Solutions to Exercise Set 2

Prepared by: Neeraj, Julien, Mansoor

February 24, 2009

Solutions to Exercises from the Book

Exercise R-2.11

Read it.
Ship it.
Buy it.
Read it.
Box it.
Read it.

Exercise C-2.1

Inheritance in Java allows for specialized classes to be built from generic classes. Because of this progression from generic to specialized in the class hierarchy, there can never be a circular pattern of inheritance. In other words, there cannot be a superclass A and derived classes B and C such that B extends A, then C extends B, and finally A extends C. Such a cycle is impossible because A is the generic superclass from which C is eventually extended, thus it is impossible from A to extend C, for this would mean A is extending itself. Therefore, there can never occur a circular relationship which would cause an infinite loop in the dynamic dispatch.

Solutions to Questions on Interfaces

1. Yes. It is not only correct, but also is recommended because it allows the implementation class to change without having to change the method signature and/or method implementation.

   See the following example.

```
// TestFace.java
interface IFace{
  void drawFace();
}

class RealFace implements IFace {
  public void drawFace(){
    System.out.println("I'm drawing a beautiful face");
  }
}

public class TestFace {
  // This code works on an interface, and know nothing about
  // implementation underneath.
  public static void callIFace(IFace face) {
    face.drawFace();
  }
  public static void main(String[] args) {
    // If a FaceFactory was used here instead, you would not
    // need to make a specific reference to class RealFace.  The
    // class name could instead be read from an external
    // properties file, resulting in code that is much more
    // scalable.
    IFace face = new RealFace();
    // Pass the interface to the method.  We could technically
    // call face.drawFace() directly, but here we are using the
    // callIFace method to illustrate that an interface can be
    // passed as a method parameter.
    callIFace(face);
  }
}
```

2. Yes, we can define a variable in an interface. The variables defined in an interface are implicitly static and final. (See the footnote 2 of lecture 16, page 1: http://www.cim.mcgill.ca/%7Elanger/250/lecture16.pdf)

3. Public

4. An abstract class can have instance methods that implement a default behavior. An interface can only declare constants and instance methods, but cannot implement default behavior and all methods are implicitly abstract. An interface has all public members and no implementation. An abstract class is a class which may have the usual flavors of class members (private, protected, etc.), but has some abstract methods.

5. Choice D is the correct answer. The code does not compile because the method g() is not implemented as public in the class Test. Choice A is incorrect because there is no mistake in redefining a superclass's private method in the subclass, though this would not be overriding. Choice B is incorrect because SuperTest is declared abstract and hence does not need to implement the methods from the interface implemented by it. Since the code does not actually compile, choice C is also incorrect.

Solutions to Questions on Abstract Classes

1. Yes, as long as you do not instantiate the abstract class or call its abstract methods. Abstract class cannot be instantiated. However, if the main method only calls static method of the class, you can run it just like you run any C program.

   Consider the following example:

```
public abstract class A {
  static public int sum(int a, int b) {
      return a + b;
  }

  abstract public void aMethod();
```

```
        // abstract static combination not allowed
        // abstract public static void aStaticMethod();

        public static void main(String args[]) {
            System.out.println("Sum of 3 and 5 is : " + sum(3, 5));

            // The following code will not be compilable
            // A a = new A();
            // a.aMethod();
        }
      }
```

2. (d)

3. (a). All abstract and interface methods are implicitly non-static, abstract and public. An abstract method cannot be static, final since abstract methods MUST be overridden by concrete implementations, and static and final methods cannot be overridden by definition.

   (a). We cannot have abstract static methods. You are not required to know for the quiz why this is the case, but for those of you who are curious, consider the following code snippet:

```
public class TestClass {
     public static void main(String[] args){
          String s1 = "test1";
          String s2 = "test2";
          Comparable c1 = s1;
          Comparable c2 = s2;
          System.out.println(c1.compareTo(c2));
     }
}
```

   In the code snippet above, we are calling c1.compareTo(). But at compile time, the compiler does not know what concrete class is represented by c1. The compiler knows that c1 has an implementation of the compareTo() method (since it implements the Comparable interface), but does not know which implementation of the compareTo() method is being used until runtime. This is called "dynamic binding".

   When a method is declared as static, then "static binding" is used instead. With static binding, the compiler expects to know exactly which implementation of the method is being used at compile time, but this is impossible if the method is declared as abstract and implemented differently by different classes.

4. (b). Consider the following valid code snippet:

```
abstract class Demo1{
     public static void demo(){
          System.out.println("Static method in abstract class");
     }
     abstract void printMessage();
}
```

5. (b). You can create an example similar to 5 to show this.

6. (b). You can create an example similar to 5 to show this.

7. (a). Final Class cannot be extended. But abstract methods must be overridden which is not possible as the class is Final and cannot be extended.

Solutions to Questions on Generics

Question 1:

1. Non-generic LinkedLists can be used with any object, which means two things: We have no guarantee about the type of object it can contain. After we get an object from the list we must cast it to the type we expect the object to be.

2. No, because it's possible to define a LinkedList<Object>, which is exactly equivalent to the non-generic LinkedList.

3. Use LinkedList<Object>.

Question 2:

1. (c)

2. (b)

3. (c)

4. No