Chapter 11: Creating Classes

Objectives

Students should

- Recall the meaning of classes and objects in Java
- Know the components in the definition of a Java class
- Understand how constructors work
- Be able to create class and object methods
- Be able to create new Java classes and use them correctly

Define Your Own Data Type

Recall that there are two categories of data in Java, namely primitive data type and class. As mentioned earlier, we cannot create a new primitive data type. However, most of the time programmers want to create new data types; they can do so by creating new classes containing *attributes* and *behaviors* of the desired data types. Creating a new class, we write a *class definition* associated with that class in specific Java syntax, and save the definition in a separated .java file named after the name of the class. Once the definition is created, other programs can utilize the newly created data type or class in a similar fashion to the primitive data types or other existing classes. Go back and consult Chapter 5 if you cannot recall the meaning of "objects" and how they are related to classes.

Suppose that we would like to create a new data type for representing points in a Cartesian co-ordinate, we could create a new class called *MyPoint*, whose definition follows the following structure.

```
public class MyPoint
{
    // a blank class definition
    // there're no details yet
}
```

This definition has to be saved using the name MyPoint.java. Then, we can write another program that makes use of this class. For example, we could create another program that looks like:

```
public class TestMyPoint1                                   1
{                                                           2
    public static void main(String[] args)                  3
    {                                                       4
        MyPoint p, q;                                       5
        p = new MyPoint();                                  6
        q = new MyPoint();                                  7
    }                                                       8
}                                                           9
```

In the above program, variables `p` and `q` are declared as variables of the type *MyPoint* on line 5 using a similar syntax as when variables of other types are declared. On line 6 and line 7, `p` and `q` are assigned with, or in other words, are made to refer to, new instances, or objects, of the class *MyPoint* using the keyword `new`. This program just shows us a valid way to make use of the newly created class. It has not done anything useful since we did not define anything inside the definition of *MyPoint*.

Notice that source codes of Java programs that we have written so far, they take the same structure as class definitions. Actually, they are in fact class definitions. However, Java programs are classes that contain the methods named *main()* which make the class executable.

In reality, we want to put something useful in the definition of our classes so that they are not just blank as it appears in *MyPoint* above.

Components of Class Definitions

The main functionality of a class definition is to define attributes and behaviors of that class. Attributes are entities defining properties of an object. Behaviors are actions (or reactions) of an object. The table below shows example attributes and behaviors of some objects.

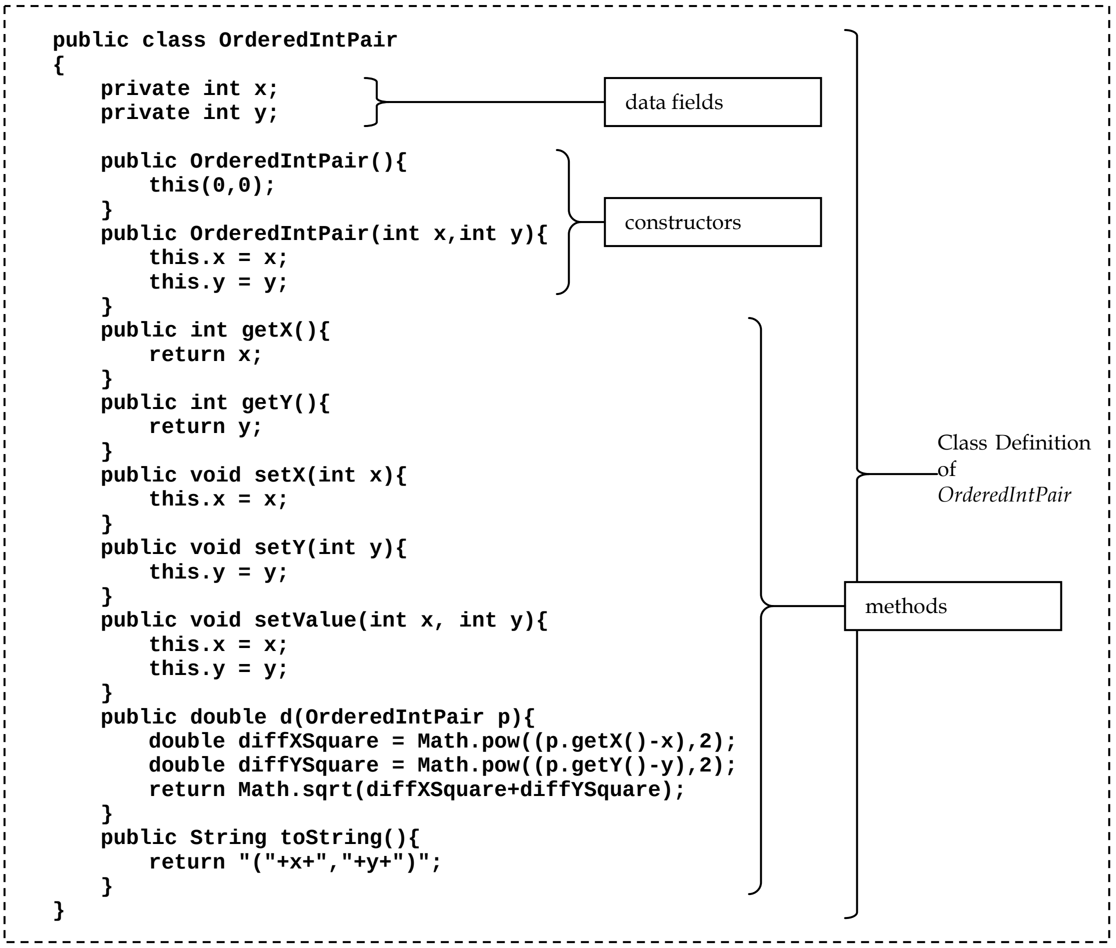
Object type	Attributes	Behaviors
Point in a 2D space	The x coordinate The y coordinate etc.	Moving the point a specified location Calculating distance from the point to a specified location etc.
Graphical line in a 3D space	Location of the starting point Location of the ending point Color etc.	Calculating the length of the line Moving the starting point to a specified location Moving the ending point to a specified location Changing the color of the line etc.
Complex number	Value of the real part Value of the imaginary part etc.	Adding the object with another complex object, Multiplying the object with another complex object Finding the conjugate of the object Setting the real part to a specific number Showing String representation of the object etc.
Matrix	Members etc.	Adding elements to the object Finding determinant Adding the object with another matrix object Finding the inverse of the object Raising to the power of n etc.
Car	Body color Dimensions Weight Number of doors Manufacturer Engine status etc.	Starting the engine Shutting down the engine Showing the name of its manufacturer Accelerating Decelerating etc.
Bank account	Account name Owner Account type Balance etc.	Showing the current balance Showing all info associated with the account Withdrawing money from the account Depositing to the account Closing the account etc.
Customer	Customer ID First name Family name Credit line Gender Favorite products etc.	Showing all info of the customer Changing the credit line Checking whether the customer's favorite product consists of a specified product etc.

To describe attributes and behaviors of objects of the class, a class definition can consist of the following components.

1. data member or fields
2. methods
3. constructors

An object's attribute is represented using a *data member*. Variables used for storing data members are called *instance variables*. The behaviors of an object are described using *methods*. *Constructors* are special methods invoked whenever objects of the class are created.

The class definition shown below serves as an example aiming at giving you a very broad overview the structure and syntaxes of a class definition. Details are reserved for next sections.



```java
public class OrderedIntPair
{
    private int x;
    private int y;

    public OrderedIntPair(){
        this(0,0);
    }
    public OrderedIntPair(int x,int y){
        this.x = x;
        this.y = y;
    }
    public int getX(){
        return x;
    }
    public int getY(){
        return y;
    }
    public void setX(int x){
        this.x = x;
    }
    public void setY(int y){
        this.y = y;
    }
    public void setValue(int x, int y){
        this.x = x;
        this.y = y;
    }
    public double d(OrderedIntPair p){
        double diffXSquare = Math.pow((p.getX()-x),2);
        double diffYSquare = Math.pow((p.getY()-y),2);
        return Math.sqrt(diffXSquare+diffYSquare);
    }
    public String toString(){
        return "("+x+","+y+")";
    }
}
```

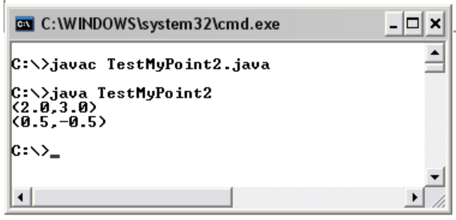
Data Members

Instance variables are used for storing data members. Instance variables can be declared, and possibly initialized, using the same syntax used with variables in methods (such as **`int x;`**, **`String s;`**, **`double [] d = {1.0, 2.0};`**, and etc.). Furthermore, modifiers determining access rights, including **`public`**, **`private`**, and **`protected`**, can be used to determine which classes are allowed to access the data members in those instance variables. For example, an object of the class *MyPoint* can have two **`double`** values representing the x-coordinate and the y-coordinate of the point represented by that object. Therefore, the class definition could look like:

```java
public class MyPoint
{
    public double x;
    public double y;
}
```

The modifier **`public`** identifies that anyone can access the two instance variables using the dot operator. The following example program demonstrates the accessing of the instance variables.

```
public class TestMyPoint2                                  1
{                                                          2
    public static void main(String[] args)                 3
    {                                                      4
        MyPoint p = new MyPoint();                         5
        MyPoint q = new MyPoint();                         6
        p.x = 2;                                           7
        p.y = 3;                                           8
        q.x = 0.5;                                         9
        q.y = -0.5;                                       10
        System.out.println("("+p.x+","+p.y+")");          11
        System.out.println("("+q.x+","+q.y+")");          12
    }                                                     13
}                                                         14
```



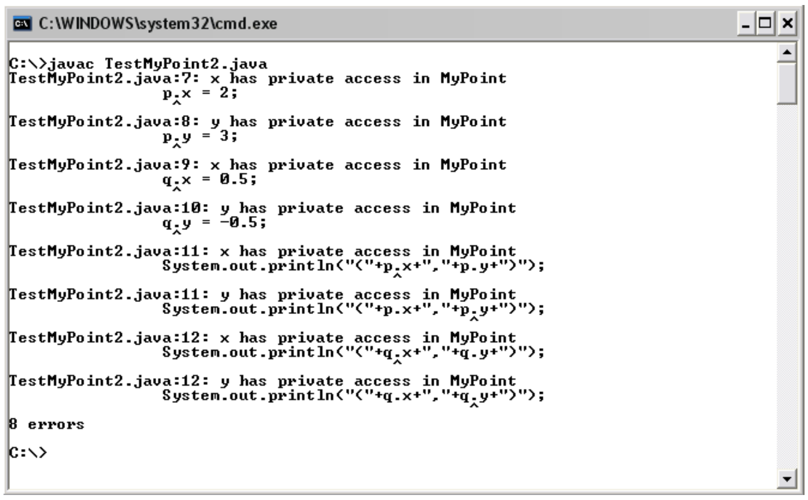
```
C:\>javac TestMyPoint2.java

C:\>java TestMyPoint2
(2.0,3.0)
(0.5,-0.5)

C:\>_
```

On line 5 and line 6, the variables named **p** and **q** are created. Each of them is made to refer to a new *MyPoint* object. The instance variable **x** of the object referred to by **p** is set to 2 while **y** is set to 3 on line 7 and line 8. On the next two lines, the instance variable **x** of the object referred to by **q** is set to 0.5 while **y** is set to -0.5. The code on line 11 and line 12 print the output on the screen. They use the values of **x** and **y** in both objects through **p.x**, **p.y**, **q.x**, and **q.y**.

Now if we change the class definition of *MyPoint* to:

```
public class MyPoint
{
    private double x;
    private double y;
}
```

Compiling TestMyPoint2.java again will lead to compilation errors as shown in the picture below.



```
C:\>javac TestMyPoint2.java
TestMyPoint2.java:7: x has private access in MyPoint
                p.x = 2;
                 ^
TestMyPoint2.java:8: y has private access in MyPoint
                p.y = 3;
                 ^
TestMyPoint2.java:9: x has private access in MyPoint
                q.x = 0.5;
                 ^
TestMyPoint2.java:10: y has private access in MyPoint
                q.y = -0.5;
                 ^
TestMyPoint2.java:11: x has private access in MyPoint
                System.out.println("("+p.x+","+p.y+")");
                                        ^
TestMyPoint2.java:11: y has private access in MyPoint
                System.out.println("("+p.x+","+p.y+")");
                                               ^
TestMyPoint2.java:12: x has private access in MyPoint
                System.out.println("("+q.x+","+q.y+")");
                                        ^
TestMyPoint2.java:12: y has private access in MyPoint
                System.out.println("("+q.x+","+q.y+")");
                                               ^
8 errors

C:\>
```

The modifier **private** makes instance variables private to the class they are declared. That means the instance variables can be used or accessed by that class or in the class definition of that class only. Errors occur whenever the private instance variables **x** and **y** of any instances of *MyPoint* are accessed directly by other classes. In this case, the class trying to access those variables is *TestMyPoint2*. The modifier **`private`** allows the creator of the class to hide data members from the outside world. Doing this is crucial to the *data encapsulation* concept in *Object-Oriented Programming (OOP)*. However, we do not intend to elaborate on OOP concepts in this course.

Another modifier determining access levels is the modifier **`protected`**. Protected elements cannot be access by any classes other than the class they are declared and their *subclasses*. Subclasses will be discussed in the next chapter. The default access level for Java is **`protected`**. That means if no access level is specified, it is, by default, **`protected`**.

Data members of a class can be objects of other classes, both standard and user-defined. For example, let's suppose we would like to create a class representing polygons, each of which has its associated text label. We might decide that its data members include an array of *MyPoint* objects for storing the location of every vertex of the polygon, and a *String* object representing the label. The class definition could be listed as the code below.

```
public class MyLabelledPolygon
{   private MyPoint [] vertices;
    private String label;

    // ... other elements are omitted ...
}
```

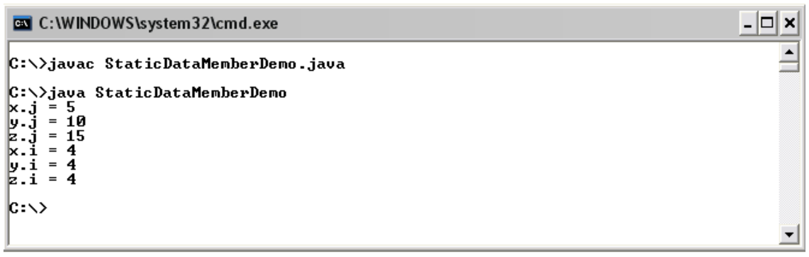
Static and Non-static Data Members

Data members can be either *static* or *non-static*. Non-static data members are attributes of instances of the class, while static data members are attributes of the class itself. In other words, each instance of the class has its own copy of non-static data members, while static data members are shared among every instances of the class. Data members are non-static by default. To make a data member static, we use the modifier **`static`** in front of the declaration of variables storing the data members. Therefore, to be precise, we will not call variables storing static data members instance variables since the variables are not the attributes of any specific instances but they are shared among every instances.

The following example shows how static and non-static variables are declared and used.

```
public class L11A                                           1
{                                                           2
    public static int i;                                    3
    public int j;                                           4
}                                                           5
```

```
public class StaticDataMemberDemo                           1
{                                                           2
    public static void main(String[] args)                  3
    {                                                       4
        L11A x = new L11A();                                5
        L11A y = new L11A();                                6
        L11A z = new L11A();                                7
        x.j = 5;                                            8
        y.j = 10;                                           9
        z.j = 15;                                          10
        System.out.println("x.j = "+x.j);                  11
        System.out.println("y.j = "+y.j);                  12
        System.out.println("z.j = "+z.j);                  13
        x.i = 0;                                           14
        y.i++;                                             15
        z.i += 3;                                          16
        System.out.println("x.i = "+x.i);                  17
        System.out.println("y.i = "+y.i);                  18
        System.out.println("z.i = "+z.i);                  19
    }                                                      20
}                                                          21
```



```
C:\>javac StaticDataMemberDemo.java

C:\>java StaticDataMemberDemo
x.j = 5
y.j = 10
z.j = 15
x.i = 4
y.i = 4
z.i = 4

C:\>
```

On line 5, line 6, and line 7, three instances of *L11A* are created and referred to by **`x`**, **`y`** and **`z`**. On line 8, line 9, and line 10, the values of 5, 10, and 15 are assigned to the instance variables **`j`** belonging to the objects referred to by **`x`**, **`y`**, and **`z`**, respectively. These objects do not share the value of **`j`**. However, the variable **`i`** is shared by the three objects. The statement **`x.i = 0`** on line 14 assign 0 to **`i`**. Note that at this point **`y.i`** and **`z.i`** are also 0 since they refer to the same thing. **`i`** can be modified via any objects. Therefore, we can see that the resulting value of **`i`**, shared by **`x`**, **`y`** and **`z`**, is 4.

Methods

Methods describe behaviors of objects of the class. We have learned how to use methods defined in existing classes in Chapter 5. In Chapter 5, we also mentioned that there were two types of methods: static (class) methods and non-static (instance) methods. A (public) method defined in a class definition is by default non-static and it can be invoked by other classes via the instance name of an object of that class using the dot operator. To make a method static, the keyword *static* is put in the method header. This way, the method can be invoked using the dot operator with the name of the class. The general syntax of defining a method in a class is similar to what we have already been familiar with in Chapter 8. This time, we will look at the syntax in a more general view. The syntax follows:

```
(public|private|protected) (static) returnType methodName(argumentList){
    methodBody
}
```

An access level modifier (either **`public`**, **`private`**, or **`protected`**) can be specified at the beginning of the method header. It determines whether which classes can make use of this method. The access levels specified by **`public`**, **`private`**, and **`protected`** are similar to when they are used with data members. If this modifier is not specified, the default access level is **`protected`**.

The keyword **`static`** makes the method static, or a class method. If omitted, the method is considered as non-static, or an instance method.

The other parts of the method definition are the same as what we discussed in Chapter 8.

We can define as many methods as we would like in the class definition. If the definition contains a public method named *main()*, the class can be executed. In other words, the class is in fact a Java program.

Accessor , Mutator Methods

Typically, in OOP, data members in a class are defined as **`private`** to prevent users of the class accessing the data members directly. Instead, the creator of the class usually provides public methods for reading or changing some data members. Methods provided for other classes to read the values of data members are called "*accessor methods*", while methods provided for changing the values of data members are called "*mutator methods*".

toString()

Whenever an object of a class needs to be converted to its *String* representation, Java automatically calls a specific method called *toString()*. Therefore, in order to provide a meaningful *String* representation of the class we create, it is sensible to provide the method named exactly as *toString()* that returns the *String* representation we want.

Example

The following code shows a more complex class definition of *MyPoint*. The definition provide appropriate methods, including mutator methods, accessor methods, *toString()* as well as some other useful methods.

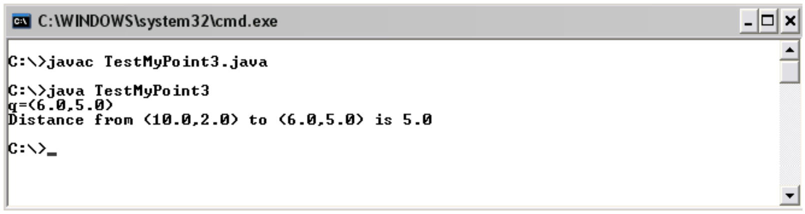
```
public class MyPoint                                              1
{                                                                 2
    // data members                                               3
    private double x;                                             4
    private double y;                                             5
                                                                  6
    // accessor methods                                           7
    public double getX(){                                         8
        return x;                                                 9
    }                                                            10
    public double getY(){                                        11
        return y;                                                12
    }                                                            13
                                                                 14
    // mutator methods                                           15
    public void setX(double x){                                  16
        this.x = x;                                              17
    }                                                            18
    public void setY(double y){                                  19
        this.y = y;                                              20
    }                                                            21
                                                                 22
    // other methods                                             23
    public void setLocation(double x, double y){                 24
        this.x = x;                                              25
        this.y = y;                                              26
    }                                                            27
    public double distanceTo(MyPoint p){                         28
        double diffXSquare = Math.pow((p.getX()-x),2);           29
        double diffYSquare = Math.pow((p.getY()-y),2);           30
        return Math.sqrt(diffXSquare+diffYSquare);               31
    }                                                            32
    public String toString(){                                    33
        return "("+x+","+y+")";                                  34
    }                                                            35
}                                                                36
```

The methods *getX()* and *getY()* declared on line 8 and line 11 allows other classes to read the values of the private variables **x** and **y**, respectively. These are accessor methods. The methods *setX()* and *setY()* declared on line 16 and line 19 allows other classes to set the values of the private variables **x** and **y**. These are mutator methods.

You should notice the usage of **this**. **this** is a reference used for referring to the current instance of the class. On line 17 and line 20, **this.x** and **this.y** refer to the instance variables **x** and **y** of the current instance, i.e. the instance from which the methods are invoked.

Now, let's use this class. Observe the following program and its output.

```
public class TestMyPoint3                                              1
{                                                                      2
    public static void main(String[] args)                             3
    {                                                                  4
        MyPoint p = new MyPoint();                                     5
        MyPoint q = new MyPoint();                                     6
        p.setX(6.0);                                                   7
        p.setY(5.0);                                                   8
        q.setLocation(p.getX(),p.getY());                              9
        System.out.println("q="+q);                                   10
        p.setLocation(10.0,2.0);                                      11
        System.out.print("Distance from "+p+" to ");                  12
        System.out.println(q+" is "+p.distanceTo(q));                 13
    }                                                                 14
}                                                                     15
```



```
C:\>javac TestMyPoint3.java

C:\>java TestMyPoint3
q=(6.0,5.0)
Distance from (10.0,2.0) to (6.0,5.0) is 5.0

C:\>_
```

On line 7, *setX()* is invoked from **p**. This set the value of **x** belonging to the *MyPoint* object referred to by **p** to the value input to the method. Similarly, the value of **y** belonging to the *MyPoint* object referred to by **p** is set to 5.0 on line 8.

On line 9, **p.getX()** and **p.getY()** return the value of the instance variables **x** and **y** belonging to the *MyPoint* object referred to by **p**. These values are used as input parameters to *setLocation()* invoked from **q**, in which the instance variable **x** of the object referred to by **q** is assigned with the first input parameter to the method, while the instance variable **y** of the object referred to by **q** is assigned with the other input parameter.

Whenever the *String* representation of a *MyPoint* object is needed, for example in argument lists of *print()* and *println()* on line 10, line 12, and line 13, *toString()* of that object is invoked.

Example

Static methods are also useful when we would like to build a class providing useful functionalities to be used by other classes or programs, such as the standard *Math* class. Such a class is not commonly instantiated, or in other words, it is not common to create an object of such a class. Therefore, the functionalities are provided through its public static methods.

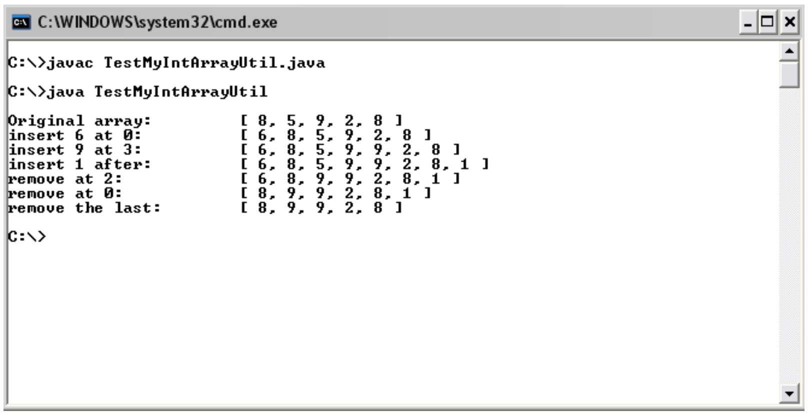
Here is a sample class made up for providing some functionality for **int** array manipulations. Note that this class serves as an example when static methods are used. There are some smarter ways to manipulate arrays.

```
public class MyIntArrayUtil
{
    public static int [] createRandomElements(int n,int min, int max){
        int [] a = new int[n];
        for(int i=0;i<n;i++){
            a[i] = (int)Math.round(Math.random()*(max-min)+min);
        }
        return a;
    }
    public static void showElements(int [] a)
    {
        System.out.print("[ "+a[0]);
        for(int i=1;i<a.length;i++){
            // continue on the next page
            System.out.print(", "+a[i]);                            14
        }                                                           15
        System.out.print(" ]\n");                                   16
    }                                                               17
    public static int [] removeAt(int [] a,int n){                  18
        if(n<0 || n>a.length-1) return a;                           19
        int [] b = new int[a.length-1];                             20
        for(int i=0;i<n;i++){                                       21
            b[i] = a[i];                                            22
        }                                                           23
        for(int i=n+1;i<a.length;i++){                              24
            b[i-1] = a[i];                                          25
        }                                                           26
        return b;                                                   27
    }                                                               28
    public static int [] insertAt(int [] a, int n, int k){          29
        if(n<0 || n>a.length) return a;                             30
        int [] b = new int[a.length+1];                             31
        for(int i=0;i<n;i++){                                       32
            b[i] = a[i];                                            33
        }                                                           34
        b[n] = k;                                                   35
        for(int i=n;i<a.length;i++){                                36
            b[i+1] = a[i];                                          37
        }                                                           38
        return b;                                                   39
    }                                                               40
}                                                                   41
```

The class *MyIntArrayUtil* created here contains four public static methods. The first one defined on line 3 creates an **int** array of length **n** whose elements are integer randomly chosen from **min** to **max**, inclusively. The method defined on line 10 prints all elements of the input array on screen. The method defined on line 18 removes the element at a specified position. Defined on line 19, the method inserts a given value to a specified position of the input array.

The following program makes use of the public static methods in *MyIntArrayUtil*. Observe the output of the program by yourself.

```
public class TestMyIntArrayUtil
{
    public static void main(String[] args)
    {
        System.out.print("\nOriginal array:\t\t");
        int [] a = MyIntArrayUtil.createRandomElements(5,1,10);
        MyIntArrayUtil.showElements(a);
        System.out.print("insert 6 at 0:\t\t");
        a = MyIntArrayUtil.insertAt(a,0,6);
        MyIntArrayUtil.showElements(a);
        System.out.print("insert 9 at 3:\t\t");
        a = MyIntArrayUtil.insertAt(a,3,9);
        MyIntArrayUtil.showElements(a);
        System.out.print("insert 1 after:\t\t");
        a = MyIntArrayUtil.insertAt(a,a.length,1);
        MyIntArrayUtil.showElements(a);
        System.out.print("remove at 2:\t\t");
        a = MyIntArrayUtil.removeAt(a,2);
        MyIntArrayUtil.showElements(a);
        System.out.print("remove at 0:\t\t");
        a = MyIntArrayUtil.removeAt(a,0);
        MyIntArrayUtil.showElements(a);
        System.out.print("remove the last:\t");
        a = MyIntArrayUtil.removeAt(a,a.length-1);
        MyIntArrayUtil.showElements(a);
    }
}
```



```
C:\>javac TestMyIntArrayUtil.java

C:\>java TestMyIntArrayUtil

Original array:         [ 8, 5, 9, 2, 8 ]
insert 6 at 0:          [ 6, 8, 5, 9, 2, 8 ]
insert 9 at 3:          [ 6, 8, 5, 9, 9, 2, 8 ]
insert 1 after:         [ 6, 8, 5, 9, 9, 2, 8, 1 ]
remove at 2:            [ 6, 8, 9, 9, 2, 8, 1 ]
remove at 0:            [ 8, 9, 9, 2, 8, 1 ]
remove the last:        [ 8, 9, 9, 2, 8 ]

C:\>
```

Constructors

Constructors are special methods invoked whenever an object of the class is created. Constructors are defined in the same fashion as defining methods. However, constructors must have the same name as the class name, there must not be any return types specified at the header of the constructors, and they have to be **public**. Constructors are usually for initializing or setting instance variables in that class. How they are set is described in the body of the constructor.

Below is an example of a no-argument (no input) constructor for *MyPoint*.

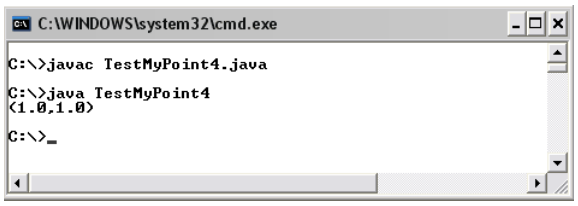
```
public MyPoint(){
    x = 1.0;
    y = 1.0;
}
```

Adding this constructor to the class definition of *MyPoint*, we obtain:

```
public class MyPoint
{
    // data members
    private double x;
    private double y;

    // constuctors
    public MyPoint(){
        x = 1.0;
        y = 1.0;
    }

    // ………………………………………… Here, details are omitted…………………………………

    public String toString(){
        return "("+x+","+y+")";
    }
}
```

Once *MyPoint* is defined this way, let's observe the result of the following program.

```
public class TestMyPoint4
{
    public static void main(String[] args)
    {
        MyPoint p = new MyPoint();
        System.out.println(p);
    }
}
```



```
C:\>javac TestMyPoint4.java

C:\>java TestMyPoint4
(1.0,1.0)

C:\>_
```

From the output, we can see that the values of **x** and **y** belonging to the *MyPoint* object referred to by **p** are both 1.0. The values are set in the constructor when it is called due to the creation of a new *MyPoint* instance. It should be obvious now that operations that should be performed once an instance of the class is created can be put in a constructor.

Constructors can be overloaded just like methods. A class can have multiple constructors with different input arguments. Which constructor to be called when an instance of the class is created depends on the input arguments of the *new* statement. Considered a new class definition of *MyPoint* listed below when overloaded constructors are added.

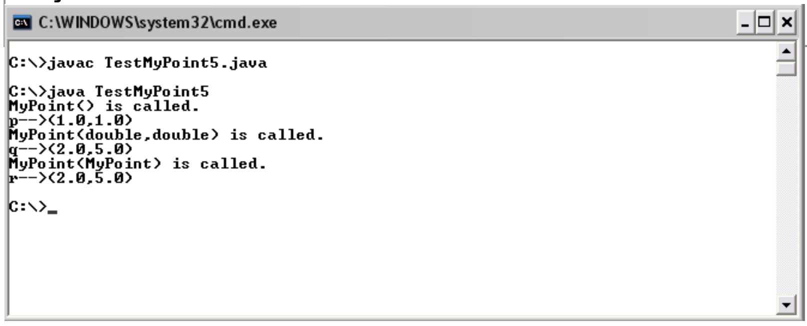
```
public class MyPoint
{
    // data members
    private double x;
    private double y;

    // constructors
    public MyPoint(){
        x = 1.0;
        y = 1.0;
        System.out.println("MyPoint() is called.");
    }
    public MyPoint(double x,double y){
        this.x = x;
        this.y = y;
        System.out.println("MyPoint(double,double) is called.");
    }
    public MyPoint(MyPoint p){
        x = p.getX();
        y = p.getY();
        System.out.println("MyPoint(MyPoint) is called.");
    }

    // ……………………………………………… Here, details are omitted……………………………………

    public String toString(){
        return "("+x+","+y+")";
    }
}
```

The first constructor, **MyPoint()**, does not take any input arguments. Therefore, it is called via the statement **new Mypoint()**. Such a constructor is usually called a *no-argument constructor*. **MyPoint(double x, double y)** is a constructor that takes two **double** values as its input. It is called via the statement **new Mypoint(a,b)**, where **a** and **b** are any double values. This constructor initializes the instance variables to the input values. Such a constructor that requires the values of the instance variables as its input is usually referred to as a *detailed constructor*. The last constructor shown above is **MyPoint(MyPoint q).** This constructor is invoked as a response to the statement **new Mypoint(c)**, where c is an instance of *MyPoint*. In this constructor the value of **x** is set to the value of **x** from the instance of *MyPoint* supplied as the input to the constructor, and the value of **y** is set to the value of **y** from the same instance. Such a constructor that copies all attributes from the input instance is usually referred to as a *copy constructor*. Just like method overloading, you should notice that constructors are not limited to the ones shown in this example. Also note that we add an invocation of *println()*

inside each of the constructor to observe that which one of the constructors is invoked when each instance is created.

Observe the output of the following program by yourself. Pay attention to the order of constructors invoked through the messages printed out on the screen.

```
public class TestMyPoint5
{
    public static void main(String[] args)
    {
        MyPoint p = new MyPoint();
        System.out.println("p-->"+p);
        MyPoint q = new MyPoint(2.0,5.0);
        System.out.println("q-->"+q);
        MyPoint r = new MyPoint(q);
        System.out.println("r-->"+r);
    }
}
```



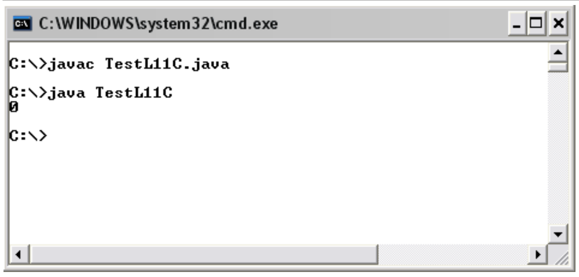
```
C:\>javac TestMyPoint5.java

C:\>java TestMyPoint5
MyPoint() is called.
p-->(1.0,1.0)
MyPoint(double,double) is called.
q-->(2.0,5.0)
MyPoint(MyPoint) is called.
r-->(2.0,5.0)

C:\>_
```

When there is no constructor provided, Java automatically adds a default no-argument constructor, inside which all variables in the class are initialized with default values based on their data types (zero for numeric data type, **`false`** for **`boolean`**, and **`null`** for non-primitive types). However, if there is at least one constructor defined in the class, the default no-argument will not be added automatically.

The following code runs fine since the compiler automatically adds a default no-argument constructor which is called in response to **`new L11C()`**.

```
public class L11C
{
    private int a;

    public int getA(){
        return a;
    }
}
```

```
public class TestL11C
{
    public static void main(String[] args)
    {
        L11C x = new L11C();
        System.out.println(x.getA());
    }
}
```

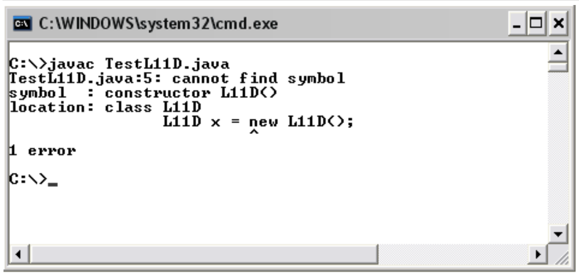


```
C:\>javac TestL11C.java

C:\>java TestL11C
0

C:\>
```

However, the following code leads to compilation error since the compiler cannot find any constructors for `new L11C()`. The compiler does not add a default no-argument constructor automatically since a constructor has already been defined.

```
public class L11D
{
    private int a;

    public L11D(int a){
        this.a = a;
    }

    public int getA(){
        return a;
    }
}
```

```
public class TestL11D
{
    public static void main(String[] args)
    {
        L11D x = new L11D();
        System.out.println(x.getA());
    }
}
```



```
C:\>javac TestL11D.java
TestL11D.java:5: cannot find symbol
symbol  : constructor L11D()
location: class L11D
                L11D x = new L11D();
                         ^
1 error

C:\>_
```

Calling a Constructor from Other Constructors

A constructor can be invoked within another constructor using *this(*`[argument list]`*)*, where `[argument list]` is the list of arguments corresponding to the argument list of the constructor to be called. Given a detailed constructor, other constructors can be implemented by purely calling the detailed constructor. There is one limitation that you need to keep in mind. If the invocation of a constructor via *this()* statement is used, the statement must be the first statement in the constructor. Otherwise, it will lead to a compilation error. The constructors of *MyPoint* listed below works in a similar fashion to the one described in the last example (except for the omission of *println()*).

```
// constructors
public MyPoint(){
    this(1.0,1.0);
}
public MyPoint(double x,double y){
    this.x = x;
    this.y = y;
}
public MyPoint(MyPoint p){
    this(p.getX(),p.getY());
}
```

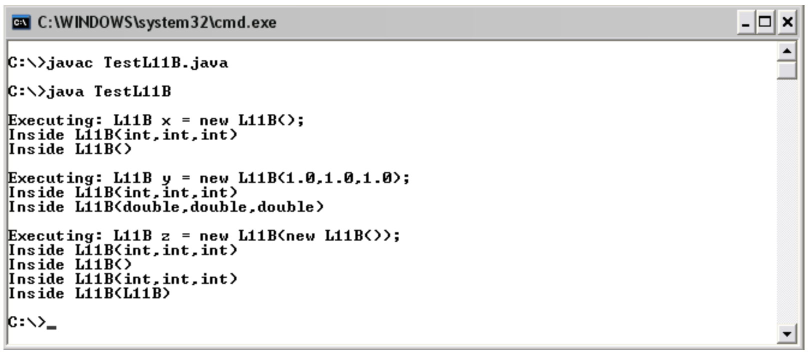
Example

Consider the class definition listed below.

```
public class L11B
{
    private int a,b,c;
    public L11B(){
        this(1,2,3);
        System.out.println("Inside L11B()");
    }
    public L11B(int a,int b, int c){
        this.a = a;
        this.b = b;
        this.c = c;
        System.out.println("Inside L11B(int,int,int)");
    }
    public L11B(double a,double b,double c){
        this((int)Math.round(a),(int)Math.round(b),(int)Math.round(c));
        System.out.println("Inside L11B(double,double,double)");
    }
    public L11B(L11B x){
        this(x.a,x.b,x.c);
        System.out.println("Inside L11B(L11B)");
    }
}
```

Now observe the program listed below and make sure you can follow series of constructor invocation through what are printed on the screen.

```
public class TestL11B
{
    public static void main(String[] args)
    {
        System.out.println("\nExecuting: L11B x = new L11B();");
        L11B x = new L11B();
        System.out.println("\nExecuting: L11B y = new L11B(1.0,1.0,1.0);");
        L11B y = new L11B(1.0,1.0,1.0);
        System.out.println("\nExecuting: L11B z = new L11B(new L11B());");
        L11B z = new L11B(new L11B());
    }
}
```



```
C:\>javac TestL11B.java

C:\>java TestL11B

Executing: L11B x = new L11B();
Inside L11B(int,int,int)
Inside L11B()

Executing: L11B y = new L11B(1.0,1.0,1.0);
Inside L11B(int,int,int)
Inside L11B(double,double,double)

Executing: L11B z = new L11B(new L11B());
Inside L11B(int,int,int)
Inside L11B()
Inside L11B(int,int,int)
Inside L11B(L11B)

C:\>_
```

Example

A complex number is of the form $a+jb$, where a and b are real numbers, and j is a quantity representing $\sqrt{-1}$. We would like to define a new class for complex numbers. Complex numbers are added, subtracted, and multiplied by formally applying the associative, commutative and distributive laws of algebra, together with the equation $j^2 = -1$. Therefore,

$$\begin{aligned}(a+jb)+(c+jd) &= (a+c)+j(b+d)\\(a+jb)-(c+jd) &= (a-c)+j(b-d) \quad .\\(a+jb)(c+jd) &= (ac-bd)+j(bc+ad)\end{aligned}$$

The reciprocal or multiplicative inverse of a complex number can be written as:

$$(a+jb)^{-1} = \left(\frac{a}{a^2+b^2}\right) + j\left(\frac{-b}{a^2+b^2}\right),$$

when the complex number is non-zero.

Division between two complex numbers is defined as:

$$\frac{(a+jb)}{(c+jd)} = (a+jb)(c+jd)^{-1}.$$

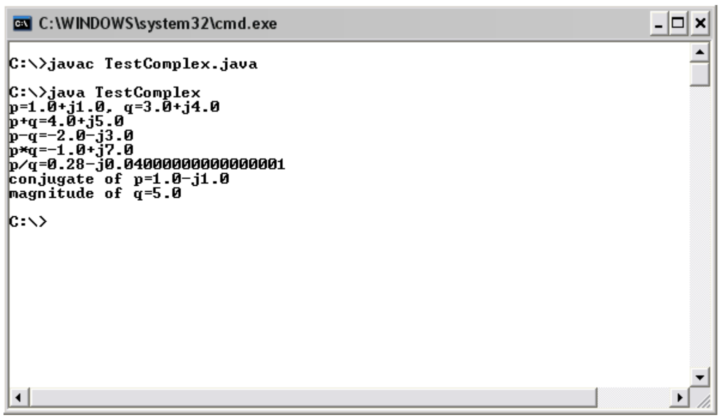
Complex conjugate of a complex number $a+jb$ is $a-jb$, while the magnitude of $a+jb$ is calculated by $\sqrt{(a^2+b^2)}$.

Here is an example of the class definition for *Complex*, a class we use for representing complex numbers.

```
public class Complex
{
    // attributes: (re) + j(im)
    private double re;
    private double im;

    // constructors
    public Complex(){
        this(0,0);
    }
    public Complex(double r, double i){
        re = r;
        im = i;
    }
    public Complex(Complex z){
        this(z.getRe(),z.getIm());
    }

    //accessor methods
    public double getRe(){
        return re;
    }
    public double getIm(){
        return im;
    }

    //mutator methods
    public void setRe(double r){
        re = r;
    }
    public void setIm(double i){
        im = i;
    }

    //other methods
    public Complex adds(Complex z){
        return new Complex(re+z.getRe(),im+z.getIm());
    }
    public Complex subtracts(Complex z){
        return new Complex(re-z.getRe(),im-z.getIm());
    }
    public Complex multiplies(Complex z){
        double r = re*z.getRe()-im*z.getIm();
        double i = im*z.getRe()+re*z.getIm();
        return new Complex(r,i);
    }
    // continue on the next page
    public Complex divides(Complex z){
        return this.multiplies(z.multInverse());
    }
    public Complex multInverse(){
        double den = Math.pow(this.magnitude(),2);
        return new Complex(re/den,-im/den);
    }
    public Complex conjugate(){
        return new Complex(re,-im);
    }
    public double magnitude(){
        return Math.sqrt(re*re+im*im);
    }
    public String toString(){
        if(im>=0)
            return re+"+j"+im;
        else
            return re+"-j"+(-im);
    }
}
```

The following program shows the class *Complex* in action.

```
public class TestComplex
{
    public static void main(String[] args)
    {
        Complex p = new Complex(1,1);
        Complex q = new Complex(3,4);
        System.out.println("p="+p+", q="+q);
        System.out.println("p+q="+p.adds(q));
        System.out.println("p-q="+p.subtracts(q));
        System.out.println("p*q="+p.multiplies(q));
        System.out.println("p/q="+p.divides(q));
        System.out.println("conjugate of p="+p.conjugate());
        System.out.println("magnitude of q="+q.magnitude());
    }
}
```



```
C:\>javac TestComplex.java

C:\>java TestComplex
p=1.0+j1.0, q=3.0+j4.0
p+q=4.0+j5.0
p-q=-2.0-j3.0
p*q=-1.0+j7.0
p/q=0.28-j0.04000000000000001
conjugate of p=1.0-j1.0
magnitude of q=5.0

C:\>
```