2. Interfaces :: Common Behaviors in Unrelated Classes

In Section 1, we saw that objects of classes which are all related to a common superclass can possess similar (common) behavior (*Mammal*s making a sound) although that behavior can be class-specific (*Dog*s bark, *Cat*s meow). This is accomplished by declaring a superclass method that is inherited and overridden by the subclasses.

However, often in an OO design we desire an object of a class to implement some class-specific behavior that is similar to (or in common with) the behaviors of objects of **unrelated classes**, i.e., the classes do not belong to the same inheritance hierarchy. For example:

```
public abstract class Mammal {     // Abstract class.
  public void makeSound();         // Abstract method. All Mammals make a sound.
}
public class Dog extends Mammal {  // Dog is a subclass of Mammal.
  @Override
  public void makeSound() {        // Overrides makeSound() inherited
    System.out.println("Bark");    // from Mammal. Dogs bark.
  }
}
public class Cat extends Mammal {  // Cat is a subclass of Mammal.
  @Override
  public void makeSound() {        // Overrides makeSound() inherited
    System.out.println("Meow");    // from Mammal. Cats meow.
  }
}
```

2. Interfaces :: Common Behaviors in Unrelated Classes (continued)

```
public abstract class Insect {          // Abstract class.
  public void makeSound();              // Abstract method. All insects make sounds.
}

public class Cricket extends Insect {   // Cricket is a subclass of Insect.
  @Override
  public void makeSound() {             // Overrides makeSound() inherited
    System.out.println("Chirp");        // from Insect. Crickets chirp.
  }
}
```

The first thing to notice in this code is that we **declared the same abstract method** *makeSound*() in both the *Mammal* and *Insect* classes—it even has the same signature. In general, in programming we like to avoid copying-and-pasting code or duplicating code in multiple places. The fact that *Mammal* and *Insect* both declare the same abstract method bothers me.

We can summarize this problem: when objects of classes—whether those classes are related or unrelated—need to implement similar, but object-specific behavior, it is desirable that the declaration of that common behavior only appear once in the code.

2. Interfaces :: Common Behaviors in Unrelated Classes (continued)

There is another problem with *Mammal* and *Insect* both declaring the same abstract class. Suppose we wish to call *beNoisy*() passing *Dog*s, *Cat*s, and *Insect*s as arguments to it, because at some point in the application all of those critters need to make the appropriate sound.

```
public void beNoisy(Mammal pCritter) {
  ...
  pCritter.makeSound();  // This is a polymorphic method call.
  ...
}
```

So we write:

```
public void someMethod() {
  Dog spot = new Dog();
  Cat bucky = new Cat();
  Cricket jiminy = new Cricket();
  beNoisy(spot);                  // Bark, Spot, bark.
  beNoisy(bucky);                 // Meow, Bucky, meow.
  beNoisy(jiminy);                // Chirp, Jiminy, chirp.
}
```

This code will not compile and the reason is that *jiminy* is a *Cricket* and not a *Mammal*. We can only substitute objects of subclasses of *Mammal* for *pCritter* in *beNoisy*().

2. Interfaces :: Common Behaviors in Unrelated Classes (continued)

To fix this code we would have to write a separate *beNoisy*() method for *Insect*s:

```
public void beNoisy(Insect pCritter) {
   ...
   pCritter.makeSound();  // This is a polymorphic method call.
   ...
}
```

Where the ... part would be the same in *beNoisy*(*Mammal*) and *beNoisy*(*Insect*). This is not good.

2. Interfaces :: Common Behaviors in Unrelated Classes (continued)

To summarize the situation:

1. All *Mammal*s and *Insects* make sounds.
2. Each type of *Mammal* (*Dog*, *Cat*) makes a sound in its own unique way by overriding *makeSound*() inherited from *Mammal*.
3. Each type of *Insect* (*Cricket*, ...) makes a sound in its own unique way by overriding *makeSound*() inherited from *Insect*.
4. It does not make sense to declare the same abstract *makeSound*() method in both *Mammal* and *Insect* (in general, we do not like to duplicate code).
5. Even if Item 4 were not such a bad idea, we cannot pass a *Cricket* to *beNoisy*(*Mammal*) because the *Cricket* and *Mammal* classes are **not related**. A *Cricket* object cannot be substituted for a *Mammal* because a *Cricket* is not a *Mammal*.
6. The solution was to copy-and-paste *beNoisy*(*Mammal*) and change the type of the input parameter to *Insect*.
7. Any time you find yourself copying-and-pasting code because you need to do (almost) the same thing in a different location there is usually a better way to achieve your goal.

Therefore, what we need is a way to:

1. Specify that objects of **unrelated classes** implement the **same behavior**, but each in its own unique way based on the class of the object ...
2. so that we may pass those objects (of unrelated classes) as parameters to methods ...
3. so those methods may make **polymorphic method** calls on the objects.