Building Java Programs

Chapter 3+5: Classes

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`

**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?

**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`
  - `Scanner`
  - `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.
  - This is also like following a blueprint.

```
Music player factory
state: # of players made
behavior: directions on how to build a music player
creates

Music player #1
state: station/song, volume, battery life
behavior: power on/off
change station/song
change volume
choose random song

Music player #2
state: station/song, volume, battery life
behavior: power on/off
change station/song
change volume
choose random song

Music player #3
state: station/song, volume, battery life
behavior: power on/off
change station/song
change volume
choose random song
```

Recall: Point objects

- Java has a class of objects named `Point`.
  - To use `Point`, you must write: `import java.awt.*;`

- Constructing a `Point` object, general syntax:
  
  ```
  Point <name> = new Point(<x>, <y>);
  Point <name> = new Point(); // the origin, (0, 0)
  ```

- Examples:
  - `Point p1 = new Point(5, -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(p)</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>translate(dx, dy)</td>
<td>adjusts the point's x and y by the given amounts</td>
</tr>
</tbody>
</table>

- `Point` objects can also be printed using `println` statements:
  - `Point p = new Point(5, -2);`
  - `System.out.println(p);` // java.awt.Point[x=5,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.

A Point class

```java
public class Point {
    public Point() {
    }
    public Point(int x, int y) {
    }
}
```

 Allows construction of

Point object #1
- state:
  - int x, y
- behavior:
  - distance(Point p)
  - equals(Point p)
  - setLocation(int x, int y)
  - toString()
  - translate(int dx, int dy)

Point object #2
- state:
  - int x, y
- behavior:
  - distance(Point p)
  - equals(Point p)
  - setLocation(int x, int y)
  - toString()
  - translate(int dx, int dy)

Point object #3
- state:
  - int x, y
- behavior:
  - distance(Point p)
  - equals(Point p)
  - setLocation(int x, int y)
  - toString()
  - translate(int dx, int dy)

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:
  - an int named x,
  - an int 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
    double gpa;    // name and gpa data field
}
```
Accessing fields

- Code in other classes can access your object's fields.
  - Accessing a field, general syntax:
    \[ \text{<variable name> . <field name>} \]
  - Modifying a field, general syntax:
    \[ \text{<variable name> . <field name> = <value> ;} \]

Examples:

```java
System.out.println("the x-coord is " + p1.x); // access
p2.y = 13; // 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 client code below (PointMain.java) uses our Point class.

```java
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point();
        p1.x = 5;
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;
        p2.y = 3;
        // print each point
        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ");
        // move p2 and then print it again
        p2.x += 2;
        p2.y += 4;
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ");
    }
}
```

OUTPUT:

- p1 is (5, 2)
- p2 is (4, 3)
- p1 is (6, 7)
- p2 is (6, 7)

Client code question

- Write a client program that uses our new Point class to produce the following output:
  - p1 is (7, 2)
  - p1's distance from origin = 7.280109889280518
  - p2 is (4, 3)
  - p2's distance from origin = 5.0
  - p1 is (18, 8)
  - p2 is (16, 10)

Recall that the formula to compute distance between two points \((x_1, y_1)\) and \((x_2, y_2)\) is:

\[
\sqrt{ (x_2 - x_1)^2 + (y_2 - y_1)^2 }
\]
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 += 2;
p2.y += 4;
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, 2, 4);
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.
  ```java
  translate(p2, 2, 4);
  ```
- The whole point of writing classes 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.
  ```java
  // move p2 and then print it again
  p2.translate(2, 4);
  System.out.println("p2 is (" + p2.x + ", " + 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.

- The object can refer to its own fields or methods as necessary.

**instance method declaration, general syntax**:

```java
public <type> <name> ( <parameter(s)> ) { <statement(s)> ; }
```

- Example (this code appears inside the Point class):
  ```java
  public void translate(int dx, int dy) {
      ...
  }
  ```
Think of each Point object as having its own copy of the translate method, which operates on that object’s state:

```java
Point p1 = new Point();
p1.x = 7;
p1.y = 2;
```

```java
Point p2 = new Point();
p2.x = 4;
p2.y = 3;
```

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(11, 6);`, the object referred to by `p1` is the implicit parameter.
  - During the call `p2.translate(1, 7);`, the object referred to by `p2` is the implicit parameter.
- The instance method can refer to that object’s fields. (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:

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

Tracing instance method calls

What happens when the following calls are made?

```java
p1.translate(11, 6);
p2.translate(1, 7);
```

Point class, version 2

- This second version of `Point` gives a method named `translate` to each `Point` object:

```java
public class Point {
    int x;
    int y;

    // Changes the location of this Point object.
    public void translate(int dx, int dy) {
        x += dx;
        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: 
  \[
  \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}
  \]

- 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 `setLocation` 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 `dx` and `dy`.
  - 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 PointMain2 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point();
        p1.x = 5;
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;
        p2.y = 3;

        // print each point
        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
    }
}
```

OUTPUT:

```
p1 is (5, 2) p2 is (4, 3) p2 is (6, 7)
```

Client code question

Recall our client program that produces this output:

```
p1 is (7, 2) p1's distance from origin = 7.280109889280518
p2 is (4, 3) p2's distance from origin = 5.0
p1 is (18, 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 = 3.1622776601683795
```
Object initialization: constructors

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 = 3;
p.y = 8;              // 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 `Point` object.
  
  - Constructor syntax:
  ```java
  public <class name> (<parameter(s)>) {
    <statement(s>)
  }
  ```

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

 Point class, version 3

This third version of the `Point` class provides a constructor to initialize `Point` objects:

```java
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 p1 = new Point(7, 2);

public Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
}

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

Client code, version 3

The following client code (stored in PointMain3.java) uses our Point constructor:

```java
public class PointMain3 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);
        // print each point
        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
    }
}
```

OUTPUT:
```
p1 is (5, 2) p2 is (4, 3)
p2 is (6, 7)
```

Client code question

Recall our client program that produces this output:
p1 is (7, 2)
p1's distance from origin = 7.280109889280518
p2 is (4, 3)
p2's distance from origin = 5.0
p1 is (18, 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
    private <type> <name>;
    ```
  - 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
    PointMain.java:11: x has private access in Point
    System.out.println("p1 is ("+ p1.x + "," + p1.y + 
    ^
    ```

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 newX) {
    x = newX;
  }
  ```
  - 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 BankAccount object’s balance.
- Encapsulation allows the class author to change the internal representation later if necessary.
  - For example, if so 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 could remain the same.
Point class, version 4

// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

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

Pre/postconditions

- **precondition**: Something that you assume to be true when your method is called.
- **postcondition**: Something you promise to be true when your method exits.
- Pre/postconditions are often documented as comments.
- **Example:**
  // Sets this Point's location to be the given (x, y).
  // **Precondition**: newX >= 0 && newY >= 0
  // **Postcondition**: x >= 0 && y >= 0
  public void setLocation(int newX, int newY) {
      x = newX;
      y = newY;
  }

Class invariants

- **class invariant**: An assertion about an object's state that is true throughout the lifetime of the object.
  - An invariant can be thought of as a postcondition on every constructor and mutator method of a class.
  - **Example**: "No BankAccount object's balance can be negative."
  - **Example**: Suppose we want to ensure that all Point objects' x and y coordinates are never negative.
    - We must ensure that a client cannot construct a Point object with a negative x or y value.
    - We must ensure that a client cannot move an existing Point object to a negative (x, y) location.
Violated preconditions

- What if your precondition is not met?
  - Sometimes the client passes an invalid value to your method.
  - Example:
    ```java
    Point pt = new Point(5, 17);
    Scanner console = new Scanner(System.in);
    System.out.print("Type the coordinates: ");
    int x = console.nextInt();  // what if the user types
    int y = console.nextInt();  // a negative number?
    pt.setLocation(x, y);
    ```

- How can we prevent the client from misusing our object?

Dealing with violations

- One way to deal with this problem would be to return
  out of the method if negative values are encountered.
  - However, it is not possible to do something similar in the
    constructor, and the client doesn’t expect this behavior.

- A more common solution is to have your object
  throw an exception.

- `exception`: A Java object that represents an error.
  - When a precondition of your method has been violated, you can
    generate (“throw”) an exception in your code.
  - This will cause the client program to halt. (That'll show 'em!)

Throwing exceptions example

- Throwing an exception, general syntax:
  ```java
  throw new <exception type> ();
  ```
  - or
  ```java
  throw new <exception type> ("<message>");
  ```
  - The `<message>` will be shown on the console when the
    program crashes.

- Example:
  ```java
  // Sets this Point's location to be the given (x, y).
  // Throws an exception if newX or newY is negative.
  // Postcondition: x >= 0 && y >= 0
  public void setLocation(int newX, int newY) {
    if (newX < 0 || newY < 0) {
      throw new IllegalArgumentException();
    }
    x = newX; y = newY;
  }
  ```

Encapsulation and invariants

- Encapsulation helps you enforce invariants.
  - Ensure that no Point is constructed with negative x or y:
    ```java
    public Point(int initialX, int initialY) {
      if (initialX < 0 || initialY < 0) {
        throw new IllegalArgumentException();
      }
      x = initialX;
      y = initialY;
    }
    ```

- Ensure that no Point can be moved to a negative x or y:
  ```java
  public void translate(int dx, int dy) {
    if (x + dx < 0 || y + dy < 0) {
      throw new IllegalArgumentException();
    }
    x += dx;
    y += dy;
  }
  ```

- Other methods require similar modifications.
Special instance methods:

- `toString`
- `equals`

Problem: object printability

- By default, Java doesn't know how to print the state of your objects, so it prints a strange result:

```java
Point p = new Point(10, 7);
System.out.println("p is " + p); // p is Point@9e8c34
System.out.println("p is " + p); // p is Point@9e8c34
```

- We can instead print a more complex string that shows the object's state, but this is cumbersome.

```java
System.out.println("(" + p.x + ", " + p.y + ")");
```

- We'd like to be able to print the object itself and have something meaningful appear.

  // desired behavior:
  System.out.println("p is " + p); // p is (10, 7)

The `toString` method

The special method `toString` tells Java how to convert your object into a `String` as needed.

- The `toString` method is called when your object is printed or concatenated with a `String`.

```java
Point pl = new Point(7, 2);
System.out.println("pl is " + pl);
```

- If you prefer, you can write the `.toString()` explicitly.

```java
System.out.println("pl is " + pl.toString());
```

- Every class contains a `toString` method, even if it isn't written in your class's code.

  - The default `toString` behavior is to return the class's name followed by a hexadecimal (base-16) number:

```java
Point@9e8c34
```

The `toString` method syntax

You can replace the default behavior by defining an appropriate `toString` method in your class.

- Example: The `Point` class in `java.awt` has a `toString` method that converts a `Point` into a `String` such as:

```java
"java.awt.Point[x=7,y=2]"
```

- The `toString` method, general syntax:

```java
public String toString() {
    <statement(s) that return an appropriate String>
}
```

  - The method must have this exact name and signature.

- Example:

  // Returns a String representing this Point.
  public String toString() {
      return "(" + x + ", " + y + ");
  }
Recall: comparing objects

- The `==` operator does not work well with objects.
  - `==` compares references to objects and only evaluates to `true` if two variables refer to the same object.
  - It doesn't tell us whether two objects have the same state.
- Example:
  ```java
  Point p1 = new Point(5, 3);
  Point p2 = new Point(5, 3);
  if (p1 == p2) { // false
      System.out.println("equal");
  }
  ```

The equals method

- The `equals` method compares the state of objects.
  - When we write our own new classes of objects, Java doesn't know how to compare their state.
  - The default `equals` behavior acts just like the `==` operator.
    ```java
    if (p1.equals(p2)) { // still false
        System.out.println("equal");
    }
    ```
- We can replace this default behavior by writing an `equals` method.
  - The method will actually compare the state of the two objects and return `true` for cases like the above.

Initial flawed equals method

- You might think that the following is a valid implementation of the `equals` method:
  ```java
  public boolean equals(Point other) {
      if (x == other.x && y == other.y) {
          return true;
      } else {
          return false;
      }
  }
  ```
  - However, it has several flaws that we should correct.
- One initial flaw: the body can be shortened to:
  ```java
  return x == other.x && y == other.y;
  ```
equals and the Object class

- `equals` should not accept a parameter of type `Point`.
  - It should be legal to compare `Point`s to any other object, e.g.:
    ```java
    Point p = new Point(7, 2);
    if (p.equals("hello")) {   // false
        ...
    }
    ```
- The `equals` method, general syntax:
  ```java
  public boolean equals(Object <name>) {
      <statement(s) that return a boolean value> ;
  }
  ```
  - The parameter to a proper `equals` method must be of type `Object` (meaning that an object of any type can be passed).
Another flawed version

You might think that the following is a valid implementation of the \texttt{equals} method:

```java
public boolean equals(Object o) {
    if (x == o.x && y == o.y) {
        return true;
    } else {
        return false;
    }
}
```

However, it does not compile.

```
Point.java:36: cannot find symbol
  symbol : variable x
location: class java.lang.Object
if (x == o.x && y == o.y) {
  ^
```

Type-casting objects

The object that is passed to \texttt{equals} can be cast from \texttt{Object} into your class's type.

- Example:

```java
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

Type-casting with objects behaves differently than casting primitive values.

- We are really casting a reference of type \texttt{Object} into a reference of type \texttt{Point}.
- We're promising the compiler that \texttt{o} refers to a \texttt{Point} object.

Casting objects diagram

Client code:

```java
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
if (p1.equals(p2)) {
    System.out.println("equal");
}
```

Comparing different types

Our \texttt{equals} code still is not complete.

- When we compare \texttt{Point} objects to any other type of objects,

```java
Point p = new Point(7, 2);
if (p.equals("hello")) {   // false
    ...
}
```

Currently the code crashes with the following exception:

```java
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
    at Point.equals(Point.java:25)
    at PointMain.main(PointMain.java:25)
```

The culprit is the following line that contains the type-cast:

```java
public boolean equals(Object o) {
    Point other = (Point) o;
```
The instanceof keyword

- We can use a keyword called `instanceof` to ask whether a variable refers to an object of a given type.
- The `instanceof` keyword, general syntax:
  `<variable> instanceof <type>`

- The above is a boolean expression that can be used as the test in an if statement.

Examples:

<table>
<thead>
<tr>
<th>expression</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>s instanceof Point</td>
<td>false</td>
</tr>
<tr>
<td>s instanceof String</td>
<td>true</td>
</tr>
<tr>
<td>p instanceof Point</td>
<td>true</td>
</tr>
<tr>
<td>p instanceof String</td>
<td>false</td>
</tr>
<tr>
<td>null instanceof String</td>
<td>false</td>
</tr>
</tbody>
</table>

Final version of equals method

- This version of the `equals` method allows us to correctly compare `Point` objects against any other type of object:

```java
public boolean equals(Object o) {
    if (o instanceof Point) {
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        return false;
    }
}
```

Using the keyword `this`

- `this`: A reference to the implicit parameter.
  - Recall: The implicit parameter is the object on which an instance method or constructor is being called.

- Usage of the `this` keyword, general syntax:
  - To refer to a field:
    ```java
    this.<field name>
    ```
  - To refer to a method:
    ```java
    this.<method name>(<parameters>);
    ```
  - To call a constructor from another constructor:
    ```java
    this(<parameters>);
    ```
Variable shadowing

- **shadowed variable**: A field that is "covered up" by a local variable or parameter with the same name.
  - Normally it is illegal to have two variables in the same scope with the same name, but in this case it is allowed.
  - To avoid shadowing, we named our `setLocation` parameters `newX` and `newY`:
    ```java
    public void setLocation(int newX, int newY) {
        if (newX < 0 || newY < 0) {
            throw new IllegalArgumentException();
        }
        x = newX;
        y = newY;
    }
    ```

Avoiding shadowing with this

- The `this` keyword lets us use the same names and still avoid shadowing:
  ```java
  public void setLocation(int x, int y) {
      if (x < 0 || y < 0) {
          throw new IllegalArgumentException();
      }
      this.x = x;
      this.y = y;
  }
  ```
  - When `this` is not seen, the parameter is used.
  - When `this` is seen, the field is used.

Multiple constructors

- It is legal to have more than one constructor in a class.
  - The constructors must accept different parameters.
    ```java
    public class Point {
        private int x;
        private int y;
        
        public Point() {
            x = 0;
            y = 0;
        }
        
        public Point(int initialX, int initialY) {
            x = initialX;
            y = initialY;
        }
        ...}
    ```

Multiple constructors w/ `this`

- One constructor can call another using `this`.
  - We can also use the `this` field syntax so that the constructor parameters' names can match the field names.
    ```java
    public class Point {
        private int x;
        private int y;
        
        public Point() {
            this(0, 0); // calls the (x, y) constructor
        }
        
        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
        ...}
    ```
More class problems

Object practice problem

- Create a class named **Circle**.
  - A circle is represented by a point for its center, and its radius.
  - Make it possible to construct the unit circle, centered at (0, 0) with radius 1, by passing no parameters to the constructor.
  - Circles should be able to tell whether a given point is contained inside them.
  - Circles should be able to draw themselves using Graphics.
  - Circles should be able to be printed on the console, and should be able to be compared to other circles for equality.

Object practice problem

- Create a class named **LineSegment**.
  - A line segment is represented by two endpoints \((x_1, y_1)\) and \((x_2, y_2)\).
  - A line segment should be able to compute its slope \((y_2-y_1) / (x_2-x_1)\).
  - A line segment should be able to tell whether a given point intersects it.
  - Line segments should be able to draw themselves using a Graphics object.
  - Line segments should be able to be printed on the console, and should be able to be compared to other lines for equality.

Object practice problem

- Create a class named **Calculator**.
  - A calculator has a method to add digits to a running total.
  - The user can also press operator keys such as + or * and then enter digits of a second number.
  - When the user presses the = button, the calculator computes the result based on the numbers entered so far and the operator chosen. The user can then make further computations.
Calculator client code

- Use your Calculator with a client such as the following:

```java
public class CalculatorMain {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        // first computation: calculate 329 + 1748 = 2077
        calc.addDigit(3);
        calc.addDigit(2);
        calc.addDigit(9);
        calc.setOperator("+");
        calc.addDigit(1);
        calc.addDigit(7);
        calc.addDigit(4);
        calc.addDigit(4);
        int result = calc.compute();
        System.out.println(calc);
        System.out.println("result = " + result);
    }
}
```