# 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. Suppose you design a class, Set, whose members behave like finite, unordered mathematical sets of integers, and can support the operations of membership query, union of two sets, intersection of two sets, and difference of two sets.

   Consider the intersection operation. There are at least two ways of declaring such an operation in the class Set:

   ```
   public Set intersect(Set otherSet)
   ```

   or

   ```
   public static Set intersect(Set firstSet, Set secondSet)
   ```

   Give one pro and one con for the static version.

   **SOLUTION**

   The static version of the method is semantically closer to the mathematical idea of set operations. The static approach makes it clear to programmers that the intersect method is a symmetric operation. The drawback of a static definition is that it cannot be overridden by subclasses if we wanted to extend the Set class and apply polymorphism.

---

3. 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);
  }
}
```

---

4. Suppose we need to have the Point and ColoredPoint classes provide functionality to parse text representations of points and colored points (as would be returned by the toString method), and return Point and ColoredPoint objects, respectively.
    1. Show how you would implement this functionality.
    2. How much of the Point implementation of this functionality is reused in ColoredPoint? (Reuse meaning using code from Point by calling on it in ColoredPoint.) If yes, indicate which part, else explain why not.
    3. Does your implementation give rise to dynamic binding of the parsing functionality? (Recall that dynamic binding means the subclass version of a method is "bound" to the call made via an object reference that is statically typed to the superclass.)

**SOLUTION**

1. In Point class:

```
public static Point parsePoint(String pointStr) {
  String[] tokens = pointStr.split(",");
  if (tokens.length != 2) {
    throw new IllegalArgumentException();
  }
  try {
    int x = Integer.parseInt(tokens[0]);
    int y = Integer.parseInt(tokens[1]);
    return new Point(x,y);
  } catch (Exception e) {
    throw new IllegalArgumentException();
  }
}
```

   In ColoredPoint class:

```
public static ColoredPoint parsePoint(String pointStr) {
  String[] tokens = pointStr.split(",");
  if (tokens.length != 3) {
    throw new IllegalArgumentException();
  }
  try {
    int x = Integer.parseInt(tokens[0]);
    int y = Integer.parseInt(tokens[1]);
    return new ColoredPoint(x,y,tokens[2]);
  } catch (Exception e) {
    throw new IllegalArgumentException();
  }
}
```

2. No code is reused. The only way to reuse code in ColoredPoint.parsePoint would be with a call to Point.parsePoint. However, the latter returns a Point object, which would not fit in the former's implementation because it has no use for a Point object. (The idea of reuse arises because parsing a colored point equals parsing a point, plus parsing the color part.)

3. No, there is no dynamic binding since the methods involved are both static. So, binding is done purely on the static/compile-time type of the reference variable, without regard to what object it is pointing to. This means if you have:

```
Point ptest = new ColoredPoint(3,4,"orange");
```

   then a call such as ptest.parsePoint(...) would invoke the Point class' parsePoint method, not the ColoredPoint's.