CSCI-142 Exam 1 Review

Presented by the RIT Computer Science Community

September 21, 2017

http://csc.cs.rit.edu

1. Provide a detailed explanation of what the following code does:

```
1 public boolean checkString(String a, String b) {
2     return a == b;
3 }
```

Given the two strings *a* and *b*, `checkString` returns `true` if both string objects have the same memory address. That means they are both references to the same thing. The function does not check to see if the two strings have the same sequence of characters. It is possible for two different strings to have the same value ("hellos") but with different memory addresses.

By default, Object's `equals` method compares memory addresses, but it can be extended by subclasses to check the objects' attributes. In Java, it is not possible to override how `==` works.

To check if the contents of the strings are the same, use the `equals` method, defined in the `Object` class and overridden in the `String` class. For example, `"yes".equals("yes")`.

2. If an instance variable is declared with `protected` access, who can access it?

The class that the variable is defined in, any subclasses of that class, and any classes in the same package as that class.

3. What is the difference between overriding and overloading methods? Give a example situation where each should be used.

Overriding: method with the exact same method declaration as a method in a superclass; overwrites and/or extends the functionality of the superclass's method.

Overloading: 2 or more methods with the same name that either:

(a) have a different number of parameters
OR
(b) have parameters of different types

Override a method when you want to *replace* the implementation of a superclass's method. Overload a method (such as a constructor) when you want to have more than one implementation of a method for different circumstances.

4. Explain the differences between:

(a) `class` vs `object`

A class defines methods and fields—it can be viewed as a template. An object is an instance of a class. Think of classes as molds and objects as individual things created by those molds.

(b) constant vs non-constant field (variable). Declare a constant.

A constant field cannot be changed during run time (it may be set *once*).

```
public final int NUM_PEOPLE_WHO_LIKE_JAVA = 1;
```

Note that marking mutable types `final` only prevents them from being reassigned; they can still call methods that mutate themselves. For example, you can declare a `final ArrayList`, but this does not prevent you from modifying the contents of it, such as through the use of the `add(T toInsert)` method.

(c) final vs non-final method.

A final method can't be overridden by a subclass.

(d) class vs instance variable. Declare variables of both types.

Class (static) variables are associated with the class, not to instances of that class. All instances of a class access the same static variable. However, instance (non-static) variables are unique to each instance.

```
class Types {
    public static int im_a_class_var = 1;
    public int im_an_instance_var = 2;
}
```

5. Define polymorphism and explain a situation in which you would use it.

   Polymorphism: when objects of various types define a common interface of operations (in other words, objects of different subclasses appear and act as a shared superclass). References to and collections of a super class may hold instances of subclasses. Methods invoked on these objects determine the correct (type-specific) behavior at runtime. That is, an object instantiated from a subclass will use the subclass's implementation of a method, even if it is stored in a reference to its superclass. This allows different subclasses to be plugged in and used by superclass variable.

   Example:

```
ArrayList<Shape> shapes = new ArrayList<Shape>();

// Square with side length of 1
shapes.add(new Square(1.0));

// Circle with radius of 9
shapes.add(new Circle(9.0));

// Triangle with base of 4 and height of 5
shapes.add(new Triangle(4.0, 5.0));

// Outputs the correct area of each shape
for( int i = 0; i < shapes.size(); i++ ){
    double area = shapes.get(i).getArea();
    System.out.println(area);
}
```

6. What is the difference between an abstract class and an interface? Why might you want to use an interface over an abstract class?

   Abstract classes can have fields, whereas interfaces cannot. Classes can implement multiple interfaces, but can only extend a single class. Because of this, if you want a class to be able to be treated as two or more unrelated types, make those types interfaces. In Java 7 and earlier, interfaces can only have method signatures, without any implementation. Abstract classes can have a mix of abstract methods and methods with an implementation. This allows abstract classes to provide default behavior for their subclasses, as well as common base behavior that subclasses can extend from.

7. Describe the difference between `Comparable` and `Comparator`.

   Both are interfaces. `Comparable` specifies a method called `compareTo()` that takes one argument: the object to compare with the one on which the method was called. `Comparator` specifies a method called `compare()` that instead takes two arguments and compares those objects to each other.

   Classes implementing `Comparator` are classes whose sole purpose is to define a comparing mechanism for a particular type. There can be multiple `Comparator`s for a type, if that type has multiple ways to be compared. If there is a single natural way to compare instances of a particular class, that class can implement the `Comparable` interface and define the comparison itself. This removes the need to store and pass the comparison mechanism separate from the objects it is comparing. A Comparator can also override a class's natural ordering.

8. What is the Java Collections Framework?

   A unified set of interfaces, algorithms and concrete implementations provided by Java to support collections of objects.

9. What gets printed by the following code?

```
1  public class Class1 {
2      public Class1() {
3          System.out.println( "Class1()" );
4      }
5      public void print1() {
6          System.out.println( "Class1.print1()" );
7      }
8      public void print2() {
9          System.out.println("Class1.print2()" );
10     }
11 }
12
13 public class Class2 extends Class1 {
14     public Class2() {
15         System.out.println( "Class2()" );
16     }
17     public void print1() {
18         System.out.println("Class2.print1()");
19     }
20 }
21
22 public class Class3 extends Class1{
23     private Class2 class2;
24     public Class3() {
25         System.out.println( "Class3()" );
26         class2 = new Class2();
27     }
28     public void print1() {
29         class2.print1();
30     }
31     public void print2(){
32         System.out.println("Class3.print2()");
33         super.print2();
34     }
35 }
36
37 public class TestClass {
38     public static void main( String[] args ) {
39         Class1 c1 = new Class2();
40         c1.print1();
41         c1.print2();
42         System.out.println();
43         Class1 c2 = new Class3();
44         c2.print1();
45         c2.print2();
46     }
47 }
```

```
Class1()
Class2()
Class2.print1()
Class1.print2()
```

```
Class1()
Class3()
Class1()
Class2()
Class2.print1()
Class3.print2()
Class1.print2()
```

10. Find at least 3 errors related to inheritance and interfaces in the following code:

```
1  public interface Vehicle {
2      public int getSpeed();
3      public void accelerate(int speed_inc);
4      public void brake(int speed_dec);
5  }
6  public class Car implements Vehicle {
7      private int speed;
8      public Car(int initialSpeed){
9          this.speed = initialSpeed;
10     }
11     public int getSpeed(){
12         return speed;
13     }
14     public int accelerate(int speed_inc) {
15         speed += speed_inc;
16         return speed;
17     }
18     public int brake(int speed_dec) {
19         speed -= speed_dec;
20         return speed;
21     }
22 }
23
24 public class Toyota extends Car {
25     public long getSpeed(){
26         return speed;
27     }
28 }
29 public class Truck implements Vehicle {
30     public void accelerate(int speed_inc) {
31         super.accelerate(speed_inc/2);
32     }
33     public void brake(int speed_dec) {
34         super.brake(speed_dec/2);
35     }
36 }
37 public class demolitonDerby {
38     public static void main(String[] args) {
39         Vehicle prius, mack, impreza;
40         prius = new Toyota();
41         mack = new Truck();
42         impreza = new Car();
43
44         impreza.accelerate(5);
45         prius.brake(2);
46         prius.accelerate(impreza.getSpeed());
47         mack.accelerate(5);
48     }
49 }
```

Line #	Error
14, 18	Returns `int`, should return `void`.
25	`getSpeed`'s return type (`long`) differs from that of the `getSpeed` from `Car` or `Vehicle`.
26	`speed` was declared `private` in `Car`, and so is inaccessible from `Toyota`.
29	`Truck` implements `Vehicle`, but does not have `getSpeed`.
31, 34	Truck doesn't extend a class with `accelerate` or `brake` methods, so it can't call those methods on `super`.
42	the constructor for `Car` requires an `int` parameter

11. Convert the following code to use generics.

```
interface StringCondition {
    boolean checkString(String s);
}

interface IntegerCondition {
    boolean checkInteger(Integer i);
}

class StringContainer {
    ArrayList<String> values;

    // Add and other methods are defined correctly here...

    String getFirstWhereHolds(StringCondition condition) {
        for (String s : values) {
            if (condition.checkString(s))
                return s;
        }
        return null;
    }
}

class IntegerContainer {
    ArrayList<Integer> values;

    // Add and other methods are defined correctly here...

    Integer getFirstWhereHolds(IntegerCondition condition) {
        for (Integer i : values) {
            if (condition.checkInteger(i))
                return i;
        }
        return null;
    }
}
```

```
interface Condition<E> {
    boolean check(E value);
}

public class Container<E> {
    ArrayList<E> values;
    E getFirstWhereHolds(Condition<E> condition){
        for (E v : values) {
            if (condition.check(v))
                return v;
        }
        return null;
    }
}
```