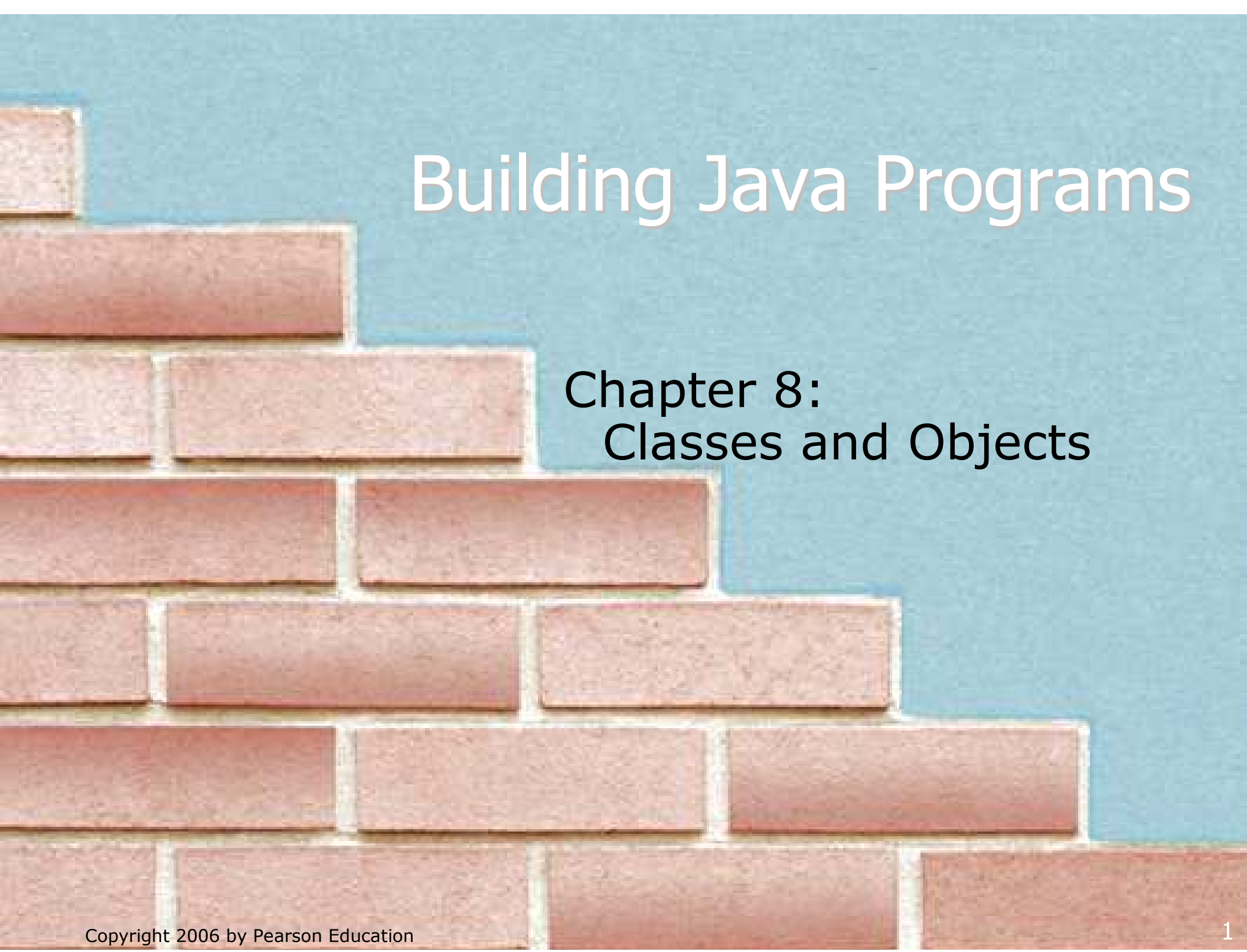


Chapter outline

- 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, classes, and types

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

- **class**: 1. A program. or,
  2. A category / type of objects.
  - classes we've used so far:

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

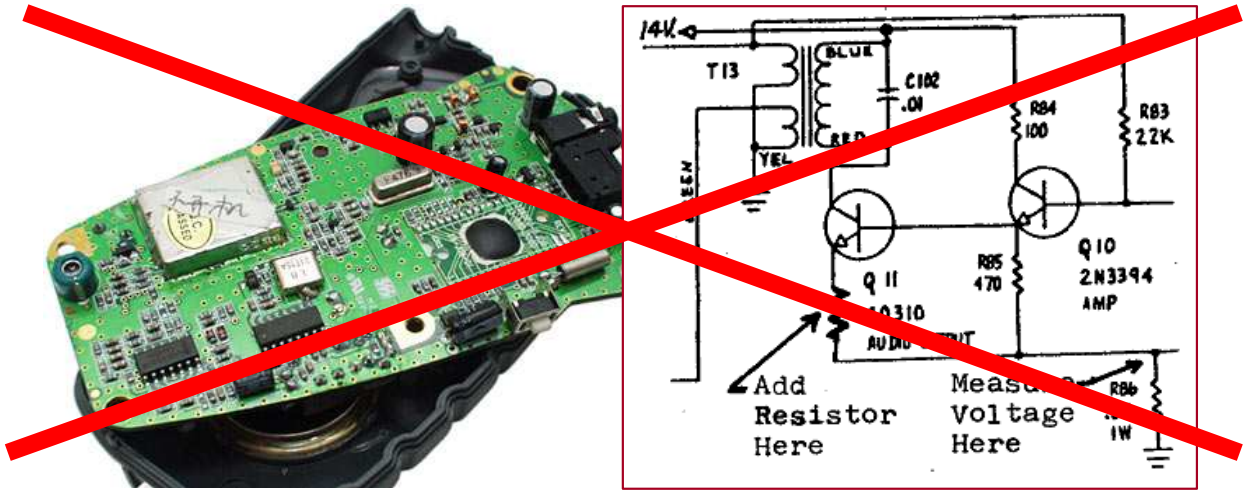
- We can write classes to define new types of objects.
  - Why would we want to do this?

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).



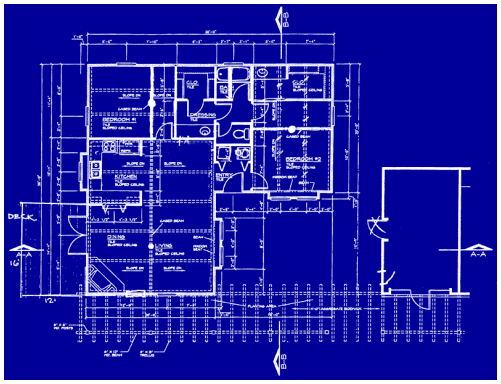
Blueprint analogy

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

Music player blueprint

state:
current song
volume
battery life

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



creates

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 = "Sandstorm"
volume = 9
battery life = 3.41 hrs

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

Music player #3

state:
song = "Code Monkey"
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:

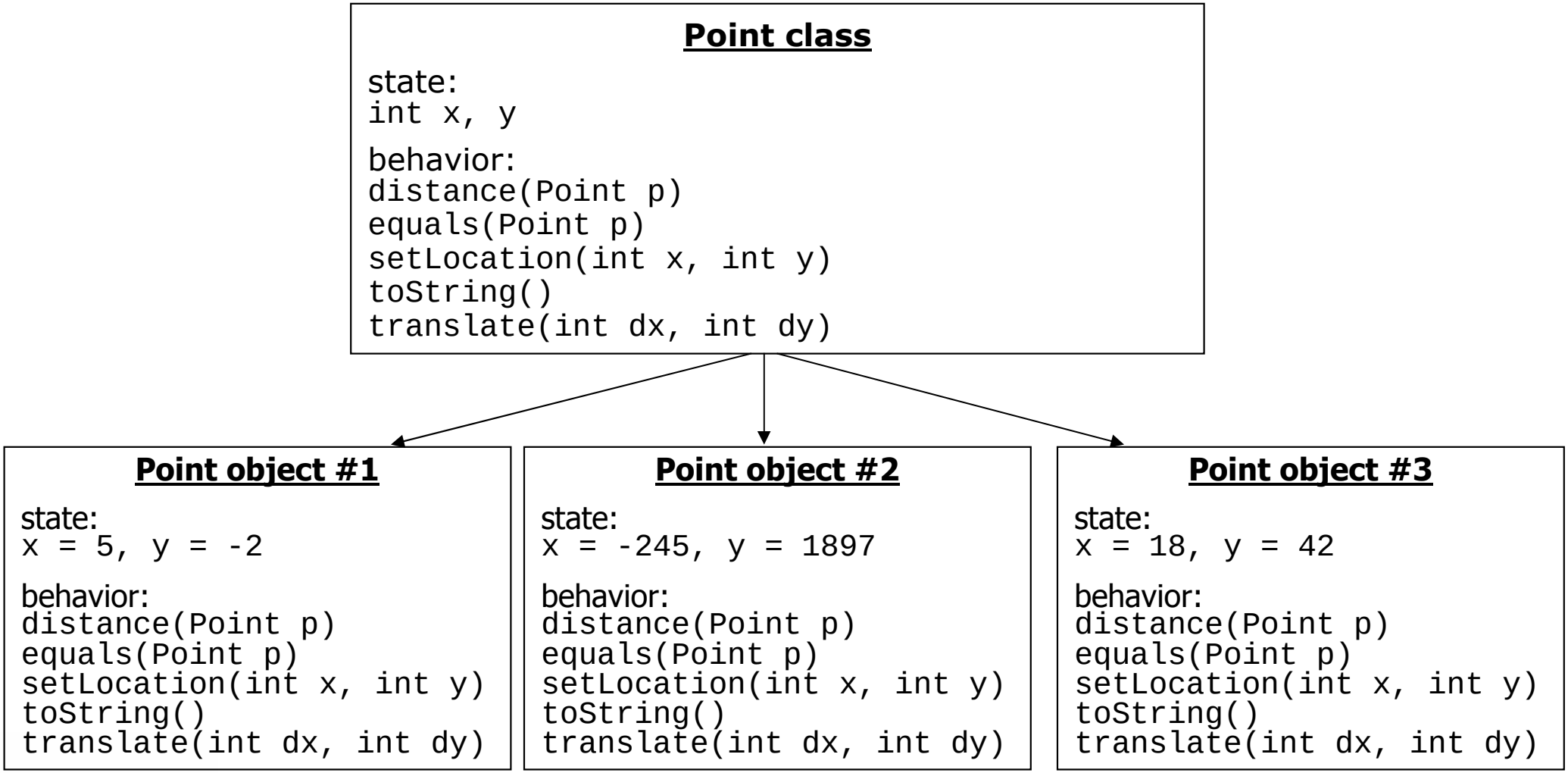
Field name	Description
x	the point's x-coordinate
y	the point's y-coordinate

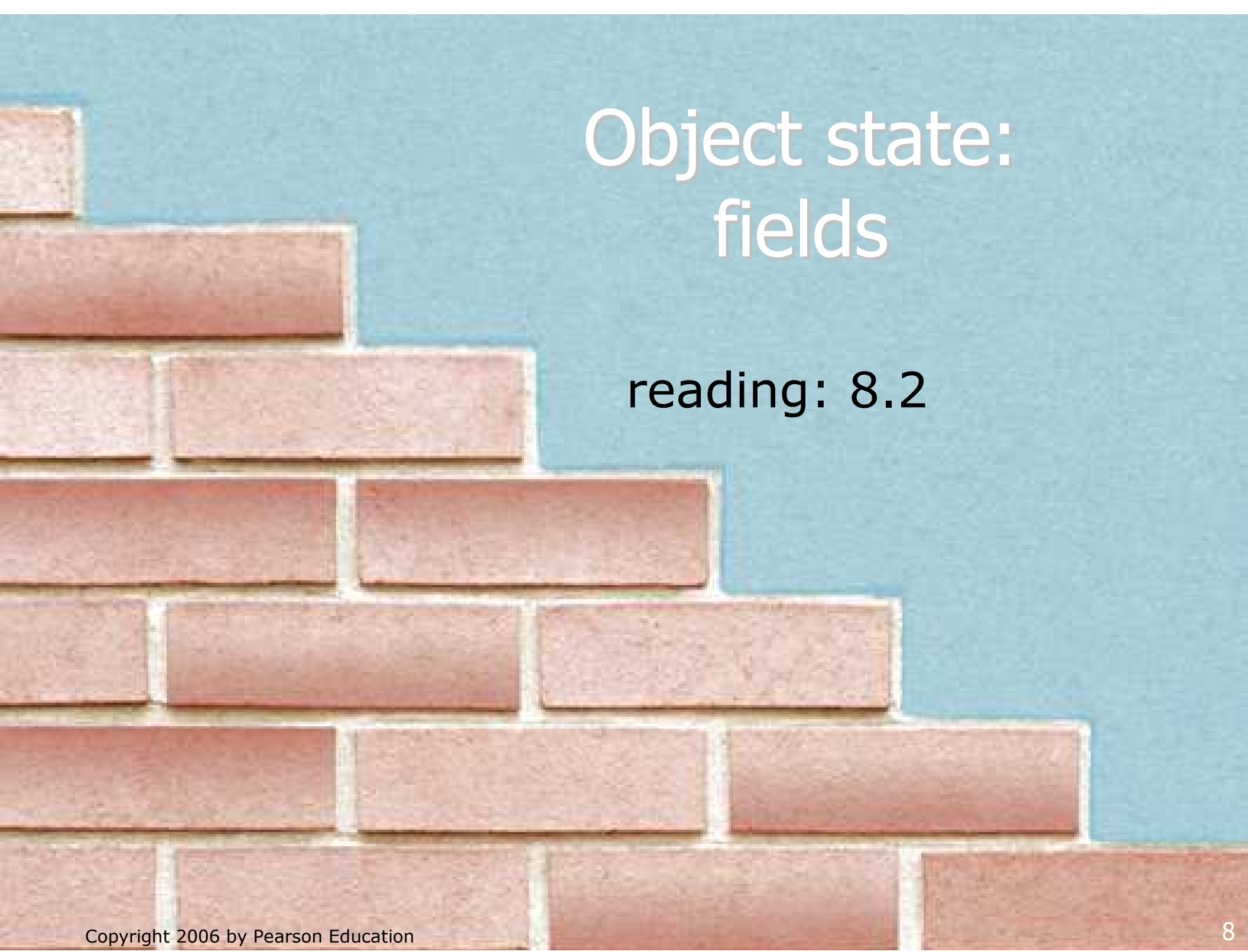
- Behavior (methods) of each `Point` object:

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.





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, general syntax:

  <type> <name> ;

  - Examples:

```
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:
    <variable name> . ***<field name>***
  - Modifying a field, general syntax:
    <variable name> . ***<field name>*** = ***<value>*** ;

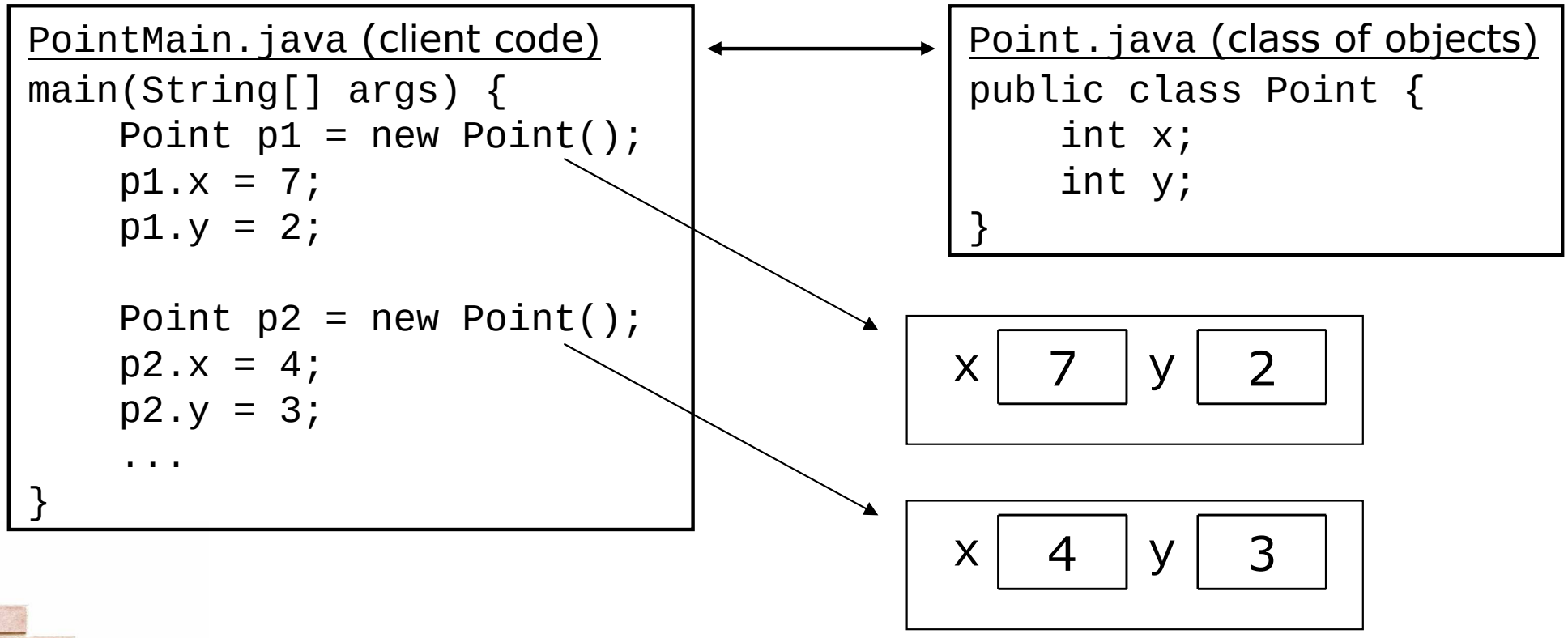
- Examples:

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

  - Later we'll learn about *encapsulation*, which will change the way we access the data inside objects.

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.



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)
```

Client code question

- Write a client program that uses our `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 (5, 10)
distance from p1 to p2 = 13.0
```

  - Recall: 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 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.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, general syntax:

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

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

Instance method example

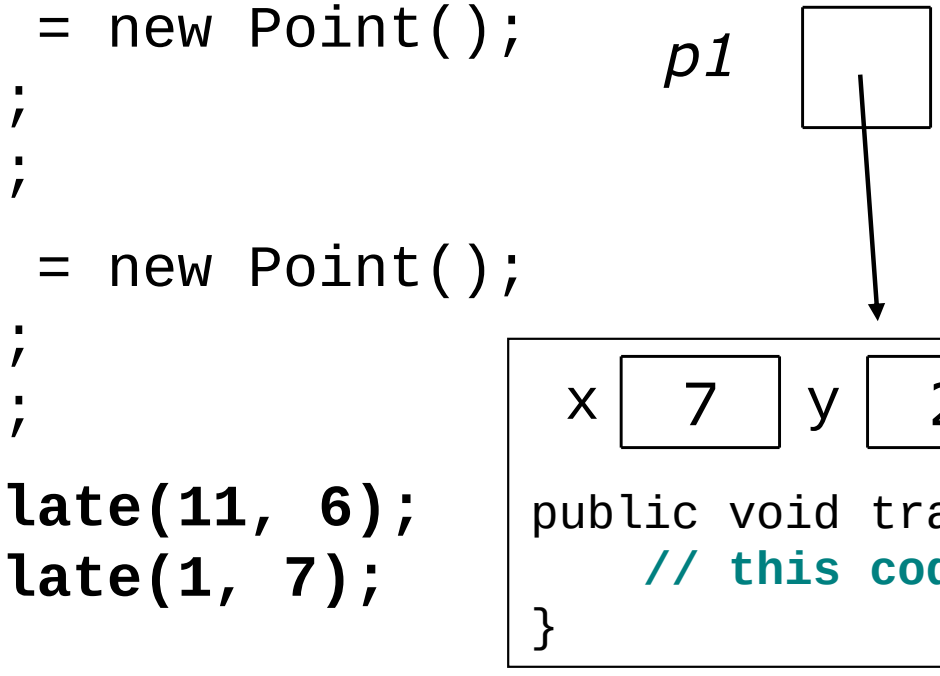
```
public class Point {
    int x;
    int y;

    // 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);
```



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

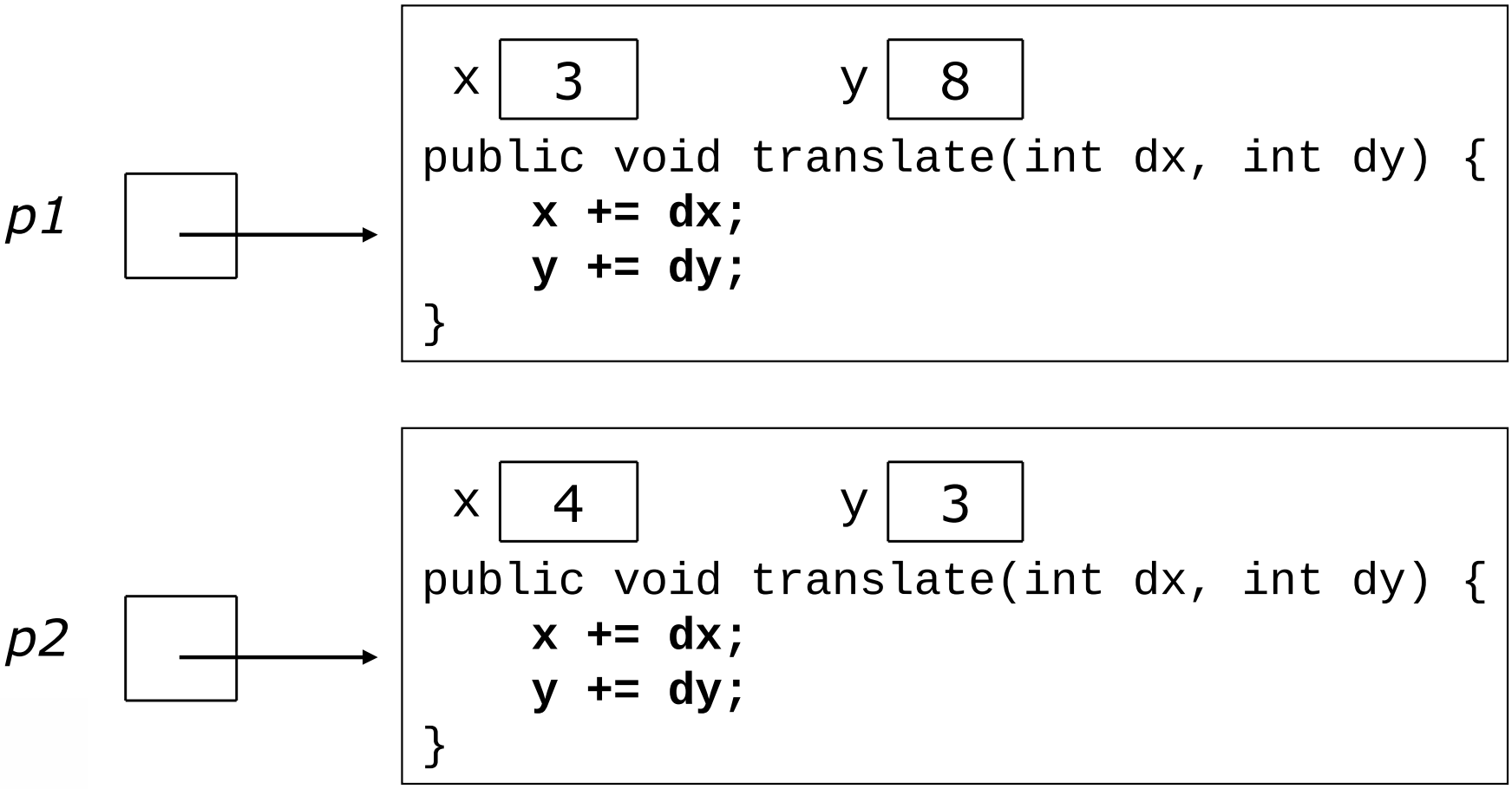
```
public class Point {
    int x;
    int y;

    // 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);
```



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.

Accessors and mutators

Two common categories of instance methods:

- **accessor**: Provides information about an object.
  - The information comes from (or is computed using) the fields.
  - Examples: `distanceFromOrigin`, `distance`

- **mutator**: Modifies an object's state.
  - Sometimes the change is based on parameters (e.g. `dx`, `dy`).
  - Examples: `translate`, `setLocation`

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

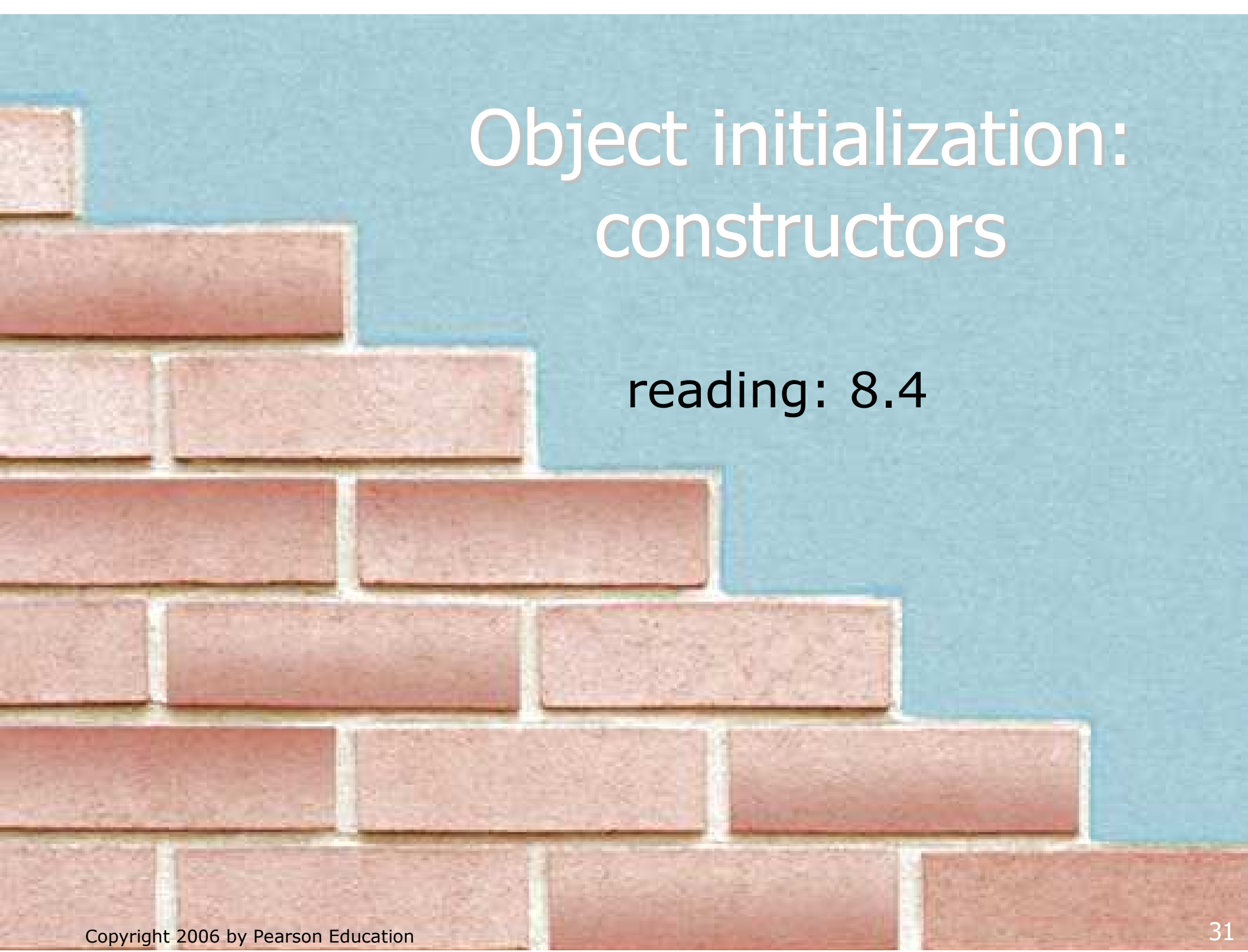
- Recall our client program that produces this output:

```
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));
    }
}
```



Initializing objects

- It is tedious to construct an object and assign values to all of its data fields one by one.

```
Point p = new Point();
p.x = 3;
p.y = 8;                        // tedious
```

- We'd rather pass the fields' initial values as parameters:

```
Point p = new Point(3, 8);  // better!
```

  - We were able to this with Java's built-in `Point` class.

Constructors

- **constructor**: Initializes the state of new objects.
  - Constructor syntax:

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

  - A constructor runs when the client uses the `new` keyword.
  - A constructor does not specify a return type;
    it implicitly returns the new object being created.
  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all the object's fields to 0.

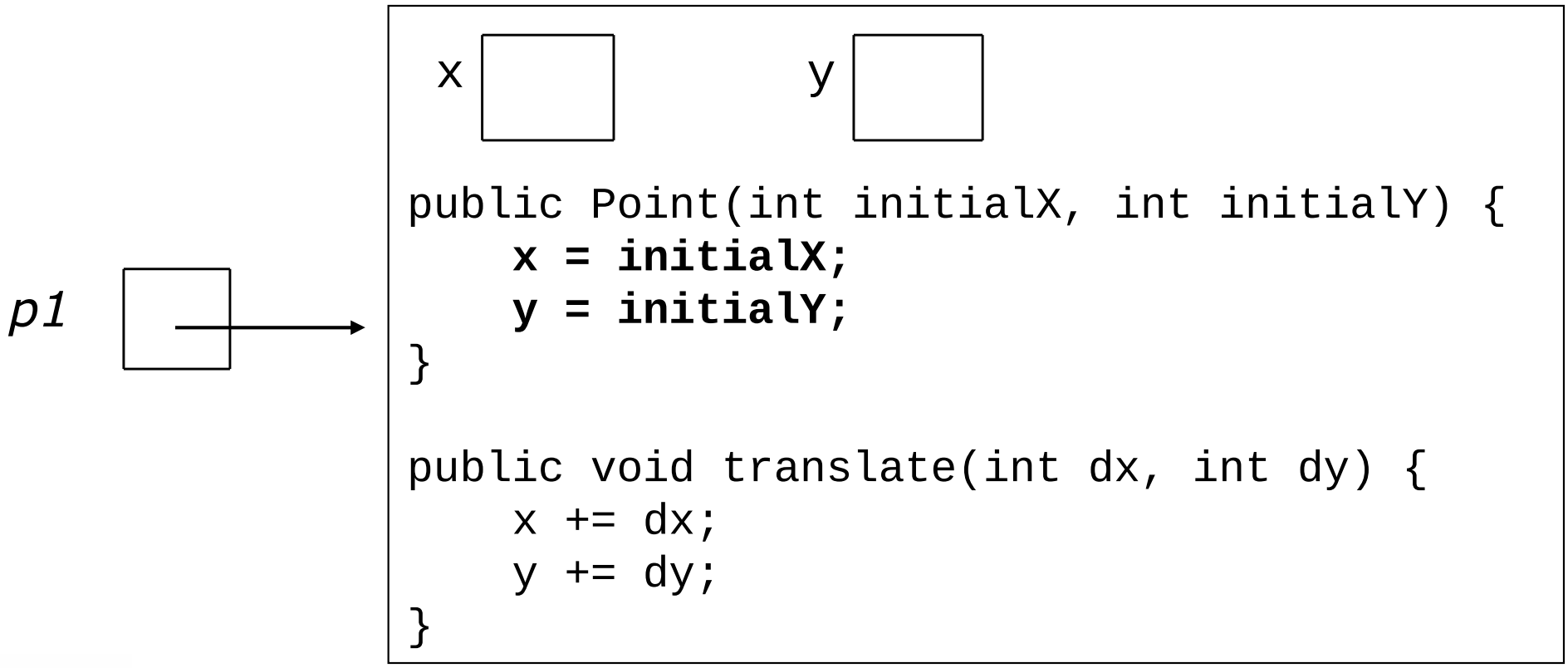
Point class, version 3

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

    // Constructs a Point at the given x/y coordinates.
    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);
```



Client code, version 3

```
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)
distance from p1 to p2 = 13.0
```

- Modify the program to use our new constructor.

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(7, 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 + ")");

        // 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));
    }
}
```

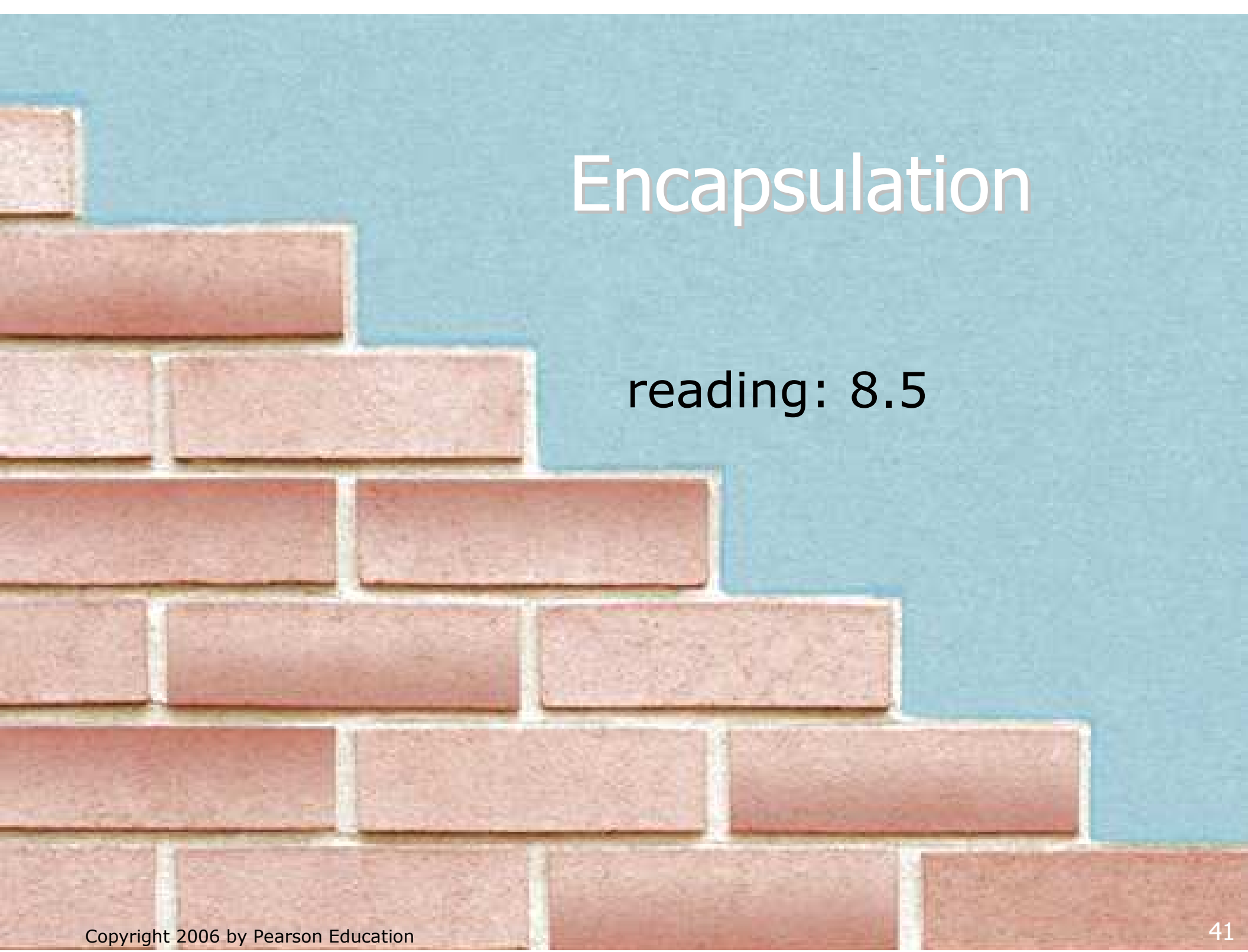
State/behavior question

- Write a class named `Parent` that represents a parent driving children to an exciting place (e.g. DisneyLand).
  - The children ask the parent, "Are we there yet?"
  - The parent becomes increasingly annoyed.

- The `Parent` class has a method named `areWeThereYet` that returns a `String` for the parent's response.
  - The first 2 times it is called, return `"Just a little farther."`
  - The next 2 times it is called, return `"NO."`
  - The next time it is called, return `"STOP ASKING ME THAT!"`
  - For all subsequent calls, return `"You're grounded."`

State/behavior answer

```
public class Parent {
    private int calls;   // counts areWeThereYet calls

    public Parent(String theName) {
        calls = 0;
    }

    public String areWeThereYet() {
        calls++;
        if (calls == 1 || calls == 2) {
            return "Just a little farther.";
        } else if (calls == 3 || calls == 4) {
            return "NO.";
        } else if (calls == 5) {
            return "STOP ASKING ME THAT!";
        } else {
            return "You're grounded.";
        }
    }
}
```

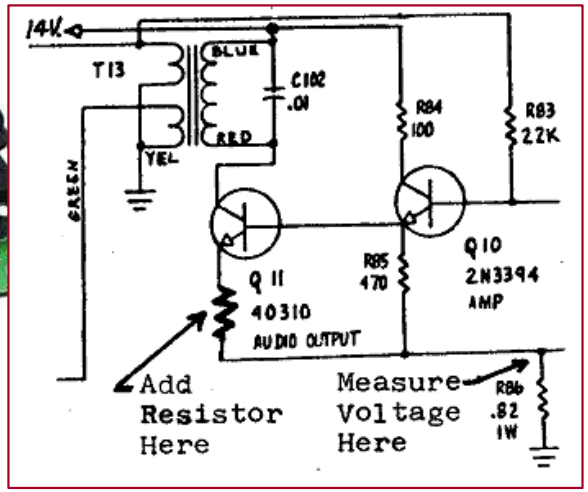


Encapsulation

- **encapsulation**:
  Hiding implementation details of an object from clients.

- Encapsulation provides *abstraction*;
  we can use objects 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 access or change them.
  - Declaring a private field, general syntax:

    ```
    private <type> <name> ;
    ```

  - Examples:

    ```
    private int x;
    private String name;
    ```

- Once fields are private, client code cannot access them:

```
PointMain.java:11: x has private access in Point
System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
                                  ^
```

Accessors and mutators

- We provide accessor methods to examine their values:

```
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:

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

  - Client code will look more like this:

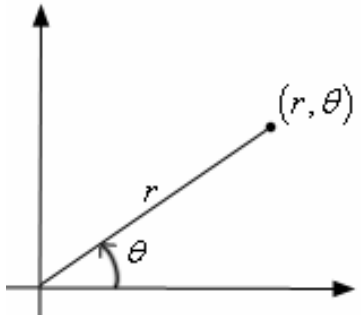
```
System.out.println("p1 is (" + p1.getX() + ", " + p1.getY() + ")");
p1.setX(14);
```

Benefits of encapsulation

- Provides abstraction between an object and its clients.

- Protects an object from unwanted access by clients.
  - Example: If 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.

- Allows you to change the class implementation later.
  - Example: The `Point` class could be rewritten to use polar coordinates (a radius *r* and an angle θ from the origin), but the external behavior and methods could remain the same.



Point class, version 4

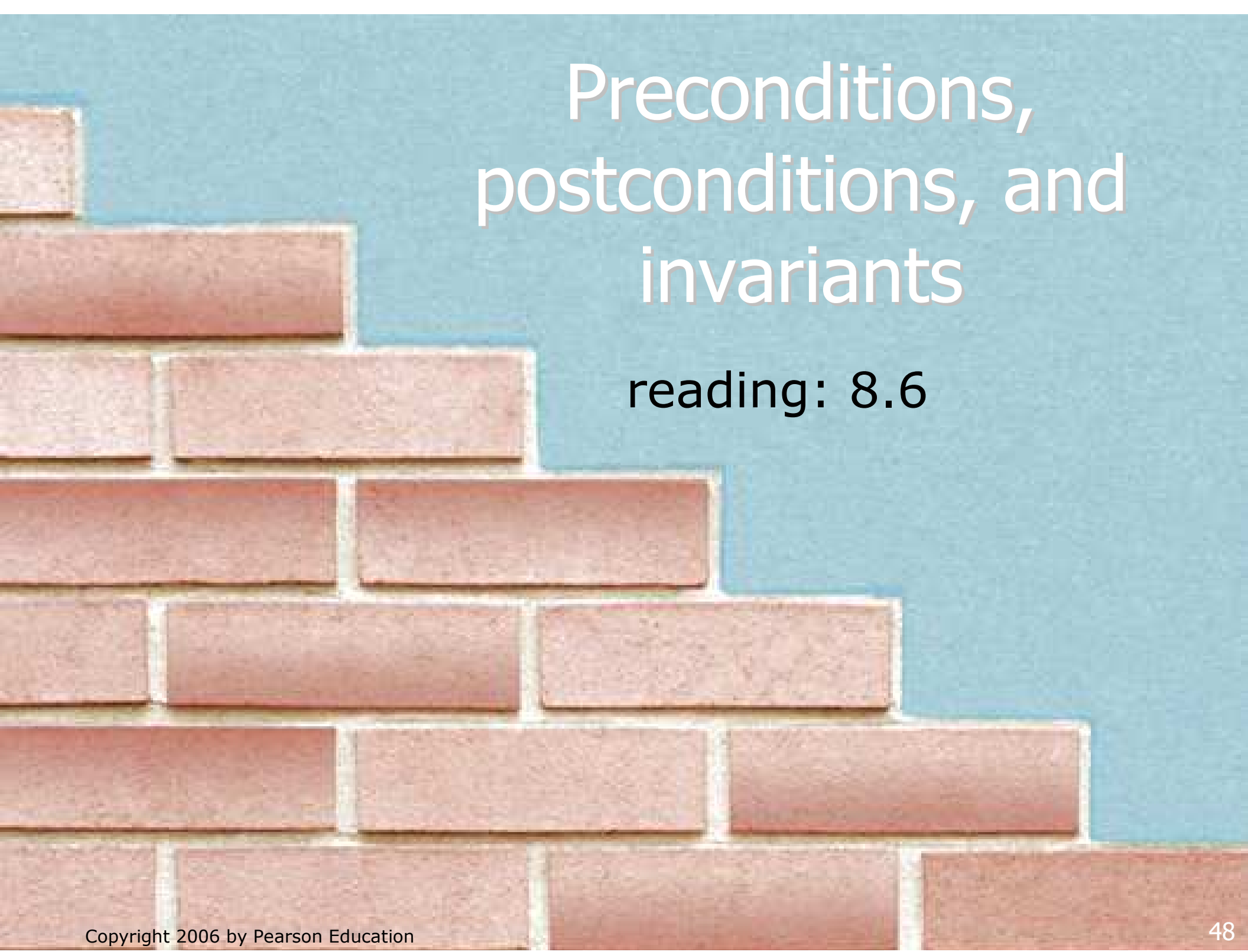
```
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    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;
    }
}
```

Client code, version 4

```
public class PointMain4 {
    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.getX() + ", " + p1.getY() + ")");
        System.out.println("p2 is (" + p2.getX() + ", " + p2.getY() + ")");

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

OUTPUT:

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



Preconditions, postconditions, and invariants

reading: 8.6

Pre/postconditions

- **precondition**:
  Something assumed to be true when a method is called.
- **postcondition**:
  Something promised to be true when a 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.

  Examples:
  - "No BankAccount object's balance can be negative."
  - "The speed of a SpaceShip object must be ≤ 10."

- Let's add an invariant to the `Point` class:
  - "No `Point` object's x and y coordinates can be negative."

  To enforce this invariant, we must prevent clients from:
  - constructing a `Point` object with a negative x or y value
  - moving a `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:

```
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

Ways to deal with violated preconditions:

- Return out of the method if negative values are found.

  Drawbacks:

  - It is not possible to do this in the constructor.
  - The client doesn't expect this behavior.
  - Fails "silently"; client doesn't realize something has gone wrong.

- Have the object *throw an exception*. (better)
  - This will cause the client program to halt.

Throwing exceptions

- Throwing an exception, general syntax:

```
    throw new <exception type> ();
or  throw new <exception type> ("<message>");
```

  - ***<message>*** will be shown on console when program crashes.

- Example:

```
// 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

- Ensure that no `Point` is constructed with negative `x` or `y`:

```
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`:

```
public void translate(int dx, int dy) {
    if (x + dx < 0 || y + dy < 0) {
        throw new IllegalArgumentException();
    }
    x += dx;
    y += dy;
}
```

Printing objects

- By default, Java doesn't know how to print objects:

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

- We can print a better string (but this is cumbersome):

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

- We'd like to be able to print the object itself:

```
// 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.
  - Is called when an object is printed or concatenated to a `String`.

```
Point p1 = new Point(7, 2);
System.out.println("p1 is " + p1);
```

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

```
System.out.println("p1 is " + p1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - The default `toString` returns the class's name followed by a hexadecimal (base-16) number:

```
"Point@9e8c34"
```

toString method syntax

- You can replace the default behavior by defining a `toString` method in your class.

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

  - Example:

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

Client code question

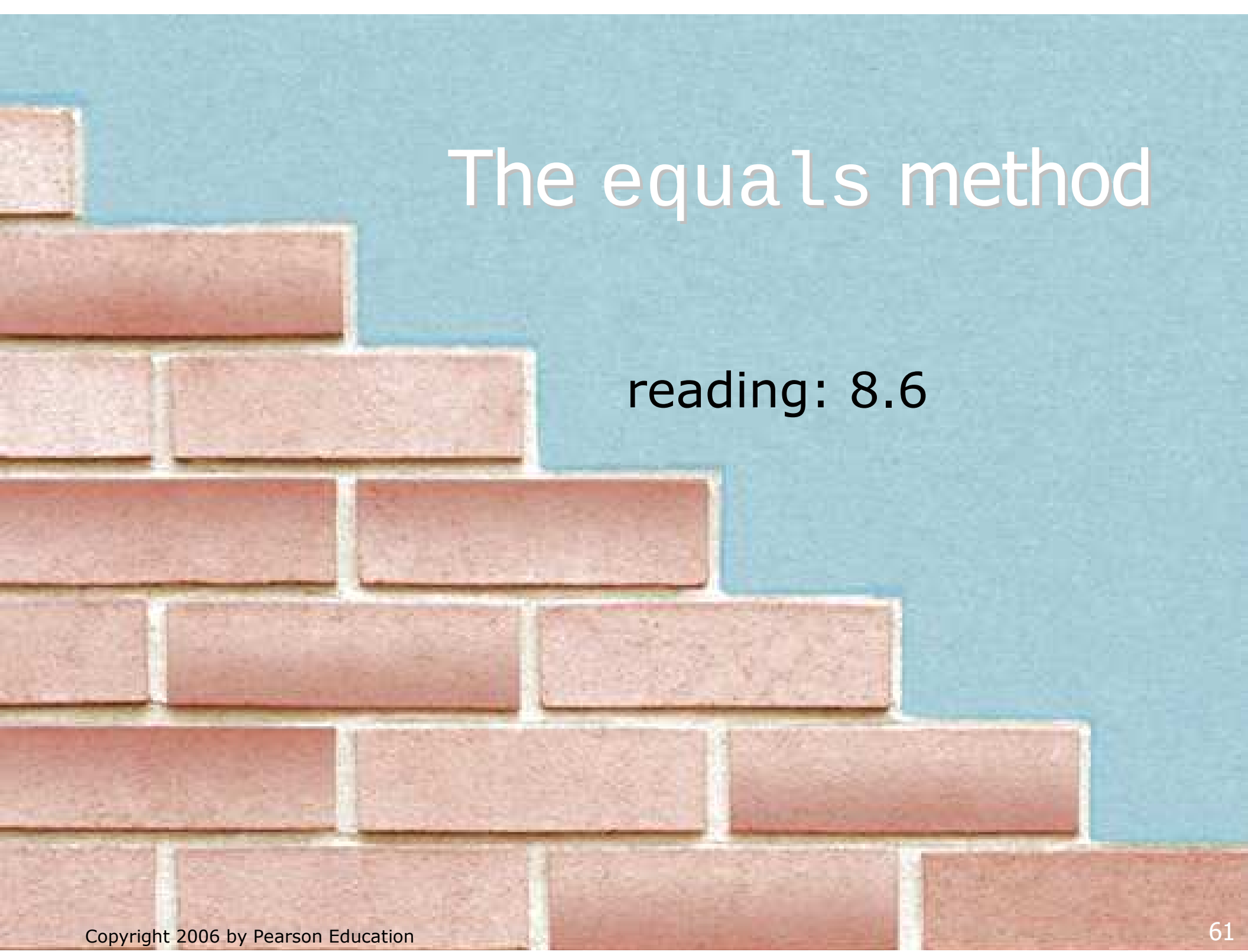
- Recall our client program that produces this output:

```
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)
distance from p1 to p2 = 13.0
```

- Modify the program to use our new `toString` method.

Client code answer

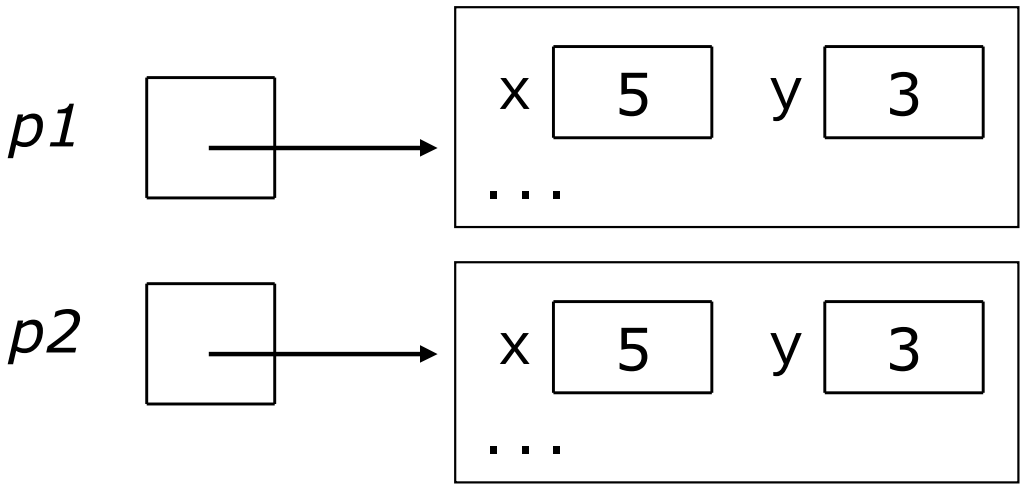
```
// This client program uses the Point class.
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(7, 2);
        Point p2 = new Point(4, 3);

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

        // 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);
        System.out.println("p2 is " + p2);

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



Recall: comparing objects

- The == operator does not work well with objects.
  - == compares references to objects, not their state.
  - Example:

```
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.
  - The default `equals` behavior acts just like the == operator.

```
if (p1.equals(p2)) {   // false
    System.out.println("equal");
}
```

- We can change this behavior by writing an `equals` method.
  - The method should compare the state of the two objects and return `true` for cases like the above.

Initial flawed equals method

- A flawed implementation of the `equals` method:

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

Flaws in equals method

- The body can be shortened to the following:

```
// boolean zen
return x == other.x && y == other.y;
```

- It should be legal to compare a `Point` to any object (not just other `Point` objects):

```
// this should be allowed
Point p = new Point(7, 2);
if (p.equals("hello")) {   // false
    ...
```

  - `equals` should always return false if a non-`Point` is passed.

equals and the Object class

- `equals` method, general syntax:

```
public boolean equals(Object <name>) {
    <statement(s) that return a boolean value> ;
}
```

  - The parameter to `equals` must be of type `Object`.
  - `Object` is a general type that can match any object.
  - Having an `Object` parameter means *any* object can be passed. (We'll learn more about the `Object` class in Chapter 9.)

Another flawed version

- Another flawed `equals` implementation:

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

- It does not compile:

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

  - The compiler is saying,
    "o could be any object. Not every object has an x field."

Type-casting objects

- Solution: *Type-cast* the object parameter to a `Point`.

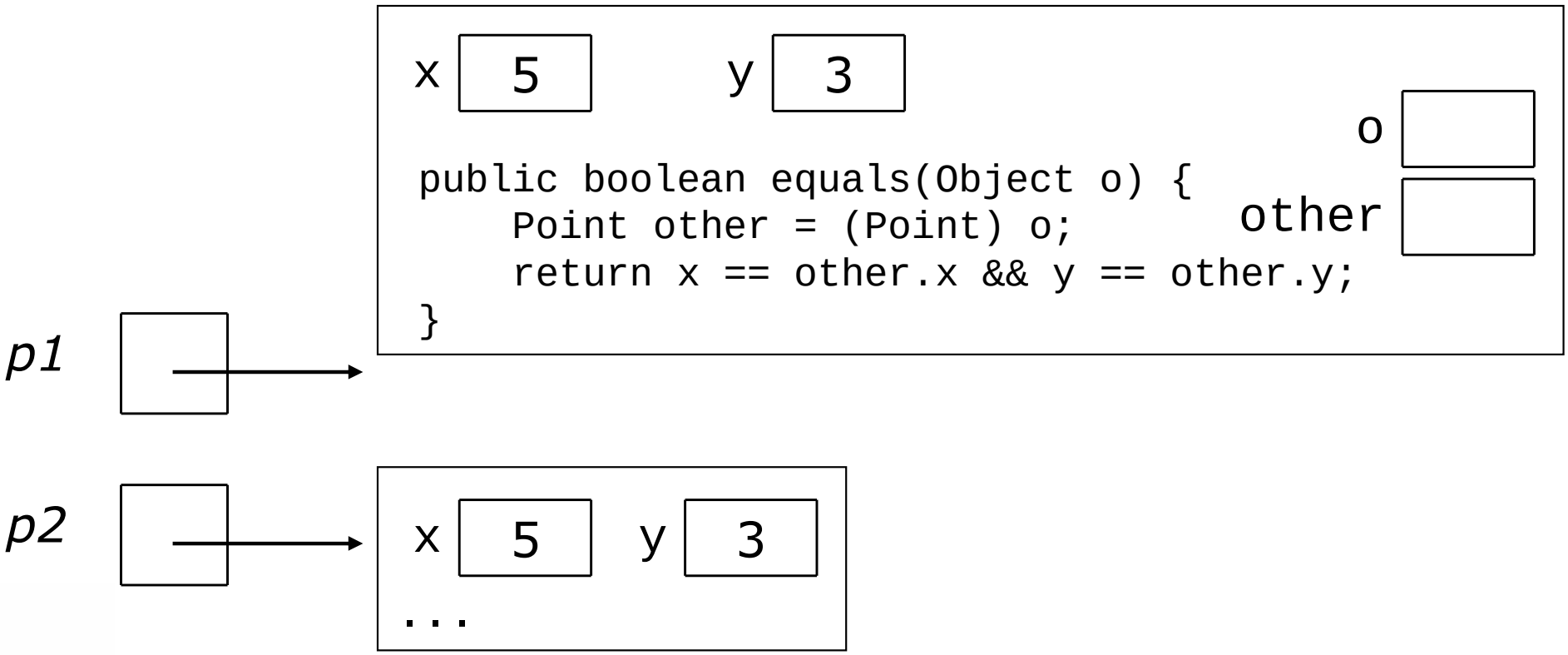
```
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

- Casting objects is different than casting primitives.
  - We're really casting an `Object` reference into a `Point` reference.
  - We're promising the compiler that `o` refers to a `Point` object.

Casting objects diagram

- Client code:

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



Comparing different types

- When we compare `Point` objects to other types:

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

  - Currently the code crashes:

```
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 line with the type-cast:

```
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.

  - Examples:

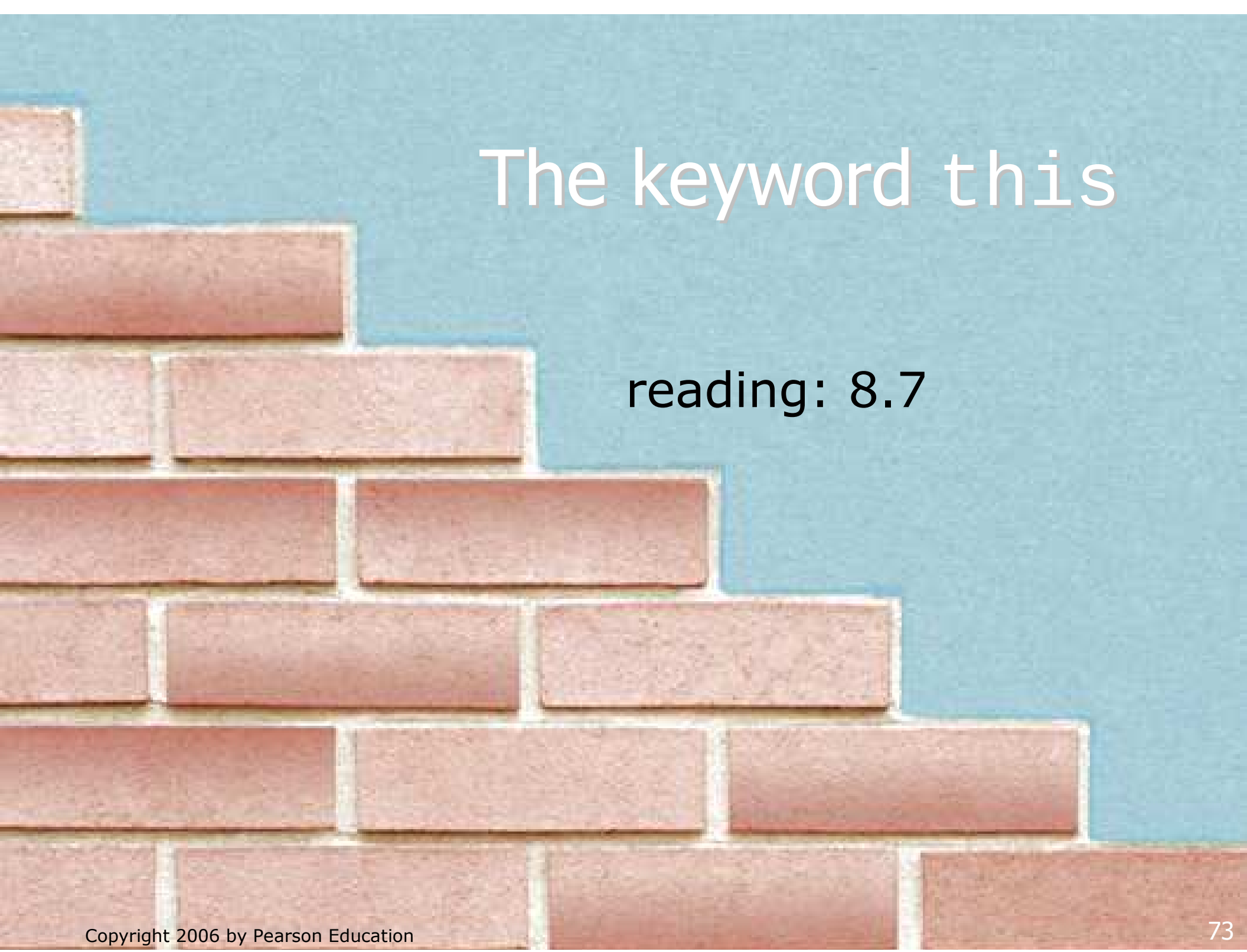
```
String s = "hello";
Point p = new Point();
```

expression	result
`s instanceof Point`	`false`
`s instanceof String`	`true`
`p instanceof Point`	`true`
`p instanceof String`	`false`
`null instanceof String`	`false`

Final version of equals method

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point object.
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and compare it
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        // o is not a Point; cannot be equal
        return false;
    }
}
```

- This version correctly compares `Points` to any type of object.



Using the keyword `this`

- **`this`** : A reference to the implicit parameter.
  - *implicit parameter:* object on which a method/constructor is called

- `this` keyword, general syntax:
  - To refer to a field:
    `this.`***`<field name>`***
  - To call a method:
    `this.`***`<method name>`***`(`***`<parameters>`***`);`
  - To call a constructor from another constructor:
    `this(`***`<parameters>`***`);`

Variable names and scope

- Usually it is illegal to have two variables in the same scope with the same name.

- Recall: `Point` class's `setLocation` method:
  - Params named `newX` and `newY` to be distinct from fields `x` and `y`

```
public class Point {
    int x;
    int y;
    ...
    public void setLocation(int newX, int newY) {
        if (newX < 0 || newY < 0) {
            throw new IllegalArgumentException();
        }
        x = newX;
        y = newY;
    }
}
```

Variable shadowing

- However, a class's method can have a parameter whose name is the same as one of the class's fields.
  - Example:

```
// this is legal
public void setLocation(int x, int y) {
    ...
}
```

  - Fields `x` and `y` are *shadowed* by parameters with same names.
  - Any `setLocation` code that refers to `x` or `y` will use the parameter, not the field.

- **shadowed variable**: A field that is "covered up" by a parameter or local variable with the same name.

Avoiding shadowing with this

- The keyword this prevents shadowing:

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

    ...

    public void setLocation(int x, int y) {
        if (x < 0 || y < 0) {
            throw new IllegalArgumentException();
        }
        this.x = x;
        this.y = y;
    }
}
```

Inside the setLocation method:

- When this.x is seen, the *field* x is used.
- When x is seen, the *parameter* x is used.

Multiple constructors

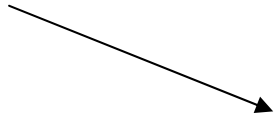
- It is legal to have more than one constructor in a class.
  - The constructors must accept different parameters.

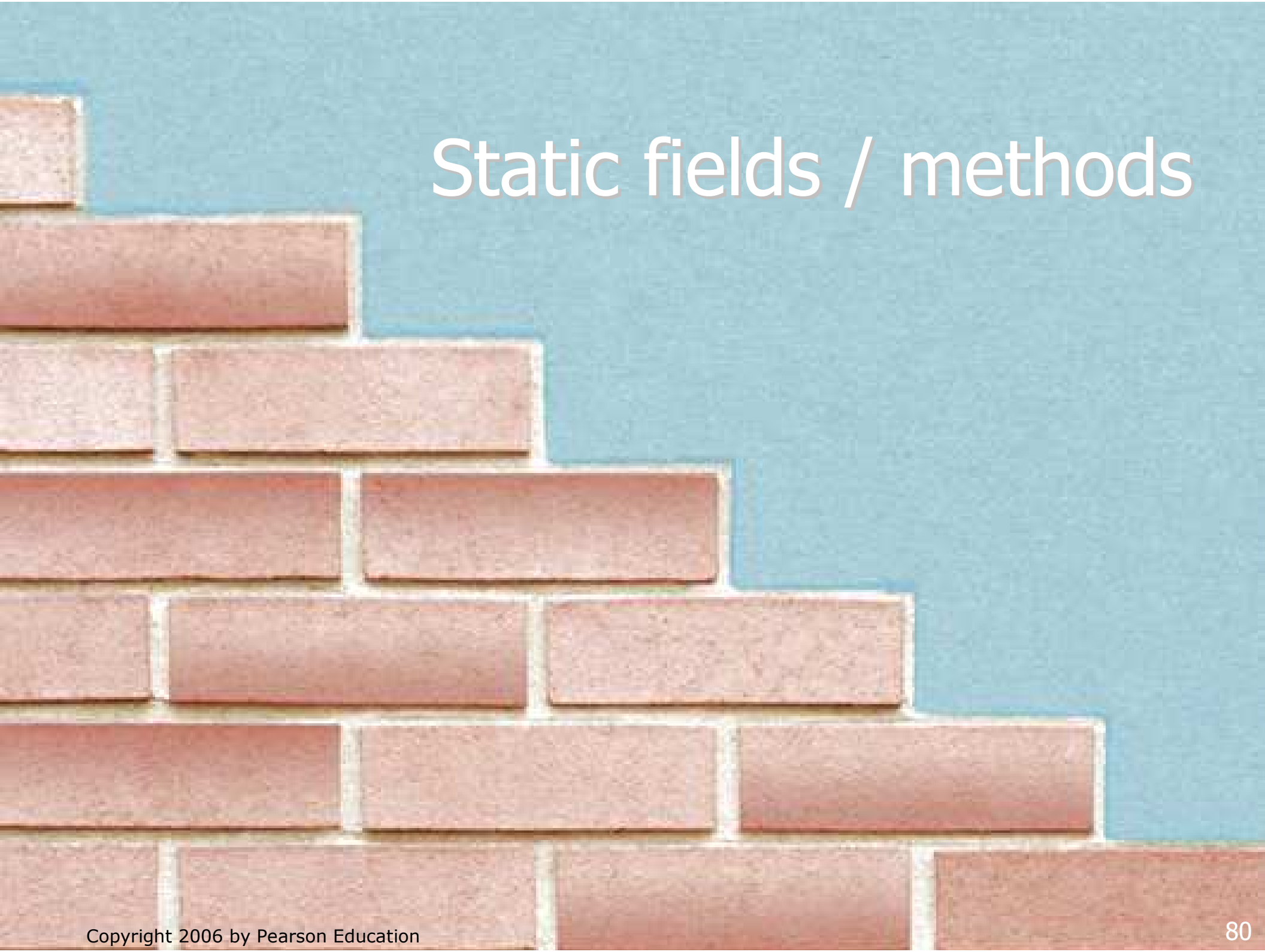
```
public class Point {
    private int x;
    private int y;

    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 rename the parameters and use `this.` field syntax.

```
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;
    }

    ...
}
```





Static fields vs. fields

- **static**: Part of a class, rather than part of an object.
  - Classes can have static fields.
  - Unlike fields, static fields are not replicated into each object; instead a single field is shared by all objects of that class.

- static field, general syntax:

```
private static <type> <name>;
```

or,

```
private static <type> <name> = <value>;
```

  - Example:

```
private static int count = 0;
```

Static field example

- Count the number of Husky objects created:

```
public class Husky implements Critter {
    // count of Huskies created so far
    private static int objectCount = 0;

    private int number;    // each Husky has a number

    public Husky() {
        objectCount++;
        number = objectCount;
    }

    ...

    public String toString() {
        return "I am Husky #" + number +
               "out of " + objectCount;
    }
}
```

Static methods

- **static method**: One that's part of a class, not part of an object.
  - good places to put code related to a class, but not directly related to each object's state
  - shared by all objects of that class
  - does not understand the *implicit parameter*; therefore, cannot access fields directly
  - if `public`, can be called from inside or outside the class

- Declaration syntax: *(same as we have seen before)*

```
public static <return type> <name>(<params>) {
    <statements>;
}
```

Static method example 1

- Java's built-in `Math` class has code that looks like this:

```
public class Math {
    ...
    public static int abs(int a) {
        if (a >= 0) {
            return a;
        } else {
            return -a;
        }
    }

    public static int max(int a, int b) {
        if (a >= b) {
            return a;
        } else {
            return b;
        }
    }
}
```

Static method example 2

- Adding a static method to our `Point` class:

```
public class Point {
    ...
    // Converts a String such as "(5, -2)" to a Point.
    // Pre: s must be in valid format.

    public static Point parse(String s) {
        s = s.substring(1, s.length() - 1);  // "5, -2"
        s = s.replaceAll(",", "");           // "5 -2"

        // break apart the tokens, convert to ints
        Scanner scan = new Scanner(s);
        int x = scan.nextInt();              // 5
        int y = scan.nextInt();              // 2

        Point p = new Point(x, y);
        return p;
    }
}
```

Calling static methods, outside

- Static method call syntax (*outside* the class):

  ***<class name>*.*<method name>*(*<values>*);**

  - This is the syntax client code uses to call a static method.

  - Examples:

```
int absVal = Math.max(5, 7);

Point p3 = Point.parse("(-17, 52)");
```

Calling static methods, inside

- Static method call syntax (*inside* the class):

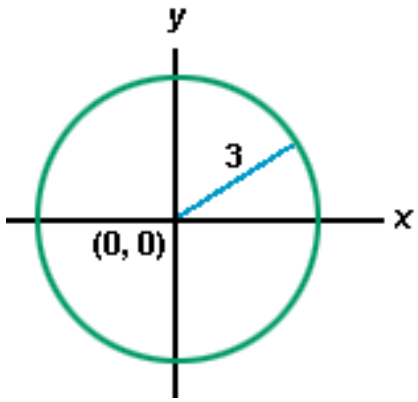
  <method name>(***<values>***);

  - This is the syntax the class uses to call its own static method.
  - Example:

```
public class Math {
    // other methods such as ceil, floor, abs, etc.
    // ...
    public static int round(double d) {
        if (d - (int) d >= 0.5) {
            return ceil(d);
        } else {
            return floor(d);
        }
    }
}
```

Object practice problem

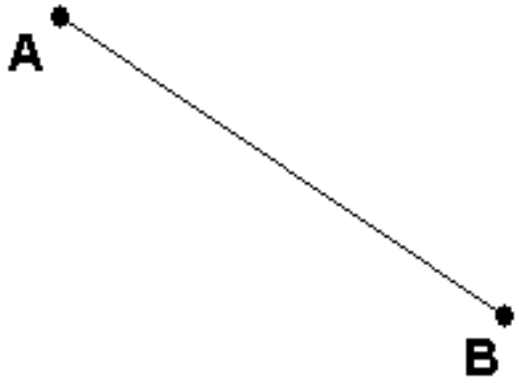
- Create a class named `Circle`.
  - A circle is represented by a point for its center, and its radius.
  - Make it possible to construct the unit circle, centered at (0, 0) with radius 1, by passing no parameters to the constructor.



  - Circles should be able to tell whether a given point is contained inside them.
  - Circles should be able to draw themselves using a 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

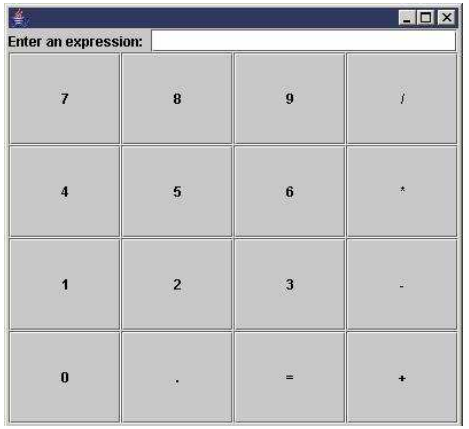
- Create a class named `LineSegment`.
  - A line segment is represented by two endpoints (x_1, y_1) and (x_2, y_2).



  - 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:

```
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(8);

        int result = calc.compute();

        System.out.println(calc);
        System.out.println("result = " + result);
    }
}
```