Exercise 14

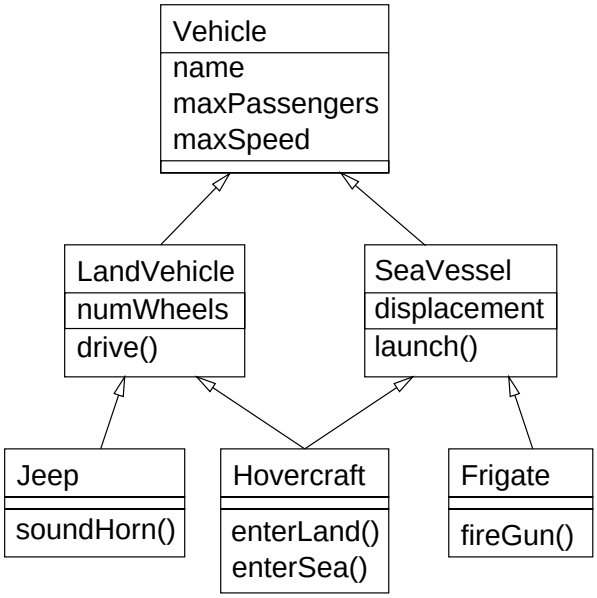
Interfaces

By the end of this exercise you will be able to

- Use *interfaces* as a solution to the problem of *multiple inheritance*.
- Understand how an interface is similar to an abstract class with all methods abstract and no properties except static constants.

Introduction

Consider the following UML diagram.



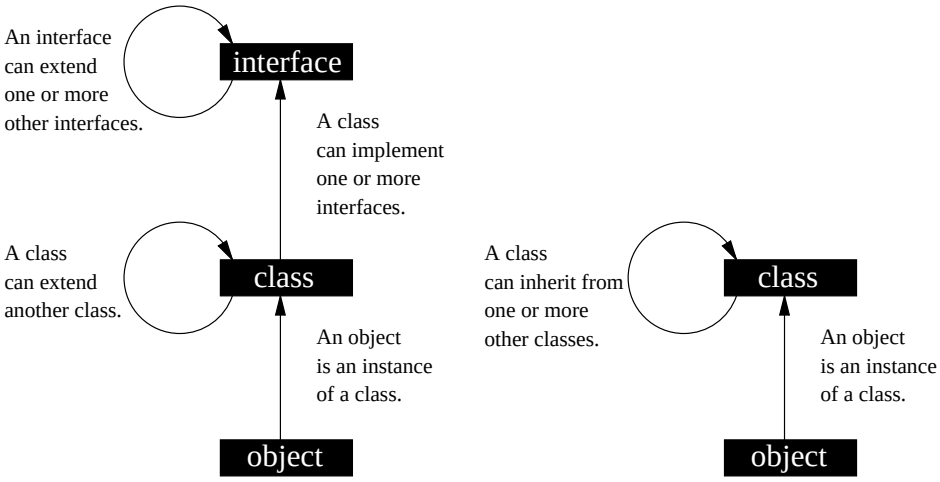
The `Hovercraft` class shown in the diagram inherits from both `LandVehicle` and `SeaVessel` since the hover-craft is in the rather unique position of being able to travel on land and sea. The `Hovercraft` class cannot be expressed in Java since Java does not have the facility for *multiple inheritance*. All other classes in the diagram use only *single inheritance* and so they can be expressed in Java.

One of the problems with multiple inheritance is in deciding what to do with properties in a class like `Vehicle` that is an indirect superclass of `Hovercraft` in two different ways, via `LandVehicle` and via `SeaVessel`. The hover-craft in being able to drive on land and sea might have two different maximum speeds, one for land travel and the other for sea travel. This leads to a problem of what should be the appropriate value for the `maxSpeed` property of `Hovercraft` objects? We could set `maxSpeed` to be the maximum of the two speed values but then this might badly affect the behaviour of the `drive` method which, because it is defined in the `LandVehicle` class, might

assume that the value of `maxSpeed` is the maximum speed attainable on land. A similar problem arises with the `launch` method.

Another approach would be for the `Hovercraft` class to possess two separate `maxSpeed` properties, one for the maximum speed on land and the other for the maximum speed on the sea. The C++ language gives the programmer a choice between having one or two copies of `maxSpeed` with the option of using "virtual" inheritance rather than "normal" inheritance, whereas Java avoids this extra complexity by not allowing multiple inheritance.

So that the Java programmer is not disadvantaged by the lack of multiple inheritance, Java has the `interface` feature, which allows for a kind of multiple inheritance involving interfaces, without the complexity of multiple inheritance of classes that is present in languages like C++. Below left is a diagram showing how interfaces relate to the Java concepts of classes and objects, that you should already be familiar with. Below right is a diagram showing the equivalent concepts in C++.

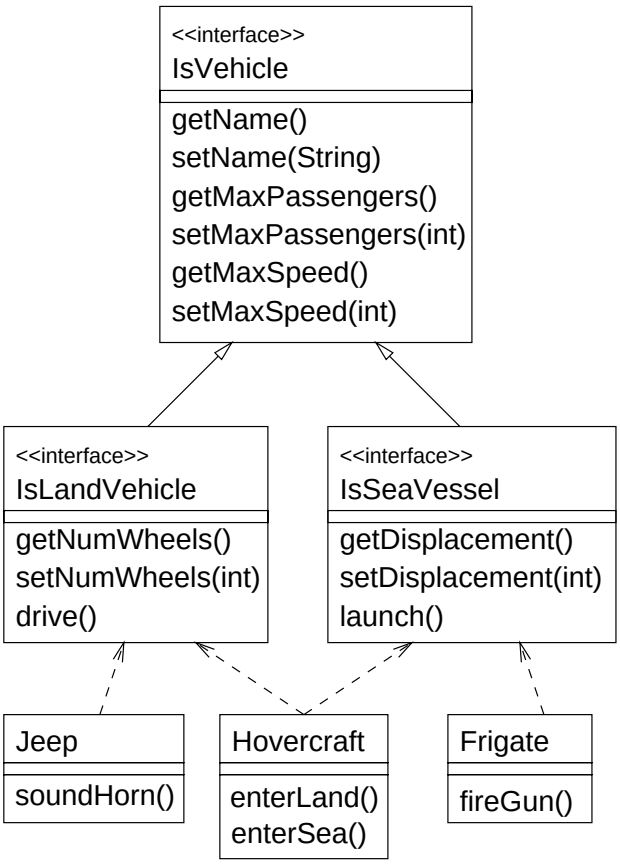


Java C++

The diagram shows that in a sense interfaces are a "higher level concept" than classes, since you can never create an instance of an interface, only instances of classes that implement that interface. Interfaces have no constructors.

The most important feature of interfaces is that a class can implement more than one interface. Interfaces are limited in two respects. Firstly, they are not allowed to have any properties except static constants, and secondly the methods of an interface must be defined without bodies, like abstract methods. These two limitations prevent interfaces from suffering from the problem that occurred with the `maxSpeed` property in the previous UML diagram.

We can re-work the previous UML diagram into something that can be expressed within the Java language by replacing the classes `Vehicle`, `LandVehicle` and `SeaVessel` with interfaces `IsVehicle`, `IsLandVehicle` and `IsSeaVessel`, respectively. The dotted arrows in the next UML diagram indicate classes implementing interfaces.



Note that the `Hovercraft` class implements both the `IsLandVehicle` and `IsSeaVessel` interfaces, rather than inheriting from two classes which is not allowed in Java.

Since an interface is not allowed to have any properties except static constants, we have replaced the properties that existed in the classes `Vehicle`, `LandVehicle` and `SeaVessel` with "getter" and "setter" methods. That is to say that, for each property `X`, there is now a pair of methods `getX` and `setX`. A `getX`, `setX` pair of public methods in a class is logically equivalent for users of the class to a public property called `X`.

Since the methods of the interfaces are defined without bodies, they are defined in the classes `Jeep`, `Hovercraft` and `Frigate` that implement the three interfaces.

Questions

1. Fetch the file `InterTest.java` which contains the program code for the UML diagram above.
2. By copying the pattern from the other interfaces in this file, write an interface `IsEmergency` which extends no other interface and contains just one method `soundSiren` which takes no arguments and returns no value.
3. Write a class `PoliceCar` that implements the `IsEmergency` and `IsLandVehicle` interfaces.
4. In addition to the methods you have written for the `PoliceCar` class, think of a new method or property that police cars have and add it to the class.
5. Add the `PoliceCar` class and the `IsEmergency` interface to the second UML diagram. Show all methods and properties.
6. Construct a `PoliceCar` object and add it to the array `myArray` in the `main` method.

7. By copying the pattern for the existing code inside the `for` loop, add some code that tests the array elements to see if they are instances of classes that implement the `IsEmergency` interface and if so, calls the `soundSiren` method.

   NOTE: In an expression like:

   ```
   if (myArray[i] instanceof IsLandVehicle) {
   ```

   we are testing whether or not the object referenced by `myArray[i]` is an instance of a class that implements the `IsLandVehicle` interface. Remember from the introduction that it is technically not possible to have an object being an instance of an interface. The name of the `instanceof` operator is a bit misleading in the case of interfaces.