COMP132 – Computer Science II
Fall 2015

Homework #3
Interfaces and Polymorphism
Solutions

1. Define an interface named `Flies` to be implemented by any object that can fly. Include at least 2 relevant methods in your interface. You do not need to provide JavaDoc for this interface. Copy and paste the code for your interface as the solution to this problem.

```
public interface Flies {
  public void fly();
  public int maxHeight();
}
```

2. Define a class `Airplane` that implements both the `Flies` interface and the `MakesSound` interfaces. The MakesSound interface can be found in the `comp132.examples.interfaces` package of the `132SampleCode` project. You do not need to provide JavaDoc for this class. Copy and paste the code for your `Airplane` class as the solution to this problem.

```
public class Airplane implements MakesSound, Flies {

  private int volume;
  private int height;

  public Airplane(int volume, int maxHeight) {
      this.volume = volume;
      height = maxHeight;
  }
  public int howLoud() { // In MakesSound Interface
      return volume;
  }
  public void makeSound() { // In MakesSound Interface
      System.out.println("Zoom Woosh");
  }
  public void fly() { // In Flies Interface
      System.out.println("Spinning my turbo prop.");
  }
  public int maxHeight() { // In Flies Interface
      return height;
  }
}
```

3. Consider the two interfaces defined below:

```
public interface SmellsBad {
    public void stink();
}
```

```
public interface LooksBad {
    public void yuck();
}
```

Now imagine two classes:
Class `Foo` implements the `SmellsBad` interface.
Class `Bar` implements both the `SmellsBad` and the `LooksBad` interfaces.

An instance of each of these classes can be created as follows:

```
Foo f1 = new Foo();
Bar b1 = new Bar();
```

Note: You may not compile or run any of the given code when answering this question.

Hint: Drawing object diagrams will help with this problem!

A. Given the above objects indicate which of the following statements are legal (i.e. will compile as written) and which are illegal (i.e. will generate compiler errors):

i. `f1.stink();` **Legal**
`f1` refers to a `Foo` object, the `Foo` class implements the `SmellsBad` interface, therefore `f1` can be used to invoke the `stink` method.

ii. `f1.yuck();` **Illegal**
`f1` refers to a `Foo` object, the `Foo` class does not implement the `LooksBad` interface, therefore `f1` cannot be used to invoke the `yuck` method (NOTE: It is possible that `Foo` defines a `yuck` method without implementing the `LooksBad` interface. In that case this statement would be legal.)

iii. `b1.stink();` **Legal**
`b1` refers to a `Bar` object, the `Bar` class implements the `SmellsBad` interface, therefore `b1` can be used to invoke the `stink` method.

iv. `b1.yuck();` **Legal**
`b1` refers to a `Bar` object, the `Bar` class implements the `LooksBad` interface, therefore `b1` can be used to invoke the `yuck` method.

B. Given the above objects indicate which of the following assignments are legal (i.e. will compile as written) and which are illegal (i.e. will generate a compiler error):

i. `SmellsBad sb1 = f1;` **Legal**
The object referred to by `f1` is a `Foo` object and the class `Foo` implements the `SmellsBad` interface. So the reference `sb1` of type `SmellsBad` can refer to this object.

ii. `LooksBad lb1 = f1;` **Illegal**
The object referred to by `f1` is a `Foo` object and the `Foo` class does not implement the `LooksBad` interface. So the reference `lb1` of type `LooksBad` cannot refer to this object.

iii. `SmellsBad sb2 = b1;` **Legal**
The object referred to by `b1` is a `Bar` object and the class `Bar` implements the `SmellsBad` interface. So the reference `sb2` of type `SmellsBad` can refer to this object.

iv. `LooksBad lb2 = b1;` **Legal**
The object referred to by `b1` is a `Bar` object and the class `Bar` also implements the `LooksBad` interface. So the reference `lb2` of type `LooksBad` can refer to this object.

C. Assuming that the legal assignments in Part B have been executed, indicate which of the following statements are legal (i.e. will compile as written) and which are illegal (i.e. will generate a compiler error):

i. `sb2.stink();` **Legal**
`sb2` is a reference of type `SmellsBad` and the method `stink` is defined in the `SmellsBad` interface.

ii. `sb2.yuck();` **Illegal**
`sb2` is a reference of type `SmellsBad` and the method `yuck` is not defined in the `SmellsBad` interface.

iii. `lb2.stink();` **Illegal**
`lb2` is a reference of type `LooksBad` and the method `stink` is not defined in the `LooksBad` interface.

iv. `lb2.yuck();` **Legal**
`lb2` is a reference of type `LooksBad` and the method `yuck` is defined in the `LooksBad` interface.

D. Assuming that the legal assignments in Part B have been executed, indicate if each of the following type casts are *legal* (i.e. will compile as written), will generate a *compiler error* or will generate a *runtime error*:

i. `Foo f2 = (Foo) b1;` **Compiler Error**

`b1` is a reference of type `Bar`. A reference of type `Bar` can never actually refer to an object of type `Foo`. Therefore, the reference in `b1` cannot be copied into `f2`, even with a cast.

ii. `Foo f3 = (Foo) sb1;` **Legal**

`sb1` is a reference of type `SmellsBad`. A reference of type `SmellsBad` could be referring to an object of type `Foo`. Thus, with a cast this statement will not cause a compiler error. Further, `sb1` actually refers to an object of type `Foo`, thus the cast will also be successful at runtime.

iii. `Bar b2 = (Bar) sb2;` **Legal**

`sb2` is a reference of type `SmellsBad`. A reference of type `SmellsBad` could be referring to an object of type `Bar`. Thus, with a cast this statement will not cause a compiler error. Further, `sb2` actually refers to an object of type `Bar`, thus the cast will also be successful at runtime.

iv. `SmellsBad sb3 = (SmellsBad) lb2;` **Legal**

`lb2` is a reference of type `LooksBad`. A reference of type `LooksBad` could be referring to an object that also implements `SmellsBad` (e.g. `Bar`). Thus, with a cast this statement will not cause a compiler error. Further, `lb2` actually refers to an object of type `Bar`, which implements `SmellsBad`, thus the cast will also be successful at runtime.

v. `LooksBad lb3 = (LooksBad) sb2;` **Legal**

`sb2` is a reference of type `SmellsBad`. A reference of type `SmellsBad` could be referring to an object that also implements `LooksBad` (e.g. `Bar`). Thus, with a cast this statement will not cause a compiler error. Further, `sb2` actually refers to an object of type `Bar`, which implements `LooksBad`, thus the cast will also be successful at runtime.

vi. `LooksBad lb4 = (LooksBad) sb1;` **Runtime Error**

`Sb1` is a reference of type `SmellsBad`. A reference of type `smellsBad` could be referring to an object that also implements `SmellsBad` (e.g. `Bar`). Thus, with a cast this statement will not cause a compiler error. However, `sb2` actually refers to an object of type `Foo`, which does not implement `LooksBad`, thus the cast will fail at runtime, generating a `ClassCastException`.

E. Assuming that the legal assignments in Part B have been executed, indicate which of the following type casts are necessary and which are unnecessary:

i. `Bar b3 = (Bar) lb2;` **Necessary**

`lb2` is a reference of type `LooksBad`. A reference of type `LooksBad` could be referring to a `Bar` object but it might also be referring to another type of object that implements `LooksBad`. Therefore, this cast could fail at runtime and thus a cast is required.

ii. `LooksBad lb5 = (LooksBad) b1;` **Unnecessary**

`b1` is a reference of type `Bar`. The `Bar` class implements `LooksBad`. Therefore, this cast can never fail fail at runtime and thus a cast is not required.

iii. `SmellsBad sb4 = (SmellsBad) sb2;` **Unnecessary**

`sb2` is a reference of type `SmellsBad`, so whatever object it refers to must implement `SmellsBad`. Therefore, this cast can never fail at runtime and thus a cast is not required.

iv. `SmellsBad sb5 = (SmellsBad) lb2;` **Necessary**

`lb2` is a reference of type `LooksBad`. A reference of type `LooksBad` could be referring to an object that also implements `SmellsBad` (e.g. `Bar`). But it might also be referring to another type of object that implements `LooksBad` but does implement `SmellsBad` (e.g. `Foo`). Therefore, this cast could fail at runtime and thus a cast is required.

v. `Foo f4 = (Foo) f1;` **Unnecessary**

`f1` is a reference of type `Foo`, so it will always refer to a `Foo` object. Therefore, this cast can never fail at runtime and thus a cast is not required.

F. Assuming that the legal assignments in Part B have been executed indicate the value (`true` or `false`) of each of the following Boolean expressions:

i. `b1 instanceof SmellsBad` **true**

`b1` refers to an object of type `Bar` which, because `Bar` implements `SmellsBad`, is an instance of the type `SmellsBad`.

ii. `sb1 instanceof Bar` **false**

From part B, `sb1` is referring to an object of type `Foo`, which is not an instance of the type `Bar`.

iii. `sb2 instanceof Bar` **true**

From part B, `sb2` is referring to an object of type `Bar`, which is an instance of the type `Bar`.

iv. `lb2 instanceof SmellsBad;` **true**

From part B, `lb2` is referring to an object of type `Bar` which, because `Bar` implements `SmellsBad,` is an instance of the type `SmellsBad`.

v. `sb1 instanceof LooksBad;` **false**

From part B, `sb1` is referring to an object of type `Foo`, but `Foo` does not implement `LooksBad`, and therefore is not an instance of the type `LooksBad`.

4. Consider the following interface definition:

```
public interface Mystery {
    public void abc();
    public int def();
}
```

The following two classes both implement the `Mystery` interface:

```
public class Strange
  implements Mystery {

    private int x;

    public Strange() {
        x = 5;
    }

    public void abc() {
        x = x - 1;
    }

    public int def() {
        return 2*x;
    }
}
```

```
public class Unknown
  implements Mystery {

    private int x;

    public Unknown() {
        x = 3;
    }

    public void abc() {
        x = x + 7;
    }

    public int def() {
        return x + 6;
    }
}
```

Now assume that the following method is available to be invoked:

```
public static void doIt(Mystery my) {
  my.abc();
  System.out.println(my.def());
}
```

Given all of the above, what output would generated by each of the calls to the `doIt` method?

```
Strange s1 = new Strange();
Unknown u1 = new Unknown();

doIt(s1);
doIt(u1);

Mystery m1 = s1;
doIt(m1);

m1 = u1;
doIt(m1);
```

```
doIt(s1);          ➔ 8
doIt(u1);          ➔ 16

Mystery m1 = s1;
doIt(m1);          ➔ 6

m1 = u1;
doIt(m1);          ➔ 23
```

Remember that the type of the object determines what happens. Thus, a really good way to understand this question is to draw an object diagram and use it to keep close track of which object the code in the `doIt` method is operating on.

5. Consider the following snippet of code that creates an a `ArrayList` of objects that implement the `MakeSound` method.

```
ArrayList<MakesSound> soundMakers = new ArrayList<MakesSound>();
soundMakers.add(new Duck("Mallard"));
… additional statements omitted …
soundMakers.add(new Car("Volvo", 3));
```

A. Give a snippet of code that displays the sound that is made by each object in the `ArrayList`.

```
for (MakesSound ms :soundMakers) {
  ms.makeSound();
}
```

B. Give a snippet of code that invokes the `swim()` method on every object in the `ArrayList` that implements the `Swims` interface.

```
for (MakesSound ms : soundMakers) {
  if (ms instanceof Swims) {
    Swims s = (Swims)ms;
    s.swim();
  }
}
```