COMP 249: Object Oriented Programming II

Tutorial 9: Interfaces & Inner Classes

Abstract Classes

An **abstract class** has one or more **abstract methods** (declared without a body), and **may not be instantiated**:

```
abstract public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    abstract public void talk(); // no definition
}

/** In main() */
Person p = new Person("John", 23); // illegal!
```

Abstract Classes

At some point, once a new class has inherited the abstract class and fully implements all of its abstract methods, it (the child class) can be instantiated.

```
/** In main() */

/** Still illegal */
Person p = new Person(“Alex”, 65);

/** Works, because the instance is of WeirdPerson.
The LHS merely tells us which methods we have at
our disposal */
Person p = new WeirdPerson(“Alex”, 65);
```

```
abstract public class Person {
     private String name;
     private int age;

     public Person(String name, int age) {
          this.name = name;
          this.age = age;
     }

     abstract public void talk(); // no definition
}


public class WeirdPerson extends Person {

     public WeirdPerson(String name, int age) {
          super(name, age);
     }

     /** A definition was added, so WeirdPerson is not an
     abstract class, and can be instantiated! */
     public void talk()
     {
          System.out.println(“I think I know how to code well”);
     }
}
```

Interfaces

An **interface** in Java is a type bound by the following constraints:

- It may not be instantiated
- It must be declared `public`
- It may only have `public` members (attributes and methods)
- Attributes are `static` and `final`
- It may only extend another interface

Some key points to keep in mind when implementing an interface:

- A class may implement any number of interfaces
- That class must implement **all the methods** declared in those interfaces
- There is a **conflict** if two interfaces declare methods with the same signature but different return types.
  - This is similar to overriding where the return type is different. It will not work.

Interfaces

An example of a conflict when implementing two interfaces:

```
public interface SomeInterface {
    /** having void return type*/
    public void doSomething(int a, int b);
}

public interface AnotherInterface {
    /** having int return type */
    public int doSomething(int a, int b);
}

/** implementing both are not allowed */
public class ImplementationClass implements SomeInterface, AnotherInterface
{

}
```

Interfaces

An interface can have three types of methods:

1) Abstract
2) Default
3) Static

```
public interface Methods {

    /** An abstract method – no code in its body. It must be
    implemented in the class that implements this interface
    */
    void function1();

    /** A default method – code fully defined. The class that
    implements this interface may decide to override it with
    its own implementation */
    default void function2()
    {
        System.out.println(“Inside function 2”);
    }

    /** A static method – code fully defined. The method
    cannot be static AND abstract because we must be able to
    call the method statically like Methods.function3(), so a
    static method in an interface must have a definition */
    static void function3()
    {
        System.out.println(“inside function 3”);
    }
}
```

Interfaces

If a developer needs to add a method to an interface after it is already in use…:

1) Create a new interface
2) Extend the old one in the new one
3) Add the new method(s) to the new interface
4) Leave it up to your clients to decide if they want to use this new interface.
    1) Why? One or more of the client's classes uses the older interface. If a new method is put in it, those classes won't be able to compile until a definition is written for this newly added interface method.

```
/** You write an interface for a Client and send it to them. The Client implements this interface in one of their classes */
public interface OldInterface {
        void function1();
}
```

```
/** A month later, you want to add a new function (function2) to this interface. You do so, and send it to the Client. Their class no longer compiles because they need to implement function2() in the class first. You are preventing their code from running in this way… Don't do it this way */
public interface OldInterface {
        void function1();
        void function2();
}
```

```
/** You fix the problem by creating a new interface that extends the old, and add the method there. Since the Client never used this NewInterface, their code will continue to compile and run. They can implement this interface when they are ready. */
public interface NewInterface extends OldInterface {
        void function2();
}
```

Inner Classes

An **inner class** is defined inside the body of another class.

- It may be `private` (only the outer class may use it) or `public`
- It may be declared `static`
- Both inner and outer classes may access each other's `private` members
- If the outer class is extended, the inner class can be accessed by the derived class

Creating an instance of an inner class (outside of the outer class):

```
/** If the inner class is not static, an object of the outer class must be created: */
OuterClass outer = new OuterClass();

/** note the position of new */
InnerClass inner = outer.new InnerClass;

/** If the inner class is static, the process is more intuitive */
OuterClass.InnerClass inner = new OuterClass.InnerClass();
```

Coding Exercise 1

You are writing a video game engine, and notice common features between different objects that the player can operate. You decide to create a class hierarchy to represent these similarities.

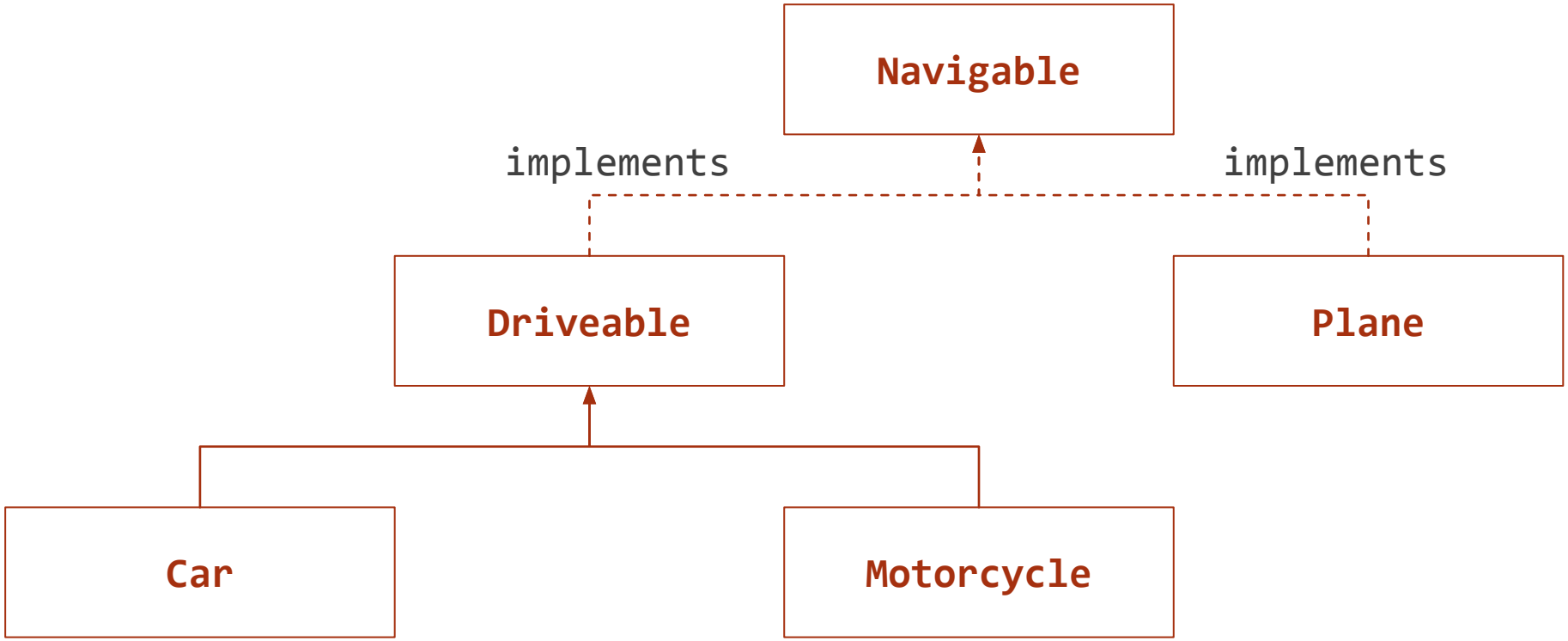
The **`Plane`** and **`Driveable`** classes both implement a **`Navigable`** interface, and make use of the **`startEngine()`** and **`stopEngine()`** methods. The main program loop can invoke them in a seamless way (have them print out simple information about the navigable).

In addition, your system must adhere to the following constraints:

- The **`Plane`** class must have a **`capacity`** attribute;
- The **`Driveable`** class must have the attributes **`machineID`**, and **`make`**;
- Classes **`Car`** and **`Motorcycle`** both extend the **`Driveable`** class, and have **`fuelType`** and **`helmetRequired`** attributes, respectively.

Coding Exercise 1

Below is a simplified UML class diagram for the system:



Coding Exercise 1

Ensure that your program works with the following `main()` method:

```
public static void main(String[] args) {
		// create an array of Navigables
		Navigable navigables[] = new Navigable[5];

		// populate the array with some objects
		navigables[0] = new Plane(300);
		navigables[1] = new Plane(150);
		navigables[2] = new Car(4000, "Lexus", "Hybrid");
		navigables[3] = new Car(4100, "Tesla", "Electric");
		navigables[4] = new Motorcycle(5000, "Ducati", true);

		// output Navigable information
		for (Navigable nav : navigables) {
			nav.startEngine();
			nav.stopEngine();
		}

}
```

Coding Exercise 1

Your output should be similar to:

```
Plane with 300 capacity has taken off
Plane with 300 capacity has landed
Plane with 150 capacity has taken off
Plane with 150 capacity has landed
Lexus has started its engine
Lexus has stopped its engine
Tesla has started its engine
Tesla has stopped its engine
Ducati has started its engine
Ducati has stopped its engine
```

Coding Exercise 2

Below is a class named **InventoryItem**. Each instance of the class has a name and a unique ID:

```
public class InventoryItem {
    private String name;
    private int itemID;
}
```

- Complete the class with appropriate constructors, accessors, and mutators.
- The `itemID` is assigned by the store, and can be set from outside the **InventoryItem** class. Your code does not have to ensure that they are unique.
- Your class should implement the **Comparable** interface.
- The `compareTo()` method should compare `itemID` attributes.

Coding Exercise 2

Test your class using the following **Driver** class, which creates an array of arbitrary **InventoryItem** objects. Complete the `sort()` method, which takes as input an array of **Comparable** objects.

```
public class Driver {
    public static void sort(Comparable[] objects) {
        // place your code here
    }

    public static void main(String[] args) {
        InventoryItem[] inventoryItems = new InventoryItem[5];
        inventoryItems[0] = new InventoryItem("Book", 18);
        inventoryItems[1] = new InventoryItem("Computer", 77);
        inventoryItems[2] = new InventoryItem("Printer", 4);
        inventoryItems[3] = new InventoryItem("Desk", 12);
        inventoryItems[4] = new InventoryItem("Chair", 9);

        sort (inventoryItems);
        for (int i = 0; i < inventoryItems.length; ++i)
            System.out.println(inventoryItems[i]);
    }
}
```