Java Interfaces

Lab Setup

1. Create a new Java project and call it "JavaInterface".
2. Add a driver main class to the project.

Interfaces

Interfaces provide an outline for implementing classes using methods, but the methods are abstract. They do not have a body, only a method signature. All fields declared in an interface are `public`, `static`, and `final`.

Classes implement interfaces using the keyword `implements`. All classes that implement an interface must provide the definition of all abstract methods in the interface. In this way, interfaces describe how the implementing class should behave.

We will use the example of space-faring vessels i.e., spaceships. We will use interfaces to designate types of spaceships and specify what those spaceships should be able to do.

1. We will first define the interfaces, `Lander` and `InAtmosphereShip`

```
// Lander.java
interface Lander {
    boolean land();
}

// InAtmosphereShip.java
interface InAtmosphereShip {
    boolean takeOff();
}
```

Figure 1 Lander and InAtmosphereShip interfaces.

With these two interfaces we can implement classes which are Landers and InAtmosphereShips. Landers must be able to land i.e. implement the `land()` method, and InAtmosphereShips must be able to take off i.e. implement the `takeOff()` method.

1. We can create two classes which implement each interface.

   First, create the class `ZhuShip` which implements `InAtmosphereShip`

Eclipse will underline the class and you will see a problem that tells you `ZhuShip` must define the method `takeOff()`.

Define `takeOff` as follows and create a constructor for `ZhuShip`.

```
// ZhuShip.java
class ZhuShip implements InAtmosphereShip {

      public ZhuShip(){

      }

      boolean takeOff(){
          System.out.println("ZhuShip takes off");
          return true;
      }

}
```

Figure 2 ZhuShip class

By looking at the declaration of `ZhuShip` we can see that it can at least "take off" because it must. Landing for a `ZhuShip` is not necessarily defined.

Second, create the class `CorvegaShip` which implements `Lander.`

```
// CorvegaShip.java
class CorvegaShip implements Lander {

    public CorvegaShip(){

    }

    public boolean land(){
        System.out.println("CorvegaShip lands");
        return true;
    }
}
```

Figure 3 CorvegaShip class

What can we infer about `CorvegaShip` from its declaration?

2. An interface can extend another interface.

```
interface Interplanetary {
    void jump(Planet p);
}

interface Interstellar extends Interplanetary{
    void jump(SolarSystem s);
}
```

Figure 4 Interplanetary and Interstellar interfaces

An interface extending another interface can be thought of as adding additional functionality. In Figure 4, we have `Interstellar` extend `Interplanetary` to say that an `Interstellar` ship can also do what an `Interplanetary` ship can.

Graded Submission Task 1

Provide two classes which implement `Interplanetary` and `Interstellar` individually. For the types `Planet` and `SolarSystem`, you can define classes which have a String field representing the name of the `Planet` or `SolarSystem`.

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 1. You will submit this copy at the end of this lab.

Graded Submission Task 2

Remember how interfaces represent an "is-a" relationship? Let's define a root interface so we can say all the above interfaces are spaceships.

```
interface Spaceship{
    void move();
}
```

Figure 5 Spaceship interface

1. Rewrite the other interfaces so that they `extend` Spaceship directly or indirectly. `Interstellar` will indirectly `extend` Spaceship. The implementing classes will now need to provide an implementation for `move()`.

   We cannot instantiate interfaces, but we can declare variables of interfaces.

2. Declare a `Spaceship` variable. An object which is referred to by this variable will only be aware of the `Spaceship` method `move()` even if the object itself implements an interface farther down the interface tree.

3. Declare a variable for each interface and instantiate objects of each implementing class.

4. For each object, call the methods it inherited from the interface it implemented.

** STOP **

Ensure you have saved the source files in your project. Make a copy of your Eclipse project at this point (you can do this from your system's file explorer). Name the file labelling it as Task 2. You will submit this copy at the end of this lab.

Lab wrap-up

In this lab we discussed interfaces. Interfaces allow us to define behaviors that a set of classes should have. Classes implement interfaces using the keyword implements. If a class implements an interface, then it must provide the definition of all abstract methods in the interface. At a glance, we can know what functionality a class has if it implements an interface.

Using interface variables we can write code that will work for any class which implements the interface.

Submission notes for graded tasks:

- Submit a zipped folder containing the Eclipse projects for each graded task.
- For each project, ensure you have included the src folder containing the .java file, the .classpath file, and the .project file.
- Ensure that each project is named appropriately to correspond with each task.
- Zip the projects together and name the zipped folder "Lab10_firstName_lastName".
- Submit the zipped file to the respective Canvas link.

Rubrics	
Task 1 (points allotted): - Compilation (2) - Define a class which implements Interstellar (4) - Define a class which implements Interplanetary (4)	Task 2 (points allotted): - Compilation (2) - Define the Spaceship interface (2) - Define the other interfaces to extend Spaceship (2) - Declare a variable for each interface and show the methods which are visible to it by calling them (4)

Appendix

This section describes default methods and static methods that can be declared within interfaces.

Default methods

Default methods can be used to add new functionality to existing interfaces while ensuring compatibility with older versions. This aspect is discussed in more detail in a future lab. However, here is an example of a default method declared in an interface.

```
interface Filter {
    default void test(String fileName) {
        setup(); // Notice how this is a call to a method that is not necessa
rily defined yet

        System.out.println("Completed test");
    }

    void setup();
}
```

Static methods

Static methods can be declared in interfaces similar to how static methods are declared in classes. A static method should only be used if it is intended for all implementing classes to have use the method. An example of this is provided below.

```
interface Filter {
    default void test(String fileName) {
        long start, end;
        start = getTime();
        setup();
        end = getTime();
        long setupDuration = end - start;

        start = getTime();
        filter();
        end = getTime();
        long filterDuration = end - start;

        System.out.println("""
                           Setup time: %d
                           Filter time: %d
                           """.formatted(setupDuration, filterDuration));
    }

    static long getTime() {
        return System.nanoTime();
```

```
    }

    void setup();
    List<Player> filter(String game);
}
```