More on objects and inheritance

How to define **equals**

We just discussed how every class we write is automatically a child class of the `Object` class, and that the `Object` class has a method called `equals`. You are supposed to override this class with a proper definition. Sometimes you might forget, because if you use built-in classes, the equals method does what you would expect.

For example consider the following code:

```
          ArrayList<String> list = new ArrayList<>();
          list.add("foo");
          list.add("bar");
          list.add("zot");

          System.out.println(list.contains("bar"));
          System.out.println(list.contains("uaa"));
```

This outputs “true” and “false” as you would expect. How does the contains() method work? It scans through each element in the ArrayList and called equals to see if it returns true when compared to “bar” and “uaa”. It works because someone wrote the method as part of the String class that correctly compares the characters in the strings to see if they match.

What if we try this with our own class?

```
     class Myclass
     {
          private int val;
          public Myclass()
          {
               val = 0;
          }
          public Myclass(int newVal)
          {
               val = newVal;
          }
     }

     public static void main(String[] args)
     {
          Myclass class1 = new Myclass(1);
          Myclass class2 = new Myclass(1);

          ArrayList<Myclass> list = new ArrayList<>();
          list.add(class1);
          list.add(class2);
```

```
        System.out.println(list.contains(new Myclass(1)));
        System.out.println(list.contains(class2));
    }
```

The contains method doesn't work on a new Myclass(1), even though the classes contain basically the same thing. It does work when we send in class2. This is because we didn't define our own equals method for our class, so Java just compares the addresses in memory where each object is stored to see if they match. The only object where the addresses match is with class2.

To fix this problem, we override the equals method.

First, what is the difference between these two methods?

```
    public boolean equals(Myclass otherclass)
    {
        return otherclass.val == this.val;
    }

    public boolean equals(Object otherclass)
    {
        Myclass otherObj = (Myclass) otherclass;
        return otherObj.val == this.val;
    }
```

The two equals methods have different parameter types, so the first one actually doesn't override the definition of equals defined at the Object level. We have merely overloaded the method equals. The class now has two methods named equals. If passed in a Myclass object, it will use the first definition.

With the second definition, we will get "true" returned for both tests for contains. This is because Java will now call the equals method for each element in the array and we get true when the values are the same. Note that we have to "downcast" from Object to Myclass. If we were given something that isn't really a Myclass object, then Java will likely crash (in general it is safe to "upcast").

To flesh out our implementation more, we should check to make sure that a Myclass object is passed in and that it is not null. If it's null we will get an error trying to access .val:

```
    public boolean equals(Object otherclass)
    {
        if (otherclass == null)
            return false;
        else if (getClass() != otherclass.getClass())
            return false;
        else
        {
            Myclass otherObj = (Myclass) otherclass;
            return otherObj.val == this.val;
        }
    }
```

The getClass() method returns a representation of the class used to create the object. This method is marked final, which means it can't be overridden. The return value can be used with == and !=.

There is a similar method, instanceof, that tells you if an object is an instance of another class. For example:

```
        else if (!otherclass instanceof Myclass)
                return false;
```

The only problem with this is that if you have a class and subclass (like in our Candy and Twix example) and pass a Twix object in for a Candy object (e.g. candy.equals(twix)) then this will return false since twix is not an instance of a Candy object.

Class Exercise: Given the following code, re-write it to use principles of inheritance and polymorphism.

Question: everyone familiar with generics and for each loop?

```
import java.util.ArrayList;

class Alien
{
    public static final int ROBOT_ALIEN = 0;
    public static final int OCTOPUS_ALIEN = 1;
    public static final int MARSHMALLOW_MAN_ALIEN = 2;

     public int type;

     public Alien(int type)
     {
          this.type = type;
     }
}

class PackOfAliens
{
     private ArrayList<Alien> al;

     public PackOfAliens()
     {
          al = new ArrayList<Alien>();
     }

     public void addAlien(Alien a)
```

```
	{
		al.add(a);
	}

	public ArrayList<Alien> getAliens()
	{
		return al;
	}

	public int calculatePackDamage()
	{
		int s = 0;
		for (Alien a : al)
		{
			if (a.type==Alien.ROBOT_ALIEN) {
				s+=10;		// Robot does 10 damage
			}
			else if (a.type==Alien.OCTOPUS_ALIEN) {
				s+=6;	// Octopus does 6 damage
			}
			else {
				s+=2;	// Marshmallow does 2 damage
			}
		}
		return s;
	}
}


public class Test
{
	public static void main(String[] args)
	{
		Alien r1 = new Alien(Alien.ROBOT_ALIEN);
		Alien r2 = new Alien(Alien.ROBOT_ALIEN);
		Alien o = new Alien(Alien.OCTOPUS_ALIEN);
		Alien m  = new Alien(Alien.MARSHMALLOW_MAN_ALIEN);
		PackOfAliens p = new PackOfAliens();
		p.addAlien(r1);
		p.addAlien(r2);
		p.addAlien(o);
		p.addAlien(m);
		System.out.println("Pack damage is " +
p.calculatePackDamage());
	}
}
```

Abstract Classes

Sometimes there are classes where you should never make an instance of the class. For example, in our animal hierarchy of moose and deer, there is a parent class of Cervid. However, in nature, there is no animal that exists that is a cervid. Only child classes exist.

The same thing goes for our Candy hierarchy. There is never a “Candy” object that exists in the real world, only specific instances of candy, like Twix or Peanut Butter cups. The Candy class is an idealized abstraction, not a physical reality.

When we have this type of abstraction we can denote it by specifying the **abstract** keyword:

```
public abstract class Candy
{
      …
}
```

If a class is abstract then Java does not allow us to create an instance of the class.

We can also define a specific method to be abstract. This makes sense when we can’t really define the method at the parent level, but it makes sense at a lower level. An example of this was with our computeNumTwix() method that computed how many twix we needed to eat to survive. We don’t know how to calculate this at the Person class, but need to calculate it at the Man or Woman class. To specify an empty placeholder for the method, it would look like this in the Person class:

```
public abstract int computeNumTwix();
```

A class that has at least one abstract method is an abstract class, and must have the abstract keyword added to the class. An abstract class can have multiple abstract methods, but can also have concrete, filled-in methods. An abstract method cannot be private.