CENG 1004
Introduction to Object Oriented Programming

Spring 2018

WEEK 5

Case Study

- Design a drawing application with following functionalities
  - User can add a triangle by specifying 3 points
  - User can add a circle by specifying a point and radius
  - User can move any shape by specifying distance in x and y coordinates

Case Study

- Which classes can be defined in this application?
- Which properties can be defined in these classes?
- Which methods can be defined in these classes?

static fields and methods

static field

- Applies to fields and methods
- Means the field/method
  - Is defined for the class declaration,
  - Is not unique for each instance

static field

- Keep track of the number of points

```
public class Point {

    int xCoordinate;
    int yCoordinate;

    static int count = 0;

    public Point(int x, int y){
        xCoordinate = x;
        yCoordinate = y;
    }


    public void move(int xDistance, int yDistance){
        xCoordinate += xDistance;
        yCoordinate += yDistance;
    }
}
```

static field

- Keep track of the number of points

```
public class Point {

    int xCoordinate;
    int yCoordinate;

    static int count;

    public Point(int x, int y){
        xCoordinate = x;
        yCoordinate = y;
        count ++;
    }


    public void move(int xDistance, int yDistance){
        xCoordinate += xDistance;
        yCoordinate += yDistance;
    }
}
```

static field

```
public static void main(String[] args) {

        System.out.println(Point.count);

        Point point1 = new Point(10,10);
        Point point2 = new Point(15, 22);

        System.out.println(Point.count);

}
```

static field

```
public class Counter {
   int myCount = 0;
   static int ourCount = 0;
   void increment() {
      myCount++;
      ourCount++;
   }
   public static void main(String[] args) {
      Counter counter1 = new Counter();
      Counter counter2 = new Counter();
      counter1.increment();
      counter1.increment();
      counter2.increment();
      System.out.println("Counter 1: " +
counter1.myCount + " " + counter1.ourCount);
      System.out.println("Counter 2: " +
counter2.myCount + " " + counter2.ourCount);
   }
}
```

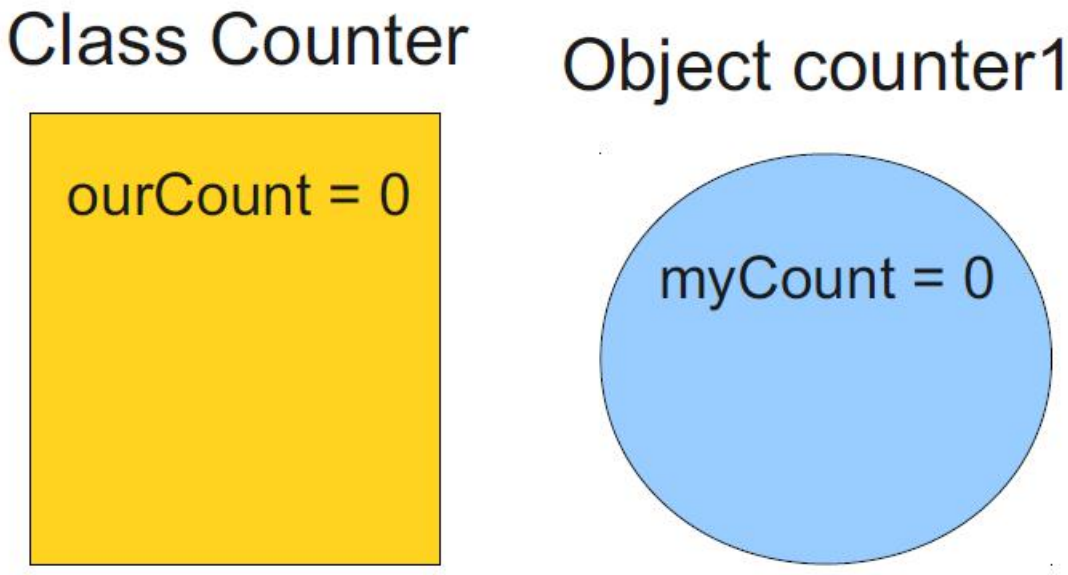
static field

```
public class Counter {
   int myCount = 0;
   static int ourCount = 0;
   void increment() {
      myCount++;
      ourCount++;
   }
   public static void main(String[] args) {
      Counter counter1 = new Counter();
      Counter counter2 = new Counter();
      counter1.increment();
      counter1.increment();
      counter2.increment();
      System.out.println("Counter 1: " +
counter1.myCount + " " + counter1.ourCount);
      System.out.println("Counter 2: " +
counter2.myCount + " " + counter2.ourCount);
   }
}
```

Fields

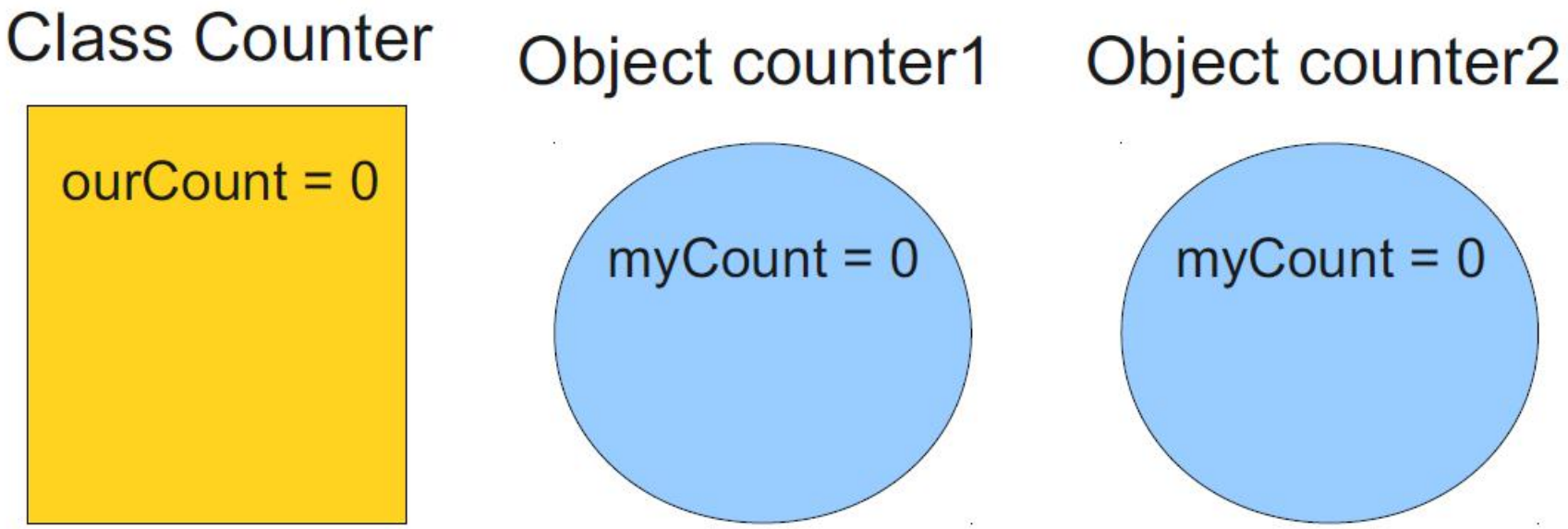
Method

static field



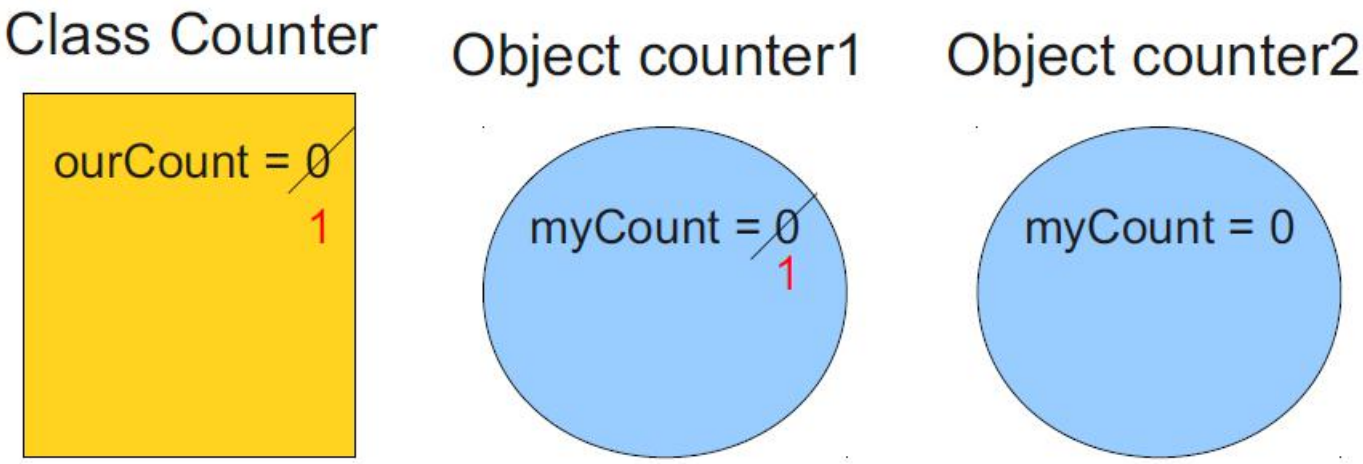

```
Counter counter1 = new Counter();
```

static field



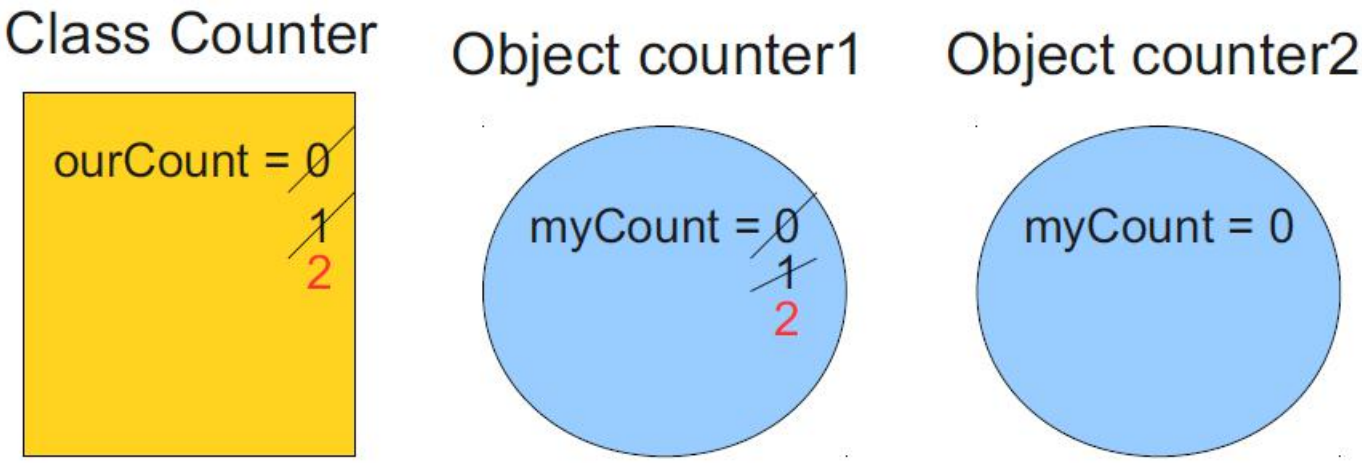

```
Counter counter1 = new Counter();
Counter counter2 = new Counter();
```

static field



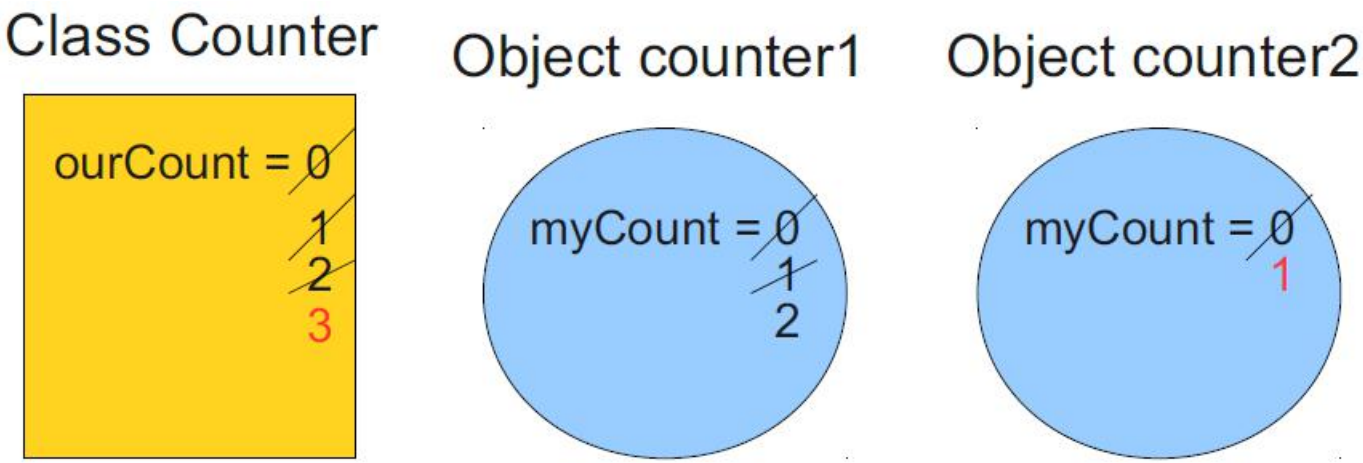

```
Counter counter1 = new Counter();
Counter counter2 = new Counter();
counter1.increment();
```

static field




```
Counter counter1 = new Counter();
Counter counter2 = new Counter();
counter1.increment();
counter1.increment();
```

static field




```
Counter counter1 = new Counter();
Counter counter2 = new Counter();
counter1.increment();
counter1.increment();
counter2.increment();
```

Access control

Access Control

```
public class CreditCard {
  String cardNumber;
  double expenses;
  void charge(double amount) {
    expenses = expenses + amount;
  }
  String getCardNumber(String password) {
    if (password.equals("SECRET!3*!")) {
      return cardNumber;
    }
      return "jerkface";
  }
}
```

Malicious

```
public class Malicious {
   public static void main(String[] args) {
      maliciousMethod(new CreditCard());
   }
   static void maliciousMethod(CreditCard card)
   {
      card.expenses = 0;
      System.out.println(card.cardNumber);
   }
}
```

Public vs. Private

- Public: others can use this
- Private: only the class can use this

public/private applies to any
field or **method**

Access Control

```
public class CreditCard {
  String cardNumber;
  double expenses;
  void charge(double amount) {
    expenses = expenses + amount;
  }
  String getCardNumber(String password) {
    if (password.equals("SECRET!3*!")) {
      return cardNumber;
    }
      return "jerkface";
  }
}
```

Access Control DONE

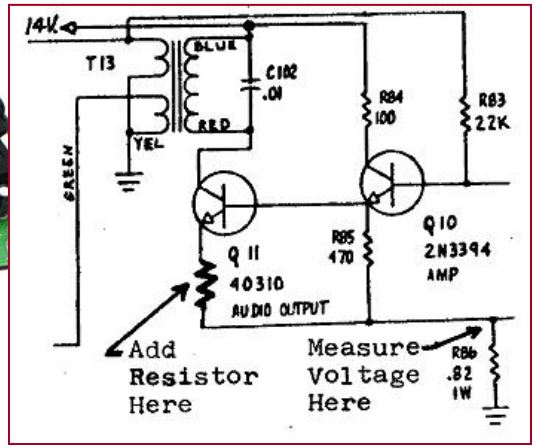
```
public class CreditCard {
   private String cardNumber;
   private double expenses;
   public void charge(double amount) {
      expenses = expenses + amount;
   }
   public String getCardNumber(String password)
   {
      if (password.equals("SECRET!3*!")) {
         return cardNumber;
      }
      return "jerkface";
   }
}
```

Why Access Control

- Protect private information
- Clarify how others should use your class
- Keep implementation seperate from interface

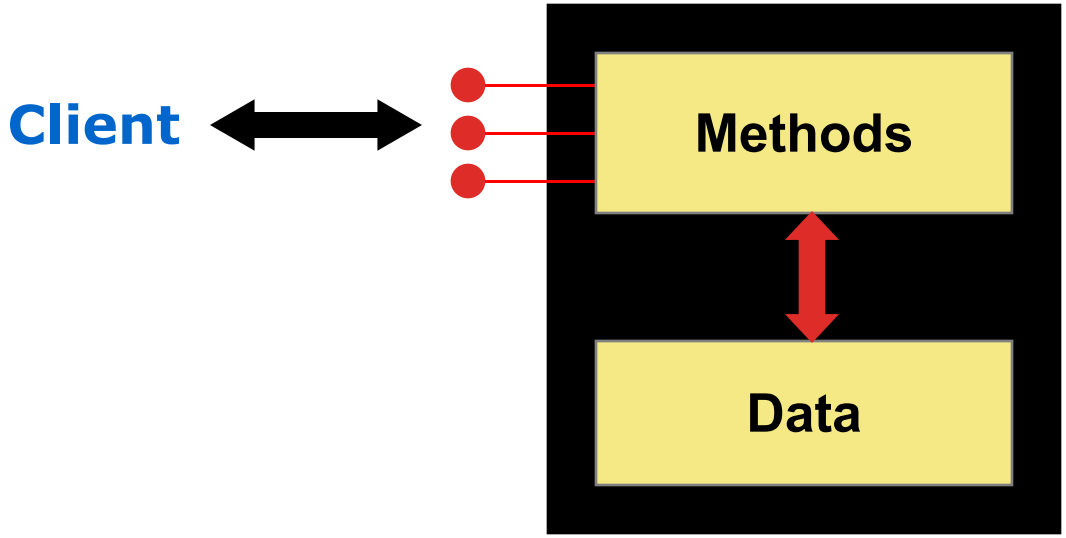
Encapsulation

- **encapsulation**: Hiding implementation details of an object from its clients.
  - Encapsulation provides *abstraction*.
    - separates external view (behavior) from internal view (state)
  - Encapsulation protects the integrity of an object's data.



Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data



Private fields

- A field can be declared *private*.
  - No code outside the class can access or change it.

```
private type name;
```

  - Examples:

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

- Client code sees an error when accessing private fields:

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

Accessing private state

- We can provide methods to get and/or set a field's value:

```
// 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;
}
```

Point class

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

    public int getX() {
        return x;
    }

    public int getY() {
        return 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;
    }
}
```

Client code

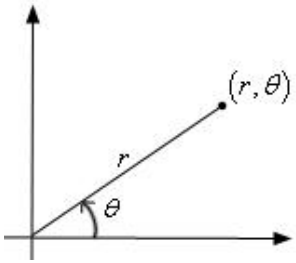
```
public class PointMain {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");
  }
}
```

OUTPUT:

```
p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)
```

Benefits of encapsulation

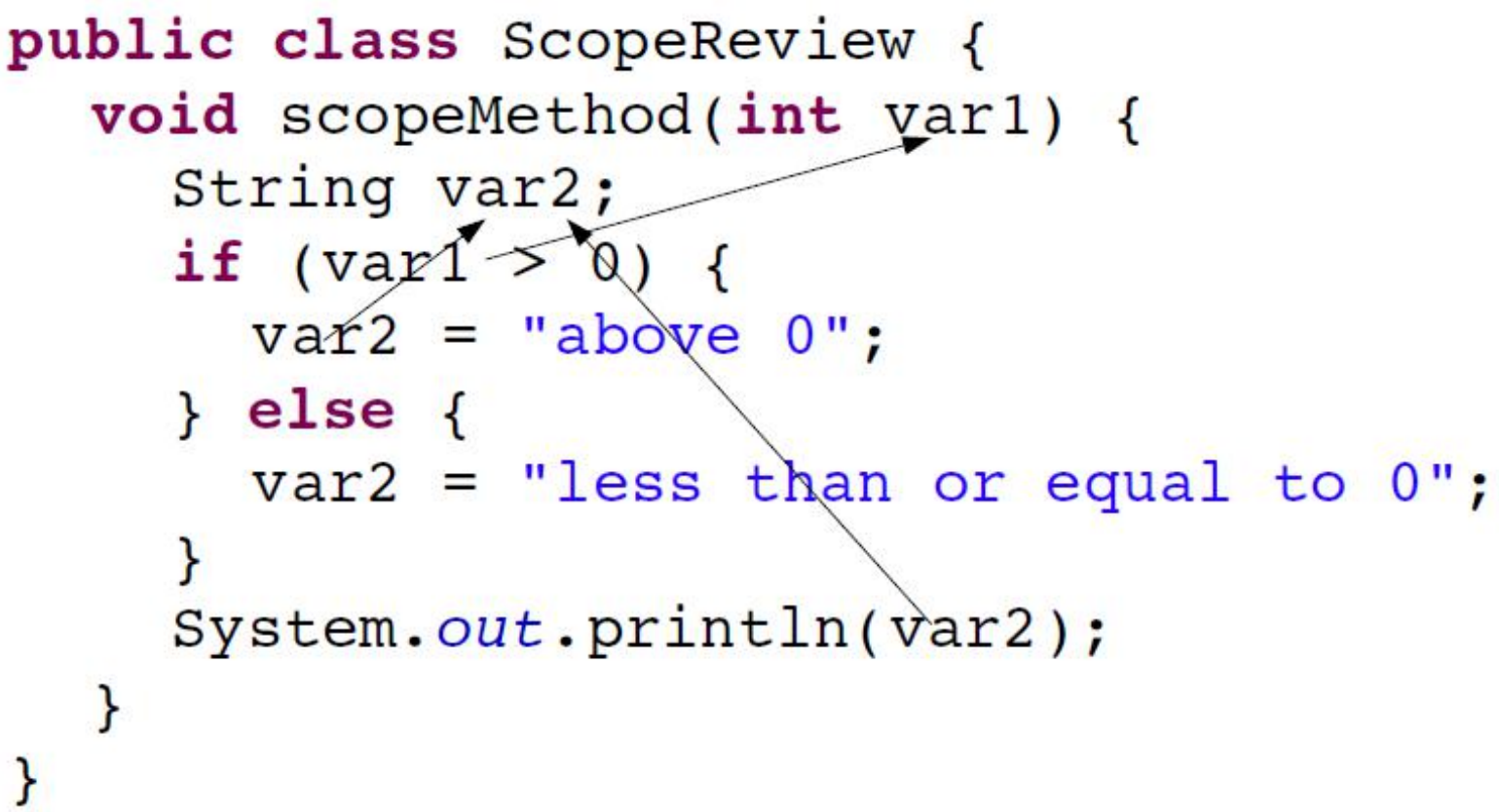
- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.
  - A bank app forbids a client to change an `Account`'s balance.
- Allows you to change the class implementation.
  - `Point` could be rewritten to use polar coordinates (radius *r*, angle *θ*), but with the same methods.



- Allows you to constrain objects' state.
  - Example: Only allow `Point`s with non-negative coordinates.

Class Scope

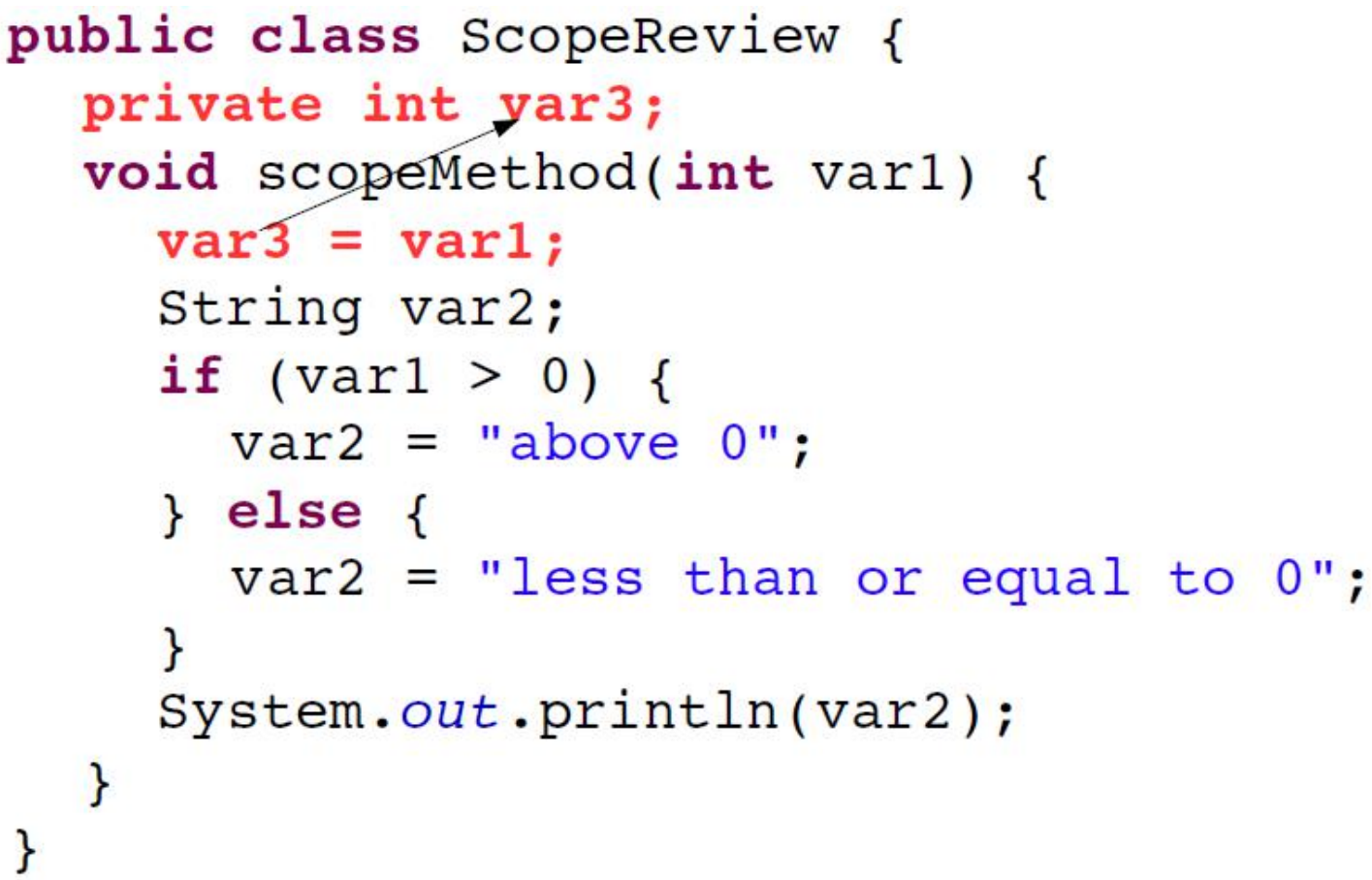
Scope Review



Scope Review

```
public class ScopeReview {
   private int var3;
   void scopeMethod(int var1) {
      var3 = var1;
      String var2;
      if (var1 > 0) {
         var2 = "above 0";
      } else {
         var2 = "less than or equal to 0";
      }
      System.out.println(var2);
   }
}
```

Class Scope



```
public class ScopeReview {
   private int var3;
   void scopeMethod(int var1) {
      var3 = var1;
      String var2;
      if (var1 > 0) {
         var2 = "above 0";
      } else {
         var2 = "less than or equal to 0";
      }
      System.out.println(var2);
   }
}
```

Variable names and scope

- Usually it is illegal to have two variables in the same scope with the same name.

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

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

  - The parameters to `setLocation` are named `newX` and `newY` to be distinct from the object's fields `x` and `y`.

Variable shadowing

- An instance method parameter can have the same name as one of the object's fields:

```
// this is legal
public void setLocation(int x, int y) {
    ...
}
```

  - Fields `x` and `y` are *shadowed* by parameters with same names.
  - Any `setLocation` code that refers to `x` or `y` will use the parameter, not the field.

Avoiding shadowing w/ `this`

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

    ...

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

- Inside the `setLocation` method,
  - When `this.x` is seen, the *field* `x` is used.
  - When `x` is seen, the *parameter* `x` is used.

'this' keyword

- Clarifies scope
- Means 'my object'

Usage:

```
class Example {
    int memberVariable;
    void setVariable(int newVal) {
        this.memberVariable += newVal;
    }
}
```

Multiple constructors

- It is legal to have more than one constructor in a class.
  - The constructors must accept different parameters.

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

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

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

    ...
}
```

Constructors and `this`

- One constructor can call another using `this`:

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

    public Point() {
        this(0, 0);  // calls the (x, y) constructor
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    ...
}
```

References

- http://math.hws.edu/javanotes/
- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/
- https://docs.oracle.com/javase/tutorial/java
- http://www.cs.utep.edu/vladik/cs2401.10a/Ch_11_Inheritance_Polymorphism.ppt
- https://people.cs.umass.edu/~moss/187/lectures/lecture-d-inheritance.ppt