Additional inheritance example

As another example of inheritance, perhaps we would like to build an application about candy. For starters, let’s say we want to do something with Twix bars and something with Reese’s Peanut Butter Cups. We might make classes like the following:

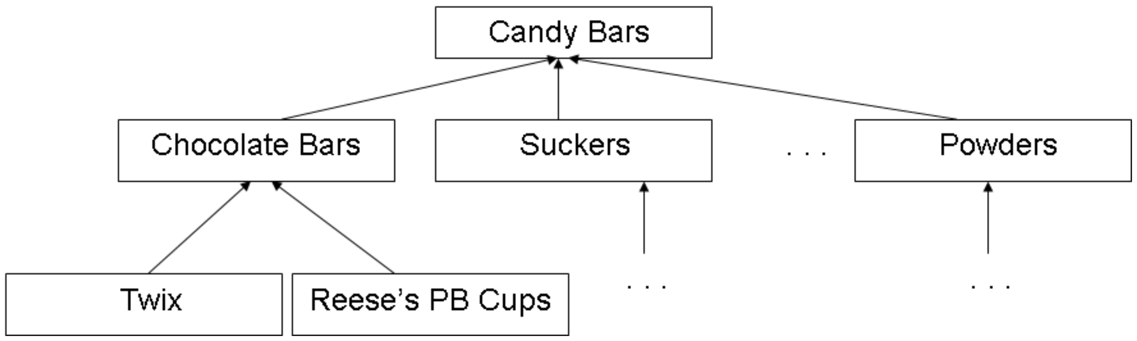
```
public class Twix
{
      private int calories;
      private String ingredients = "";

      public Twix()
      {
            calories = 285;
            ingredients = "chocolate, sugar, cookie, caramel";
      }

      public String getInfo()
      {
            return "Calories: " + calories + ". Ingredients: " + ingredients;
      }
}

public class PBCups
{
      private int calories;
      private String ingredients = "";

      public PBCups()
      {
            calories = 210;
            ingredients = "chocolate, sugar, peanut butter";
      }

      public String getInfo()
      {
            return "Calories: " + calories + ". Ingredients: " + ingredients;
      }
}
```

You should already be familiar with how one might use these classes. For example, the following code creates two candy bars and prints their info:

```
    public static void main(String[] args)
    {
          Twix t = new Twix();
          PBCups p = new PBCups();

          System.out.println(t.getInfo());
          System.out.println(p.getInfo());
    }
```

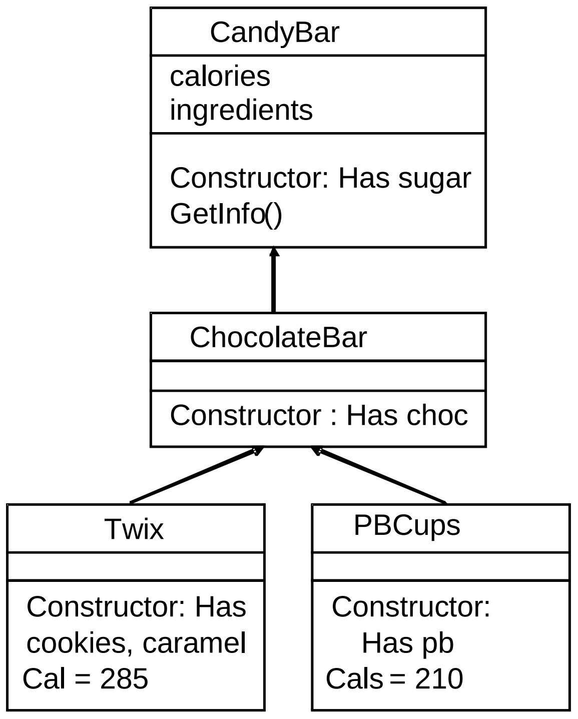
This might be fine for some applications, but right off the bat we can see that we are duplicating a lot of the same code. For example, the getInfo() subroutine is going to be almost the same for any candy bar. As the program is now, if we had 100 different candy bars, we would have 100 different getInfo() subroutines.

Instead, we can take advantage of a natural ordering of candy bars. We can visualize the types of candy bars in a hierarchy as follows:



A Twix Bar is a Chocolate Bar which in turn is a Candy Bar. This means that a Twix bar has all the properties that Chocolate Bars have, which in turn have all the properties that Candy Bars have.

Here is an example of the inheritance hierarchy for the Twix, PBCups, ChocolateBar, and CandyBar classes:



In this example, we define the calories and ingredients variables in the CandyBar class since these are variables that apply to any Candy Bar. These variables are inherited by all classes below it, so classes like Twix, PBCups, and ChocolateBar automatically get access to the variables without having to redefine them.

At the CandyBar level we also have a method, getInfo(). It returns a string of the calories and ingredients. It is also accessible by any class defined below it, so the method only exists in one place.

When we create an object, the constructors for all the parent classes will also be invoked. Consequently, when we make a Twix object, Java will first invoke the constructor for CandyBar, then the constructor for ChocolateBar, and finally the Twix constructor would be last.

```
public class CandyBar
{
	protected int calories;
	protected String ingredients = "";

	public CandyBar()
	{
		ingredients = "sugar";
	}

	public String getInfo()
	{
		return "Calories: " + calories + ". Ingredients: " + ingredients;
	}
}
```

```
public class ChocolateBar extends CandyBar
{
      public ChocolateBar()
      {
            ingredients = ingredients + ", chocolate";
      }
}
public class Twix extends ChocolateBar
{
      public Twix()
      {
            ingredients = ingredients + ", cookie, caramel";
            calories = 285;
      }
}
public class PBCups extends ChocolateBar
{
      public PBCups()
      {
            ingredients = ingredients + ", peanut butter";
            calories = 210;
      }
}
```

Here we can run the same code to create instances of the objects as before:

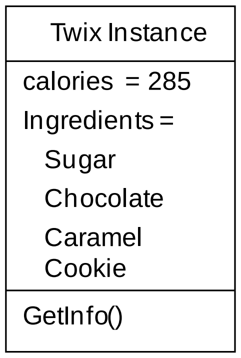
```
      public static void main(String[] args)
      {
            Twix t = new Twix();
            PBCups p = new PBCups();

            System.out.println(t.getInfo());
            System.out.println(p.getInfo());
      }
```

The output is identical to before, but you can see that it is much easier now to add new chocolate bars or candy bars without duplicating code.

When we create an instance of a Twix object (via Twix t = new Twix()) here is what happens:

1. The twix object inherits its own variables of "calories" and "ingredients"
2. The parent constructors are called first:
    a. CandyBar's constructor adds "sugar" to ingredients
    b. ChocolateBar's constructor adds "chocolate" to ingredients
    c. Twix's constructor adds "caramel" and "cookie" to ingredients and sets calories to 285

Graphically, the Twix object looks something like this:



Invoking GetInfo() outputs all of the ingredients and calories for each object. Since GetInfo() is defined only in the CandyBar class, this eliminates repeat code since all child objects share the same code base.

It is sometimes desirable to override the definition of a method in a parent class. Here is a version of the Twix class where we have defined a new version of getInfo:

```
public class Twix extends ChocolateBar
{
      public Twix()
      {
            ingredients = ingredients + ", cookie, caramel";
            calories = 285;
      }

      @Override
      public String getInfo()
      {
                  return "Need a moment?  Two for me, none for you! " +
                         super.getInfo();
      }
}
```

The @override directive is optional and is for people to see that the method is overriding a method of the same name in a parent class. The line super.getInfo() invokes the getInfo() method from the parent class. If we run our code in main, we now get the output:

```
Need a moment? Two for me, none for you! Calories: 285, Ingredients:
sugar, chocolate, cookie, caramel
```