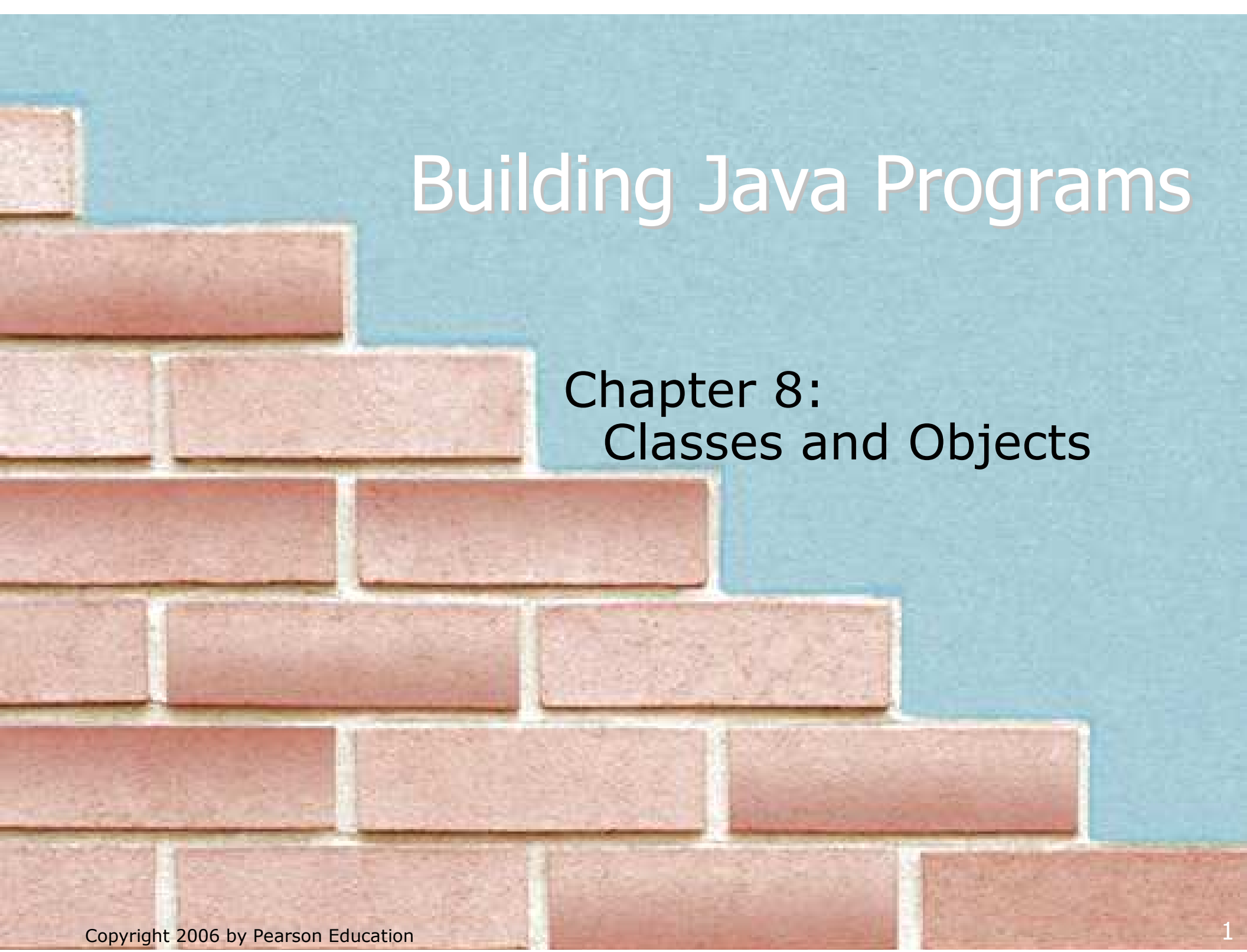
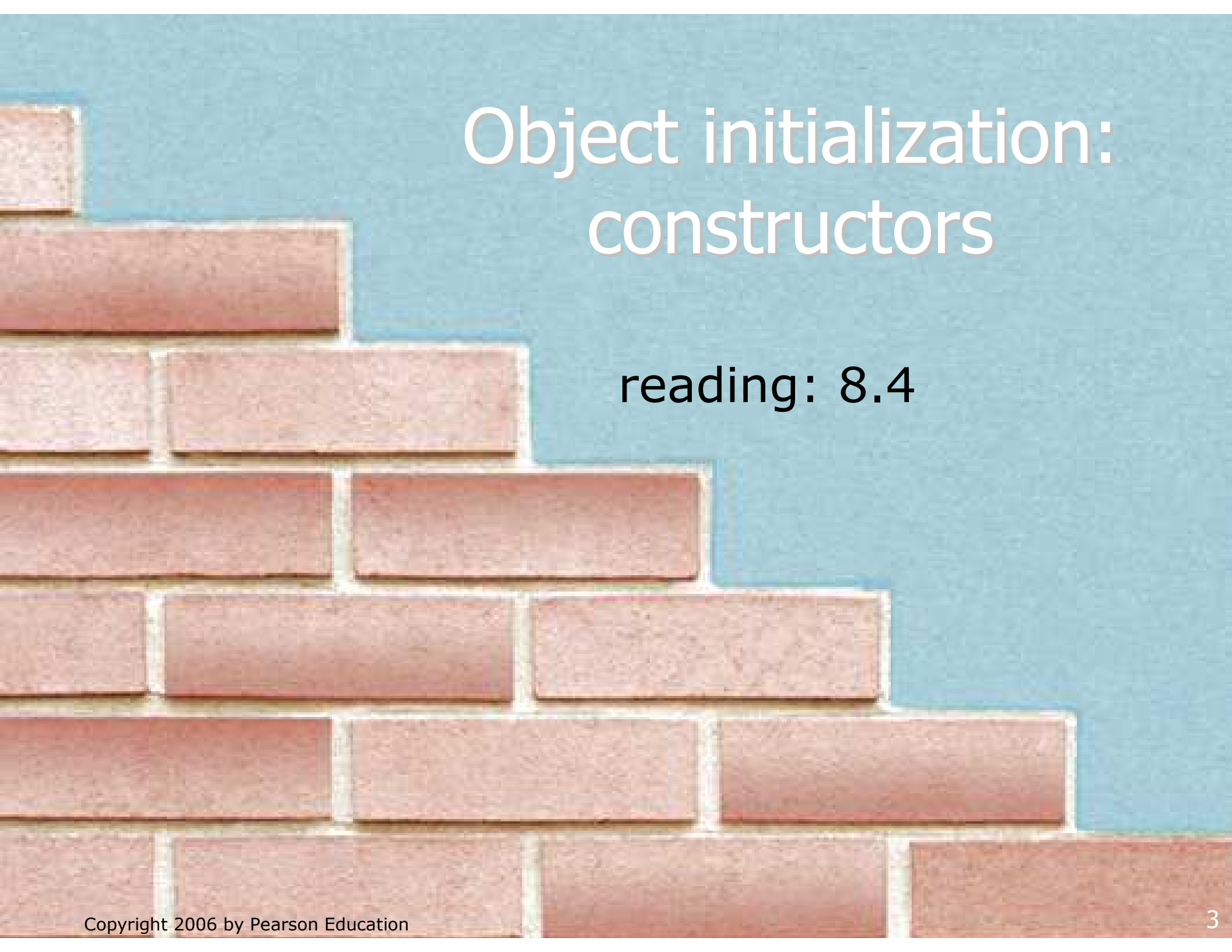


Lecture outline

- anatomy of a class, continued
  - constructors
  - encapsulation
- preconditions, postconditions, and invariants



Object initialization: constructors

reading: 8.4

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.

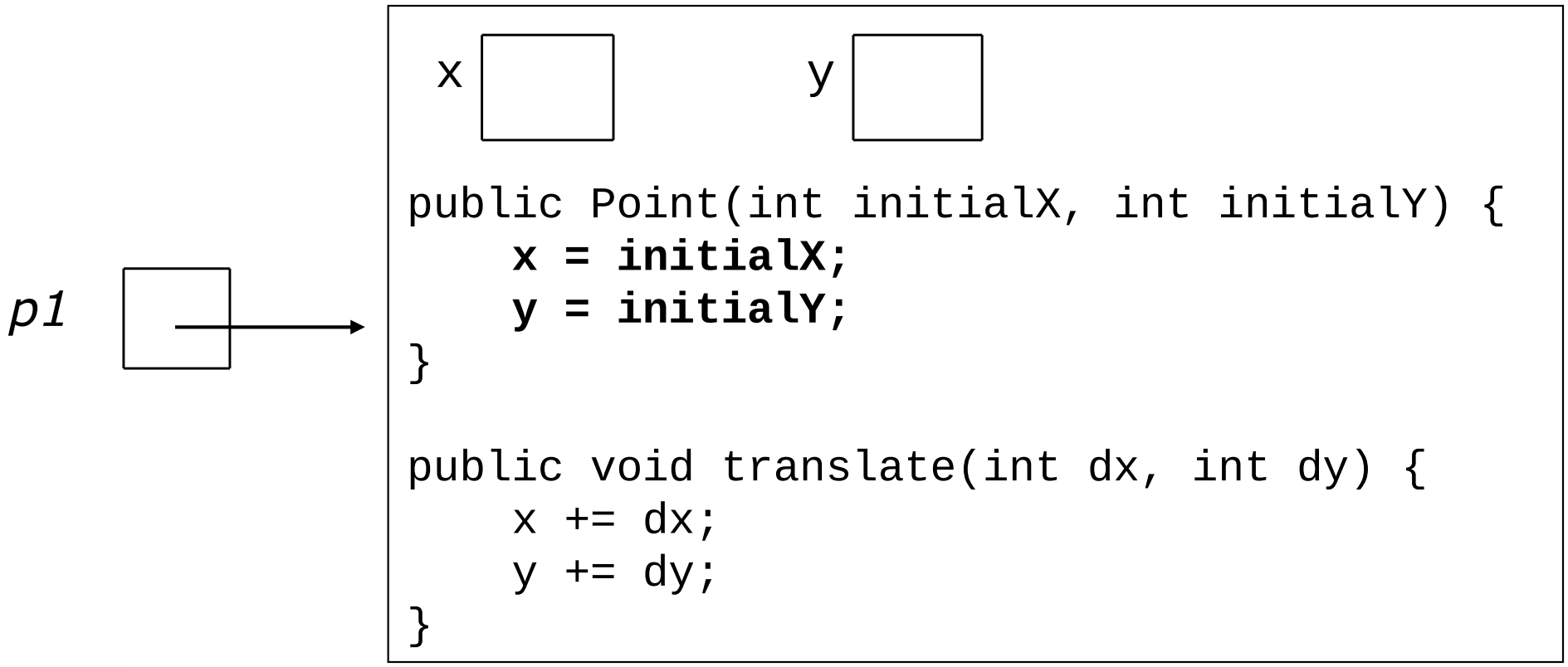
Point class, version 3

```
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

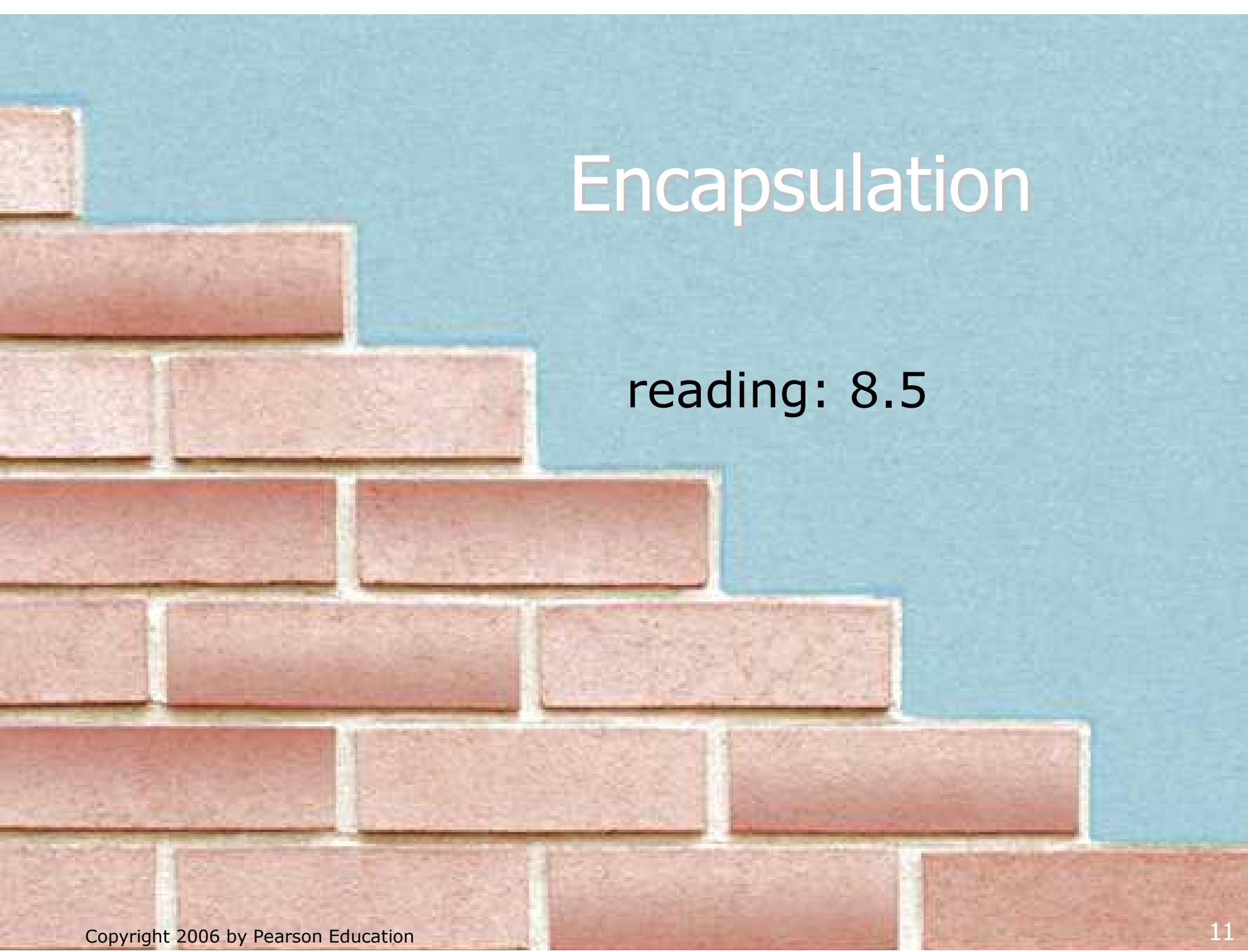
- Recall our client program that produces this output:

```
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));
    }
}
```

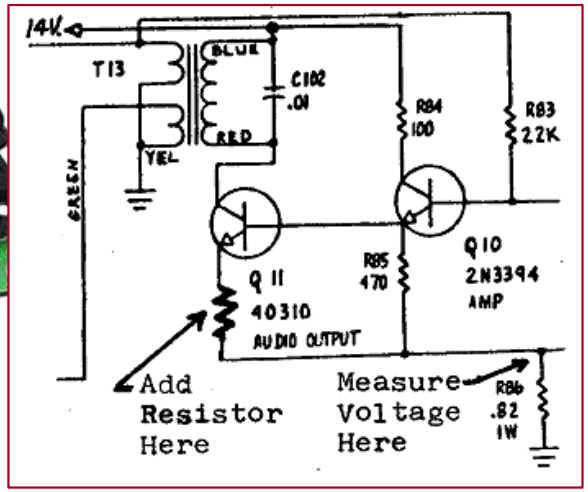


Encapsulation

- **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 + ")");
                                  ^
```

Accessing encapsulated state

- We can provide 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 methods to change it:

```
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);
```

Accessors and mutators

Two common categories of instance methods used with encapsulated objects:

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

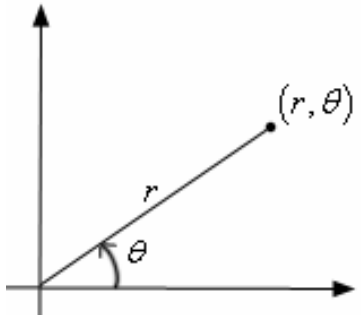
- **mutator**: Modifies an object's state.
  - Sometimes the change is based on parameters (e.g. `dx`, `dy`).
  - Examples: `translate`, `setLocation`, `setY`

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

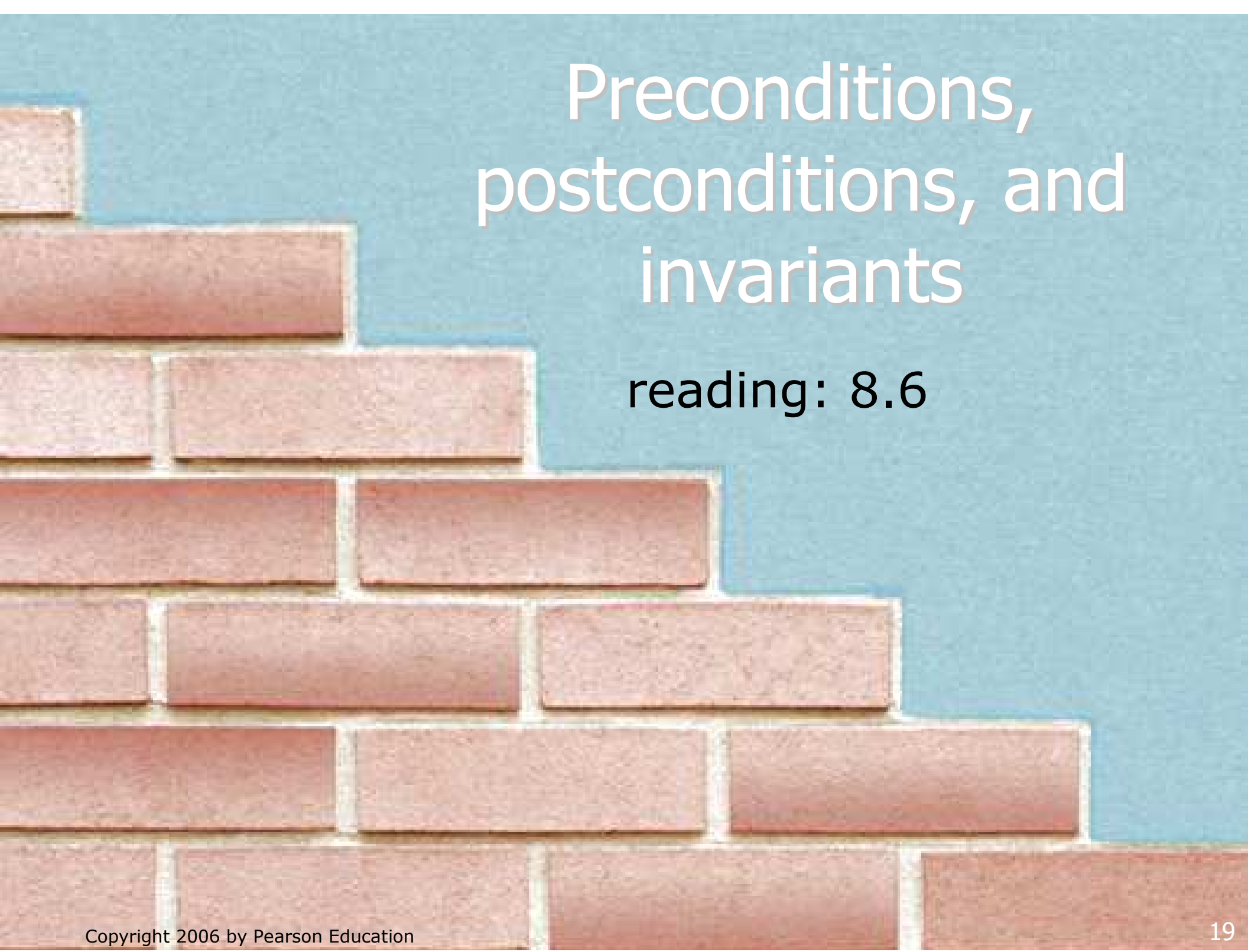
```
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    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;
}
```