# Recitation 8 - Solution

## Interface Default Methods, OO Design

---

1. Interface Default Methods.

   For each of the following, tell whether the code will compile. If so, tell what will be printed, with reasoning. If not, explain why.

   1. ```
      public interface I {
         private default void m() {
            System.out.println("I:m");
         }
      }
      public class X implements I {
         public static void main(String[] args) {
            new X().m();
         }
      }
      ```

      **ANSWER**

      No. Method `m` in interface `I` is declared private, but interface methods are always public and abstract.

   2. ```
      public interface I {
         default void m1() {
            System.out.println("I:m1");
      }
      public class X implements I {
         public void m1() {
            System.out.println("X:m1");
         }
         public static void main(String[] args) {
            new X().m1();
         }
      }
      ```

      **ANSWER**

      Yes. Prints "X:m1". X gets the default method `m` by virtue of implementing interface `I`, but overrides it with its own implementation.

   3. ```
      public class X {
         public void m1() {
            System.out.println("X:m1");
         }
      }
      public interface I {
         default void m1() {
            System.out.println("I:m1");
      }
      public class XI extends X implements I {
         public static void main(String[] args) {
            new XI().m1();
         }
      }
      ```

      **ANSWER**
      ```

Yes. Prints "X:m1". Class `XI` gets conflicting method implementations of `m1` from superclass `X` and interface `I`. But according to the resolution rules for conflicts, class `X`'s implemetation of `m1` trumps `I`'s.

4.
```
public interface A {
   default void hello() {
      System.out.println("A!");
   }
}
public interface B extends A {
   default void hello() {
      System.out.println("B!");
   }
}
public class C implements A, B {
   public static void main(String[] args) {
       new C().hello();
   }
}
```

**ANSWER**

Yes. Prints "B!". Since `B` is more specific than `A`, class `C` gets `B`'s implementation of method `hello`.

5.
```
public interface I {
   default double getNumber() {
      return 3.5;
   }
}
public interface J {
   default int getNumber() {
      return 3;
   }
}
public class X implements I, J {
   public static void main(String[] args) {
       System.out.println(new X().getNumber());
   }
}
```

**ANSWER**

No. Methods `getNumber` from `I` and `J` are conflicting.

6.
```
public interface I {
   default void name() {
      System.out.println("I");
   }
}

public interface J extends I { }

public interface K extends I { }

public class X implements J,K {
   public static void main(String[] args) {
       new X().name();
   }
}
```

**ANSWER**

Yes. Neither `J` nor `K` overrides the `name` method implementation of `K`, so there are no conflicting methods in `X`.

---

2. (Adapted from an example in *Interface Oriented Design* by Ken Pugh, sec. 4.2)

   Suppose you want to create a printing subsystem for several different printers, each with different capabilities/features (*e.g.* color printing, high-resolution, duplex, multiple trays, etc.). How could you place the different capabilities into an interface? Should you use a single interface or multiple interfaces (*e.g.* ColorPrinter, DuplexPrinter, MultiTrayPrinter)? Describe the advantages and disadvantages of both approaches.

   **SOLUTION**

   It is generally better to produce several interfaces that are more specific to the various types of printers than to produce one large interface that covers all possibilities. If the interface is modified, then only code utilizing the modified interface must be changed. Programming each separate interface can take place independently and in parallel, so that the tasks can be distributed to different programmers.

   Also, any printer that did not make use of certain features of a single, combined interface, would have to write method stubs (empty method bodies) for non-implemented features, and throw exceptions if those methods were called. This is bad design, since a class should support every single feature for which it has a public method (otherwise why is that method in the class?)

---

3. Suppose you built a Java library of sorting algorithms: insertion sort, quicksort, and heapsort. You want to sell this library. How would you package your library so users could use any of these algorithms in their applications, and switch from using one to another (plug-n-play) with the least amount of code rewrite? Come up with two alternate designs: one with interfaces, and one without. Which one is better? Why?

   **SOLUTION**

   **Interface**

   Write an interface called `SortingAlgorithm` with one or more methods called `sort`, and then write various sorting classes for the different sorting algorithms, that implement the `SortingAlgorithm` interface.

   **Abstract Classes**

   This is the abstract superclass of all the sorting algorithms:

```
public abstract class Sort<T extends Comparable<T>> {
   T[] list;
   public Sort(T[] list) {    // implemented constructor
      this.list = list;
   }
   public abstract void sort();  // abstract sort method
}
```

   Here's an example of one of the sorting implementations (just the shell):

```
public class InsertionSort<T extends Comparable<T>> extends Sort<T> { // note the syntax
   public InsertionSort(T[] list) {
      super(list);
   }
    public void sort() {   // implement the overridden abstract method
      // implementation goes here
      ...
    }
}
```

   And the usage in client code:

```
String[] list = new String[n];
// populate list with strings
...
Sort<String> sorter = new InsertionSort<String>(list);
sorter.sort();
// list is now in sorted order
```

The interface solution is better. Abstract classes offer the potential of code being reused in subclasses, but this comes at the cost of the subclasses not being able to extend another class. In this case, there is very little reusable code in the abstract class since there is little in common in the sorting algorithms that can be generalized. So the cost overrides the benefit.

---

4. A game developer asks you to make a set of classes to represent the monsters in a game. There are at least two different types of monsters, those that walk and those that bounce, but the code you write needs to be easily expandable to different types. Specifically, the code to keep track of a monster's appearance and to draw the monster needs to be in only one place. You are given an interface to start out:

```
public interface Monster {
    void drawMonster();
    void setMonsterImage(Image i);
    void updatePosition();
}
```

Create an abstract base class `MovingMonster` for all monsters, and one subclass for each of two types discussed, `WalkingMonster` and `BouncingMonster`. Each monster will need to keep track of its own position and update it when the `updatePosition()` method is invoked. Assumethat the Image class has a method `draw(int x, int y)`. The contents of the `updatePosition()` method are not important, but it has to change the monster's position and be different for either monster.

**SOLUTION**

An example implementation:

```
public abstract class MovingMonster implements Monster {
    protected Image image;
    protected int xPosition;
    protected int yPosition;

    public abstract void updatePosition();

    public void drawMonster() {
        // Some implementation here.
    }
    public void setMonsterImage(Image i) {
        image = i;
    }

    public MovingMonster(Image image) {
        this.image = image;
        xPosition = yPosition = 0;
    }
}

public class WalkingMonster extends MovingMonster {
    public void updatePosition() {
        xPosition++;
    }

    public WalkingMonster(Image image) {
        super(image);
```

```
        }
    }

    public class BouncingMonster extends MovingMonster {
        public void updatePosition() {
            xPosition++;
            yPosition = (yPosition + 1) % 2;
        }

        public BouncingMonster(Image image) {
            super(image);
        }
    }
```