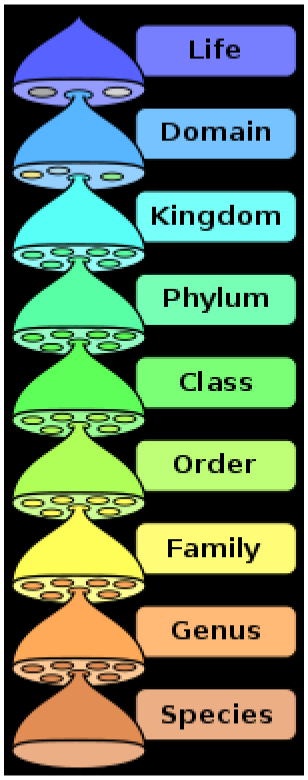
Inheritance

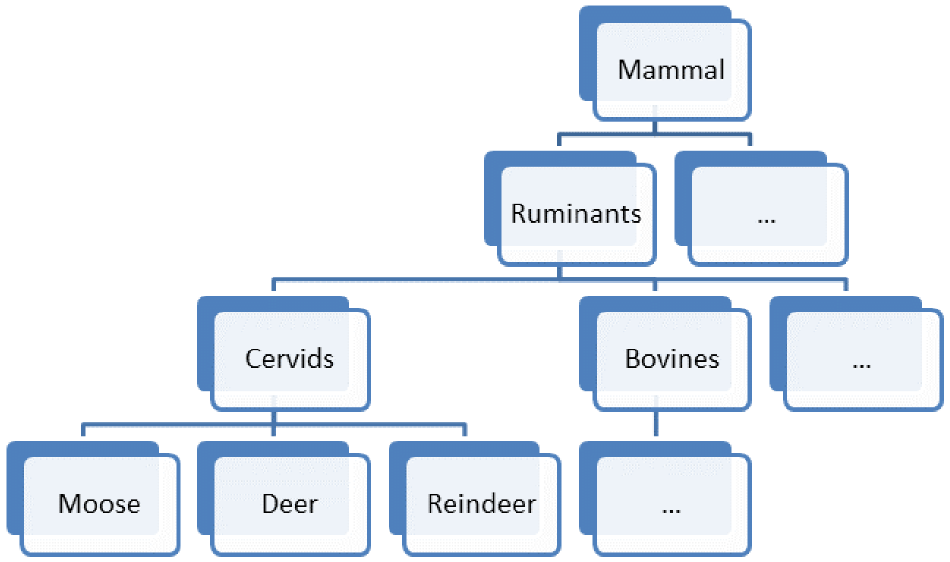
The concept of inheritance is one of the key features of an object-oriented programming language. Inheritance allows a programmer to define a general class, and then later define more specific classes that share or *inherit* all of the properties of the more general class. This allows the programmer to save time and energy that might otherwise be spent writing duplicate code.

Related to inheritance is the concept of *polymorphism*. Polymorphism allows us to invoke the "correct" method in an inheritance hierarchy. We'll see how this is used later.

The idea of inheritance is similar to the taxonomy of living things. You probably recall the following hierarchy from a biology class:



For example, to zoom in on the deer family (cervid), we could make a tree like the following.



Note that this hierarchy is an "IS-A" hierarchy. A deer "IS-A" cervid. In turn, a cervid "IS-A" ruminant, and so forth. The term "parent" is the term used to describe concepts going up the hierarchy, and "child" is the term when going down the hierarchy (imagine a family tree). For example, a deer is a "child" of cervid, and cervid is the "parent" of deer. Note that any property that holds for a parent also holds for a child. For example, if we know that mammals are warm-blooded, then we know that everything listed below mammal is also warm-blooded. We'll take advantage of this concept from a programming perspective to reduce duplicative code.

As an example, recall the twix bar calculator we wrote earlier in the semester. Here was the formula for men to compute how many twix bars are needed to be eaten to maintain one's weight:

```
calories =  10 * kilograms + 6.25 * centimeters - 5 * age + 5;
numTwix = calories / 568;
```

For women we had the formula:

```
calories =  10 * kilograms + 6.25 * centimeters - 5 * age -  161;
numTwix = calories / 568;
```

If we're making an object-oriented version of our twix calculator then it seems reasonable to make a `Man` class and a `Woman` class. Here is how we might create each one:

```java
public class Man
{
      private int age; // Private instance variables describing the man
      private int weight; // Normally there would be accessors/mutators
      private int feet;   // In this case, I only made one for age
      private int inches;

      // Default values in the default constructor
      public Man()
      {
            age = 20;
            weight = 150;
            feet = 5;
            inches = 8;
      }

      // Constructor that sets all the instance variables
      public Man(int age, int weight, int feet, int inches)
      {
            this.age = age;
            this.weight = weight;
            this.feet = feet;
            this.inches = inches;
      }

      // Accessor for age
      public int getAge()
      {
            return age;
      }
      // Mutator for age
      public void setAge(int newAge)
      {
            this.age = newAge;
      }

      // Compute how many twix bars to maintain weight
      public double howManyTwix()
      {
            double kilograms = weight / 2.2;
            double centimeters = ((feet * 12) + inches) * 2.54;

            double calories =  10 * kilograms + 6.25 *
                               centimeters - 5 * age + 5;
            double numTwix = calories / 568;    // 568 cals per pack

            return numTwix;
      }
}
```

We might use this class in something like the following, which could be in a main method. It makes a new "Man" object of age 40, weight 140, that is 5 feet 6 inches tall, then outputs the age and how many twix bars the man needs to eat.

```
	Man kenrick = new Man(40, 140, 5,6);
	System.out.println("This person of age " + kenrick.getAge() +
	               " should eat " + kenrick.howManyTwix() +
	               " twix bars.");
```

The output is:

```
	This person of age 40 should eat 2.62 twix bars.
```

We can create a `Woman` class in a similar manner:

```
public class Woman
{
	private int age;
	private int weight;
	private int feet;
	private int inches;

	// Default values in the default constructor
	public Woman()
	{
		age = 20;
		weight = 120;
		feet = 5;
		inches = 6;
	}

	// Constructor that sets all the instance variables
	public Woman(int age, int weight, int feet, int inches)
	{
		this.age = age;
		this.weight = weight;
		this.feet = feet;
		this.inches = inches;
	}

	// Accessor/Mutator for age
	public int getAge()
	{
		return age;
	}
	public void setAge(int newAge)
	{
		this.age = newAge;
	}

	// Twix calculation
	public double howManyTwix()
	{
		double kilograms = weight / 2.2;
```

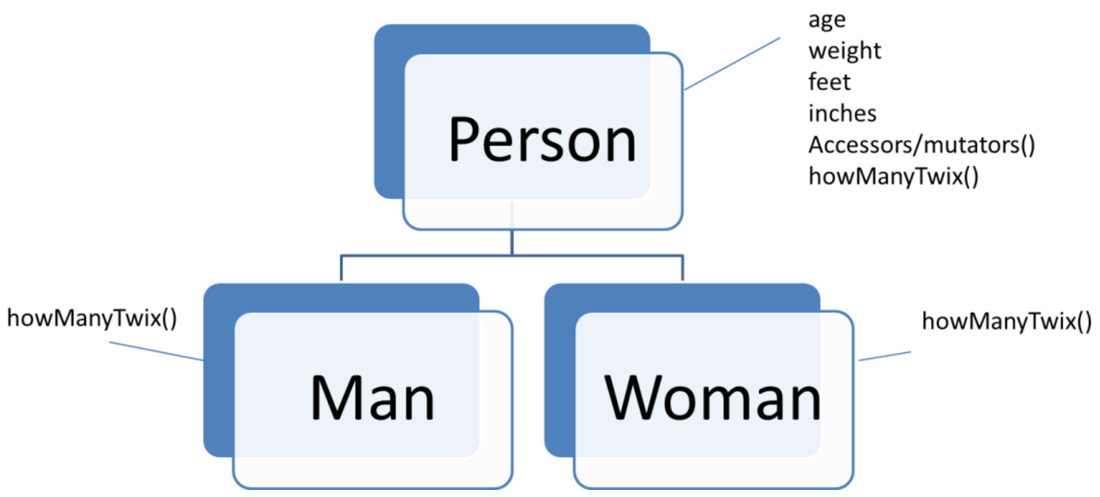
```
                double centimeters = ((feet * 12) + inches) * 2.54;

                double calories =  10 * kilograms + 6.25 * centimeters -
                                   5 * age - 161;
                double numTwix = calories / 568;

                return numTwix;
        }
}
```

These two classes will work fine, but it's totally inefficient. The code is almost identical except for some values in the default constructor and the formula for how twix bars are calculated. There are various approaches we could take, such as creating a method that takes a parameter indicating man or woman, but if want to keep the object structure then inheritance lets us eliminate lots of duplicate code.

In this case we'll make a parent class called `Person` and define the `Man` and `Woman` classes as children of the `Person` class. In OOP terminology, `Person` is called the **base** class and `Man` and `Woman` are called the **derived** classes. Derived classes will inherit all of the public (or protected) methods and instance variables of its parents so we don't have to write these more than once!



Private instance variables defined in a parent class exist in derived classes, but can't be accessed directly by name. Instead they must be accessed through a method. Private methods defined in a parent class can't be accessed directly from a derived class.

To create a derived class, we add the keyword **extends** to the class definition, followed by the name of the base class. E.g.:

```
        public class Man extends Person
```

Here is the first version of our new class for `Person`. We only put methods and instance variables in the creature class if they are applicable to all sub-classes.

```
public class Person
{
      private int age;
      private int weight;
      private int feet;
      private int inches;

      // Default values in the default constructor
      public Person()
      {
            age = 20;
            weight = 150;
            feet = 5;
            inches = 8;
      }

      // Constructor that sets all the instance variables
      public Person(int age, int weight, int feet, int inches)
      {
            this.age = age;
            this.weight = weight;
            this.feet = feet;
            this.inches = inches;
      }

      // Accessor for age
      public int getAge()
      {
            return age;
      }
      // Mutator for age
      public void setAge(int newAge)
      {
            this.age = newAge;
      }

      // Don't know so just return 0
      // This makes the method just a placeholder
      public double howManyTwix()
      {
            return 0;
      }
}
```

This basically contains all of the common code from the original version of the `Man` and the `Woman` class. We made `howManyTwix()` just return 0 since we don't know which formula to use for a generic `Person`. Later we'll see a way to define this as a true placeholder (it's called an abstract method).

Here is the beginning of the `Man` class:

```
public class Man extends Person
{
      // Default constructor
      public Man()
      {
            super();    // This invokes the default constructor
                        // of the base class.  It is done automatically
                        // if you leave it off.
      }

      // Constructor taking arguments
      public Man(int age, int weight, int feet, int inches)
      {
            super(age, weight, feet, inches); // Invoke
                  // the parent constructor that takes arguments
      }
}
```

If we run the same sample code as before:

```
      Man kenrick = new Man(40, 140, 5,6);
      System.out.println("This person of age " + kenrick.getAge() +
                    " should eat " + kenrick.howManyTwix() +
                    " twix bars.");
```

The output is:

```
      This person of age 40 should eat 0.0 twix bars.
```

In the constructor for the `Man` class we are first invoking the method `super( )`. This invokes the default constructor for the base class, `Person`. If we do this, it must be the first line in the constructor. If we leave this line out, the default constructor of the base class gets invoked anyway, so often programmers will just leave this out. Note that we do not use the name of the constructor, instead we use the keyword `super`.

The second constructor takes four parameters, one for age, weight, feet, inches. To invoke the base class constructor with four arguments we use `super( )` again but fill in the arguments: `super(age, weight, feet, inches)`.

Note what we've done – we greatly simplified the `Man` class and even though there is no `getAge()` method defined, we can invoke it! Similarly for `howManyTwix()`, we can invoke that too, although it doesn't give us the right answer (yet). We get access to these methods because they are inherited from the parent, or base class.

If we try to add the howManyTwix() code to the Man class we'll get an error:

```
public double howManyTwix()
{
      double kilograms = weight / 2.2;
      double centimeters = ((feet * 12) + inches) * 2.54;
      double calories =  10 * kilograms + 6.25 * centimeters –
                        5 * age + 5;
      double numTwix = calories / 568;

      return numTwix;
}
```

There will be a compiler error because the `weight`, `feet`, `inches`, and `age` variables are all private to the `Person` class. One way to make them available would be to make public accessor methods, like we did for age.   Then we can use `getAge()` when we need to access age.

Another way is to make the variables **protected**. This modifier allows access to the instance variable from within the class and from any derived class.

The variables in the Person class now look like this:

```
public class Person
{
      protected int age;
      protected int weight;
      protected int feet;
      protected int inches;
```

The complete `Man` class follows:

```
public class Man extends Person
{
      // Default constructor
      public Man()
      {
            super();
      }

      // Constructor taking arguments
      public Man(int age, int weight, int feet, int inches)
      {
            super(age, weight, feet, inches);
      }
```

```
	public double howManyTwix()
	{
		double kilograms = weight / 2.2;
		double centimeters = ((feet * 12) + inches) * 2.54;

		double calories =  10 * kilograms + 6.25 * centimeters
		                     - 5 * age + 5;
		double numTwix = calories / 568;

		return numTwix;
	}
}
```

Our earlier demo code of:

```
	Man kenrick = new Man(40, 140, 5,6);
	System.out.println("This person of age " + kenrick.getAge() +
	               " should eat " + kenrick.howManyTwix() +
	               " twix bars.");
```

Now gives the identical output to before:

```
	This person of age 40 should eat 2.62 twix bars.
```

You may have noticed that we define the method `howManyTwix()`, in both `Person` and `Man`. How do we know which is invoked? The rule is that the derived class **overrides** the base class. That is, the most specific method to a particular object is the one that is invoked. In the above example, the `Man` class definition of `howManyTwix()` is invoked, not the `Person` class (which would return 0).

We can define the `Woman` class similarly to the `Man` class:

```
public class Woman extends Person
{
	// Default constructor
	public Woman()
	{
		super(20, 140, 5, 6);  // Different defaults for women
	}

	// Constructor taking arguments
	public Woman(int age, int weight, int feet, int inches)
	{
		super(age, weight, feet, inches);
	}

	public double howManyTwix()
	{
		double kilograms = weight / 2.2;
		double centimeters = ((feet * 12) + inches) * 2.54;
```

```
            double calories =  10 * kilograms + 6.25 * centimeters
                                   - 5 * age - 161;
            double numTwix = calories / 568;

            return numTwix;
      }
}
```

We could make the classes even simpler by abstracting out the shared calorie calculation, but that is an exercise for another time (and one you should be able to figure out!)

The takeaways from this example are:

1. A derived class inherits public (or protected) methods from the base class
2. Derived methods override methods of the same name in the base class
3. Use the keyword super to invoke methods from the base class
4. Private variables are inherited by the derived class, but are not directly accessible by name. Public or protected variables are directly accessible by name.

Miscellaneous Gotcha's:

Private methods are not inherited by the derived class. Private methods are only available in the class they are defined.

It might seem like if we have a chain of methods, we could invoke super.super.method() to invoke the base class of a base class. Unfortunately Java does not allow this, one may only use a single super.

Multi-Level Classes

In our example we only had one parent and one child class, but we could have a chain of multiple parents.

It turns out that every class we make is actually a descendant of the predefined class named **Object**. This means that every class we create will inherit methods from Object. So what methods are available to something so general?

Some of the methods we get automatically are methods **equals()** and **toString().** The intent is for these methods to check if another object is equal to the current object, and to convert the current object into a String. However, these inherited methods will not work correctly for just about every class you create. This means it is your job to override these methods with your own that work properly.