Teach yourself Java module
COL106 (Sem I, 2015-16)

Chinmay Narayan and Prathmesh Kallurkar

July 29, 2015

The aim of this Java lab module is to kick-start students in programming Java by introducing them to the basic syntax and useful/essential features of Java. This module is by no means complete reference of Java as it is nearly impossible to squeeze a books worth of material into 2-3 lab session modules. This module is divided in three lab sessions:

- Session 1- Introduction to Java, compiling and executing a java program, basic control structures
- Session 2- String, array, container classes
- Session 3- Exception handling, file handling, Object oriented programming concepts

Each session covers a set of Java features with the help of example programs. Students are advised to compile and execute each of them. A set of programming exercises have been created in every section. Following points must be noted:

- These exercises are hosted on the course website in a tar-ball titled "`JavaModuleExercises.tar.gz`". The tar ball contains 22 directories, one directory for each exercise in the manual.
- There are two type of exercises: (a) *self-help*, and (b) *programming*.

  The *self-help* exercises are marked yellow in the manual. For each self-help exercise, one file is given in the exercise directory: `intro.java`. The `intro.java` file contains the code required to understand the exercise.

  The *programming* exercises are marked blue in the manual. For each programming exercise, three files are given in the exercise directory: `checker.java`,

`program.java, Makefile`. You are supposed to modify the *test()* function in `program.java`. `checker.java` will call the *test* function of `program.java`, and it should return the expected output. You should not modify the signature of the *test* function (a function signature includes the function's return type, the number of arguments, and the types of arguments). Type *make* to compile and execute the exercise. **DO NOT MODIFY** `checker.java`. This file contains the test cases which should pass with your program. At the time of demo, we will change the test cases.

- Completing an exercise means your program has passed all test cases for that exercise. This must be shown to your TA.
- Completing a lab session means completing all exercises of that session.
- If you find any problem in solving the exercises, contact `prathmesh.kallurkar@cse.iitd.ernet.in`

Example for *self-help* exercise:
$ tar -xvf JavaModuleExercises.tar.gz
$ cd JavaLabModule/exercise1
$ javac intro.java
$ ls

Example for *programming* exercise:
$ tar -xvf JavaModuleExercises.tar.gz
$ cd JavaLabModule/exercise8
..... Open program.java, and un-comment line 12 .....
$ make

If the make command is not working, replace it with following commands:
$ javac checker.java
$ javac program.java
$ java checker

Session 1

In imperative programming languages like C, a program is made of a set of functions which are invoked by a main function in some order to perform a task. In some sense, the most basic unit of computation in these languages is function and data. In Java, or in any other Object oriented programming language (OOP), the most basic unit of computation is an object or a class.

Class For any variable x in a programming language, the *type* of x defines what kind of operations are valid on that variable. Java, like every programming language, has a set of `primitive types` to represent *integer*, *character*, *real* etc. Java also allows user to create new types by means of `class` declaration. A `class` is an encapsulation of data and the methods operating on this data in a single unit of type. The notion of `Class` in Java is somewhat similar to `Type` in `SML` programming language and `structure` in `C`. A class in Java is written as below.

Listing 1: An empty java class.

```
1 class intro
2 {
3
4 }
```

A class in Java is called a *public class* if the keyword `public` is added before the keyword `class` in a class definition. For example the code of listing 2 creates a public class `newintro`.,

Listing 2: An empty public java class.

```
1 public class newintro
2 {
3
4 }
```

Object Object is an instantiation of a class type. For the example class of listing 1, to *declare* an object of type `intro`, we write

```
intro var;
```

where `var` is any string. Above syntax declares that `var` is a *reference/handle* of type `intro`. **It is to be noted that the above syntax only creates a handle/reference to an `intro` object. Actual object is not created by this syntax.**. Java provides a keyword `new` to create a new object of a given type and assign it to a reference of that type.

```
intro var = new intro();
```

Above syntax declares a reference variable `var` of type `intro` as well as initialize it with a newly created `intro` object. In absence of `new intro()` part, reference variable is only initialized to `null` and any operation on such uninitialized reference will generate run-time error. **Therefore we must make sure that every reference is properly initialized before using it**.

Fields and Methods A class consists of a set of variables, also called *fields*, and a set of functions, also called *methods*. For an object of a given class, the fields and methods of that class are accessed using `.` operator. A field of a class is of the following form **[Qualifier] type variablename** where `Qualifier` is optional and can take following values,

- public: It means that this field can be accessed from outside the class by just using *objectname.fieldname*.
- static: It means that to access this field you do not need to create an object of this class. Therefore this field is also sometimes called as *class variable/field*.

For the example program in listing 3, `credits` is a `public static` variable and hence can be accessed from outside this class as `intro.credits`. On the other hand, `age` is only a `pubic` variable and hence can only be accessed after creating an object of `intro` type as below.

```
intro var = new intro;
var.age = 10;
```

Above code first creates an object of `intro` class and then assign value 10 to the field `age` of this object. It is to be noted that every object of a class, contains different copies of non-static variable (e.g. `age` in listing 3) and the same copy of static variable (`credits` in listing 3). You do not require to create an object to access the static members of a class.

Listing 3: field qualifiers

```
1 class intro
2 {
3  public int age;
4  public static int credits;
5 }
```

methods A method/function is defined inside a class as **[Qualifier] return-type function-name (arguments)**. A method in Java also specifies *exception signature* which we will discuss in detail in *error handling*. Two important qualifiers for methods are same as for fields,

- public: A public method can be invoked from outside the class using the object of that class.
- static: A static method can be called without creating an object of that class.

Following restrictions are applied on invocation of `static` methods.

- static function can not call non-static functions of that class.
- static function can not access non-static variables/fields of that class.

Every class in Java has a public method by default, it is called `Constructor` of that class. The name of the constructor function is same as the name of the class without any return type, but it can take a set of arguments like any method. This method is called when an object of this class is created using `new` operator as shown earlier.

Listing 4: Filling up methods and fields

```
1  class intro
2  {
3   public int age;
4   public static int credits;
5   public intro(int a)
6   {
7         age = a;
8   }
9   public void setage(int newage)
10    {
11        age = newage;
12    }
13  public static int getcredit()
14    {
15        return credits;
16    }
17 }
```

In the example program of listing 4, static function `getcredit` returns the value of static field `credits`. Function `setage` takes an integer argument and set the value of the field `age` to this value. Return type of `setage` is `void` as it does not return any value, while the return value of `getcredit` is `int`. We also have a constructor in this class which takes an integer and assign it to field `age` of that object. Constructors are mainly used to make sure that all fields of an object are initialized properly at the time of creation. To create an object of class `intro`, we use following syntax,

```
1  intro handler = new intro(4);
```

Notice that we pass an integer (4 in this case) while creating an object with `new` operator. This is because the constructor of class `intro` takes an integer argument. Therefore the constructor of a class, when defined, also tells us what all arguments must be passed while creating an object of that class. If no constructor is defined explicitly, java itself puts a **zero argument constructor** in that class.

Having covered the most essential aspect of a class definition in Java, now let us look at the process of compiling and executing a Java program.

File naming convention for java programs

- Any java program must be saved in a file with extension ".java".
- A java file can contain **at most one** public class.
- If a java file contains a public class then the name of the file **must be same as the name of the public class**.

- If a java file contains no public class then the name of the file **must be same as any of the non-public class** present in that file.
- Smallest non-empty java file, which is compilable, must have at least one class declaration.

Compiling a java program The compiler of Java is a program called "javac". To compile a java program which is saved in a file, say "first.java", execute the following command on shell/command line.

```
javac first.java
```

As a result of this compilation a set of ".class" files are generated, one for each class declared in "first.java". These files are like compiled object files of "C".

Exercise 1: Compile the code of listing 4. What file name will you use to save and compile this program? What all new files are generated by compilation?

Executing a java program Once you get a set of class files after compiling a java program, you execute the program with the following command.

```
java first
```

where "first" is the name of the public class, if any, which was declared in "first.java". It is to be noted that **to execute a java program it must contain at least one public class with a `public static` method *main* declared in it**, as below.

```
1 public static void main(String args[])
2 {
3
4 }
```

Why a method main is needed? Method `main` is like entry point for a java program. It must be `static` so that the java runtime do not need to instantiate any object to executed it. It must be `public` so that it can be invoked by java runtime.

Exercise 2: Try to execute the program of listing 4 and check what error do you get? Now add function `main` (of same syntax as given above) in the class `intro` of listing 4 and again try to compile and execute it.

Using library functions/helper functions defined elsewhere In **C**, helper functions or library functions (like `printf`, `scanf` etc.) are used by including appropriate header file with **# include** syntax. In Java, the equivalent of a header file is *Package* which is included in your java code using **import** syntax. A package name, say *X.Y.Z* denotes a directory structure where the topmost directory is named as X, inside it there is a subdirectory named Y which in

turn contains a compiled java file Z.class. The syntax `import X.Y.Z` in a java program makes all public functions of class Z.class available to this program. One most important package which contain the helper functions for console input/output (like `printf` and `scanf`) is `java.lang`. This package is include by default in every java program.

Now we are ready to write, compile and execute our first java program which prints a string on command line/ console.

Listing 5: Printing "hello world" on screen

```
1  public class intro
2  {
3   public static void main(String args[])
4     {
5          System.out.println("Hello world");
6     }
7   }
```

Exercise 3: Compile and execute the program of listing 5.

We can modify the program of listing 5 as following

- Create a public function in `class intro` which prints *Hello world* on screen.
- Create an object of this class in function `main` and invoke/call the function define above on this object.

Along the same line,

Exercise 4: Complete the program of listing 6 (in place of $\cdots$) such that it prints "Hello world" on console.

Listing 6: Modified hello world program

```
1  public class intro
2  {
3   public void printinfo()
4   {
5        System.out.println("Hello world");
6   }
7   public static void main(String args[])
8     {
9        ...
10       ...
11    }
12  }
```

Reading from console To read from command line we use the class `Scanner` which is defined in the package **java.util.Scanner**. Following program takes one string (name) and an integer (age) from the command line and store it in two fields.

Listing 7: Reading from command line

```
1 import java.util.Scanner;
2  public class intro
3  {
4   public String name;
5   public int age;
6
7   public static void main(String args[])
8     {
9       Scanner s = new Scanner(System.in);
10      intro obj = new intro();
11
12      System.out.println("Enter your name");
13      obj.name = s.nextLine();
14      System.out.println("Enter your age ");
15      obj.age = s.nextInt();
16      s.close();
17
18      System.out.println("Name: " + obj.name);
19      System.out.println("Age: " + obj.age);
20    }
21  }
```

- In line 1, we import the package `java.util.Scanner` which contain the class `Scanner` and associated methods.
- We have two public fields, `name` and `age`, declared in class `intro`.
- In main, we create a Scanner object with argument `System.in` which represents standard input. We will see later in *file handling* section that how the same scanner class, just by changing this input stream parameter, can be used to read data from files.
- We create an object of `intro` class in line 10.
- As it is clear from the names, `nextLine()` and `nextInt()` stops at command prompt and allows user to enter a string and integer respectively.
- Notice the + operator in `System.out.println`. This operator is used to concatenate two strings. Even though `obj.age` is of integer type, java implicitly converts it into string.

Exercise 5: Compile the code of listing 7 and check the output.

loop and condition constructs Java has three constructs for implementing loops,

- `do{Statements} while(boolean condition);`,
- `while(boolean condition){Statements}`, and
- `for(initialization; terminating condition; post-operation){Statements}`.

Listing 8 shows a program to print all numbers from 10 to 100 using all three looping constructs.

Listing 8: Different looping constructs in Java

```
1  public class intro
2  {
3
4   public static void main(String args[])
5     {
6       int i=10;
7       System.out.println("Using do-while");
8       do{
9           System.out.println(i);
10          i++;
11        }while(i<100);
12
13
14      for(i=10; i< 100; i++){
15          System.out.println(i);
16        }
17
18      i=10;
19      while(i<100){
20          System.out.println(i);
21          i++;
22        }
23
24
25
26    }
27  }
```

For selection (if then else), java provides two constructs.

- `if(condition) {Statements} else {Statements}`
- Switch-case. `break` is used to break the flow of program once a case is satisfied.

An example program to check if a given number is even or odd is implemented using both conditional constructs in listing 9.

Listing 9: Different conditional constructs in Java

```
1 import java.util.Scanner;
2  public class intro
3  {
4
5   public static void main(String args[])
6     {
7
8       Scanner s = new Scanner(System.in);
9       System.out.println("Enter a number");
10      int inp = s.nextInt();
11
12      System.out.println("Using if-else construct");
13      if(inp % 2 == 0)
14            System.out.println(inp + " is even");
15      else
```

```
16              System.out.println(inp + " is odd");
17
18        System.out.println("Using switch case construct");
19        switch(inp % 2){
20          case 0:
21                  System.out.println(inp + " is even");
22                  break;
23          case 1:
24                  System.out.println(inp + " is odd");
25                  break;
26        }
27      }
28   }
```

Exercise 6: Compile and execute the program of listing 9. After that remove the `break` of line 22 and again execute this modified program. What is the difference?

Passing arguments In Java, all arguments are **passed by value**. What does it mean for non-primitive types? As we saw earlier that when we create a variable of a non-primitive type (abstract data type/ class) then effectively we only create a reference to an object of that type. For example, the syntax

```
intro introobj;
```

only creates a handle/reference `introobj` of the type `intro`. No object is created as a result of this declaration. To create an object and assign that object to this reference, we use `new` keyword.

```
introobj = new intro();
```

When we pass an object as an argument to another function, we only pass reference/handle of that object. This reference is also **passed as value** but because this is a reference hence any change in the called function using this reference changes the original object as well.

It is to be noted that the declaration of primitive type (like **int**, **float** etc.) variables does not create reference/handle but creates the actual storage for that variable. When a primitive type variable is passed as an argument to another function, it is **passed by value**. But because actual value is copied to the called function, instead of the reference, hence any change done to this copy is not reflected back in the original copy of this variable in calling function. Let us try to understand it with the help of an example in listing 10.

Listing 10: Parameters are always passed by value; Objects are no exception

```
1 import java.util.Scanner;
2  class holder{
3          public int old;
4  }
5
6  public class intro
7  {
8
```

```
 9   public static void main(String args[])
10     {
11
12       holder h;
13       h = new holder();
14       h.old = 20;
15       System.out.println("Value of old " + h.old);
16
17       update(h,30);
18       System.out.println("Value of old " + h.old);
19
20       }
21
22    static void update(holder k, int newval)
23      {
24         k.old = newval;
25      }
26     }
```

Exercise 7: Compile and execute the program of listing 10.

Exercise 8: Leap year- Given a year, return true if it is a leap year otherwise return false. Please note that years that are multiples of 100 are not leap years, unless they are also multiples of 400.

Exercise 9: Same last digit- Given two non-negative integers, return true if they have the same last digit, such as with 27 and 57. Note that the % "mod" operator computes remainder, so 17%10 is 7.

Exercise 10: Least common multiple- Given two integers n and m, the objective is to compute the LCM (least common multiple) of n and m. LCM is the smallest number that is divisble by both n amd m. For e.g. is n is 12 and m is 14, the LCM is 84. If n is 32 and m is 16, the LCM is 32.

Exercise 11: Roots of polynomial- Write a Java program that given b and c, computes the roots of the polynomial x*x+b*x+c. You can assume that the roots are real valued and need to be return in an array.

Session 2

Arrays In java arrays are declared as

Listing 11: Array declaration in Java

```
1  int[] record;
2  float[] points;
3  Student[] sectionb;
```

Notice that you can not specify number of elements in array by only declaring the array variable. A declaration as in listing 11 only declares that the variable is an array of given type.

```
int[] record = new int[5];
```

declares that the variable `record` is an array of integer type and also allocates the space of 5 integers in this array. Similarly,

```
int[] record = {1,2,3};
```

declares that the variable `record` is an array of integer type and contains elements 1,2 and 3. Its size has been fixed by this declaration to three elements. Similarly,

```
Student[] sectionb = new Student[5];
```

declares that the variable `sectionb` is an array of `Student` type and also allocates the **space/slots for 5 Student objects** in this array. **Notice that this declaration does not create individual object for each of these 5 slots. You must create 5 Student objects by `new` keyword and assign them to these slots explicitly, otherwise these slots will remain uninitialized and therefore contain `null` instead of `Student` objects.**

Given an array variable, say `record`, the length of the array is given by the field `length`, as in listing 12.

Listing 12: Manipulating arrays in Java

```
1 import java.util.Scanner;
2
3  public class intro
4  {
5   public static void main(String args[])
6     {
7         Scanner s = new Scanner(System.in);
8         System.out.println("Enter a number n");
9         int n = s.nextInt();
10
11        int[][] data = new int[n][n+2];
12        System.out.println("number of rows = " + data.length
13                + " and number of columns = " + data[0].length);
14        for(int i=0 ; i< data.length; i++){
15                for(int j=0; j < data[0].length; j++)
16                        data[i][j] = 1;
17        }
18
19        int[][][] datam = new int[n][n+2][n+4];
```

```
20          System.out.println("Length along outermost indices = "
21                  + datam.length);
22          System.out.println("Length along middle indices = "
23                  + datam[0].length);
24          System.out.println("Length along innermost indices = "
25                  + datam[0][0].length);
26
27          for(int i=0 ; i< datam.length; i++){
28                  for(int j=0; j < datam[0].length; j++)
29                          for(int k=0; k< datam[0][0].length; k++)
30                          datam[i][j][k] = 1;
31          }
32
33
34      }
35 }
```

- Program of listing 12 reads an integer from user and create a two dimensional integer array `data` at line 11 with number of rows equal to `n` and number of columns equal to `n+2`.
- It is important to notice the use of `length` field of array variable for finding out the length along different dimensions of a multi-dimensional array.
- `data.length` gives length of the array variable `data` along outermost dimension.
- You keep on adding index 0 to array variable to get the reference along that dimension. Field `length` on that reference gives the length along that dimension as shown in lines 20, 22 and 24. This point is important when you want to traverse along different dimensions of an array as shown in the loops of line 14 and 27.

Exercise 12: Compile and execute the program of listing 12.

strings and associated helper functions In java, strings are declared as below.

```
String name;
```

where `name` is a reference to an object of string type. To create actual string object and assign that to the reference `name`, we need to do the following,

```
name = "CSL 201";
```

Initialization can be done either at the time of declaration of reference or later.

Listing 13: Manipulating Strings in Java

```
1 import java.util.Scanner;
2
3 public class stringex
4 {
```

```
5   public static void main(String args[])
6     {
7          Scanner s = new Scanner(System.in);
8          String line = s.nextLine();
9          String newline = "";
10         for(int i=0; i< line.length(); i++)
11         {
12              if (line.charAt(i)!=' ')
13                 newline = newline + line.substring(i,i+1);
14         }
15         System.out.println("String after removing all spaces: " +
16                 newline);
17
18
19    }
20 }
```

Consider an example of listing 13. It reads a line from console and store it in a string variable `line`. Then it iterates over all characters of this string and check if the character at any index (using `charAt` function) is equal to whitespace (`' '`). If it is not equal to whitespace then it concatenates the string representation of that character (obtained by `substring` function) in a temporary string `newline`. When loop terminates, `newline` points to a string which is same as the original one without any whitespace. Function `substring` takes two arguments, starting index and end index, and return the substring present in between them. This substring does not include the character present at the end index.

To get more useful functions on `String` class of Java, please refer to the following link.

`http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/String.html`

This link/site host the documentation of all classes and associated functions of Java language. If you want to know about any function of any utility class, say `Date`, `File`, `String` etc., just go to the above link and look for the appropriate class name and its member functions.

Container classes in Java We are going to look at two very important *container* classes provided by Java, `Vector` and `HashMap`.

Vector `Vector` is equivalent to expandable array. You can create a `Vector` object without defining the length of it. The length of the `Vector` object keeps on adjusting based on the current size of this `Vector` object. To specify that a `Vector` object contains objects of a particular type, we instantiate the object as following,

```
Vector<Integer> v = new Vector<Integer>();
```

It is to be noted that vector stores a set of objects derived from *Object* class and because primitive types like `int`, `float` etc are not derived from `Object` class therefore in that case we use corresponding wrapper classes `Integer`, `Float` etc. Above syntax create a new `Vector` object to contain integers. Notice that we do not specify the length of the vector. To add an element to a vector, we invoke `add` function on vector object.

```
v.add(3);
v.add(4);
```

Java also provides a class called `Collection` which has a set of algorithm, like *sort*, *swap* etc. These algorithms give a very concise way of operating on `Vector` class for sorting, swapping and other functions. For example the program in listing 14 sorts a vector of integers and then reverse the order of elements in a vector.

Listing 14: Vectors in Java

```
1 import java.util.*;
2
3 public class vectorex
4 {
5  public static void main(String args[])
6    {
7        Vector<Integer> v = new Vector<Integer>();
8        v.add(3);
9        v.add(1);
10       v.add(7);
11       v.add(5);
12       Collections.sort(v);
13       for(int i=0; i< v.size(); i++)
14       {
15               System.out.println(v.get(i));
16       }
17
18       Collections.reverse(v);
19       for(int i=0; i< v.size(); i++)
20       {
21               System.out.println(v.get(i));
22       }
23    }
24 }
```

More about collection class and the associated algorithms can be read from the following link.

```
http://docs.oracle.com/javase/tutorial/collections/
http://docs.oracle.com/javase/tutorial/collections/algorithms/index.html
```

A very important concept in Java, for handling traversal of container classes like `Set`, `List`, `Vector`, `HashMap` etc., is the notion of `Iterator`. An iterator over a container class is an object of class `Iterator` which allows us to traverse over the elements of that container class without revealing the internal structure. For example, a `List` class might be constructed using array or linked list, but the user of class `List` can traverse it with the help of `Iterator` object of that class. The function `hasNext()` of `Iterator` object allows us to check if a container contains any more element or not. Another function `next()` returns the next element (with respect to the current position of iterator in that container). Program in listing 15 traverses a vector using `Iterator`.

Listing 15: Iterator over vector

```
1 import java.util.*;
```

```
2
3 public class vectorex
4 {
5  public static void main(String args[])
6    {
7        Vector<Integer> v = new Vector<Integer>();
8        v.add(3);
9        v.add(1);
10       v.add(7);
11       v.add(5);
12       Iterator<Integer> it = v.iterator();
13       while(it.hasNext()){
14               System.out.println(it.next());
15       }
16
17    }
18 }
```

There are few things to be noted about `Iterator` as shown in listing 15.

- The type of Iterator must match with the **type of elements in the container class which is to be traversed**. Therefore in line 12 of listing 15, we declare `it` as a reference of `Iterator` class and this class is instantiated with `Integer` class. This is because we want to store the iterator of the vector `v`, which is instantiated with `Integer` type, in `it`.
- Function `it.next()` returns the element at the current position of iterator in `v` and increment the current position.
- Function `it.hasNext()` checks if the current position contains any element or not.

HashMap Class `HashMap` is a container class to store **key-value** pair mapping. This is useful in those scenarios where we want to store a mapping between two elements of same or different types, also called as **key** and **value**, such that only one **value** can be associated with a **key**. This container class also provides fast retrieval of **value** given a **key**. Similar to the `Vector` class, this class is also instantiated with the type of **key** and **value**. For example,

```
HashMap<Integer, String> hm = new HashMap<Integer, String>();
```

creates an object of `HashMap` type such that the key is of `Integer` type and the value is of `String` type. Listing 16 shows a program which uses HashMap to store Integer-String pair. It also uses iterator to iterates over all elements of a HashMap.

Listing 16: HashMap example

```
1 import java.util.*;
2
3 public class hashex
4 {
5  public static void main(String args[])
6    {
7        HashMap<Integer, String> hm = new HashMap<Integer, String>();
8        hm.put(3, "three");
```

```
9           hm.put(5, "five");
10          if(hm.get(3)!=null)
11                  System.out.println("3 is present with value = " +
12                          hm.get(3));
13          else
14                  System.out.println("3 is absent");
15
16          if(hm.get(4)!=null)
17                  System.out.println("4 is present with value = " +
18                          hm.get(4));
19          else
20                  System.out.println("4 is absent");
21
22          Iterator<Integer> it= hm.keySet().iterator();
23          while(it.hasNext())
24          {
25                  Integer tmp = it.next();
26                  System.out.println("key = " + tmp + " value = " +
27                          hm.get(tmp));
28          }
29      }
30 }
```

- Function `put` adds a key-value pair to hashmap.
- Function `get` returns the value associated with the given key, if any. If there is no value stored in hashmap for this key then `null` is returned. (as checked in line 10 and 16 of listing 16).
- Function `keySet()` returns the set of keys stored in a hashmap. In line 22, we get the iterator of this set and store it in `it`. After that we iterate over this set, as we did in listing 15, to find all key-value pairs stored in hashmap `hm`.

Exercise 13: What happens if we modify the last while loop of listing 16 as below.

Listing 17: Modified while loop

```
1           while(it.hasNext())
2           {
3           System.out.println("key = " + it.next() + " value = " +
4                   hm.get(it.next()));
5           }
```

Exercise 14: Average of numbers- Given an array of integers finds the average of all the elements. For e.g. for {4,7,9,4} the average is 6 and for {1,3,8,5} the average is 4.25. Please note that if the array has no elements, the average should be 0.

Exercise 15: Remove zeros- Given an array of integers return an array in the same order with all 0's removed. For e.g. is the input is {1,2,3,4,5,0,1,2,0,0,2} the expected output is {1,2,3,4,5,1,2,2}. If the input is {0,0,1,2} the output is {1,2}. If the input is {0,0,0,0} the expected output is {}.

Exercise 16: Hex to binary- Given a string representing a number in hexadecimal format, convert it into its equivalent binary string. For e.g. if the input if "1F1" then its binary equivalent is "111110001". If the input is "13AFFFF", the output should be "100111010111111111111111".

Exercise 17: Java files- You have been given the list of the names of the files in a directory. You have to select Java files from them. A file is a Java file if it's name ends with ".java". For e.g. "FileNames.java" is a Java file, "FileNames.java.pdf" is not. If the input is {"can.java","nca.doc","and.java","dan.txt","can.java","andjava.pdf"} the expected output is {"can.java","and.java","can.java"}

Exercise 18: Most frequent digit- Given a number, the objective is to find out the most frequently occuring digit in the number. If more than 2 digits have the same frequency, return the smallest digit. The number is input as a string and the output should be the digit as an integer. For e.g. if the number is 12345121, the most frequently occuring digit is 1. If the number is 9988776655 the output should be 5 as it is the smallest of the digits with the highest frequency.

Exercise 19: Matrix addition- Given two matrices M1 and M2, the objective to add them. Each matrix is provided as an int[][], a 2 dimensional integer array. The expected output is also 2 dimensional integer array.

Exercise 20: String concat- Given two strings s1,s2 and two indices m,n return a string having chars from index m to end of s1 and then 0 to n of s2 (both m and n are inclusive). For eg, if s1="hello", s2="world", m=3, n=0, then answer is "low"

Exercise 21: Score Marks- The "key" array is an array containing the correct answers to an exam, like {'a','a','b','c'}. The "answers" array contains a student's answers, with '?' representing a question left blank. The two arrays are not empty and are the same length. Return the score for the provided array of answers, giving a +4 for each correct answer, -1 for each incorrect answer and 0 for each blank answer. For e.g.
key = {'a','c','d','b'}
answers = {'c','c','?','b'}
then score is -1 + 4 + 0 + 4 = 7

Session 3

Error handling- Exceptions Java provides error handling by way of *Exceptions*. A function, during its execution, can `throw` as set of exceptions. For each method description in `Javadoc`, a set of exceptions thrown by that method are also specifed as shown below.

getAbsolutePath

public String getAbsolutePath()
Returns the absolute pathname string of this abstract pathname. If this abstract pathname is already absolute, then the pathname string is simply returned as if by the getPath() method. If this abstract pathname is the empty abstract pathname then the pathname string of the current user directory, which is named by the system property user.dir, is returned. Otherwise this pathname is resolved in a system-dependent way. On UNIX systems, a relative pathname is made absolute by resolving it against the current user directory. On Microsoft Windows systems, a relative pathname is made absolute by resolving it against the current directory of the drive named by the pathname, if any; if not, it is resolved against the current user directory.

Returns:
The absolute pathname string denoting the same file or directory as this abstract pathname
Throws:
SecurityException - If a required system property value cannot be accessed.

When such function is called then the caller has to make sure that if this function fails and throws an exception then that exception must be handled properly in the caller. Java provides `try-catch` syntax to handle this issue. Any call to a method, which can thrown an exception, must be surrounded by `try-catch` block as shown in listing 18.

Listing 18: Handling exception using try-catch block

```
1 import java.util.*;
2 public class arrayex
3  {
4   public static void main(String args[])
5     {
6         int[] data = new int[20];
7
8         try{
9                 for(int i=0 ; i < 21; i++)
10                        data[i] = 2*i;
11        }
12        catch(ArrayIndexOutOfBoundsException en)
13        {
14                System.err.println("error in accessing array index");
15        }
16    }
```

```
17 }
```

Here `ArrayIndexOutOfBoundsException` is an exception class. `en` is an object of this exception class which was thrown by runtime while executing the above program. If There were no try-catch surrounding this code then the program would immediately come out from execution because of not finding any handler for this exception. Exception handlers allow programmer to handle the case of expected or unexpected errors and gracefully proceed or abort as required.
To read further about exception and exception hierarchy, please refer to the following link.

`http://docs.oracle.com/javase/tutorial/essential/exceptions/`

File handling Example program in listing 19 print the content of this file.

Listing 19: Reading file

```
1 import java.util.*;
2 import java.io.*;
3 public class fileex
4  {
5   public static void main(String args[])
6     {
7                 try {
8                          FileInputStream fstream =
9                                 new FileInputStream("file.txt");
10                         Scanner s = new Scanner(fstream);
11                         while (s.hasNextLine())
12                                  System.out.println(s.nextLine());
13                } catch (FileNotFoundException e) {
14                        System.out.println("File not found");
15                }
16    }
17 }
```

- See the similarity of the code of listing 19 with the code of listing 7. In listing 7 we were reading from console (standard input) and hence we instantiated Scanner object with `System.in` object.
- In listing 19 we want to read from file and therefore we first open a file input stream as in line 8. After that we pass this object to Scanner class to instantiate a new Scanner object, in line 9.
- After that we use a while loop to print all lines present in the given file.
- It is clear that you must have a file named "file.txt" in current directory in which you are going to execute this program. If it is not the case then you will see the error message of catch block (File not found).

Now consider another example in which we write a sequence of comma separated integers, from 1 to 100, in a file.

Listing 20: Writing in a file

```
1 import java.util.*;
2 import java.io.*;
3 public class fileex
4  {
5  public static void main(String args[])
6     {
7       try {
8             FileOutputStream fs = new FileOutputStream("out.txt",true);
9             PrintStream p = new PrintStream(fs);
10            int i=1;
11            while(i<100){
12                  p.print(i + ",");
13                  i++;
14            }
15            p.print(i);
16        } catch (FileNotFoundException e1) {
17                 System.out.println("File not found");
18                 }
19    }
20 }
```

- In the program of listing 20, we first create an output stream object (we created an input stream object for reading in listing 19).
- Constructor of `FileOutputStrem` class takes two arguments. First one specifies the filename that must be opened for writing. If the second argument is true it means that the new content must be appended to the existing content of this file (when true). When this argument is false, it implies that the new content will overwrite the old content of this file.
- After that we pass this object `fs` of `FileOutputStream` class to `PrintStream` class constructor and create a new object `p`, in line 9.
- After that we invoke `print` function on this `PrintStream` object to write integers 1 to 100 separated by comma in between.

For further reading on file IO in Java, please look at the following link.

`http://docs.oracle.com/javase/tutorial/essential/io/fileio.html`

Object oriented part of Java Even a modest introduction of Java will not be complete without discussing that part of its syntax which helps in object oriented programming. In this last part of lab session, we wish to give an overview of syntax and semantics of OOPs concepts in Java.

Inheritance It must be clear now that a java program is a collection of classes/objects which work together (by means of method invocation) to implement a given specification. Each `class` is essentially a collection of a set of fields (data) and a set of methods operating on this data. The functionality of a class is mainly determined by the methods defined in that class. Sometime, we need a `class` which borrows most of its required functionalities from some existing class. In that case we have two choices; either to copy paste those methods from existing class to this newly created class or to inherit the new class from this existing class. `Inheritance` is a mechanism which allows a class to borrow

already implemented functionalities in another class without copying the code. It is to be noted that the concept of `Inheritance` is central to nearly all object oriented programming languages. Each language might only differ in the syntax for implementing `inheritance`. In java, we use a keyword `extends` to inherit the functionalities of a class as given below.

```
class newc extends oldc
{


}
```

Here `newc` is a **derived class/ subclass** of the **base class/super class/ parent class** `oldc`. After inheritance, all public fields and public methods of the base class are inherited by the derived class. If derived class has a function `f` having same name and argument types as in the base class then function `f` is said to **override** the base class implementation. Consider the program of listing 21.

Listing 21: Inheritance and function overriding example

```
1 class first {
2  public int data;
3  public void test(){
4   System.out.println("Function:test of base class");
5  }
6 }
7
8 public class second extends first {
9  public void newtest(){
10    System.out.println("Newly added function in derived class");
11  }
12
13  public void test(){
14    System.out.println("Function:test of derived class");
15  }
16
17  public static void main(String args[])
18  {
19    first basec = new second();
20    first based = new first();
21
22    basec.test();
23    based.test();
24  }
25
26 }
```

- In above example (listing 21), we first create a class `first` with public field `data` and public method `test`.
- In line 8, we create another class `second` which inherits from `first`.
- This class, `second`, overrides the function `test` and define a new function `newtest`.
- In line 19 of main, we create an object of class `second` and assign it to `basec`, a reference of class `first`.

- In line 20 of main, we create an object of class `first` and assign it to `based`, a reference of class `first`.
- Then we invoke function `test` on these two references and check the output.
- Function call in line 22 invokes `test` method of derived class even though the reference was of base class. This is an example of changing the behaviour (invoking different functions) based on the actual object (`second` in this case) stored in the reference variable (`basec` in this case). This is an example of Function Polymorphism.
- Function call in line 23 invokes `test` method of base class because the actual object stored in `based` reference is also of type `first`.

Other subtle points involved in *Inheritance* and *Polymorphism* can be read from the following link.

`http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html`

Function overloading In Java, you can define more than one function having same name in a class. This will not produce a compiler error as long as all arguments of these functions are not of same types. While invoking such function, compiler picks the correct function based on the type of arguments passed to these functions. This concept is referred to as function overloading and is a well known concept of object oriented programming languages.

Listing 22: Function overloading example

```
1 public class funoverload {
2
3  public void test(){
4     System.out.println("Function:test with no parameter");
5  }
6
7  public void test(int a){
8     System.out.println("Function:test with one parameter");
9  }
10
11  public static void main(String args[])
12  {
13         funoverload fun = new funoverload();
14         fun.test();
15         fun.test(3);
16  }
17
18
19
20 }
```

Exercise 22: 1. Compile and execute the program of listing 22.
2. Add following public function in class `funoverload` and then compile it.

```
1 public int test()
2 {
3     System.out.println("Function:tst with one integer parameter");
4 }
```

- Function overloading works only when the argument list of two functions, having same name, does not match.
- It is not possible to overload a function with another function having same name and argument list but different return type, as shown by above exercise.

Interface and implementing an intereface In large projects, where many developers are involved, different people write different module (functionalities). In order to make sure that the code developed by one developer is easiliy integrated with the code developed by another developer, both of them must agree on a set of common notations (function names, argument lists, return values etc.). In case of java, this is achieved by specifying an `Interface` class. An `Interface` is very similar to a class except that it only contains methods without any implementation. An `Interface` can also contain constants. For an interface, any class implementing that interface must provide definitions for all functions defined in that `Interface`. For example consider the program of listing 23.

Listing 23: Interface example

```
1 interface myinterface
2 {
3  public void test();
4 }
5
6 class secondimpl implements myinterface {
7
8  public void test(){
9     System.out.println("Function:test in second implemented class");
10  }
11
12 }
13
14
15 public class interfaceex implements myinterface {
16
17  public void test(){
18     System.out.println("Function:test in first implemented class");
19  }
20
21  public static void main(String args[])
22  {
23          myinterface fun = new interfaceex();
24          myinterface fun2 = new secondimpl();
25          fun.test();
26          fun2.test();
27  }
28
29 }
```

- Above example declares an interface with keyword `interface`. We **can not** provide any implementation of the functions defined in an interface.
- Then we create two classes, `secondimpl` and `interfaceex`, implementing this interface.
- These classes **must** implement (give definition) the function `test` which is declared in the interface.
- In main, we create objects of these two clasess and store them in the reference of `interface` (in line 23,24).
- When we invoke function `test` on this reference, based on which object was stored in it, test function of `secondimpl` or `interfaceex` is executed.
- It is to be noted that we can not have something like this, `myinterfaceex f = new myinterfaceex()` in our code.

For further reading about `interface` please look at the following link.

`http://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html`

This completes the broad overview/tutorial of java features which you might require during your assignments. In case of any difficulty in Java, internet (google) is your best friend.