methods
- Scala, on the other hand does not allow that
- However, it provides singleton objects and companion classes