Advanced Programming Techniques in Java

COSI 12B

Object Oriented Programming II

Lecture 9

Class Objectives

- Constructor methods (Section 8.3)
- Add more behavior to Point (Section 8.2)
- equals()
- `this` keyword (Section 8.3)

Review: Object Behavior: Methods

- **Definition**
  - An **instance method** (or **object method**) is a method that exists inside each object of a class and gives behavior to each object
- **Syntax**

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

- **Example**

```
public void shout() {
     System.out.println("HELLO THERE!");
}
```

- Same syntax as static methods, but without `static` keyword

Review: The `toString` Method (cont.)

- **Syntax**

```
public String toString() {
        code that returns a String representing this object;
}
```

- **Example**

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

- Method name, return, and parameters must match exactly

Review: The `toString` Method Facts

- It is recommended to write a `toString()` method in every class you write
- Do not place `println` statements in the `toString()` method
  - `toString()` simply return a String that the client can use in a `println` statement
- Keep in mind that well formed classes of objects do not contain any `println` statement at all

Review: Constructor

- **Definition**
  - A **constructor** initialize the state of a new object
- **Syntax**

```
public <class name>(<type> <name>, …, <type> <name>) {
            statement(s);
}
```

- **Example**

```
//Constructs a new point with given location
public Point(int initialX, int initialY) {
            x = initialX;
            y = initialY;
}
```

Review: Constructor

- The constructor run 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 supplies a default constructor with no parameter
  - The default constructor initialize all fields to zero-equivalent values

```
public <class name>(<type> <name>, …, <type> <name>) {
            statement(s);
}
```

Point Class (ver. 4) with Constructor

Point.java

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

    // constructs a new point with the given (x, y) location
    public Point(int initialX, int initialY){
            x = initialX;
            y = initialY;
    }


    // shifts points location by the given amount
     public void translate (int dx, int dy){
            x += dx;
            y += dy;
     }

     // toString method
     public String toString(){
            return "(" + x + " , " + y + ")";
    }
}
```

same as the class's name

Constructors could also call class methods

Once you write your own constructor
Java will NOT supply the default one

PointMain.java (ver. 4)

PointMain.java

```
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 is "+ p1);
          System.out.println("p2 is "+ p2);

          //Translate each point to a new location

          p1.translate(11, 6);
          p2.translate(1, 7);

          //Print the points again
          System.out.println("p1 is "+ p1);
          System.out.println("p2 is "+ p2);
   }
}
```

PointMain.java (ver. 4)

PointMain.java

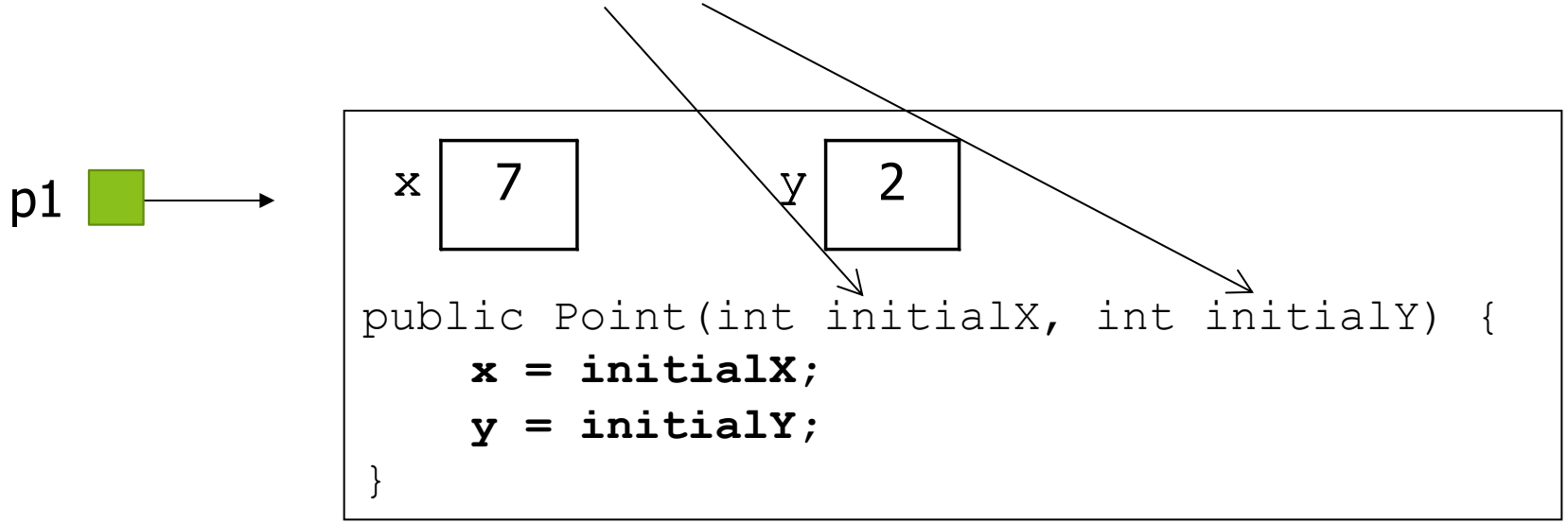
```
public class PointMain {
   public static void main(String[] args){
          //Create two Point objects
          Point p1 = new Point(5, 2);
          Point p2 = new Point(4, 3);

          //Print each point
          System.out.println("p1 is "+ p1);
          System.out.println("p2 is "+ p2);

          //Translate each point to a new location

          p1.translate(11, 6);
          p2.translate(1, 7);

          //Print the points again
          System.out.println("p1 is "+ p1);
          System.out.println("p2 is "+ p2);
   }
}
```

- Note: `Point p1 = new Point();` **`//will not compile for this version`**

Tracing a Constructor Call

- What happens when the following call is made?

```
Point p1 = new Point(7, 2);
```



Multiple Constructors

- A class can have multiple constructors to provide multiple ways for clients to construct objects of that class
  - Each constructor must accept a unique set of parameters (must have a different signature)
- Write a `Point` constructor with no parameters that initializes the point to (0, 0)

```
//Construct a Point at (0,0) location
public Point() {
        x = 0;
        y = 0;
}
```

```
//Create two Point objects
Point p1 = new Point(5, 2);
Point p2 = new Point();
```

Point Class (ver. 4)

Point.java

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

   // constructs a new point with the given (x, y) location
   public Point(int initialX, int initialY){
          x = initialX;
          y = initialY;
   }
   public Point(){
          x = 0;
          y = 0;
   }
    // shifts points location by the given amount
    public void translate (int dx, int dy){
          x += dx;
          y += dy;
    }
    // toString method
    public String toString(){
          return "(" + x + " , " + y + ")";
   }
}
```

Common Programming Bugs

- Using `void` with a constructor

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

- Constructors aren't supposed to have return types
- Tough to catch, because the `Point.java` file still compiles successfully

Common Programming Bugs

- Re-declaring fields in a constructor

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

- Behaves in an odd way
- It compiles successfully, but when the client code constructs a Point object its initial coordinates are always (0, 0)

Why?

Common Programming Bugs

- Re-declaring fields in a constructor

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

- Behaves in an odd way
- It compiles successfully, but when the client code constructs a Point object its initial coordinates are always (0, 0)

Why?

- We say that these local x and y variables **shadow** our `x` and `y` fields

Add more methods

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

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

- Write an alternative method `translate` that uses `setLocation`

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

Add more methods (cont.)

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

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

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

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

Mutators and Accessors

- **Definition** A **mutator** is an instance method that modifies the object's internal state
  - Examples: `setLocation`, `translate`
  - Has a void return type

- **Definition** An **accessor** is an instance method that provides information about the state of an object without modifying it
  - Examples: `distance`, `distanceFromOrigin`
  - Often has a non-void return type

Point Class (ver. 5)

Point.java

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

  // constructor
  public Point(int initialX, int initialY){
    x = initialX;
    y = initialY;
  }

  // constructor
  public Point(){
    x = 0;
    y = 0;
  }

  // shifts points location by the given amount
  public void translate (int dx, int dy){
    x += dx;
    y += dy;
  }

  // computes the distance between two points
  public double distance(Point other){
    int dx = x - other.x;
    int dy = y - other.y;
    return Math.sqrt(dx * dx + dy * dy);
  }
…
```

Point.java (cont.)

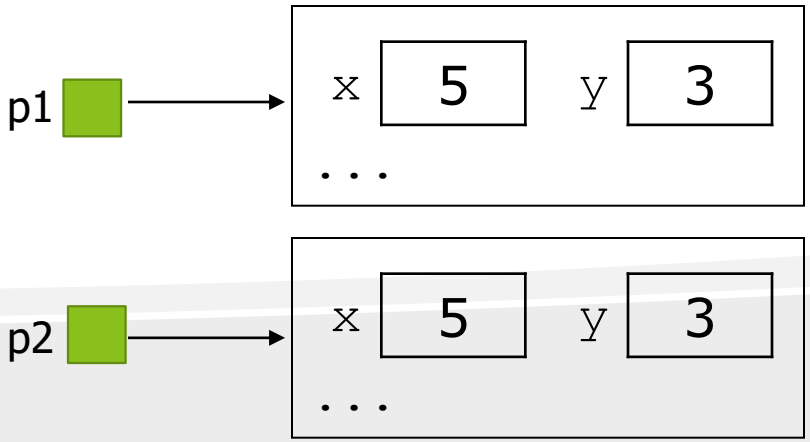
```
…
   // computes the distance between a point and the origin
  public double distanceFromOrigin() {
    Point origin = new Point();
    return distance(origin);
  }

  public String toString(){
    return "(" + x + " , " + y + ")";
  }
}
```

`equals()` Method

Comparing objects

- The == operator does not work well with objects
  - == compares references to objects and only evaluates to `true` if two variables refer to the same object (it doesn't tell us whether two objects have the same state)
- **Example**:

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
if (p1 == p2) {  // false
    System.out.println("equal");
}
```



The `equals` method

- The `equals` method compares the state of objects
  - When we write our own classes of objects, Java doesn't know how to compare their state
  - The default `equals` behavior acts just like the `==` operator

```
if (p1.equals(p2)) {   // still false
    System.out.println("equal");
}
```

- We can replace this default behavior by writing an `equals` method
  - The method will compare the state of the two objects and return true for cases like the above

Initial `equals` method

- This is one implementation of the equals method for the objects of the class Point

```
public boolean equals(Point other) {
    if (x == other.x && y == other.y) {
        return true;
    } else {
          return false;
    }
}
```

- Do we like this method?

Initial `equals` method

- This is one implementation of the equals method for the

```
public boolean equals(Point other) {
    if (x == other.x && y == other.y) {
        return true;
    } else {
        return false;
    }
}
```

- Do we like this method?

Initial flawed `equals` method

- You might think that the following is a valid implementation of the equals method:

```
public boolean equals(Point other) {
    if (x == other.x && y == other.y) {
        return true;
    } else {
        return false;
    }
}
```

- However, it has several flaws that we should correct
- One initial improvement: the body can be shortened

```
x == other.x && y == other.y;
```

```
public boolean equals(Point other) {
   return x == other.x && y == other.y;
}
```

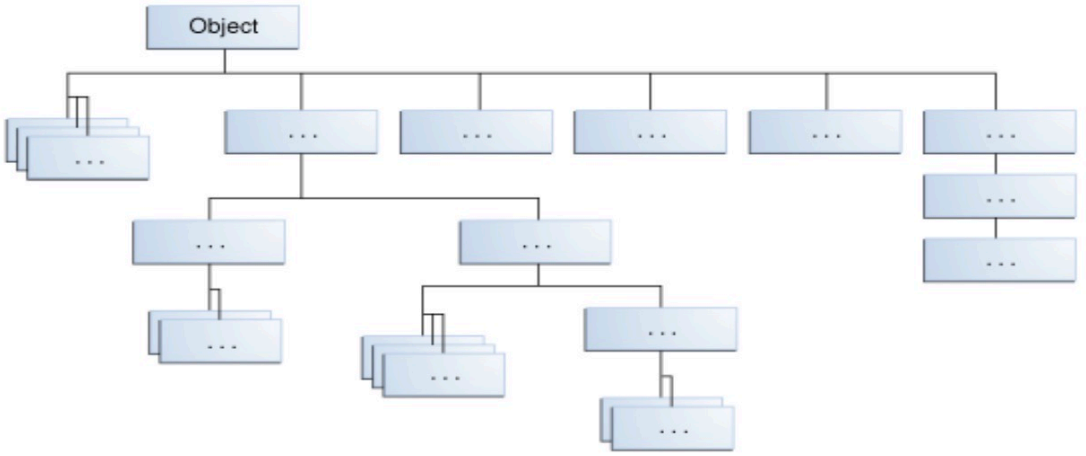
`equals` and the `Object` class

- The `equals` method should not accept a parameter of type `Point`
- It should be legal to compare `Points` to any other object, e.g.:

```
Point p = new Point(7, 2);
if (p.equals("hello")) {   // false
    ...
}
```

- The parameter to a proper equals method must be of type `Object` (meaning that an object of any type can be passed)

The Object class



- The Object class sits at the top of every class in the Java system
- It defines the basic state and behavior that all objects must have, such as the ability to compare oneself to another object, to convert to a string, etc.

* We will talk more about the Object class later

`equals` and the `Object` class

- **Syntax**:

```
public boolean equals(Object <name>) {
      <statement(s) that return a boolean value> ;
}
```

```
public boolean equals(Object o) {
   return x == o.x && y == o.y;
}
```

Another flawed version

- You might think that the following is a valid implementation of the `equals` method:

```
public boolean equals(Object o) {
     if (x == o.x && y == o.y) {
            return true;
     } else {
            return false;
     }
}
```

or

```
public boolean equals(Object o) {
   return x == o.x && y == o.y;
}
```

- However, it does not compile

```
Point.java:36: cannot find symbol
symbol  : variable x
location: class java.lang.Object
if (x == o.x && y == o.y) {
           ^
```

Type-casting objects

- The object that is passed to the `equals` method can be casted from `Object` into your class's type
- **Example**:

```
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

- Type-casting with objects behaves differently than casting primitive values
  - We are really casting a reference of type `Object` into a reference of type `Point`
  - We're promising the compiler that `o` refers to a `Point` object

Comparing different types

- Currently when we compare `Point` objects to any other type of objects

```
Point p = new Point(7, 2);
if (p.equals("hello")) {   // false
    ...
}
```

- The code crashes with the following exception:

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
at Point.equals(Point.java:25)
at PointMain.main(PointMain.java:25)
```

- The culprit is the following line that contains the type-cast:

```
public boolean equals(Object o) {
        Point other = (Point) o;
```

The `instanceof` keyword

- We can use a keyword called `instanceof` to ask whether a variable refers to an object of a given type
- **Syntax**: `<variable> instanceof <type>`
  - The above is a `boolean` expression that can be used as the test in an if statement
- **Example**:

```
String s = "hello";

Point p = new Point();
```

expression	result
`s instanceof Point`	`false`
`s instanceof String`	`true`
`p instanceof Point`	`true`
`p instanceof String`	`false`
`null instanceof String`	`false`

Final version of `equals` method

- This version of the `equals` method allows us to correctly compare `Point` objects against any other type of object:

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point object
public boolean equals(Object o) {
    if (o instanceof Point) {
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        return false;
    }
}
```

you still have to keep the casting

Template for your `equals()` methods

```
public boolean equals (Object o){
   if (o instanceof <type>){
          <type> other = (<type>) o;
          //compare the state and return the result
   }
   else {
          return false;
   }
}
```

`this` keyword

Remember … Common Programming Bug

- Re-declaring fields in a constructor

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

- Behaves in an odd way
- It compiles successfully, but when the client code constructs a Point object its initial coordinates are always (0, 0)

Why?

- We say that these local x and y variables **shadow** our `x` and `y` fields

Variable shadowing

- **Definition** **Shadowing** indicates two 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

- Use the keyword **`this`**

```
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`

The `this` keyword

- **Definition** The **`this`** keyword refers to the current object in a method or constructor
- The `this` keyword is used to eliminate confusion between class attributes and parameters with the same name
  - Refer to a field: `this.`**field**
  - Call a method: `this.`**method**(**parameters**);
  - One constructor can call another: `this`(**parameters**);
- So far, the compiler was converting expressions automatically
  - `x → this.x`
  - `setLocation(10,12) → this.setLocation(10,12)`

Programming style: shadowing is preferred

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

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

- Clearer style
- Matches client code that call methods as `object.method`
- You don't have to invent new variable names

The `this` keyword

- Using `this` with a constructor
  - From within a constructor, you can also use the `this` keyword to call another constructor in the same class

```
public class Point {
  int x;
  int y;
  public Point() {
        this(0, 0);     // calls (x, y) constructor
  }
  public Point(int x, int y) {
        setLocation(x,y);
  }
  ...
}
```

- Avoids redundancy between constructors
- Only a constructor (not a method) can call another constructor
- You cannot call `Point(0,0)`, it is illegal

Exercise

- Write a constructor that accepts a Point as a parameter and initializes this new Point to have the same (x,y) values

Exercise

- Write a constructor that accepts a Point as a parameter and initializes this new Point to have the same (x,y) values
- **Option 1**

```
public Point(Point p){
    //you have access to x, y directly within the class
    this.x = p.x;
    this.y = p.y;
}
```

- **Option 2, preferable**

```
public Point(Point p){
     this(p.x, p.y);
}
```