Objects, Part 1

Slides based on slides by:

Mike Scott, used with permission, https://www.cs.utexas.edu/~scottm/cs312/

Chand John, used with permission, https://www.cs.utexas.edu/~chand/cs312/

Omar Ibrahim, CC BY-NC-SA 4.0, https://courses.cs.washington.edu/courses/cse142/21su/syllabus/

Larry Baker for IB Computer Science

Objects

- Group together related variables into an **object**
  - Like creating your own data structure out of Java building blocks

```
public class <object name> {
    <field(s)>;
}
```

- Syntax to use this data structure:

```
<object> <variable> = new <object>();
```

Objects

- Group together related variables into an **object**
  - Like creating your own data structure out of Java building blocks

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

- Syntax to use this data structure:

```
Point p1 = new Point();
```

Classes and Objects

- A *class* is a piece of the program’s source code.
  It can be either:
  - A program / module, or
  - A template for a particular type of object.

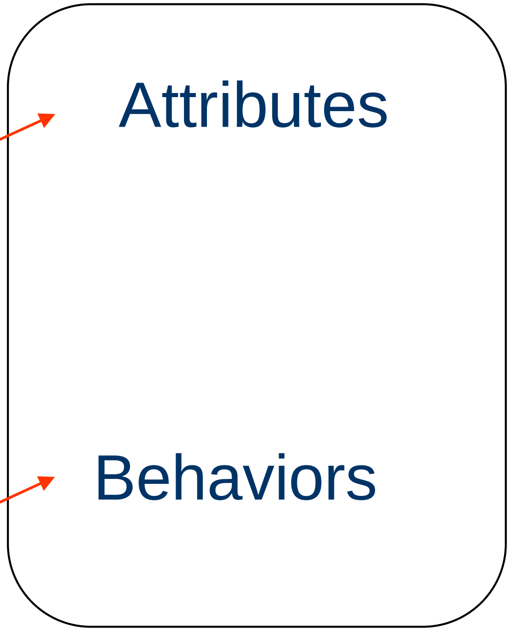
Classes and Objects

- A *class* is a piece of the program’s source code.
  It can be either:
  - A program / module, or
  - **A template for a particular type of object.**
    OO programmers call them “*class definitions*”.
- In case 2:
  An object is called an *instance* of a class.
  A program can create and use more than one object (instance) of the same class.

Class

- A blueprint for objects of a particular type
- Defines the structure (number, types) of the attributes
- Defines available behaviors of its objects

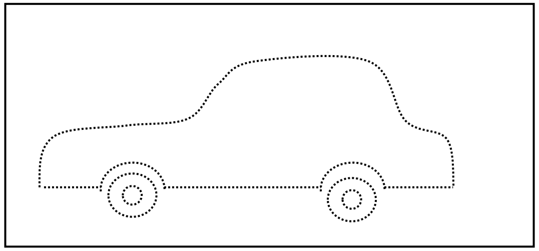
Object



Attributes

Behaviors

Class: Car



Attributes:

String model
Color color
int numPassengers
double amountOfGas

Behaviors:

Add/remove a passenger
Get the tank filled
Report when out of gas

Object: a car

Attributes:

model = "Mustang"
color = Color.YELLOW
numPassengers = 0
amountOfGas = 16.5

Behaviors:

Class vs. Object

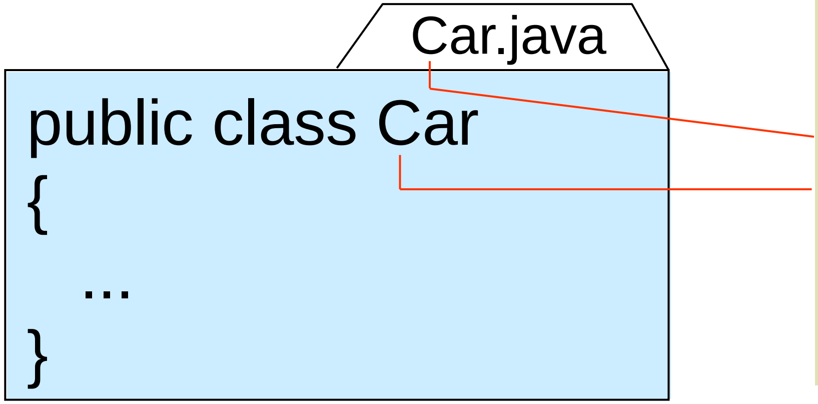
• A piece of the program’s source code	• An entity in a running program
• Written by a programmer	• Created when the program is running (by the main method or a constructor or another method)

Class vs. Object

• Specifies the structure (the number and types) of its objects’ attributes — the same for all of its objects	• Holds specific values of attributes; these values can change while the program is running
• Specifies the possible behaviors of its objects	• Behaves appropriately when called upon

Classes and Source Files

- Each class is stored in a separate file
- The name of the file must be the same as the name of the class, with the extension .java



```
public class Car
{
   ...
}
```

By convention, the name of a class (and its source file) always starts with a capital letter.

(In Java, all names are case-sensitive.)

Let's create an example class and see it's parts

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)
```

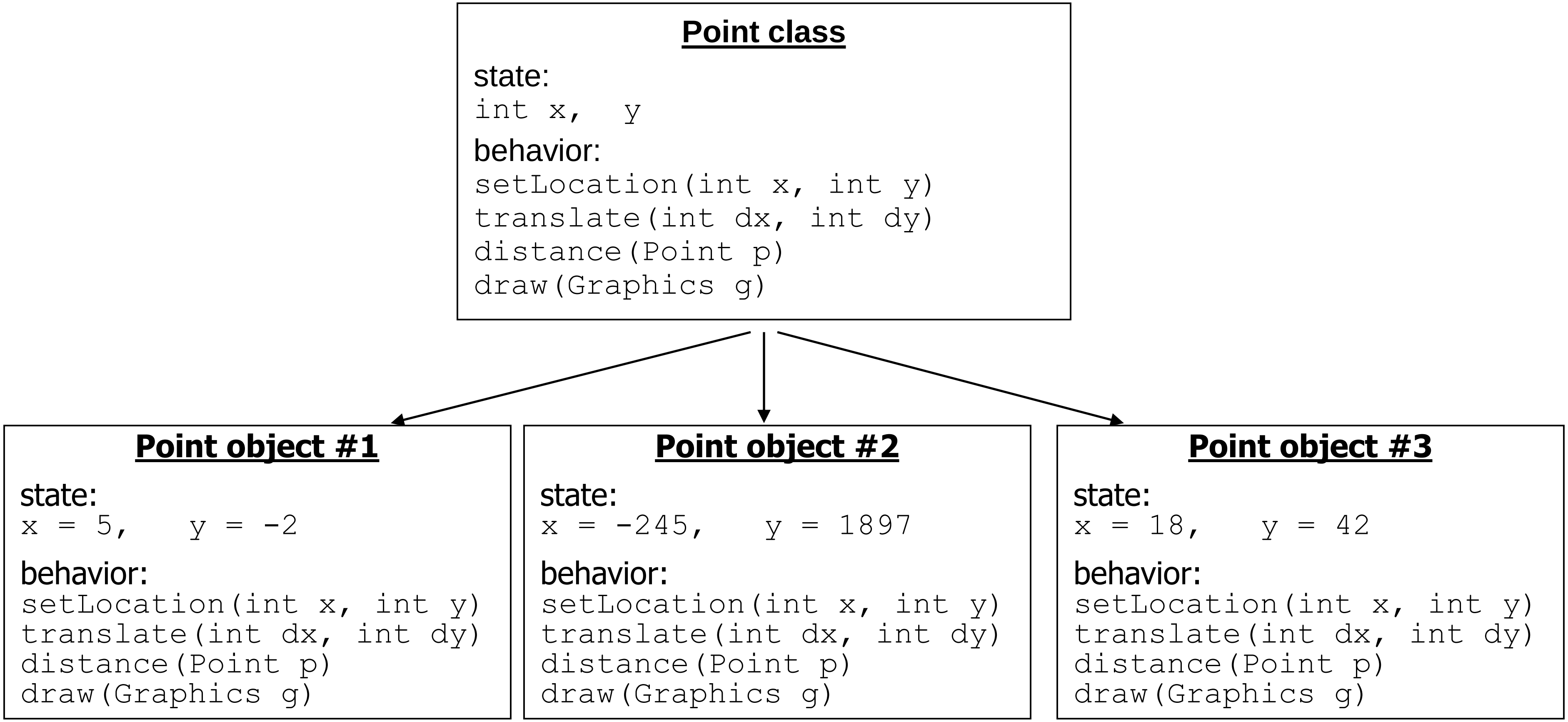
- Data in each Point object:

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



Point class

state:

```
int x,  y
```

behavior:

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

Point object #1

state:

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

behavior:

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

Point object #2

state:

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

behavior:

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

Point object #3

state:

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

behavior:

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

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

Point class, version 1

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

NO static !

- 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:

  access_modifier type name;

  - Example:

```
public class Student {
    // each Student object has a name and
    // gpa field (instance variable)
    private String name;
    private double gpa;
}
```

Accessing fields

- Other classes can access/modify an object's fields.
  - *depending on the access modifier*
  - 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

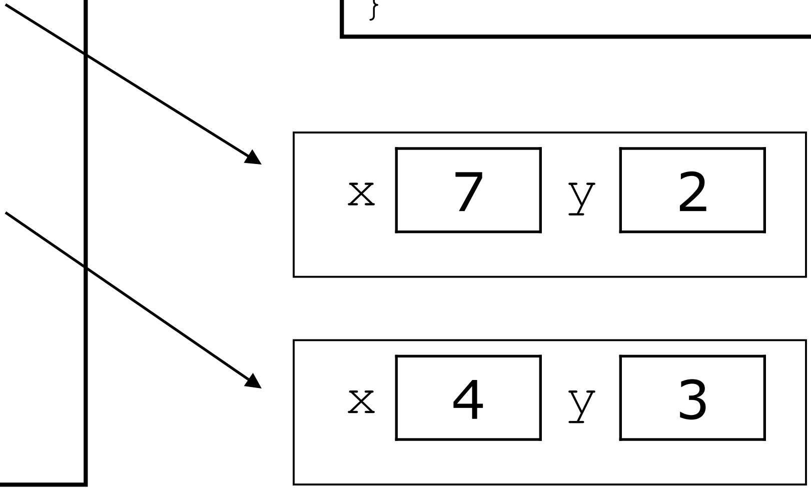
- `Point.java` is not, by itself, a runnable program.
  - A class can be used by **client** programs.

PointMain.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;
}
```



NO static !

Variables and without "static" are called "instance variables".
They belong to a specific object.

Variables with "static" belong to the ENTIRE class.

Instance variables can be different for each object.
Static variables are the same for all objects in the class.

Now, let's talk about methods

Behavior

- Objects can tie related data and *behavior* together

- **instance method:** A method inside an object that operates on that object

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

- Syntax to use method:

  <variable>.<method>(<parameter(s)>);

- Example:

```
p1.translate(11, 6);
```

Instance method example

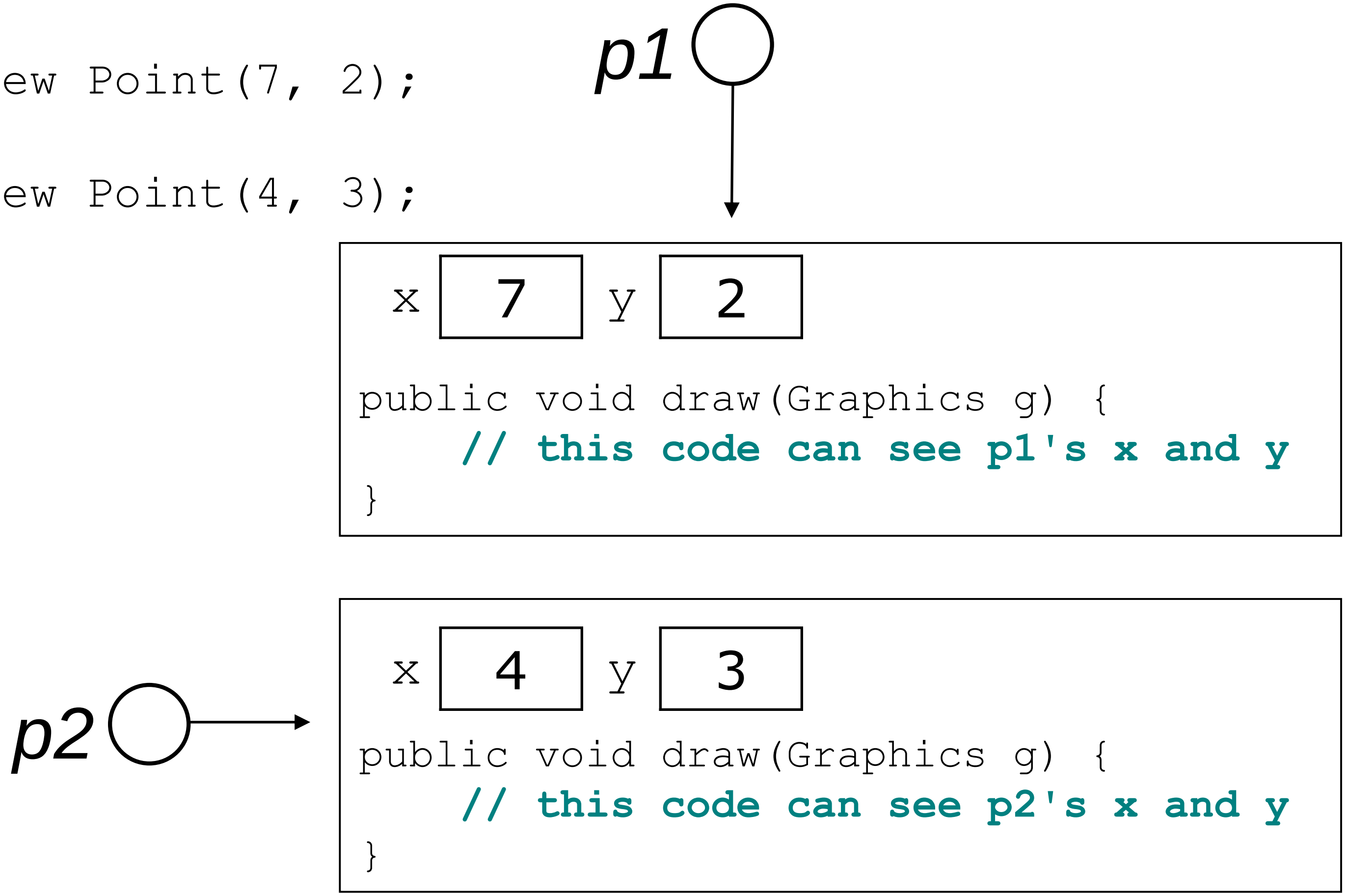
```
public class Point {
    private int x;
    private int y;

   // 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(7, 2);

Point p2 = new Point(4, 3);

p1.draw(g);
p2.draw(g);
```



```
x 7  y 2

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

```
x 4  y 3

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

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.

Not true for static methods !

static methods are the same for the all objects in the class.

non-static methods are called "instance methods".
Instance methods can see an object's instance variables - those are the "implied parameters".

```
public myMethod1 () {
  String value = stuff;   // stuff is an instance variable
}
```

Static methods cannot see instance variables unless you do extra code like:

```
public static myMethod2 (Object param) {
  String value = param.stuff;
}
```

When should you make a method static?
1. It is a "utility" method. It's just a nice tool relevant to the object.
1a. The method does not modify the state of an object.
OR
2. The method does not need to access any instance variables anyways.

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.

Now let's talk about privacy and "encapsulation"

Encapsulation and Information Hiding

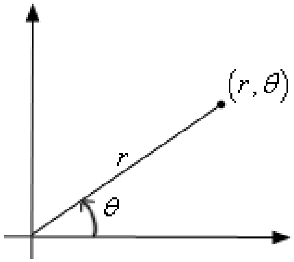
- A class interacts with other classes only through constructors and public methods
- Other classes do not need to know the mechanics (implementation details) of a class to use it effectively
- Encapsulation facilitates team work and program maintenance (making changes to the code)

Benefits of encapsulation

- Abstraction between object and clients

- Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.

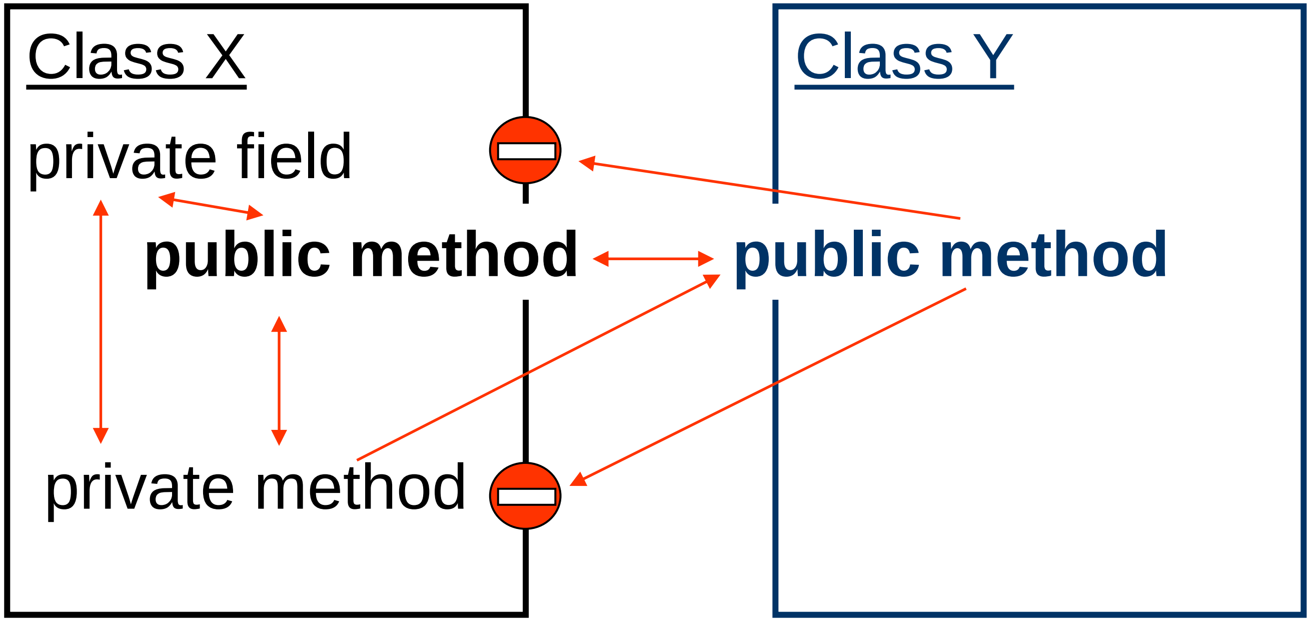
- Can change the class implementation later
  - Example: `Point` could be rewritten in polar coordinates (*r*, *θ*) with the same methods.



- 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.

Methods (cont'd)

- Constructors and methods can call other public and private methods of the same class.
- Constructors and methods can call only **public** methods of another class.



Private fields

A field that cannot be accessed from outside the class

```
private type name;
```

– Examples:

```
private int id;
private String name;
```

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

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

Accessors

- **accessor**: An instance method that provides information about the state of an object.

- Example:

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

- This gives clients "read-only" access to the object's fields.

Mutators

- **mutator**: An instance method that modifies the object’s internal state.

- Example:

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

- This gives clients both read and write access to code.

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, 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 int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

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

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

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

One more thing...

Initializing objects

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 pass the fields' initial values as parameters:

```
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.

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

  - runs when the client uses the `new` keyword
  - 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 fields to 0.

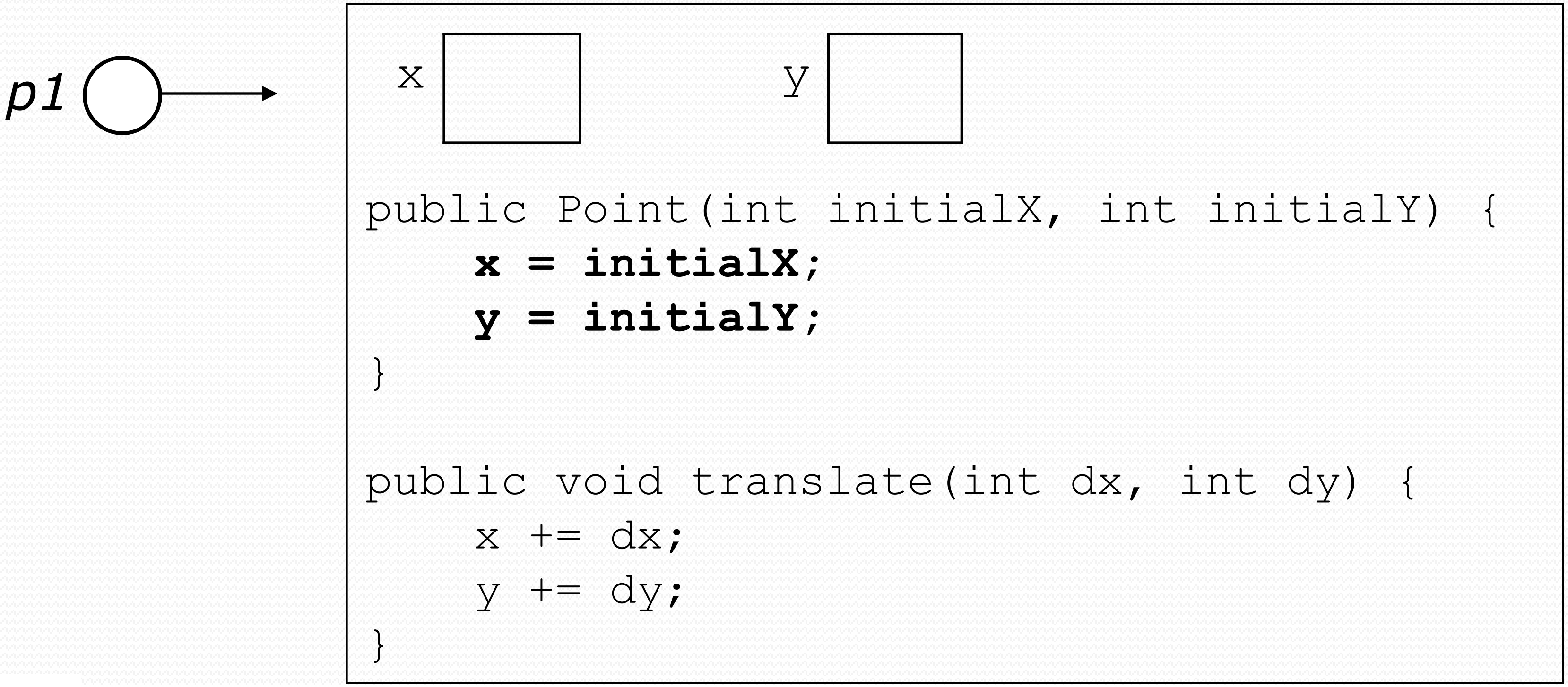
Constructor example

```
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 += dx;
        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, 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)
```

Common constructor bugs

- Accidentally writing a return type such as `void`:

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

  - This is not a constructor at all, but a method!

- Storing into local variables instead of fields ("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.

Multiple constructors

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

- Write a constructor for Point objects that accepts no parameters and initializes the point to the origin, (0, 0).

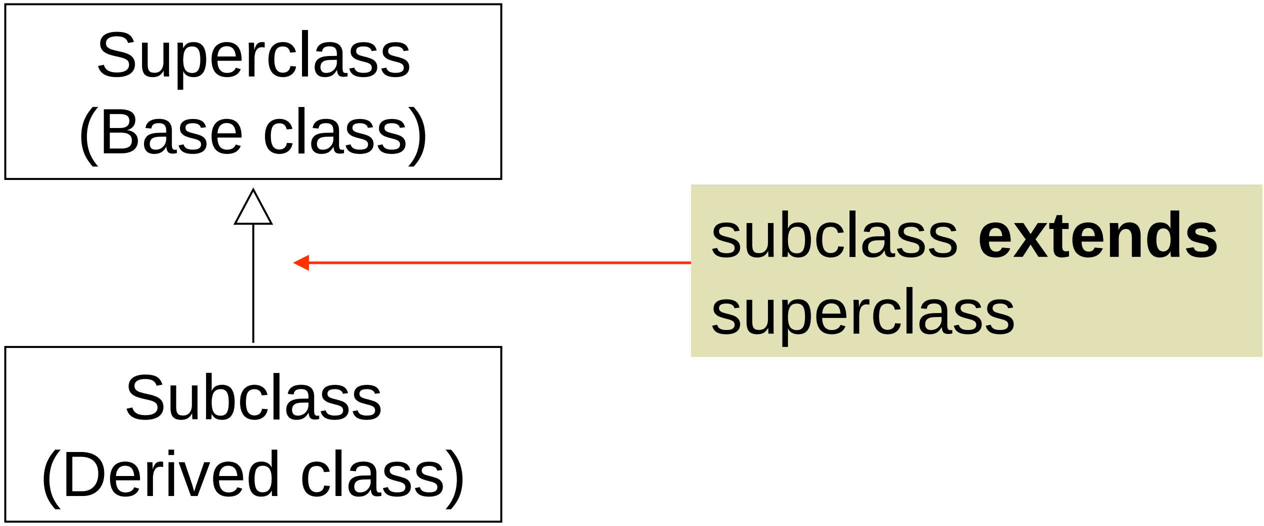
```
// Constructs a new point at (0, 0).
public Point() {
    x = 0;
    y = 0;
}
```

One LAST thing...

An OOP concept called "inheritance"

Inheritance

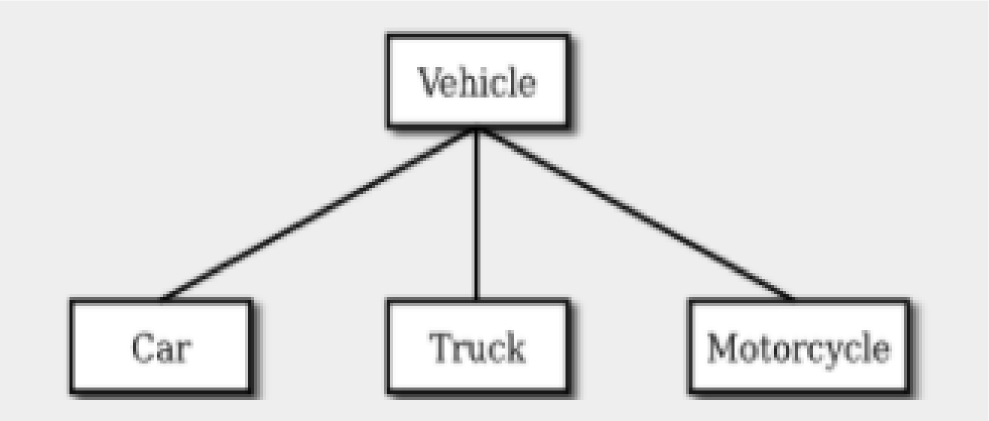
- In OOP a programmer can create a new class by extending an existing class



Superclass (Base class)

Subclass (Derived class)

subclass **extends** superclass


Vehicle
Car
Truck
Motorcycle

A Subclass...

- inherits fields and methods of its superclass
- can add new fields and methods
- can redefine (override) a method of the superclass
- must provide its own constructors, but calls superclass’s constructors
- does not have direct access to its superclass’s private fields