AP CS:
Quick Review of Classes & Objects Vocabulary & Concepts

Subset of the Supplement Lesson slides from: Building Java Programs, Chapter 8.1 – 8.4
by Stuart Reges and Marty Stepp (http://www.buildingjavaprograms.com/) & thanks to Ms Martin.

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.

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 {
    ...
}
```

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

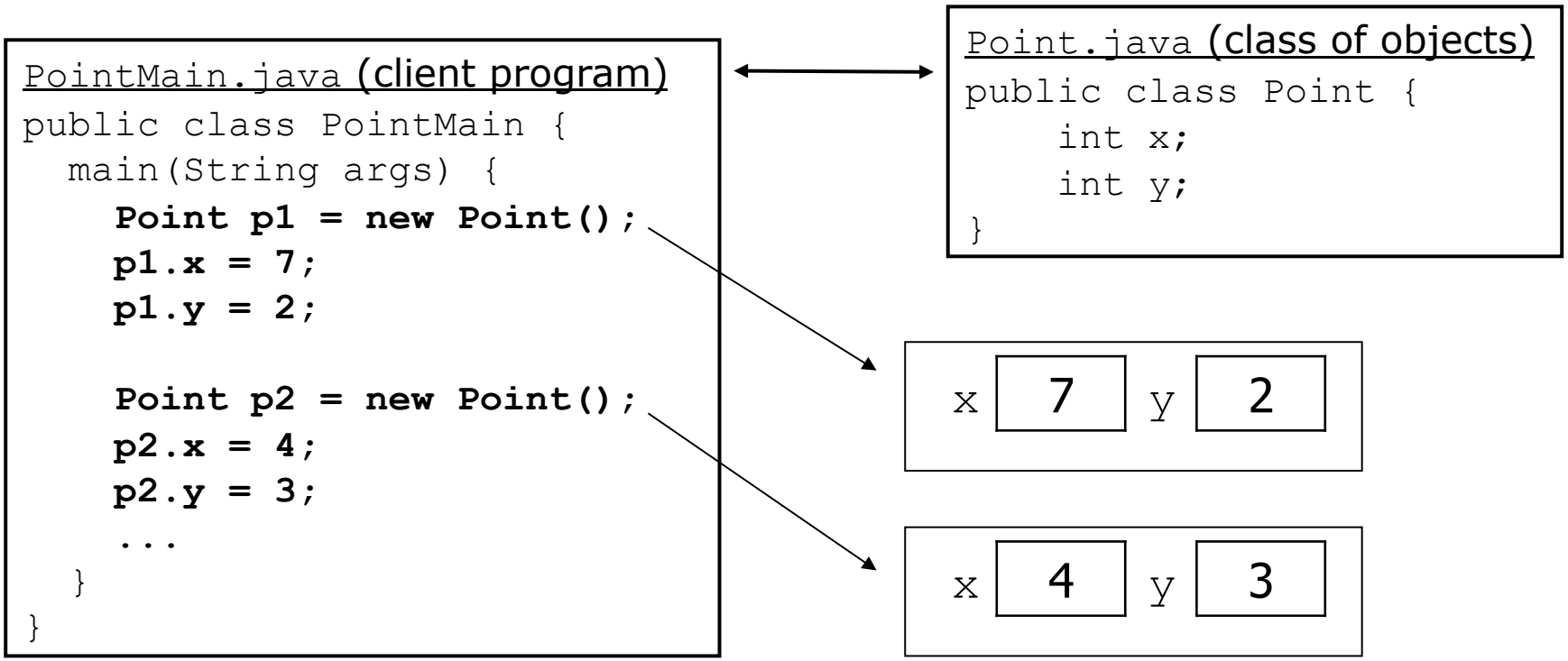
- Other classes can access/modify an object's fields.
  - access: **variable.field**
  - modify: **variable.field = value;**

- 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.



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:

```
// Changes the location of this Point object.
    public void draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, 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.
  - **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.

Kinds of Object methods

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

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

Examples

Write a **mutator** method setLocation that changes both coordinates of a Point's location to the given newx, newy values

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

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

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

Constructors

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

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

  - where the "type" is the Object's name
  - 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.

Multiple constructors

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

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

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

Array of Elements requires Two-phase initialization

- **Array of Objects**: you can create an array of any kind of objects, but the elements of an array of objects are initialized to `null`.
- **`null` :** A value that does not refer to any object (yet).

  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
}
```

- **dereference**: To access field data or methods of an object with the dot notation: such as a field: p1.x or a method: s.length()
  - It is illegal to dereference null (causes an exception).
  - null is not any object, so it has no methods or data.

The `toString` method

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

- Method name, return, and parameters must match exactly [these won't work: tostring() or ToString()].

- Syntax Sample:

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

- 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
```

Encapsulation: Private fields

Encapsulation: Hiding implementation details from clients using Private Fields...

Private Field: *A field that cannot be accessed from outside the class*

```
private type name;
```

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

- Client code won't compile if it accesses private fields but...
- An Object's method can return those private field values:

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

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.

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**`);`

"this" can *fix* (clarify) variable shadowing:

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