Java Classes and Objects

Source http://java.about.com/od/beginningjava/l/aa_objintro_1.htm

Java is an object-based language, though, and real-world programs use dozens, if not hundreds, of classes. One cannot avoid using objects in Java, so an understanding of objects and object-oriented programming (OOP) is useful even to beginning Java programmers.

What is a Type?

A type is a category or grouping that describes a piece of data. Every datum in Java can be identified by its type. Data of the same type share common characteristics. If a datum is of type int, for example, then the programmer knows that it is a whole number that cannot be larger than 231 because these are characteristics that all ints share.

Java has two general categories of types: primitives and classes. Examples of primitives are int, float, and boolean. Primitive types' characteristics cannot be extended or modified by programmers. They represent simple pieces of data such as numbers, bytes, and characters. Classes are user-defined types and are created by programmers to represent more complex objects when simple primitives are insufficient. Many useful classes have already been defined by Sun engineers as part of the J2SE, J2EE, and J2ME libraries. Classes are frequently extended and modified by programmers.

Classes Describe Objects

Classes are the basic building blocks of Java programs. Classes can be compared to the blueprints of buildings. Instead of specifying the structure of buildings, though, classes describe the structure of "things" in a program. These things are then created as physical software objects in the program. Things worth representing as classes are usually important nouns in the problem domain. A Web-based shopping cart application, for example, would likely have classes that represent customers, products, orders, order lines, credit cards, shipping addresses and shipping providers.

Unlike data structures in other languages which only contain data, Java classes consist of both attributes and behaviors. Attributes represent the data that is unique to an instance of a class, while behaviors are methods that operate on the data to perform useful tasks.

Class Syntax

Use the following syntax to declare a class in Java:

```
   //Contents of SomeClassName.java
   [ public ] [ ( abstract | final ) ] class SomeClassName [ extends SomeParentClass ] [
implements SomeInterfaces ]
   {
      // variables and methods are declared within the curly braces
   }
```

- A class can have public or default (no modifier) visibility.
- It can be either abstract, final or concrete (no modifier).
- It must have the class keyword, and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend java.lang.Object.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces '{}'.
- Each .java source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.

Here is an example of a Horse class. Horse is a subclass of Mammal, and it implements the Hoofed interface.

```
public class Horse extends Mammal implements Hoofed
{
  //Horse's variables and methods go here
}
```

What is an Object?

Objects are the physical instantiations of classes. They are living entities within a program that have independent lifecycles and that are created according to the class that describes them. Just as many buildings can be built from one blueprint, many objects can be instantiated from one class. Many objects of different classes can be created, used, and destroyed in the course of executing a program.

Attributes

Although several buildings may be built from the same blueprint and, hence, have the same attributes, such as twelve windows, exterior paint, and a driveway, each instance of the building may have different values for these attributes. One building may have twelve double-pane windows while another may have twelve single-pane windows. One building may have hunter green exterior paint while another may have white exterior paint. The attributes are identical but the values of the attributes are different. Similarly, a class describes what attributes an object will have but each object will have its own values for those attributes. In our shopping cart example, a customer class may specify that customer objects have a first name, last name, and phone number. Each Customer object, though, will have different values for first name, last name, and phone number.

Attributes are represented by non-local variables. These are variables that are not declared within method bodies.

Behaviors

A building has behaviors in addition to attributes: Toilets are flushed; furnaces fire to maintain temperature; doors open and close. Classes describe all of the behaviors of its objects. In our shopping cart example: A customer may place an order; a credit card may reject a purchase; an order may tally line items to generate a total.

Behaviors are represented by methods. An object may call, or invoke, its own methods, or it may call another object's methods that are visible to it.

Constructors

In order to be used by a program, an object must first be instantiated from its class definition. A special type of method called a constructor is used to define how objects are created. A constructor is called with the new keyword. new tells the JVM to allocate memory, initialize instance variables, and assign the object a reference code that uniquely identifies it within the JVM.

Let's create an instance of a Car:

```
Car myYugo = new Car();
```

Messages

Objects cooperate and communicate with other objects in a program by passing messages to one another. When an object invokes a method on itself or another object, it passes a message to the object that contains the target method. The message identifies the method to be invoked and any required data (known as arguments) that the method needs. When the method finishes executing, either a return value or void is passed back to the original invoking object, completing the message.

Let us create another Car object and pass an accelerate message to it.

```
Car myYugo = new Car();
int currentSpeed;
currentSpeed = myYugo.accelerate();
```

Class Inheritance via extends and implements

As you saw in the Horse example, a class can extend one other class and implement many Java interfaces. Extending and implementing is the Java mechanism for representing class inheritance. Inheritance represents an "is a" relationship between two classes. The Horse is a Mammal, and it is a Hoofed. When a child class, or subclass, extends a parent class, or super class, it inherits the parent class's visible variables and methods. The child class can read and write the parent class's visible attributes, it can pass messages to the parent class's visible methods, and it can even override (or re-implement) the parent class's visible methods. We'll cover inheritance in-depth in a future installment of the Java Programming Tutorial.

Now let's continue on and see a working example of using multiple objects in Java.

Now that we have covered some of the theory of objects, it is time to have a little fun. We'll create an object hierarchy with a Car super class and a Mustang subclass. Then we'll take them both for a test drive with the TestDrive class.

Below is the complete code for the Car, Mustang, and TestDrive classes. Create a subdirectory named objintro. Create a file in objintro for each class below. Copy the contents of the classes into the files, and save the files so the class and file name match (remember the .java extension) . So, Car would be saved into a file named Car.java, Mustang into Mustang.java, TestDrive into TestDrive.java. Compile the classes with the javac command. Then run TestDrive with the java command. See First Java Program for more details on compiling and executing Java programs.

Car

```
package objintro;

/**
 *  Represents a car.
 *  The attributes are speed and color.
 *  The methods are accelerate,
 *  decelerate, getSpeed, getColor, getAcceleration
 *  and getMaxSpeed.
 *
 *  Add your own attributes and behaviors
 *  for breaking and for shifting gears.
 *
 * @author Kevin Taylor guide@java.about.com
 */
public class Car {

  protected int speed = 0;
  protected String color = "black";
  private static final int MIN_SPEED = 0;
  private static final int CAR_MAX_SPEED = 100;
  private static final int CAR_ACCELERATION = 10;

  /**
   * Default constructor to create a new
   * Car object.
   */
  public Car() {
    //creates a new object
  }

  /**
   * Simulates pressing the accelerator.
   * @return the new speed
   */
  public int accelerate() {
    int newSpeed = speed + getAcceleration();
    if(newSpeed <= getMaxSpeed()) {
      speed = newSpeed;
    }
    else {
      speed = getMaxSpeed();
    }
    return speed;
  }

  /**
   * Simulates releasing the accelerator.
   * @return the new speed
   */
  public int decelerate() {
    if(speed > MIN_SPEED) {
      speed--;
    }
    return speed;
  }

  /**
   * @return the current speed
   */
  public int getSpeed() {
    return speed;
  }

  /**
   * @return the max speed
   */
  public int getMaxSpeed() {
    return CAR_MAX_SPEED;
  }

  /**
   * @return the max speed
   */
  public String getColor() {
    return color;
  }
```

```
    /**
     * @return the car's acceleration
     */
    public int getAcceleration() {
      return CAR_ACCELERATION;
    }
  }
```

Mustang

```
  package objintro;

  /**
   * Represents a Mustang.
   * It overrides (changes) the getMaxSpeed method,
   * which sets an upper limit for the accelerate
   * method. It also overrides getAcceleration.
   *
   * @author Kevin Taylor guide@java.about.com
   */
  public class Mustang extends Car {

    private static final int MUSTANG_MAX_SPEED = 150;
    private static final int MUSTANG_ACCELERATION = 20;

    /**
     * Constructors are used to initialize
     * the objects variables.
     *
     * The Mustang color is define when the constructor
     * is called, whereas the Car objects are always black.
     */
    public Mustang(String passedColor) {
      // creates a new instance of Mustang and
      // assigns a color
      color = passedColor;
    }

    /**
     * This method is implemented in the Car class and
     * in the Mustang class. The Mustang version overrides
```

```
      * the Car version.
      *
      * @return  The max speed for a Mustang.
      */
     public int getMaxSpeed() {
       return MUSTANG_MAX_SPEED;
     }

     /**
      * This method is implemented in the Car class and
      * in the Mustang class. The Mustang version overrides
      * the Car version.
      *
      * @return  The acceleration for a Mustang.
      */
     public int getAcceleration() {
       return MUSTANG_ACCELERATION;
     }
   }
```

TestDrive

```
   package objintro;

   /**
    * TestDrive demonstrates creating and calling
    * methods on Car and Mustang objects.
    *
    * @author Kevin Taylor java.guide@about.com
    */
   public class TestDrive {

     //The Java virtual machine (JVM) always starts
     //execution with the 'main' method of the class passed
     //as a argument to the java command
     public static void main(String []args) {
       TestDrive td = new TestDrive();
       td.start();
       //exit TestDrive
     }
     private void start() {
       //Create a Car
       Car yugo = new Car();
       //Take it for a drive
       System.out.println("Starting yugo test drive!");
       driveCar(yugo);

       //Create a Mustang
       //Remember, myMustang is a Mustang AND a Car
       Car pony = new Mustang("red");
       //Take it for a drive
       System.out.println("Starting mustang test drive!");
       driveCar(pony);
     }

     public static void driveCar(Car c) {
       System.out.println("Car color is: " + c.getColor());
       //press the accelerator 15 "times"
       for(int i = 0; i < 15; i++) {
         System.out.println("accelerating: " + c.accelerate());
       }
       //release the accelerator 5 "times"
       for(int i = 0; i < 5; i++) {
         ;
         System.out.println("decelerating: " + c.decelerate());
       }
       System.out.println("final cruising speed: " + c.getSpeed());
     }
   }
```

When you compile and run the program, you should see something like the following:

```
  ktaylor$ javac objintro/Car.java objintro/Mustang.java objintro/TestDrive.java
  ktaylor$ java objintro.TestDrive
  Starting yugo test drive!
  Car color is: black
  accelerating: 10
  accelerating: 20
  accelerating: 30
  accelerating: 40
  accelerating: 50
  accelerating: 60
  accelerating: 70
```

```
accelerating: 80
accelerating: 90
accelerating: 100
accelerating: 100
accelerating: 100
accelerating: 100
accelerating: 100
accelerating: 100
decelerating: 99
decelerating: 98
decelerating: 97
decelerating: 96
decelerating: 95
final cruising speed: 95
Starting mustang test drive!
Car color is: red
accelerating: 20
accelerating: 40
accelerating: 60
accelerating: 80
accelerating: 100
accelerating: 120
accelerating: 140
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
accelerating: 150
decelerating: 149
decelerating: 148
decelerating: 147
decelerating: 146
decelerating: 145
final cruising speed: 145
```

*~~~ End of Article ~~~*