Building Java Programs

Chapter 8: Classes

Classes, types, and objects

- class:
  1. A module that can be run as a program.
  2. A template for a type of objects.

- We can write Java classes that are not programs in themselves, but instead define new types of objects.
  - We can use these objects in our programs if we so desire.

- Why would we want to do this?

Chapter outline

- objects, classes, and object-oriented programming
  - relationship between classes and objects
  - abstraction

- anatomy of a class
  - fields
  - instance methods
  - constructors
  - encapsulation

- advanced classes
  - preconditions, postconditions, and invariants
  - special methods: toString and equals
  - the keyword this

Objects and "OOP"

- object: An encapsulation of data and behavior.

- object-oriented programming (OOP): Writing programs that perform most of their useful behavior through interactions with objects.

- So far, we have interacted with objects such as:
  - String
  - Point
  - Stacker
  - DrawingPanel
  - Graphics
  - Color
  - Random
  - File
  - PrintStream

Abstraction

- abstraction: A distancing between ideas and details.
  - The objects in Java provide a level of abstraction, because we can use them without knowing how they work.

- You use abstraction every day when interacting with technological objects such as a portable music player.
  - You understand its external behavior (buttons, screen, etc.)
  - You DON'T understand its inner workings, nor do you need to.

Factory/blueprint analogy

- In real life, a factory can create many similar objects.
  - These are like following a blueprint.

- Music player factory
  - Uses different methods to build a music player
  - create
Recall: Point objects

- Java has a class of objects named `Point`
- To use `Point`, you must write: `import javax.swing.*;`
- Constructing a `Point` object, general syntax:
  ```java
  Point <name> = new Point(x, y);
  ```
  ```java
  Point <name> = new Point(); // the origin, (0, 0)
  ```
- Examples:
  ```java
  Point p1 = new Point(3, -2);
  Point p2 = new Point();
  ```
- Point objects are useful for several reasons:
  - They store two values, an (x, y) pair, in a single variable.
  - They have useful methods we can call in our programs.

Recall: Point data/methods

- Data stored in each `Point` object:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>the point's x-coordinate</td>
</tr>
<tr>
<td>y</td>
<td>the point's y-coordinate</td>
</tr>
</tbody>
</table>

- Useful methods of each `Point` object:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance()</td>
<td>how far away the point is from point p</td>
</tr>
<tr>
<td>setLocation(x, y)</td>
<td>sets the point's x and y to the given values</td>
</tr>
<tr>
<td>setLocation(x, y)</td>
<td>moves the point's x and y by the given amounts</td>
</tr>
</tbody>
</table>

- Point objects can also be printed using `println` statements:
  ```java
  Point p = new Point(3, -2);
  System.out.println(p); // java.awt.Point[x=3,y=-2]
  ```

A Point class

- A `Point` class might look something like this:
- Each object contains its own data and methods.
- The class has the instructions for how to construct individual objects.

Object state: fields

- reading: 8.2

Point class, version 1

- The following code creates a new class named `Point`.
  ```java
  public class Point {
      int x;
      int y;
      
      // We'd save this code into a file named Point.java.
      
      // Each object contains two pieces of data:
      // * x is named x
      // * y is named y
      // * Point objects [so far] do not contain any behavior.
  }
  ```

Fields

- field: A variable inside an object that represents part of the internal state of the object.
- Each object will have its own copy of the data fields we declare.

- Declaring a field, general syntax:
  ```java
  <type> <name>;
  ```

- Examples:
  ```java
  public class student {
      String name; // each student object has a name
      int gpa; // each student object has a gpa
  }
  ```
**Accessing fields**

- Code in other classes can access your object's fields.
  - Accessing a field, general syntax:
    `<variable name> . <field name>`
  - Modifying a field, general syntax:
    `<variable name> . <field name> = <value>`

**Examples:**

```java
System.out.println("The x-coordinate is "+ p1.x); // access
p2.y = 10; // modify
```

Later in this chapter, we’ll learn about encapsulation, which will change the way we access the data inside objects.

---

**Client code**

- `client code`: Code that uses an object.
  - The code below [PointMain.java] uses our `Point` class.
    ```java
    public class PointMain {
        public static void main(String[] args) {
            Point p1 = new Point(7, 3);
            Point p2 = new Point(4, 3);
            System.out.println("p1 is "+ p1.x+" , "+ p1.y); // print point p1
            System.out.println("p2 is "+ p2.x+" , "+ p2.y); // print point p2
            p2.x = 5; // modify
            System.out.println("p2 is "+ p2.x+" , "+ p2.y); // print point p2
        }
    }
    ```

**Output:**

```
Point p1 = (7, 3)
Point p2 = (4, 3)
p2.x = 5
```

---

**Object behavior: instance methods**

* reading: 8.3

---

**Client code question**

- Write a client program that uses our new `Point` class to produce the following output:
  
  ```java
  p1 is (7, 3)
  p1's distance from origin = 7.2010889820318
  p2 is (4, 3)
  p2's distance from origin = 5.0
  p1 is (18, 8)
  p2 is (5, 15)
  ```

  * Recall that the formula to compute distance between two points 
    
    \[ \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2} \]

---

**Client code redundancy**

* Our client program had code such as the following to translate a `Point` object's location.
  ```java
  // move p2 and then print it again
  p2.x = 5;
  System.out.println("p2 is (" + p2.x + ", " + p2.y + ");
  ```

  * If we translate several points, the above code would be repeated several times in the client program.

---

**Eliminating redundancy, v1**

* We could eliminate the redundancy with a static method
  in the client for translating point coordinates.
  ```java
  // Shifts the location of the given point.
  public static void translate(Point p, int dx, int dy) {
     p.x += dx;
     p.y += dy;
  }
  ```

  * Why doesn't the method need to return the modified point?

  * The client would call the method as follows:
    ```java
    // move p2 and then print it again
    translate(p2, 4, 8);
    System.out.println("p2 is (" + p2.x + ", " + p2.y + ");
    ```
Classes with behavior

- The static method solution isn’t a good idea:
  - The syntax doesn’t match the way we’re used to using objects:
    - translate(p2, 2, 4);
  - The whole point of giving classes the ability to store state and behavior together is to put related state and behavior together. This behavior is closely related to the x/y data of the Point object, so it belongs in the Point class.
- The objects we’ve used contain behavior inside them:
  - When we wanted to use that behavior, we called a method of the object using the dot notation:
    - // move p2 and then print it again
    - p2.translate(2, 4);
    - System.out.println("x = " + p2.x + ", y = " + p2.y);
  - In this section, we’ll see how to add methods to our Point objects.

Instance methods

- Instance method: a method (without the static keyword) that defines the behavior for each object.
  - An object can refer to its own fields or methods as necessary.
- Instance method declaration, general syntax:
  - public <Type> <name> <parameter(s)> 
  - {<statement(s)> ;
  - }
  
  - Example: (this code appears inside the Point class):
    - public void translate(int dx, int dy) {
    - ... }

Point object diagrams

- Think of each Point object as having its own copy of the translate method, which operates on that object’s state:
  - Point p1 = new Point();
  - p1.x = 7;
  - p1.y = 2;
  - Point p2 = new Point();
  - p2.x = 4;
  - p2.y = 3;

  - public void translate(int dx, int dy) {
    - this.x = this.x + dx;
    - this.y = this.y + dy;
    - }

  - Therefore the complete translate method should be:
    - public void translate(int dx, int dy) {
    - x = x + dx;
    - y = y + dy;
    - }

The implicit parameter

- Implicit parameter: The object on which an instance method is called:
  - Each instance method call happens on a particular object:
    - During the call p1.translate(4, 6), the object referred to by p1 is the implicit parameter.
    - During the call p2.translate(4, 6), the object referred to by p2 is the implicit parameter.
  - The instance method can refer to that object’s fields (If we sometimes say that instance method code operates in the context of a particular object on each call.)
- Therefore the complete translate method should be:
  - public void translate(int dx, int dy) {
  - x = x + dx;
  - y = y + dy;
  - }

Tracing instance method calls

- What happens when the following calls are made?
  - p1.translate(1, 6);
  - p2.translate(1, 7);

  - public void translate(int dx, int dy) {
    - x = x + dx;
    - y = y + dy;
    - }

  - Each Point object now contains one method of behavior, which modifies its x and y coordinates by the given parameter values.

Point class, version 2

- This second version of Point gives a method named translate to each Point object:
  - public class Point {
    - int x;
    - int y;
    - // Changes the location of this Point object.
    - public void translate(int dx, int dy) {
      - x = x + dx;
      - y = y + dy;
    - }
    - }

    - Each Point object now contains one method of behavior, which modifies its x and y coordinates by the given parameter values.
Instance method questions

- Write an instance method named `distanceFromOrigin` that computes and returns the distance between the current `Point` object and the origin, (0, 0).
- Use the following formula: \( \sqrt{x^2 + y^2} \)
- Write an instance method named `distance` that accepts a `Point` as a parameter and computes the distance between it and the current `Point`.
- Write an instance method named `newLocation` that accepts x and y values as parameters and changes the `Point`'s location to be those values.
- You may wish to refactor your `Point` class to use this method.
- Modify the client code to use these new methods.

Accessors and mutators

Two common categories of instance methods:

- **accessor:** A method that provides access to information about an object.
  - Generally the information comes from (or is computed using) the object's state stored in its fields.
  - The `distanceFromOrigin` and distance methods are accessors.
- **mutator:** A method that modifies an object's state.
  - Sometimes the modification is based on parameters that are passed to the mutator method, such as the `translate` method with parameters for cx and cy.
  - The `translate` and `setLocation` methods are mutators.

Client code, version 2

The following client code (stored in `PointMain2.java`) uses our modified `Point` class:

```java
public class TestPoint2 {
    public static void main(String[] args) {
        // Create two Point objects
        Point p1 = new Point();
        Point p2 = new Point();
        p1.x = 5;
        p1.y = 3;
        // Print each point
        System.out.printf("p1: (%d, %d)\n", p1.x, p1.y);
        System.out.printf(" p2: (%d, %d)\n", p2.x, p2.y);
        // Move p1 and then print it again
        p1.x = 7;
        System.out.printf(" p1: (%d, %d)\n", p1.x, p1.y);
        System.out.printf(" p2: (%d, %d)\n", p2.x, p2.y);
    }
}
```

Client code question

- Recall our client program that produces this output:
  ```
  p1 is (7, 2)
  p1's distance from origin = 7.20833918935516
  p2 is (4, 3)
  p2's distance from origin = 5.0
  p1 is (1, 8)
  p2 is (5, 10)
  ```
- Modify the program to use our new instance methods. Also add the following output to the program:
  ```
  distance from p1 to p2 = 3.1422738011883795
  ```

Object initialization: constructors

reading: 8.4

Initializing objects

- It is tedious to have to construct an object and assign values to all of its data fields manually.
  ```java
  Point p = new Point();
  p.x = 8;
  p.y = 5;
  // tedious
  ```
- We'd rather be able to pass in the fields' values as parameters, as we did with Java's built-in `Point` class.
  ```java
  Point p = new Point(3, 8); // better!
  ```
- To do this, we need to learn about a special type of method called a constructor.
Constructors

- constructor: Initializes the state of new objects.
- Constructors may accept parameters to initialize the object.
- A constructor looks like a method, but it doesn't specify a return type, because it implicitly returns a new object.
- Constructor syntax:
  
  public <type> <parameter(s)> { 
  <statement(s)> 
  }

- Example:
  public Point(int initialX, int initialY) { 
  ... 
  } 

Point class, version 3

- This third version of the Point class provides a constructor to initialize points:
  
  public class Point { 
  int x;
  int y;
  public Point(int initialX, int initialY) { 
  x = initialX;
  y = initialY;
  }
  public void translate(int dx, int dy) { 
  x += dx;
  y += dy;
  }
  }

Tracing constructor calls

- What happens when the following call is made?
  
  Point pl = new Point(7, 2);

Client code question

- Recall our client program that produces this output:
  
  pl is (7, 2)
  pl's distance from origin = 7.2010988280518
  p2 is (4, 3)
  p2's distance from origin = 5.0
  pl is (10, 8)
  p2 is (5, 10)

- Modify the program to use our new constructor.

Encapsulation

- reading: 8.5
Encapsulation

- encapsulation: Hiding the implementation details of an object from the clients of the object.
- (Protecting the object's fields from modification by clients)
- Encapsulating objects provides abstraction; we can use them without knowing how they work. The object has:
  - an external view (its behavior)
  - an internal view (the state that accomplishes the behavior)

Implementing encapsulation

- Fields can be declared private to indicate that no code outside their own class can change them.
  - Declaring a private field, general syntax:
    ```java
class NameSpace {
    int x;
    }
```
  - Examples:
    ```java
    private int x;
    private String name;
    ```
  - Once fields are private, client code cannot directly access them. The client receives an error such as:
    ```java
    ...new Point(3, 4); x has private access......
    ```

Encapsulation and accessors

- Once fields are private, we often provide accessor methods to examine their values:
  ```java
  public int getX() {
    return x;
  }
  ```
  - This gives clients "read-only" access to the object's fields.
- If so desired, we can also provide mutator methods:
  ```java
  public void setX(int newValue) {
    x = newValue;
  }
  ```
  - Question: Is there any difference between a public field and a private field with a get and set method?

Benefits of encapsulation

- Encapsulation helps provide a clean layer of abstraction between an object and its clients.
- Encapsulation protects an object from unwanted access by clients.
  - For example, perhaps we write a program to manage users' bank accounts. We don't want a malicious client program to be able to arbitrarily change a bank account's balance.
- Encapsulation allows the class author to change the internal representation later if necessary.
  - For example, if we desired, the Point class could be rewritten to use polar coordinates (a radius \( r \) and an angle \( \theta \) from the origin), but the external view would remain the same.

Point class, version 4

```java
// A Point class is represented by x, y. 
// @version 4.0
public class Point {
  private int x, y; 
  public Point(int x, int y) {
    this.x = x; 
    this.y = y; 
  }
  public void setX(int newValue) {
    x = newValue;
  }
  public int getX() {
    return x;
  }
  public void setY(int newValue) {
    y = newValue;
  }
  public int getY() {
    return y;
  }
  public double distanceTo(Point p) {
    return Math.sqrt(Math.pow(x - p.x, 2) + Math.pow(y - p.y, 2));
  }
  public static void main(String[] args) {
    Point p = new Point(3, 4);
    System.out.println(p.distanceTo(new Point(1, 2))); 
  }
}
```