Classes and Objects

Based on The Java™ Tutorial
(http://docs.oracle.com/javase/tutorial/java/)
Based on the notes from David Fernandez-Baca and Steve Kautz

Bryn Mawr College
CS206 Intro to Data Structures

Classes and Objects

- Classes
- Objects
- More on Classes
  - The this keyword.
  - Class vs. instance members.
  - Access control.
- Enum Types

Civilization advances by extending the number of important operations which we can perform without thinking.

— Alfred North Whitehead

Modularity and Abstraction

There are two important ideas in software design:

- **Modularity** means building systems out of components that can be developed independently of each other. The goal is to reduce **coupling** between components. That is, changes in one component should not force other components to change.
- **Abstraction** is a way to reduce coupling: View components in terms of their essential features, ignoring details that aren't relevant to our particular concerns. Each component provides a well-defined **interface** specifying exactly how we can interact with it.

Object-oriented programming (OOP) can help us achieve these goals.

Objects

- OOP allows us to easily create small modules that closely model one coherent thing or concept in the problem domain.
- Instead of a collection of data with procedures that act on the data, a modern software system is a collection of interacting **objects** that communicate through well-defined interfaces.
- E.g., bank account. An object models a bank account, keeps track of its own attribute and state, and provides the rest of the system with an interface to update and query its state.

Object and Class

- **State**: instance variables that hold their values between method calls.
- **Identity**: once you create an object, you can refer to it again, and distinguish it from others
- **Operations:** its public interface or API(Application Programming Interface). These are the public methods.

A **class** is a type of object, say a bank account. Many objects of the same class might exist; for instance, myAccount and yourAccount.

Principles of Object-Oriented Programming

- **Encapsulation** means that objects only interact through a well-defined set of operations (the public **interface** or API), while the representation of the state, and the implementation of the operations are kept hidden.
- **Inheritance** allows us to derive a class of objects from a more general class. A derived class inherits properties from the superclass from which it derives. For example, the `SavingsAccount` class might inherit from the `BankAccount` class the property of storing a balance.
- **Polymorphism** is the ability to have one method work on several different classes of objects, even if those classes need different implementations of the method. For example, one line of code might be able to call the add method on **every** kind of List, even though adding an item to a list of `BankAccounts` might be completely different from adding an item to a list of integers.

Declaring Classes

Class declaration

Class body

```
public class Bicycle {
    //the Bicycle class has two fields
    public int speed = 0;
    public int gear = 1;

    //the Bicycle class has one constructor
    public Bicycle(int startSpeed, int startGear){
        gear = startGear;
        speed = startSpeed;
    }

    //the Bicycle class has 3 methods
    public void setGear(int newValue) {
        gear = newValue;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

    public void applyBrakes(int decrement) {
        speed -= decrement;
    }
}
```

Declarations for the fields that provide the state of the class and objects

Constructor initialize new objects

Declarations for methods to implement the behavior of the class and its objects

Declaring Member Variables

- Access modifiers: let you control what other classes have access to a member field.
  - **public** modifier: the field is accessible from all classes.
  - **private** modifier: the field is accessible only within its own class.
- In the spirit of encapsulation, it is common to make fields private.

```
public class Bicycle {
   //the Bicycle class has two fields
   public int speed = 0;
   public int gear = 1;

   ……
}
```

```
public class Bicycle {
   private int speed = 0;
   private int gear = 1;

   public Bicycle(int startSpeed, int startGear){
      gear = startGear;
      speed = startSpeed;
   }

   public int getGear() {
      return gear;
   }
   public void setGear(int newValue) {
      gear = newValue;
   }
}
```

Overloading Methods

- Method overloading: methods with the same name.
- Java can distinguish between methods with different *method signatures*
- Method signature: the method's name and the parameter types.

```
public double calculateAnswer(double wingSpan,
        int numberOfEngines, double length,
        double grossTons) {
   //do the calculation here
}
```

Signature:
calculateAnswer(double, int, double, double)

```
public class DataArtist {
   public void draw(int i) {
      …
   }

   public void draw(String s) {
      …
   }

   public void draw(double f) {
      …
   }

   public void draw(int i, double f){
      …
   }
}
```

Constructors

Constructor declarations look like method declarations, but

- They use the name of the class name
- No return type

```
public Bicycle() {
    gear = 1;
    cadence = 10;
    speed = 0;
}
public Bicycle(int startSpeed, int startGear) {
    gear = startGear;
    speed = startSpeed;
}
```

To create a new Bicycle object called myBike:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

Note: Creating a new instance is also called *instantiation*. To instantiate an object we must invoke a *constructor*, whose role is to establish the initial values of the instance variables.

Passing information to a Method or a Constructor

- Primitive arguments, such as an int or a double, are passed into methods **by value**.
  - Any changes to the values of the parameters exist only within the scope of the method.
- Reference data type parameters, such as objects, are also passed into methods **by value**.
  - when the method returns, the passed-in reference still references the same object as before.
  - **However**, the values of the object's fields **can** be changed in the method, if they have the proper access level.

Objects

1. allocate memory for a new object and return a reference to that memory.
2. Invoke the object constructor.

Declare and create a point object and two rectangle objects.

display rectOne's width, height, and area

Set rectTwo's position

display rectTwo's position

move rectTwo and display its new position

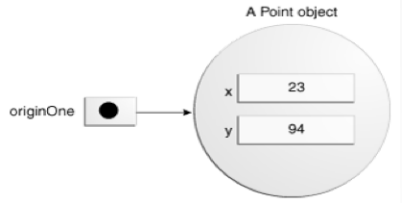
```
1.  public class CreateObjectDemo {
2.      public static void main(String[] args) {
3.          Point originOne = new Point(23, 94);
4.          Rectangle rectOne = new Rectangle(originOne, 100, 200);
5.          Rectangle rectTwo = new Rectangle(50, 100);
6.
7.          System.out.println("Width of rectOne: " + rectOne.width);
8.          System.out.println("Height of rectOne: " + rectOne.height);
9.          System.out.println("Area of rectOne: " + rectOne.getArea());
10.
11.         rectTwo.origin = originOne;
12.
13.         System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
14.         System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
15.
16.         rectTwo.move(40, 72);
17.         System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
18.         System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
19.     }
20. }
```

Objects – Rectangle and Point

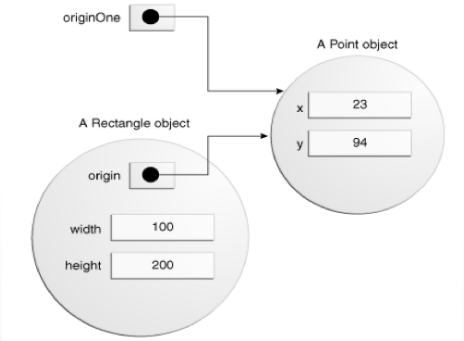
```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
    origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
    origin = p;
    }
    public Rectangle(int w, int h) {
    origin = new Point(0, 0);
    width = w;
    height = h;
    }
    public Rectangle(Point p, int w, int h) {
    origin = p;
    width = w;
    height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
    origin.x = x;
    origin.y = y;
    }

   // a method for computing the area of the rectangle
    public int getArea() {
    return width * height;
    }
}

public class Point {
    public int x = 0;
    public int y = 0;
    // a constructor!
    public Point(int a, int b) {
    x = a;
    y = b;
    }
}
```

Objects

After executing line 3:



After executing line 4:



Using the this Keyword

Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called.

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 0, 0);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

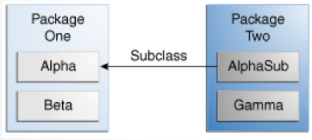
Controlling Access to Members of a Class

- Access level modifiers determine whether other classes can use a particular field or invoke a particular method.
  - top level—public, or *package-private* (no explicit modifier).
  - member level—public, private, protected, or *package-private* (no explicit modifier).
- public: class is visible to all classes everywhere.
- no modifier (the default): visible only within its own package
- private: the member can only be accessed in its own class.
- protected: the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

Access Levels and Visibility

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N



The visibility of the members of the Alpha:

Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Tips on Choosing an Access Level

- Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
- Avoid public fields except for constants. Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Class Variables

- *instance variables*: each object has its own distinct copy
- *static fields* or *class variables*:
  - They are common to all objects.
  - Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

Enum Types

- An *enum type* is a special data type that enables for a variable to be a set of predefined constants.
- The variable must be equal to one of the values that have been predefined for it.
- The enum declaration defines a *class* (called an *enum type*).
- It has a static *values* method (automatically added by the compiler) that returns an array containing all of the values of the enum in the order they are declared.

Enum Example

```
public class EnumTest {
   Day day;

   public EnumTest(Day day) {
      this.day = day;
   }

   public void tellItLikeItIs() {
      switch (day) {
         case MONDAY:
            System.out.println("Mondays are bad.");
            break;

         case FRIDAY:
            System.out.println("Fridays are better."); break;

         case SATURDAY: case SUNDAY:
            System.out.println("Weekends are best.");
            break;

         default:
            System.out.println("Midweek days are so-so.");
            break;
      }
   }

   public static void main(String[] args) {
      EnumTest firstDay = new
                  EnumTest(Day.MONDAY);
      firstDay.tellItLikeItIs();
      EnumTest thirdDay = new
                  EnumTest(Day.WEDNESDAY);
      thirdDay.tellItLikeItIs();
      EnumTest fifthDay = new
                  EnumTest(Day.FRIDAY);
      fifthDay.tellItLikeItIs();
      EnumTest sixthDay = new
                  EnumTest(Day.SATURDAY);
      sixthDay.tellItLikeItIs();
      EnumTest seventhDay = new
                  EnumTest(Day.SUNDAY);
      seventhDay.tellItLikeItIs();
   }
}

   public enum Day {
      SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
      THURSDAY, FRIDAY, SATURDAY
   }
```