MACQUARIE
University

WCOM125/COMP125

Fundamentals of Computer Science

Classes and Objects - 2

Gaurav Gupta

July 11, 2018

Contents

1 Overview **4**

2 this keyword **5**

3 Comparing objects (compareTo method) **7**

3.1 **compareTo** in action 9

4 Multi-criteria comparison **12**

5 Unit testing methods of a class **14**

6 JUnit testing **15**

6.0.1 Corrected version by looking at JUnit failure 16

7 Static members vs. Instance members **17**

7.1 Static member access 17

7.2 Instance member access 18

7.3 Typical static methods 19

8 Manipulating references **20**

8.1 Shallow copy 20

8.2 Deep copy 22

8.3 Copy constructor 24

8.3.1 Copy constructor call 25

9 Sample solutions for exercises **26**

List of exercises

Exercise 1 6
Exercise 2 10
Exercise 3 13
Exercise 4 18
Exercise 5 21
Exercise 6 23
Exercise 7 24
Exercise 8 25

Section 1.

Overview

We explore classes and objects in more detail on the following points:

- **this** keyword
- Comparing objects
- Unit testing methods of a class
- **static** vs instance members
- Manipulating references

Section 2.

this keyword

Consider the following class definition,

```
1 public class Circle {
2   private double radius;
3   public void setRadius(double radius) {
4     radius = Math.abs(radius);
5   }
6   //assume getter is also defined
7 }
```

Because the parameter and instance variable have the same name (**radius**), it is not clear which one is affected in the assignment statement on line 4 above.

Java provides a keyword **this** that refers to the calling object and gives access to its instance variables and methods. Line 4 now shows that the instance variable **radius**

```
1 public class Circle {
2   private double radius;
3   public void setRadius(double radius) {
4     this.radius = Math.abs(radius);
5   }
6   //assume getter is also defined
7 }
```

on line 2 will be affected by the assignment statement. As you can see, the **Math.abs** method is using the parameter variable **radius** .

Exercise 1

Disambiguate assignment operation

Get rid of the ambiguity in the `setSide` method

```
1 public class Square {
2   private double side;
3   public void setSide(double side) {
4     side = Math.max(0, side);
5   }
6   //assume getter is also defined
7 }
```

Write your answer here

(SOLUTION 1)

Section 3.

Comparing objects (compareTo method)

The method **compareTo** provides a way to define an order on two objects (say a and b). The method is called on object a and the object b is passed as a parameter, the result indicating how a compares to b.

⊑ *NOTE* ⊒

You can access **private** instance members of the object passed inside a method directly, as long as the object is of the same class as the one the method is in.

Consider the following method in the **Circle** class,

```
1  public class Circle {
2    // other methods and instance members
3
4    /*
5    comparison criterion: radius.
6    return 1 if the calling object is ``more'' than other
7    -1 if its ``less''
8     0 if they are ``equal''
9     */
10   public int compareTo(Circle other) {
11       if(this.radius > other.radius)
12             return 1;
13       if(this.radius < other.radius)
14         return -1;
15       return 0;
16   }
17 }
```

3.1 compareTo in action

```
1  Circle myCircle = new Circle(12);
2  Circle yourCircle = new Circle(18);
3  Circle ourCircle = new Circle(7);
4  Circle theirCircle = new Circle(12);
5
6  int s1 = myCircle.compareTo(yourCircle); //-1
7  int s2 = myCircle.compareTo(ourCircle); //1
8  int s3 = myCircle.compareTo(theirCircle); //0
9  int s4 = theirCircle.compareTo(ourCircle); // ??
10 int s5 = ourCircle.compareTo(yourCircle); // ??
11 int s6 = yourCircle.compareTo(yourCircle); // ??
```

Exercise 2

Implement compareTo method

Add a compareTo method in class Square that returns,

- 1 if calling object's area is more than parameter object's area
- -1 if calling object's area is less than parameter object's area
- 0 if calling object's area is equal to parameter object's area

```
1 public class Square {
2   private double side;
3   //assume getter and setter
4   public double area() {
5     return side * side;
6   }
7 }
```

Write your answer here

(SOLUTION 2)

Section 4.

Multi-criteria comparison

What happens if there are multiple levels of comparison criteria? For example, if we compare two rectangles based on area, but they have the same area, we then compare them on perimeter, and if even that's the same, return 0.

Exercise 3

Implement multi-criteria compareTo method

Add a compareTo method in class Rectangle that returns,

- 1 if calling object's area is more than parameter object's area, or if they have the same area, but calling object's perimeter is more than parameter object's perimeter.
- -1 if calling object's area is less than parameter object's area, or if they have the same area, but calling object's perimeter is less than parameter object's perimeter.
- 0 if calling object's area is equal to parameter object's area and calling object's perimeter is equal to parameter object's perimeter.

```
1 public class Rectangle {
2   private double width, height;
3   //assume getters and setters
4   public double area() { return width * height; }
5   public double perimeter() { return 2*(width + height); }
6 }
```

Write your answer here

(SOLUTION 3)

Section 5.

Unit testing methods of a class

An example class with a method that will not produce the expected result in all situations.

```
public class Line {
  private int x1, y1, x2, y2;
  //other parts
  public double getMidX() {
    return x1+x2/2; //i know.
  }
}
```

Section 6.

JUnit testing

JUnit provides a framework for testing individual methods. It works based on following assertions:

1. **assertEquals(expected double, double returned by method, tolerance)** :

   passes if,

   ```
   Math.abs(expected double - double returned by method) <= tolerance
   ```

2. **assertEquals(expected integer, integer returned by method)** : passes if,

   ```
   expected integer == integer returned by method
   ```

3. **assertTrue(boolean value)** : passes if parameter is **true** .
4. **assertTrue(boolean value)** : passes if parameter is **false** .
5. **assertNull(Object/array)** : passes if parameter is **null** .
6. **assertNotNull(Object/array)** : passes if parameter is **NOT** **null** .
7. More assertions can be found at:
   http://junit.org/junit4/javadoc/latest/org/junit/Assert.html

```
1
2  import static org.junit.Assert.*;
3  import org.junit.Test;
4  public class LineTest {
5    @Test
6    public void testGetMidX() {
7      Line a = new Line(0, 10, 8, 12);
8      assertEquals(4, a.getMidX(), 0.001); //passes
9      Line b = new Line(5, 10, 6, 12);
10     assertEquals(5.5, b.getMidX(), 0.001); //fails
11 }
```

6.0.1 Corrected version by looking at JUnit failure

```
1  public class Line {
2    private int x1, y1, x2, y2;
3    //other parts
4    public double getMidX() {
5      return (x1+x2)/2.0;
6    }
7  }
```

Section 7.

Static members vs. Instance members

Static members are the ones that are accessed in the context of a class. You don't need to create objects of that class in order to access them. For example, consider the number of eyes **dinosaurs** have. Note, we didn't say, how many eyes does Dorothy the dinosaur, or Tyrone the dinosaur have.

We don't even need any dinosaur to be alive to answer that question, since it's an attribute of the *collective* (the class) rather than an *individual* (an object).

On the other hand, the variables **weight, height** are different for each dinosaur that was there. Hence, they are *instance variables*. Similarly, the body mass index (defined as weight divided by square of height) is an *instance method*, that must be called on an *individual* (the object), not the *collective* (the class)

Consider the following class:

```
class Dinosaur {
  public static int nEyes = 2;
  public static int nEars = 2;
  public static double eyesToEarsRatio() {
        return nEyes * 1.0 / nEars;
  }

  private double weight, height;
  //assume getters, setters
  //assume parameterized constructors
  public double bodyMassIndex() {
    return weight/(height*height);
  }
}
```

7.1 Static member access

```
int eyeCount = Dinosaur.nEyes;
double prop = Dinosaur.eyesToEarsRatio();
```

7.2 Instance member access

```
Dinosaur dorothy = new Dinosaur(450,3.6);
double w = dorothy.getWeight();
double h = dorothy.getHeight();
double bmi = dorothy.bodyMassIndex();
```

Note that in this example the instance variables are private, otherwise (if they were public), they could be accessed in the same way (on a calling object).

Exercise 4

Accessing static and instance members

Display the members `data1, data2, method1, method2` of class `MyClass` (or object `obj` of class `MyClass`) in the client code.

```
public class MyClass {
        public int data1;
        public static int data2;
        public static int method1() { return data2; }
        public int method2() { return data1; }
}

class Client {
        public static void main(String[] args) {
                MyClass obj = new MyClass();
                //your code
        }
}
```

Write your answer here

(SOLUTION 4)

7.3 Typical static methods

Typically, methods are classified as static, if the values they operate on are passed to the methods. For example:

```
class StringService {
  public static char lastItem(String s) {
    if(s == null || s.length() == 0)
      return (char)0;
    return s.charAt(s.length() - 1);
  }
 }
```

```
class IntegerService {
        public static int getNumberOfDigits(int num) {
                if(num == 0)
                        return 0;
                // Remove any sign from number,
                // add it to an empty string,
                // return its length.
                return (Math.abs(num) + "").length();
        }
}
```

Section 8.

Manipulating references

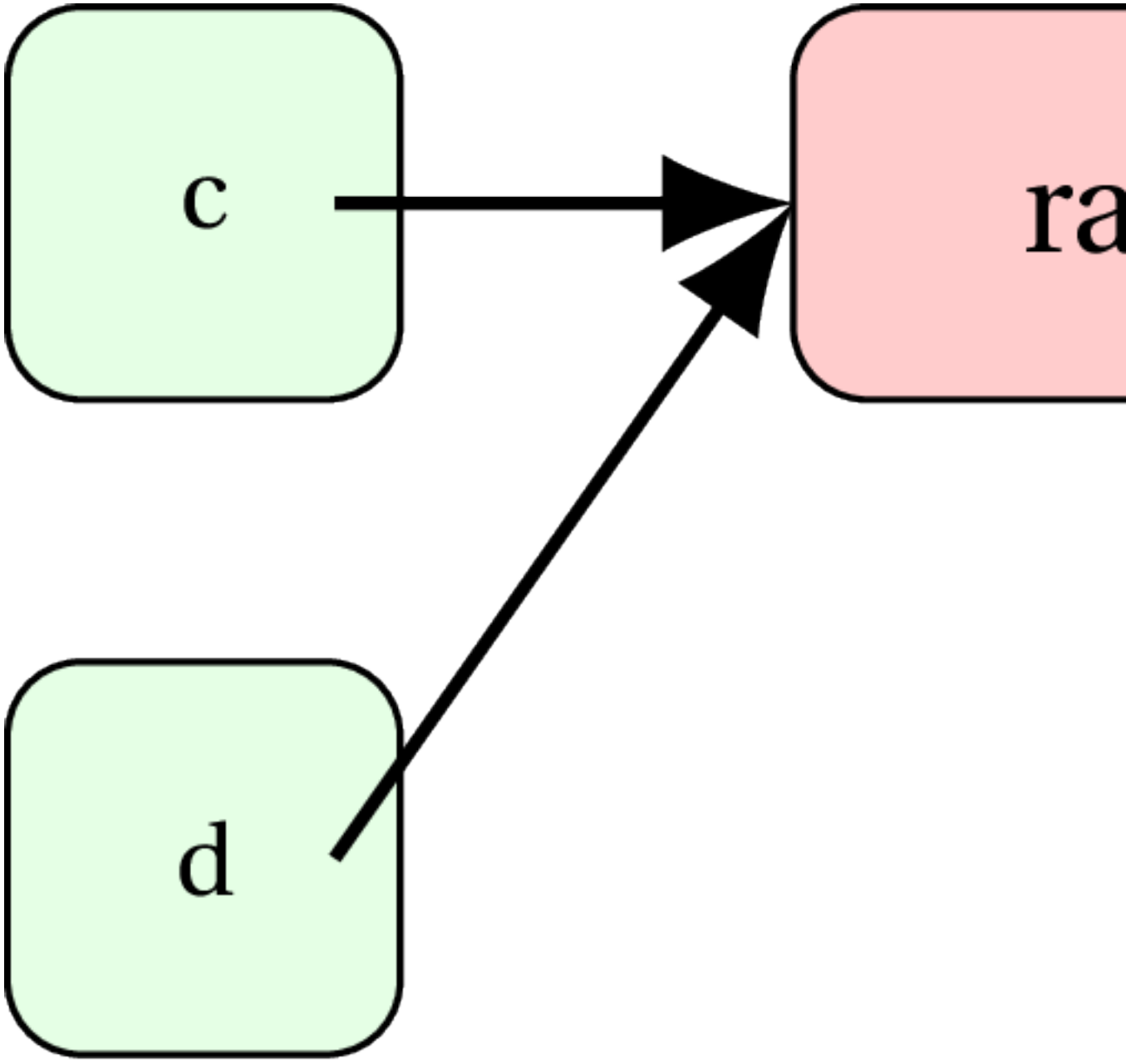
8.1 Shallow copy

Consider the following class definition and client code:

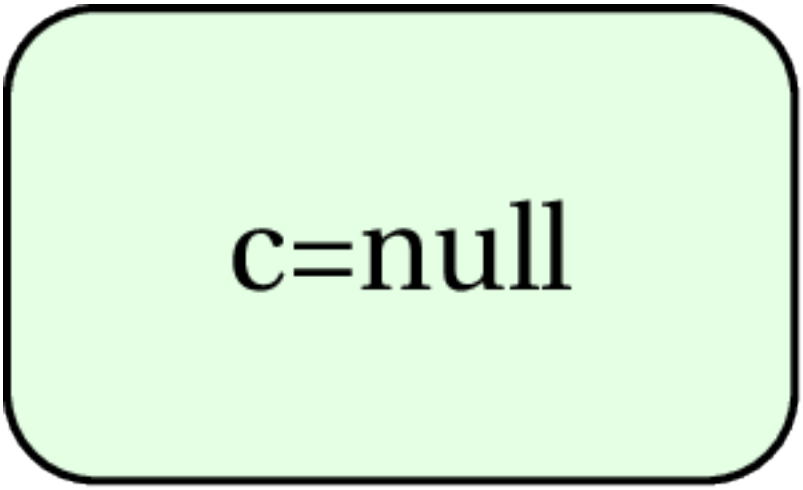
```
1  public class Circle {
2        private double radius;
3        //other parts
4  }
```

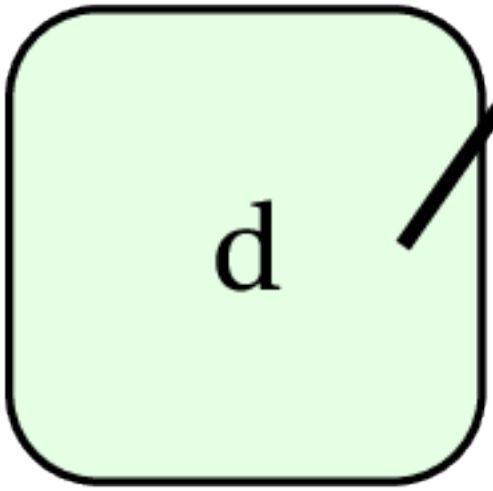
```
1  Circle c = new Circle(30);
2  Circle d = c;
3  //c and d refer to same
4  //instance variable space
5  d.setRadius(50);
6  System.out.println(c.getRadius()); //??
```

Object **d** holds reference to the same physical object as **c**. Therefore, when the **radius** of object **d** is modified, **radius** of object **c** also gets updated.


c
ra
d

Let's say, **c** changes to **null**.


c=null


d

Exercise 5

Create a shallow copy

Create a shallow copy of `myObj` into `yourObj`. Increase the radius of `yourObj` by 2. What is the new radius of `myObj`?

```
1  public class Circle {
2          private double radius;
3          //assume getters, setters
4          // and constructors defined
5  }
6
7  class Client {
8          public static void main(String[] args) {
9                  Circle myObj = new Circle(1.5);
10                 //your code
11         }
12 }
```

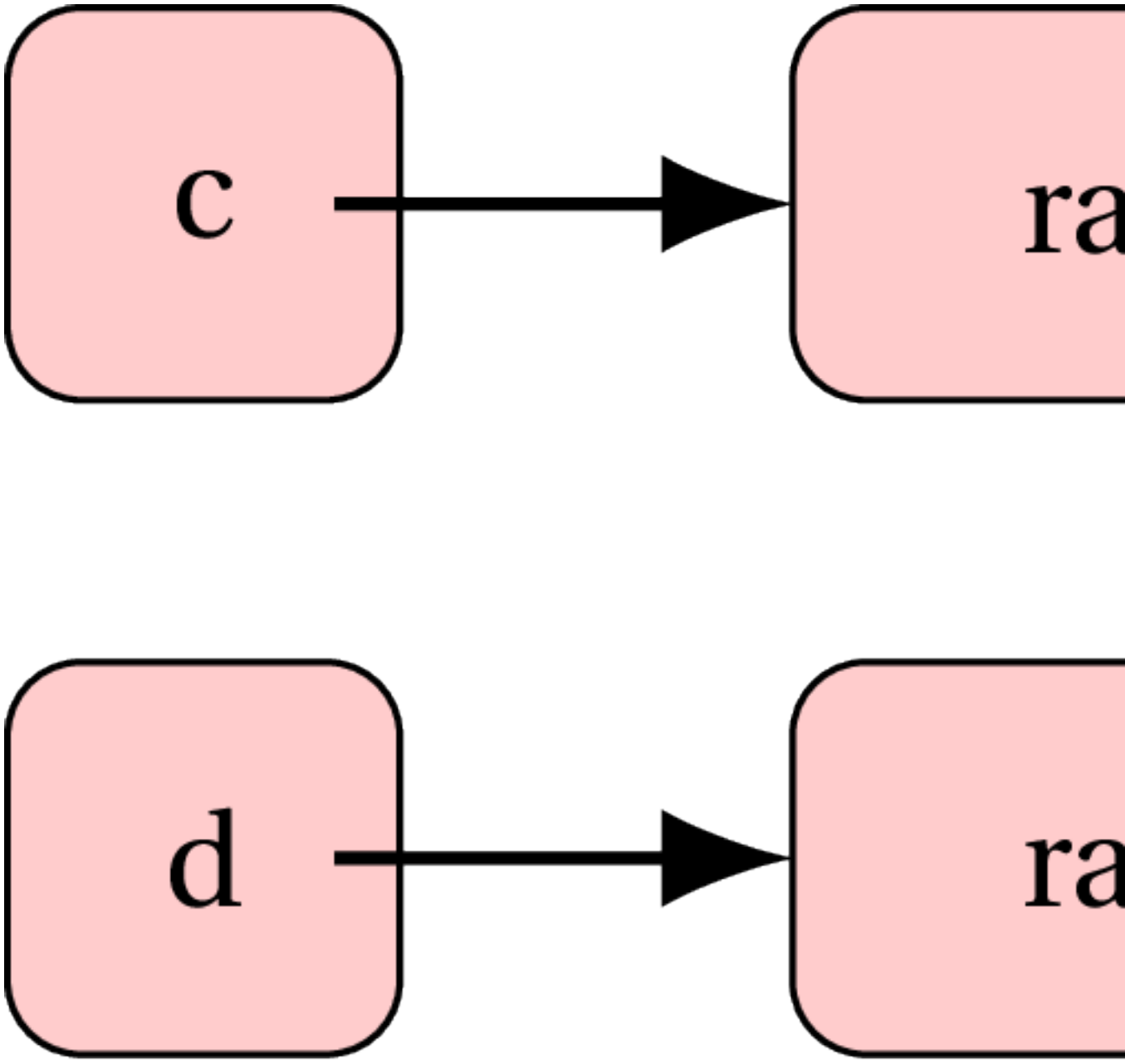
Write your answer here

(SOLUTION 5)

8.2 Deep copy

Object **d** is a clone of object **c**. Object **c** and **d** are independent objects. Modifying one does not alter the other.

```
1  Circle c = new Circle(30);
2  Circle d = new Circle();
3  d.setRadius(c.getRadius());
4  //c's radius copied into d's radius
5  System.out.println(c.getRadius()); //??
```



Exercise 6

Create a deep copy

Create a deep copy of myObj into yourObj . Increase the radius of yourObj by 2. What is the new radius of myObj ?

```
1  public class Circle {
2          private double radius;
3          //assume getters, setters
4          // and constructors defined
5  }
6
7  class Client {
8          public static void main(String[] args) {
9                  Circle myObj = new Circle(1.5);
10                 //your code
11         }
12 }
```

Write your answer here

(SOLUTION 6)

8.3 Copy constructor

```java
public class Circle {
	private double radius;
	//setters, getters

	public Circle(double radius) {
		setRadius(radius);
	}

	public Circle(Circle other) {
		setRadius(other.radius);
	}
}
```

Exercise 7

Define a copy constructor

Define a copy constructor in class Rectangle .

```java
public class Rectangle {
	private double width, height;
	//assume getters, setters defined
	//define copy constructor here
}
```

Write your answer here

(SOLUTION 7)

8.3.1 Copy constructor call

```
Circle c = new Circle(30);
Circle d = new Circle(c);
```

Exercise 8

Call a copy constructor

Deep copy myObj into yourObj using the copy constructor defined in class Square .

```
public class Square {
        private double side;
        //assume getters, setters defined
        public Square(Square other) {
                setSide(other.side);
        }
}

class Client {
        public static void main(String[] args) {
                Square myObj = new Square(2.4);
                //your code
        }
}
```

Write your answer here

(SOLUTION 8)

Section 9.

Sample solutions for exercises

Solution: Exercise 1

```
public class Square {
  private double side;
  public void setSide(double side) {
    this.side = Math.max(0, side);
  }
  //assume getter is also defined
}
```

Solution: Exercise 2

```
public int compareTo(Square other) {
        if(area() > other.area())
                return 1;
        if(area() < other.area())
                return -1;
        //in all other cases:
        return 0;
}
```

Solution: Exercise 3

```
public int compareTo(Rectangle other) {
        //first key comparison
        if(area() > other.area())
                return 1;
        if(area() < other.area())
                return -1;

```

```
        //second key comparison
        if(perimeter() > other.perimeter())
                return 1;
        if(perimeter() < other.perimeter())
                return -1;

        //still nothing?
        return 0;
}
```

Solution: Exercise 4

```
//instance variable called on object
System.out.println(obj.data1);

//static variable called on class
System.out.println(MyClass.data2);

//static method called on class
System.out.println(MyClass.method1());

//instance method called on object
System.out.println(obj.method2());
```

Solution: Exercise 5

```
Circle yourObj = myObj; //shallow copy
youObj.setRadius(yourObj.getRadius() + 2);
System.out.println(myObj.getRadius()); //will be 3.5
```

Solution: Exercise 6

```
//create a brand-spanking new object in memory
Circle yourObj = new Circle();
//get the value for radius from myObj
yourObj.setRadius(myObj.getRadius());
youObj.setRadius(yourObj.getRadius() + 2);
```

```
System.out.println(myObj.getRadius()); //will still be 1.5
```

Solution: Exercise 7

```
public Rectangle(Rectangle source) {
        setWidth(source.getWidth());
        setHeight(source.getHeight());
}
```

Solution: Exercise 8

```
Square yourObj = new Square(myObj);
```