What if we wanted a category of trainable animals, like this?

Creating an Interface

```java
public interface Trainable{
    public void sit();
    public void stand();
}
```

- An interface is like a class except:
  - The keyword `interface` is used instead of `class`
  - All of its methods are body-less
  - It has no instance variables
  - It can not be instantiated (Illegal: new Trainable())
- An interface is like a contract, protocol, role, or point of view
- Code written for an interface type works with any object whose class implements it
  - It can assume that all of the subtypes have the methods listed in the interface (e.g. sit and stand)
A Class implements an Interface

```java
public class Dog implements Trainable{
    public void sit(){ // code for sit method }
    public void stand(){ // code for stand method }
}
```

A class that implements an interface
- Must provide concrete methods for each interface method
- May have additional methods
- May implement multiple interfaces

```java
public class Dog implements Trainable, Comparable{
    // code
}
```

Abstract class vs. Interface

- When a partial implementation is feasible, abstract classes make sense as they can provide some functionality with the methods
- Pure abstract classes (with all abstract methods) in Java are functionally equivalent to an interface, but restricted to single inheritance
- Java will allow you to implement more than one interface
  - Multiple interface inheritance allows an object to inherit many different method signatures with the caveat that the inheriting object must implement those inherited methods.
- You can use access modifiers (e.g. protected) in an abstract class though. Interfaces are always public.

Example: Trainable Interface

```java
> Lion lion = new Lion();
> Dog dog = new Dog();
> Trainable beast;
> beast = lion; // beast is of type Trainable pointing to lion object
> beast.sit(); // calls lion's sit method
> beast = dog;
> beast.sit(); // calls dog's sit method
```

Examples of Interfaces

- Interfaces are typically "lightweight", with only a few method headers
- Interfaces are intended to be extended, not instantiated

```java
public interface Displayable{
    public void display();
}
```

```java
public interface XYZProtocol {
    public void connect();
    public void disconnect();
    public int receive();
}
```
**instanceof Operator with Interfaces**

- **Usage:** `lvalue instanceof T`
- The expression is true if `lvalue` has type `T` or is a subtype of `T`.
- **Example:**
  ```java
  public class Rectangle implements Displayable { .. }
  public class Square extends Rectangle { .. }

  Rectangle r = new Rectangle(2,3);
  r instanceof Rectangle      true
  r instanceof Displayable    true
  r instanceof Square        true
  r = new Square(10);
  r instanceof Square        true
  ```

**A Design Problem**

- What if we want to write graphics code for all the animals, and want to require that they all have a `display()` method? (There are many ways to accomplish this.)

**One of Many Solutions**

- **Extending and Implementing**
  - Class can only extend one other class
  - But can implement multiple interfaces
  - The exact order must be used as shown below in declaring a class below:
    ```java
    public class SubclassName extends SuperclassName implements Interface1, Interface2 {
    }
    ```
  - Otherwise there is compile error
# Summary

<table>
<thead>
<tr>
<th></th>
<th>Regular Class</th>
<th>Abstract Class</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Methods</strong></td>
<td>all concrete</td>
<td>concrete and/or abstract</td>
<td>all abstract</td>
</tr>
<tr>
<td><strong>May have instance variables</strong></td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><strong>May be instantiated</strong></td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>