

## Chapter 8 Inheritance and Polymorphism

1. The printout is  
The default constructor of A is invoked
2. The default constructor of B attempts to invoke the default of constructor of A, but class A's default constructor is not defined.
3. The following lines are erroneous:

```
{  
    radius = radius; // Must use this.radius = radius  
}
```

```
class Cylinder extends Circle (missing extends)
```

```
{  
    Circle(radius); // Must use super(radius)  
    length = length; // Must use this.length = length  
}
```

```
public double findArea()  
{  
    return findArea()*length; // super.findArea()  
}
```

```
{  
    radius = radius; // Must use this.radius = radius  
}
```

```
class Cylinder extends Circle (missing extends)
```

```
{  
    Circle(radius); // Must use super(radius)  
    length = length; // Must use this.length = length  
}
```

```
public double findArea()  
{  
    return findArea()*length; // super.findArea()  
}
```

4. Method overloading defines methods of the same name in a class. Method overriding modifies the methods that are defined in the superclasses.
5. Yes, because these two methods are defined in the Object class; therefore, they are available to all Java classes. The subclasses usually override these methods to provide specific information for these methods.

The toString() method returns a string representation of the object; the equals() method compares the contents of two objects to determine whether they are the same.

6. B's constructor is invoked  
A's constructor is invoked

The default constructor of Object is invoked, when new A(3) is invoked. The Object's constructor is invoked before any statements in B's constructor are executed.

7. The output is false if the Circle class in (A) is used. The Circle class has two **overloaded** methods: equals(Circle circle) defined in the Circle class and equals(Object circle) defined in the Object class, inherited by the Circle class. At compilation time, circle1.equals(circle2) is matched to equals(Circle circle), because equals(Circle circle) is more specific than equals(Object circle).

The output is true if the Circle class in (B) is used. The Circle class overrides the equals(Object circle) method defined in the Object class. At compilation time, circle1.equals(circle2) is matched to equals(Circle circle) and at runtime, the equals(Object circle) method implemented in the Circle class is invoked.

8. (a)  
(circle instanceof Cylinder)  
**Answer:** False

(cylinder instanceof Circle)  
**Answer:** True

- (b)  
Yes, because you can always cast from subclass to superclass.

- (c)  
Causing a runtime exception (ClassCastException)

9.

Is fruit instanceof Orange true?    false

Is fruit instanceof Apple true? true  
Is fruit instanceof GoldDelicious true? true  
Is fruit instanceof Macintosh true? false  
Is orange instanceof Orange true? true  
Is orange instanceof Fruit true? true  
Is orange instanceof Apple true? false

Suppose the method makeApple is defined in the Apple class. Can fruit invoke this method? Yes

Can orange invoke this method? No

Suppose the method makeOrangeJuice is defined in the Orange class. Can orange invoke this method? Yes.

Can fruit invoke this method? No.

10.

Object apple = (Apple)fruit;  
Causes a runtime ClassCastException.

11.

2 // because b's declared type is the class B.  
4 // because b's declared type is the class B.  
1 // because a's declared type is the class A.  
3 // because a's declared type is the class A.  
1 // because a's actually an object of B, so the m()  
method in B is invoked.  
7 // because a's declared type is A, so the static  
method m1() in A is invoked.

12. default visibility modifier.

13. protected.

14. Yes.

15. protected

16. The finalize, clone, and getClass methods are defined in the Object class. The finalize method is invoked on the object when the object is destroyed.

17. The clone method is defined in Object as protected. So this method cannot be invoked from every object. To invoke

it the Class for the object must implement it and make it public and also the class must implement the Cloneable interface. See Chapter 9 for discussions on interfaces.

18. The return type of the `getClass` method is `java.lang.Class`.

19. The output are the following for (A):

```
j is 2 (from Line 16)
i is 1 (from Line 12)
```

The static initialization block (Lines 15-17) is executed when class A is loaded. The instance non-static initialization block (Line 11-13) is executed when A's default constructor is invoked.

The output are the following for (B)

```
i is 0 (from Line 17)
j is 4 (from Line 18)
i is 5 (from Line 12)
j is 4 (from Line 13)
```

When `new A()` is invoked (Line 3), A's superclass's default constructor is executed. So, `m()` in B's constructor is executed. Since `m()` is defined in B (Lines 27-28), but overridden in A (Lines 11-14), the body of `m()` defined in A is invoked by polymorphism. The statement `System.out.println` in Line 17 is now executed. At this time, 5 has not been assigned to `i` yet. Therefore, the printout from Line 17 is (1) `i is 0`. The printout from Line 18 is (2) `j is 4`, since `j` is a static variable and it is initialized when class A is loaded. 5 will be assigned to `i` (Line 8) after A's superclass constructor is invoked. When the `System.out.println` statement in Line 12 is executed, `i is 5`. Therefore, the printout from Line 12 is `i is 5`. The printout from Line 13 is (4) `j is 4`.

20. Find these terms in this chapter.

21. Indicate true or false for the following statements:

1. True.
2. False. (But yes in a subclass that extends the class where the protected datum is defined.)
3. True.

4. A final class can have instances.  
**Answer:** True
5. A final class can be extended.  
**Answer:** False
6. A final method can be overridden.  
**Answer:** False
7. You can always successfully cast a subclass to a superclass.  
**Answer:** True
8. You can always successfully cast a superclass to a subclass.  
**Answer:** False
9. The order in which modifiers appear before a method is important.  
**Answer:** False

A student asked a good question: What are advantages of dynamic binding?

Dynamic binding enables new classes be loaded on the fly without recompilation. There is no need for developers to create, and for users to install, major new software versions. New features can be incorporated transparently as needed. For example, suppose you have placed the classes Test, GraduateStudent, Student, Person in four separate files. If you change GraduateStudent as follows:

```
class GraduateStudent extends Student {  
    public String toString() {  
        return "Graduate Student";  
    }  
}
```

You have a new version of GraduateStudent with a new toString method, but you don't have to recompile the classes Test, Student, and Person. When you run Test, the JVM dynamically bind the new toString method for the object of GraduateStudent when executing m(new GraduateStudent()).