Recitation 1 Solution

Inheritance, Static Members

---

1. Consider the following class definition:

```
public class A {
  public A () {}
  public A (int a, int b) {}
}

public class B extends A {
    public B (int r) {}
    public B (int r, int w) {
      super (r, w);
    }
}
```

Which of the following are legitimate calls to construct instances of the B class? For those that are not, explain why.

a. `B c = new B();`
b. `A s = new B(1);`
c. `B c = new B(1, 9);`
d. `A t = new B(1, 9, 4);`
e. `B t = (new B(1)).new B(1);`
f. `B b = new A(1, 2);`

SOLUTION

a. Invalid - `B` does not have a no-arg constructor
b. Valid
c. Valid
d. Invalid - no 3-argument constructor defined
e. Invalid - gibberish
f. Invalid - b is of static type B, and cannot refer to object of dynamic type A since A is not a subclass of B

---

2. Inheritance/Dynamic Binding

a. Will the following code compile? If not, where exactly will it fail to compile?

```
public class A {
  public int x;
  public A(int x) {
    this.x = x;
  }
}

public class B extends A {
  public int y;
  public B(int y) {
    this.y = y;
  }
}
```

ANSWER:

The class B will not compile. The error is that the constructor B(int y) implicitly calls the constructor A(), which does not exist. Constructors of subclasses must call, explicitly or implicitly, a valid constructor of the superclass (or transitively, through a call to another constructor in the same class), before any other statement in the method.

b. Given:

```
public class B {
  public int x;
  public String toString() {
    return x + "";
  }
}

public class E extends B {
  public int y=3;
  public String toString() {
    return (x + y) + "";
  }
}
```

What is the output of the following code segment:

```
B b = new E();
b.x = 5;
System.out.println(b);
```

ANSWER:

Output: 8

With dynamic binding, the version of toString() in E is called.

c. Given:

```
public class B {
  private int x;
  public int getX() {
    return x;
  }
  public String toString() {
    return x + "";
  }
}

public class E extends B {
  public int y=3;
  public String toString() {
    return getX() + y + "";
  }
}
```

What is the output of the following code segment, which is in a different class than B or E:

```
B b = new E();
System.out.println(b);
```

ANSWER:

Output: 3

1. When an instance of E is created in the first statement above (before any constructor is called to initialize the object), its inherited field x gets a value of 0 by default, and its field y is explicitly set to 3
2. E does not have an explicit constructor, so the default constructor comes into play, which is called in the first statement above to initialize a new E object.
3. The default constructor implictly calls super() on superclass B
4. Since B does not explicitly define a constructor, again its default constructor comes into play, which does no initialization. This means the inherited x in E retains its default value of 0 set at object creation time
5. In the second, println statement, the toString() version of E is called, by dynamic binding, which returns 0 (for getX()) + 3 (for y)

d. Given:

```
public class V {
  public static int stuff() {
    return 1;
  }
}

public class W extends V {
  public static int stuff() {
    return 2;
  }
}
```

What is the output of the following code segment, which is in a different class than W or V:

```
V v = new W();
System.out.println(v.stuff());
```

ANSWER:

Output: 1

Since stuff() is static, dynamic binding is not done when it is called. Since the call is via v, whose static (compile time) type is V, the V version of stuff() is called, even though v refers to a W instance.

e. Given:

```
public class G {
  public int g;
}

public class H extends G {
  public int h;
  public boolean equals(Object o) {
    if (o == null || !(o instanceof H)) {
      return false;
    }
    return g == ((H)o).g;
  }
}
```

What is the output of the following code segment, which is in a different class than H or G:

```
G ag = new H(); ag.g = 15;
G bg = new G(); bg.g = 15;
if (ag.equals(bg)) {
  System.out.println(10);
} else {
  System.out.println(20);
}
```

ANSWER:

Output: 20

1. ag.equals(bg) ends up calling the H version of equals by dynamic binding
2. Which checks if object bg is of type H. bg is not, so this check fails and so equals(bg) returns false, and 20 is printed

f. Given:

```
public class B {
  public int x;
  public String toString() {
    return x + "";
  }
}

public class E extends B {
  public int y=3;
  public String toString() {
    return (x + y) + "";
  }
}
```

What is the output of the following code segment, which is in a different class than B or E:

```
B b = new E();
System.out.println(b.y);
```

ANSWER:

Compile error.

The compiler sees the statement b.y, and will pass it if there is a y field in class B, since b is of static type B. Since B does not have a y field, the compiler flags an error.

---

3. What is the output of this code? Why?

```
class GrandParent { }
class Parent extends GrandParent{ }
class Child extends Parent { }

class Foo {
  public void bar(GrandParent p) {
    System.out.println("called with type GrandParent");
  }
  public void bar(Parent p) {
    System.out.println("called with type Parent");
  }
```

```
}

public class Test {
  public static void main(String[] args) {
    new Foo().bar(new Child());
  }
}
```

SOLUTION

The output is:

```
called with type Parent
```

The method call matches the argument type (Child) with the closest matching parameter type (Parent). "Closest" means either the same type, or the nearest matching type when going up the object ancestry following the chain of superclasses, toward the ultimate ancestor, Object. Since Parent is the immediate superclass of Child, the method Foo.bar(Parent) is called.

---

4. In this exercise we will try to see how static and final methods work with inheritance. (A final method is one that cannot be overridden in a subclass.) Consider the two classes defined below, Parent and Child:

```
public class Parent {
  /*
   public static final void printClassName() {
     System.out.println("I am in class Parent, static invocation.");
   }
  */

   public final void printName() {
     System.out.println("I am in class Parent, dynamic invocation.");
   }
}

public class Child extends Parent {
  /*
   public static void printClassName() {
     System.out.println("I am in class Child, static invocation.");
   }
  */

   public void printName() {
     System.out.println("I am in class Child, dynamic invocation.");
   }
}
```

   A. Compile both the classes. What do you see?
   B. Now comment out the printName() method in the Child class and uncomment the printClassName() method in both classes. Compile both classes. What do you see? Is it different from part (A)? Why?
   C. Uncomment the Child.printName() method and remove the final modifier from Parent.printName() and Parent.printClassName(). Recompile both classes.
   D. Compile and run the following class:

```
class App {
  public static void main(String[] args) {
    Parent p = new Child();
```

```
    p.printName();     // WHAT IS PRINTED?
    p.printClassName(); // WHAT IS PRINTED?
  }
}
```

Explain the difference in *which* methods are invoked in these two method calls.

SOLUTION

A. Compiling the classes results in the Java compiler printing the following error message:

```
1. ERROR in Child.java (at line 10)
public void printName()
       ^^^^^^^^^^^
Cannot override the final method from Parent
```

B. Compiling the classes results in the Java compiler printing the following error message:

```
1. ERROR in Child.java (at line 3)
public static void printClassName()
            ^^^^^^^^^^^^^^^^
Cannot override the final method from Parent
```

It is different from part (A), but in essence it is the same error: trying to override a final method in a subclass.

C. It compiles successfully.

D. The App.java program compiles, but the compiler issues a warning because printClassName() should be invoked on the class, not on an instance. The printName() call invokes the Child class's implementation because p refers to an instance of Child. The printClassName() call invokes the Parent class's implementation because p is a reference of type Parent.

---

5. Here's a Widget class:

```
public class Widget {
  float mass;

  private static float MAX_MASS = 20;

  public static final float G = 9.81f;

  public Widget(float mass) {
    if (mass > MAX_MASS) {
      throw new IllegalArgumentException();
    }
    this.mass = mass;
  }

  public static float getMaxMass() {
    return MAX_MASS;
  }

  public float getWeight() {
    return mass*G;
```

```
  }
}
```

Now suppose there is a certain set of widgets that are "heavy", so their maximum mass is 40 instead of the usual 20. Write a class called HeavyWidget, as a subclass of Widget. Do you encounter any implementation issues when you do this? Can you get around these issues? If so, show how. If not, explain why.

SOLUTION

Here's the try to implement HeavyWeight:

```
public class HeavyWidget extends Widget {

  private static float MAX_MASS = 40;

  public HeavyWidget(float mass) {
    super(mass);
    if (mass > MAX_MASS) {
      throw new IllegalArgumentException();
    }
    this.mass = mass;
  }
  ...
```

The problem is the call super(mass) in the constructor. This call needs to be made because the first statement in a subclass constructor must be a call to a superclass constructor. Since there is only one constructor in the superclass Widget, which accepts a float parameter, we have no choice but to write in the super(mass) call. However, the Widget constructor would throw an exception if the mass exceeds 20, even though HeavyWeight allows mass to be as much as 40. So this is an issue.

We can try to get around the issue by putting in a try-catch like this:

```
public HeavyWidget(float mass) {
  try {
   super(mass);
  } catch (IllegalArgumentException e) {

  }
  if (mass > MAX_MASS) {
    throw new IllegalArgumentException();
  }
  this.mass = mass;
}
```

But this does not compile because try is now taken to be the first statement, which makes the compiler throw in a super() call *before* the try. And the compiler complains that there isn't a no-arg constructor in the Widget class. Also, it complains that the super(mass) statement is not the first statement.

The fact that this issue doesn't have a meaningful workaround suggests that there may be some flaw in the design. In particular, making HeavyWidget a subclass of Widget implies that EVERY heavy widget is a widget. (Along the lines of every student is a person, or every tiger is an animal). But is this true?

If it were true, then EVERY heavy widget must pass the widget test. Which won't happen for all heavy widgets that have mass greater than 20. So it must be that every heavy widget is NOT a widget.

So then, could we flip it around, and ask whether EVERY widget is a heavy widget? In other words, if you were to make Widget a subclass of HeavyWidget, could you come up with clean code that works? Or is it

that neither of them can be a subclass of the other?