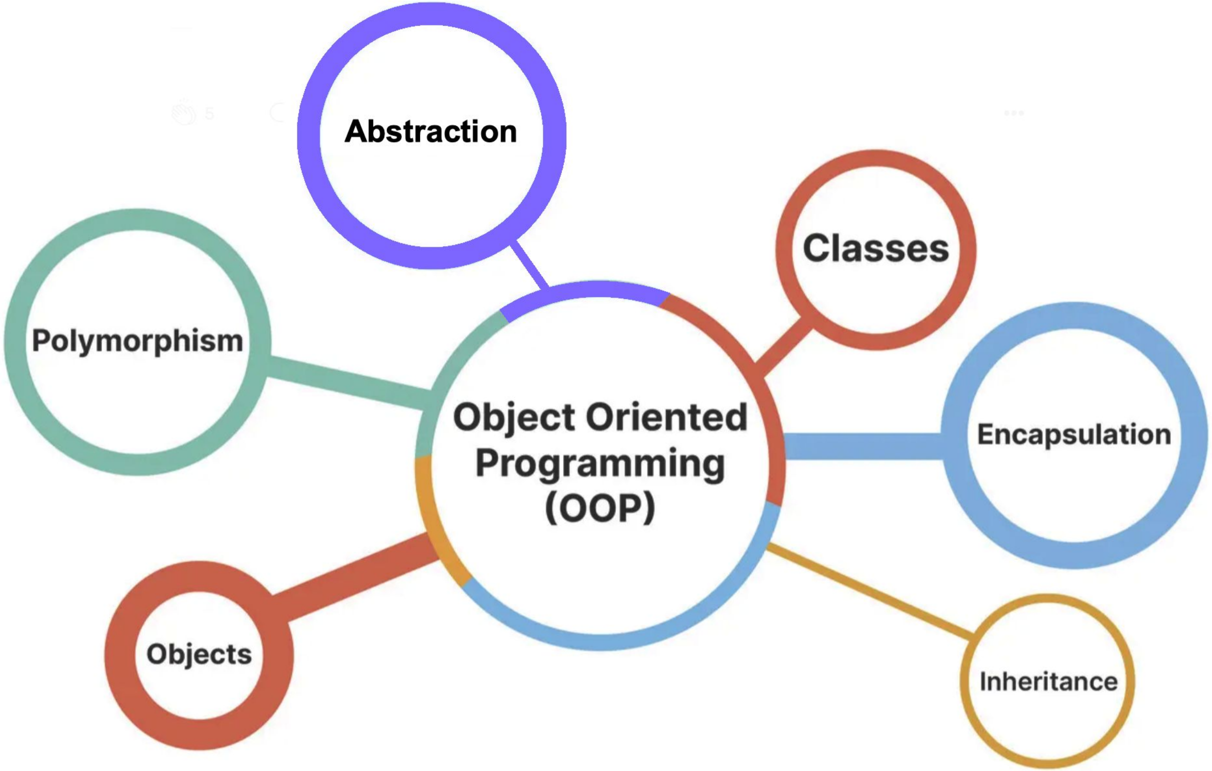
COSC 121: Computer Programming II



Today's Key Concepts

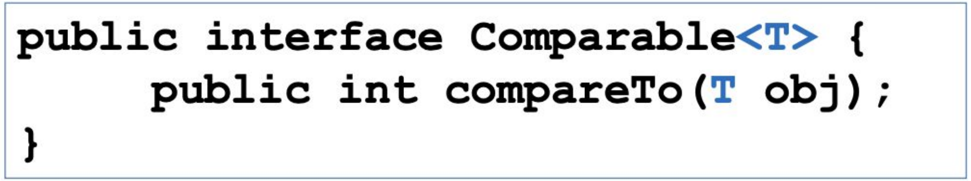
- Abstract classes are special classes that model generic concepts but never get instantiated
    - Keyword: abstract
- An interface defines the blueprint for a class, defining the set of methods that must be implemented without defining their specific implementation
    - User defined interfaces
        - Methods are all implicitly public and abstract
        - Java 8/9: Methods can be default, static, private
    - Java standard interfaces: Comparable and Cloneable
- Interfaces or abstract classes?

Java Standard Interfaces

- Built-in interfaces provided within Java
- Define common behaviors or capabilities that developers can leverage in their own classes
- This class, two interfaces:
    - Comparable
        - Allows comparing or sorting objects
    - Cloneable
        - Allows cloning objects

The Comparable Interface

- Definition:

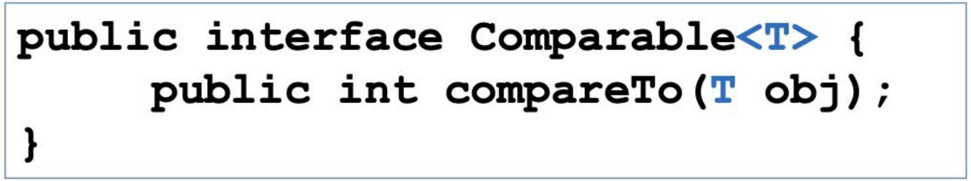


```
public interface Comparable<T> {
     public int compareTo(T obj);
}
```

- Is a generic interface
    - Generic type <T> is replaced by a concrete type when implementing this interface

The Comparable Interface

- Definition:

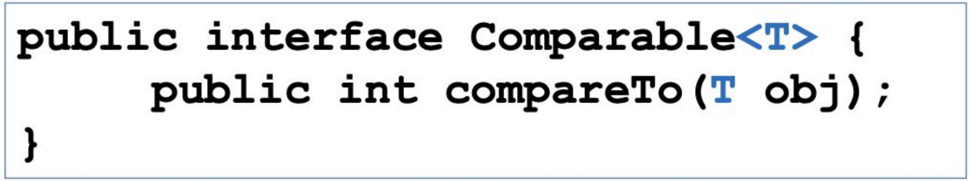


```
public interface Comparable<T> {
     public int compareTo(T obj);
}
```

- Is a generic interface
  - Generic type <T> is replaced by a concrete type when implementing this interface
- Contains one abstract method called compareTo()
  - Returns an integer *(Why?)*

The Comparable Interface

- Definition:



```
public interface Comparable<T> {
     public int compareTo(T obj);
}
```

- Is a generic interface
    - Generic type <T> is replaced by a concrete type when implementing this interface
- Contains one abstract method called compareTo()
    - Returns an integer *(Why?)*
- Use when you want to compare two objects of the same type
- Any class that implements the Comparable interface must define an appropriate measure of comparison in compareTo()

String implements Comparable

- Recall that String class lets us compare strings by lexicographical order
- Ex: "abc" < "bcd"

String implements Comparable

- Recall that String class lets us compare strings by lexicographical order
- Ex: "abc" < "bcd"
- Example code using compareTo():

```
String one = "abc";
String two = "bcd";
if( one.compareTo( two ) < 0 )
    System.out.println( one + " is less than " + two );
```

String implements Comparable

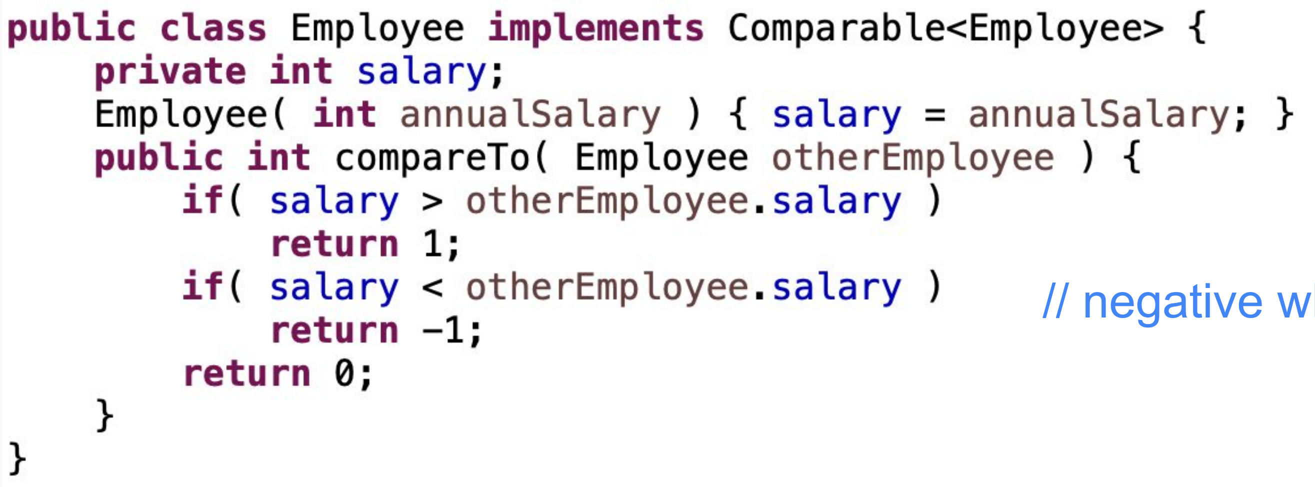
- Recall that String class lets us compare strings by lexicographical order
- Ex: "abc" < "bcd"
- Example code using compareTo():

```
String one = "abc";
String two = "bcd";
if( one.compareTo( two ) < 0 )
    System.out.println( one + " is less than " + two );
```

- By convention and by the Comparable contract, the returned answer should be negative when this object is “smaller than” the object passed in

Example Using Comparable

- Implementing Comparable:

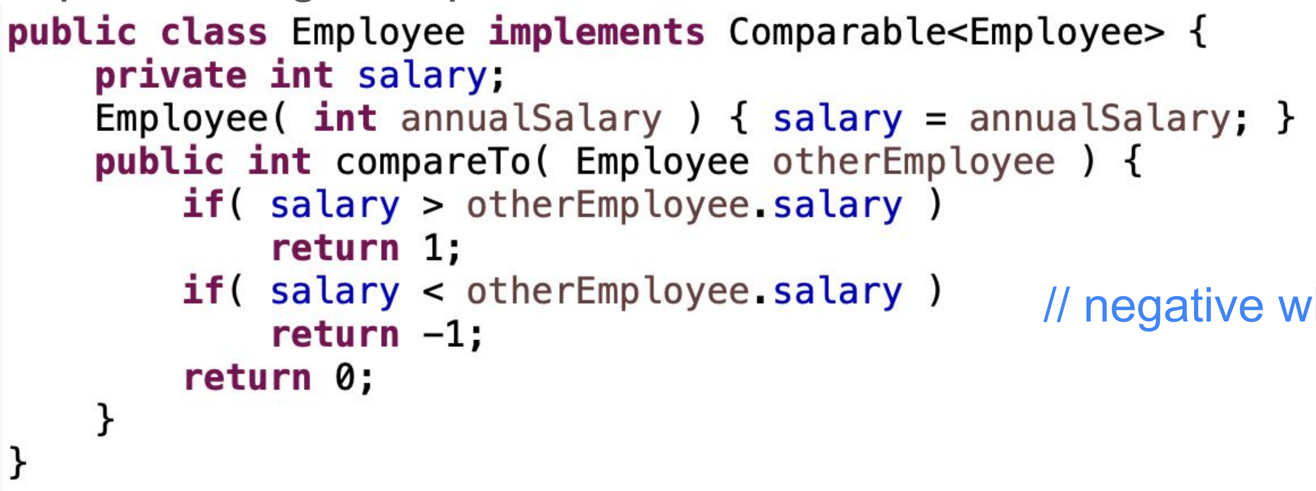


```
public class Employee implements Comparable<Employee> {
    private int salary;
    Employee( int annualSalary ) { salary = annualSalary; }
    public int compareTo( Employee otherEmployee ) {
        if( salary > otherEmployee.salary )
            return 1;
        if( salary < otherEmployee.salary )
            return -1;
        return 0;
    }
}
```

// negative when this is smaller

Example Using Comparable

- Implementing Comparable:

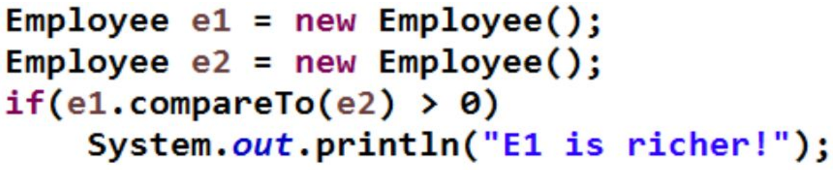


```
public class Employee implements Comparable<Employee> {
    private int salary;
    Employee( int annualSalary ) { salary = annualSalary; }
    public int compareTo( Employee otherEmployee ) {
        if( salary > otherEmployee.salary )
            return 1;
        if( salary < otherEmployee.salary )
            return -1;
        return 0;
    }
}
```

add name of class

// negative when this is smaller

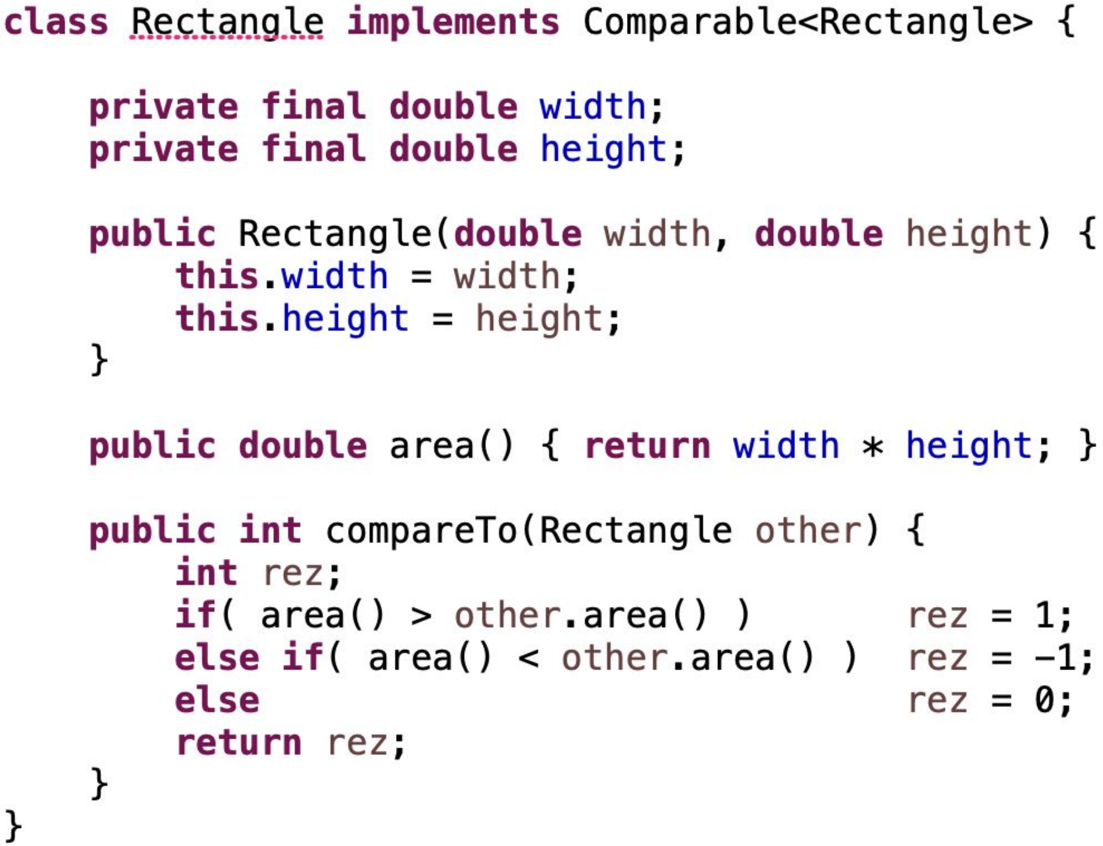
- Invoking the method:



```
Employee e1 = new Employee();
Employee e2 = new Employee();
if(e1.compareTo(e2) > 0)
    System.out.println("E1 is richer!");
```

Syntax:
o1.compareTo(o2)

Given Rectangle Class



```
class Rectangle implements Comparable<Rectangle> {

    private final double width;
    private final double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double area() { return width * height; }

    public int compareTo(Rectangle other) {
        int rez;
        if( area() > other.area() )         rez = 1;
        else if( area() < other.area() )    rez = -1;
        else                                rez = 0;
        return rez;
    }
}
```

add name of class

typical class definition

unique compareTo() definition

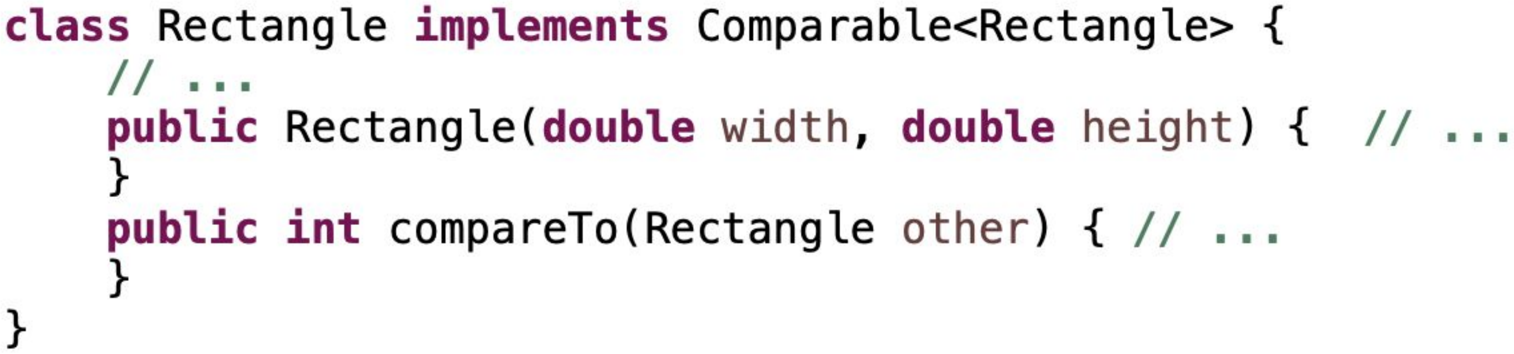
- returns comparison result

How to Test Rectangle class? (~2 min)

- Write a test class that creates two Rectangle objects and compares them to see which is bigger. Display the comparison result in English.

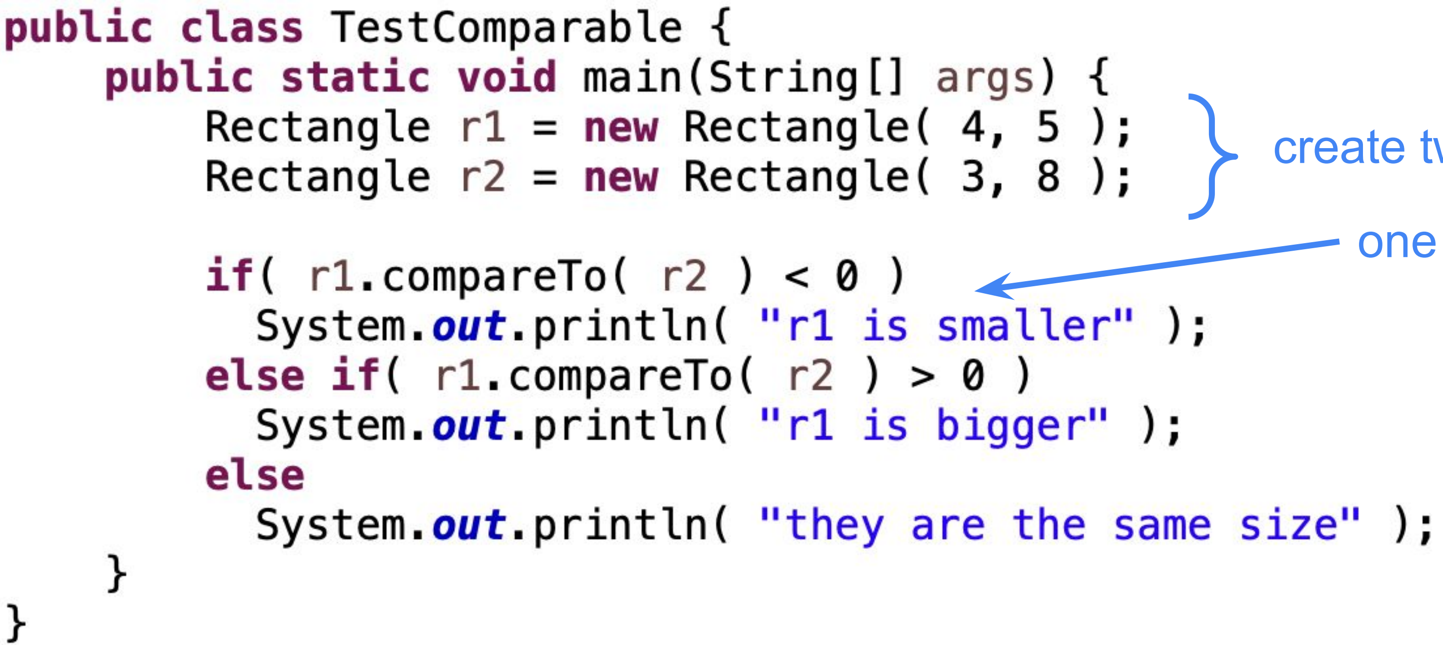
Write out the code

What you need from the Rectangle class:



```
class Rectangle implements Comparable<Rectangle> {
    // ...
    public Rectangle(double width, double height) {  // ...
    }
    public int compareTo(Rectangle other) { // ...
    }
}
```

Sample Solution



```
public class TestComparable {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle( 4, 5 );
        Rectangle r2 = new Rectangle( 3, 8 );

        if( r1.compareTo( r2 ) < 0 )
          System.out.println( "r1 is smaller" );
        else if( r1.compareTo( r2 ) > 0 )
          System.out.println( "r1 is bigger" );
        else
          System.out.println( "they are the same size" );
    }
}
```

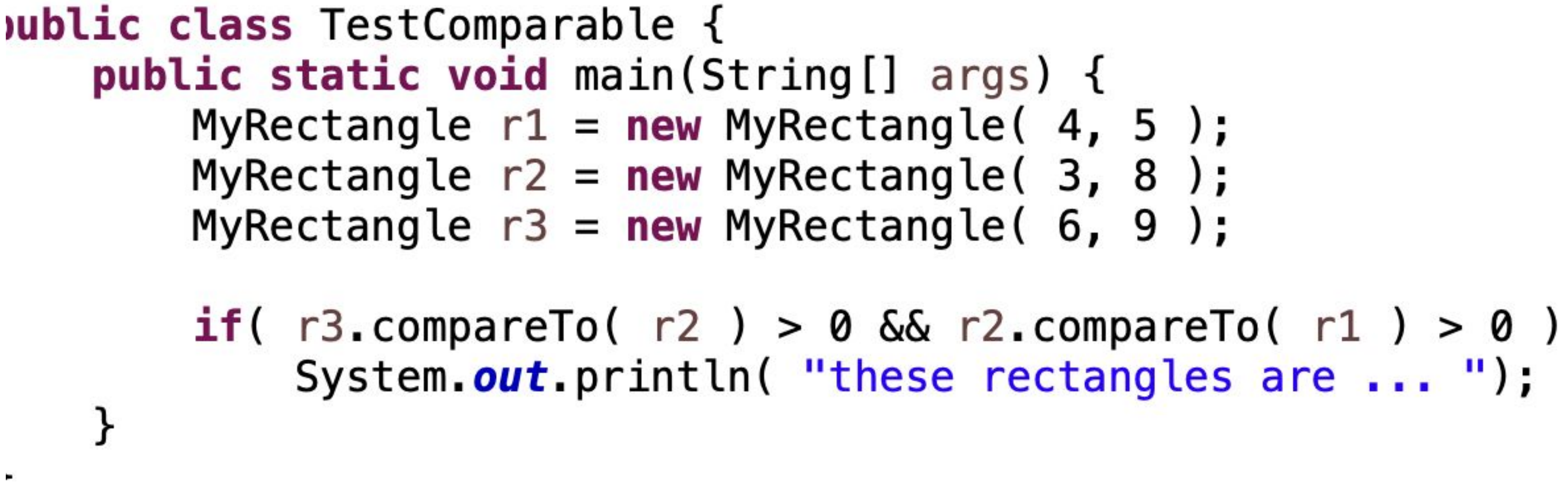
create two objects

one comparison

check all possible results

Output?

Is Comparable Transitive?



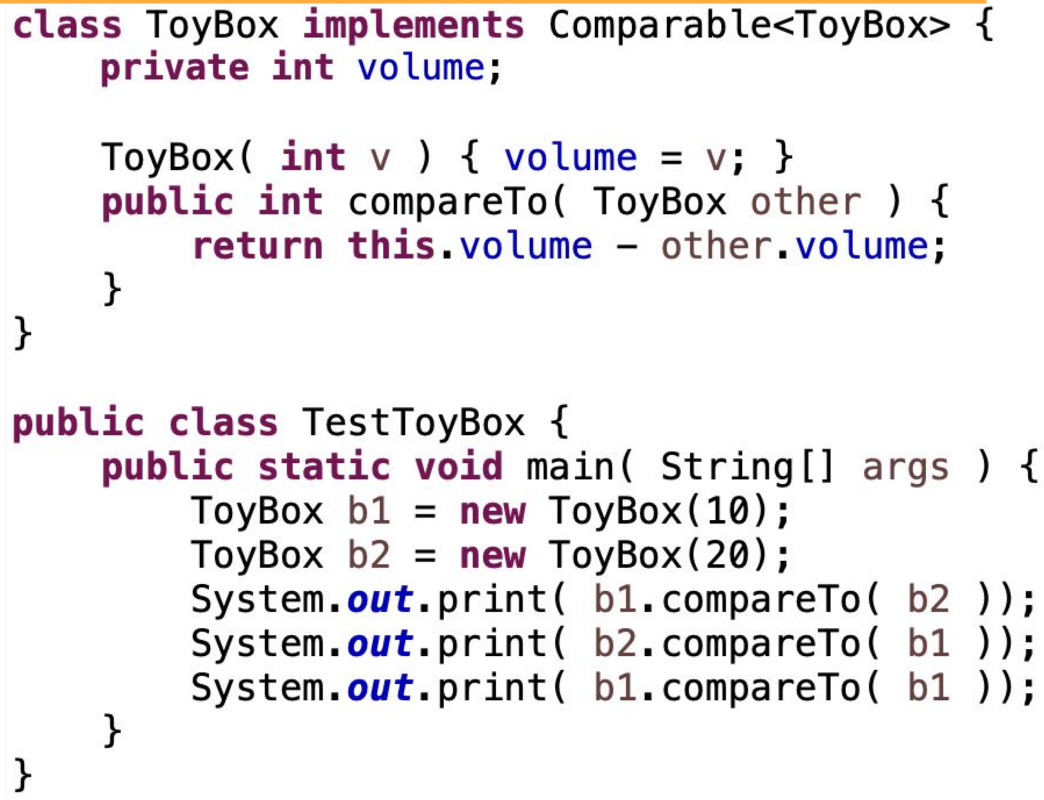
```
ublic class TestComparable {
    public static void main(String[] args) {
        MyRectangle r1 = new MyRectangle( 4, 5 );
        MyRectangle r2 = new MyRectangle( 3, 8 );
        MyRectangle r3 = new MyRectangle( 6, 9 );

        if( r3.compareTo( r2 ) > 0 && r2.compareTo( r1 ) > 0 )
            System.out.println( "these rectangles are ... ");
    }
```

if the if-statement is true,
what must be true about r1 and r3?

iClicker Question

What's the output for the ToyBox code?

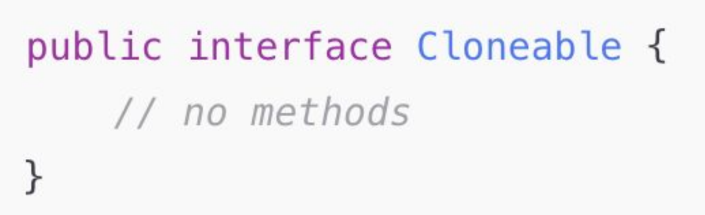
A. -10 10 0
B. -1 1 0
C. false true true
D. 10 -10 0



```
class ToyBox implements Comparable<ToyBox> {
    private int volume;

    ToyBox( int v ) { volume = v; }
    public int compareTo( ToyBox other ) {
        return this.volume - other.volume;
    }
}

public class TestToyBox {
    public static void main( String[] args ) {
        ToyBox b1 = new ToyBox(10);
        ToyBox b2 = new ToyBox(20);
        System.out.print( b1.compareTo( b2 ));
        System.out.print( b2.compareTo( b1 ));
        System.out.print( b1.compareTo( b1 ));
    }
}
```

The Cloneable Interface

- A special interface called a marker interface (i.e., no methods!)
- It serves as a signal to the Object.clone() method, rather than defining behavior itself
- Conceptually:
  - A marker interface is like a sign on the door, not a tool in the room.
  - The sign says what’s allowed
  - Someone else decides what to do because of that sign
- Definition:



```
public interface Cloneable {
    // no methods
}
```

recall we saw this example in last class's clicker question

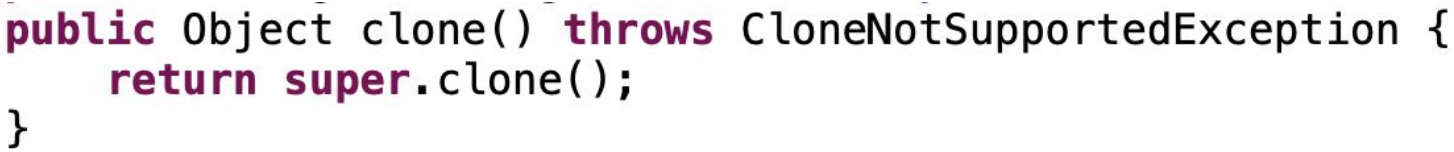
What the Cloneable Interface Allows

- In order to create a clone of an object (i.e., a field-for-field copy of the object), a class must implement the Cloneable interface

What the Cloneable Interface Allows

- In order to create a clone of an object (i.e., a field-for-field copy of the object), a class must implement the Cloneable interface
- By convention, classes that implement this interface should override Object.clone (which is protected) using a **public** method

  Ex:



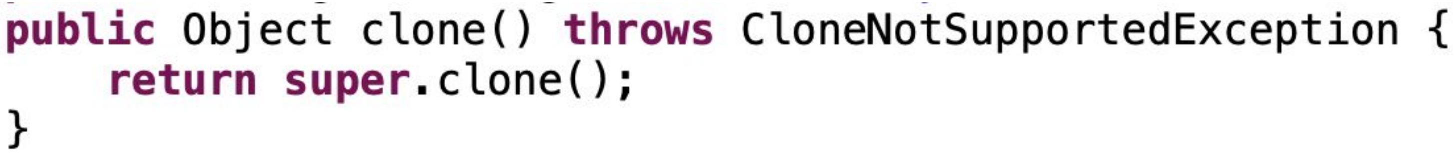
```
public Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

  (we will discuss exceptions next time)

What the Cloneable Interface Allows

- In order to create a clone of an object (i.e., a field-for-field copy of the object), a class must implement the Cloneable interface
- By convention, classes that implement this interface should override Object.clone (which is protected) using a **public** method

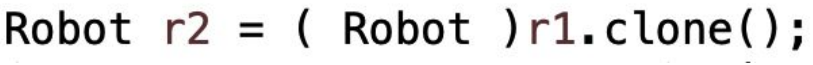
  Ex:

```
public Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

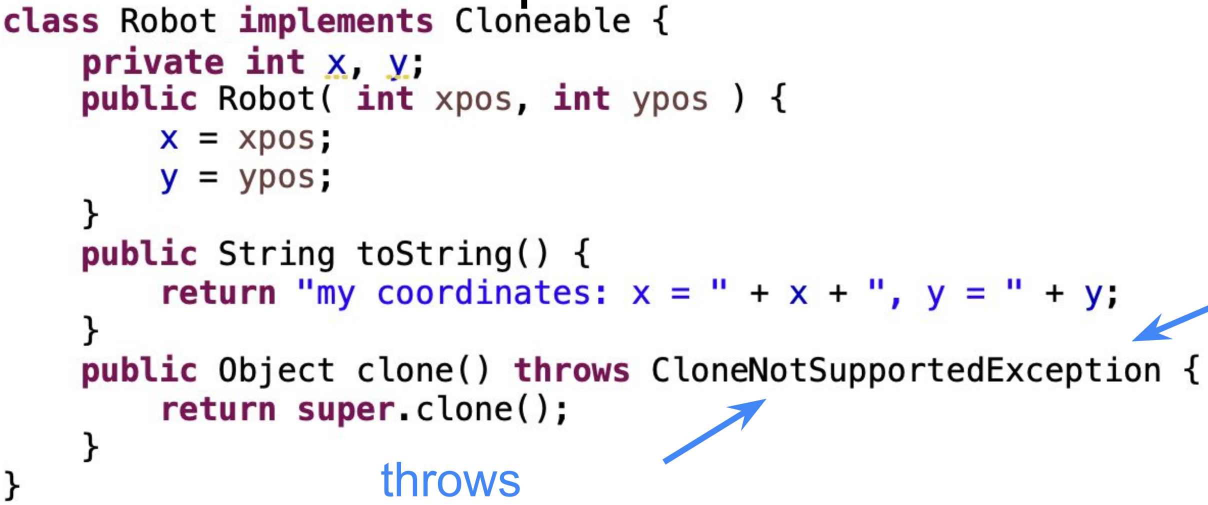
  (we will discuss exceptions next time)
- Casting is therefore required when invoking clone() method:

  Ex:

```
Robot r2 = ( Robot )r1.clone();
```

Full Robot Example

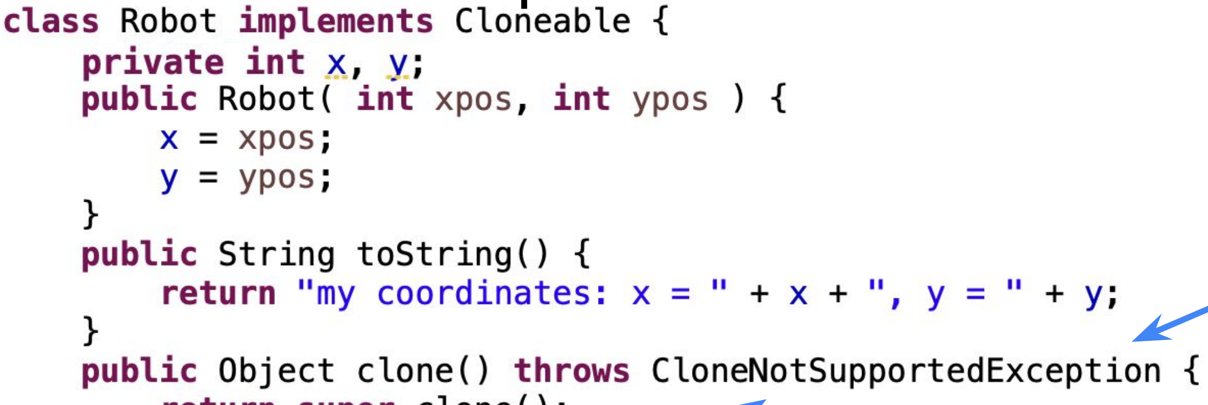


```
class Robot implements Cloneable {
    private int x, y;
    public Robot( int xpos, int ypos ) {
        x = xpos;
        y = ypos;
    }
    public String toString() {
        return "my coordinates: x = " + x + ", y = " + y;
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

the clone() method

throws
exception

Full Robot Example

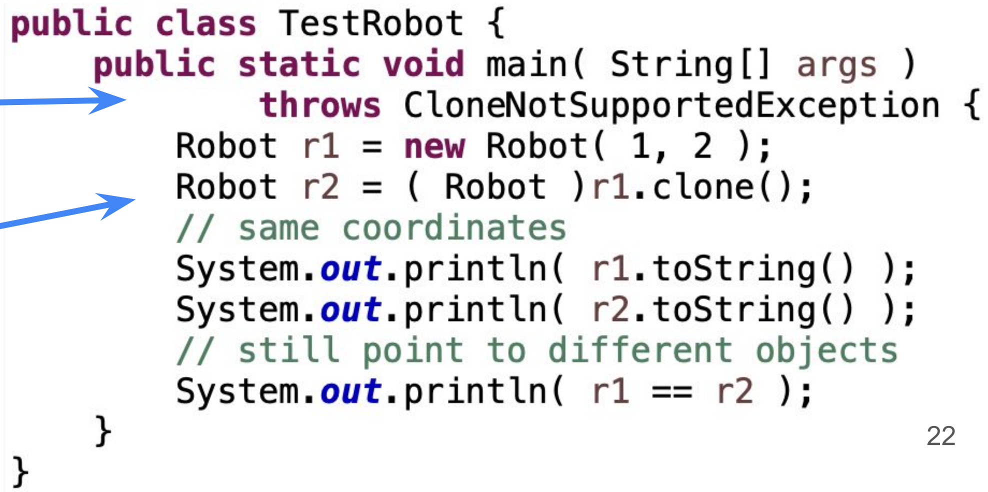


```
class Robot implements Cloneable {
    private int x, y;
    public Robot( int xpos, int ypos ) {
        x = xpos;
        y = ypos;
    }
    public String toString() {
        return "my coordinates: x = " + x + ", y = " + y;
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

the clone() method

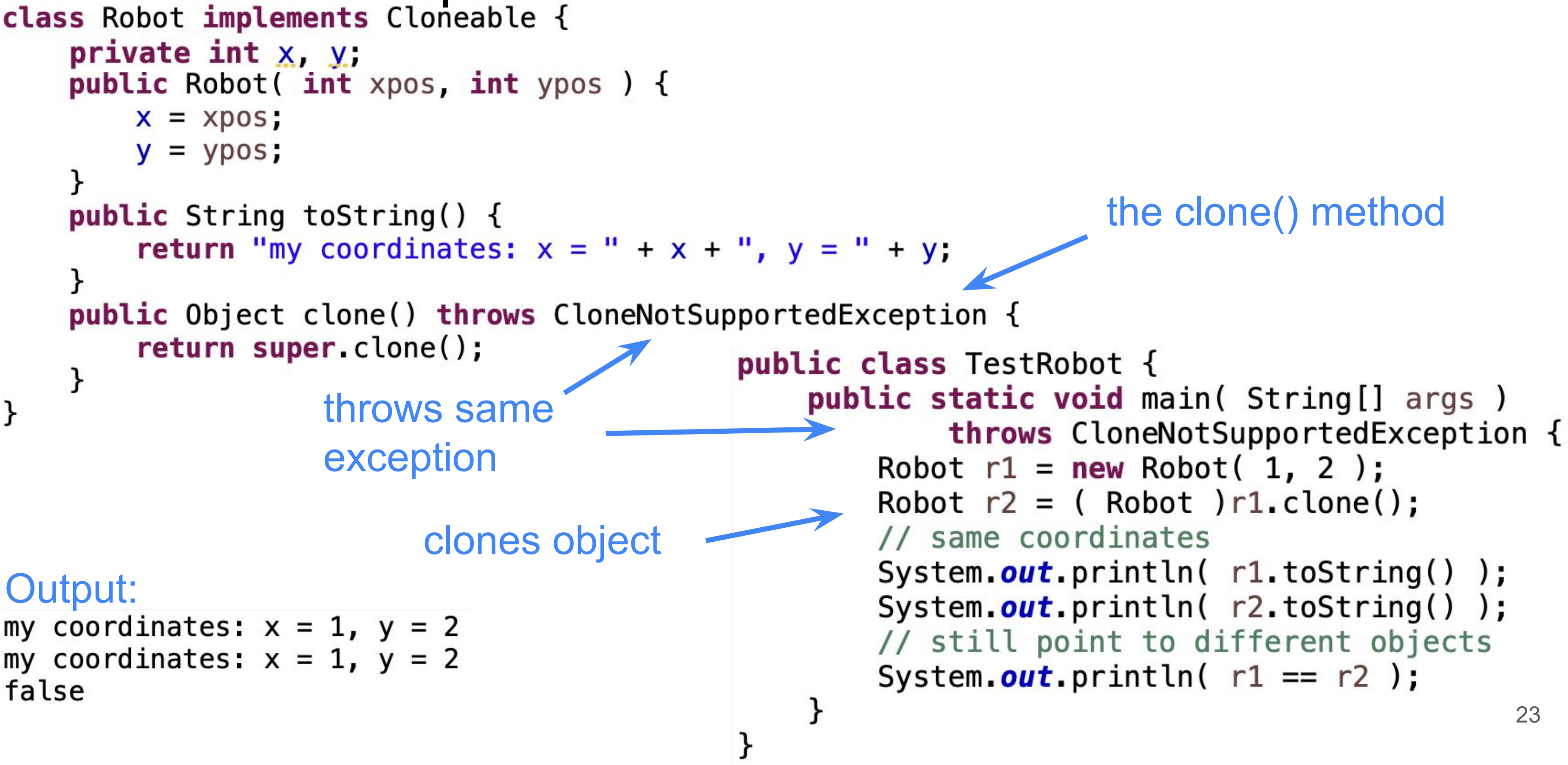
throws same exception

clones object



```
public class TestRobot {
    public static void main( String[] args )
            throws CloneNotSupportedException {
        Robot r1 = new Robot( 1, 2 );
        Robot r2 = ( Robot )r1.clone();
        // same coordinates
        System.out.println( r1.toString() );
        System.out.println( r2.toString() );
        // still point to different objects
        System.out.println( r1 == r2 );
    }
}
```

Full Robot Example



```java
class Robot implements Cloneable {
    private int x, y;
    public Robot( int xpos, int ypos ) {
        x = xpos;
        y = ypos;
    }
    public String toString() {
        return "my coordinates: x = " + x + ", y = " + y;
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

the clone() method

throws same exception

clones object

```java
public class TestRobot {
    public static void main( String[] args )
            throws CloneNotSupportedException {
        Robot r1 = new Robot( 1, 2 );
        Robot r2 = ( Robot )r1.clone();
        // same coordinates
        System.out.println( r1.toString() );
        System.out.println( r2.toString() );
        // still point to different objects
        System.out.println( r1 == r2 );
    }
}
```

Output:

```
my coordinates: x = 1, y = 2
my coordinates: x = 1, y = 2
false
```

Shallow Copy

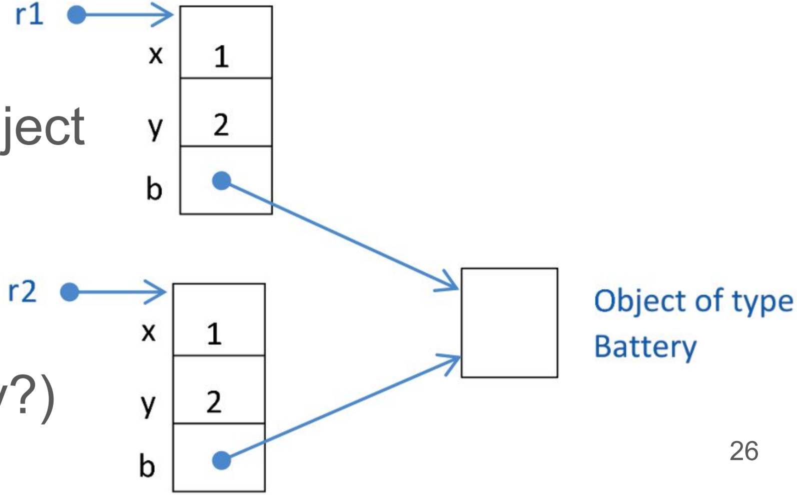
- The Object’s clone method performs a shallow copy
- Suppose: Robot r2 = (Robot) r1.clone();

Shallow Copy

- The Object's clone method performs a shallow copy
- Suppose: Robot r2 = (Robot) r1.clone();
- **For a primitive field**, its value is copied
    - e.g., the value of x and y (integers) are copied from r1 to r2
- **For an object field**, the reference is copied (not the contents)

Shallow Copy

- The Object's clone method performs a shallow copy
- Suppose: Robot r2 = (Robot) r1.clone();
- **For a primitive field**, its value is copied
  - e.g., the value of x and y (integers) are copied from r1 to r2
- **For an object field**, the reference is copied (not the contents)
  - e.g., if Robot class had a Battery object and r2 is a clone of r1, then r1 and r2 refers to same Battery object (do two robots use the same battery?)

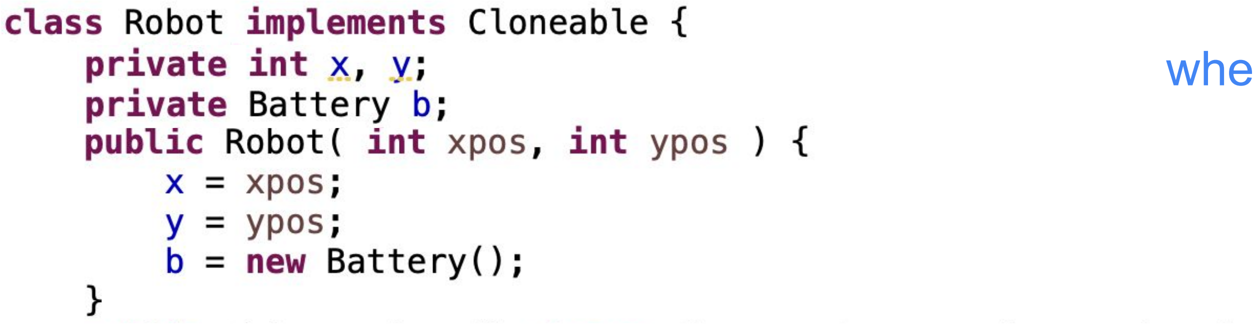


Deep Copy

- To perform a deep copy you need to clone each internal object

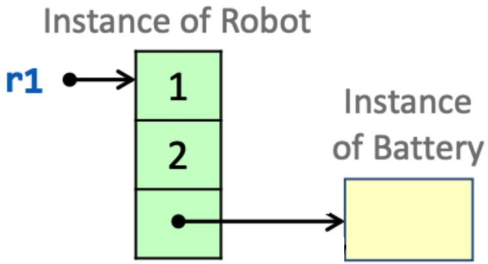
Deep Copy

- To perform a deep copy you need to clone each internal object
- Add battery attribute to Robot and redefine clone()



```
class Robot implements Cloneable {
    private int x, y;
    private Battery b;
    public Robot( int xpos, int ypos ) {
        x = xpos;
        y = ypos;
        b = new Battery();
    }
```

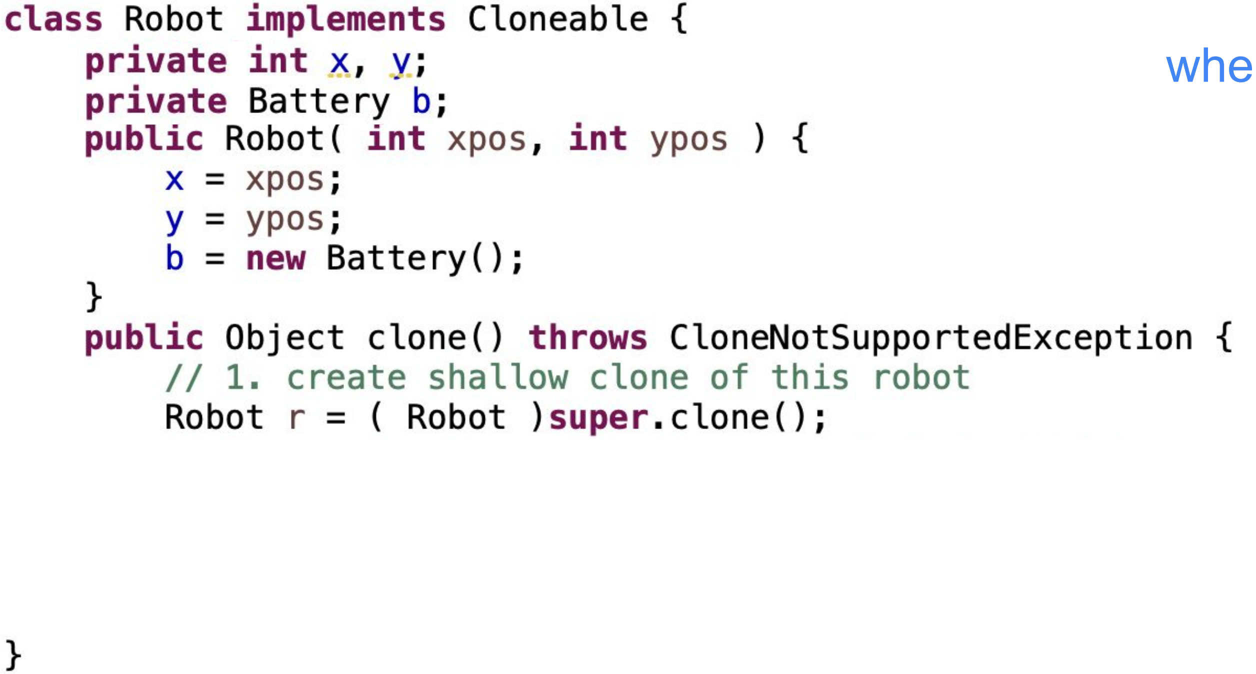
when r1 is first created:



```
}
```

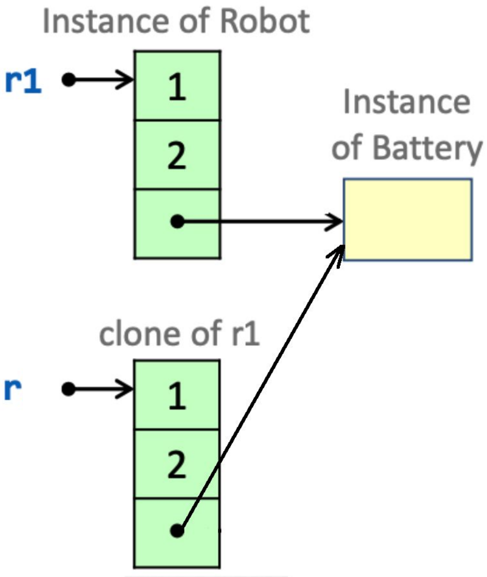
Deep Copy

- To perform a deep copy you need to clone each internal object
- Add battery attribute to Robot and redefine clone()



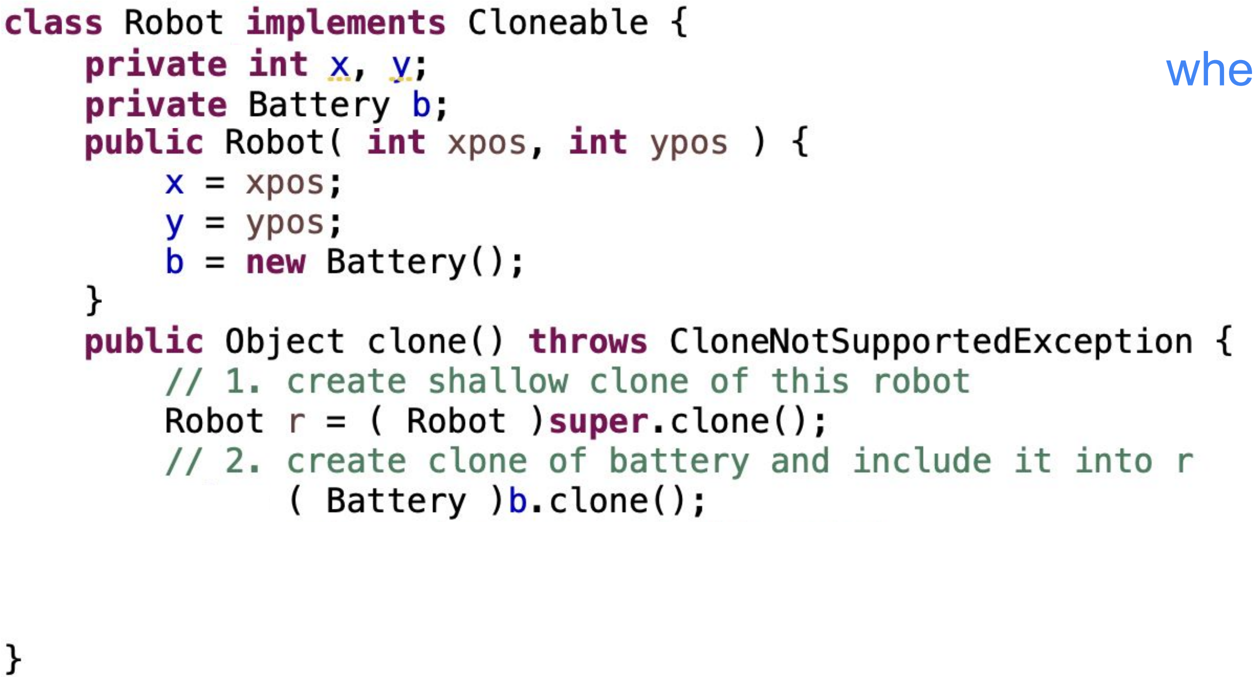
```java
class Robot implements Cloneable {
    private int x, y;
    private Battery b;
    public Robot( int xpos, int ypos ) {
        x = xpos;
        y = ypos;
        b = new Battery();
    }
    public Object clone() throws CloneNotSupportedException {
        // 1. create shallow clone of this robot
        Robot r = ( Robot )super.clone();



}
```

when r1.clone() is called:



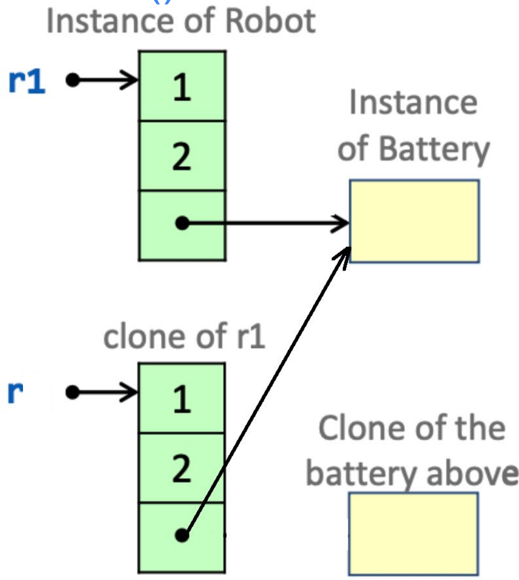
Deep Copy

- To perform a deep copy you need to clone each internal object
- Add battery attribute to Robot and redefine clone()



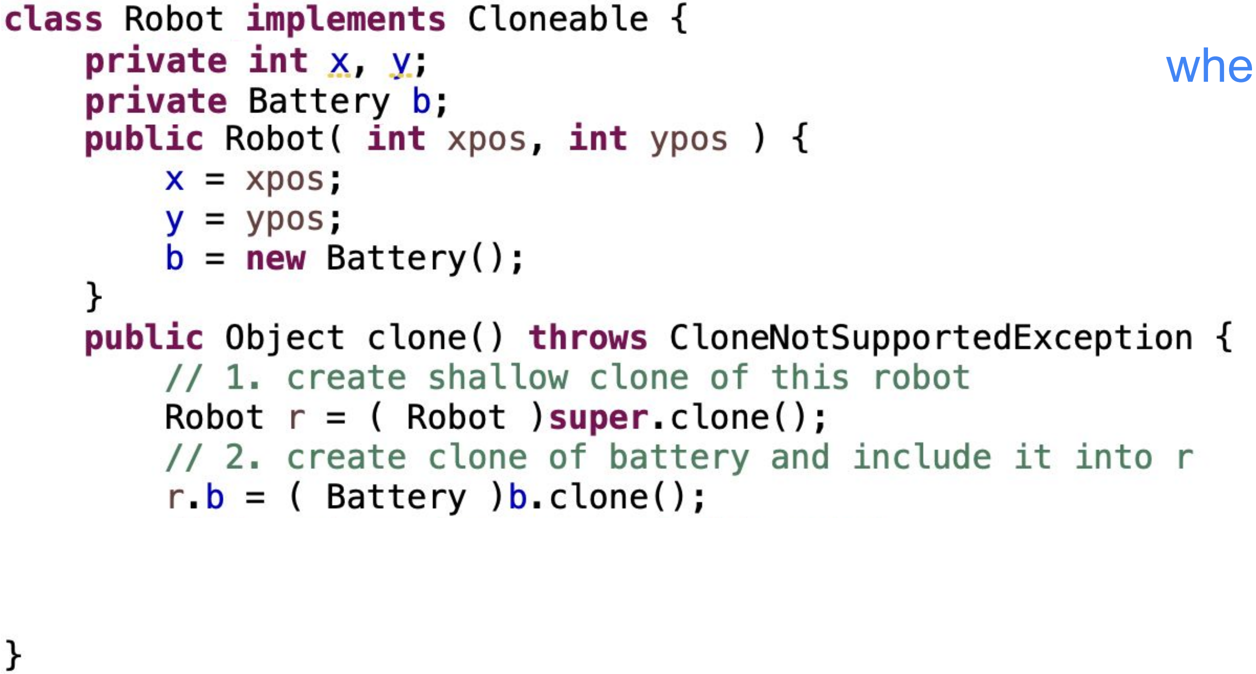
```
class Robot implements Cloneable {
    private int x, y;
    private Battery b;
    public Robot( int xpos, int ypos ) {
        x = xpos;
        y = ypos;
        b = new Battery();
    }
    public Object clone() throws CloneNotSupportedException {
        // 1. create shallow clone of this robot
        Robot r = ( Robot )super.clone();
        // 2. create clone of battery and include it into r
              ( Battery )b.clone();


}
```

when r1.clone() is called:



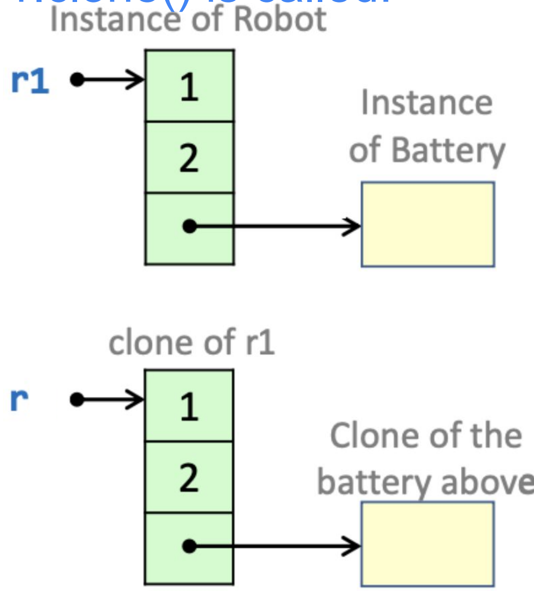
Deep Copy

- To perform a deep copy you need to clone each internal object
- Add battery attribute to Robot and redefine clone()



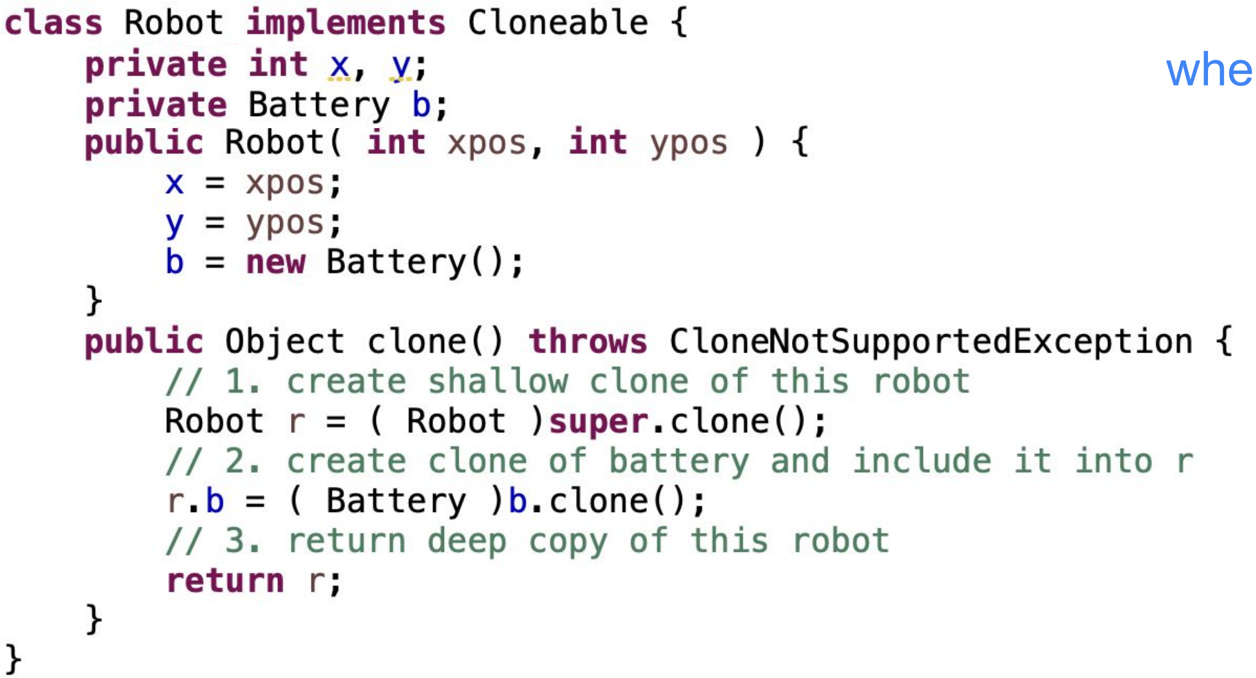
```
class Robot implements Cloneable {
    private int x, y;
    private Battery b;
    public Robot( int xpos, int ypos ) {
        x = xpos;
        y = ypos;
        b = new Battery();
    }
    public Object clone() throws CloneNotSupportedException {
        // 1. create shallow clone of this robot
        Robot r = ( Robot )super.clone();
        // 2. create clone of battery and include it into r
        r.b = ( Battery )b.clone();



}
```

when r1.clone() is called:



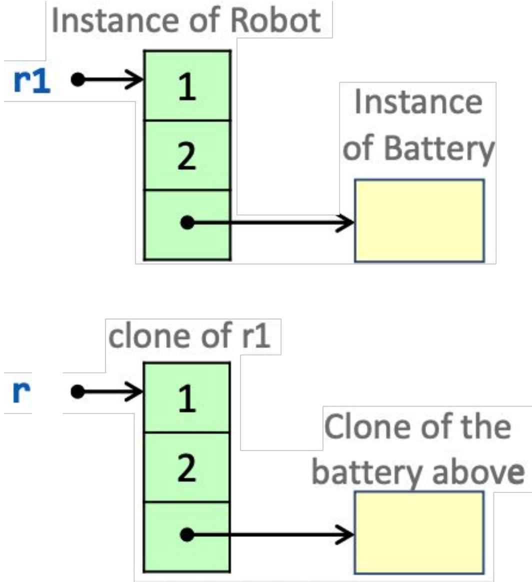
Deep Copy

- To perform a deep copy you need to clone each internal object
- Add battery attribute to Robot and redefine clone()



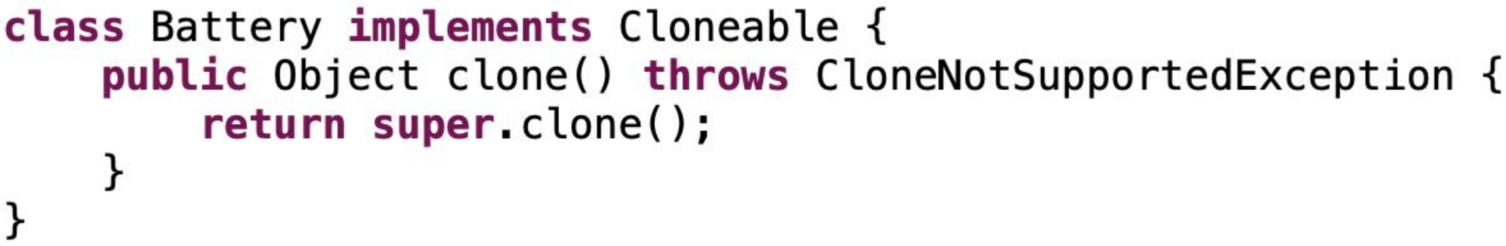
```
class Robot implements Cloneable {
    private int x, y;
    private Battery b;
    public Robot( int xpos, int ypos ) {
        x = xpos;
        y = ypos;
        b = new Battery();
    }
    public Object clone() throws CloneNotSupportedException {
        // 1. create shallow clone of this robot
        Robot r = ( Robot )super.clone();
        // 2. create clone of battery and include it into r
        r.b = ( Battery )b.clone();
        // 3. return deep copy of this robot
        return r;
    }
}
```

when r1.clone() is called:



Deep Copy (cont.)

- Since Robot's clone() now has a statement:
  r.b = (Battery)b.clone()
- That means Battery class also needs to implement clone()
- Ex:



```java
class Battery implements Cloneable {
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

iClicker Question

Suppose we want to clone a Player who has a Position attribute (see code below), what is the output after the following statements in a main() method?

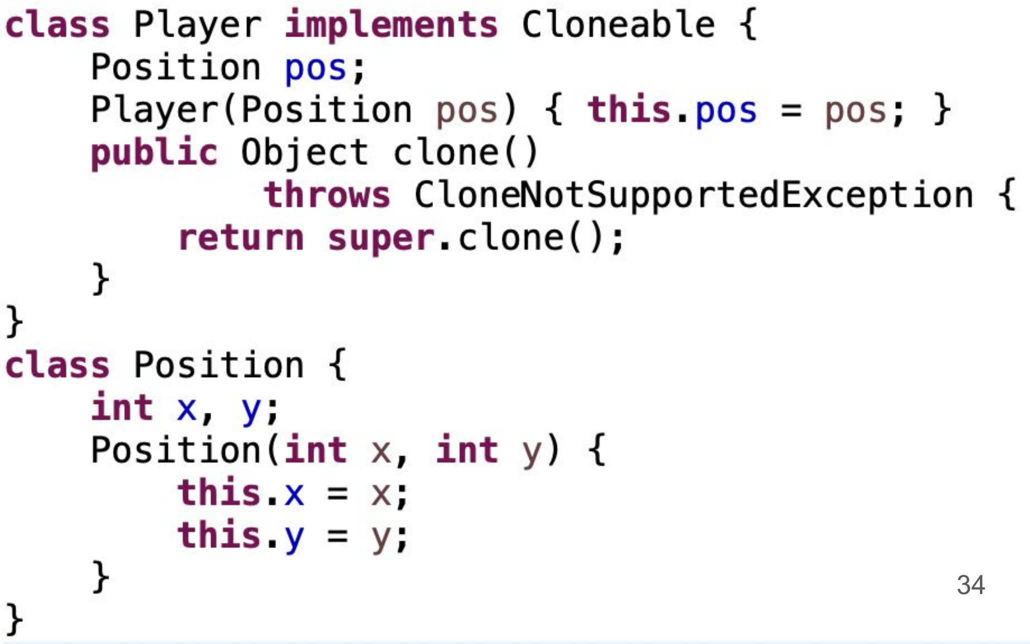
```
Position p = new Position(1, 2);
Player p1 = new Player( p );
Player p2 = ( Player )p1.clone();
p2.pos.x = 99;
System.out.println( p1.pos.x );
```

A. 1

B. 2

C. 99

D. Error

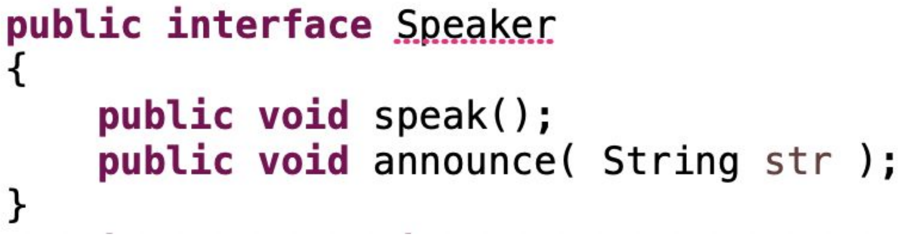


```
class Player implements Cloneable {
    Position pos;
    Player(Position pos) { this.pos = pos; }
    public Object clone()
            throws CloneNotSupportedException {
        return super.clone();
    }
}
class Position {
    int x, y;
    Position(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Polymorphism via Interfaces

- Can also use interfaces to setup polymorphic references
- Follows the same rules as inheritance
- New situation:

  Suppose we have:



```
public interface Speaker
{
        public void speak();
        public void announce( String str );
}
```

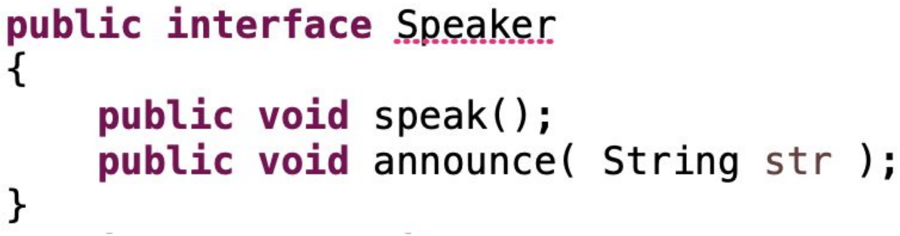
- Can**not** write in test class: Speaker presenter = new Speaker();

  Why?

Polymorphism via Interfaces

- Can also use interfaces to setup polymorphic references
- Follows the same rules as inheritance
- New situation:

  Suppose we have:



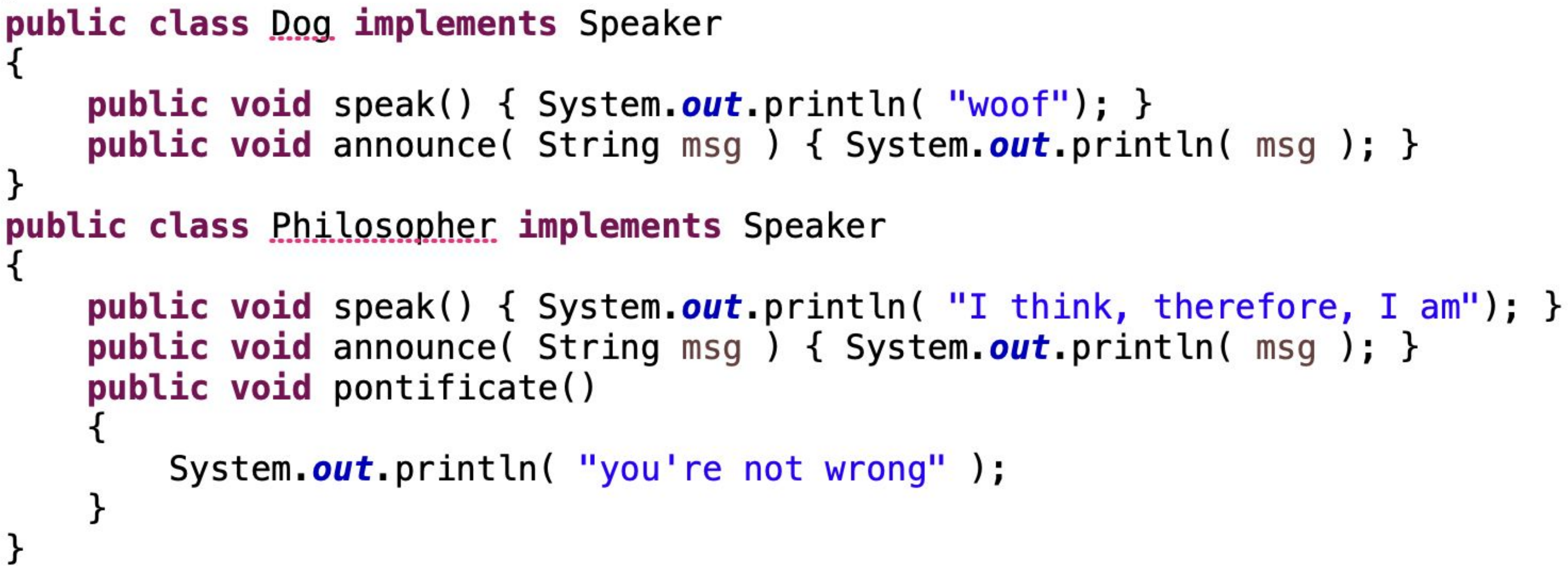
```
public interface Speaker
{
    public void speak();
    public void announce( String str );
}
```

- Can**not** write in test class: Speaker presenter = new Speaker();

  Why? Cannot instantiate interface objects

Speaker Example (cont.)

- Let's suppose we have these Dog and Philosopher classes



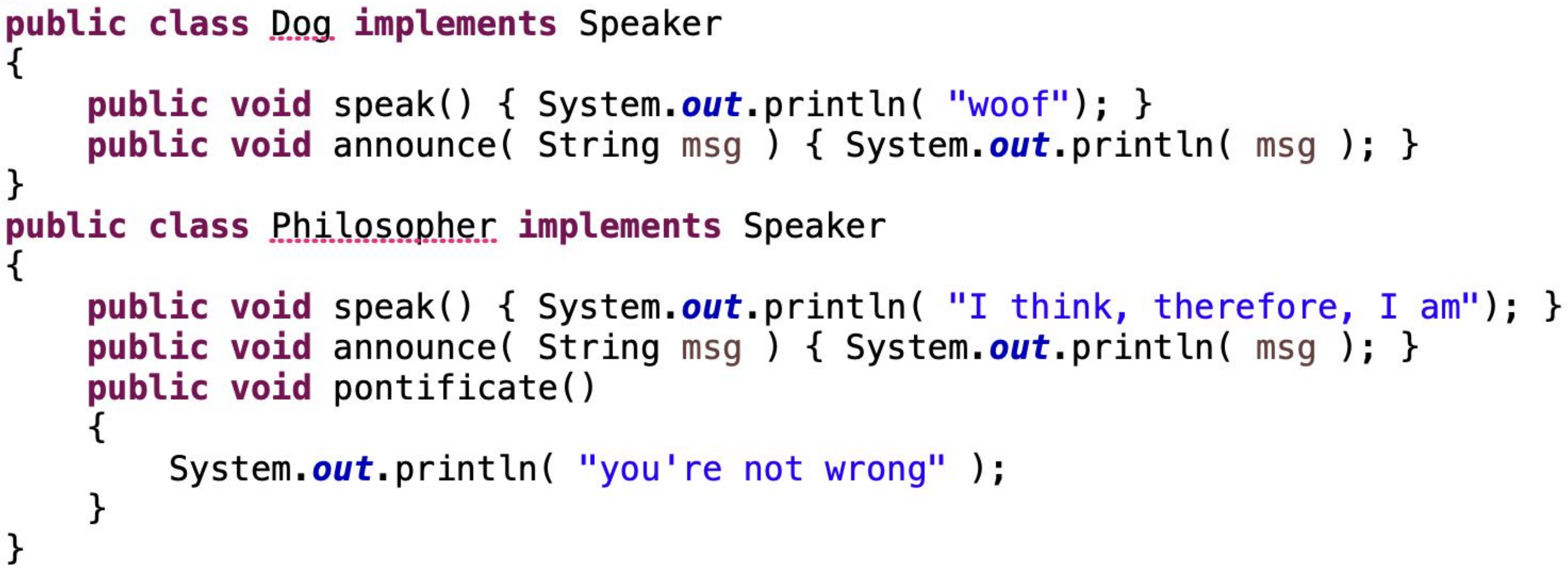
```
public class Dog implements Speaker
{
    public void speak() { System.out.println( "woof"); }
    public void announce( String msg ) { System.out.println( msg ); }
}
public class Philosopher implements Speaker
{
    public void speak() { System.out.println( "I think, therefore, I am"); }
    public void announce( String msg ) { System.out.println( msg ); }
    public void pontificate()
    {
        System.out.println( "you're not wrong" );
    }
}
```

- Cannot write: Philosopher presenter = new Speaker();

  Why?

Speaker Example (cont.)

- Let's suppose we have these Dog and Philosopher classes



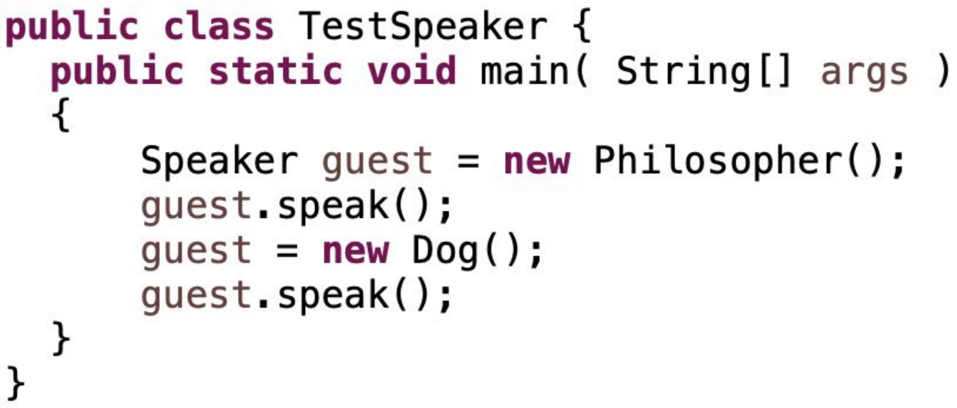
```
public class Dog implements Speaker
{
    public void speak() { System.out.println( "woof"); }
    public void announce( String msg ) { System.out.println( msg ); }
}
public class Philosopher implements Speaker
{
    public void speak() { System.out.println( "I think, therefore, I am"); }
    public void announce( String msg ) { System.out.println( msg ); }
    public void pontificate()
    {
        System.out.println( "you're not wrong" );
    }
}
```

- Cannot write: Philosopher presenter = new Speaker();

  Why? Cannot instantiate interface implementer "is-not" an interface ("Rule 1")

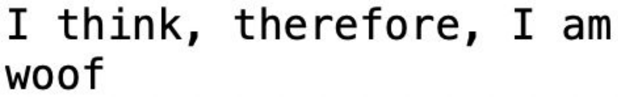
Speaker Example (cont.)

- Test class:



```
public class TestSpeaker {
  public static void main( String[] args )
  {
       Speaker guest = new Philosopher();
       guest.speak();
       guest = new Dog();
       guest.speak();
  }
}
```

- Output:



```
I think, therefore, I am
woof
```

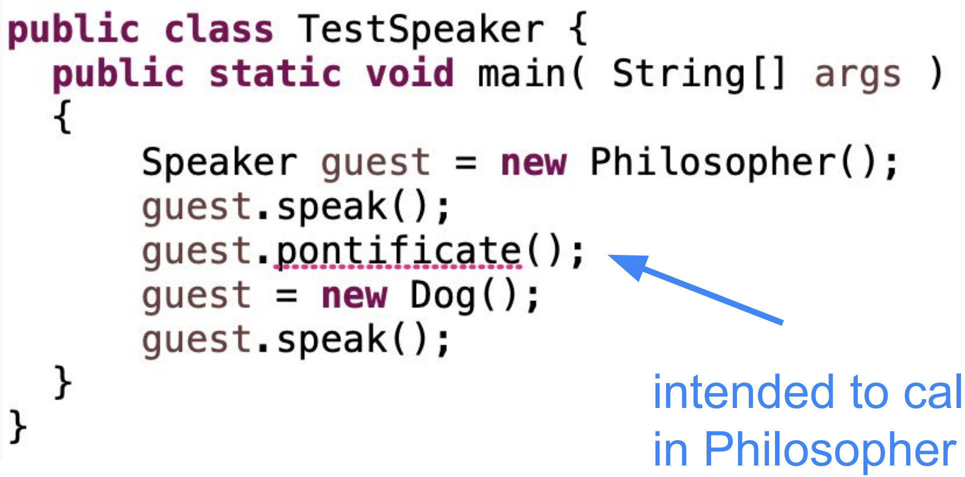
// from Philosopher class

// from Dog class

Dynamic binding at play (Rule 3)

Speaker Example (cont.)

- Revised Test class:



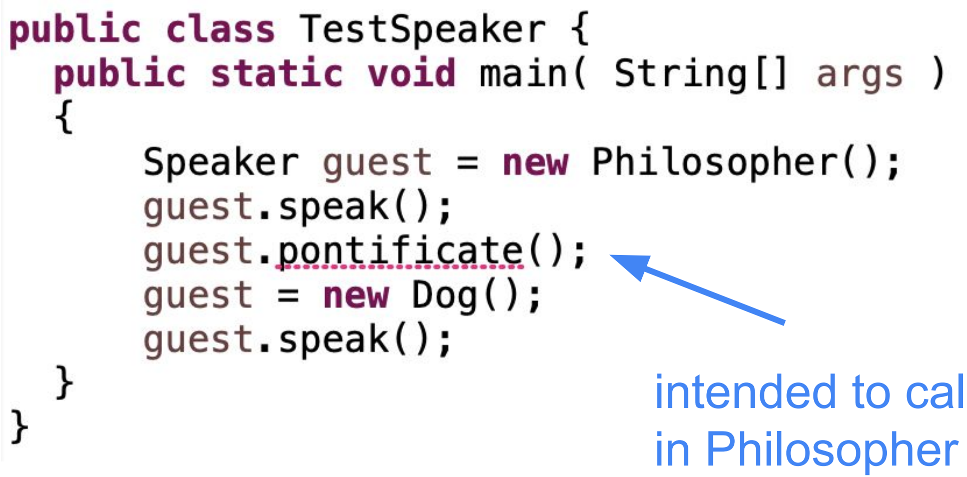
```
public class TestSpeaker {
  public static void main( String[] args )
  {
      Speaker guest = new Philosopher();
      guest.speak();
      guest.pontificate();
      guest = new Dog();
      guest.speak();
  }
}
```

intended to call extra method in Philosopher class

- is this allowed?

Speaker Example (cont.)

- Revised Test class:



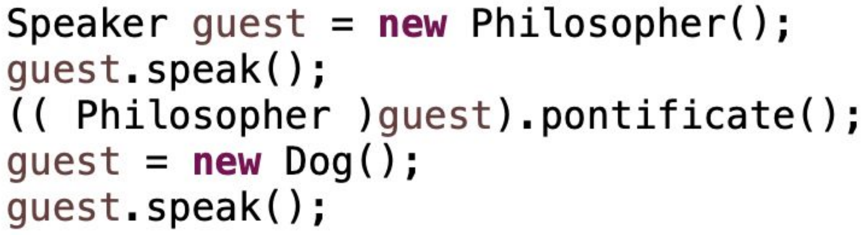
```
public class TestSpeaker {
  public static void main( String[] args )
  {
      Speaker guest = new Philosopher();
      guest.speak();
      guest.pontificate();
      guest = new Dog();
      guest.speak();
  }
}
```

intended to call extra method in Philosopher class

- is this allowed?
  No! "Rule 2"

Speaker Example (cont.)

- Solution? Tell compiler guest really is a Philosopher object:



```java
Speaker guest = new Philosopher();
guest.speak();
(( Philosopher )guest).pontificate();
guest = new Dog();
guest.speak();
```

Stepping Back

- When we first introduce shapes to children, there are typically three shapes we tell them about: square, circle, and triangle. From these basic shapes, other shapes are derived.
- Let's say we have an app to teach children about shapes. Now, consider the classes Circle and Oval that share common attributes and methods.

- Would you relate these two via an **inheritance** relationship or an **interface** relationship? Why?

Stepping Back

- When we first introduce shapes to children, there are typically three shapes we tell them about: square, circle, and triangle. From these basic shapes, other shapes are derived.
- Let's say we have an app to teach children about shapes. Now, consider the classes Circle and Oval that share common attributes and methods.

- Would you relate these two via an **inheritance** relationship or an **interface** relationship? Why?

An Oval is a kind of Circle: Inheritance

Stepping Back (2)

- Imagine a game in which players can attack other players' game elements, such as balloons, mirrors, etc. Each type of element belongs to a different class. E.g., there is a Balloon class, where many balloon objects of different colors are created, there is a Mirror class, etc.
- These elements have a break() method in common so when the other player taps it, it breaks. The elements also have a isBroken() method that returns a boolean representing whether it is broken or not.

- Should the game elements Balloon, Mirror, etc. be related via **inheritance** or **interface**? Why?

Stepping Back (2)

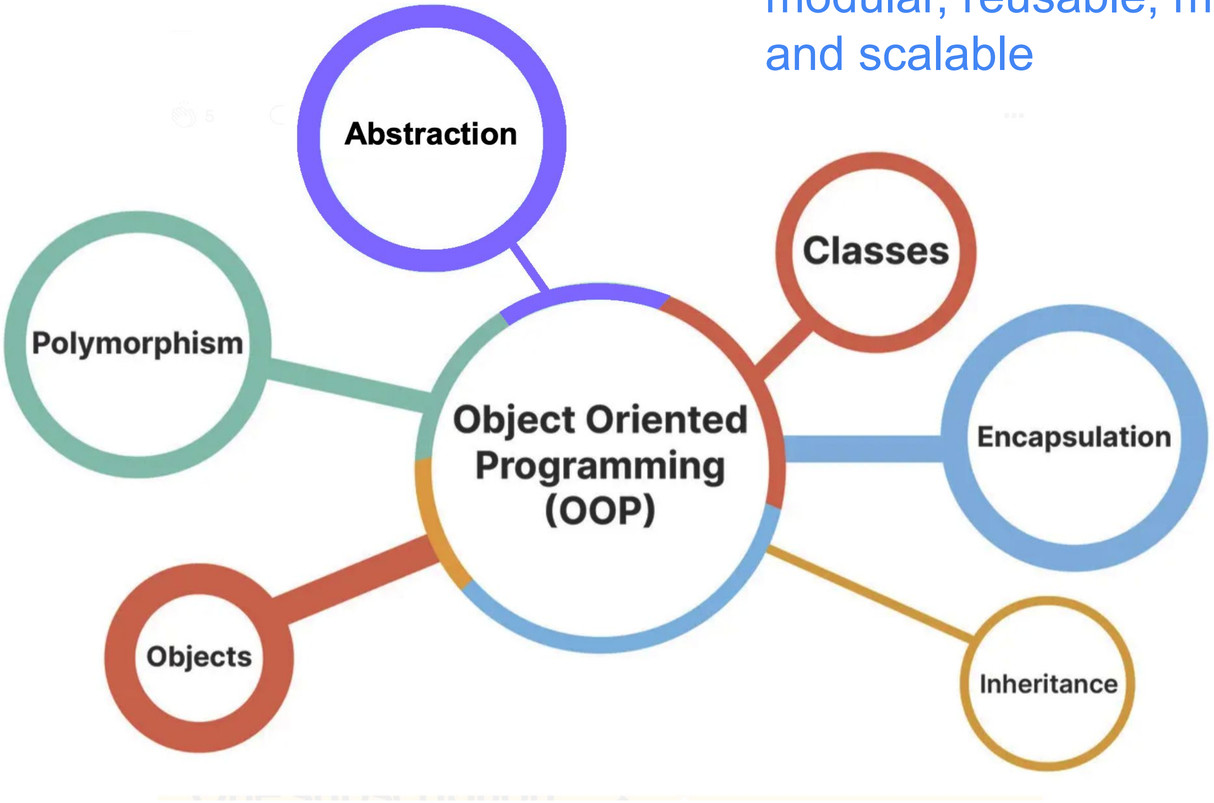
- Imagine a game in which players can attack other players' game elements, such as balloons, mirrors, etc. Each type of element belongs to a different class. E.g., there is a Balloon class, where many balloon objects of different colors are created, there is a Mirror class, etc.
- These elements have a break() method in common so when the other player taps it, it breaks. The elements also have a isBroken() method that returns a boolean representing whether it is broken or not.

- Should the game elements Balloon, Mirror, etc. be related via **inheritance** or **interface**? Why?

  Mirrors and Balloons don't seem semantically related: Interface

End of OOP Concepts

OOP is for software design, making programs more organized, modular, reusable, maintainable, and scalable



Review: Overloading versus Overriding

- How are they different?
    - **Overriding:**
    - **Overloading:**

Review: Overloading versus Overriding

- How are they different?
    - **Overriding:**
    - Happens between parent and children classes
    - Methods have same signatures
    - **Overloading:**
    - Happens in the same class
    - Methods have different signatures

Review: Inheritance versus Interfaces

- How are they different?
  - **Inheritance:**
  - **Interface:**

Review: Inheritance versus Interfaces

- How are they different?
  - **Inheritance:**
  - When a class is-a another class
  - Can only extends one class
  - Has attributes and methods
  - **Interface:**
  - Classes do not have to be semantically related
  - Can implements multiple classes
  - Has constants and abstract methods mostly, Java 8/9 allows for additional modifiers: default, static, private, and private static

Review: Abstract Classes versus Interfaces

- How are they different?
    - **Abstract Class:**
    - **Interface:**

Review: Abstract Classes versus Interfaces

- How are they different?
  - **Abstract Class:**
  - Can have attributes
  - Methods may be abstract or not
  - Methods can have different visibility
  - **Interface:**
  - No attributes
  - Methods are mostly abstract, Java 8+ allows for default and static
  - Methods are mostly public, Java 9+ allows for private and private static