Lecture 11: Intro to Classes

Building Java Programs: A Back to Basics Approach
by Stuart Reges and Marty Stepp

Classes and objects

- **class**: A program entity that represents either:
  1. A client/driver class: Has main method. Runs the program.
  2. Module: Utility class with useful methods, e.g. Math class.
  3. **Object class: A template for a new type of objects.**

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

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

Classes and objects

Example:

The Ipod **class** provides the template or blueprint for **state(data)** and **behavior(methods)** of an ipod **object.**

Its state or data can include the current song, current volume and battery life. Its behavior or methods can include change song, change volume, turn on/off, etc…

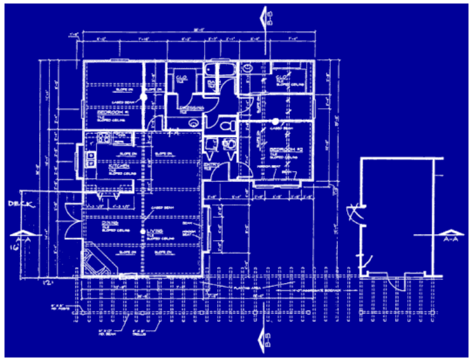
Two different ipod objects can have different states/data. However, their template share the same implementation/code.

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 = "Uptown Funk"
volume = 17
battery life = 2.5 hrs

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

iPod #2

state:
song = "You make me wanna"
volume = 9
battery life = 3.41 hrs

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

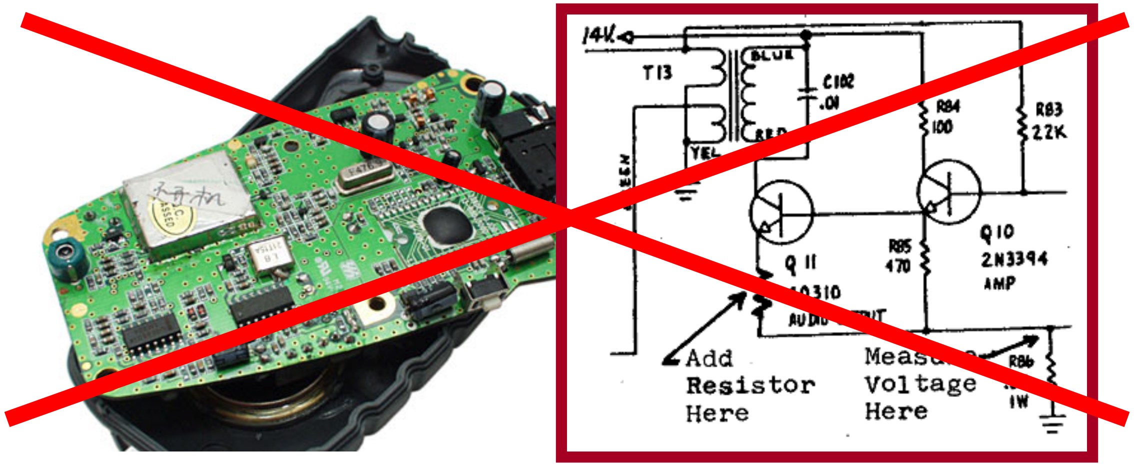
iPod #3

state:
song = "Trumpets"
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.



Abstraction

- **abstraction**: A distancing between ideas and details.
  - We can use objects without knowing how they work.

- abstraction in the String class:

  We don’t understand/see how the methods `substring`, `indexOf`, `length`, `toLowerCase`, etc… work but we still can use it.

More Examples

Suppose you are writing a RPG(role playing game). What are some classes and their corresponding objects?

Example:

The **Character** Class represents characters in the game.

State/Data: String name, int numberOfLives, boolean isAlien.

Behavior/Methods: shoot(), runLeft(), runRight(), jump().

Objects:

Character player1, player2; //declaring objects of type Character

Character enemy1, enemy2;

More Examples

Your game might have more than one classes.

Classes: Character, Boss, MysteryBox, Obstacle.

Objects:

```
Boss level1, level2;
MysteryBox yellow; // give player 3 extra lives
MysteryBox red; // give player 100 coins
Obstacle wall; //immovable
Obstacle poison; // kills player
```

Object state/data: Fields

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`.
    - int variables `x` and `y` are called the **fields** of the class Point
  - `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 a gpa field
}
```

Accessing fields

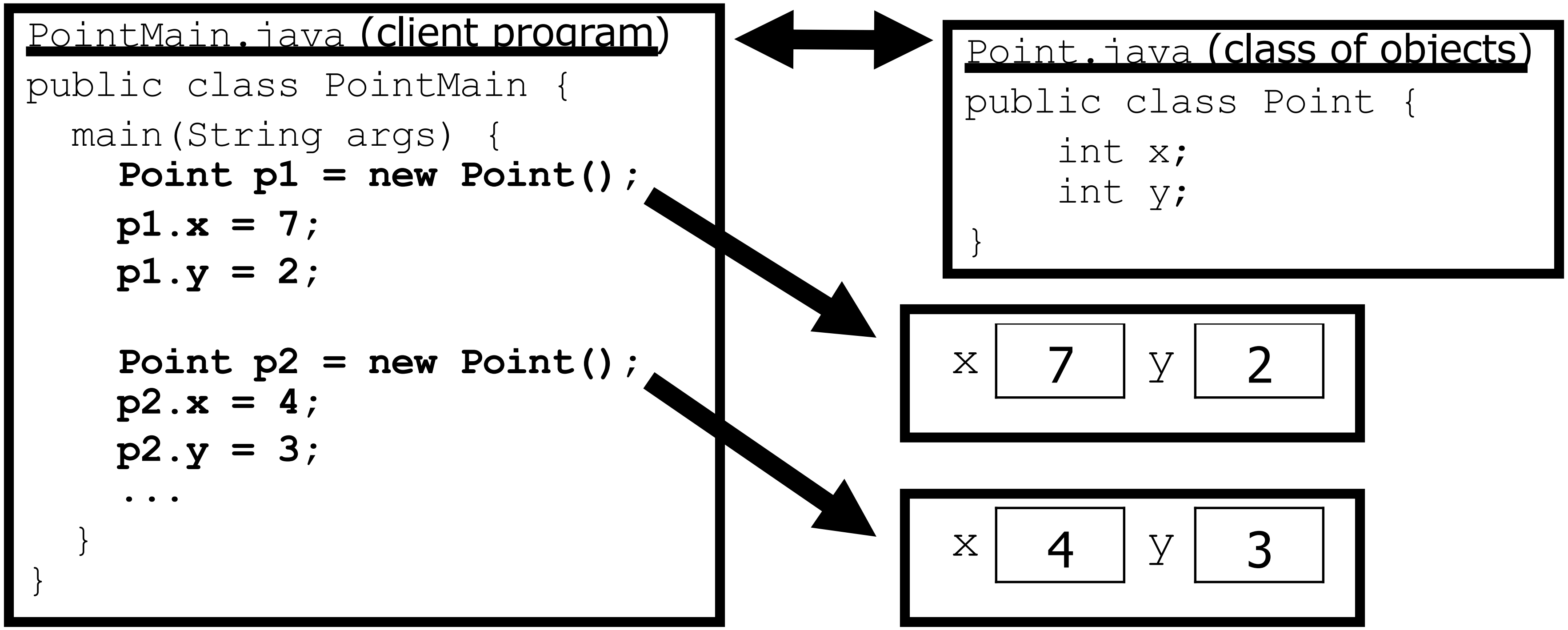
- Other classes can access/modify an object's fields.
  - access: **variable**.**field**
  - modify: **variable**.**field** = **value**;

- Example: (In class with main method)

```
public static void main(String[] args)
{
  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. It doesn't have a main method.
  - A class can be used by **driver/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;
}
```

PointMain client example

```
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point();
        p1.x = 0, p1.y = 2;
        Point p2 = new Point();
        p2.x = 3, p2.y = 7;
        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);    // 5, 8
    }
}
```

Object behavior: Methods

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 methods

We have used instance methods before.

```
 public class String{
  …
 public int length() {…}
 public String substring(int index){…}
 public int indexOf(String str){…}
}
```

Example: In our driver class.

```
String str=“This is a string.”, str2=“hello”;
System.out.println(str2.length()); //5
System.out.println(str.length()); //16
int index=str.indexOf(“is”); //2
```

Instance method example

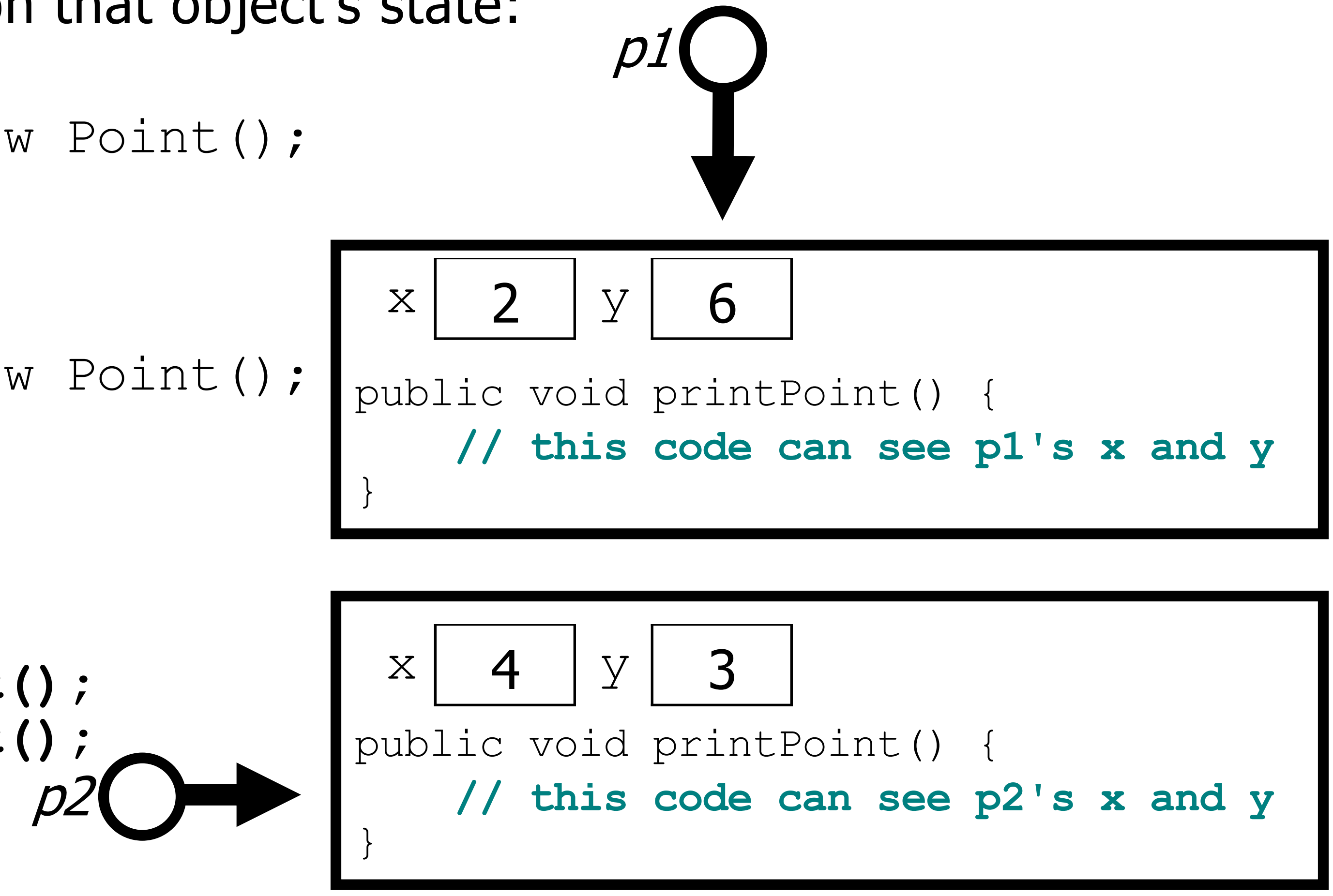
```
public class Point {
    int x;
    int y;

    // print this Point object.
    public void printPoint() {
        System.out.println("("+x+","+y+")");

    }
}
```

– How will the method know which point to print?
    - How will the method access that point's x/y data?

Implicit Parameter

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


p1
x 2 y 6
public void printPoint() {
// this code can see p1's x and y
}
p2
x 4 y 3
public void printPoint() {
// this code can see p2's x and 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`
  - often has a `void` return type

Mutator method questions

- Write a method `setLocation` that changes a `Point`'s location to the (*x*, *y*) values passed.

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

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

Mutator method answers

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

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

// alternative solution that utilizes setLocation
// public void translate(int dx, int dy) {
//     setLocation(x + dx, y + dy);
// }
}
```

Accessor method questions

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

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

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

  – Modify the client code to use these methods.

Method answers

```
public class Point{
  int x, y;
public void setLocation(int newX, int newY) {
    x = newX;
    y = newY;
}
public void translate(int dx, int dy) {
    x = x + dx;
    y = y + dy;
}
public double distance(Point other) {
    int dx = x - other.x;
    int dy = y - other.y;
    return Math.sqrt(dx * dx + dy * dy);
}
public double distanceFromOrigin() {
    return Math.sqrt(x * x + y * y);
}
```

Driver Class

```
public class DriverClass{
  public static void main(String[] args){
    Point p1 = new Point();
    p1.x = 7;
    p1.y = 2;
    p1.setLocation(4,-5); //x=4,y=-5
    p1.translate(-4,5); //x=0, y=0
    Point p2 = new Point();
    p2.x = 4;
    p2.y = 3;
    System.out.println(p2.distanceFromOrigin());
    //5.0
    System.out.println(p1.distance(p2)); //5.0
    p1.printPoint(); //(0,0)
    p2.printPoint(); //(4,3)
   }
}
```

Object Initialization: Constructors

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 this with most types of objects in Java.

Constructors

- **constructor**: Initializes the state of new objects. **Has the same name as the class.**

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

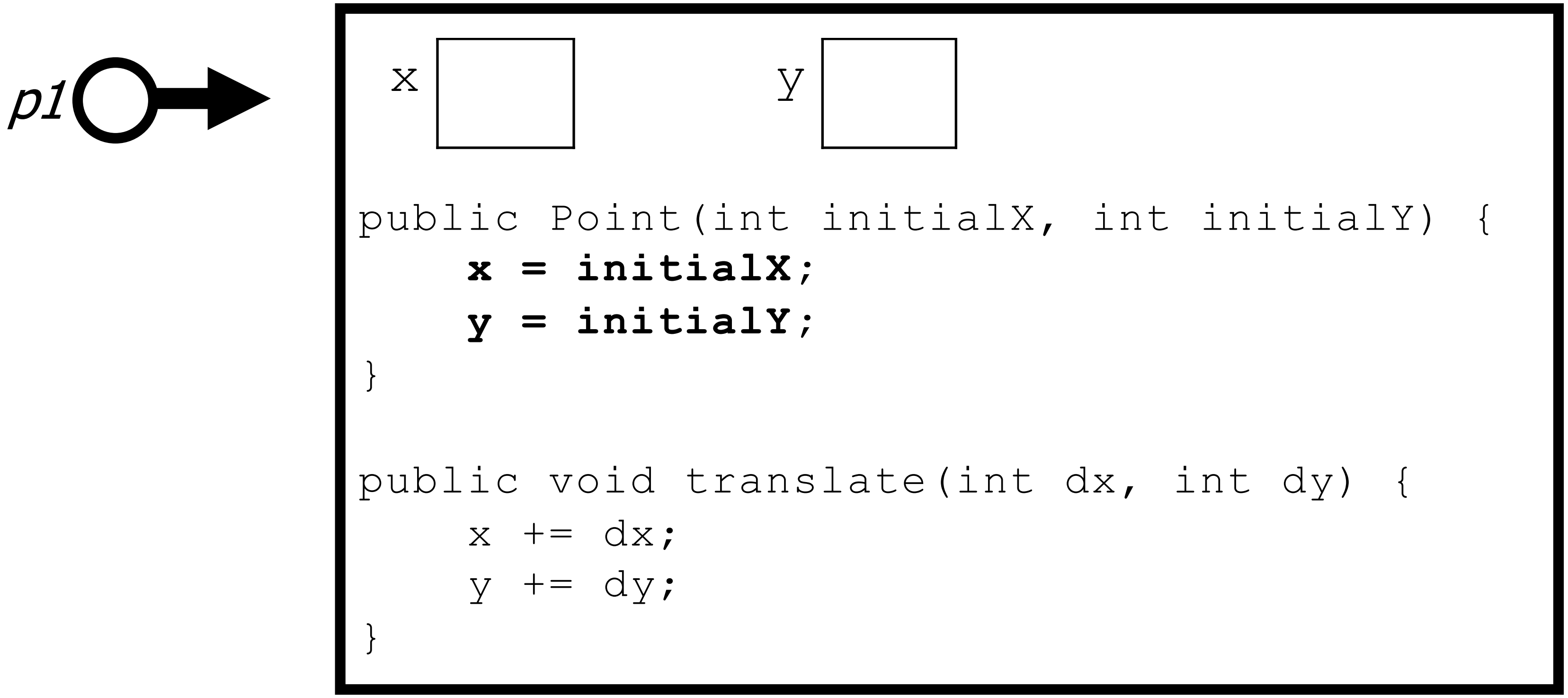
  - runs when the client uses the `new` keyword
  - no return type is specified;
    it implicitly "returns" the new object being created
  - If a class has no constructor, Java gives it a *default constructor* (one with no parameters) that sets all fields to 0. However, if a class has at least one constructor(with or without parameters), this default constructor is overridden by the new constructor(s).

```
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;
}
```

Client code

```
public class PointMain {
    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 an additional `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;
}
```

```
public class Point {
    int x;
    int y;
    // Constructs a Point at the origin.
    public Point(){
        x=0;
        y=0;
    }
    // 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;
    }
```

Client code

```
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point();

        // 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: (0, 0)
p2: (2, 4)
```

Common constructor bugs

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

```
public Point(int initialX, int initialY) {
    int x = initialX;
    int y = initialY;
    System.out.println(x);//prints the local x
}
```

   – 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`

Constructors

If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all integer fields to 0, booleans to false, Strings to **null**, etc...

However, if a class has at least one constructor(with or without parameters), this default constructor is overridden by the new constructor(s).

Default constructor

```
  public class Point {
      int x;
      int y;
      //no constructors
}

 public class PointMain {
   public static void main(String[] args) {
     Point p2 = new Point(); // ok, uses default constr
     Point p1 = new Point(5, 2); //error, no such constr

}
}
```

Overriding Default Constructors

```
  public class Point {
        int x;
        int y;
        //override default constructor
        public Point(int newX, int newY){
            x=newX;
            y=newY;
        }
}

 public class PointMain {
    public static void main(String[] args) {
        Point p2 = new Point(4,-2); // ok
        Point p1 = new Point(); //error, no default constr.

}
}
```

Lab 1

Rewrite the Point class with **all** of the methods in this lecture. Save it as Point.java. Then write a driver(different) class called PointTester with the main method. Create some point objects in the main method. Print out objects' data by accessing its fields and by calling its methods.

Remember that you must compile both classes separately but run only the driver class!!

Lab 1

Write the Circle class. This class has the following field variables(data/state): double x, double y, double radius, String color.

It has the following instance methods: getArea(), boolean isOnCircle(Point a), setColor(String str), translate(double dx, double dy), twiceRadius().

Use the same driver class from the previous lab to test the Circle class. Create multiple Circle objects and call all of the methods.

Notice that the same drive class can run and test both the Point and Circle classes.

Lab 1

Modify both the Point and Circle class to each include at least two constructors. At least one of the constructors should include enough parameters to initialize all of the fields.

In the driver class, create some Point and Circle objects by using different constructors.