Chapter 7 – Classes and Objects, OO Programming, Part A

Sections	Pages	Review Questions	Programming Exercises
7.1-7.11	230-253	1-20	1-4

Custom Objects

1. Consider this example of creating and using a custom object. A box has length, width, and height and a method to calculate its volume. Run program and discuss.

```
public class Box
{
   public double length, width, height;

   public Box( double l, double w, double h )
   {
      length = l;
      width = w;
      height = h;
   }

   public static void main(String[] args)
   {
      Box b = new Box(4,6,2);

      System.out.println( b.toString() );

      b.length = 6;

      System.out.println( b.toString() );
   }

   public double volume()
   {
      return length*width*height;
   }

   public String toString()
   {
      String format = "This box has length, width, and height: ";
      format += "(%.2f, %.2f, %.2f),\n  and volume: %.2f";

      String msg = String.format( format, length, width, height,
                                  volume() );

      return msg;
   }
}
```

The output is:

```
This box has length, width, and height: (4.00, 6.00, 2.00),

   and volume: 48.00

This box has length, width, and height: (6.00, 6.00, 2.00),

   and volume: 72.00
```

2. Constructors – Notice in *main* where we create a box:

```
Box b = new Box(4,6,2);
```

This special statement invokes the *constructor* for the Box class:

```
public Box( double l, double w, double h )
{
   length = l;
   width = w;
   height = h;
}
```

In other words, the constructor *creates* (*instantiates*) a box by setting the values for the length, width, and height.

3. Suppose we want to compare two boxes and determine the percentage difference in their volumes. We could add a method to the Box class:

```
public double volumeDifference( Box b )
{
   double thisBoxVolume = volume();

   double otherBoxVolume = b.volume();

   double percentDifference =
       ( thisBoxVolume - otherBoxVolume ) / otherBoxVolume * 100;

   return percentDifference;
}
```

And we can test this code with:

```
Box b1 = new Box(6,6,2);

Box b2 = new Box(4,6,2);

System.out.println( b1.volumeDifference( b2 ) );
```

4. Notice that we have used *main* to test the Box class. Next, solve a problem: Read the dimensions of two boxes and compare their volumes with percentage differences. Thus, *main* needs to read these dimensions and build boxes. We could write this code straight away, but there would be a lot of duplicate code. Or, we could use a method to help *main*. Notice that when *main* needs help, we write *static methods*. So, we add this method:

```
public static Box createBox()
{
   Scanner s = new Scanner( System.in );

   System.out.print("Enter length: ");
   double len = s.nextDouble();

   System.out.print("Enter width: ");
   double wid = s.nextDouble();
   System.out.print("Enter height: ");
   double ht = s.nextDouble();

   Box b = new Box( len, wid, ht );

   return b;

}
```

And we code the *main* this way:

```
Box b1 = createBox();
Box b2 = createBox();

double percentDiff = b1.volumeDifference( b2 );

String size = percentDiff > 0 ? "larger" : "smaller";

percentDiff = Math.abs( percentDiff );

String format = "The first box is %.0f%% " + size +
                " than the second box.";

System.out.printf( format, percentDiff );
```

5. Usually, however, it is simpler to write two different classes. For instance, we write the Box class which contains only information about a box. And, we write a BoxCompare class to solve our problem. We can put these two classes in the same file, *BoxCompare.java* (see code below). Notice how we have kept all the information about a box in the Box class and all the information about the *problem* in the BoxCompare class.

```java
import java.util.*;

public class BoxCompare
{
   public static void main(String[] args)
   {
      Box b1 = createBox();
      Box b2 = createBox();

      double percentDiff = b1.volumeDifference( b2 );

      String size = percentDiff > 0 ? "larger" : "smaller";

      percentDiff = Math.abs( percentDiff );

      String format = "The first box is %.0f%% " + size +
                      " than the second box.";

      System.out.printf( format, percentDiff );
   }

   public static Box createBox()
   {
      Scanner s = new Scanner( System.in );

      System.out.print("Enter length: ");
      double len = s.nextDouble();

      System.out.print("Enter width: ");
      double wid = s.nextDouble();
      System.out.print("Enter height: ");
      double ht = s.nextDouble();

      Box b = new Box( len, wid, ht );

      return b;
   }
}
```

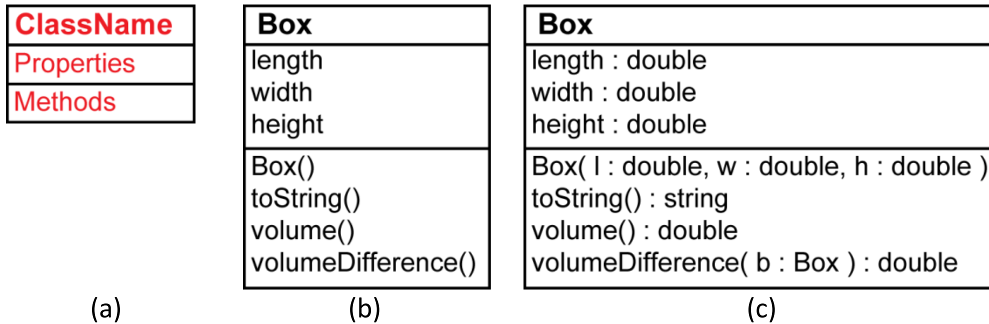
```java
class Box
{
   public double length, width, height;

   public Box( double l, double w, double h )
   {
      length = l;
      width = w;
      height = h;
   }

   public double volume()
   {
      return length*width*height;
   }

   public double volumeDifference( Box b )
   {
      double thisBoxVolume = volume();
      double otherBoxVolume = b.volume();
      double percentDifference =
             ( thisBoxVolume - otherBoxVolume ) / otherBoxVolume * 100;
      return percentDifference;
   }

   public String toString()
   {
      String format = "This box has length, width, and height: ";
      format += "(%.2f, %.2f, %.2f),\n  and volume: %.2f";

      String msg = String.format( format, length, width, height,
                                  volume() );

      return msg;
   }
}
```

6. Finally, we could put the two classes in separate files, put the BoxCompare class in *BoxCompare.java* and the Box class in *Box.java.* Each must be compiled and we run the program as before (java BoxCompare). This will work as long as *Box.java* is in the same folder as *BoxCompare.java*. In most development projects, each class is in a separate file. This has an advantage because then we can put a *main* in each class that tests each that particular class, individually. In fact, that is what we did in the example on the first page, *main* for the Box class simply called its methods and set some values. When you run a class (java BoxCompare), it's *main* is called. If that code subsequently uses a different class in another file, the *main* in this other class (Box) is ignored.

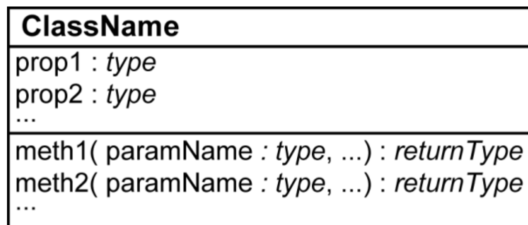
Object Oriented Methodology Overview

1. A *class* is a template for making objects. A class specifies *properties (data fields)* and *methods (behaviors, responsibilities, services)*. When an object is created from a class, it's properties and methods can be accessed.

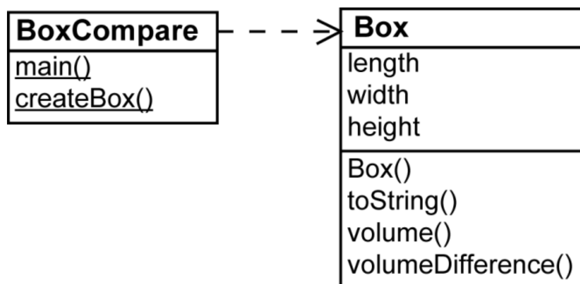
2. A convenient way to represent a class is to use UML (Unified Modeling Language). We model a class with a figure as shown in (a). For instance the Box class can be represented as in (b) or (c), where the data types and return types are also shown. We call either (b) or (c) a class diagram.



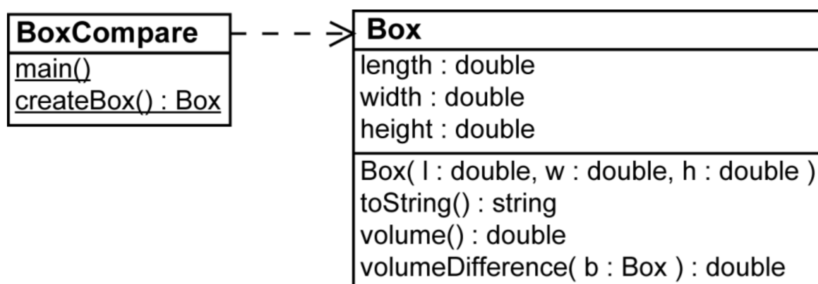
3. For now, the syntax for a class diagram looks like this:



4. The class diagram for the entire problem is shown below. Notice that we underline static methods and we use a *dashed arrow* to indicate that the BoxCompare class *depends* on the Box class.



Or:



Encapsulation

1. It is generally recommended that you have *no* public data fields in a class. The reason is that we don't want to allow a *client* program to be able to change values to anything they want. For instance, we wouldn't want a client program to change the *curPrice* to -22.33. So, we need to *protect* our state (data) so that it is not corrupted. After all, we are designing a class that *assumes* that prices will always be positive. So, it makes sense to not assume this, but to *enforce* this. We do this by providing a *private* instance variable and a *public setter (mutator):*

```
private double curPrice;

public void setCurPrice( double price )
{
      if ( price >= 0.0 )
            curPrice = price;
}
```

As it stands, however, we cannot access *curPrice* from outside the object. If this access is required, then we provide a *getter (accessor)* method. It allows a *client* (another program/class) to retrieve the value of a private instance variable.

```
public double getCurPrice()
{
      return curPrice;
}
```

This process of declaring private instance variables and providing public *getters* and *setters* is referred to as *data field encapsulation* (or just *encapsulation*). *Encapsulation* in OO means that we are *wrapping up our state*, to protect it. When we provide both a getter and a setter, we say that this is a *read-write* property because we can both read the value of the property and change (write) the value of the property.

Even though we no longer have any public state variables, we still say that part of the state of the Stock class is the *current price property*.

2. Read-only Properties – Sometimes, we want to have properties that can be read, but not written to. These are called *read-only* properties. To implement this is very simple; we just omit the *setter*. This does raise the question, how does the property ever obtain a value. One common way is through the constructor. Consider an employee that has an ID. We may want to give an employee an ID when she is created, but once created, the person's ID can never change.

```
class Employee
{
     private int id;

     public int getId()
     {
        return id;
     }

     Employee( int idNum )
     {
        id = idNum;
     }
}
```

We say that a property is *immutable* if it can never be changed. Here, *id* is an immutable property.

3. An *immutable* class is one where the state cannot be changed once the object is created. In other words, all data fields are private and there are no setters. Why would we want to do such a thing? The answer is simply *protection*. It will be a long time (several classes), probably, before the full implications of immutability are understood.

Local Variables, Precedence, and *this*

1. A private or public variable in a class is visible anywhere in that class: in the constructor, public or private methods, or used in expressions in the declaration of other variables. Variables can be declared inside methods in a class and these are called *local variables.* If a *local variable* has the same name as a class variable, then it has precedence. For instance, consider this class:

```
class A
{
    private int x;

    public A()
    {
       x = 0;
    }

    public void doIt( int val )
    {
       int x;
       x = val;
    }

    public int getX() { return x; }

}
```

If we run this test code,

```
A a = new A();

a.doIt( 10 );

System.out.println( a.getX() );
```

We print the value 0. So, we see that the local variable took precedence over the class variable.

2. If we want the class variable to take precedence, we use the keyword, *this*. In the example above, if we wanted to use the class variable, *x* then we could refer to it as *this.x*:

```
public void doIt( int val )
{
      int x;
      this.x = val;
}
```

Which would display the value 10 with the test code above. We frequently see this when we write setters and constructors.

```
public void setX( int x )
{
      this.x = x;
}
```

Some programmers will use *this* for every use of the class variable to make a consistent differentiation between class variables and local variables. In a similar way, you can also use *this* to refer to an instance method. For example, suppose we wanted to provide a setter that only updates the value of the private variable if the new value was greater than the current value. Either of these two examples is ok:

```
public void setX( int x )
{
      if( x > getX() )

            this.x = x;
}
```

or

```
public void setX( int x )
{
      if( x > this.getX() )

            this.x = x;
}
```

More formally, *this* refers to the *instance* that invoked a particular method.

Calling other Constuctors

1. The keyword *this* can also be used inside a constructor to invoke another constructor in the same class. For instance we may want a Circle class that has a constructor that makes a circle with a specified radius. We may also want a constructor that make a *default* circle of radius 1.

```
class Circle
{
    private double radius = false;
    private boolean defaultCircle;

    public Circle( double radius )
    {
       this.radius = radius;
    }

    public Circle()
    {
       this( 1.0 );
       this.defaultCircle = true;
    }

}
```

Note that when the default constructor is called:

```
Circle c = new Circle();
```

it immediately calls the constructor that takes a double as an argument, passing it a value of 1.0**. If this technique is used, *this(...)* must be the first statement in the constructor**. Other statements may follow.

More about Constructors

2. A *constructor* is responsible for creating an object. It may or may not require parameters. A constructor is not required. This is valid code:

```
class A
{
    int x;
}
```

No constructor is specified here, but the **null constructor is assumed to be present**. The null constructor is

```
  public A() {}
```

So that class *A* below is the same as class *A* above:

```
class A
{
    int x;

    public A() {}
}
```

3. We can also supply a constructor that takes an argument:

```
class A
{
    int x;

    public A( int x )
    {
      this.x = x;
    }
}
```

However, in this particular case, this is allowed:

```
        A a = new A(3);
```

and this is not:

```
        A a2 = new A();
```

because **if any constructor is specified, then there is no null constructor**. It will generate this compile error:

```
Constructors.java:14: cannot find symbol
 symbol  : constructor A()
 location: class A
        A a2 = new A();
                   ^
```

In this situation, if a null constructor was also needed, then it would need to be explicitly defined:

```
class A
{
     int x;

     public A() {}

     public A( int x )
     {
        this.x = x;
     }
}
```

Object Oriented Methodology

1. A *class* is a template for making objects.

2. An *object* (*instance*) has:

    a. *Identity* – It is unique. It can be distinguished from other objects.

    b. *State* – The things that it remembers.

    c. *Behavior* – The things that it can do

    d. *Interface* – The messages it can respond to.

3. What is *identity*? Consider this class:

```
class Stock
{
     double price;

     public Stock()
     {
       price = 0.0;
     }
}
```

Now, consider the creation of two stock objects (two instances of the Stock class):

```
Stock s1 = new Stock();
Stock s2 = new Stock();
```

Note that *s1* and *s2* are *references* to two distinct objects. The objects themselves occupy different physical spaces in memory. Though *s1* and *s2* are technically *references* to the actual objects, it is convenient to refer to them as the objects themselves. In other words, we may say, “the *s1* object,” as opposed to, “the object that *s1* refers to.”

Note, for these two particular objects, at this time, they have identical *state* (explained next).

4. What is *state*? It is the information that is stored with each object. It is any information that *characterizes* an object, *attributes* of an object, *properties* that an object has. Some of the state may be *private* and some may be *public*. *Public* information is available from *outside an object*, from a *calling method*. In the example above, the *state* of a Stock object is the *price* and it is public (by default).Thus, a calling program can directly access the public state:

```
Stock s1 = new Stock();

System.out.println( s1.price );
```

Private information is available only *inside* an object. Consider this class:

```
class Stock
{
    private String name;
    double price;

    public Stock( String n )
    {
        price = 0.0;
        name = n;
    }
}
```

This class (or an object created from this class) has *name* and *price* for it's state. Note that *name* is private and *price* is public. The fact that *name* is private means that it is not accessible from outside the object that contains it. In other words, the following code is not allowed. It generates a compile error:

```
Stock s1 = new Stock( "Sun" );

System.out.println( s1.name );
```

Notice that *name* can be used inside the object. In the example above, it is used in the constructor (copied below):

```
public Stock( String n )
{
     price = 0.0;
     name = n;
}
```

In this particular example, the *name* property is not very useful. We store it when the object is created, but it has no other use. We'll see more about this later.

An important part of OO modeling is deciding whether variables need to be public or private.

5. What is *behavior*? These are the things that an object can do, the actions it can perform, responsibilities that the object has. For instance, a Stock may have the ability to update its price. We *implement* a *behavior* in Java with a *method*. Thus, the Stock class has an *updatePrice()* behavior (method) shown below and also a *percentDifference()* method.

```
class Stock
{
     double curPrice;
     double prevPrice;

     Stock( double cPrice )
     {
          curPrice = cPrice;
          prevPrice = 0.0;
     }

     public void updatePrice( double price )
     {
        prevPrice = curPrice;
        curPrice = price;
     }

     double percentDifference()
     {
          return (curPrice-prevPrice)/prevPrice * 100.0;
     }
}
```

A class can have many behaviors and each one is either public or private. A public behavior (method) can be *invoked* from the outside (calling method) by using the *dot* operator:

```
Stock s1 = new Stock( 22.0 );

s1.updatePrice( 33.0 );

double d = s1.percentDifference();
```

We can also have private methods which are sometimes called *helper methods*. They are only accessible inside an object and are used to help a public method carry out its task. For instance, suppose that we wanted to ensure that *prevPrice* is greater than 0 (otherwise we'll have a run-time error in the previous code). We might implement the Stock class as shown below:

```
class Stock
{
     double curPrice;
     double prevPrice;

     Stock( double cPrice )
     {
        curPrice = cPrice;
        prevPrice = 0.0;
     }

     public void updatePrice( double price )
     {
        previousClosingPrice = currentPrice;
        currentPrice = price;
     }

     double percentDifference()
     {
        if ( isValidPrevPrice () )
              return (curPrice-prevPrice)/prevPrice * 100.0;
        else
              return 0.0;
     }

     private boolean isValidPrevPrice()
     {
        return ( prevPrice > 0.0 );
     }
}
```

Notice that we use the private helper method, *isValidPrevPrice()* to carry out the responsibility of the public behavior, *percentDifference*.

6. An *interface* for a class (or object) is the set of *public members* (public fields and methods). In other words, the interface specifies exactly what messages can be sent to an object from the outside. Thus, a programmer who is *using* a class is usually only interested in the *public interface* for the class because those are the only things it can use. When we refer to an *interface,* we almost always mean the *public* interface.

   We sometimes say that an object can *receive messages.* This is just another way that we can refer to using an objects properties and methods. In other words, a message asks the receiving object to respond in some way. A message might ask for the value of the state:

```
Stock s1 = new Stock(22.33, 33.44);

double p = s1.curPrice;
```

   Or to carry out a responsibility (invoke a behavior):

```
double d = s1.percentDifference();
```