Building Java Programs

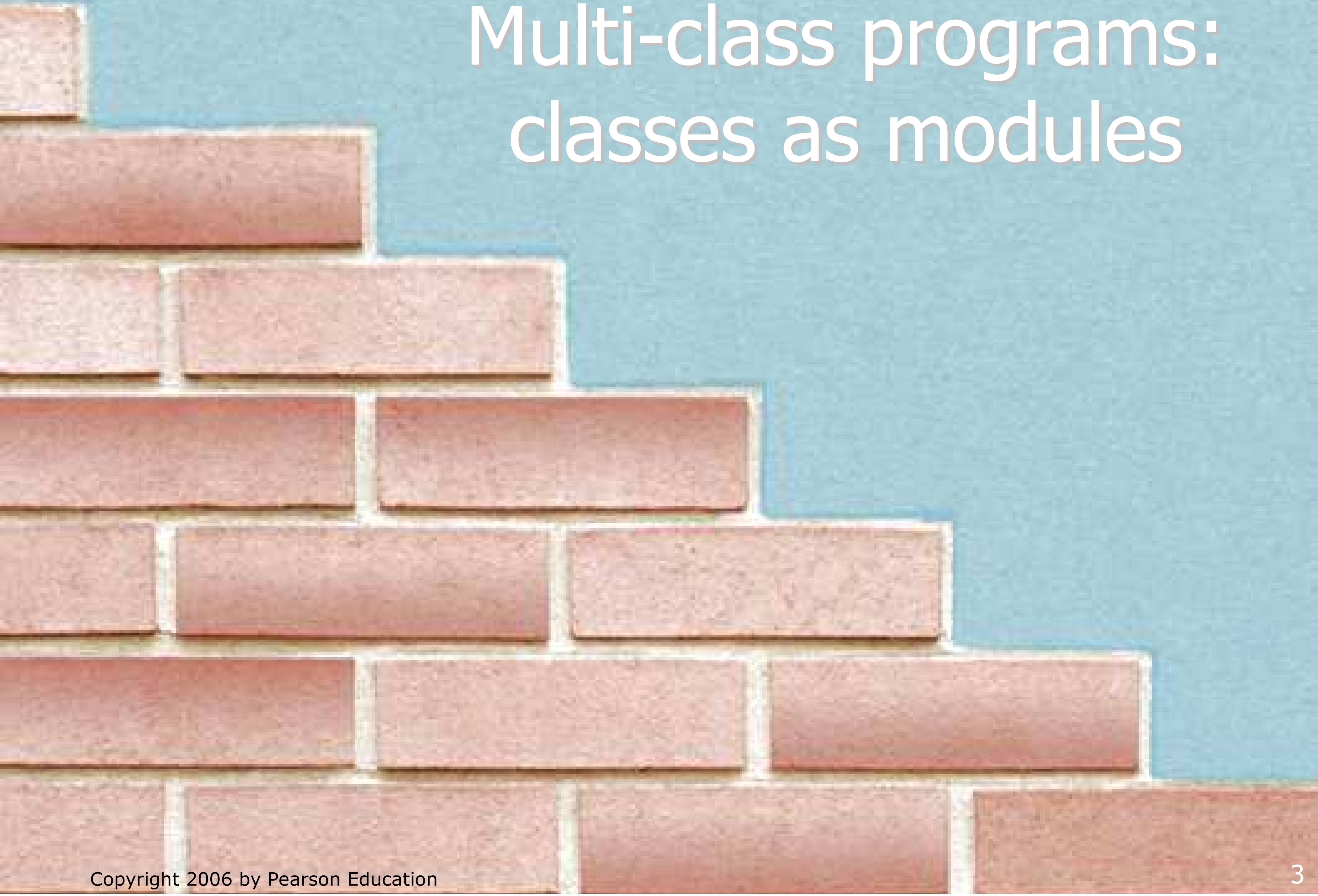
Chapter 8:
Classes and Objects

Lecture outline

- objects, classes, object-oriented programming
  - classes as modules (multi-class programs)
  - classes as types
  - relationship between classes and objects
  - abstraction

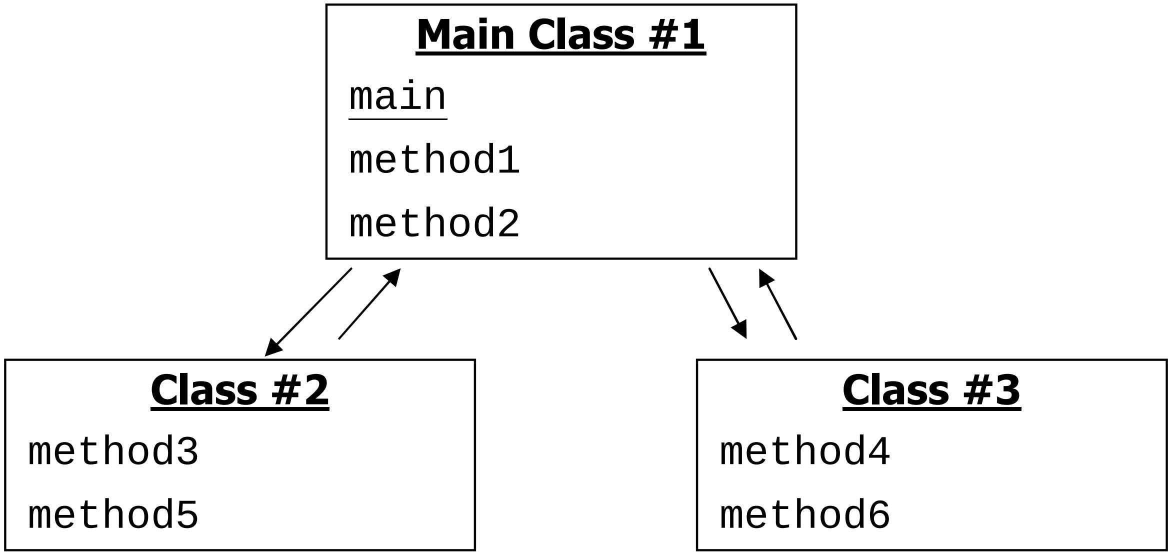
- anatomy of a class
  - fields
  - instance methods

Multi-class programs: classes as modules



Large software

- Most large software systems consist of many classes.
- One main class runs and calls methods of the others.
- Advantages:
  - code reuse
  - splits up the program logic into manageable chunks



Main Class #1

main

method1

method2

Class #2

method3

method5

Class #3

method4

method6

Redundant programs 1

- Consider the following program:

```java
// This program sees whether some interesting numbers are prime.
public class Primes {
    public static void main(String[] args) {
        int[] nums = {1234517, 859501, 53, 142};
        for (int i = 0; i < nums.length; i++) {
            if (isPrime(nums[i])) {
                System.out.println(nums[i] + " is prime");
            }
        }
    }

    // Returns the number of factors of the given integer.
    public static int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
            if (number % i == 0) {
                count++;   // i is a factor of the number
            }
        }
        return count;
    }

    // Returns true if the given number is prime.
    public static boolean isPrime(int number) {
        return countFactors(number) == 2;
    }
}
```

Redundant programs 2

- The following program is very similar to the first one:

```
// This program prints all prime numbers up to a maximum.
public class Primes2 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("Max number? ");
        int max = console.nextInt();
        for (int i = 2; i <= max; i++) {
            if (isPrime(i)) {
                System.out.print(i + " ");
        }   }
        System.out.println();
    }

    // Returns the number of factors of the given integer.
    public static int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
            if (number % i == 0) {
                count++;   // i is a factor of the number
        }   }
        return count;
    }

    // Returns true if the given number is prime.
    public static boolean isPrime(int number) {
        return countFactors(number) == 2;
    }
}
```

Classes as modules

- **module**: A reusable piece of software.
  - A class can serve as a module by containing common code.
  - Example module classes: `Math`, `Arrays`, `System`

```
// This class is a module that contains useful methods
// related to factors and prime numbers.
public class Factors {
    // Returns the number of factors of the given integer.
    // Assumes that a non-negative number is passed.
    public static int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
            if (number % i == 0) {
                count++;   // i is a factor of the number
            }
        }

        return count;
    }

    // Returns true if the given number is prime.
    public static boolean isPrime(int number) {
        return countFactors(number) == 2;
    }
}
```

More about modules

- A module is a partial program, not a complete program.
  - Modules generally do not have a `main` method.
  - You don't run a module from your Java editor directly.

- Modules are meant to be utilized by other classes.
  - We say that the other classes are **clients** (users) of the module.

  - Syntax for calling a module's static method:

    <class name> . ***<method name>*** (***<parameters>***)

  - Example:

```
int factorsOf24 = Factors.countFactors(24);
```

Using a module

- The redundant programs can now use the module:

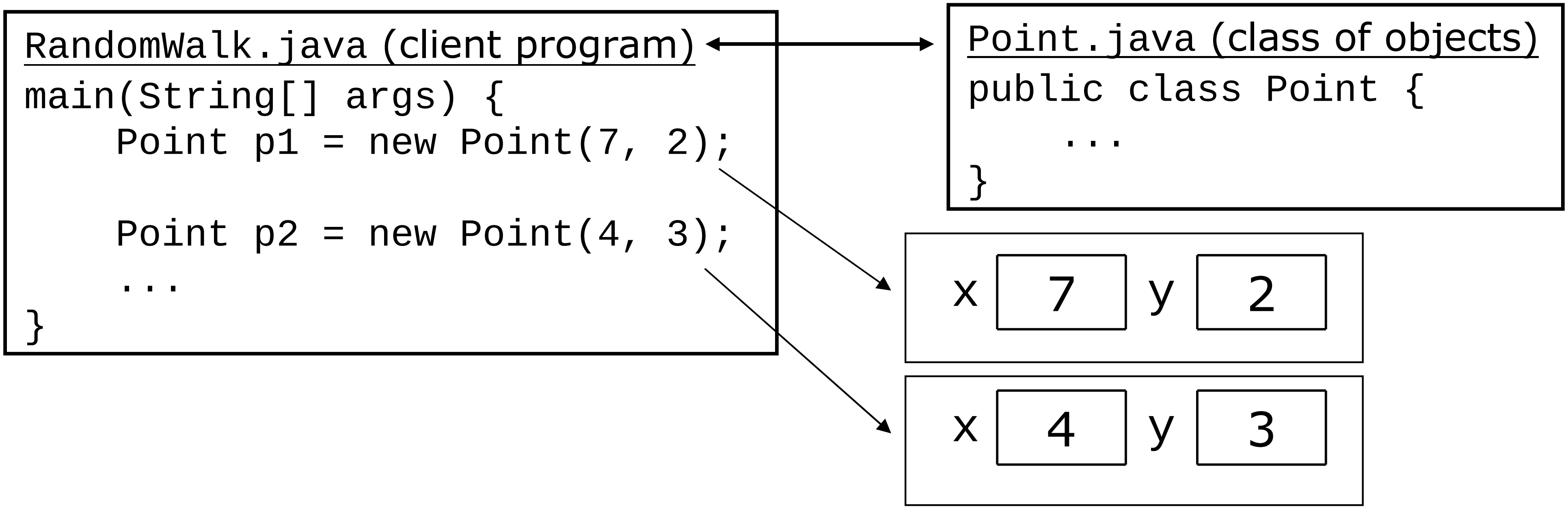
```
// This program sees whether some interesting numbers are prime.
public class Primes {
    public static void main(String[] args) {
        int[] nums = {1234517, 859501, 53, 142};
        for (int i = 0; i < nums.length; i++) {
            if (Factors.isPrime(nums[i])) {
                System.out.println(nums[i] + " is prime");
            }
        }
    }
}

// This program prints all prime numbers up to a maximum.
public class Primes2 {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("Max number? ");
        int max = console.nextInt();
        for (int i = 2; i <= max; i++) {
            if (Factors.isPrime(i)) {
                System.out.print(i + " ");
        }   }
        System.out.println();
    }
}
```

Creating types of objects: classes as types

- reading: 8.1

Using objects

- Many large programs benefit from using objects.
  - Your program is a **client** (user) of these objects.
  - Example: `RandomWalk` uses `Point` and `Random` objects.
  - Example: `PersonalityTest` uses `Scanner`, `File`, `PrintStream`.



RandomWalk.java (client program)

```
main(String[] args) {
    Point p1 = new Point(7, 2);

    Point p2 = new Point(4, 3);
    ...
}
```

Point.java (class of objects)

```
public class Point {
    ...
}
```

- What should we do if our program would benefit from a type of objects that *doesn't exist yet* in Java?
  - Example: `Birthdays` would benefit from `Date` objects.

Objects, classes, types

- **class**: A program entity that represents either:

  1. A program / module, or

  2. A template for a new type of objects.

  - classes of objects we've used so far:

    String, Point, Scanner, DrawingPanel, Graphics, Color, Random, File, PrintStream

  - We can write classes that define new types of objects.

- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Programs that perform most of their behavior as interactions between objects.

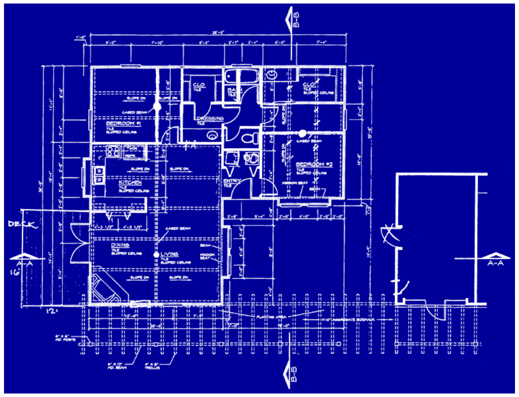
Blueprint analogy

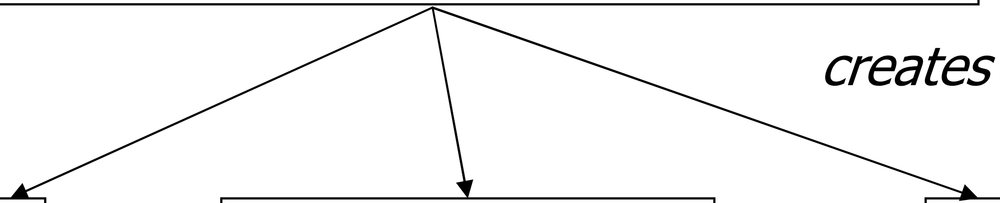
- A single blueprint can be used to create many objects.

Music player blueprint

state:
current song
volume
battery life

behavior:
power on/off
change station/song
change volume
choose random song





Music player #1

state:
song = "Thriller"
volume = 17
battery life = 2.5 hrs

behavior:
power on/off
change station/song
change volume
choose random song

Music player #2

state:
song = "Lovesong"
volume = 9
battery life = 3.41 hrs

behavior:
power on/off
change station/song
change volume
choose random song

Music player #3

state:
song = "Closer"
volume = 24
battery life = 1.8 hrs

behavior:
power on/off
change station/song
change volume
choose random song

Recall: Point objects

```
Point p1 = new Point(5, -2);
Point p2 = new Point();
```

- State (data) of each `Point` object:

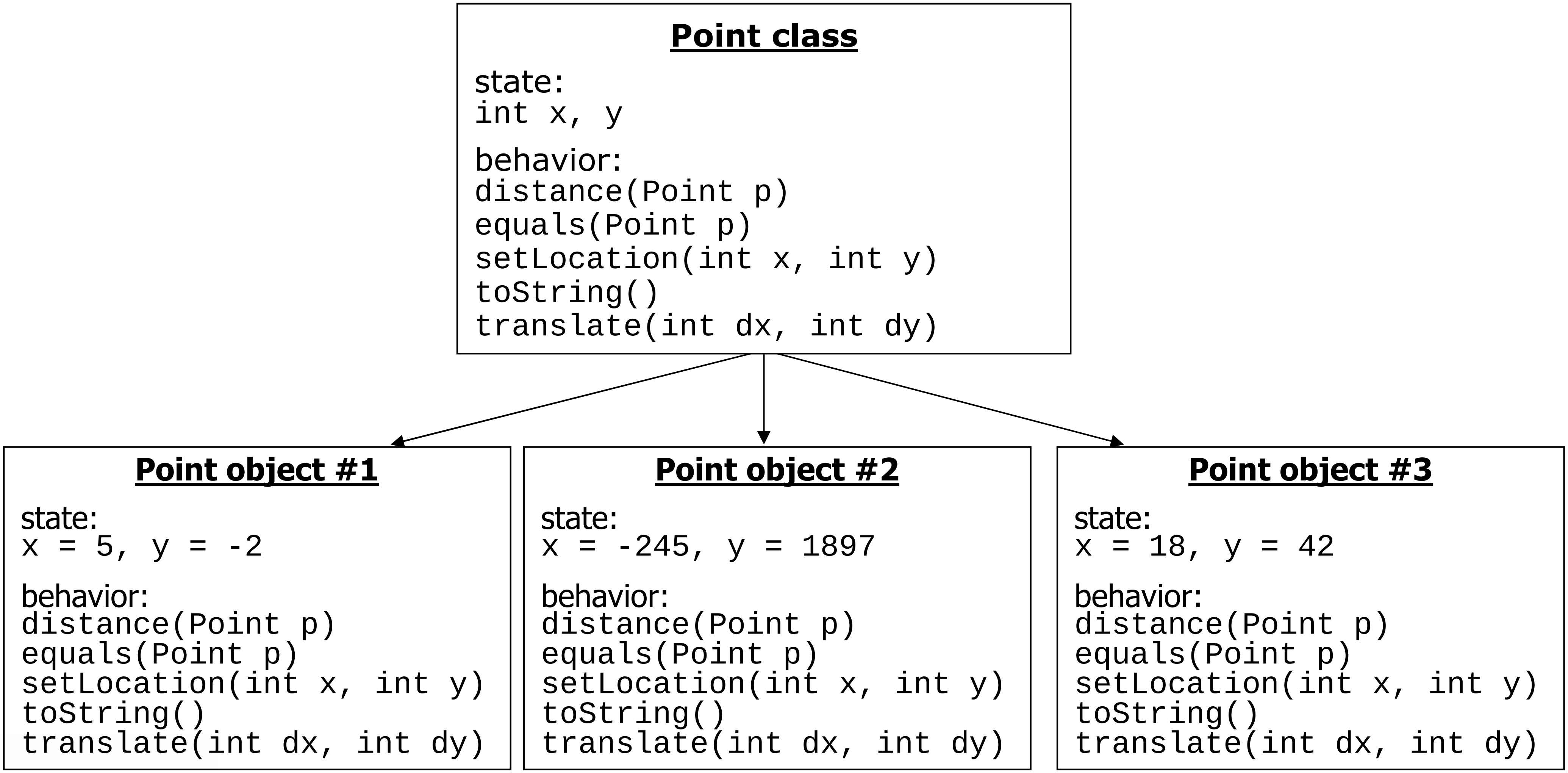
Field name	Description
`x`	the point's x-coordinate
`y`	the point's y-coordinate

- Behavior (methods) of each `Point` object:

Method name	Description
`distance(`*p*`)`	how far away the point is from point *p*
`setLocation(`*x*`, `*y*`)`	sets the point's x and y to the given values
`translate(`*dx*`, `*dy*`)`	adjusts the point's x and y by the given amounts

A Point class

- The class (blueprint) knows how to create objects.
- Each object contains its own data and methods.



Point class

state:
```
int x, y
```

behavior:
```
distance(Point p)
equals(Point p)
setLocation(int x, int y)
toString()
translate(int dx, int dy)
```

Point object #1

state:
```
x = 5, y = -2
```

behavior:
```
distance(Point p)
equals(Point p)
setLocation(int x, int y)
toString()
translate(int dx, int dy)
```

Point object #2

state:
```
x = -245, y = 1897
```

behavior:
```
distance(Point p)
equals(Point p)
setLocation(int x, int y)
toString()
translate(int dx, int dy)
```

Point object #3

state:
```
x = 18, y = 42
```

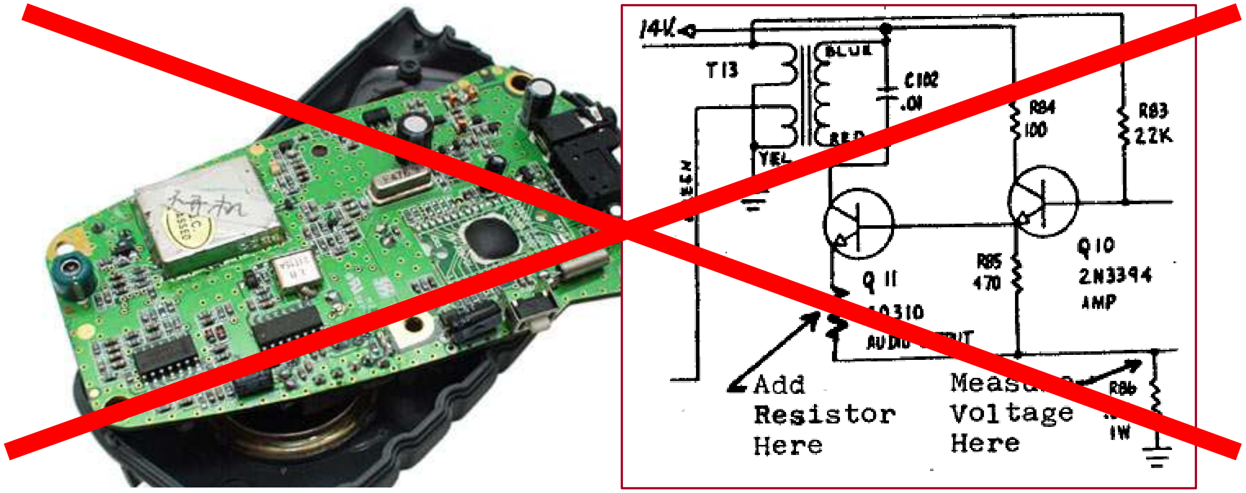
behavior:
```
distance(Point p)
equals(Point p)
setLocation(int x, int y)
toString()
translate(int dx, int dy)
```

Abstraction

- **abstraction**: A distancing between ideas and details.
  - Objects in Java provide abstraction:
    We can use them without knowing how they work.

- You use abstraction every day.

  Example: Your portable music player.
  - You understand its external behavior (buttons, screen, etc.)
  - You don't understand its inner details (and you don't need to).



Copyright 2006 by Pearson Education

Our task

- In the following slides, we will re-implement Java's Point class as a way of learning about classes.
  - We will define our own new type of objects named Point.
  - Each Point object will contain x/y data called **fields**.
  - Each Point object will contain behavior called **methods**.
  - Programs called **client programs** will use the Point objects.
- After we understand Point, we will also implement other new types of objects such as Date.

Object state: fields

reading: 8.2

Point class, version 1

- The following code creates a new class named `Point`.

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

  - Save this code into a file named `Point.java`.

- Each `Point` object contains two pieces of data:
  - an `int` named `x`,
  - an `int` named `y`.

  - `Point` objects do not contain any behavior (yet).

Fields

- **field**: A variable inside an object that holds part of its state.
  - Each object has *its own copy* of each field we declare.

- Declaring a field, syntax:

  <type> <name> ;

  - Example:

```
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 the object's fields.
  - Accessing a field, syntax:

    <variable name>* . *<field name>

  - Modifying a field, syntax:

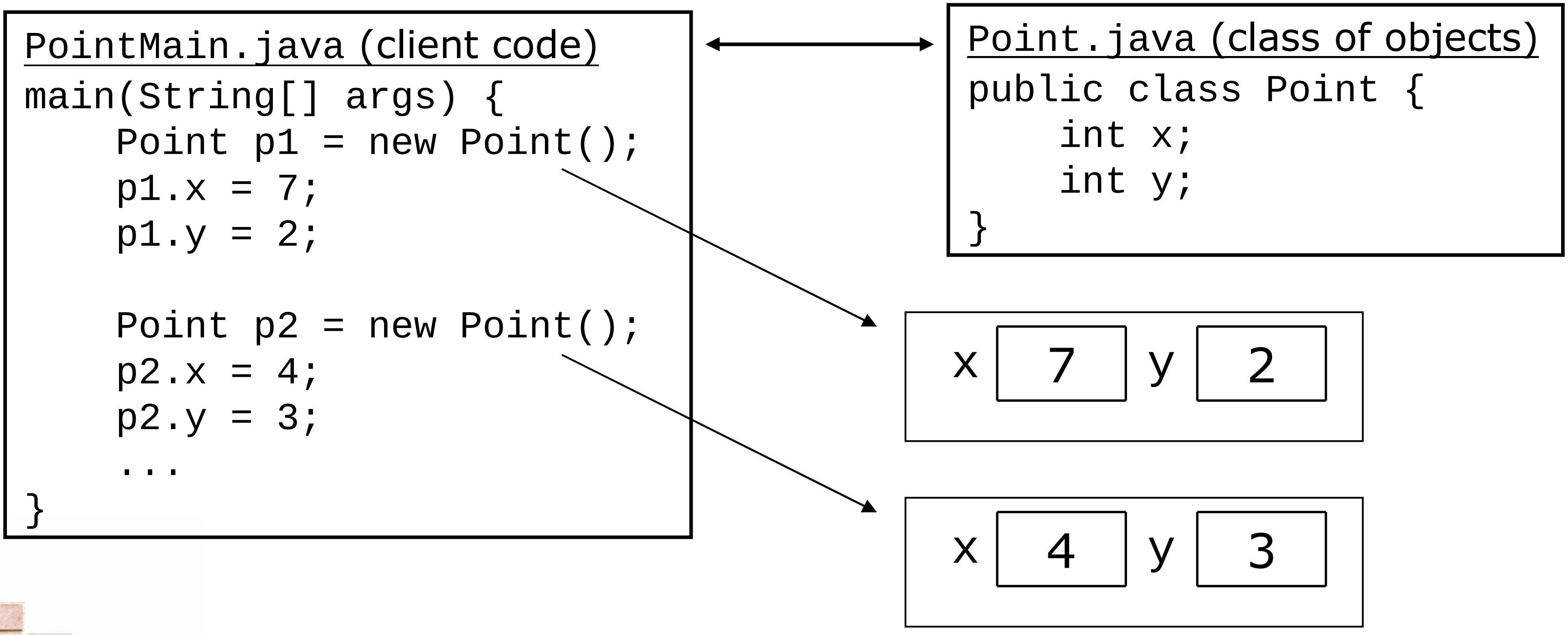
    <variable name>* . *<field name> = ***<value>*** ;

- Examples:

```
System.out.println("the x-coord is " + p1.x);    // access
p2.y = 13;                                       // modify
```

Recall: Client code

- Point.java is not, by itself, a runnable program.
  - Classes of objects are modules that can be used by other programs stored in separate .java files.
- **client code***:* Code that uses a class and its objects.
  - The client code is a runnable program with a main method.



PointMain.java (client code)

```
main(String[] args) {
    Point p1 = new Point();
    p1.x = 7;
    p1.y = 2;

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

Point.java (class of objects)

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

Point client code

- The client code below (`PointMain.java`) uses our `Point` class.

```
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point();
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;

        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");

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

OUTPUT:

```
p1 is (0, 2)
p2 is (6, 1)
```

More client code

```
public class PointMain2 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point();
        p1.x = 7;
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;
        p2.y = 3;

        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");

        // compute/print each point's distance from the origin
        double dist1 = Math.sqrt(p1.x * p1.x + p1.y * p1.y);
        double dist2 = Math.sqrt(p2.x * p2.x + p2.y * p2.y);
        System.out.println("p1's distance from origin = " + dist1);
        System.out.println("p2's distance from origin = " + dist2);

        // move p1 and p2 and print them again
        p1.x += 11;
        p1.y += 6;
        p2.x += 1;
        p2.y += 7;
        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");

        // compute/print distance from p1 to p2
        int dx = p1.x - p2.x;
        int dy = p2.y - p2.y;
        double distp1p2 = Math.sqrt(dx * dx + dy * dy);
        System.out.println("distance from p1 to p2 = " + distp1p2);
    }
}
```

Object behavior: methods

- reading: 8.3

Client code redundancy

- Our client program translated a Point object's location:

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

- To translate several points, the code must be repeated:

```
p1.x += 11;
p1.y += 6;

p2.x += 2;
p2.y += 4;

p3.x += 1;
p3.y += 7;
...
```

Eliminating redundancy, v1

- We can eliminate the redundancy with a static method:

```
// Shifts the location of the given point.
public static void translate(Point p, int dx, int dy) {
    p.x += dx;
    p.y += dy;
}
```

- main would call the method as follows:

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

  - (Why doesn't `translate` need to return the modified point?)

Problems with static solution

- 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);    // ours   (bad)
    ```

  - If we wrote several client programs that translated `Points`, each would need a copy of the `translate` method.

- The point of classes is to combine state and behavior.
  - The behavior of `translate` is closely related to the data of the `Point`, so it belongs inside each `Point` object.

    ```
    p2.translate(2, 4);     // Java's (better)
    ```

Instance methods

- **instance method**:
  One that defines behavior for each object of a class.

- instance method declaration syntax:

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

  (same as with static methods, but without the `static` keyword)

- Instance methods allow client code to access or modify an object's state (called **accessors** and **mutators**).

Instance method example

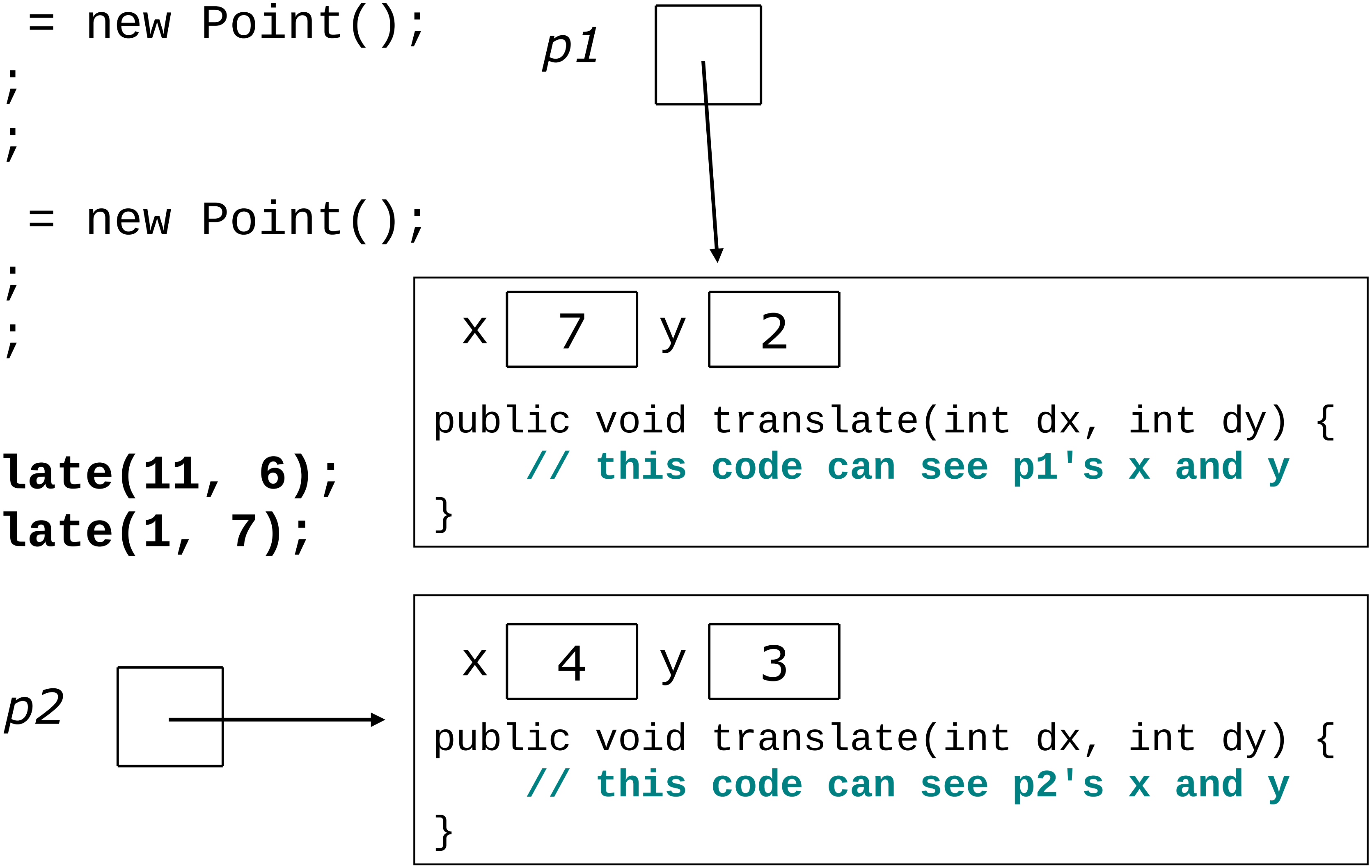
```
public class Point {
    int x;
    int y;

    // Changes the location of this Point object.
    public void translate(int dx, int dy) {
        ...
    }
}
```

- The `translate` method no longer accepts the `Point p` as a parameter. How does the method know which point to move?

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;


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



```
public void translate(int dx, int dy) {
    // this code can see p1's x and y
}
```

```
public void translate(int dx, int dy) {
    // this code can see p2's x and y
}
```

The implicit parameter

- **implicit parameter**:
  The object on which an instance method is called.

  - 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 say that it executes in the *context* of a particular object.
    - Example: The `translate` method can refer to `x` and `y`, meaning the `x` and `y` fields of the object it was called on.

Point class, version 2

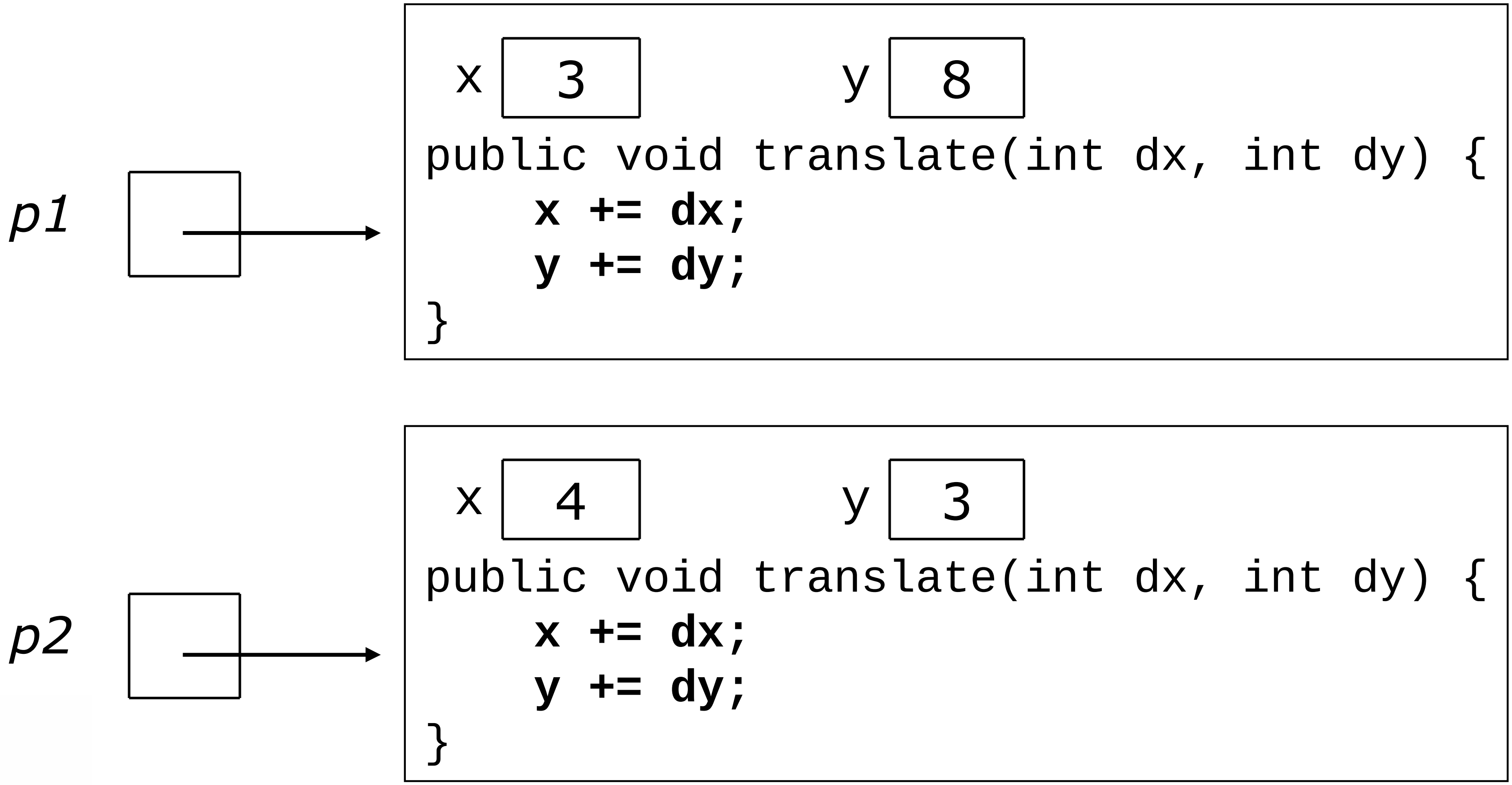
```
public class Point {
    int x;
    int y;

    // Changes the location of this Point object.
    public void translate(int dx, int dy) {
        x += dx;
        y += dy;
    }
}
```

- Now each `Point` object contains a method named `translate` that modifies its `x` and `y` fields by the given parameter values.

Tracing instance method calls

- What happens when the following calls are made?

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



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

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

Instance method questions

- Write a method `distanceFromOrigin` that returns the distance between a `Point` and the origin, (0, 0).

  Use the following formula: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

- Write a method `distance` that computes the distance between a `Point` and another `Point` parameter.

- Write a method `setLocation` that changes a `Point`'s location to the (x, y) values passed.
  - You may want to refactor your `Point` class to use this method.

- Modify the client code to use these new methods.

Client code, version 2

```
public class PointMain2 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point();
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;

        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");

        // move p2 and then print it
        p2.translate(2, 1);
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");
    }
}
```

OUTPUT:

```
p1 is (0, 2)
p2 is (6, 1)
```

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 methods.

Client code answer

```
// This client program uses the Point class.
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point();
        p1.setLocation(7, 2);
        Point p2 = new Point();
        p2.setLocation(4, 3);

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

        // compute/print each point's distance from the origin
        System.out.println("p1's distance from origin = " + p1.distanceFromOrigin());
        System.out.println("p2's distance from origin = " + p1.distanceFromOrigin());

        // move p1 and p2 and print them again
        p1.translate(11, 6);
        p2.translate(1, 7);
        System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
        System.out.println("p2 is (" + p2.x + ", " + p2.y + ")");

        // compute/print distance from p1 to p2
        System.out.println("distance from p1 to p2 = " + p1.distance(p2));
    }
}
```