Problem 1. The file *Control.java* contains the most commonly used Java control structures.

The for loop (whose syntax is the same as C) prints the elements of **numArray**. Modify the loop so that it only prints the first 3 elements. Next, change the print statement so that it prints double the value stored in the array.

The syntax of the if statement is also the same as in C. Modify the condition so that the else branch is run.

Java also has another type of for loop, which has the form

```
for(Type variablename : array){
   ...
}
```

This iterates through each element of the array. On each iteration of the loop, the loop variable stores the ***value*** of the current array element, which can be manipulated directly without having to index the array. Perform different operations within the loop to experiment with the behavior of this type of looping. Do not modify the array in the body of the loop - this may cause unexpected behavior. When you are confident you understand the behavior of this loop, modify it so that only numbers greater than 2 are printed.

Problem 2. In the file *Car.java*, create a class `Car` with the following attributes:

- A public member `maker` of type `String`, which stores the maker of the car.
- A public member `engineCapacity` of type `Int` which stores the capacity of the Car's engine.
- A single constructor taking two arguments, a `String` and an `Int` used to initialize the two members above.

Ensure that all files begin with the line

```
package Lab3;
```

Any class defined within this package can be used by any other class in the package. Modify *Control.java* to operate on `Cars` and arrays of `Cars`. When recompiling *Control.java* you will need also need to compile *House.java* simultaneously. How you do this will depend on your environment. If you are running Java locally you can run

```
javac Control.java Car.java
```

You can then run individual files by changing to the *Labs* directory and running, e.g.

```
java Lab3.Control
```

If you are using an IDE or web environment this may automatically be configured, or you may need to do so yourself.

Problem 3. The file *IO.java* demonstrates how to get user input. Input and output in Java are handled using `System.out` and `System.in`, which are instances of the `Stream` class. We have already seen how to print to `System.out`. We can read input from the user using the `Scanner` class. `Scanner` has a method `nextLine` which takes a String as input from the user. Using the Scanner's `nextBoolean` and `nextInt` methods, get boolean and integer inputs from the user and print appropriate responses.

Problem 4. Create two files *Taxicab.java* and *Racecar.java*. Ensure that they are part of the package Lab3. Create in them `Taxi` and `Racecar` classes respectively as subclasses of `Car`. These two classes should be the same as `Car`, with the following changes:

- Taxi should have a method `fare` which takes as input an integer `distance` and returns an integer equal to 20 times the distance, representing the fare charged for a trip of that distance.
- Racecar should have an additional member `topspeed`, giving the top speed of the car as an integer. You will need to write a suitable constructor with topspeed as the third argument.

Take advantage of inheritance to minimize the amount of code you need to write.

Problem 5. (Optional) Modify `Car` to include a public `topspeed` member, initialized by the constructor not as an argument but as one fifth the engine capacity. Run the file `Inheritance.java` and note any unusual behavior.

Problem 6. Open the file *ILG.java*. (ILG is short for Interfaces, Lists, Generics). An interface is like a class, but all its methods are left empty. A class **implements** an interface by providing concrete implementations of all its methods. In this way, interfaces abstract away a class's behavior from its internal representation.

In this file, we import the `List` interface (located in java.util). A list is an ordered collection of object, and provides methods to `add`, `remove` and `get` elements. Since it is just an interface, there is no canonical way of implementing it - we could use a linked list, an array, or any number of other data structures. Lists are homogeneous - they only contain elements of a single type. They are also a **Generic** interface. This means that unlike C, where we have to create a new `List` struct for each type, a single Java class (or interface) suffices for any List. To use a generic class, we provide the type in angle braces `List<Type>`.

We cannot create new `List` objects, since `List` has no concrete implementation. Instead, we can create objects of a class which implements `List`. The `ArrayList` class uses arrays to implement `List`. ArrayList is also generic, so we create a new ArrayList of strings by calling

```
new ArrayList<String>();
```

Observe how lists are manipulated in Java, then change the ILG class to display information about lists of Cars.