

# **Building Java Programs**

## **Chapter 8**

### Classes

Copyright (c) Pearson 2013.  
All rights reserved.

# 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 that lists all the cities that are within a given radius which are affected when a "bomb" is dropped :

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

# 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 `Points` to see whether the bomb hit a given city.
  - 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 `String` class is a template for creating `String` objects.
- **object:** An entity that combines state and behavior.
  - **object-oriented programming (OOP):** Programs that perform their behavior as interactions between objects.

# Blueprint analogy

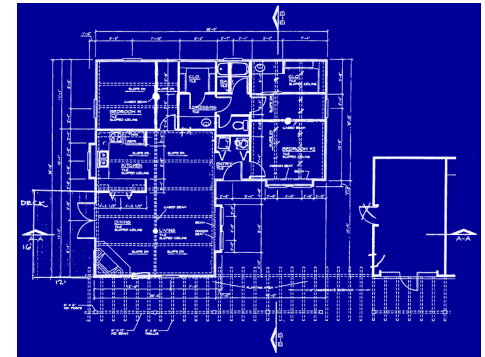
## iPod blueprint

### 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

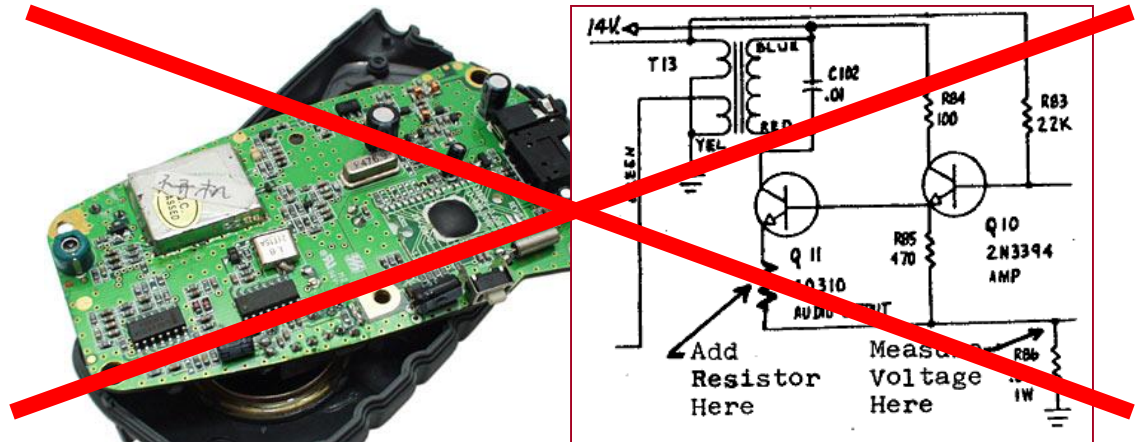
#### behavior:

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)
```

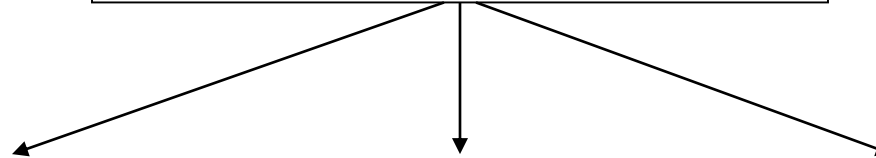
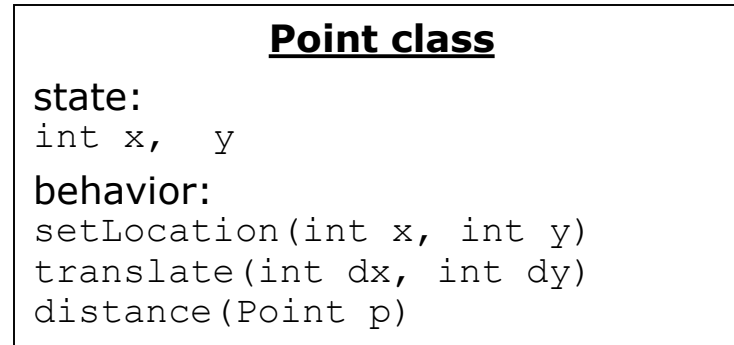
- Data in each `Point` object:

Field name	Description
<code>x</code>	the point's x-coordinate
<code>y</code>	the point's y-coordinate

- Methods in each `Point` object:

Method name	Description
<code>setLocation(<b>x</b>, <b>y</b>)</code>	sets the point's x and y to the given values
<code>translate(<b>dx</b>, <b>dy</b>)</code>	adjusts the point's x and y by the given amounts
<code>distance(<b>p</b>)</code>	how far away the point is from point <i>p</i>
<code>display()</code>	print the coordinates

# Point class as blueprint



**Point object #1**

```
state:
x = 5, y = -2
behavior:
setLocation(int x, int y)
translate(int dx, int dy)
distance(Point p)
display()
```

**Point object #2**

```
state:
x = -245, y = 1897
behavior:
setLocation(int x, int y)
translate(int dx, int dy)
distance(Point p)
display()
```

**Point object #3**

```
state:
x = 18, y = 42
behavior:
setLocation(int x, int y)
translate(int dx, int dy)
distance(Point p)
display()
```

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

# **Object state: 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`.
  - `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.

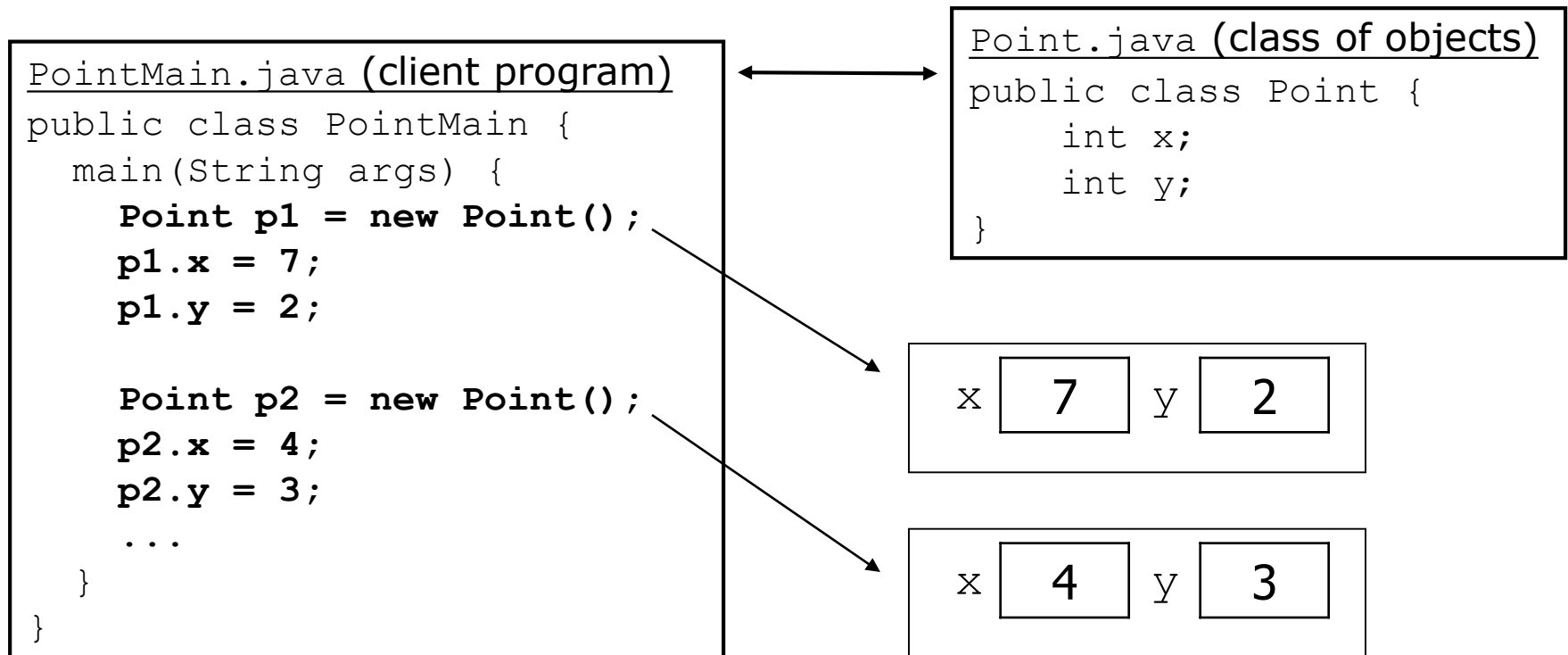
- access:           **variable.field**
- modify:           **variable.field = value;**

- 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

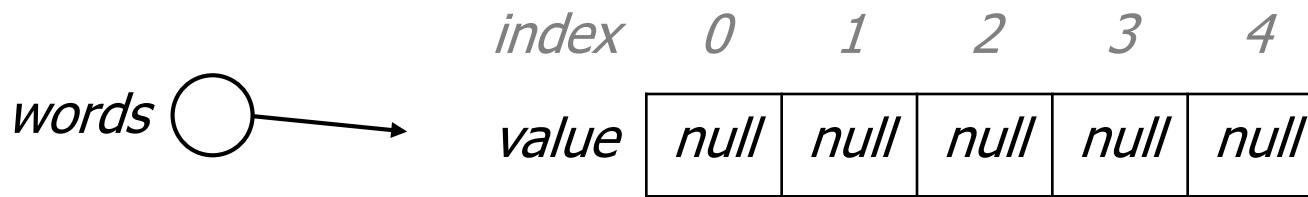
```
public class PointMain {  
    public static void main(String[] args) {  
        // create two Point objects  
        Point p1 = new Point();  
        p1.y = 2;  
        Point p2 = new Point();  
        p2.x = 4;  
  
        System.out.println(p1.x + ", " + p1.y);    // 0, 2  
  
        // move p2 and then print it  
        p2.x += 2;  
        p2.y++;  
        System.out.println(p2.x + ", " + p2.y);    // 6, 1  
    }  
}
```



# 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];
```



# 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 any object, so it has no methods or data.

```
String[] words = new String[5];  
System.out.println("word is: " + words[0]);  
words[0] = words[0].toUpperCase();    // ERROR
```

<i>index</i>	<i>0</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>
<i>value</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>

Output:

```
word is: null
```

```
Exception in thread "main"  
java.lang.NullPointerException  
    at Example.main(Example.java:8)
```

# Looking before you leap

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

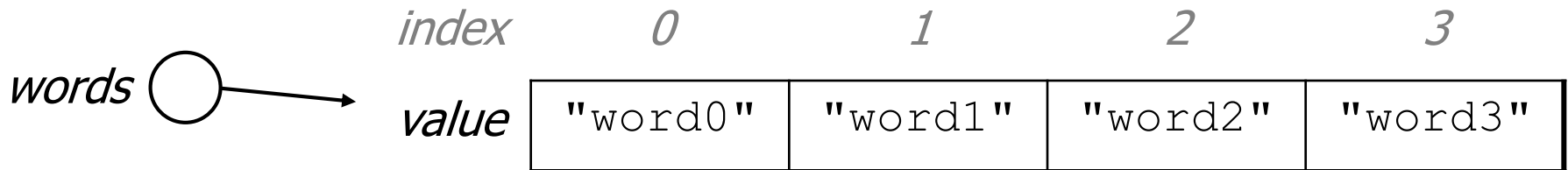
```
String[] words = new String[5];  
words[0] = "hello";  
words[2] = "goodbye";    // words[1], [3], [4] are null  
  
for (int i = 0; i < words.length; i++) {  
    if (words[i] != null) {  
        words[i] = words[i].toUpperCase();  
    }  
}
```



# 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++) {
    words[i] = "word" + i;                // phase 2
}
```



# Bomb answer 1

```
import java.io.*;
import java.util.*;

// Displays a set of cities and simulates dropping a "bomb" on them.
public class Bomb {
    public static void main(String[] args) throws FileNotFoundException {

        Scanner input = new Scanner(new File("cities.txt"));
        Point[] cities = readCities(input);

        // drop the "bomb"
        Scanner console = new Scanner(System.in);
        Point bomb = new Point();
        System.out.print("Blast site x? ");
        bomb.x = console.nextInt();
        System.out.print("Blast site y? ");
        bomb.y = console.nextInt();
        System.out.print("Blast radius? ");
        int radius = console.nextInt();
        boom(bomb, radius, cities);
    }
    ...
}
```

# Bomb answer 2

```
// Reads input file of cities and returns them as array of Points.
public static Point[] readCities(Scanner input) {
    int numCities = input.nextInt();    // first line = # of cities
    Point[] cities = new Point[numCities];
    for (int i = 0; i < cities.length; i++) {
        cities[i] = new Point();
        cities[i].x = input.nextInt();    // read city x/y from file
        cities[i].y = input.nextInt();
    }
    return cities;
}

// Simulates dropping a bomb at the given location on the given cities.
public static void boom(Point bomb, int radius, Point[] cities) {
    for (int i = 0; i < cities.length; i++) {
        int dx = cities[i].x - bomb.x;
        int dy = cities[i].y - bomb.y;
        double distance = Math.sqrt(dx * dx + dy * dy);
        if (distance <= radius) {
            System.out.println("(" + cities[i].x + ", " + cities[i].y + ")");
        }
    }
}
}
```

# **Object behavior: Methods**



# Client code redundancy

- Our client program wants to display `Point` objects:

```
// print each city
```

```
System.out.println("(" + p.x + ", " + p.y + ")");
```

- To print them in other places, the code must be repeated.
  - We can remove this redundancy using a method.

# Eliminating redundancy, v1

- We can eliminate the redundancy with a static method:

```
// Prints the give point on the screen
public static void display(Point p) {
    System.out.println ("(" + p.x + ", " + p.y + ")");
}
```

- `main` would call the method as follows:

```
// display each city
display(cities[i]);
```

# Problem with static method

- We are missing a major benefit of objects: code reuse.
  - Every program that displays `Points` would need a `display` method.
- The syntax doesn't match how we're used to using objects.

```
display(cities[i]);    // static (bad)
```

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

```
cities[i].display();    // inside 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;  
  
    // Print out the coordinates  
    public void display() {  
        ...  
    }  
}
```

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

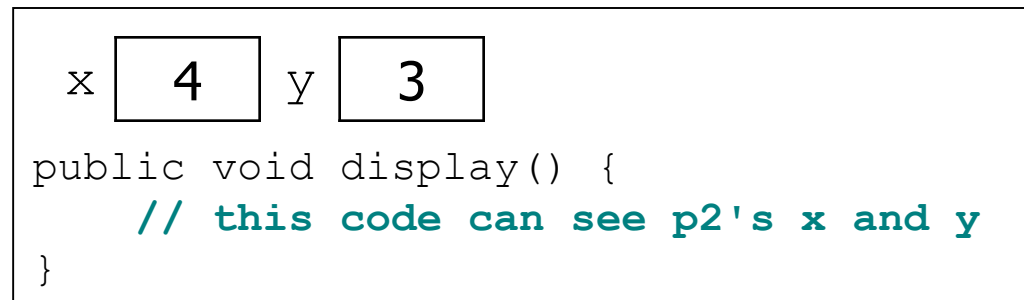
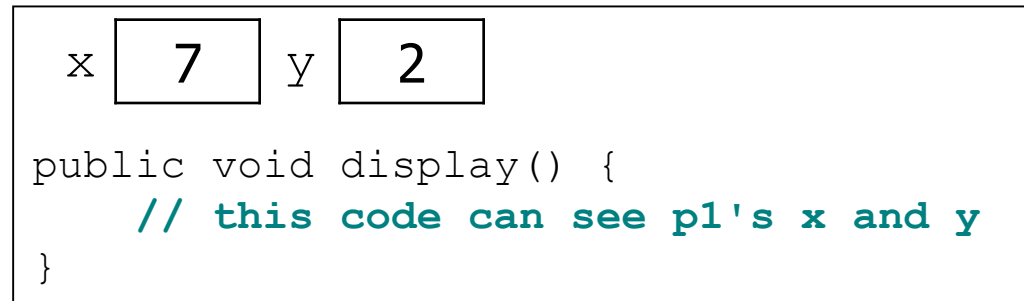
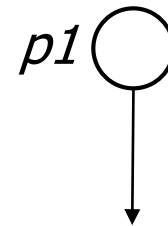
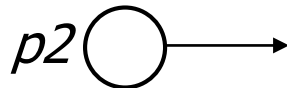
# Point objects w/ method

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



# The implicit parameter

- **implicit parameter:**

The object on which an instance method is called.

- During the call `p1.display()` ;  
the object referred to by `p1` is the implicit parameter.
- During the call `p2.display()` ;  
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.
  - `display` 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 display() {  
        System.out.println ("(" + x + ", " + y + ")");  
    }  
}
```

- Each `Point` object contains a `display` method that prints the `x/y` coordinates at that point.



# 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`

# 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 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.

# Accessor method answers

```
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);  
}
```

**// alternative solution that uses distance**

```
public double distanceFromOrigin() {  
    Point origin = new Point();  
    return distance(origin);  
}
```

# 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 + ")";  
}
```



# **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.

```
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.

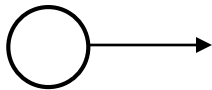
# Constructor example

```
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);
```

*p1* 

x  y 

```
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  
        p1.display();  
        p2.display();  
  
        // move p2 and then print it again  
        p2.translate(2, 4);  
        p2.display();  
    }  
}
```

OUTPUT:

```
(5, 2)  
(4, 3)  
(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;  
}
```

# 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;  
}
```

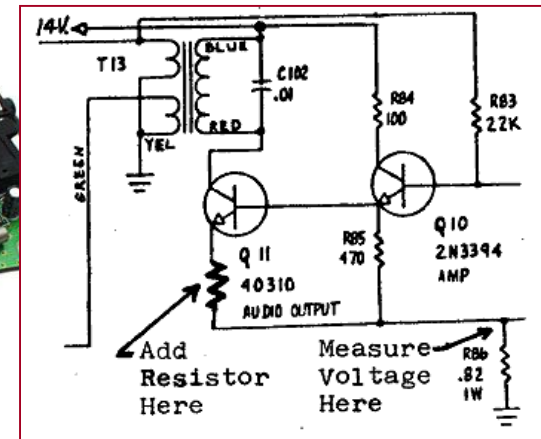
- This is actually not a constructor, but a method named `Point`



# Encapsulation

# 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 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) ;  
                        ^
```

# 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) ;
```

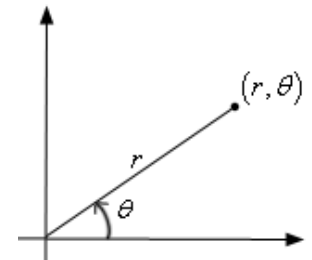
# Point class, version 4

// 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);  
    }  
}
```

# 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 `Accounts` with non-negative balance.
  - Example: Only allow `Dates` with a month from 1-12.



# 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 {  
    private int x;  
    private 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 {  
    private int x;  
    private 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 {  
    private int x;  
    private int y;  
  
    public Point() {  
        this(0, 0);           // calls (x, y) constructor  
    }  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    ...  
}
```

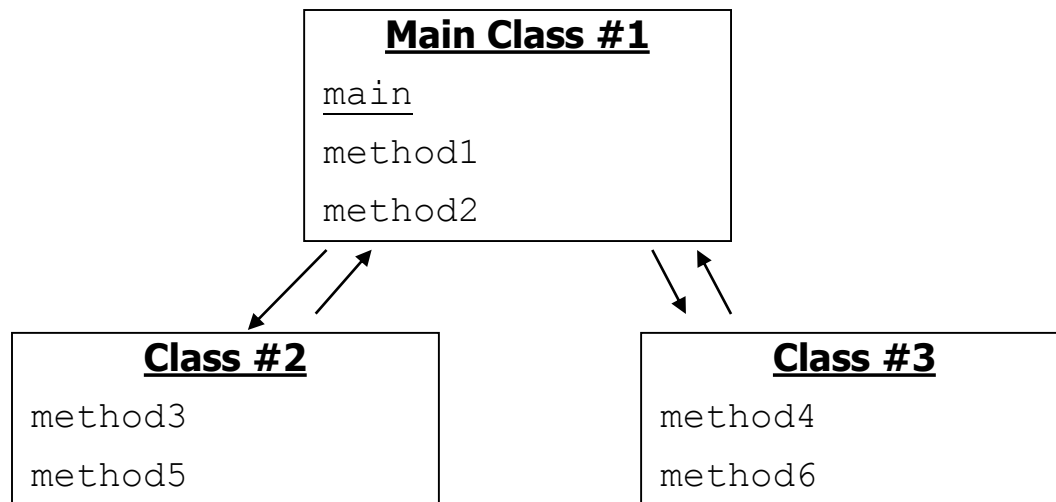


- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor

# **Static methods/fields**

# 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

**// 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

**// 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;
    }
}
```

# 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) ;
```



# 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();
    }
}
```

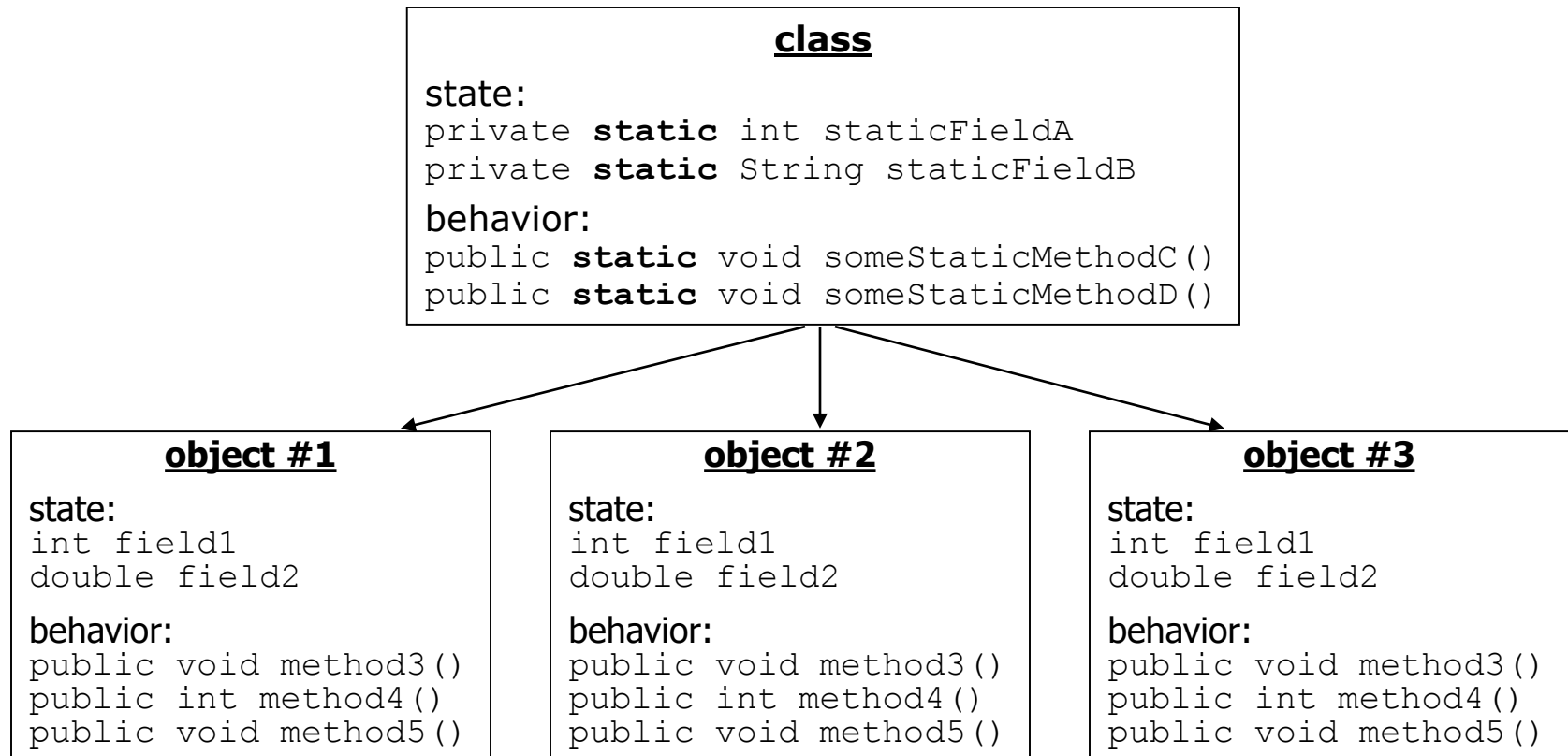
# Modules in Java libraries

**// 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;  
    }  
}
```

# 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.



# 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`
- Write a `BankAccount` class shown previously so that each account is automatically given a unique ID.

# 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

```
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`