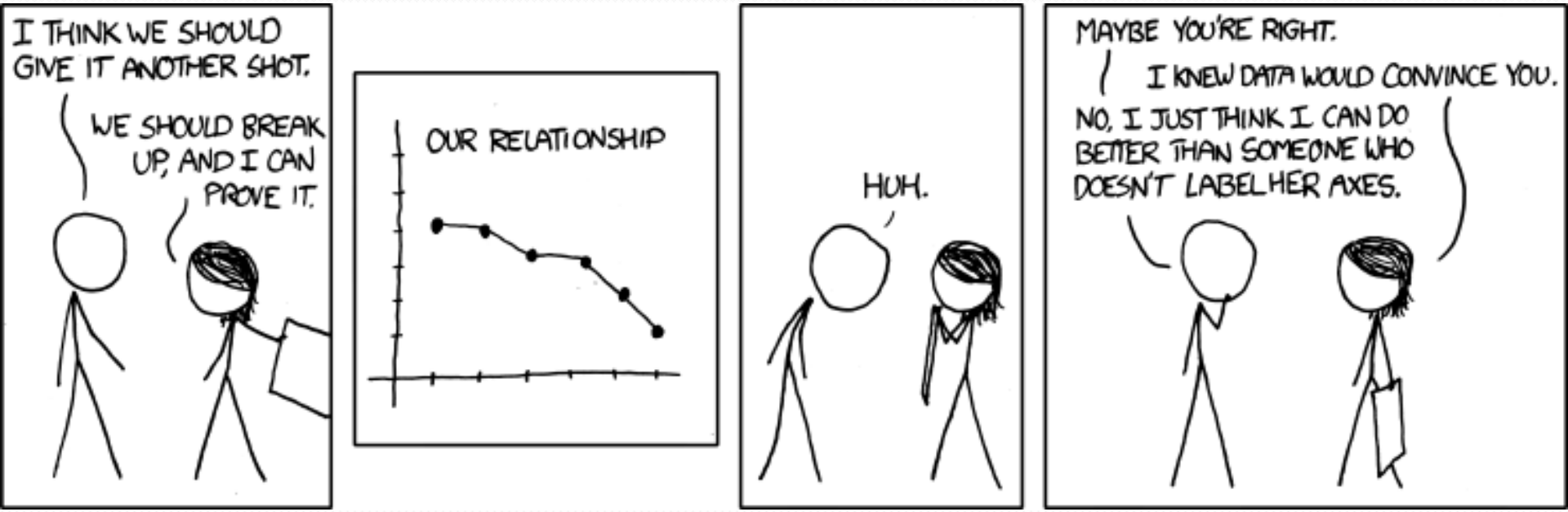
Building Java Programs

Chapter 8

Lecture 18: Classes and Objects

reading: 8.1 - 8.2

(Slides adapted from Stuart Reges, Hélène Martin, and Marty Stepp)



File output

reading: 6.4 - 6.5

Output to files

- **`PrintStream`**: An object in the `java.io` package that lets you print output to a destination such as a file.
  - Any methods you have used on `System.out` (such as `print`, `println`) will work on a `PrintStream`.

- Syntax:

```
PrintStream <name> = new PrintStream(new File("<filename>"));
```

Example:

```
PrintStream output = new PrintStream(new File("out.txt"));
output.println("Hello, file!");
output.println("This is a second line of output.");
```

Details about `PrintStream`

```
PrintStream <name> = new PrintStream(new File("<filename>"));
```

- If the given file does not exist, it is created.
- If the given file already exists, it is overwritten.

- The output you print appears in a file, not on the console. You will have to open the file with an editor to see it.

- Do not open the same file for both reading (`Scanner`) and writing (`PrintStream`) at the same time.
  - You will overwrite your input file with an empty file (0 bytes).

`System.out` and `PrintStream`

- The console output object, `System.out`, is a `PrintStream`.

```
PrintStream out1 = System.out;
PrintStream out2 = new PrintStream(new File("data.txt"));
out1.println("Hello, console!");   // goes to console
out2.println("Hello, file!");      // goes to file
```

  - A reference to it can be stored in a `PrintStream` variable.
    - Printing to that variable causes console output to appear.
  - You can pass `System.out` to a method as a `PrintStream`.
    - Allows a method to send output to the console or a file.

Objects and classes

reading: 8.1 – 8.2

Clients of objects

- **client program**: A program that uses objects.
  - Example: `Shapes` is a client of `DrawingPanel` and `Graphics`.

`Shapes.java` (client program)

```
public class Shapes {
    main(String[] args) {
        new DrawingPanel(...)
        new DrawingPanel(...)
        ...
    }
}
```

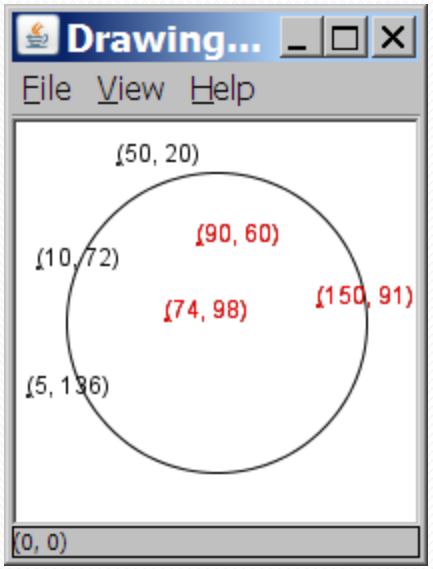
`DrawingPanel.java` (class)

```
public class DrawingPanel {
    ...
}
```

A programming problem

- Given a file of cities' (x, y) coordinates, which begins with the number of cities:

```
6
50 20
90 60
10 72
74 98
5 136
150 91
```



- Write a program to draw the cities on a `DrawingPanel`, then simulates an earthquake that turns all cities red that are within a given radius:

```
Epicenter x? 100
Epicenter y? 100
Affected radius? 75
```

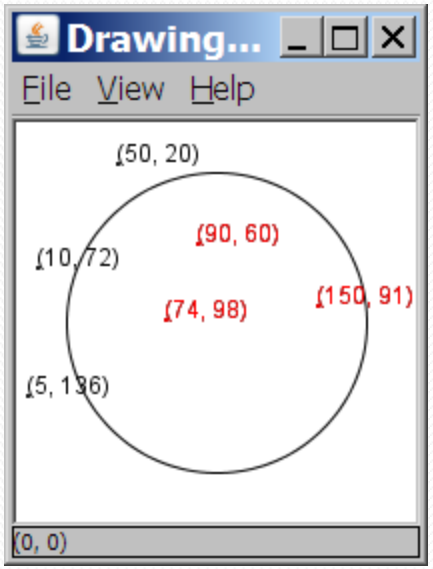
A bad solution

```
Scanner input = new Scanner(new File("cities.txt"));
int cityCount = input.nextInt();
int[] xCoords = new int[cityCount];
int[] yCoords = new int[cityCount];

for (int i = 0; i < cityCount; i++) {
    xCoords[i] = input.nextInt();   // read each city
    yCoords[i] = input.nextInt();
}
...
```

- **parallel arrays**: 2+ arrays with related data at same indexes.
  - Considered poor style.

Observations

- The data in this problem is a set of points.
- It would be better stored as `Point` objects.
  - A `Point` would store a city's x/y data.
  - We could compare distances between `Point`s to see whether the earthquake hit a given city.
  - Each `Point` would know how to draw itself.
  - The overall program would be shorter and cleaner.



Classes and objects

- **class**: A program entity that represents either:
  1. A program / module, or
  2. **A template for a new type of objects.**

  - The `DrawingPanel` class is a template for creating `DrawingPanel` objects.

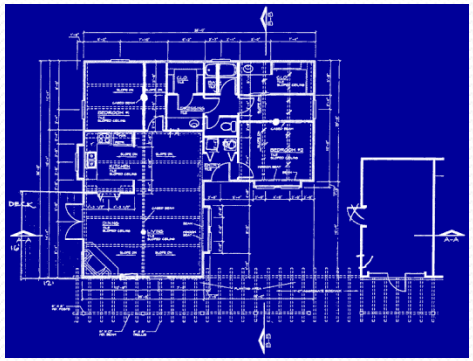
- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.

Blueprint analogy

iPod blueprint

state:
- current song
- volume
- battery life

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



creates

iPod #1

state:
- song = "1,000,000 Miles"
- volume = 17
- battery life = 2.5 hrs

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

iPod #2

state:
- song = "Letting You"
- volume = 9
- battery life = 3.41 hrs

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

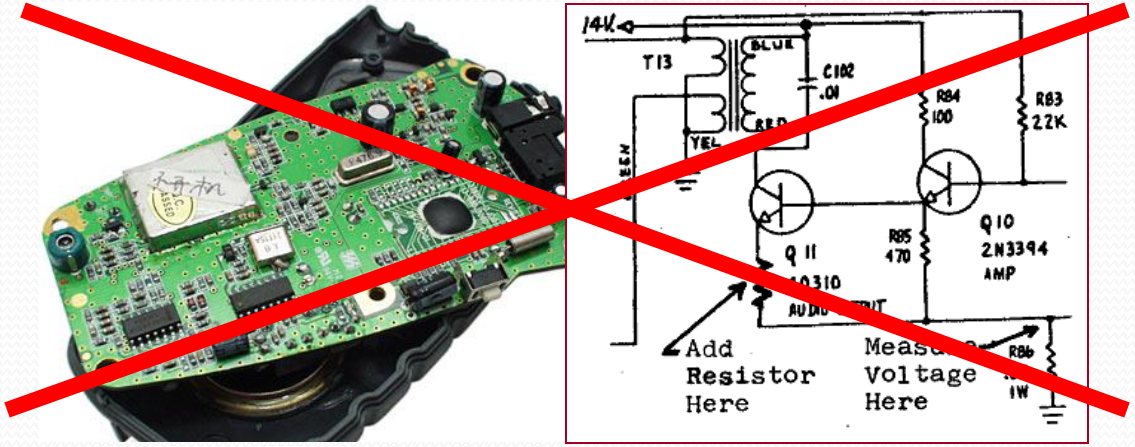
iPod #3

state:
- song = "Discipline"
- volume = 24
- battery life = 1.8 hrs

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

Abstraction

- **abstraction**: A distancing between ideas and details.
  - We can use objects without knowing how they work.
- abstraction in an iPod:
  - You understand its external behavior (buttons, screen).
  - You don't understand its inner details, and you don't need to.



Our task

- In the following slides, we will implement a `Point` class as a way of learning about defining classes.
  - We will define a type of objects named `Point`.
  - Each `Point` object will contain x/y data called **fields**.
  - Each `Point` object will contain behavior called **methods**.
  - **Client programs** will use the `Point` objects.

`Point` objects (desired)

```
Point p1 = new Point(5, -2);
Point p2 = new Point();          // origin, (0, 0)
```

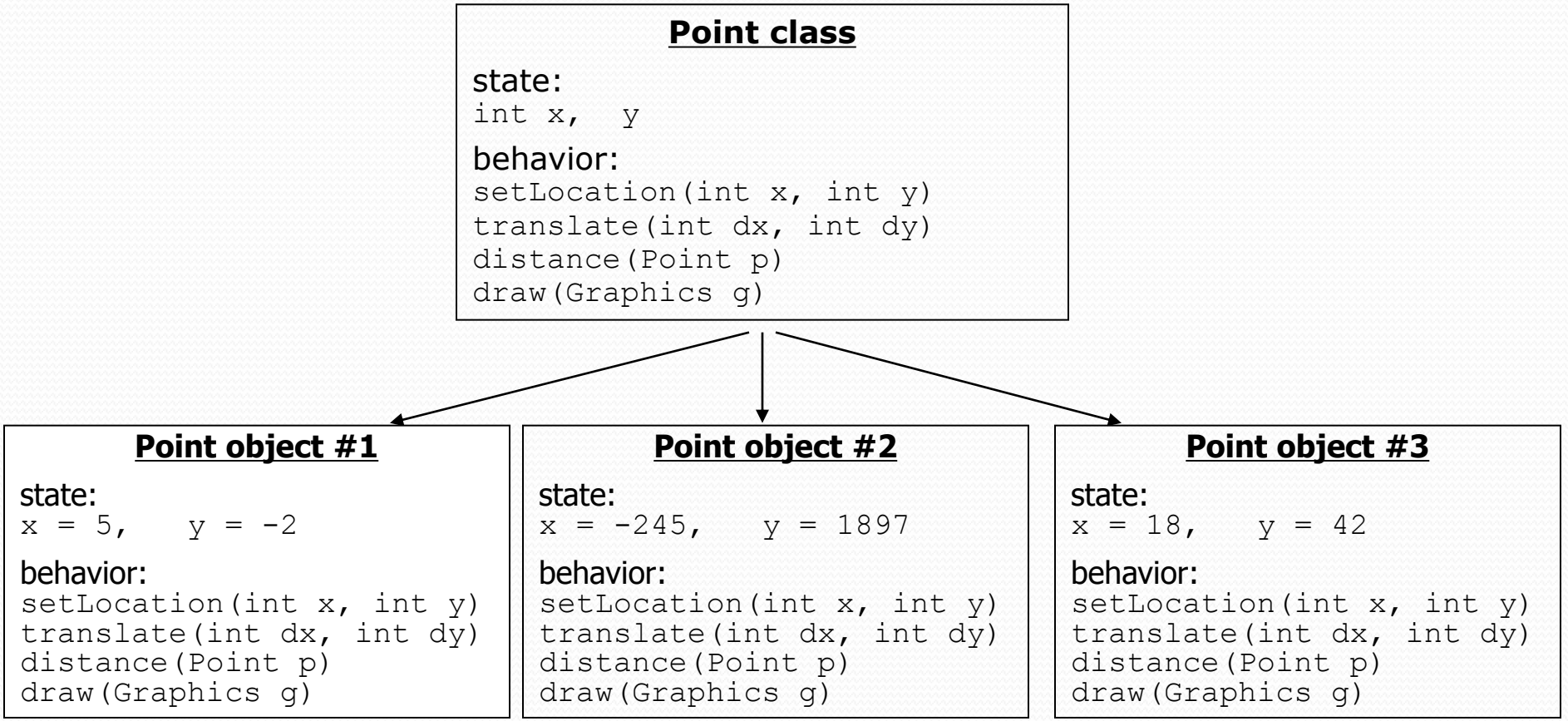
- Data in each `Point` object:

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

- Methods in each `Point` object:

Method name	Description
`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
`distance(`**p**`)`	how far away the point is from point *p*
`draw(`**g**`)`	displays the point on a drawing panel

`Point` class as blueprint



- The class (blueprint) will describe how to create objects.
- Each object will contain its own data and methods.

Object state: Fields

reading: 8.2

Point class, version 1

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

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

- The above code creates a new type named `Point`.
  - Each `Point` object contains two pieces of data:
    - an `int` named `x`, and
    - an `int` named `y`.
  - `Point` objects do not contain any behavior (yet).

Fields

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

- Declaration syntax:

```
type name;
```

  - Example:

```
public class Student {
    String name;     // each Student object has a
    double gpa;      // name and gpa field
}
```

Accessing fields

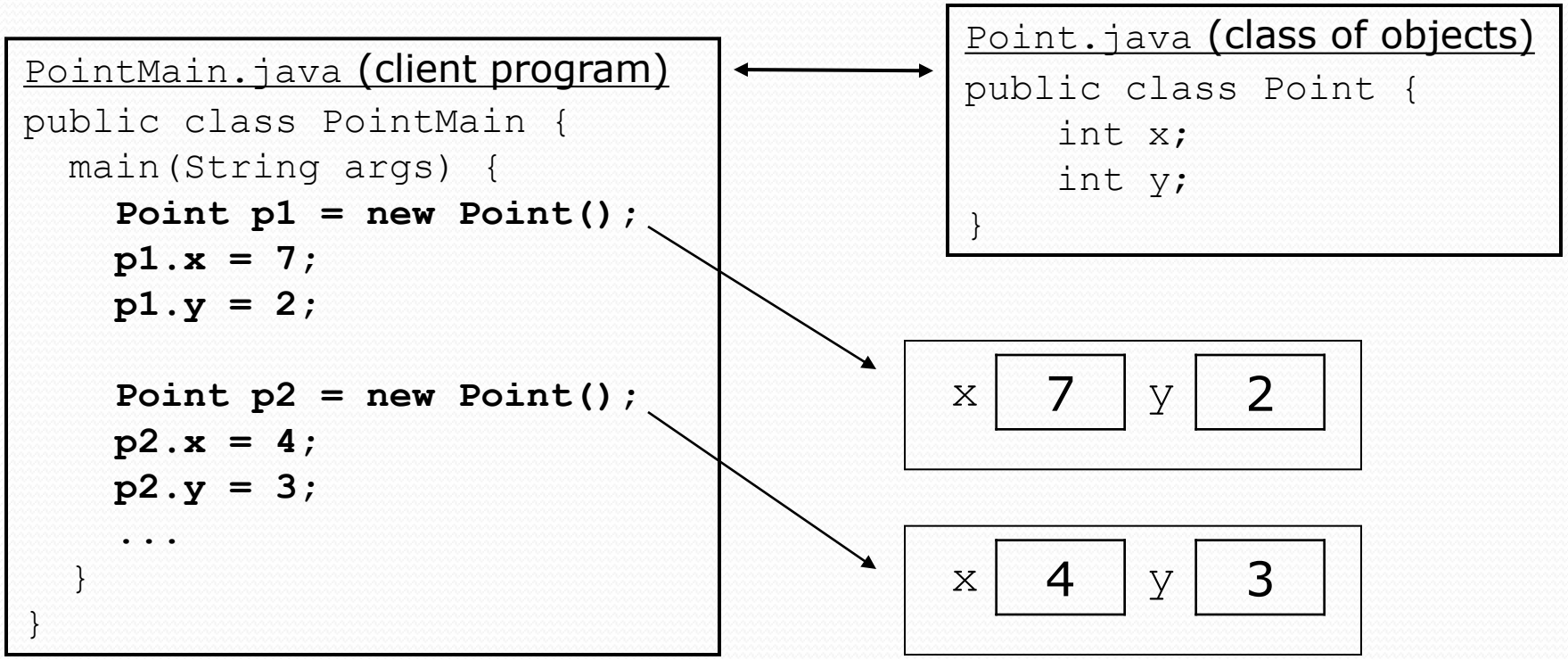
- Other classes can access/modify an object's fields.
  - access: **variable.field**
  - modify: **variable.field** = **value**;

- Example:

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

A class and its client

- `Point.java` is not, by itself, a runnable program.
  - A class can be used by **client** programs.



PointMain client example

```
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.x + ", " + p1.y);   // 0, 2

        // move p2 and then print it
        p2.x += 2;
        p2.y++;
        System.out.println(p2.x + ", " + p2.y);   // 6, 1
    }
}
```

Object behavior: Methods

reading: 8.3

Client code redundancy

- Suppose our client program wants to draw `Point` objects:

```
// draw each city
Point p1 = new Point();
p1.x = 15;
p1.y = 37;
g.fillOval(p1.x, p1.y, 3, 3);
g.drawString("(" + p1.x + ", " + p1.y + ")", p1.x, p1.y);
```

- To draw other points, the same code must be repeated.
  - We can remove this redundancy using a method.

Eliminating redundancy, v1

- We can eliminate the redundancy with a static method:

```
// Draws the given point on the DrawingPanel.
public static void draw(Point p, Graphics g) {
    g.fillOval(p.x, p.y, 3, 3);
    g.drawString("(" + p.x + ", " + p.y + ")", p.x, p.y);
}
```

- `main` would call the method as follows:

```
draw(p1, g);
```

Problems with static solution

- We are missing a major benefit of objects: code reuse.
  - Every program that draws `Point`s would need a `draw` method.

- The syntax doesn't match how we're used to using objects.

```
draw(p1, g);     // static (bad)
```

- The point of classes is to combine state and behavior.
  - The `draw` behavior is closely related to a `Point`'s data.
  - The method belongs *inside* each `Point` object.

```
p1.draw(g);      // inside the object (better)
```

Instance methods

- **instance method** (or **object method**): Exists inside each object of a class and gives behavior to each object.

```
public type name(parameters) {
    statements;
}
```

  - same syntax as static methods, but without `static` keyword

Example:

```
public void shout() {
    System.out.println("HELLO THERE!");
}
```

Instance method example

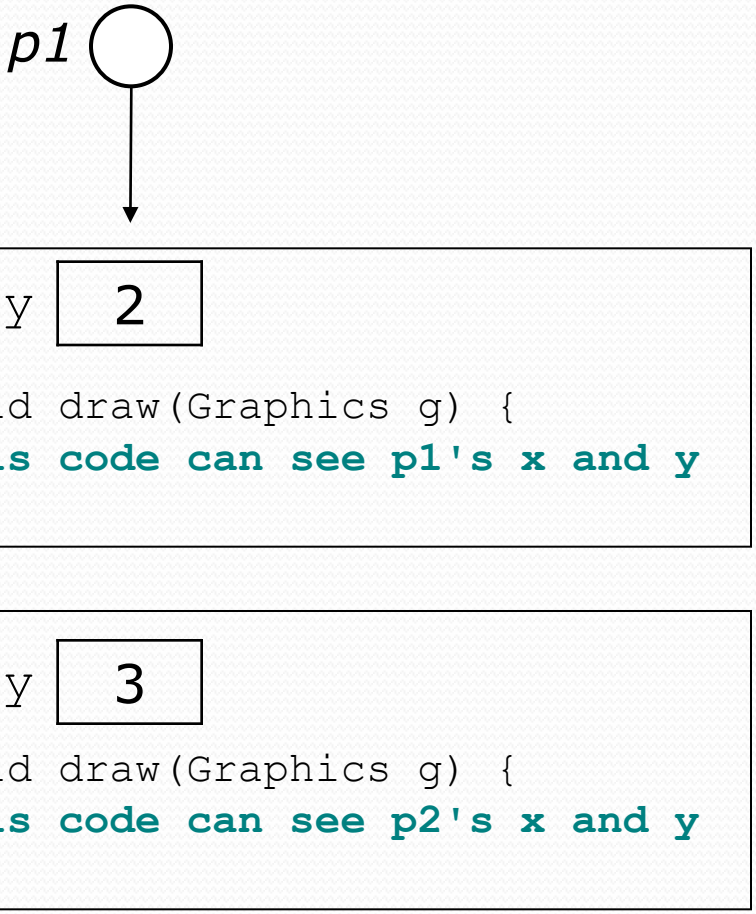
```
public class Point {
    int x;
    int y;

    // Draws this Point object with the given pen.
    public void draw(Graphics g) {
        ...
    }
}
```

- The `draw` method no longer has a `Point p` parameter.
- How will the method know which point to draw?
  - How will the method access that point's x/y data?

`Point` objects w/ method

- Each `Point` object has its own copy of the `draw` 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.draw(g);
p2.draw(g);
```



The implicit parameter

- **implicit parameter**:
  The object on which an instance method is called.
  - During the call `p1.draw(g);`
    the object referred to by `p1` is the implicit parameter.
  - During the call `p2.draw(g);`
    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.
    - `draw` can refer to the `x` and `y` 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 draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, y);
    }
}
```

- Each `Point` object contains a `draw` method that draws that point at its current `x`/`y` position.

Class method questions

- Write a method `translate` that changes a `Point`'s location by a given *dx*, *dy* amount.

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

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

  - Modify the `Point` and client code to use these methods.

Class method answers

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

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

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

Kinds of methods

- **accessor**: A method that lets clients examine object state.
  - Examples: `distance`, `distanceFromOrigin`
  - often has a non-`void` return type

- **mutator**: A method that modifies an object's state.
  - Examples: `setLocation`, `translate`

Why objects?

- Primitive types don't model complex concepts well
  - Cost is a double. What's a person?
  - Classes are a way to define new types
  - Many objects can be made from those types

- Values of the same type often are used in similar ways
  - Promote code reuse through instance methods

Object initialization: constructors

reading: 8.3

Initializing objects

- Currently it takes 3 lines to create a `Point` and initialize it:

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

- We'd rather specify the fields' initial values at the start:

```
Point p = new Point(3, 8);    // desired; doesn't work (yet)
```

  - We are able to this with most types of objects in Java.

Constructors

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

```
public type(parameters) {
    statements;
}
```

  - runs when the client uses the `new` keyword
  - no return type is specified;
    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 fields to 0.

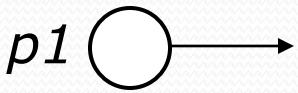
Constructor example

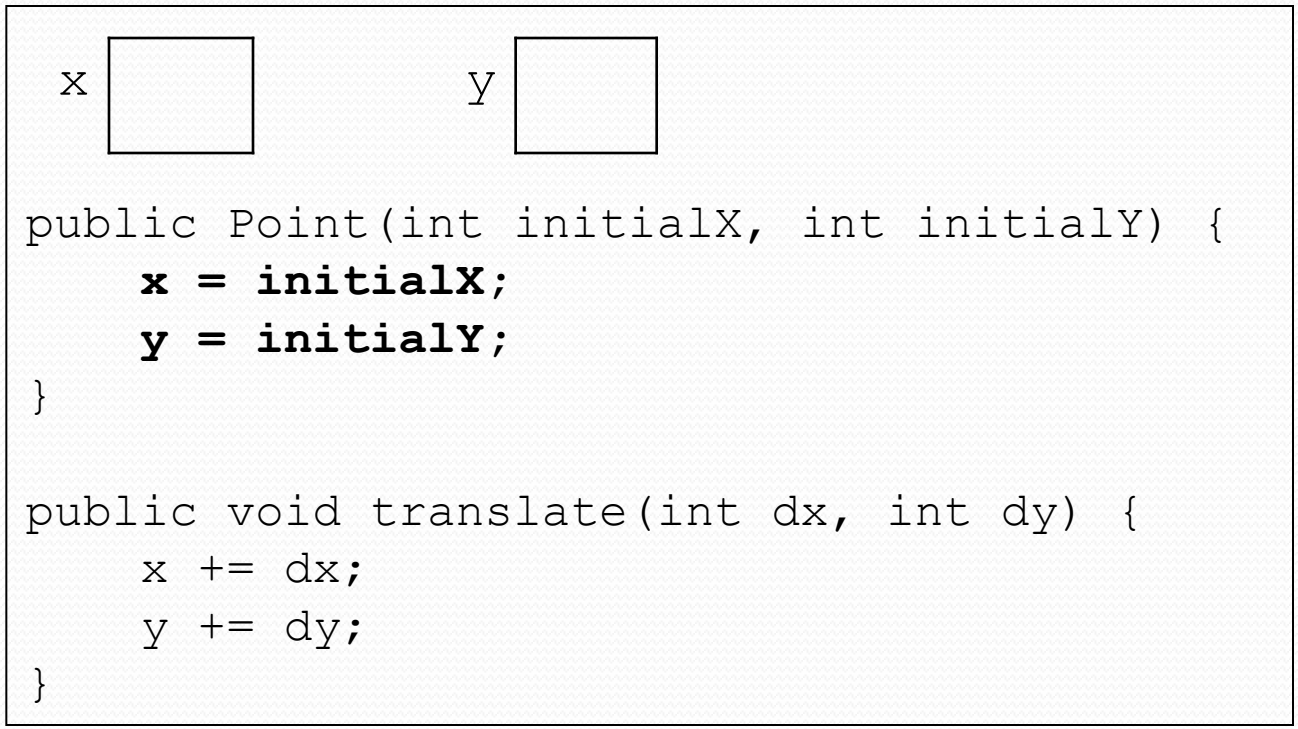
```
public class Point {
    int x;
    int y;

    // Constructs a Point at the given x/y location.
    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }


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

    ...
}
```

Tracing a constructor call

- 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;
}
```

Common constructor bugs

1. Re-declaring fields as local variables ("shadowing"):

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

- This declares local variables with the same name as the fields, rather than storing values into the fields. The fields remain 0.

2. Accidentally giving the constructor a return type:

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

- This is actually not a constructor, but a method named `Point`

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

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

OUTPUT:

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

Multiple constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.

- *Exercise:* Write a `Point` constructor with no parameters that initializes the point to (0, 0).

```
// Constructs a new point at (0, 0).
public Point() {
    x = 0;
    y = 0;
}
```

Printing objects

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

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


// better, but cumbersome;              p is (10, 7)
System.out.println("p is (" + p.x + ", " + p.y + ")");


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

The `toString` method

tells Java how to convert an object into a `String`

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

// the above code is really calling the following:
System.out.println("p1: " + p1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - Default: class's name `@` object's memory address (base 16)

```
Point@9e8c34
```

`toString` syntax

```
public String toString() {
    code that returns a String representing this object;
}
```

- Method name, return, and parameters must match exactly.
- Example:

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

The keyword `this`

reading: 8.7

this

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

- Syntax for using `this`:
  - To refer to a field:
    `this.`**`field`**
  - To call a method:
    `this.`**`method`**`(`**`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.

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

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

  - The parameters to `setLocation` are named `newX` and `newY` to be distinct from the object's fields `x` and `y`.

Variable shadowing

- An instance method parameter can have the same name as one of the object's fields:

```
// 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.

Avoiding shadowing w/ `this`

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

    ...

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

    ...
}
```

Constructors and `this`

- One constructor can call another using `this`:

```
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;
    }

    ...
}
```