# Building Java Programs

## A Back to Basics Approach

CS 210

### CHAPTER 8

### CLASSES

Please download the PPT, and use Slide Show for a better viewing experience

***Winnie Li***

07/30/2021

# Topics will be covered

- Classes and Objects
- Object state: Fields
- Object behavior: Methods
- The `null` reference
- The `toString` method
- Object initialization: Constructors
- The keyword `this`
- Encapsulation

# Classes and Objects

CS 210

## reading: 8.1

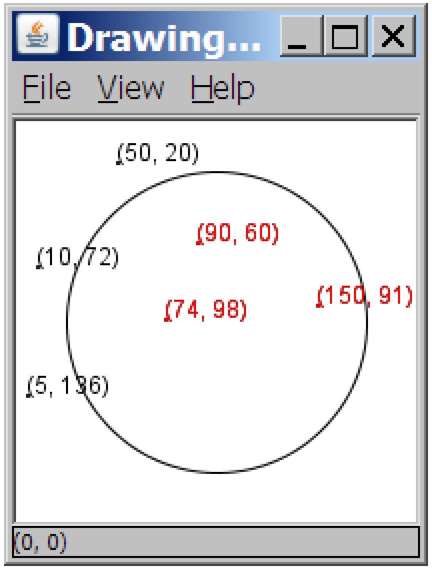
07/30/2021

# A programming problem

CS 210

- 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 drop a "bomb" that turns all cities red that are within a given radius:

```
Blast site x? 100
Blast site y? 100
Blast radius? 75
Kaboom!
```

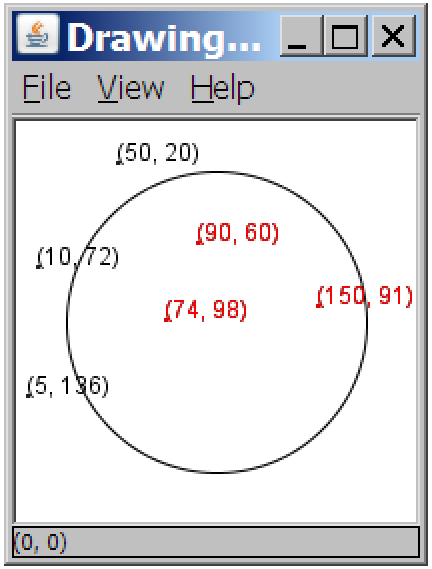
# A possible solution

```
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 x and y coordinates in this program are actually points.
  - They don't really have any meaning separately.



- It would be better if we had `Point` objects:
  - A `Point` would store a city's x/y data.
  - Each `Point` would know how to draw itself.
  - We could compare distances between `Points` to see whether to bomb a given city.

- This would make the overall program 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.

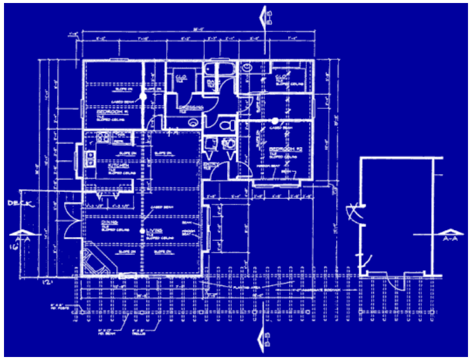
- **object**: An entity that combines state and behavior.
  - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.

# 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/son
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

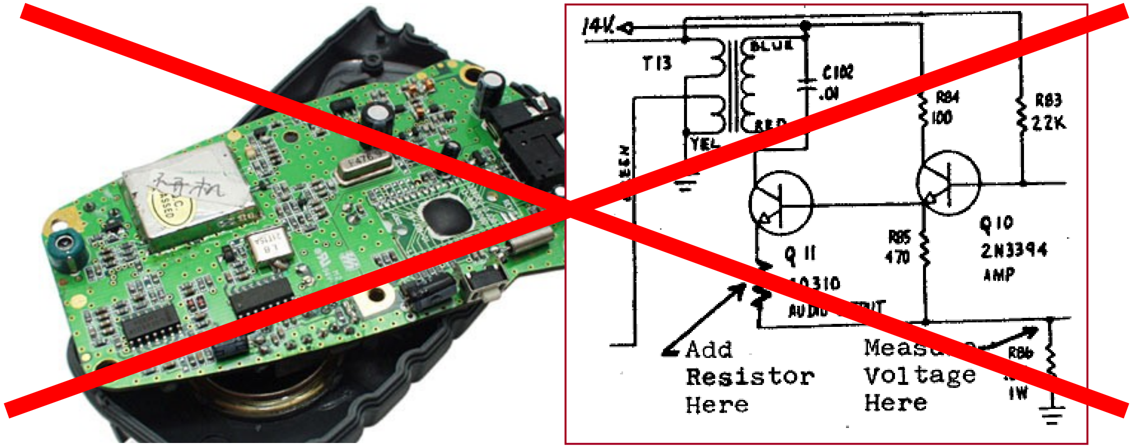
**iPod #3**

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

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

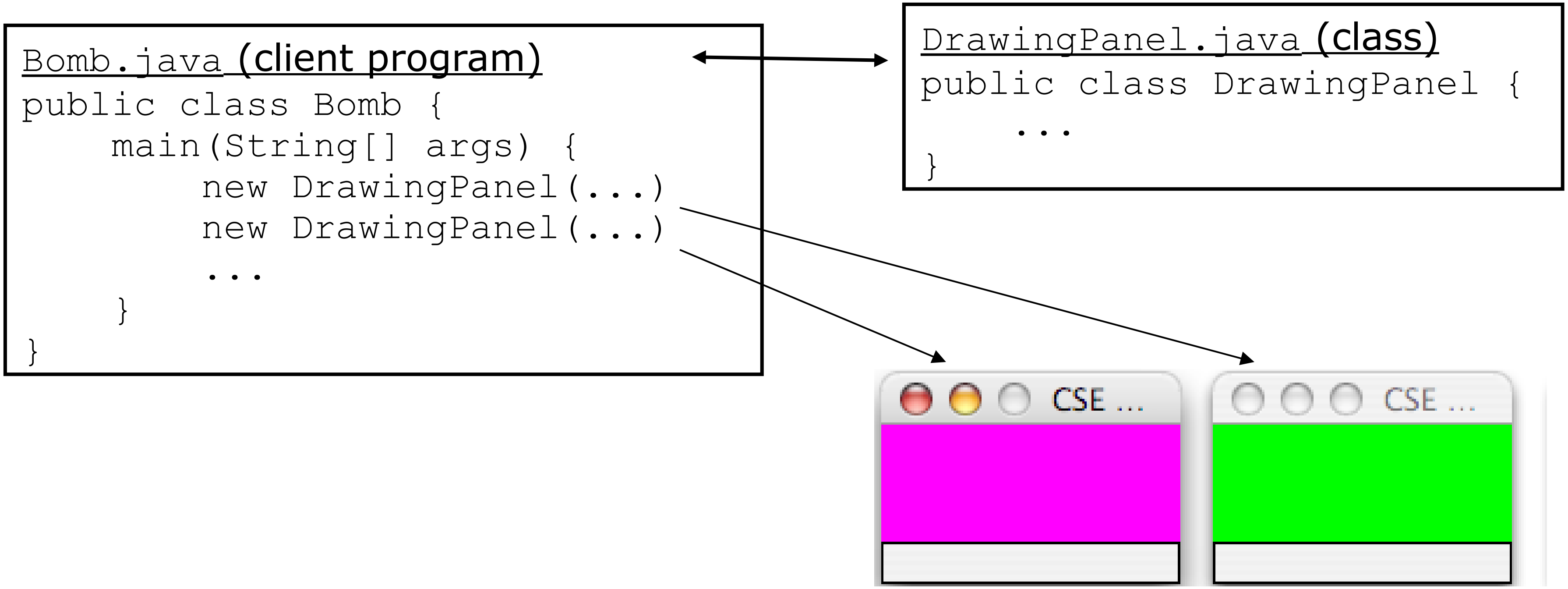
# Abstraction

- **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.**



# Clients of objects

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



Bomb.java (client program)

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

DrawingPanel.java (class)

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

# Our task

- We will implement a `Point` class as a way of learning about defining classes and to make our bomb program simpler.

- We will define a type named `Point`.
  - Each `Point` object will contain data called **fields**.
    - e.g. The x and y coordinates.
  - Each `Point` object will contain behavior called **methods**.
    - e.g. Calculate the distance from another point
  - **Client programs** will use the `Point` objects.
    - Our bomb program is one such client.

# Point objects (desired)

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

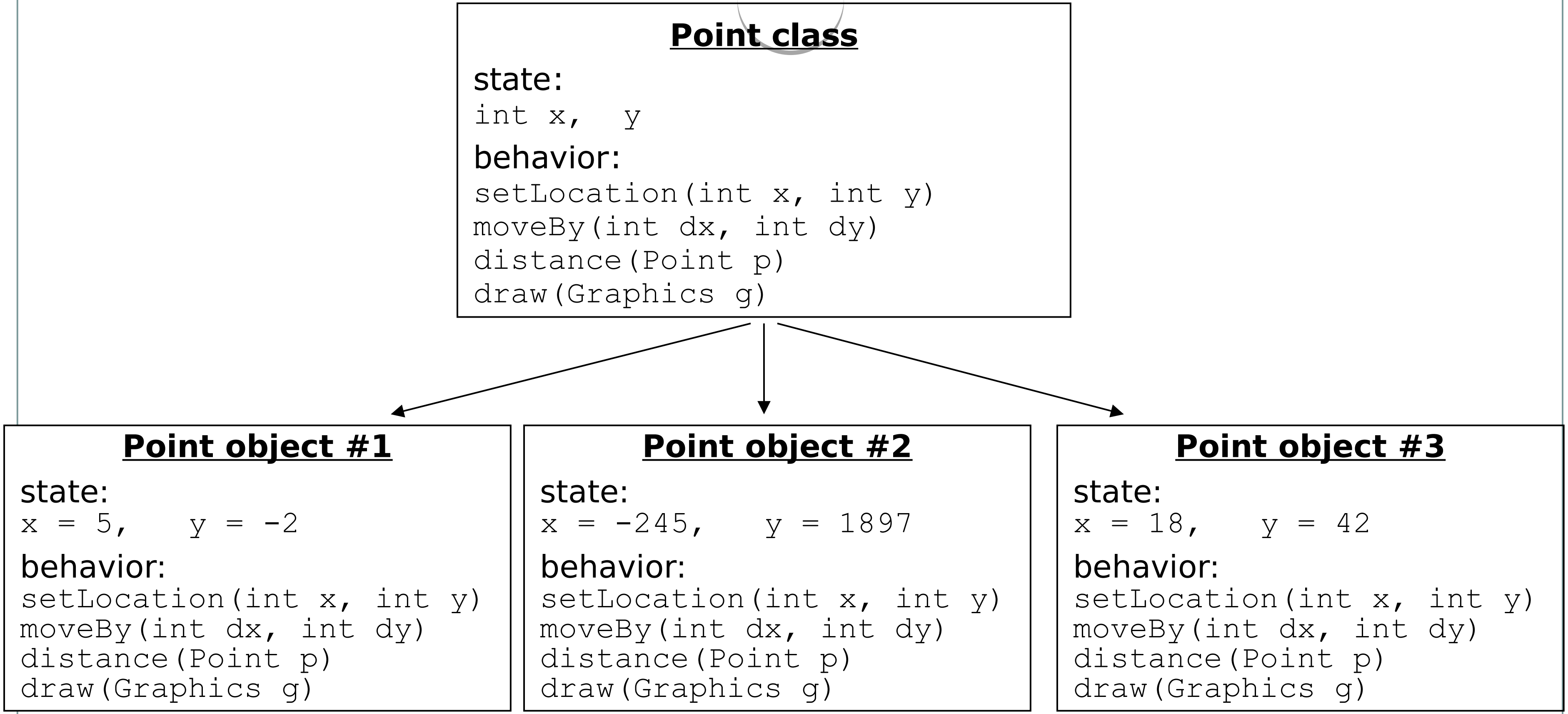
- Data in each `Point` object:

| 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 |
| `moveBy(`**`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



**Point class**

state:

```
int x,  y
```

behavior:

```
setLocation(int x, int y)
moveBy(int dx, int dy)
distance(Point p)
draw(Graphics g)
```

**Point object #1**

state:

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

behavior:

```
setLocation(int x, int y)
moveBy(int dx, int dy)
distance(Point p)
draw(Graphics g)
```

**Point object #2**

state:

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

behavior:

```
setLocation(int x, int y)
moveBy(int dx, int dy)
distance(Point p)
draw(Graphics g)
```

**Point object #3**

state:

```
x = 18,    y = 42
```

behavior:

```
setLocation(int x, int y)
moveBy(int dx, int dy)
distance(Point p)
draw(Graphics g)
```

# Object state: Fields

CS 210

**reading: 8.2**

# 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
}
```

# 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).

# Accessing fields

- Other classes can access/modify an object's fields.

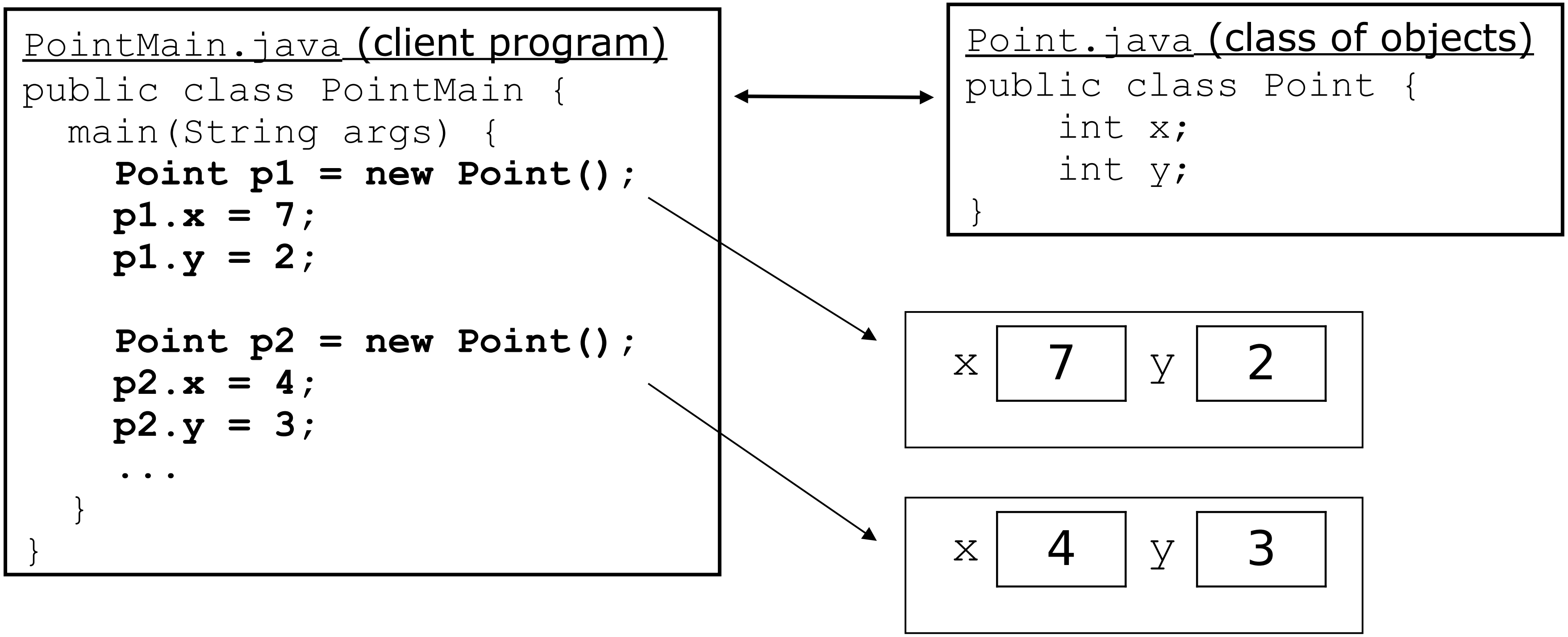
  - access: **variable . field**
  - modify: **variable . field** = **value ;**

- 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. Why not?
  - It does not contain a `main` method.
- A class can be used by **client** programs



PointMain.java (client program)

```
public class PointMain {
  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;
}
```

# Bomb client v1

```
public class Bomb1 {
   ...

   // read in city locations
   int cityCount = input.nextInt();
   Point[] cities = new Point[cityCount];

   for (int i = 0; i < cityCount; i++) {
       cities[i] = new Point();
       cities[i].x = input.nextInt();
       cities[i].y = input.nextInt();
   }
   ...

   public static void drawCities(Graphics g, Point[] cities) {
       for (int i = 0; i < cities.length; i++) {
       g.fillOval(cities[i].x, cities[i].y, 3, 3);
       g.drawString("(" + cities[i].x + ", " + cities[i].y + ")",
            cities[i].x, cities[i].y);
   }
   ...

}
```

# Object behavior: Methods

CS 210

# Client code redundancy

- Our code to draw and bomb cities is redundant:

```
public static void drawCities(Graphics g, Point[] cities) {
      for (int i = 0; i < cities.length; i++) {
          g.fillOval(cities[i].x, cities[i].y, 3, 3);
          g.drawString("(" + cities[i].x + ", " + cities[i].y + ")",
                    cities[i].x, cities[i].y);
      }
}

public static void bombCities(Graphics g, Point[] cities, Point
   quarPoint, int quarRad) {
      ...
      if (distanceBetween(quarPoint, cities[i]) <= quarRad) {
           g.fillOval(cities[i].x, cities[i].y, 3, 3);
           g.drawString("(" + cities[i].x + ", " + cities[i].y + ")",

                          cities[i].x, cities[i].y);
      ...
}
```

- We *could* use a static method to resolve this, but we can do better.

# Limitations with static solution

- We are missing a major benefit of objects: code reuse.
  - Every program that draws `Points` would need a `draw` method.
  - If we wanted to change how `Points` are drawn, we'd need to change every program that uses `Points`.

- The whole 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>) {
     <statement(s)>;
}
```

  - 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 data?

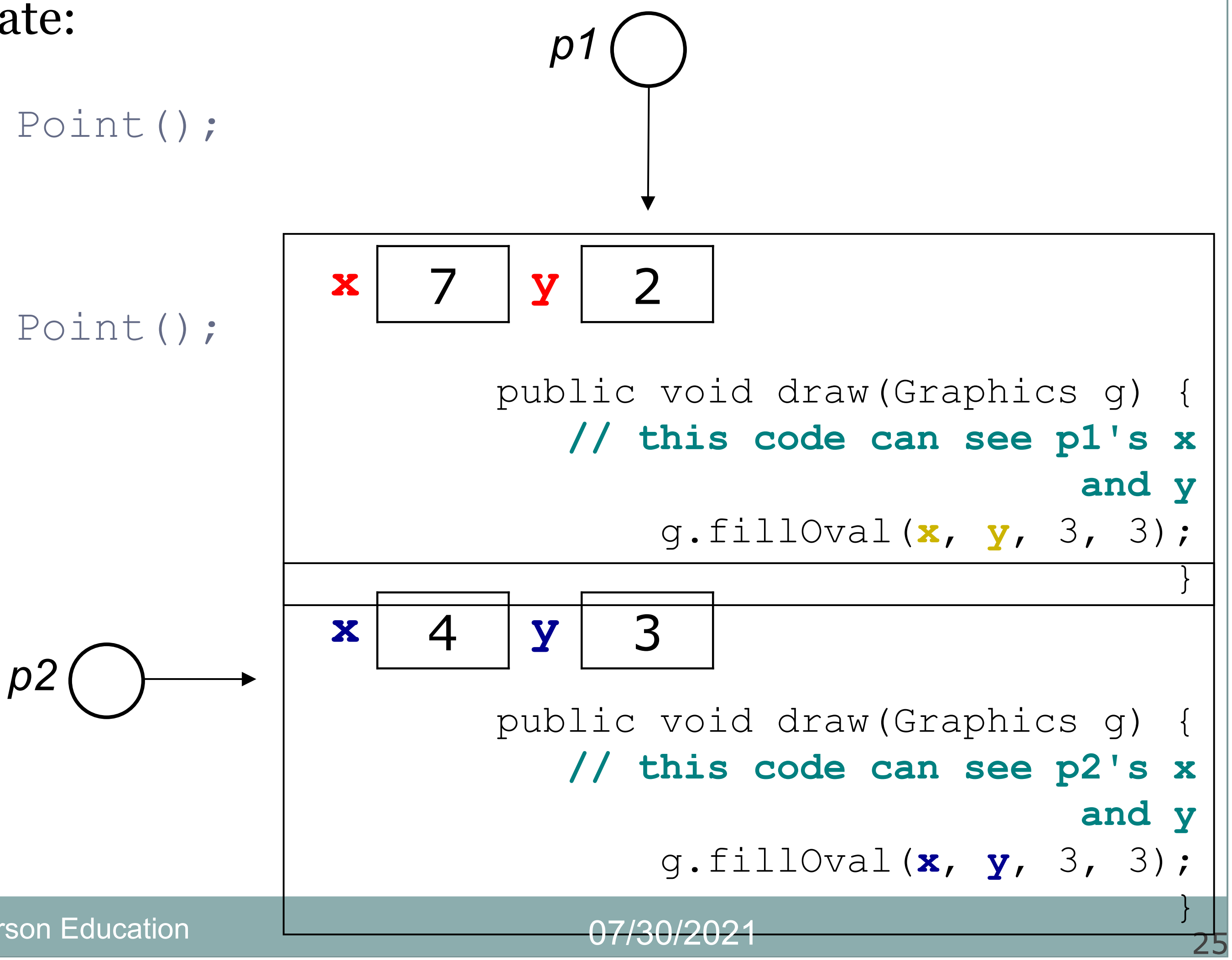
Instance method may access the data field directly, no need to pass the parameter in.

# `Point` objects w/ method

CS 210

- 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);
```



```
public void draw(Graphics g) {
    // this code can see p1's x
                           and y
         g.fillOval(x, y, 3, 3);
}
```

```
public void draw(Graphics g) {
    // this code can see p2's x
                           and y
         g.fillOval(x, y, 3, 3);
}
```

# Kinds of methods

CS 210

- **accessor**: A method that lets clients examine object state.
  - Usually has a non-`void` return type
  - Write a method `distance` that computes the distance between a `Point` and another `Point` parameter.

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

```
public double distance(Point other) {
    int dx = x - other.x;
    int dy = y - other.y;
    return Math.sqrt(dx * dx + dy * dy);
}
```

- **mutator**: A method that modifies an object's state.
  - Usually has a `void` return type
  - Write a method `setLocation` that changes a `Point`'s location to the (*x*, *y*) values passed.

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

# Point class, version 2

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

    // Draw this Point.
    public void draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, y);
    }

    // Changes the location of this Point object.
    public void setLocation(int newX, int newY) {
        x = newX;
        y = newY;
    }

    // Shift this Point the given amounts.
    public void moveBy(int dx, int dy) {
        setLocation(x + dx, y + dy);
    }

    // Calculate the distance between this Point and another one.
    public double distance(Point other) {
        int dx = x - other.x;
        int dy = y - other.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

# bomb client, v2

```
public class Bomb2 {
 ...

 public static void drawCities(Graphics g, Point2[] cities) {
      for (int i = 0; i < cities.length; i++) {
         cities[i].draw(g);
      }
   }

   public static void bombCities(Graphics g, Point2[] cities, Point2

                                       quarPoint, int quarRad) {
      g.setColor(Color.RED);
      System.out.println(“Bombed cities: ");

      g.drawOval(quarPoint.x - quarRad, quarPoint.y - quarRad,
                 2 * quarRad, 2 * quarRad);

      for (int i = 0; i < cities.length; i++) {
         if (quarPoint.distance(cities[i]) <= quarRad) {
           cities[i].draw(g);
           System.out.println("\t(" + cities[i].x + ", " + cities[i].y + ")");
         }
      }
   }
}
```

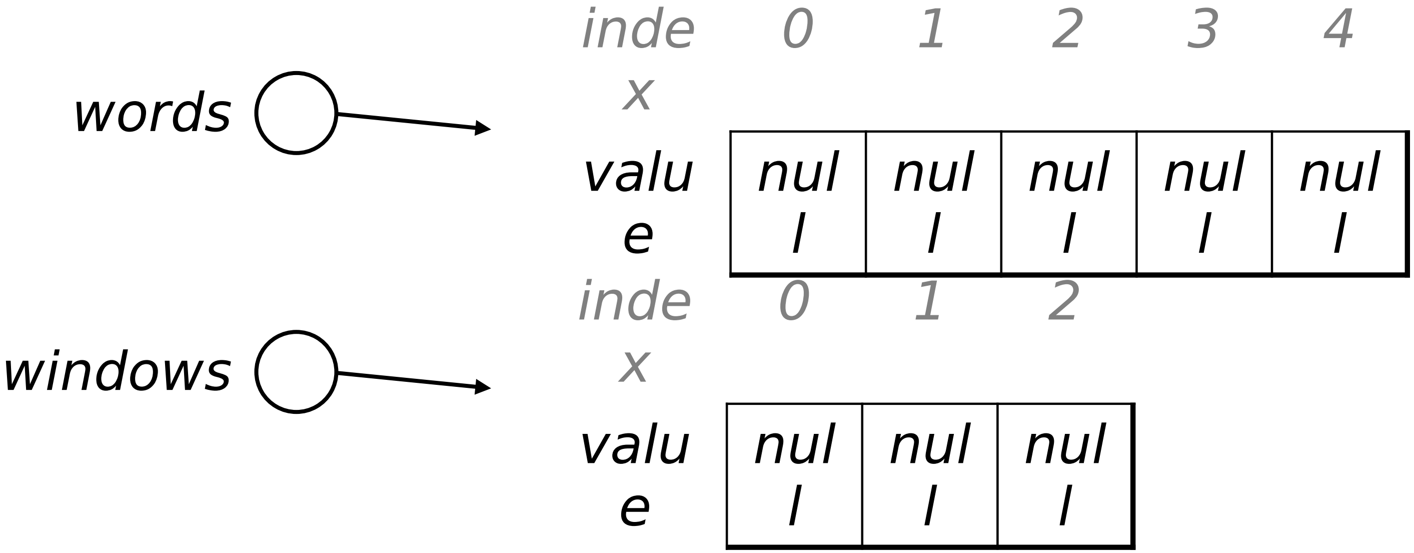
# The `null` reference

## reading: 8.3

# Arrays of objects

- `null` **:** A value that does not refer to any object.
  - The elements of an array of objects are initialized to `null`.

```
String[] words = new String[5];
DrawingPanel[] windows = new DrawingPanel[3];
```



words

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | null | null | null | null | null |

windows

| index | 0 | 1 | 2 |
|---|---|---|---|
| value | null | null | null |

# Things you can do w/ `null`

- store `null` in a variable or an array element

```
String s = null;
words[2] = null;
```

- print a `null` reference

```
System.out.println(s);         // null
```

- ask whether a variable or array element is `null`

```
if (words[2] == null) { ...
```

- pass `null` as a parameter to a method

```
System.out.println(null);   // null
```

- return `null` from a method (often to indicate failure)

```
return null;
```

# Null pointer exception

- **dereference**: To access data or methods of an object with the dot notation, such as `s.length()` .
  - It is illegal to dereference `null` (causes an exception).
  - `null` is not an object, so it has no methods or data.

```
String[] words = new String[5];
words[0] = "hello";
words[2] = "goodbye";   // words[1], [3], [4] are null
for (int i = 0; i < words.length; i++) {
    System.out.println("word is: " + words[i]);
    words[i] = words[i].toUpperCase();     // ERROR
}
```

**Output:**

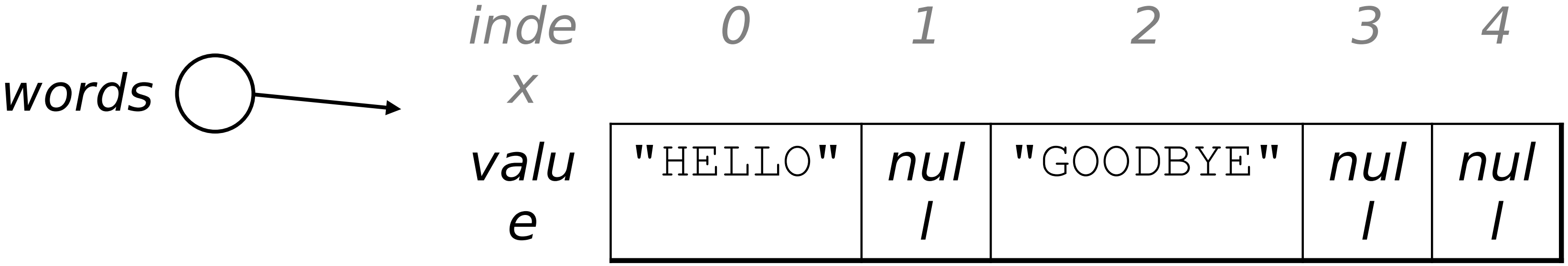
```
word is: hello
word is: null
Exception in thread "main"
java.lang.NullPointerException
```

| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | "HELLO" | null | "goodbye" | null | null |

# Looking before you leap

- You can check for `null` before calling an object's methods.

```
String[] words = new String[5];
words[0] = "hello";
words[2] = "goodbye";    // words[1], [3], [4] are null

for (int i = 0; i < words.length; i++) {
    if (words[i] != null) {
        words[i] = words[i].toUpperCase();
    }
}
```



words

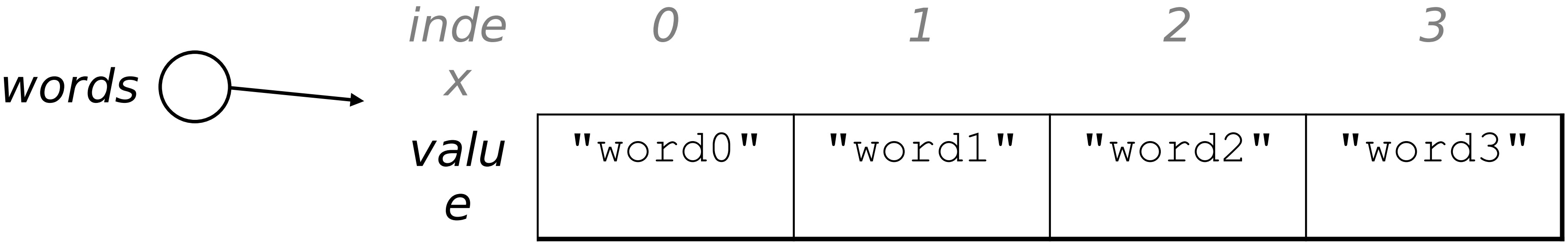
| index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| value | "HELLO" | *null* | "GOODBYE" | *null* | *null* |

# Two-phase initialization

1) initialize the array itself (each element is initially `null`)

2) initialize each element of the array to be a new object

```
String[] words = new String[4];              // phase 1
for (int i = 0; i < words.length; i++) {
    coords[i] = "word" + i;                  // phase 2
}
```



| index | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| value | "word0" | "word1" | "word2" | "word3" |

# The `toString` method

CS 210

## reading: 8.2

# Any redundancies?

```
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point();
        p1.x = 5;
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;
        p2.x = 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.moveBy(2, 4);
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");
    }
}
```

**OUTPUT:**

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

# 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

- A method that tells Java how to convert an object into a string
  - Implicitly called when you use an object in a `String` context:

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

  - The above code is really doing the following:

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

- Every class has a `toString`, even if you don't define one.
  - Default: class's name `@` object's memory address

    ```
    Point@9e8c34
    ```

# toString syntax

```
public String toString() {
    <code that returns a String representation>;
}
```

- Method header must match exactly.

- Example:

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

# bomb **client, v3**

```
public class Bomb3 {
   ...

   public static void bombCities(Graphics g, Point3[] cities,
                       Point3 quarPoint, int quarRad) {
        g.setColor(Color.RED);
        System.out.println("Quarantined cities: ");

        g.drawOval(quarPoint.x - quarRad, quarPoint.y - quarRad,
                   2 * quarRad, 2 * quarRad);

        for (int i = 0; i < cities.length; i++) {
           if (quarPoint.distance(cities[i]) <= quarRad) {
              cities[i].draw(g);

              System.out.println("\t" + cities[i]);
           }
        }
     }
  }
```

# Object initialization: constructors

CS 210

## 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);    // better!
```

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

# Constructors

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

```
public <ClassName>(<parameters>) {
    <statement(s)>;
}
```

  - runs when the client uses the `new` keyword
  - How does this differ from other methods?
    - no return type is specified;
      it implicitly "returns" the new object being created
    - ***This is not the same as having a*** `void` ***return type!!***
  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to 0 (or equivalent).

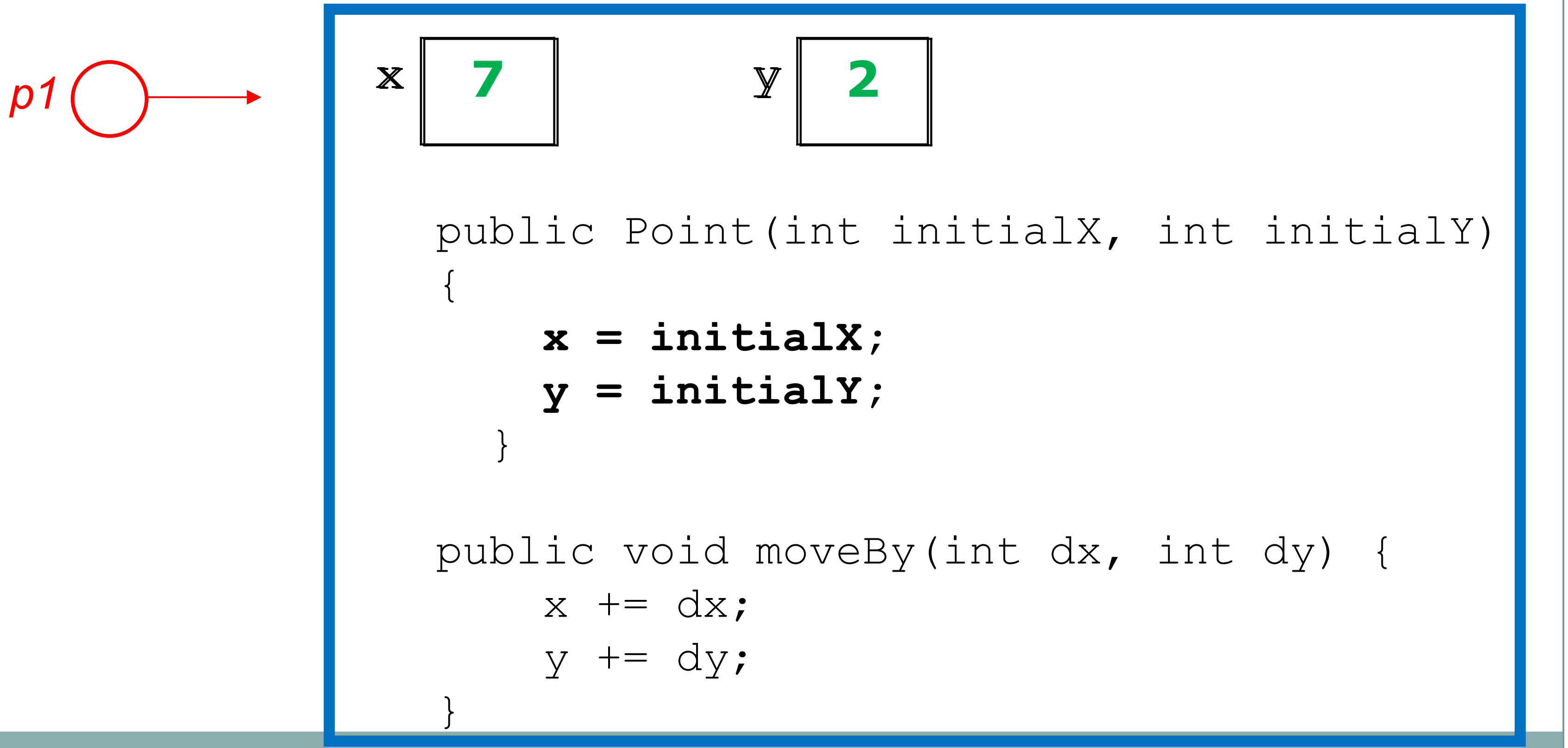
# Constructor example

```
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 moveBy(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 moveBy(int dx, int dy) {
     x += dx;
     y += dy;
}
```

# Common constructor bugs 1

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. Giving parameters the same name as fields:

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

- This is another form of shadowing.
  - We'll see how to fix this soon.

# Common constructor bugs 2

3. 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`

# 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;
}
```

# bomb client, v4

CS 210

```
public class Bomb4 {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("cities.txt"));
        Scanner console = new Scanner(System.in);

        // read in city locations
        int cityCount = input.nextInt();
        Point[] cities = new Point[cityCount];
        for (int i = 0; i < cityCount; i++) {
            cities[i] = new Point(input.nextInt(), input.nextInt());
        }

       ...

       // get bomb location/radius
        System.out.print("bomb site x and y? ");
        Point quarPoint = new Point(console.nextInt(), console.nextInt());

       ...
    }
}
```

# The keyword `this`

CS 210

## reading: 8.3

# 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.

# The `this` keyword

- **`this`** : Refers to the implicit parameter inside your class.
  *(a variable that stores the object on which a method is called)*
  - Refer to a field: `this.`***<field>***
  - Call a method:
    `this.`***<method>***`(`***<parameters>***`);`
  - One constructor can call another: `this(`***<parameters>***`);`

# Variable shadowing

- **shadowing**: 2 variables with same name in same scope.
  - Normally illegal, except when one variable is a field.

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

    ...

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

  - In most of the class, `x` and `y` refer to the fields.
  - In `setLocation`, `x` and `y` refer to the method's parameters.

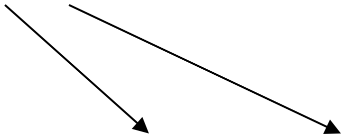
# Fixing shadowing

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

    ...

    public void setLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

- Inside `setLocation`,
  - To refer to the data field `x`, say `this.x`
  - To refer to the parameter `x`, say `x`

# Calling another constructor

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

    public Point() {
        this(0, 0);       // calls (x, y) constructor
    }


    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```



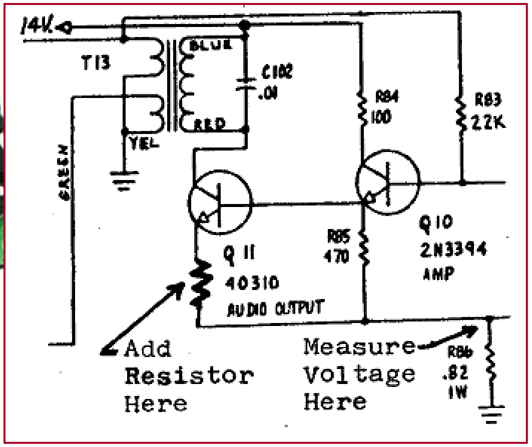
- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

# Encapsulation

CS 210

# Encapsulation

- **encapsulation**: Hiding implementation details from clients.
  - Encapsulation forces *abstraction*.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object's data



# Private fields

*A field that cannot be accessed from outside the class*

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

- Examples:

```
private int x;
private String y;
```

- Client code won't compile if it accesses private fields:

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

# Accessing private state

```
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

- Client code will look more like this:

```
System.out.println(p1.getX());
p1.setX(14);
```

# Point class

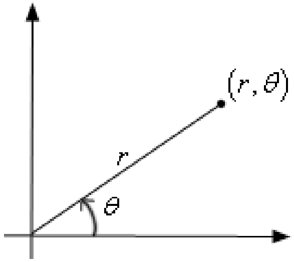
```
// A Point object represents an (x, y) location.
public class Point5 {
      private int x;
      private int y;

       // Constructs a Point at the given x/y location.
       public Point5(int x, int y) {
           this.x = x;
           this.y = y;
       }

       // Constructs a Point at the origin.
       public Point5() {
           this(0, 0);
       }

      public void setX(int x) {
           this.x = x;
       }

       public void setY(int y) {
           this.y = y;
       }

       public int getX() {
           return x;
       }

       public int getY() {
           return y;
       }

       ...
```

# Point class

```
  ...

  // Draw this Point.
  public void draw(Graphics g) {
    g.fillOval(x, y, 3, 3);
    g.drawString(toString(), x, y);
  }

  // Changes the location of this Point object.
  public void setLocation(int x, int y) {
    setX(x);
    setY(y);
  }

  // Shift this Point the given amounts.
  public void moveBy(int dx, int dy) {
    setLocation(x + dx, y + dy);
  }

  // Calculate the distance between this Point and another one.
  public double distance(Point5 other) {
    int dx = x - other.getX();
    int dy = y - other.getY();
    return Math.sqrt(dx * dx + dy * dy);
  }

  // Print a string representation of this Point
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

# Benefits of encapsulation

- Abstraction between object and clients
- Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.
- Can change the class implementation later
  - Example: `Point` could be rewritten in polar coordinates ($r$, $\theta$) with the same methods.



- Can constrain objects' state (**invariants**)
  - Example: Only allow `Account`s with non-negative balance.
  - Example: Only allow `Date`s with a month from 1-12.

Portions Copyright 2020 by Pearson Education

# The End

CS 210

## CHAPTER 8

## CLASSES

***Winnie Li***

# Static methods/fields

CS 210

# More about modules

- A module is a partial program, not a complete program.
  - It does not have a `main`. You don't run it directly.
  - Modules are meant to be utilized by other *client* classes.

- Syntax:

```
class.method(parameters);
```

- Example:

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

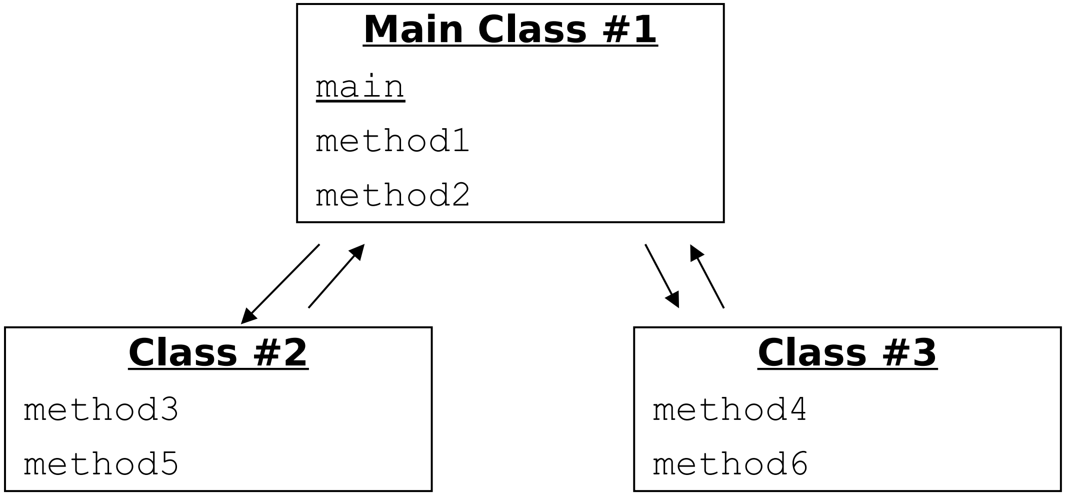
# Modules in Java libraries

```
// Java's built in Math class is a module
public class Math {
    public static final double PI = 3.14159265358979323846;

    ...

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

    public static double toDegrees(double radians) {
        return radians * 180 / PI;
    }
}
```

# Multi-class systems

- 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



# Redundant program 1

```java
// This program sees whether some interesting numbers are prime.
public class Primes1 {
    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 program 2

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

    // 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;
    }
}
```

# Classes as modules

- **module**: A reusable piece of software, stored as a class.
  - Example module classes: `Math`, `Arrays`, `System`

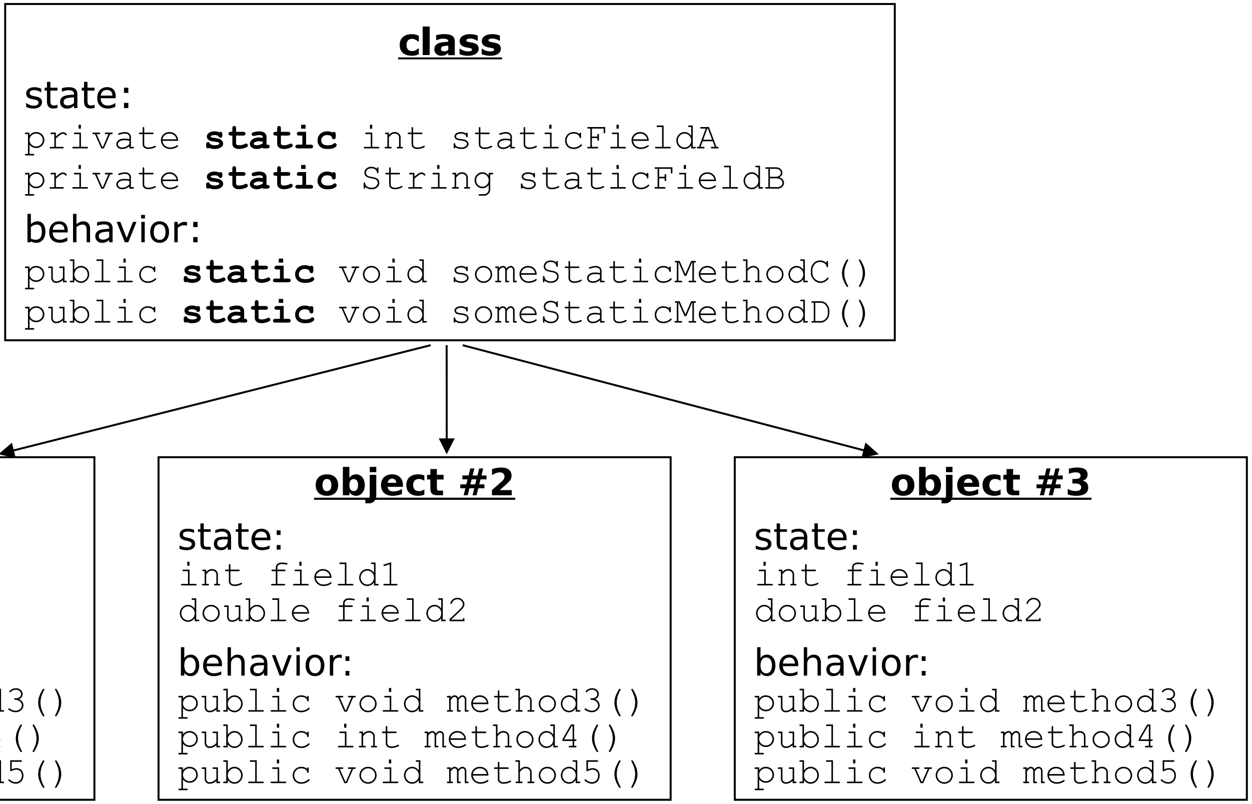
```
// This class is a module that contains useful methods
// related to factors and prime numbers.
public class Factors {
    // Returns the number of factors of the given integer.
    public static int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++) {
  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

# Using a 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 given 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();
    }
}
```

# Static members

- **static**: Part of a class, rather than part of an object.
  - Object classes can have static methods *and fields*.
  - Not copied into each object; shared by all objects of that class.



**class**

state:

```
private static int staticFieldA
private static String staticFieldB
```

behavior:

```
public static void someStaticMethodC()
public static void someStaticMethodD()
```

**object #1**

state:

```
int field2
double field2
```

behavior:

```
public void method3()
public int method4()
public void method5()
```

**object #2**

state:

```
int field1
double field2
```

behavior:

```
public void method3()
public int method4()
public void method5()
```

**object #3**

state:

```
int field1
double field2
```

behavior:

```
public void method3()
public int method4()
public void method5()
```

# Static fields

```
private static type name;
```

or,

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

- Example:

```
private static int theAnswer = 42;
```

- **static field**: Stored in the class instead of each object.
  - A "shared" global field that all objects can access and modify.
  - Like a class constant, except that its value can be changed.

# Accessing static fields

- From inside the class where the field was declared:

```
fieldName                       // get the value
fieldName = value;              // set the value
```

- From another class (if the field is `public`):

```
ClassName.fieldName             // get the value
ClassName.fieldName = value;    // set the value
```

  - generally static fields are not `public` unless they are `final`

- Exercise: Modify the `BankAccount` class shown previously so that each account is automatically given a unique ID.
- Exercise: Write the working version of `FratGuy`.

# BankAccount solution

```
public class BankAccount {

    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;        // advance the id, and
        id = objectCount;     // give number to account
    }

    ...

    public int getID() {    // return this account's id
        return id;
    }
}
```

# Static methods

```
// the same syntax you've already used for methods
public static type name(parameters) {
    statements;
}
```

- **static method**: Stored in a class, not in an object.
  - Shared by all objects of the class, not replicated.
  - Does not have any *implicit parameter*, `this`; therefore, cannot access any particular object's fields.
- Exercise: Make it so that clients can find out how many total `BankAccount` objects have ever been created.

# BankAccount solution

```java
public class BankAccount {
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // clients can call this to find out # accounts created
    public static int getNumAccounts() {
        return objectCount;
    }

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;     // advance the id, and
        id = objectCount;  // give number to account
    }

    ...

    public int getID() {   // return this account's id
        return id;
    }
}
```

# Summary of Java classes

- A class is used for any of the following in a large program:
  - a *program* : Has a main and perhaps other static methods.
    - example: `GuessingGame`, `Birthday`, `MadLibs`, `CritterMain`
    - does not usually declare any static fields (except `final`)
  - an *object class* : Defines a new type of objects.
    - example: `Point`, `BankAccount`, `Date`, `Critter`, `FratGuy`
    - declares object fields, constructor(s), and methods
    - might declare static fields or methods, but these are less of a focus
    - should be encapsulated (all fields and static fields `private`)
  - a *module* : Utility code implemented as static methods.
    - example: `Math`