How are Java objects stored in memory?

In Java, all objects are dynamically allocated on Heap. This is different from C++ where objects can be allocated memory either on Stack or on Heap. In C++, when we allocate abject using new(), the abject is allocated on Heap, otherwise on Stack if not global or static.

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new(). So the object is always allocated memory on heap

For example, following program fails in compilation. Compiler gives error *"Error here because t is not initialed".*

```
class Test {
    // class contents
    void show() {
        System.out.println("Test::show() called");
    }
}

public class Main {
    public static void main(String[] args) {
        Test t;
        t.show(); // Error here because t is not initialed
    }
}
```

Allocating memory using new() makes above program work.

```
class Test {
    // class contents
    void show() {
        System.out.println("Test::show() called");
    }
}

public class Main {
    public static void main(String[] args) {
        Test t = new Test(); //all objects are dynamically allocated
        t.show(); // No error
    }
```

What are C++ features missing in Java?

Following features of C++ are not there in Java.

No pointers
No sizeof operator
No scope resolution operator
Local variables in functions cannot be static
No Multiple Inheritance
No Operator Overloading
No preprocessor and macros
No user suggested inline functions
No goto
No default arguments
No unsigned int in Java
No -> operator in java
No stack allocated objects in java
No delete operator in java due to java's garbage collection
No destructor in java
No typedef in java
No global variables, no global function because java is pure OO.
No friend functions
No friend classes
No templates in java

final variables in Java

In Java, when final keyword is used with a variable of primitive data types (int, float, .. etc), value of the variable cannot be changed.

For example following program gives error because i is final.

```
public class Test {
    public static void main(String args[]) {
        final int i = 10;
        i = 30; // Error because i is final.
    }
}
```

When final is used with non-primitive variables (Note that non-primitive variables are always references to objects in Java), the members of the referred object can be changed. final for non-primitive variables just mean that they cannot be changed to refer to any other object

```
class Test1 {
   int i = 10;
}

public class Test2 {
    public static void main(String args[]) {
      final Test1 t1 = new Test1();
      t1.i = 30;  // Works
    }
}
```

Do we need forward declarations in Java?

Predict output of the following Java program.

```
// filename: Test2.java

// main() function of this class uses Test1 which is declared later in
// this file
class Test2 {
    public static void main(String[] args) {
         Test1 t1 = new Test1();
         t1.fun(5);
    }
}
class Test1 {
    void fun(int x) {
        System.out.println("fun() called: x = " + x);
    }
}
```

Output:

```
fun() called: x = 5
```

The Java program compiles and runs fine. Note that *Test1* and *fun()* are not declared before their use. Unlike C++, we don't need forward declarations in Java. Identifiers (class and method names) are recognized automatically from source files. Similarly, library methods are directly read from the libraries, and there is no need to create header files with declarations. Java uses naming scheme where package and public class names must follow directory and file names respectively. This naming scheme allows Java compiler to locate library files.

(Widening Primitive Conversion -Java)

Here is a small code snippet given. Try to Guess the output

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.print("Y" + "O");
        System.out.print('L' + 'O');
    }
}
```

At first glance, we expect "YOLO" to be printed.

Actual Output:
"YO155".

Explanation:
When we use double quotes, the text is treated as a string and "YO" is printed, but when we use single quotes, the characters 'L' and 'O' are converted to int. This is called widening primitive conversion. After conversion to integer, the numbers are added ('L' is 76 and 'O' is 79) and 155 is printed.

Now try to guess the output of following:

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.print("Y" + "O");
        System.out.print('L');
        System.out.print('O');
    }
}
```

Output: YOLO

Explanation: This will now print "YOLO" instead of "YO7679". It is because the widening primitive conversion happens only when '+' operator is present.

Widening primitive conversion is applied to convert either or both operands as specified by the following rules. The result of adding Java chars, shorts or bytes is an **int**:

- If either operand is of type double, the other is converted to double.
- Otherwise, if either operand is of type float, the other is converted to float.
- Otherwise, if either operand is of type long, the other is converted to long.
- Otherwise, **both operands are converted to type int**

String Class in Java

String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

Creating a String

There are two ways to create string in Java:

- ***String literal***

```
String s = "HelloWorld";
```

- **Using *new* keyword**

```
String s = new String ("HelloWorld");
```

String Methods

- **int length():** Returns the number of characters in the String.

```
"HelloWorld".length();  // returns 10
```

- **Char charAt(int i):** Returns the character at i^{th} index.

```
"HelloWorld".charAt(3); // returns  'l'
```

- **String substring (int i):** Return the substring from the i^{th} index character to end.

```
"HelloWorld".substring(3); // returns "loWorld"
```

- **String substring (int i, int j):** Returns the substring from i to j-1 index.

```
 "HelloWorld".substring(2, 5); // returns "llo"
```

- **String concat(String str):** Concatenates specified string to the end of this string.

```
String s1 = "Hello";
String s2 = "World";
String output = s1.concat(s2); // returns "HelloWorld"
```

- **int indexOf (String s):** Returns the index within the string of the first occurrence of the specified string.

```
String s = "Learn Share Learn";
int output = s.indexOf("Share"); // returns 6
```

- **int indexOf (String s, int i):** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

```
String s = "Learn Share Learn";
int output = s.indexOf('a',3);// returns 8
```

- **Int lastindexOf(int ch):** Returns the index within the string of the last occurrence of the specified string.
- `String s = ”Learn Share Learn”;`
- `int output = s.lastindexOf(‘a’); // returns 14`
- **boolean equals(Object otherObj):** Compares this string to the specified object.
- `Boolean out = “Geeks”.equals(“Geeks”); // returns true`
- `Boolean out = “Geeks”.equals(“geeks”); // returns false`
- **boolean equalsIgnoreCase (String anotherString):** Compares string to another string, ignoring case considerations.
- `Boolean out= “Geeks”.equalsIgnoreCase(“Geeks”); // returns true`
  `Boolean out = “Geeks”.equalsIgnoreCase(“geeks”); // returns true`

- **int compareTo(String anotherString):** Compares two string lexicographically.
- `int out = s1.compareTo(s2);  // where s1 ans s2 are`
- `// strings to be compared`
-
- `This returns difference s1-s2. If :`
- `out < 0  // s1 comes before s2`
- `out = 0  // s1 and s2 are equal.`
- `out >0   // s1 comes after s2.`
- **int compareToIgnoreCase(String anotherString):** Compares two string lexicographically, ignoring case considerations.
- `int out = s1.compareTo(s2);  // where s1 ans s2 are`
- `// strings to be compared`
-
- `This returns difference s1-s2. If :`
- `out < 0  // s1 comes before s2`
- `out = 0   // s1 and s2 are equal.`
- `out >0   // s1 comes after s2.`

  Note- In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).

- **String toLowerCase():** Converts all the characters in the String to lower case.
- `String word1 = “HeLLo”;`
- `String word3 = word1.toLowerCase(); // returns “hello"`
- **String toUpperCase():** Converts all the characters in the String to upper case.
- `String word1 = “HeLLo”;`
- `String word2 = word1.toUpperCase(); // returns “HELLO”`
- **String trim():** Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.
- `String word1 = “ Learn Share Learn “;`
- `String word2 = word1.trim(); // returns “Learn Share Learn”`
- **String replace (char oldChar, char newChar):** Returns new string by replacing all occurrences of *oldChar* with *newChar*.
- `String s1 = “feeksforfeeks“;`
- `String s2 = “feeksforfeeks”.replace(‘f’ ,’g’); // returns “geeksgorgeeks”`

Note:- s1 is still feeksforfeeks and s2 is geeksgorgeeks

Program to illustrate all string methods:

```
// Java code to illustrate different constructors and methods
// String class.

import java.io.*;
import java.util.*;
class Test
{
    public static void main (String[] args)
    {
        String s= "HelloWorld";
        // or String s= new String ("HelloWorld");

        // Returns the number of characters in the String.
        System.out.println("String length = " + s.length());

        // Returns the character at ith index.
        System.out.println("Character at 3rd position = "
                           + s.charAt(3));

        // Return the substring from the ith  index character
        // to end of string
        System.out.println("Substring " + s.substring(3));

        // Returns the substring from i to j-1 index.
        System.out.println("Substring  = " + s.substring(2,5));

        // Concatenates string2 to the end of string1.
        String s1 = "Geeks";
        String s2 = "forGeeks";
        System.out.println("Concatenated string  = " +
                            s1.concat(s2));

        // Returns the index within the string
        // of the first occurrence of the specified string.
        String s4 = "Learn Share Learn";
        System.out.println("Index of Share " +
                           s4.indexOf("Share"));

        // Returns the index within the string of the
        // first occurrence of the specified string,
        // starting at the specified index.
        System.out.println("Index of a  = " +
                           s4.indexOf('a',3));

        // Checking equality of Strings
        Boolean out = "Geeks".equals("geeks");
        System.out.println("Checking Equality  " + out);
        out = "Geeks".equals("Geeks");
        System.out.println("Checking Equality  " + out);
```

```
        out = "Geeks".equalsIgnoreCase("gEeks ");
        System.out.println("Checking Equality" + out);

        int out1 = s1.compareTo(s2);
        System.out.println("If s1 = s2" + out);

        // Converting cases
        String word1 = "GeeKyMe";
        System.out.println("Changing to lower Case " +
                           word1.toLowerCase());

        // Converting cases
        String word2 = "GeekyME";
        System.out.println("Changing to UPPER Case " +
                           word1.toUpperCase());

        // Trimming the word
        String word4 = " Learn Share Learn ";
        System.out.println("Trim the word " + word4.trim());

        // Replacing characters
        String str1 = "feeksforfeeks";
        System.out.println("Original String " + str1);
        String str2 = "feeksforfeeks".replace('f' ,'g') ;
        System.out.println("Replaced f with g -> " + str2);
    }
}
```

Output :

```
String length = 13
Character at 3rd position = k
Substring ksforGeeks
Substring = eks
Concatenated string = HelloWorld
Index of Share 6
Index of a = 8
Checking Equality false
Checking Equality true
Checking Equalityfalse
If s1 = s2false
Changing to lower Case geekyme
Changing to UPPER Case GEEKYME
Trim the word Learn Share Learn
Original String feeksforfeeks
Replaced f with g -> geeksgorgeeks
```

volatile keyword in Java

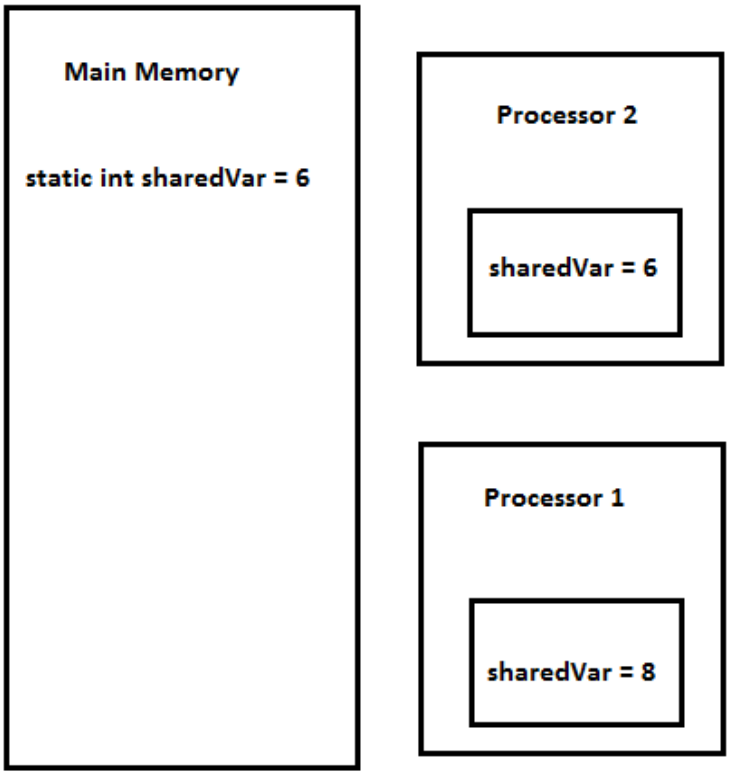
Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Consider below simple example.

```
class SharedObj
{
   // Changes made to sharedVar in one thread
   // may not immediately reflect in other thread
   static int sharedVar = 6;
}
```

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVar**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the write policy of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

Below diagram shows that if two threads are run on different processors, then value of **sharedVar** may be different in different threads.



Note that write of normal variables without any synchronization actions, might not be visible to any reading thread (this behavior is called sequential consistency). Although most modern hardware provide good cache coherence therefore most probably the changes in one cache are reflected in other but it's not a good practice to rely on hardware for to 'fix' a faulty application.

```
class SharedObj
{
   // volatile keyword here makes sure that
   // the changes made in one thread are
   // immediately reflect in other thread
   static volatile int sharedVar = 6;
}
```

Note that volatile should not be confused with static modifier. static variables are class members that are shared among all objects. There is only one copy of them in main memory.

volatile vs synchronized:
Before we move on let's take a look at two important features of locks and synchronization.

1. **Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.
2. **Visibility**: It means that changes made by one thread to shared data are visible to other threads.

Java's synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.

In some cases we may only desire the visibility and not atomicity. Use of synchronized in such situation is an overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, use of volatile is limited to very restricted set of cases as most of the times atomicity is desired. For example a simple increment statement such as x = x + 1; or x++ seems to be a single operation but is s really a compound read-modify-write sequence of operations that must execute atomically.

volatile in Java vs C/C++:
Volatile in java is different from "volatile" qualifier in C/C++. For Java, "volatile" tells the compiler that the value of a variable must never be cached as its value may change outside of the scope of the program itself. In C/C++, "volatile" is needed when developing embedded systems or device drivers, where you need to read or write a memory-mapped hardware device. The contents of a particular device register could change at any time, so you need the "volatile" keyword to ensure that such accesses aren't optimized away by the compiler.

Instance Variable Hiding in Java

In Java, if there a local variable in a method with same name as instance variable, then the local variable hides the instance variable. If we want to reflect the change made over to the instance variable, this can be achieved with the help of this reference.

```
class Test
{
    // Instance variable or member variable
    private int value = 10;

    void method()
    {
        // This local variable hides instance variable
        int value = 40;

        System.out.println("Value of Instance variable :"
                           + this.value);
        System.out.println("Value of Local variable :"
                           + value);
    }
}

class UseTest
{
    public static void main(String args[])
    {
        Test obj1 = new Test();
        obj1.method();
    }
}
```

Output:

```
Value of Instance variable :10
Value of Local variable :40
```

Does JVM create object of Main class (the class with main())?

Consider following program.

```
class Main {
    public static void main(String args[])
    {
        System.out.println("Hello");
    }
}
```

Output:

```
Hello
```

Does JVM create an object of class Main?
The answer is “No”. We have studied that the reason for main() static in Java is to make sure that the main() can be called without any instance. To justify the same, we can see that the following program compiles and runs fine.

```
// Not Main is abstract
abstract class Main {
    public static void main(String args[])
    {
        System.out.println("Hello");
    }
}
```

Output:

```
Hello
```

Since we can’t create object of abstract classes in Java, it is guaranteed that object of class with main() is not created by JVM.

Using underscore in Numeric Literals

A new feature was introduced by JDK 7 which allows to write numeric literals using the underscore character. Numeric literals are broken to enhance the readability.

This feature enables us to separate groups of digits in numeric literals, which improves readability of code. For instance, if our code contains numbers with many digits, we can use an underscore character to separate digits in groups of three, similar to how we would use a punctuation mark like a comma, or a space, as a separator.

The following example shows different ways we can use underscore in numeric literals:

```
// Java program to demonstrate that we can use underscore
// in numeric literals
class Test
{
     public static void main (String[] args)
               throws java.lang.Exception
     {
         int inum = 1_00_00_000;
         System.out.println("inum:" + inum);

         long lnum = 1_00_00_000;
         System.out.println("lnum:" + lnum);

         float fnum = 2.10_001F;
         System.out.println("fnum:" + fnum);

         double dnum = 2.10_12_001;
         System.out.println("dnum:" + dnum);
      }
}
```

Output:

```
inum: 10000000
lnum: 10000000
fnum: 2.10001
dnum: 2.1012001
```

Scope of Variables In Java

Introduction

Scope of a variable is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e., scope of a variable can determined at compiler time and independent of function call stack.

On a side note, unlike Java/C/C++, Perl supports both dynamic ans static scoping. Perl's keyword "my" defines a statically scoped local variable, while the keyword "local" defines dynamically scoped local variable i.e., scope depends on function call stack.

Java programs are organized in the form of classes. Every class is part of some package.

Java scope rules can be covered under following categories.

Member Variables (Class Level Scope)

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let's take a look at an example:

```
public class Test
{
    // All variables defined directly inside a class
    // are member variables
    int a;
    private String b
    void method1() {....}
    int method2() {....}
    char c;
}
```

- We can declare class variables anywhere in class, but outside methods.
- Access specified of member variables doesn't effect scope of them within a class.
- Member variables can be accessed outside a class with following rules

Modifier	Package	Subclass	World
public	Yes	Yes	Yes
protected	Yes	Yes	No
Default (no modifier)	Yes	No	No
private	No	No	No

Local Variables (Method Level Scope)

Variables declared inside a method have method level scope and can't be accessed outside the method.

```
public class Test
{
    void method1()
    {
       // Local variable (Method level scope)
       int x;
    }
}
```

Note : Local variables don't exist after method's execution is over.

Here's another example of method scope, except this time the variable got passed in as a parameter to the method:

```
class Test
{
    private int x;
    public void setX(int x)
    {
        this.x = x;
    }
}
```

The above code uses this keyword to differentiate between the local and class variables.

As an exercise, predict the output of following Java program.

```
public class Test
{
    static int x = 11;
    private int y = 33;
    public void method1(int x)
    {
        Test t = new Test();
        this.x = 22;
        y = 44;
        System.out.println("Test.x: " + Test.x);
        System.out.println("t.x: " + t.x);
        System.out.println("t.y: " + t.y);
        System.out.println("y: " + y);
    }

    public static void main(String args[])
    {
        Test t = new Test();
        t.method1(5);
    }
}
```

Output:

```
Test.x: 22
t.x: 22
```

```
t.y: 33
y: 44
```

Loop Variables (Block Scope)

A variable declared inside pair of brackets "{" and "}" in a method has scope withing the brackets only.

```
public class Test
{
    public static void main(String args[])
    {
        {
            // The variable x has scope within
            // brackets
            int x = 10;
            System.out.println(x);
        }

        // Uncommenting below line would produce
        // error since variable x is out of scope.

        // System.out.println(x);
    }
}
```

Output:

```
10
```

As another example, consider following program with for loop.

```
class Test
{
    public static void main(String args[])
    {
        for (int x=0; x<4; x++)
        {
            System.out.println(x);
        }

        // Will produce error
        System.out.println(x);
    }
}
```

Output:

```
11: error: cannot find symbol
        System.out.println(x);                                  ^
```

The right way of doing above is,

```
// Above program after correcting the error
```

```
class Test
{
    public static void main(String args[])
    {
        int x;
        for (x=0; x<4; x++)
        {
            System.out.println(x);
        }

       System.out.println(x);
    }
}
```

Output:

```
0
1
2
3
4
```

Let's look at tricky example of loop scope. Predict the output of following program. You may be surprised if you are regular C/C++ programmer.

```
class Test
{
    public static void main(String args[])
    {
        int x = 5;
        for (int x = 0; x < 5; x++)
        {
            System.out.println(x);
        }
    }
}
```

Output :

```
6: error: variable a is already defined in method go(int)
       for (int a = 0; a < 5; a++)
                ^
1 error
```

A similar program in C++ works. See this.

As an exercise, predict the output of following Java program.

```
class Test
{
    public static void main(String args[])
    {
        {
```

```
            int x = 5;
            {
                int x = 10;
                System.out.println(x);
            }
        }
    }
}
```

Some Important Points about Variable scope in Java:

- In general, a set of curly brackets { } defines a scope.
- In Java we can usually access a variable as long as it was defined within the same set of brackets as the code we are writing or within any curly brackets inside of the curly brackets where the variable was defined.
-
- Any variable defined in a class outside of any method can be used by all member methods.
- When a method has same local variable as a member, this keyword can be used to reference the current class variable.
- For a variable to be read after the termination of a loop, It must be declared before the body of the loop.

Scanner Class in Java

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double etc. and strings. It is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.

- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
- To read strings, we use nextLine().
- To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) funtion returns the first character in that string.

Let us look at the code snippet to read data of various data types.

```
// Java program to read data of various types using Scanner class.
import java.util.Scanner;
public class ScannerDemo1
{
    public static void main(String[] args)
    {
        // Declare the object and initialize with
        // predefined standard input object
        Scanner sc = new Scanner(System.in);

        // String input
        String name = sc.nextLine();

        // Character input
        char gender = sc.next().charAt(0);

        // Numerical data input
        // byte, short and float can be read
        // using similar-named functions.
        int age = sc.nextInt();
        long mobileNo = sc.nextLong();
        double cgpa = sc.nextDouble();

        // Print the values to check if input was correctly obtained.
        System.out.println("Name: "+name);
        System.out.println("Gender: "+gender);
        System.out.println("Age: "+age);
        System.out.println("Mobile Number: "+mobileNo);
        System.out.println("CGPA: "+cgpa);
    }
}
```

Input :

```
Geek
F
40
9876543210
9.9
```

Output :

```
Name: Geek
Gender: F
Age: 40
Mobile Number: 9876543210
CGPA: 9.9
```

Sometimes, we have to check if the next value we read is of a certain type or if the input has ended (EOF marker encountered).

Then, we check if the scanner's input is of the type we want with the help of hasNextXYZ() functions where XYZ is the type we are interested in. The function returns true if the scanner has a token of that type, otherwise false. For example, in the above code, we have used hasNextInt().To check for a string, we use hasNextLine(). Similarly, to check for a single character, we use hasNext().charAt(0).

Let us look at the code snippet to read some numbers from console and print their mean.

```
// Java program to read some values using Scanner
// class and print their mean.
import java.util.Scanner;

public class ScannerDemo2
{
    public static void main(String[] args)
    {
        // Declare an object and initialize with
        // predefined standard input object
        Scanner sc = new Scanner(System.in);

        // Initialize sum and count of input elements
        int sum = 0, count = 0;

        // Check if an int value is available
        while (sc.hasNextInt())
        {
            // Read an int value
            int num = sc.nextInt();
            sum += num;
            count++;
        }
        int mean = sum / count;
        System.out.println("Mean: " + mean);
```

```
    }
}
```

Input:

```
101
223
238
892
99
500
728
```

Output:

```
Mean: 397
```

Difference between Scanner and BufferReader Class in Java

java.util.Scanner class is a simple text scanner which can parse primitive types and strings. It internally uses regular expressions to read different types.

Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of sequence of characters

Following are differences between above two.

Issue with Scanner when nextLine() is used after nextXXX()

Try to guess the output of following code :

```
// Code using Scanner Class
import java.util.Scanner;
class Differ
{
     public static void main(String args[])
     {
          Scanner scn = new Scanner(System.in);
          System.out.println("Enter an integer");
          int a = scn.nextInt();
          System.out.println("Enter a String");
          String b = scn.nextLine();
          System.out.printf("You have entered:- "
                  + a + " " + "and name as " + b);
     }
}
```

Input:

```
50
Geek
```

Output:

```
Enter an integer
Enter a String
You have entered:- 50 and name as
```

Let us try the same using Buffer class and same Input

```
// Code using Buffer Class
import java.io.*;
class Differ
{
    public static void main(String args[])
                  throws IOException
```

```
    {
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.println("Enter an integer");
        int a = Integer.parseInt(br.readLine());
        System.out.println("Enter a String");
        String b = br.readLine();
        System.out.printf("You have entered:- " + a +
                          " and name as " + b);
    }
}
```

Input:

```
50
Geek
```

Output:

```
Enter an integer
Enter a String
you have entered:- 50 and name as Geek
```

In Scanner class if we call nextLine() method after any one of the seven nextXXX() method then the nextLine() doesn't not read values from console and cursor will not come into console it will skip that step. The nextXXX() methods are nextInt(), nextFloat(), nextByte(), nextShort(), nextDouble(), nextLong(), next().

In BufferReader class there is no such type of problem. This problem occurs only for Scanner class, due to nextXXX() methods ignore newline character and nextLine() only reads newline character. If we use one more call of nextLine() method between nextXXX() and nextLine(), then this problem will not occur because nextLine() will consume the newline character. See this for the corrected program. This problem is same as scanf() followed by gets() in C/C++.

Other differences:

- BufferedReader is synchronous while Scanner is not. BufferedReader should be used if we are working with multiple threads.
- BufferedReader has significantly larger buffer memory than Scanner.
- The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough.
- BufferedReader is a bit faster as compared to scanner because scanner does parsing of input data and BufferedReader simply reads sequence of characters.

Redirecting System.out.println() output to a file in Java

System.out.println() is used mostly to print messages to the console. However very few of us are actually aware of its working mechanism.

- *System* is a class defined in the *java.lang* package.
- out is an instance of *PrintStream* , which is a public and static member of the class *System*.
- As all instances of *PrintStream* class have a public method *println()*, hence we can invoke the same on out as well. We can assume *System.out* represents the standard Output Stream .

One interesting fact related to the above topic is, **we can use *System.out.println()* to print messages to other sources too (and not just console)** . However before doing so , we must reassign the standard output by using the following method of System class:

```
System.setOut(PrintStream p);
```

PrintStream can be used for character output to a text file. Below program creates the file A.txt and writes to the file using System.out.println(

```
// Java program to demonstrate redirection in System.out.println()
import java.io.*;

public class SystemFact
{
    public static void main(String arr[]) throws FileNotFoundException
    {
        // Creating a File object that represents the disk file.
        PrintStream o = new PrintStream(new File("A.txt"));

        // Store current System.out before assigning a new value
        PrintStream console = System.out;

        // Assign o to output stream
        System.setOut(o);
        System.out.println("This will be written to the text file");

        // Use stored value for output stream
        System.setOut(console);
        System.out.println("This will be written on the console!");
    }
```

(Addition and Concatenation in Java)

Try to predict the output of following code:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println(2+0+1+6+"HelloWorld");
        System.out.println("HelloWorld"+2+0+1+6);
        System.out.println(2+0+1+5+"HelloWorld"+2+0+1+6);
        System.out.println(2+0+1+5+"HelloWorld"+(2+0+1+6));
    }
}
```

Explanation:
The output is

9HelloWorld
HelloWorld2016
8HelloWorld2016
8HelloWorld9

This unpredictable output is due the fact that the compiler evaluates the given expression from left to right given that the operators have same precedence. Once it encounters the String, it considers the rest of the expression as of a String (again based on the precedence order of the expression).

- **System.out.println(2 + 0 + 1 + 6 + "HelloWorld");** *// It prints the addition of 2,0,1 and 6 which is equal to 9*
- **System.out.println("HelloWorld" + 2 + 0 + 1 + 6);** *//It prints the concatenation of 2,0,1 and 6 which is 2016 since it encounters the string initially. Basically, Strings take precedence because they have a higher casting priority than integers do.*
- **System.out.println(2 + 0 + 1 + 5 + "HelloWorld" + 2 + 0 + 1 + 6);** *//It prints the addition of 2,0,1 and 5 while the concatenation of 2,0,1 and 6 based on the above given examples.*
- **System.out.println(2 + 0 + 1 + 5 + "HelloWorld" + (2 + 0 + 1 + 6));** *//It prints the addition of both 2,0,1 and 5 and 2,0,1 and 6 based due the precedence of ( ) over +. Hence expression in ( ) is calculated first and then the further evaluation takes place.*

Fast I/O in Java in Competitive Programming

Using Java in competitive programming is not something many people would suggest just because of its slow input and output, and well indeed it is slow.

In this article, we have discussed some ways to get around the difficulty and change the verdict from TLE to (in most cases) AC.

For all the Programs below
Input:

```
7 3
1
51
966369
7
9
999996
11
```

Output:

```
4
```

1. **Scanner Class** – (easy, less typing, but not recommended very slow, refer this for reasons of slowness): In most of the cases we get TLE while using scanner class. It uses built-in nextInt(), nextLong(), nextDouble methods to read the desired object after initiating scanner object with input stream.(eg System.in). The following program many a times gets time limit exceeded verdict and therefore not of much use.

```
// Working program using Scanner
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Scanner;
public class Main
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt();
        int k = s.nextInt();
        int count = 0;
        while (n-- > 0)
        {
            int x = s.nextInt();
            if (x%k == 0)
               count++;
        }
```

```
            System.out.println(count);
        }
    }
```

-

- **BufferedReader** – (fast, but not recommended as it requires lot of typing): The Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. With this method we will have to parse the value every time for desired type. Reading multiple words from single line adds to its complexity because of the use of Stringtokenizer and hence this is not recommended. This gets accepted with a running time of approx 0.89 s.but still as you can see it requires a lot of typing all together and therefore method 3 is recommended.

```
// Working program using BufferedReader
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class Main
{
    public static void main(String[] args) throws IOException
    {

        BufferedReader br = new BufferedReader(
                              new InputStreamReader(System.in));

        StringTokenizer st = new StringTokenizer(br.readLine());
        int n = Integer.parseInt(st.nextToken());
        int k = Integer.parseInt(st.nextToken());
        int count = 0;
        while (n-- > 0)
        {
            int x = Integer.parseInt(br.readLine());
            if (x%k == 0)
               count++;
        }
        System.out.println(count);
    }
}
```

- - **Userdefined FastReader Class-** (which uses bufferedReader and StringTokenizer): This method uses the time advantage of BufferedReader and StringTokenizer and the advantage of user defined methods for less typing and therefore a faster input altogether. This gets accepted with a time of 1.23 s and this method is *very much recommended* as it is easy to remember and is fast enough to meet the needs of most of the question in competitive coding.

```
// Working program with FastReader
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;
import java.util.StringTokenizer;
```

```
public class Main
{
    static class FastReader
    {
        BufferedReader br;
        StringTokenizer st;

        public FastReader()
        {
            br = new BufferedReader(new
                     InputStreamReader(System.in));
        }

        String next()
        {
            while (st == null || !st.hasMoreElements())
            {
                try
                {
                    st = new StringTokenizer(br.readLine());
                }
                catch (IOException  e)
                {
                    e.printStackTrace();
                }
            }
            return st.nextToken();
        }

        int nextInt()
        {
            return Integer.parseInt(next());
        }

        long nextLong()
        {
            return Long.parseLong(next());
        }

        double nextDouble()
        {
            return Double.parseDouble(next());
        }

        String nextLine()
        {
            String str = "";
            try
            {
                str = br.readLine();
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
```

```
            return str;
        }
    }

    public static void main(String[] args)
    {
        FastReader s=new FastReader();
        int n = s.nextInt();
        int k = s.nextInt();
        int count = 0;
        while (n-- > 0)
        {
            int x = s.nextInt();
            if (x%k == 0)
               count++;
        }
        System.out.println(count);
    }
}
```

- **Using Reader Class**: There is yet another fast way through the problem, I would say the fastest way but is not recommended since it requires very cumbersome methods in its implementation. It uses inputDataStream to read through the stream of data and uses read() method and nextInt() methods for taking inputs. This is by far the fastest ways of taking input but is difficult to remember and is cumbersome in its approach. Below is the sample program using this method.

This gets accepted with a surprising time of just 0.28 s. Although this is ultra fast, it is clearly not an easy method to remember.

```
// Working program using Reader Class
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;
import java.util.StringTokenizer;

public class Main
{
    static class Reader
    {
        final private int BUFFER_SIZE = 1 << 16;
        private DataInputStream din;
        private byte[] buffer;
        private int bufferPointer, bytesRead;

        public Reader()
        {
            din = new DataInputStream(System.in);
            buffer = new byte[BUFFER_SIZE];
            bufferPointer = bytesRead = 0;
        }

        public Reader(String file_name) throws IOException
```

```
{
    din = new DataInputStream(new FileInputStream(file_name));
    buffer = new byte[BUFFER_SIZE];
    bufferPointer = bytesRead = 0;
}

public String readLine() throws IOException
{
    byte[] buf = new byte[64]; // line length
    int cnt = 0, c;
    while ((c = read()) != -1)
    {
        if (c == '\n')
            break;
        buf[cnt++] = (byte) c;
    }
    return new String(buf, 0, cnt);
}

public int nextInt() throws IOException
{
    int ret = 0;
    byte c = read();
    while (c <= ' ')
        c = read();
    boolean neg = (c == '-');
    if (neg)
        c = read();
    do
    {
        ret = ret * 10 + c - '0';
    }  while ((c = read()) >= '0' && c <= '9');

    if (neg)
        return -ret;
    return ret;
}

public long nextLong() throws IOException
{
    long ret = 0;
    byte c = read();
    while (c <= ' ')
        c = read();
    boolean neg = (c == '-');
    if (neg)
        c = read();
    do {
        ret = ret * 10 + c - '0';
    }
    while ((c = read()) >= '0' && c <= '9');
    if (neg)
        return -ret;
    return ret;
}
```

```
    public double nextDouble() throws IOException
    {
        double ret = 0, div = 1;
        byte c = read();
        while (c <= ' ')
            c = read();
        boolean neg = (c == '-');
        if (neg)
            c = read();

        do {
            ret = ret * 10 + c - '0';
        }
        while ((c = read()) >= '0' && c <= '9');

        if (c == '.')
        {
            while ((c = read()) >= '0' && c <= '9')
            {
                ret += (c - '0') / (div *= 10);
            }
        }

        if (neg)
            return -ret;
        return ret;
    }

    private void fillBuffer() throws IOException
    {
        bytesRead = din.read(buffer, bufferPointer = 0, BUFFER_SIZE);
        if (bytesRead == -1)
            buffer[0] = -1;
    }

    private byte read() throws IOException
    {
        if (bufferPointer == bytesRead)
            fillBuffer();
        return buffer[bufferPointer++];
    }

    public void close() throws IOException
    {
        if (din == null)
            return;
        din.close();
    }
}

public static void main(String[] args) throws IOException
{
    Reader s=new Reader();
    int n = s.nextInt();
    int k = s.nextInt();
```

```
        int count=0;
        while (n-- > 0)
        {
            int x = s.nextInt();
            if (x%k == 0)
               count++;
        }
        System.out.println(count);
    }
}
```

INTEST - Enormous Input Test

no tags

The purpose of this problem is to verify whether the method you are using to read input data is sufficiently fast to handle problems branded with the **enormous Input/Output** warning. You are expected to be able to process at least 2.5MB of input data per second at runtime.

Input

The input begins with two positive integers n k (n, k<=10^7). The next n lines of input contain one positive integer t_i, not greater than 10^9, each.

Output

Write a single integer to output, denoting how many integers t_i are divisible by k.

Example

Input:
```
7 3
1
51
966369
7
9
999996
11
```

Output:
```
4
```

Bitwise right shift operators in Java

In C/C++ there is only one right shift operator '>>' which should be used only for positive integers or unsigned integers. Use of right shift operator for negative numbers is not recommended in C/C++, and when used for negative numbers, output is compiler dependent (See this). Unlike C++, Java supports following two right shift operators.

1) >> (Signed right shift) In Java, the operator '>>' is signed right shift operator. All integers are signed in Java, and it is fine to use >> for negative numbers. The operator '>>' uses the sign bit (left most bit) to fill the trailing positions after shift. If the number is negative, then 1 is used as a filler and if the number is positive, then 0 is used as a filler. For example, if binary representation of number is **1**0….100, then right shifting it by 2 using >> will make it **11**…….1. See following Java programs as example '>>'

```
class Test {
    public static void main(String args[])  {
       int x = -4;
       System.out.println(x>>1);
       int y = 4;
       System.out.println(y>>1);
    }
}
```

Output:

```
-2
2
```

2) >>> (Unsigned right shift) In Java, the operator '>>>' is unsigned right shift operator. It always fills 0 irrespective of the sign of the number.

```
class Test {
    public static void main(String args[])  {

       // x is stored using 32 bit 2's complement form.
       // Binary representation of -1 is all 1s (111..1)
       int x = -1;

       System.out.println(x>>>29);  // The value of 'x>>>29' is 00...0111
       System.out.println(x>>>30);  // The value of 'x>>>30' is 00...0011
       System.out.println(x>>>31);  // The value of 'x>>>31' is 00...0001
   }
}
```

Output:

```
7
3
1
```

Comparison of Autoboxed Integer objects in Java

When we assign an integer value to an Integer object, the value is autoboxed into an Integer object. For example the statement "Integer x = 10″ creates an object 'x' with value 10.

Following are some interesting output questions based on comparison of Autoboxed Integer objects.

Predict the output of following Java Program

```
// file name: Main.java
public class Main {
    public static void main(String args[]) {
         Integer x = 400, y = 400;
         if (x == y)
            System.out.println("Same");
         else
            System.out.println("Not Same");
    }
}
```

Output:

```
Not Same
```

Since x and y refer to different objects, we get the output as "Not Same"

The output of following program is a surprise from Java.

```
// file name: Main.java
public class Main {
    public static void main(String args[]) {
         Integer x = 40, y = 40;
         if (x == y)
            System.out.println("Same");
         else
            System.out.println("Not Same");
    }
}
```

Output:

```
Same
```

In Java, values from -128 to 127 are cached, so the same objects are returned. The implementation of valueOf() uses cached objects if the value is between -128 to 127.

If we explicitly create Integer objects using new operator, we get the output as “Not Same”. See the following Java program. In the following program, valueOf() is not used.

```
// file name: Main.java
public class Main {
    public static void main(String args[]) {
          Integer x = new Integer(40), y = new Integer(40);
         if (x == y)
            System.out.println("Same");
         else
            System.out.println("Not Same");
    }
}
```

Output:

```
Not Same
```

(Addition and Concatenation in Java)

Try to predict the output of following code:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println(2+0+1+6+"HelloWorld");
        System.out.println("HelloWorld"+2+0+1+6);
        System.out.println(2+0+1+5+"HelloWorld"+2+0+1+6);
        System.out.println(2+0+1+5+"HelloWorld"+(2+0+1+6));
    }
}
```

Explanation:
The output is

9HelloWorld
HelloWorld2016
8HelloWorld2016
8HelloWorld9

This unpredictable output is due the fact that the compiler evaluates the given expression from left to right given that the operators have same precedence. Once it encounters the String, it considers the rest of the expression as of a String (again based on the precedence order of the expression).

- **System.out.println(2 + 0 + 1 + 6 + "HelloWorld");** *// It prints the addition of 2,0,1 and 6 which is equal to 9*
- **System.out.println("HelloWorld" + 2 + 0 + 1 + 6);** *//It prints the concatenation of 2,0,1 and 6 which is 2016 since it encounters the string initially. Basically, Strings take precedence because they have a higher casting priority than integers do.*
- **System.out.println(2 + 0 + 1 + 5 + "HelloWorld" + 2 + 0 + 1 + 6);** *//It prints the addition of 2,0,1 and 5 while the concatenation of 2,0,1 and 6 based on the above given examples.*
- **System.out.println(2 + 0 + 1 + 5 + "HelloWorld" + (2 + 0 + 1 + 6));** *//It prints the addition of both 2,0,1 and 5 and 2,0,1 and 6 based due the precedence of ( ) over +. Hence expression in ( ) is calculated first and then the further evaluation takes place.*

Java Numeric Promotion in Conditional Expression

The conditional operator `? :` uses the boolean value of one expression to decide which of two other expressions should be evaluated.

So, we expect the expression,

```
Object o1 = true ? new Integer(4) : new Float(2.0));
```

to be same as,

```
Object o2;
if (true)
    o2 = new Integer(4);
else
    o2 = new Float(2.0);
```

But the result of running the code gives an unexpected result.

```
// A Java program to demonstrate that we should be careful
// when replacing conditional operator with if else or vice
// versa
import java.io.*;
class GFG
{
    public static void main (String[] args)
    {
        // Expression 1 (using ?: )
        // Automatic promotion in conditional expression
        Object o1 = true ? new Integer(4) : new Float(2.0);
        System.out.println(o1);

        // Expression 2 (Using if-else)
        // No promotion in if else statement
        Object o2;
        if (true)
            o2 = new Integer(4);
        else
            o2 = new Float(2.0);
        System.out.println(o2);
    }
}
```

Output:

```
4.0
4
```

According to Java Language Specification Section 15.25, the conditional operator will implement numeric type promotion if there are two different types as 2nd and 3rd operand. The rules of conversion are defined at Binary Numeric Promotion. Therefore, according to the rules given, If either operand is of type double, the other is converted to double and hence 4 becomes 4.0.

Whereas, the if/else construct does not perform numeric promotion and hence behaves as expected.

Arrays in Java

Unlike C++, arrays are first class objects in Java. For example, in the following program, size of array is accessed using *length* which is a member of *arr[]* object.

```
// file name: Main.java
public class Main {
    public static void main(String args[]) {
       int arr[] = {10, 20, 30, 40, 50};
       for(int i=0; i < arr.length; i++)
       {
             System.out.print(" " + arr[i]);
       }
    }
}
```

Output:
10 20 30 40 50

How to compare two arrays in Java?

Predict the output of following Java program.

```
class Test
{
    public static void main (String[] args)
    {
        int arr1[] = {1, 2, 3};
        int arr2[] = {1, 2, 3};
        if (arr1 == arr2) // Same as arr1.equals(arr2)
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Output:

```
Not Same
```

In Java, arrays are first class objects. In the above program, arr1 and arr2 are two references to two different objects. So when we compare arr1 and arr2, two reference variables are compared, therefore we get the output as “Not Same” (See this for more examples).

How to compare array contents?
A simple way is to run a loop and compare elements one by one. Java provides a direct method

Arrays.equals() to compare two arrays. Actually, there is a list of equals() methods in Arrays class for different primitive types (int, char, ..etc) and one for Object type (which is base of all classes in Java).

```
// we need to import java.util.Arrays to use Arrays.equals().
import java.util.Arrays;
class Test
{
    public static void main (String[] args)
    {
        int arr1[] = {1, 2, 3};
        int arr2[] = {1, 2, 3};
        if (Arrays.equals(arr1, arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Output:

```
Same
```

How to Deep compare array contents?
As seen above, the Arrays.equals() works fine and compares arrays contents. Now the questions, what if the arrays contain arrays inside them or some other references which refer to different object but have same values. For example, see the following program.

```
import java.util.Arrays;
class Test
{
    public static void main (String[] args)
    {
        // inarr1 and inarr2 have same values
        int inarr1[] = {1, 2, 3};
        int inarr2[] = {1, 2, 3};
        Object[] arr1 = {inarr1};  // arr1 contains only one element
        Object[] arr2 = {inarr2};  // arr2 also contains only one element
        if (Arrays.equals(arr1, arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Output:

```
Not Same
```

So *Arrays.equals()* is not able to do deep comparison. Java provides another method for this Arrays.deepEquals() which does deep comparison.

```
import java.util.Arrays;
class Test
{
    public static void main (String[] args)
    {
        int inarr1[] = {1, 2, 3};
        int inarr2[] = {1, 2, 3};
        Object[] arr1 = {inarr1};  // arr1 contains only one element
        Object[] arr2 = {inarr2};  // arr2 also contains only one element
        if (Arrays.deepEquals(arr1, arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Output:

```
Same
```

How does Arrays.deepEquals() work?
It compares two objects using any custom equals() methods they may have (if they have an equals() method implemented other than Object.equals()). If not, this method will then proceed to compare the objects field by field, recursively. As each field is encountered, it will attempt to use the derived equals() if it exists, otherwise it will continue to recurse further.
This method works on a cyclic Object graph like this: A->B->C->A. It has cycle detection so ANY two objects can be compared, and it will never enter into an endless loop (Source: https://code.google.com/p/deep-equals/).

Exercise: Predict the output of following program

```
import java.util.Arrays;
class Test
{
   public static void main (String[] args)
   {
      int inarr1[] = {1, 2, 3};
      int inarr2[] = {1, 2, 3};
      Object[] arr1 = {inarr1};  // arr1 contains only one element
      Object[] arr2 = {inarr2};  // arr2 also contains only one element
      Object[] outarr1 = {arr1}; // outarr1 contains only one element
      Object[] outarr2 = {arr2}; // outarr2 also contains only one
element
      if (Arrays.deepEquals(outarr1, outarr2))
          System.out.println("Same");
      else
          System.out.println("Not same");
    }
}
```

Final arrays in Java

Predict the output of following Java program.

```
class Test
{
    public static void main(String args[])
    {
       final int arr[] = {1, 2, 3, 4, 5};  // Note: arr is final
       for (int i = 0; i < arr.length; i++)
       {
           arr[i] = arr[i]*10;
           System.out.println(arr[i]);
       }
    }
}
```

Output:

```
10
20
30
40
50
```

The array *arr* is declared as final, but the elements of array are changed without any problem. Arrays are objects and object variables are always references in Java. So, when we declare an object variable as final, it means that the variable cannot be changed to refer to anything else. For example, the following program 1 compiles without any error and program fails in compilation.

```
// Program 1
class Test
{
    int p = 20;
    public static void main(String args[])
    {
       final Test t = new Test();
       t.p = 30;
       System.out.println(t.p);
    }
}
```

Output: 30

```
// Program 2
class Test
{
    int p = 20;
    public static void main(String args[])
    {
       final Test t1 = new Test();
       Test t2 = new Test();
```

```
        t1 = t2;
        System.out.println(t1.p);
    }
}
```

Output: Compiler Error: cannot assign a value to final variable t1

So a final array means that the array variable which is actually a reference to an object, cannot be changed to refer to anything else, but the members of array can be modified.

As an exercise, predict the output of following program

```
class Test
{
    public static void main(String args[])
    {
        final int arr1[] = {1, 2, 3, 4, 5};
        int arr2[] = {10, 20, 30, 40, 50};
        arr2 = arr1;
        arr1 = arr2;
        for (int i = 0; i < arr2.length; i++)
            System.out.println(arr2[i]);
    }
}
```

Jagged Array in Java

Jagged array is array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D arrays but with variable number of columns in each row. These type of arrays are also known as ragged arrays.

Following are Java programs to demonstrate the above concept.

```
// Program to demonstrate 2-D jagged array in Java
class Main
{
    public static void main(String[] args)
    {
        // Declaring 2-D array with 2 rows
        int arr[][] = new int[2][];

        // Making the above array Jagged

        // First row has 3 columns
        arr[0] = new int[3];

        // Second row has 2 columns
        arr[1] = new int[2];

        // Initializing array
        int count = 0;
        for (int i=0; i<arr.length; i++)
            for(int j=0; j<arr[i].length; j++)
                arr[i][j] = count++;

        // Displaying the values of 2D Jagged array
        System.out.println("Contents of 2D Jagged Array");
        for (int i=0; i<arr.length; i++)
        {
            for (int j=0; j<arr[i].length; j++)
                System.out.print(arr[i][j] + " ");
            System.out.println();
        }
    }
}
```

Output:

```
Contents of 2D Jagged Array
0 1 2
3 4
```

Following is another example where i'th row has i columns, i.e., first row has 1 element, second row has two elements and so on.

```
// Another Java program to demonstrate 2-D jagged
```

```
// array such that first row has 1 element, second
// row has two elements and so on.
class Main
{
    public static void main(String[] args)
    {
        int r = 5;

        // Declaring 2-D array with 5 rows
        int arr[][] = new int[r][];

        // Creating a 2D array such that first row
        // has 1 element, second row has two
        // elements and so on.
        for (int i=0; i<arr.length; i++)
            arr[i] = new int[i+1];

        // Initializing array
        int count = 0;
        for (int i=0; i<arr.length; i++)
            for(int j=0; j<arr[i].length; j++)
                arr[i][j] = count++;

        // Displaying the values of 2D Jagged array
        System.out.println("Contents of 2D Jagged Array");
        for (int i=0; i<arr.length; i++)
        {
            for (int j=0; j<arr[i].length; j++)
                System.out.print(arr[i][j] + " ");
            System.out.println();
        }
    }
}
```

Output:

```
Contents of 2D Jagged Array
0
1 2
3 4 5
6 7 8 9
10 11 12 13 14
```

Understanding Array IndexOutofbounds Exception in Java

Java supports creation and manipulation of arrays, as a data structure. The index of an array is an integer value that has value in interval [0, n-1], where n is the size of the array. If a request for a negative or an index greater than or equal to size of array is made, then the JAVA throws a ArrayIndexOutOfBounds Exception. This is unlike C/C++ where no index of bound check is done.

The ArrayIndexOutOfBoundsException is a Runtime Exception thrown only at runtime. The Java Compiler does not check for this error during the compilation of a program.

```
// A Common cause index out of bound

public class NewClass2

{

    public static void main(String[] args)

    {

        int ar[] = {1, 2, 3, 4, 5};

        for (int i=0; i<=ar.length; i++)

          System.out.println(ar[i]);

    }

}
```

Runtime error throws an Exception:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at NewClass2.main(NewClass2.java:5)
```

Output:

```
1
2
3
4
5
```

Here if you carefully see, the array is of size 5. Therefore while accessing its element using for loop, the maximum value of index can be 4 but in our program it is going till 5 and thus the exception.

Let’s see another example using arraylist:

```
// One more example with index out of bound

import java.util.ArrayList;

public class NewClass2

{

    public static void main(String[] args)

    {

        ArrayList<String> lis = new ArrayList<>();

        lis.add("My");

        lis.add("Name");

        System.out.println(lis.get(2));

    }

}
```

Runtime error here is a bit more informative than the previous time-

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 2,
Size: 2
        at java.util.ArrayList.rangeCheck(ArrayList.java:653)
        at java.util.ArrayList.get(ArrayList.java:429)
        at NewClass2.main(NewClass2.java:7)
```

Lets understand it in a bit of detail-

- Index here defines the index we are trying to access.
- The size gives us information of the size of the list.
- Since size is 2, the last index we can access is (2-1)=1, and thus the exception

The correct way to access array is :

```
for (int i=0; i<ar.length; i++)
{
}
```

Handling the Exception:

- **Use for-each loop:** This automatically handles indices while accessing the elements of an array. Example-
- ```
  for(int m : ar){
  }
  ```

- **Use Try-Catch:** Consider enclosing your code inside a try-catch statement and manipulate the exception accordingly. As mentioned, Java won't let you access an invalid index and will definitely throw an ArrayIndexOutOfBoundsException. However, we should be careful inside the block of the catch statement, because if we don't handle the exception appropriately, we may conceal it and thus, create a bug in your application.
  ```

Comparison of static keyword in C++ and Java

Static keyword is used for almost same purpose in both C++ and Java. There are some differences though. This post covers similarities and differences of static keyword in C++ and Java.

Static Data Members: Like C++, static data members in Java are class members and shared among all objects. For example, in the following Java program, static variable *count* is used to count the number of objects created.

```
class Test {
    static int count = 0;

    Test() {
       count++;
    }
    public static void main(String arr[]) {
       Test t1 = new Test();
       Test t2 = new Test();
       System.out.println("Total " + count + " objects created");
    }
}
```

Output:

```
Total 2 objects created
```

Static Member Methods: Like C++, methods declared as static are class members and have following restrictions:

1) They can only call other static methods. For example, the following program fails in compilation. fun() is non-static and it is called in static main()

```
class Main {
    public static void main(String args[]) {
        System.out.println(fun());
    }
    int fun() {
        return 20;
    }
}
```

2) They must only access static data.

3) They cannot access *this* or *super* . For example, the following program fails in compilation.

```
class Base {
```

```
    static int x = 0;
}

class Derived extends Base
{
   public static void fun() {
       System.out.println(super.x); // Compiler Error: non-static variable
                                    // cannot be referenced from a static
context
   }
}
```

Like C++, static data members and static methods can be accessed without creating an object. They can be accessed using class name. For example, in the following program, static data member count and static method fun() are accessed without any object.

```
class Test {
    static int count = 0;
    public static void fun() {
       System.out.println("Static fun() called");
    }
}

class Main
{
    public static void main(String arr[]) {
       System.out.println("Test.count = " + Test.count);
       Test.fun();
    }
}
```

Static Block: Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initialization of a class. This code inside static block is executed only once. See Static blocks in Java for details.

Static Local Variables: Unlike C++, Java doesn't support static local variables. For example, the following Java program fails in compilation.

```
class Test {
   public static void main(String args[]) {
     System.out.println(fun());
   }
   static int fun()  {
     static int x= 10; //Compiler Error: Static local variables are not
allowed
     return x--;
   }
}
```

Static blocks in Java

Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class. This code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class). For example, check output of following Java program.

```
// filename: Main.java
class Test {
    static int i;
    int j;

    // start of static block
    static {
        i = 10;
        System.out.println("static block called ");
    }
    // end of static block
}

class Main {
    public static void main(String args[]) {

        // Although we don't have an object of Test, static block is
        // called because i is being accessed in following statement.
        System.out.println(Test.i);
    }
}
```

Output:
static block called
10

Also, static blocks are executed before constructors. For example, check output of following Java program.

```
// filename: Main.java
class Test {
    static int i;
    int j;
    static {
        i = 10;
        System.out.println("static block called ");
    }
    Test(){
        System.out.println("Constructor called");
    }
}
```

```
class Main {
    public static void main(String args[]) {

       // Although we have two objects, static block is executed only once.
       Test t1 = new Test();
       Test t2 = new Test();
    }
}
```

Output:
static block called
Constructor called
Constructor called

What if we want to execute some code for every object?
We use Initializer Block in Java

Are static local variables allowed in Java?

Unlike C/C++, static local variables are not allowed in Java. For example, following Java program fails in compilation with error *"Static local variables are not allowed"*

```
class Test {
   public static void main(String args[]) {
     System.out.println(fun());
   }

   static int fun()
   {
     static int x= 10;  //Error: Static local variables are not allowed
     return x--;
   }
}
```

Static class in Java

Can a class be static in Java ?
The answer is YES, we can have static class in java. In java, we have static instance variables as well as static methods and also static block. Classes can also be made static in Java.

Java allows us to define a class within another class. Such a class is called a nested class. The class which enclosed nested class is known as Outer class. In java, we can't make Top level class static. ***Only nested classes can be static***.

What are the differences between static and non-static nested classes?
Following are major differences between static nested class and non-static nested class. Non-static nested class is also called Inner Class.

1) Nested static class doesn't need reference of Outer class, but Non-static nested class or Inner class requires Outer class reference.

2) Inner class(or non-static nested class) can access both static and non-static members of Outer class. A static class cannot access non-static members of the Outer class. It can access only static members of Outer class.

3) An instance of Inner class cannot be created without an instance of outer class and an Inner class can reference data and methods defined in Outer class in which it nests, so we don't need to pass reference of an object to the constructor of the Inner class. For this reason Inner classes can make program simple and concise.

```
/* Java program to demonstrate how to implement static and non-static
   classes in a java program. */
class OuterClass{
   private static String msg = "HelloWorld";

   // Static nested class
   public static class NestedStaticClass{

       // Only static members of Outer class is directly accessible in nested
       // static class
       public void printMessage() {

         // Try making 'message' a non-static variable, there will be
         // compiler error
         System.out.println("Message from nested static class: " + msg);
       }
    }

    // non-static nested class - also called Inner class
    public class InnerClass{

       // Both static and non-static members of Outer class are accessible in
```

```
        // this Inner class
        public void display(){
            System.out.println("Message from non-static nested class: "+ msg);
        }
    }
}
class Main
{
    // How to create instance of static and non static nested class?
    public static void main(String args[]){

        // create instance of nested Static class
        OuterClass.NestedStaticClass printer = new
OuterClass.NestedStaticClass();

        // call non static method of nested static class
        printer.printMessage();

        // In order to create instance of Inner class we need an Outer class
        // instance. Let us create Outer class instance for creating
        // non-static nested class
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner  = outer.new InnerClass();

        // calling non-static method of Inner class
        inner.display();

        // we can also combine above steps in one step to create instance of
        // Inner class
        OuterClass.InnerClass innerObject = new OuterClass().new InnerClass();

        // similarly we can now call Inner class method
        innerObject.display();
    }
}
```

Output:

```
Message from nested static class: HelloWorld
Message from non-static nested class: HelloWorld
Message from non-static nested class: HelloWorld
```

Can we Overload or Override static methods in java ?

Let us first define Overloading and Overriding.

Overriding : Overriding is a feature of OOP languages like Java that is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in superclass (or base class).
The implementation to be executed is decided at run-time and decision is made according to the object used for call. Note that signatures of both methods must be same.

Overloading: Overloading is also a feature of OOP languages like Java that is related to compile time (or static) polymorphism. This feature allows different methods to have same name, but different signatures, especially number of input parameters and type of input paramaters. Note that in both C++ and Java, methods cannot be overloaded according to return type.

Can we overload static methods?
The answer is 'Yes'. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program.

```
// filename Test.java
public class Test {
    public static void foo() {
        System.out.println("Test.foo() called ");
    }
    public static void foo(int a) {
        System.out.println("Test.foo(int) called ");
    }
    public static void main(String args[])
    {
        Test.foo();
        Test.foo(10);
    }
}
```

Output:

```
Test.foo() called
Test.foo(int) called
```

Can we overload methods that differ only by static keyword?
We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. This behaviour is same in C++ (See point 2 of this).

```
// filename Test.java
public class Test {
    public static void foo() {
```

```
        System.out.println("Test.foo() called ");
    }
    public void foo() { // Compiler Error: cannot redefine foo()
        System.out.println("Test.foo(int) called ");
    }
    public static void main(String args[]) {
        Test.foo();
    }
}
```

Output: Compiler Error, cannot redefine foo()

Can we Override static methods in java?

We can declare static methods with same signature in subclass, but it is not considered overriding as there won't be any run-time polymorphism. Hence the answer is 'No'.

If a derived class defines a static method with same signature as a static method in base class, the method in the derived class hides the method in the base class.

```
/* Java program to show that if static method is redefined by
   a derived class, then it is not overriding. */

// Superclass
class Base {

    // Static method in base class which will be hidden in subclass
    public static void display() {
        System.out.println("Static or class method from Base");
    }

     // Non-static method which will be overridden in derived class
     public void print()  {
         System.out.println("Non-static or Instance method from Base");
    }
}

// Subclass
class Derived extends Base {

    // This method hides display() in Base
    public static void display() {
         System.out.println("Static or class method from Derived");
    }

    // This method overrides print() in Base
    public void print() {
         System.out.println("Non-static or Instance method from Derived");
   }
}

// Driver class
public class Test {
    public static void main(String args[ ])  {
       Base obj1 = new Derived();
```

```
        // As per overriding rules this should call to class Derive's static
        // overridden method. Since static method can not be overridden, it
        // calls Base's display()
        obj1.display();

        // Here overriding works and Derive's print() is called
        obj1.print();
    }
}
```

Output:

```
Static or class method from Base
Non-static or Instance method from Derived
```

Following are some important points for method overriding and static methods in Java.
1) For class (or static) methods, the method according to the type of reference is called, not according to the abject being referred, which means method call is decided at compile time.

2) For instance (or non-static) methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time.

3) An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.

```
/* Java program to show that if static methods are redefined by
   a derived class, then it is not overriding but hidding. */

// Superclass
class Base {

    // Static method in base class which will be hidden in subclass
    public static void display() {
        System.out.println("Static or class method from Base");
    }

     // Non-static method which will be overridden in derived class
     public void print()  {
         System.out.println("Non-static or Instance method from Base");
    }
}

// Subclass
class Derived extends Base {

    // Static is removed here (Causes Compiler Error)
    public void display() {
        System.out.println("Non-static method from Derived");
    }
```

```
    // Static is added here (Causes Compiler Error)
    public static void print() {
        System.out.println("Static method from Derived");
   }
}
```

4) In a subclass (or Derived Class), we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

“final” keyword in java

final keyword is used in Java in different contexts. The idea is make an entity non-modifiable. Following are different contexts where final is used.

Final variables A final variable can only be assigned once. If the variable is a reference, this means that the variable cannot be re-bound to reference another object.

```
// The following doesn't compile because final variable is assigned
// value twice.
class Main {
   public static void main(String args[]){
      final int i = 20;
      i = 30;
   }
}
```

Output

```
Compiler Error: cannot assign a value to final variable i
```

Note the difference between C++ const variables and Java final variables. const variables in C++ must be assigned a value when declared. For final variables n Java, it is not necessary. A final variable can be assigned value later, but only once. For example see the following Java program.

```
// The following program compiles and runs fine
class Main {
   public static void main(String args[]){
      final int i;
      i = 20;
      System.out.println(i);
   }
}
```

Output:

```
20
```

Final classes A final class cannot be extended (inherited)

```
//Error in following program as we are trying to extend a final class
final class Base { }
class Derived extends Base { }

public class Main {
   public static void main(String args[]) {
   }
}
```

Output:

```
 Compiler Error: cannot inherit from final Base
```

Final methods A final method cannot be overridden by subclasses.
//Error in following program as we are trying to override a final method.

```
class Base {
  public final void show() {
       System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public void show() {  // Error
       System.out.println("Derived::show() called");
    }
}

public class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
        b.show();
    }
}
```

Output:

```
Compiler Error: show() in Derived cannot override show()
in Base overridden method is final
```

Super Keyword in Java

The **super** keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

1. Use of super with variables: This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

```
/* Base class vehicle */
class Vehicle
{
    int maxSpeed = 120;
}

/* sub class Car extending vehicle */
class Car extends Vehicle
{
    int maxSpeed = 180;

    void display()
    {
        /* print maxSpeed of base class (vehicle) */
        System.out.println("Maximum Speed: " + super.maxSpeed);
    }
}

/* Driver program to test */
class Test
{
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```

Output:

```
Maximum Speed: 120
```

In the above example, both base class and subclass have a member maxSpeed. We could access maxSpeed of base class in sublcass using super keyword.

2. Use of super with methods: This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

```
/* Base class Person */
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
}

/* Subclass Student */
class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }

    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}

/* Driver program to test */
class Test
{
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

Output:

```
This is student class
This is person class
```

In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

3. **Use of super with constructors:** super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non

parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

```
/* superclass Person */
class Person
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

/* subclass Student extending the Person class */
class Student extends Person
{
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}

/* Driver program to test*/
class Test
{
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

Output:

```
Person class Constructor
Student class Constructor
```

In the above example we have called the superclass constructor using keyword ‘super’ via subclass constructor.

Other Important points:

1. Call to super() must be first statement in Derived(Student) Class constructor.
2. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.
3. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining*..

Blank Final in Java

A final variable in Java can be assigned a value only once, we can assign a value either in declaration or later.

```
final int i = 10;
i = 30; // Error because i is final.
```

A **blank final** variable in Java is a final variable that is not initialized during declaration. Below is a simple example of blank final.

```
// A simple blank final example
final int i;
i = 30;
```

How are values assigned to blank final members of objects?
Values must be assigned in constructor.

```
// A sample Java program to demonstrate use and
// working of blank final
class Test
{
    // We can initialize here, but if we
    // initialize here, then all objects get
    // the same value.  So we use blank final
    final int i;

    Test(int x)
    {
        // Since we have initialized above, we
        // must initialize i in constructor.
        // If we remove this line, we get compiler
        // error.
        i = x;
    }
}

// Driver Code
class Main
{
    public static void main(String args[])
    {
        Test t1 = new Test(10);
        System.out.println(t1.i);

        Test t2 = new Test(20);
        System.out.println(t2.i);
    }
}
```

Output:

```
10
20
```

If we have more than one constructors or overloaded constructor in class, then blank final variable must be initialized in all of them. However constructor chaining can be used to initialize the blank final variable.

```
// A Java program to demonstrate that we can
// use constructor chaining to initialize
// final members
class Test
{
    final public int i;

    Test(int val)    {  this.i = val;  }

    Test()
    {
        // Calling Test(int val)
        this(10);
    }

    public static void main(String[] args)
    {
        Test t1 = new Test();
        System.out.println(t1.i);

        Test t2 = new Test(20);
        System.out.println(t2.i);
    }
}
```

Output:

```
10
20
```

Blank final variables are used to create immutable objects (objects whose members can't be changed once initialized).

How are parameters passed in Java?

In Java, parameters are always passed by value. For example, following program prints i = 10, j = 20.

```
//  Test.java

class Test {

   // swap() doesn't swap i and j

   public static void swap(Integer i, Integer j) {

      Integer temp = new Integer(i);

      i = j;

      j = temp;

   }

   public static void main(String[] args) {

      Integer i = new Integer(10);

      Integer j = new Integer(20);

      swap(i, j);

      System.out.println("i = " + i + ", j = " + j);

   }

}
```

(Java is Strictly Pass by Value)

Consider the following Java program that passes a **primitive type** to function.

```
public class Main
{
    public static void main(String[] args)
    {
        int x = 5;
        change(x);
        System.out.println(x);
    }
    public static void change(int x)
    {
        x = 10;
    }
}
```

Output:

We pass an int to the function “change()” and as a result the change in the value of that integer is not reflected in the main method. Like C/C++, Java creates a copy of the variable being passed in the method and then do the manipulations. Hence the change is not reflected in the main method.

How about objects or references?

In Java, all primitives like int, char, etc are similar to C/C++, but all non-primitives (or objects of any class) are always references. So it gets tricky when we pass object references to methods. Java creates a copy of references and pass it to method, but they still point to same memory reference. Mean if set some other object to reference passed inside method, the object from calling method as well its reference will remain unaffected.

The changes are not reflected back if we change the object itself to refer some other location or object

If we assign reference to some other location, then changes are not reflected back in main().

```
// A Java program to show that references are also passed
// by value.
class Test
{
    int x;
    Test(int i) { x = i; }
    Test()      { x = 0; }
}

class Main
{
    public static void main(String[] args)
    {
        // t is a reference
        Test t = new Test(5);

        // Reference is passed and a copy of reference
        // is created in change()
        change(t);

        // Old value of t.x is printed
        System.out.println(t.x);
    }
    public static void change(Test t)
    {
        // We changed reference to refer some other location
        // now any changes made to reference are not reflected
        // back in main
        t = new Test();

        t.x = 10;
    }
}
```

Output:

```
5
```

Changes are reflected back if we do not assign reference to a new location or object:
If we do not change the reference to refer some other object (or memory location), we can make changes to the members and these changes are reflected back.

```
// A Java program to show that we can change members using using
// reference if we do not change the reference itself.
class Test
{
    int x;
    Test(int i) { x = i; }
    Test()      { x = 0; }
}

class Main
{
    public static void main(String[] args)
    {
        // t is a reference
        Test t = new Test(5);

        // Reference is passed and a copy of reference
        // is created in change()
        change(t);

        // New value of x is printed
        System.out.println(t.x);
    }

    // This change() doesn't change the reference, it only
    // changes member of object referred by reference
    public static void change(Test t)
    {
        t.x = 10;
    }
}
```

Output:

```
10
```

Exercise: Predict the output of following Java program

```
//  Test.java
class Main {
   // swap() doesn't swap i and j
   public static void swap(Integer i, Integer j)
```

```
    {
       Integer temp = new Integer(i);
       i = j;
       j = temp;
    }
    public static void main(String[] args)
    {
       Integer i = new Integer(10);
       Integer j = new Integer(20);
       swap(i, j);
       System.out.println("i = " + i + ", j = " + j);
    }
}
```

Returning Multiple values in Java

Java doesn't support multi-value returns. We can use following solutions to return multiple values.

If all returned elements are of same type

We can return an array in Java. Below is a Java program to demonstrate the same.

```
// A Java program to demonstrate that a method
// can return multiple values of same type by
// returning an array
class Test
{
    // Returns an array such that first element
    // of array is a+b, and second element is a-b
    static int[] getSumAndSub(int a, int b)
    {
        int[] ans = new int[2];
        ans[0] = a + b;
        ans[1] = a - b;

        // returning array of elements
        return ans;
    }

    // Driver method
    public static void main(String[] args)
    {
        int[] ans = getSumAndSub(100,50);
        System.out.println("Sum = " + ans[0]);
        System.out.println("Sub = " + ans[1]);
    }
}
```

The output of the above code will be:

```
Sum = 150
Sub = 50
```

If returned elements are of different types

We can be encapsulate all returned types into a class and then return an object of that class.

Let us have a look at the following code.

```
// A Java program to demonstrate that we can return
```

```
// multiple values of different types by making a class
// and returning an object of class.

// A class that is used to store and return
// two members of different types
class MultiDiv
{
    int mul;    // To store multiplication
    double div; // To store division
    MultiDiv(int m, double d)
    {
        mul = m;
        div = d;
    }
}

class Test
{
    static MultiDiv getMultandDiv(int a, int b)
    {
        // Returning multiple values of different
        // types by returning an object
        return new MultiDiv(a*b, (double)a/b);
    }

    // Driver code
    public static void main(String[] args)
    {
        MultiDiv ans = getMultandDiv(10, 20);
        System.out.println("Multiplication = " + ans.mul);
        System.out.println("Division = " + ans.div);
    }
}
```

Output:

```
 Multiplication = 200
Division = 0.5
```

Function overloading and return type

In C++ and Java, functions can not be overloaded if they differ only in the return type.

For example, the following program C++ and Java programs fail in compilation.

C++ Program

```
#include<iostream>
int foo() {
    return 10;
}

char foo() {  // compiler error; new declaration of foo()
    return 'a';
}

int main()
{
    char x = foo();
    getchar();
    return 0;
}
```

Java Program

```
// filename Main.java
public class Main {
    public int foo() {
        return 10;
    }
    public char foo() { // compiler error: foo() is already defined
        return 'a';
    }
    public static void main(String args[])
    {
    }
}
```

Overriding equals method in Java

Consider the following Java program:

```
class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
}

// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(10, 15);
        if (c1 == c2) {
            System.out.println("Equal ");
        } else {
            System.out.println("Not Equal ");
        }
    }
}
```

Output:

```
Not Equal
```

The reason for printing “Not Equal” is simple: when we compare c1 and c2, it is checked whether both c1 and c2 refer to same object or not (object variables are always references in Java). c1 and c2 refer to two different objects, hence the value (c1 == c2) is false. If we create another reference say c3 like following, then (c1 == c3) will give true.

```
Complex c3 = c1;  // (c3 == c1) will be true
```

So, how do we check for equality of values inside the objects? All classes in Java inherit from the Object class, directly or indirectly (See point 1 of this). The Object class has some basic methods like clone(), toString(), equals(),.. etc. We can override the equals method in our class to check whether two objects have same data or not.

```
class Complex {

    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
```

```
    // Overriding equals() to compare two Complex objects
    @Override
    public boolean equals(Object o) {

        // If the object is compared with itself then return true
        if (o == this) {
            return true;
        }

        /* Check if o is an instance of Complex or not
          "null instanceof [type]" also returns false */
        if (!(o instanceof Complex)) {
            return false;
        }

        // typecast o to Complex so that we can compare data members
        Complex c = (Complex) o;

        // Compare the data members and return accordingly
        return Double.compare(re, c.re) == 0
                && Double.compare(im, c.im) == 0;
    }
}

// Driver class to test the Complex class
public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(10, 15);
        if (c1.equals(c2)) {
            System.out.println("Equal ");
        } else {
            System.out.println("Not Equal ");
        }
    }
}
```

Output:

```
Equal
```

As a side note, when we override equals(), it is recommended to also override the hashCode() method. If we don't do so, equal objects may get different hash-values; and hash based collections, including HashMap, HashSet, and Hashtable do not work properly (see this for more details). We will be coverig more about hashCode() in a separate post.

Overriding toString() in Java

This post is similar to Overriding equals method in Java. Consider the following Java program:

```
// file name: Main.java

class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
}

// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        System.out.println(c1);
    }
}
```

Output:

```
Complex@19821f
```

The output is, *class name, then 'at' sign, and at the end hashCode of object*. All classes in Java inherit from the Object class, directly or indirectly (See point 1 of this). The Object class has some basic methods like clone(), toString(), equals(),.. etc. The default toString() method in Object prints "class name @ hash code". We can override toString() method in our class to print proper output. For example, in the following code toString() is overridden to print "Real + i Imag" form.

```
// file name: Main.java

class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    /* Returns the string representation of this Complex number.
       The format of string is "Re + iIm" where Re is real part
       and Im is imagenary part.*/
    @Override
    public String toString() {
        return String.format(re + " + i" + im);
    }
}
```

```
// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        System.out.println(c1);
    }
}
```

Output:

```
10.0 + i15.0
```

In general, it is a good idea to override toString() as we get get proper output when an object is used in print() or println().

Private and final methods in Java

When we use *final* specifier with a method, the method cannot be overridden in any of the inheriting classes. Methods are made final due to design reasons.
Since private methods are inaccessible, they are implicitly final in Java. So adding *final* specifier to a private method doesn't add any value. It may in-fact cause unnecessary confusion.

```
class Base {

   private final void foo() {}

   // The above method foo() is same as following. The keyword
   // final is redundant in above declaration.

   // private void foo() {}
}
```

For example, both 'program 1′ and 'program 2′ below produce same compiler error "foo() has private access in Base".

Program 1

```
// file name: Main.java
class Base {
    private final void foo() {}
}

class Derived extends Base {
    public void foo() {}
}

public class Main {
    public static void main(String args[]) {
        Base b = new Derived();
        b.foo();
    }
}
```

Program 2

```
// file name: Main.java
class Base {
    private void foo() {}
}

class Derived extends Base {
    public void foo() {}
}

public class Main {
```

```
    public static void main(String args[]) {
        Base b = new Derived();
        b.foo();
    }
}
```

GFact 48 | Overloading main() in Java

Consider the below Java program.

```
// A Java program with overloaded main()
import java.io.*;

public class Test {

    // Normal main()
    public static void main(String[] args) {
        System.out.println("Hi Geek (from main)");
        Test.main("Geek");
    }

    // Overloaded main methods
    public static void main(String arg1) {
        System.out.println("Hi, " + arg1);
        Test.main("Dear Geek","My Geek");
    }
    public static void main(String arg1, String arg2) {
        System.out.println("Hi, " + arg1 + ", " + arg2);
    }
}
```

Output:

```
Hi Geek (from main)
Hi, Geek
Hi, Dear Geek, My Geek
```

Important Points:
The main method in Java is no extra-terrestrial method. Apart from the fact that main() is just like any other method & can be overloaded in a similar manner, JVM always looks for the method signature to launch the program.

- The normal main method acts as an entry point for the JVM to start the execution of program.
- We can overload the main method in Java. But the program doesn't execute the overloaded main method when we run your program, we need to call the overloaded main method from the actual main method only.

Valid variants of main() in Java

Below are different variants of main() that are valid.

1. **Default prototype:** Below is the most common way to write main() in Java.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

- Output:

```
Main Method
```

Meaning of the main Syntax:

```
public:  JVM can execute the method from anywhere.
static:  Main method can be called without object.
void:    The main method doesn't return anything.
main():  Name configured in the JVM.
String[]: Accepts the command line arguments.
```

- **Order of Modifiers:** We can swap positions of static and public in main().

```
class Test
{
    static public void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

- Output:

```
Main Method
```

- **Variants of String array arguments:** We can place square brackets at different positions and we can use varargs (…) for string parameter.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
```

```
class Test
{
    public static void main(String []args)
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
class Test
{
    public static void main(String args[])
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
class Test
{
    public static void main(String...args)
    {
        System.out.println("Main Method");
    }
}
```

- Output:

```
Main Method
```

- **Final Modifier String argument:** We can make String args[] as final.

```
class Test
{
    public static void main(final String[] args)
    {
        System.out.println("Main Method");
    }
}
```

- Output:

```
Main Method
```

- **Final Modifier to static main method:** We can make main() as final.

```
class Test
{
    public final static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

- Output:

```
Main Method
```

- **synchronized keyword to static main method:** We can make main() synchronized.

```
class Test
{
    public synchronized static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

- Output:

```
Main Method
```

- **strictfp keyword to static main method:** strictfp can be used to restrict floating point calculations.

```
class Test
{
    public strictfp static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

- Output:

```
Main Method
```

- **Combinations of all above keyword to static main method:**

```
class Test
{
    final static synchronized strictfp static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

- Output:

```
Main Method
```

- **Overloading Main method:** We can overload main() with different types of parameters.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Main Method String Array");
    }
    public static void main(int[] args)
    {
        System.out.println("Main Method int Array");
    }
}
```

- Output:

```
Main Method String Array
```

- **Inheritance of Main method:** JVM Executes the main() without any errors.

```
class A
{
    public static void main(String[] args)
    {
        System.out.println("Main Method Parent");
    }
}

class B extends A
{

}
```

- Two class files, A.class and B.class are generated by compiler. When we execute any of the two .class, JVM executes with no error.

```
O/P: Java A
Main Method Parent
O/P: Java B
Main Method Parent
```

- **Method Hiding of main(), but not Overriding:** Since main() is static, derived class main() hides the base class main. (See Shadowing of static functions for details.)

```
class A
{
    public static void main(String[] args)
    {
        System.out.println("Main Method Parent");
    }
}
class B extends A
{
    public static void main(String[] args)
    {
        System.out.println("Main Method Child");
    }
}
```

Two classes, A.class and B.class are generated by Java Compiler javac. When we execute both the .class, JVM executes with no error.

```
O/P: Java A
Main Method Parent
O/P: Java B
Main Method Child
```

Variable Arguments (Varargs) in Java

Variable arguments (or varargs) were introduced in JDK 5. Below is a code snippet for illustrating the concept.

```
// A simple Java program to demonstrate varargs
class Test1
{
    // A method that takes variable number of intger
    // arguments.
    static void fun(int ...a)
    {
        System.out.println("Number of arguments is: " + a.length);

        // using for each loop to display contents of a
        for (int i: a)
            System.out.print(i + " ");
        System.out.println();
    }

    // Driver code
    public static void main(String args[])
    {
        // Calling the varargs method with different number
        // of parameters
        fun(100);         // one parameter
        fun(1, 2, 3, 4);  // four parameters
        fun();            // no parameter
    }
}
```

Output:

```
Number of arguments is: 1
100
Number of arguments is: 4
1 2 3 4
Number of arguments is: 0
```

Explanation of above program :

- The … syntax tells the compiler that varargs has been used and these arguments should be stored in the **array referred to by a**.
- The variable **a** is operated on as an array. In this case, we have defined the data type of a as int. So it can take only integer values. The number of arguments can be found out using a.length, the way we find the length of an array in Java.

A method can have normal parameters along with the variable length parameters but one should ensure that there **exists only one varargs parameter** that should be written last in the parameter list of the method declaration.

```
int nums(int a, float b, double … c)
```

In this case, the first two arguments are matched with the first two parameters and the remaining arguments belong to c.

An example program showcasing the use of varargs along with normal parameters :

```
// A simple Java program to demonstrate varargs with normal
// arguments
class Test2
{
    // Takes string as a argument followed by varargs
    static void fun2(String str, int ...a)
    {
        System.out.println("String: " + str);
        System.out.println("Number of arguments is: "+ a.length);

        // using for each loop to display contents of a
        for (int i: a)
            System.out.print(i + " ");

        System.out.println();
    }

    public static void main(String args[])
    {
        // Calling fun2() with different parameter
        fun2("HelloWorld", 100, 200);
        fun2("CSPortal", 1, 2, 3, 4, 5);
        fun2("forGeeks");
    }
}
String: HelloWorld
Number of arguments is: 2
100 200
String: CSportal
Number of arguments is: 5
1 2 3 4 5
String: forGeeks
Number of arguments is: 0
```

Below are some important facts about varargs.

- Vararg Methods can also be overloaded but overloading may lead to ambiguity.
- Prior to JDK 5, variable length arguments could be handled into two ways : One was using overloading, other was using array argument.
- There can be only one variable argument in a method.
- Variable argument (varargs) must be the last argument.

Examples of varargs that are **erroneous:**

```
void method(String... gfg, int... q)
```

```
{
    // Compile time error as there are two
    // varargs
}
void method(int... gfg, String q)
{
    // Compile time error as vararg appear
    // before normal argument
}
```

Primitive Wrapper Classes are Immutable in Java

Consider below Java program.

```
// Java program to demonstrate that prmitive
// wrapper classes are immutable
class Demo
{
    public static void main(String[] args)
    {
        Integer i = new Integer(12);
        System.out.println(i);
        modify(i);
        System.out.println(i);
    }

    private static void modify(Integer i)
    {
        i = i + 1;
    }
}
```

Output :

```
12
12
```

The parameter i is reference in modify and refers to same object as i in main(), but changes made to i are not reflected in main(), why?

Explanation:
All primitive wrapper classes (Integer, Byte, Long, Float, Double, Character, Boolean and Short) are immutable in Java, so operations like addition and subtraction create a new object and not modify the old.

The below line of code in the modify method is operating on wrapper class Integer, not an int

i = i + 1;

It does the following:

1. Unbox i to an int value
2. Add 1 to that value
3. Box the result into another Integer object
4. Assign the resulting Integer to i (thus changing what object i references)

Since object references are passed by value, the action taken in the modify method does not change i that was used as an argument in the call to modify. Thus the main routine still prints 12 after the method returns.

Clone() method in Java

Object cloning refers to creation of exact copy of an object. It creates a new instance of the class of current object and initializes all its fields with exactly the contents of the corresponding fields of this object.

Using Assignment Operator to create copy of reference variable
In Java, there is no operator to create copy of an object. Unlike C++, in Java, if we use assignment operator then it will create a copy of reference variable and not the object. This can be explained by taking an example. Following program demonstrates the same.

```
// Java program to demonstrate that assignment
// operator only creates a new reference to same
// object.
import java.io.*;

// A test class whose objects are cloned
class Test
{
    int x, y;
    Test()
    {
        x = 10;
        y = 20;
    }
}

// Driver Class
class Main
{
    public static void main(String[] args)
    {
         Test ob1 = new Test();

         System.out.println(ob1.x + " " + ob1.y);

         // Creating a new reference variable ob2
         // pointing to same address as ob1
         Test ob2 = ob1;

         // Any change made in ob2 will be reflected
         // in ob1
         ob2.x = 100;

         System.out.println(ob1.x+" "+ob1.y);
         System.out.println(ob2.x+" "+ob2.y);
    }
```

```
}
```

Output:

```
10 20
100 20
100 20
```

Creating a copy using clone() method

The class whose object's copy is to be made must have a public clone method in it or in one of its parent class.

- Every class that implements clone() should call super.clone() to obtain the cloned object reference.
- The class must also implement java.lang.Cloneable interface whose object clone we want to create otherwise it will throw CloneNotSupportedException when clone method is called on that class's object.
- Syntax:

```
protected Object clone() throws CloneNotSupportedException
```

Usage of clone() method -Shallow Copy

```
// A Java program to demonstrate shallow copy
// using clone()
import java.util.ArrayList;

// An object reference of this class is
// contained by Test2
class Test
{
    int x, y;
}

// Contains a reference of Test and implements
// clone with shallow copy.
class Test2 implements Cloneable
{
    int a;
    int b;
    Test c = new Test();
    public Object clone() throws
                    CloneNotSupportedException
    {
        return super.clone();
    }
}

// Driver class
public class Main
{
```

```
    public static void main(String args[]) throws
                        CloneNotSupportedException
    {
       Test2 t1 = new Test2();
       t1.a = 10;
       t1.b = 20;
       t1.c.x = 30;
       t1.c.y = 40;

       Test2 t2 = (Test2)t1.clone();

       // Creating a copy of object t1 and passing
       //  it to t2
       t2.a = 100;

       // Change in primitive type of t2 will not
       // be reflected in t1 field
       t2.c.x = 300;

       // Change in object type field will be
       // reflected in both t2 and t1(shallow copy)
       System.out.println(t1.a + " " + t1.b + " " +
                        t1.c.x + " " + t1.c.y);
       System.out.println(t2.a + " " + t2.b + " " +
                        t2.c.x + " " + t2.c.y);
    }
}
```

Output:

```
10 20 300 40
100 20 300 40
```

In the above example, t1.clone returns the shallow copy of the object t1. To obtain a deep copy of the object certain modifications have to be made in clone method after obtaining the copy.

Deep Copy vs Shallow Copy

- **Shallow copy** is method of copying an object and is followed by default in cloning. In this method the fields of an old object X are copied to the new object Y. While copying the object type field the reference is copied to Y i.e object Y will point to same location as pointed out by X. If the field value is a primitive type it copies the value of the primitive type.
- Therefore, any changes made in referenced objects in object X or Y will be reflected in other object.

Shallow copies are cheap and simple to make. In above example, we created a shallow copy of object.

Usage of clone() method – Deep Copy

- If we want to create a deep copy of object X and place it in a new object Y then new copy of any referenced objects fields are created and these references are placed in object Y. This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other. In below example, we create a deep copy of object.
- A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

```
// A Java program to demonstrate deep copy
// using clone()
import java.util.ArrayList;

// An object reference of this class is
// contained by Test2
class Test
{
    int x, y;
}



// Contains a reference of Test and implements
// clone with deep copy.
class Test2 implements Cloneable
{
    int a, b;

    Test c = new Test();

    public Object clone() throws
                CloneNotSupportedException
    {
        // Assign the shallow copy to new refernce variable t
        Test2 t = (Test2)super.clone();

        t.c = new Test();

        // Create a new object for the field c
        // and assign it to shallow copy obtained,
        // to make it a deep copy
        return t;
    }
}

public class Main
{
    public static void main(String args[]) throws
                              CloneNotSupportedException
    {
       Test2 t1 = new Test2();
       t1.a = 10;
       t1.b = 20;
       t1.c.x = 30;
```

```
        t1.c.y = 40;

        Test2 t3 = (Test2)t1.clone();
        t3.a = 100;

        // Change in primitive type of t2 will not
        // be reflected in t1 field
        t3.c.x = 300;

        // Change in object type field of t2 will not
        // be reflected in t1(deep copy)
        System.out.println(t1.a + " " + t1.b + " " +
                           t1.c.x + " " + t1.c.y);
        System.out.println(t3.a + " " + t3.b + " " +
                           t3.c.x + " " + t3.c.y);
    }
}
```

Output:

```
10 20 30 40
100 20 300 0
```

In the above example, we can see that a new object for Test class has been assigned to copy object that will be returned in clone method.Due to this t3 will obtain a deep copy of the object t1. So any changes made in ‘c’ object fields by t3 ,will not be reflected in t1.

Advantages of clone method:

- If we use assignment operator to assign an object reference to another reference variable then it will point to same address location of the old object and no new copy of the object will be created. Due to this any changes in reference variable will be reflected in original object.
- If we use copy constructor, then we have to copy all of the data over explicitly i.e. we have to reassign all the fields of the class in constructor explicitly. But in clone method this work of creating a new copy is done by the method itself.So to avoid extra processing we use object cloning.

Remote Method Invocation in Java

Remote Method Invocation (RMI) is an API which allows an object to invoke a method on an object that exists in another address space, which could be on the same machine or on a remote machine. Through RMI, object running in a JVM present on a computer (Client side) can invoke methods on an object present in another JVM (Server side). RMI creates a public remote server object that enables client and server side communications through simple method calls on the server object.

Working of RMI

The communication between client and server is handled by using two intermediate objects: Stub object (on client side) and Skeleton object (on server side).

Stub Object

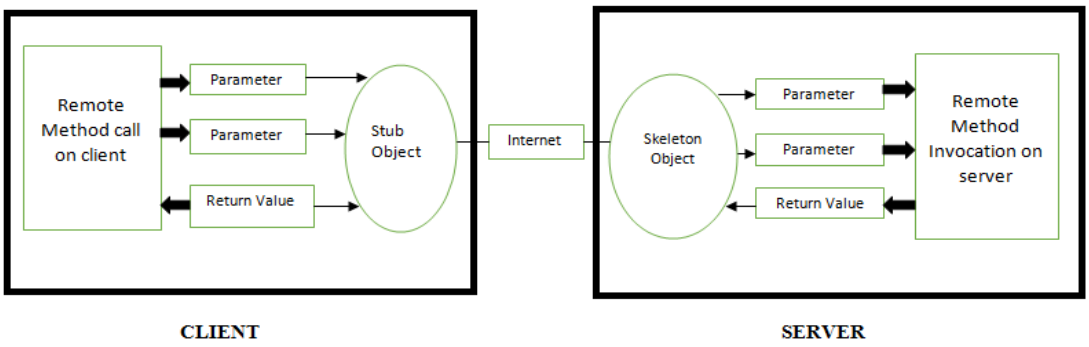
The stub object on the client machine builds an information block and sends this information to t he server. The block consists of

- An identifier of the remote object to be used
- Method name which is to be invoked
- Parameters to the remote JVM

Skeleton Object

The skeleton object passes the request from the stub object to the remote object.
It performs following tasks

- It calls the desired method on the real object present on the server.
- It forwards the parameters received from the stub object to the method.



WORKING OF RMI

Steps to implement Interface

1. Defining a remote interface
2. Implementing the remote interface
3. Creating Stub and Skeleton objects from the implementation class using rmic (rmi compli er)
4. Start the rmiregistry
5. Create and execute the server application program
6. Create and execute the client application program.

Step 1: Defining the remote interface
The first thing to do is to create an interface which will provide the description of the methods that can be invoked by remote clients. This interface should extend the Remote interface and the method prototype within the interface should throw the RemoteException.

```
// Creating a Search interface
import java.rmi.*;
public interface Search extends Remote
{
    // Declaring the method prototype
    public String query(String search) throws RemoteException;
}
```

Step 2: Implementing the remote interface

The next step is to implement the remote interface. To implement the remote interface, the class should extend to UnicastRemoteObject class of java.rmi package. Also, a default constructor needs to be created to throw the java.rmi.RemoteException from its parent constructor in class.

```
// Java program to implement the Search interface
import java.rmi.*;
import java.rmi.server.*;
public class SearchQuery extends UnicastRemoteObject
                         implements Search
{
    // Default constructor to throw RemoteException
    // from its parent constructor
    SearchQuery() throws RemoteException
    {
        super();
    }

    // Implementation of the query interface
    public String query(String search)
                        throws RemoteException
    {
        String result;
        if (search.equals("Reflection in Java"))
            result = "Found";
        else
            result = "Not Found";
```

```
        return result;
    }
}
```

Step 3: Creating Stub and Skeleton objects from the implementation class using rmic
The rmic tool is used to invoke the rmi compiler that creates the Stub and Skeleton objects. Its prototype is
rmic classname. For above program the following command need to be executed at the command prompt
rmic SearchQuery

STEP 4: Start the rmiregistry
Start the registry service by issuing the following command at the command prompt start rmiregistry

STEP 5: Create and execute the server application program
The next step is to create the server application program and execute it on a separate command prompt.

- The server program uses createRegistry method of LocateRegistry class to create rmiregistry within the server JVM with the port number passed as argument.
- The rebind method of Naming class is used to bind the remote object to the new name.

```
//program for server application
import java.rmi.*;
import java.rmi.registry.*;
public class SearchServer
{
    public static void main(String args[])
    {
        try
        {
            // Create an object of the interface
            // implementation class
            Search obj = new SearchQuery();

            // rmiregistry within the server JVM with
            // port number 1900
            LocateRegistry.createRegistry(1900);

            // Binds the remote object by the name
            // HelloWorld
            Naming.rebind("rmi://localhost:1900"+
                          "/HelloWorld",obj);
        }
        catch(Exception ae)
        {
            System.out.println(ae);
        }
    }
}
```

Step 6: Create and execute the client application program
The last step is to create the client application program and execute it on a separate command prompt .The lookup method of Naming class is used to get the reference of the Stub object.

```
//program for client application
import java.rmi.*;
public class ClientRequest
{
    public static void main(String args[])
    {
        String answer,value="Reflection in Java";
        try
        {
            // lookup method to find reference of remote object
            Search access =
                (Search)Naming.lookup("rmi://localhost:1900"+
                                      "/HelloWorld");
            answer = access.query(value);
            System.out.println("Article on " + value +
                               " " + answer+" at HelloWorld");
        }
        catch(Exception ae)
        {
            System.out.println(ae);
        }
    }
}
```

Note: The above client and server program is executed on the same machine so localhost is used. In order to access the remote object from another machine, localhost is to be replaced with the IP address where the remote object is present.
Important Observations:

1. RMI is a pure java solution to Remote Procedure Calls (RPC) and is used to create distributed application in java.
2. Stub and Skeleton objects are used for communication between client and server side.

Default constructor in Java

Like C++, Java automatically creates default constructor if there is no default or parameterized constructor written by user, and (like C++) the default constructor automatically calls parent default constructor. But unlike C++, default constructor in Java initializes member data variable to default values (numeric values are initialized as 0, booleans are initialized as *false* and references are initialized as *null*).

For example, output of the below program is

0
null
false
0
0.0

```
// Main.java
class Test {
    int i;
    Test t;
    boolean b;
    byte bt;
    float ft;
}

public class Main {
     public static void main(String args[]) {
       Test t = new Test(); // default constructor is called.
       System.out.println(t.i);
       System.out.println(t.t);
       System.out.println(t.b);
       System.out.println(t.bt);
       System.out.println(t.ft);
     }
}
```

Assigning values to static final variables in Java

Assigning values to static final variables in Java:
In Java, non-static final variables can be assigned a value either in constructor or with the declaration. But, static final variables cannot be assigned value in constructor; they must be assigned a value with their declaration.

For example, following program works fine.

```
class Test {
  final int i;  // i could be assigned a value here also
  Tets() {
    i = 10;
  }

  //other stuff in the class
}
```

If we make *i* as *static final* then we must assign value to i with the delcaration.

```
class Test {
  static final int i = 10;  // Since i is static final, it must be assigned
value here only.

  //other stuff in the class
}
```

Such behavior is obvious as static variables are shared among all the objects of a class; creating a new object would change the same static variable which is not allowed if the static variable is final.

Copy Constructor in Java

Like C++, Java also supports copy constructor. But, unlike C++, Java doesn't create a default copy constructor if you don't write your own.

Following is an example Java program that shows a simple use of copy constructor.

```
// filename: Main.java

class Complex {

    private double re, im;

    // A normal parametrized constructor
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // copy constructor
    Complex(Complex c) {
        System.out.println("Copy constructor called");
        re = c.re;
        im = c.im;
    }
```

```
    // Overriding the toString of Object class
    @Override
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}

public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);

        // Following involves a copy constructor call
        Complex c2 = new Complex(c1);

        // Note that following doesn't involve a copy constructor call as
        // non-primitive variables are just references.
        Complex c3 = c2;

        System.out.println(c2); // toString() of c2 is called here
    }
}
```

Output:

```
Copy constructor called
(10.0 + 15.0i)
```

Now try the following Java program:

```
// filename: Main.java

class Complex {

    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
}

public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(c1);  // compiler error here
    }
}
```

Comparison of Exception Handling in C++ and Java

Both languages use *try*, *catch* and *throw* keywords for exception handling, and meaning of *try*, *catch* and *free* blocks is also same in both languages. Following are the differences between Java and C++ exception handling.

1) In C++, all types (including primitive and pointer) can be thrown as exception. But in Java only throwable objects (Throwable objects are instances of any subclass of the Throwable class) can be thrown as exception. For example, following type of code works in C++, but similar code doesn't work in Java.

```
#include <iostream>
using namespace std;
int main()
{
   int x = -1;

   // some other stuff
   try {
      // some other stuff
      if( x < 0 )
      {
         throw x;
      }
   }
   catch (int x ) {
      cout << "Exception occurred: thrown value is " << x << endl;
   }
   getchar();
   return 0;
}
```

Output:
Exception occurred: thrown value is -1

2) In C++, there is a special catch called "catch all" that can catch all kind of exceptions.

```
#include <iostream>
using namespace std;
int main()
{
   int x = -1;
   char *ptr;

   ptr = new char[256];
```

```
    // some other stuff
    try {
        // some other stuff
        if( x < 0 )
        {
            throw x;
        }
        if(ptr == NULL)
        {
            throw " ptr is NULL ";
        }
    }
    catch (...) // catch all
    {
        cout << "Exception occurred: exiting "<< endl;
        exit(0);
    }

    getchar();
    return 0;
}
```

Output:
Exception occurred: exiting

In Java, for all practical purposes, we can catch Exception object to catch all kind of exceptions. Because, normally we do not catch Throwable(s) other than Exception(s) (which are Errors)

```
catch(Exception e){
  …….
}
```

3) In Java, there is a block called *finally* that is always executed after the try-catch block. This block can be used to do cleanup work. There is no such block in C++.

```
// creating an exception type
class Test extends Exception { }

class Main {
    public static void main(String args[]) {

        try {
            throw new Test();
        }
        catch(Test t) {
            System.out.println("Got the Test Exception");
        }
        finally {
            System.out.println("Inside finally block ");
        }
   }
```

```
}
```

Output:
Got the error
Inside finally block

4) In C++, all exceptions are unchecked. In Java, there are two types of exceptions – checked and unchecked. See this for more details on checked vs Unchecked exceptions.

5) In Java, a new keyword *throws* is used to list exceptions that can be thrown by a function. In C++, there is no *throws* keyword, the same keyword *throw* is used for this purpose also.

Catching base and derived classes as exceptions

Exception Handling – catching base and derived classes as exceptions:

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

If we put base class first then the derived class catch block will never be reached. For example, following C++ code prints *"Caught Base Exception"*

```
#include<iostream>
using namespace std;

class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   // some other stuff
   try {
       // Some monitored code
       throw d;
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
   catch(Derived d) {  //This catch block is NEVER executed
        cout<<"Caught Derived Exception";
   }
```

```
   getchar();
   return 0;
}
```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modifed program and it prints *"Caught Derived Exception"*

```
#include<iostream>
using namespace std;

class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   // some other stuff
   try {
       // Some monitored code
       throw d;
   }
   catch(Derived d) {
        cout<<"Caught Derived Exception";
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
   getchar();
   return 0;
}
```

In Java, catching a base class exception before derived is not allowed by the compiler itself. In C++, compiler might give warning about it, but compiles the code.
For example, following Java code fails in compilation with error message *"exception Derived has already been caught"*

```
//filename Main.java
class Base extends Exception {}
class Derived extends Base  {}
public class Main {
  public static void main(String args[]) {
    try {
       throw new Derived();
    }
    catch(Base b) {}
    catch(Derived d) {}
  }
}
```

Checked vs Unchecked Exceptions in Java

In Java, there two types of exceptions:

1) Checked: are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

For example, consider the following Java program that opens file at locatiobn “C:\test\a.txt” and prints first three lines of it. The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

Output:

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable source
code -
unreported exception java.io.FileNotFoundException; must be caught or
declared to be
thrown
        at Main.main(Main.java:5)
```

To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block. We have used throws in the below program. Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.

```
import java.io.*;

class Main {
    public static void main(String[] args) throws IOException {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

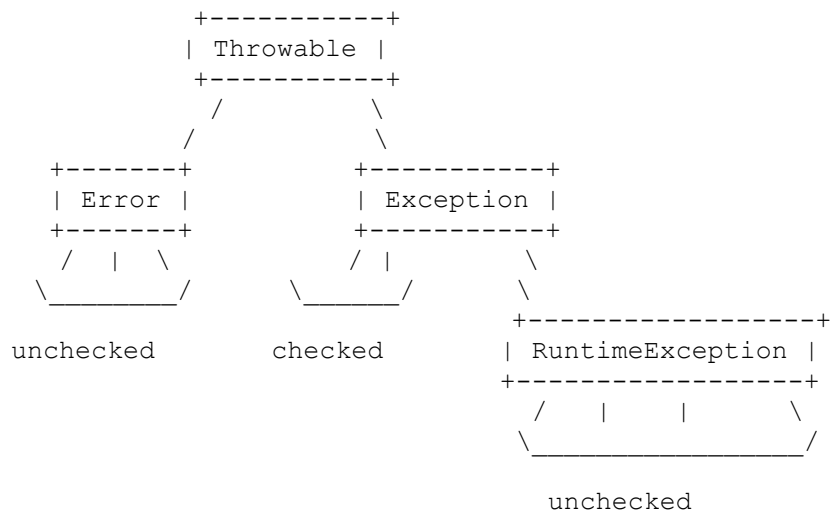
        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
```

```
}
```

Output: First three lines of file “C:\test\a.txt”

2) Unchecked are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.
In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.



Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

```
class Main {
   public static void main(String args[]) {
      int x = 0;
      int y = 10;
      int z = y/x;
  }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:5)
Java Result: 1
```

Why two types?
See Unchecked Exceptions — The Controversy for details.

Should we make our exceptions checked or unchecked?
Following is the bottom line from Java documents

If a client can reasonably be expected to recover from an exception, make it a checked exception.
If a client cannot do anything to recover from the exception, make it an unchecked exception

(User-defined Custom Exception in Java)

Java provides us facility to create our own exceptions which are basically derived classes of Exception. For example MyException in below code extends the Exception class.

We pass the string to the constructor of the super class- Exception which is obtained using "getMessage()" function on the object created.

```
// A Class that represents use-defined expception
class MyException extends Exception
{
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException("GeeksGeeks");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
```

Output:

```
Caught
GeeksGeeks
```

In the above code, constructor of MyException requires a string as its argument. The string is passed to parent class Exception's constructor using super(). The constructor of Exception class can also be called without a parameter and call to super is not mandatory.

```
// A Class that represents use-defined expception
class MyException extends Exception
{

}

// A Class that uses above MyException
public class setText
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException();
        }
        catch (MyException ex)
        {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
}
```

Output:

```
Caught
null
```

(Infinity or Exception?)

Consider the following code snippets:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        double p = 1;
        System.out.println(p/0);
    }
}
```

Output:

```
Infinity
```

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        int p = 1;
        System.out.println(p/0);
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at HelloWorld.main(HelloWorld.java:8)
```

Explanation: In the first piece of code, a double value is being divided by 0 while in the other case an integer value is being divide by 0. However the solution for both of them differs.

- In case of double/float division, the output is **Infinity**, the basic reason behind that it implements the floating point arithmetic algorithm which specifies a special values like “Not a number” OR “infinity” for “divided by zero cases” as per IEEE 754 standards.
- In case of integer division, it throws ArithmeticException.

Multicatch in Java

Background

Prior to Java 7, we had to catch only one exception type in each catch block. So whenever we needed to handle more than one specific exception, but take same action for all exceptions, then we had to have more than one catch block containing the same code.

In the following code, we have to handle two different exceptions but take same action for both. So we needed to have two different catch blocks as of Java 6.0.

```
// A Java program to demonstrate that we needed
// multiple catch blocks for multiple exceptions
// prior to Java 7
import java.util.Scanner;
public class Test
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            if (99%n == 0)
                System.out.println(n + " is a factor of 99");
        }
```

```
        catch (ArithmeticException ex)
        {
            System.out.println("Arithmetic " + ex);
        }
        catch (NumberFormatException ex)
        {
            System.out.println("Number Format Exception " + ex);
        }
    }
}
```

Input 1:

```
HelloWorld
```

Output 2:

```
Exception encountered java.lang.NumberFormatException:
For input string: "HelloWorld"
```

Input 2:

```
0
```

Output 2:

```
Arithmetic Exception encountered java.lang.ArithmeticException: / by zero
```

Multicatch

Starting from Java 7.0, it is possible for a single catch block to catch multiple exceptions by separating each with | (pipe symbol) in catch block.

```
// A Java program to demonstrate multicatch
// feature
import java.util.Scanner;
public class Test
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            if (99%n == 0)
                System.out.println(n + " is a factor of 99");
        }
        catch (NumberFormatException | ArithmeticException ex)
        {
            System.out.println("Exception encountered " + ex);
        }
    }
}
```

Input 1:

```
HelloWorld
```

Output 1:

```
Exception encountered java.lang.NumberFormatException:
For input string: "HelloWorld"
```

Input 2:

```
0
```

Output 2:

```
Exception encountered
java.lang.ArithmeticException: / by zero
```

A catch block that handles multiple exception types creates no duplication in the bytecode generated by the compiler, that is, the bytecode has no replication of exception handlers.

Important Points:

- If all the exceptions belong to the same class hierarchy, we should catching the base exception type. However, to catch each exception, it needs to be done separately in their own catch blocks.
- Single catch block can handle more than one type of exception. However, the base (or ancestor) class and subclass (or descendant) exceptions can not be caught in one statement. For Example
- `// Not Valid as Exception is an ancestor of`
- `// NumberFormatException`
- `catch(NumberFormatException | Exception ex)`
- All the exceptions must be separated by vertical bar pipe |.

Regular Expressions in Java

Regular Expressions or Regex (in short) is an API for defining String patterns that can be used for searching, manipulating and editing a text. It is widely used to define constraint on strings such as password. Regular Expressions are provided under java.util.regex package.

The java.util.regex package primarily consists of the following three classes:

1. **util.regex.Pattern** – Used for defining patterns
2. **util.regex.Matcher** – Used for performing match operations on text using patterns
3. **PatternSyntaxException**– Used for indicating syntax error in a regular expression pattern

java.util.regex.Pattern Class

1. **matches()**– It is used to check if the whole text matches a pattern. Its output is boolean.

```
// A Simple Java program to demonstrate working of
// Pattern.matches() in Java
import java.util.regex.Pattern;

class Demo
{
    public static void main(String args[])
    {
        // Following line prints "true" because the whole
        // text "HelloWorld" matches pattern "geeksforge*ks"
        System.out.println (Pattern.matches("geeksforge*ks",
                                            "HelloWorld"));

        // Following line prints "false" because the whole
        // text "geeksfor" doesn't match pattern "g*geeks*"
        System.out.println (Pattern.matches("g*geeks*",
                                            "geeksfor"));
    }
}
```

Output:

```
true
false
```

20. **compile()**– Used to create a pattern object by compiling a given string that may contain regular expressions. Input may also contains flags like Pattern.CASE_INSENSITIVE, Pattern.COMMENTS, .. etc (See this for details).
21. **split()**– It is used to split a text into multiple strings based on a delimiter pattern.

java.util.regex.Matcher Class

1. **find()** –It is mainly used for searching multiple occurrences of the regular expressions in the text.

2. **start()** – It is used for getting the start index of a match that is being found using find() method.
3. **end()** –It is used for getting the end index of a match that is being found using find() method. It returns index of character next to last matching character

Note that Pattern.matches() checks if whole text matches with a pattern or not. Other methods (demonstrated below) are mainly used to find multiple occurrences of pattern in text.

Java Programs to demonstrate workings of compile(), find(), start(), end() and split() :

1. **Java Program to demonstrate simple pattern searching**

```
// A Simple Java program to demonstrate working of
// String matching in Java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Demo
{
    public static void main(String args[])
    {
        // Create a pattern to be searched
        Pattern pattern = Pattern.compile("geeks");

        // Search above pattern in "HelloWorld.org"
        Matcher m = pattern.matcher("HelloWorld.org");

        // Print starting and ending indexes of the pattern
        // in text
        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));
    }
}
```

Output:

```
Pattern found from 0 to 4
Pattern found from 8 to 12
```

2. **Java Program to demonstrate simple regular expression searching**

```
// A Simple Java program to demonstrate working of
// String matching in Java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Demo
{
    public static void main(String args[])
    {
        // Create a pattern to be searched
        Pattern pattern = Pattern.compile("ge*");

        // Search above pattern in "HelloWorld.org"
```

```
        Matcher m = pattern.matcher("HelloWorld.org");

        // Print starting and ending indexes of the pattern
        // in text
        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));
    }
}
```

Output:

```
Pattern found from 0 to 2
Pattern found from 8 to 10
Pattern found from 16 to 16
```

47. **Java program to demonstrate Case Insensitive Searching**

```
// A Simple Java program to demonstrate working of
// String matching in Java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Demo
{
    public static void main(String args[])
    {
        // Create a pattern to be searched
        Pattern pattern = Pattern.compile("ge*",
Pattern.CASE_INSENSITIVE);

        // Search above pattern in "HelloWorld.org"
        Matcher m = pattern.matcher("HelloWorld.org");

        // Print starting and ending indexes of the pattern
        // in text
        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));
    }
}
```

Output:

```
Pattern found from 0 to 2
Pattern found from 8 to 10
Pattern found from 16 to 16
```

70. **Java program to demonstrate working of split() to split a text based on a delimiter pattern**

```
// Java program to demonstrate working of splitting a text by a
// given pattern
import java.util.regex.Matcher;
import java.util.regex.Pattern;

```

```
class Demo
{
    public static void main(String args[])
    {
        String text = "geeks1for2geeks3";

        // Specifies the string pattern which is to be searched
        String delimiter =  "\\d";
        Pattern pattern = Pattern.compile(delimiter,
                                          Pattern.CASE_INSENSITIVE);

        // Used to perform case insensitive search of the string
        String[] result = pattern.split(text);


        for (String temp: result)
            System.out.println(temp);
    }
}
```

Output:

```
geeks
for
geeks
```

Important Observations/Facts:

1. We create a pattern object by calling Pattern.compile(), there is no constructor. compile() is a static method in Pattern class.
2. Like above, we create a Matcher object using matcher() on objects of Pattern class.
3. Pattern.matches() is also a static method that is used to check if given text as a whole matches pattern or not.
4. find() is used to find multiple occurrences of pattern in text.
5. We can split a text based on a delimiter pattern using split()

Quantifiers in Java

We strongly recommend to refer below post as a prerequisite of this.

Regular Expressions in Java

Quantifiers allow user to specify the number of occurrences to match against. Below are some commonly used quantifiers in Java.

```
X*        Zero or more occurrences of X
X?        Zero or One occurrences of X
X+        One or More occurrences of X
```

```
X{n}      Exactly n occurrences of X
X{n, }    At-least n occurrences of X
X{n, m}   Count of occurrences of X is from n to m
```

The above quantifiers can be made Greedy, Reluctant and Possessive.

Greedy quantifier (Default)
By default, quantifiers are Greedy. Greedy quantifiers try to match the longest text that matches given pattern. Greedy quantifiers work by first reading the entire string before trying any match. If the entire text doesn't match, remove last character and try again, repeating the process until a match is found.

```
// Java program to demonstrate Greedy Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Test
{
    public static void main(String[] args)
    {
        // Making an instance of Pattern class
        // By default quantifier "+" is Greedy
        Pattern p = Pattern.compile("g+");

        // Making an instance of Matcher class
        Matcher m = p.matcher("ggg");

        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));

    }
}
```

Output :

```
Pattern found from 0 to 2
```

Explanation : The pattern **g+** means one or more occurrences of **g**. Text is **ggg**. The greedy matcher would match the longest text even if parts of matching text also match. In this example, **g** and **gg** also match, but the greedy matcher produces **ggg**.

Reluctant quantifier (Appending a **?** after quantifier)
This quantifier uses the approach that is opposite of greedy quantifiers. It starts from first character and processes one character at a time.

```
// Java program to demonstrate Reluctant Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
class Test
{
    public static void main(String[] args)
    {
        // Making an instance of Pattern class
        // Here "+" is a Reluctant quantifier because
        // a "?' is appended after it.
        Pattern p = Pattern.compile("g+?");

        // Making an instance of Matcher class
        Matcher m = p.matcher("ggg");

        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));

    }
}
```

Output :

```
Pattern found from 0 to 0
Pattern found from 1 to 1
Pattern found from 2 to 2
```

Explanation : Since the quantifier is reluctant, it matches the shortest part of test with pattern. It processes one character at a time.

Possessive quantifier (Appending a **+** after quantifier)
This quantifier matches as many characters as it can like greedy quantifier. But if the entire string doesn't match, then it doesn't try removing characters from end.

```
// Java program to demonstrate Possessive Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Test
{
    public static void main(String[] args)
    {
        // Making an instance of Pattern class
        // Here "+" is a Possessive quantifier because
        // a "+' is appended after it.
        Pattern p = Pattern.compile("g++");

        // Making an instance of Matcher class
        Matcher m = p.matcher("ggg");

        while (m.find())
```

```
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));
    }
}
```

Output :

```
Pattern found from 0 to 2
```

Explanation: We get the same output as Greedy because whole text matches the pattern.

Below is an example to show difference between Greedy and Possessive Quantifiers.

```
// Java program to demonstrate difference between Possessive and
// Greedy Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Test
{
    public static void main(String[] args)
    {
        // Create a pattern with Greedy quantifier
        Pattern pg = Pattern.compile("g+g");

        // Create same pattern with possessive quantifier
        Pattern pp = Pattern.compile("g++g");

        System.out.println("Using Greedy Quantifier");
        Matcher mg = pg.matcher("ggg");
        while (mg.find())
            System.out.println("Pattern found from " + mg.start() +
                               " to " + (mg.end()-1));

        System.out.println("\nUsing Possessive Quantifier");
        Matcher mp = pp.matcher("ggg");
        while (mp.find())
            System.out.println("Pattern found from " + mp.start() +
                               " to " + (mp.end()-1));

    }
}
```

Output :

```
Using Greedy Quantifier
Pattern found from 0 to 2

Using Possessive Quantifier
```

In the above example, since first quantifier is greedy, **g+** matches with whole string. If we match **g+** with whole string, **g+g** doesn't match, the Greedy quantifier, removes last character, matches

gg with **g+** and finds a match.
In Possessive quantifier, we start like Greedy. **g+** matches whole string, but matching **g+** with whole string doesn't match **g+g** with **ggg**. Unlike Greedy, since quantifier is possessive, we stop at this point.

Comparison of Inheritance in C++ and Java

The purpose of inheritance is same in C++ and Java. Inheritance is used in both languages for reusing code and/or creating is-a relationship. There are following differences in the way both languages provide support for inheritance.

1) In Java, all classes inherit from the Object class directly or indirectly. Therefore, there is always a single inheritance tree of classes in Java, and Object class is root of the tree. In Java, if we create a class that doesn't inherit from any class then it automatically inherits from Object class . In C++, there is forest of classes; when we create a class that doesn't inherit from anything, we create a new tree in forest.

Following Java example shows that Test class automatically inherits from Object class.

```
class Test {
    // members of test
}
class Main {
  public static void main(String[] args) {
    Test t = new Test();
    System.out.println("t is instanceof Object: " + (t instanceof Object));
  }
}
```

Output:

```
t is instanceof Object: true
```

2) In Java, members of the grandparent class are not directly accessible. See this G-Fact for more details.

3) The meaning of protected member access specifier is somewhat different in Java. In Java, protected members of a class "A" are accessible in other class "B" of same package, even if B doesn't inherit from A (they both have to be in the same package). For example, in the following program, protected members of A are accessible in B.

```
// filename B.java
class A {
    protected int x = 10, y = 20;
}

class B {
    public static void main(String args[]) {
        A a = new A();
        System.out.println(a.x + " " + a.y);
    }
```

```
}
```

4) Java uses *extends* keyword for inheritence. Unlike C++, Java doesn't provide an inheritance specifier like public, protected or private. Therefore, we cannot change the protection level of members of base class in Java, if some data member is public or protected in base class then it remains public or protected in derived class. Like C++, private members of base class are not accessible in derived class.
Unlike C++, in Java, we don't have to remember those rules of inheritance which are combination of base class access specifier and inheritance specifier.

5) In Java, methods are virtual by default. In C++, we explicitly use virtual keyword. See this G-Fact for more details.

6) Java uses a separte keyword *interface* for interfaces, and *abstract* keyword for abstract classes and abstract functions.

Following is a Java abstract class example.

```
// An abstract class example
abstract class myAbstractClass {

   // An abstract method
   abstract void myAbstractFun();

   // A normal method
   void fun() {
      System.out.println("Inside My fun");
   }
}

public class myClass extends myAbstractClass {
   public void myAbstractFun() {
      System.out.println("Inside My fun");
   }
}
```

Following is a Java interface example

```
// An interface example
public interface myInterface {
   // myAbstractFun() is public and abstract, even if we don't use these
keywords
   void myAbstractFun();  // is same as public abstract void myAbstractFun()
}
```

```
// Note the implements keyword also.
public class myClass implements myInterface {
   public void myAbstractFun() {
      System.out.println("Inside My fun");
   }
}
```

7) Unlike C++, Java doesn't support multiple inheritance. A class cannot inherit from more than one class. A class can implement multiple interfaces though.

8) In C++, default constructor of parent class is automatically called, but if we want to call parametrized constructor of a parent class, we must use Initializer list. Like C++, default constructor of the parent class is automatically called in Java, but if we want to call parametrized constructor then we must use super to call the parent constructor. See following Java example.

```
package main;

class Base {
    private int b;
    Base(int x) {
        b = x;
        System.out.println("Base constructor called");
    }
}

class Derived extends Base {
    private int d;
    Derived(int x, int y) {
        // Calling parent class parameterized constructor
        // Call to parent constructor must be the first line in a Derived
class
        super(x);
        d = y;
        System.out.println("Derived constructor called");
    }
}

class Main{
    public static void main(String[] args) {
      Derived obj = new Derived(1, 2);
    }
}
```

Output:

```
Base constructor called
Derived constructor called
```

Inheritance and constructors in Java

In Java, constructor of base class with no argument gets automatically called in derived class constructor. For example, output of following program is:

Base Class Constructor Called
Derived Class Constructor Called

```
// filename: Main.java
class Base {
  Base() {
    System.out.println("Base Class Constructor Called ");
  }
}

class Derived extends Base {
  Derived() {
    System.out.println("Derived Class Constructor Called ");
  }
}

public class Main {
  public static void main(String[] args) {
    Derived d = new Derived();
  }
}
```

But, if we want to call parameterized contructor of base class, then we can call it using super(). The point to note is **base class comstructor call must be the first line in derived class constructor**. For example, in the following program, super(_x) is first line derived class constructor.

```
// filename: Main.java
class Base {
  int x;
  Base(int _x) {
    x = _x;
  }
}

class Derived extends Base {
  int y;
  Derived(int _x, int _y) {
    super(_x);
    y = _y;
  }
  void Display() {
    System.out.println("x = "+x+", y = "+y);
  }
```

```
}

public class Main {
  public static void main(String[] args) {
    Derived d = new Derived(10, 20);
    d.Display();
  }
}
```

Output:
x = 10, y = 20

How does default virtual behavior differ in C++ and Java ?

Default virtual behavior of methods is opposite in C++ and Java:

In C++, class member methods are non-virtual by default. They can be made virtual by using *virtual* keyword. For example, *Base::show()* is non-virtual in following program and program prints *"Base::show() called"*.

```
#include<iostream>

using namespace std;

class Base {
public:

    // non-virtual by default
    void show() {
         cout<<"Base::show() called";
    }
};

class Derived: public Base {
public:
    void show() {
         cout<<"Derived::show() called";
    }
};

int main()
{
  Derived d;
  Base &b = d;
  b.show();
  getchar();
  return 0;
```

```
}
```

Adding *virtual* before definition of *Base::show()* makes program print *"Derived::show() called"*

In Java, methods are virtual by default and can be made non-virtual by using *final* keyword. For example, in the following java program, *show()* is by default virtual and the program prints *"Derived::show() called"*

```
class Base {

    // virtual by default
    public void show() {
       System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public void show() {
       System.out.println("Derived::show() called");
    }
}

public class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
        b.show();
    }
}
```

Unlike C++ non-virtual behavior, if we add *final* before definition of show() in *Base* , then the above program fails in compilation.

Accessing Grandparent's member in Java

Directly accessing Grandparent's member in Java:

Predict the output of following Java program.

```
// filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
```

```
    public void Print() {
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.super.Print();  // Trying to access Grandparent's Print()
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output: Compiler Error

There is error in line "super.super.print();". In Java, a class cannot directly access the grandparent's members. It is allowed in C++ though. In C++, we can use scope resolution operator (::) to access any ancestor's member in inheritance hierarchy. ***In Java, we can access grandparent's members only through the parent class.*** For example, the following program compiles and runs fine.

```
// filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
    public void Print() {
        super.Print();
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.Print();
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output:

```
Grandparent's Print()
Parent's Print()
Child's Print()
```

Shadowing of static functions in Java

In Java, if name of a derived class static function is same as base class static function then the derived class static function shadows (or conceals) the base class static function. For example, the following Java code prints *"A.fun()"*

```
// file name: Main.java
class A {
   static void fun() {
      System.out.println("A.fun()");
   }
}

class B extends A {
   static void fun() {
      System.out.println("B.fun()");
   }
}

public class Main {
   public static void main(String args[]) {
      A a = new B();
      a.fun();  // prints A.fun()
   }
}
```

If we make both A.fun() and B.fun() as non-static then the above program would print "B.fun()".

Can we override private methods in Java?

Let us first consider the following Java program as a simple example of Overriding or Runtime Polymorphism.

```
class Base {
  public void fun() {
     System.out.println("Base fun");
  }
}

class Derived extends Base {
```

```
  public void fun() {  // overrides the Base's fun()
     System.out.println("Derived fun");
  }
  public static void main(String[] args) {
      Base obj = new Derived();
      obj.fun();
  }
}
```

The program prints “Derived fun”.
The Base class reference ‘obj’ refers to a derived class object (see expression “Base obj = new Derived()”). When fun() is called on obj, the call is made according to the type of referred object, not according to the reference.

Is Overiding possible with private methods?
Predict the output of following program.

```
class Base {
  private void fun() {
     System.out.println("Base fun");
  }
}

class Derived extends Base {
  private void fun() {
     System.out.println("Derived fun");
  }
  public static void main(String[] args) {
      Base obj = new Derived();
      obj.fun();
  }
}
```

We get compiler error “fun() has private access in Base” (See this). So the compiler tries to call base class function, not derived class, means fun() is not overridden.

An inner class can access private members of its outer class. What if we extend an inner class and create fun() in the inner class?
An Inner classes can access private members of its outer class, for example in the following program, *fun()* of *Inner* accesses private data member *msg* which is fine by the compiler.

```
/* Java program to demonstrate whether we can override private method
   of outer class inside its inner class */
class Outer {
     private String msg = "HelloWorld";
     private void fun() {
          System.out.println("Outer fun()");
     }

     class Inner extends Outer {
         private void fun()  {
```

```
                System.out.println("Accessing Private Member of Outer: " +
msg);
            }
      }

      public static void main(String args[])  {

            // In order to create instance of Inner class, we need an Outer
            // class instance. So, first create Outer class instance and then
            // inner class instance.
            Outer o = new Outer();
            Inner  i   = o.new Inner();

            // This will call Inner's fun, the purpose of this call is to
            // show that private members of Outer can be accessed in Inner.
            i.fun();

            // o.fun() calls Outer's fun (No run-time polymorphism).
            o = i;
            o.fun();
      }
}
```

Output:

```
Accessing Private Member of Outer: HelloWorld
Outer fun()
```

In the above program, we created an outer class and an inner class. We extended Inner from Outer and created a method fun() in both Outer and Inner. If we observe our output, then it is clear that the method fun() has not been overriden. It is so because ***private methods are bonded during compile time and it is the type of the reference variable – not the type of object that it refers to – that determines what method to be called.***. As a side note, private methods may be performance-wise better (compared to non-private and non-final methods) due to static binding.

Comparison With C++
1) In Java, inner Class is allowed to access private data members of outer class. This behavior is same as C++ (See this).
2) In Java, methods declared as private can never be overridden, they are in-fact bounded during compile time. This behavior is different from C++. In C++, we can have virtual private methods (See this).

What happens when more restrictive access is given to a derived class method in Java?

In Java, it is compiler error to give more restrictive access to a derived class function which overrides a base class function. For example, if there is a function *public void foo()* in base class and if it is overridden in derived class, then access specifier for *foo()* cannot be anything other than public in derived class. If *foo()* is private function in base class, then access specifier for it can be anything in derived class.

Consider the following two programs. Program 1 fails in compilation and program 2 works fine.

Program 1

```
// file name: Main.java
class Base {
    public void foo() {}
}

class Derived extends Base {
    private void foo() {} // compiler error
}

public class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
    }
}
```

Program 2

```
// file name: Main.java
class Base {
    private void foo() {}
}

class Derived extends Base {
    public void foo() {} // works fine
}

public class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
    }
}
```

Interfaces in Java

Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.
- A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

```
// A simple interface
interface Player
{
    final int id = 10;
    int move();
}
```

To implement an interface we use keyword: implement

```
// Java program to demonstrate working of
// interface.
import java.io.*;

// A simple interface
interface in1
{
    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}

// A class that implements interface.
class testClass implements in1
{
    // Implementing the capabilities of
    // interface.
    public void display()
    {
```

```
        System.out.println("Geek");
    }

    // Driver Code
    public static void main (String[] args)
    {
        testClass t = new testClass();
        t.display();
        System.out.println(a);
    }
}
```

Output:

```
Geek
10
```

New features added in interfaces in JDK 8

1. Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.

   Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

   ```
   // An example to show that interfaces can
   // have methods from JDK 1.8 onwards
   interface in1
   {
       final int a = 10;
       default void display()
       {
           System.out.println("hello");
       }
   }

   // A class that implements interface.
   class testClass implements in1
   {
       // Driver Code
       public static void main (String[] args)
       {
           testClass t = new testClass();
           t.display();
       }
   }
   ```

   Output :

```
hello
```

2. Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

```
// An example to show that interfaces can
// have methods from JDK 1.8 onwards
interface in1
{
    final int a = 10;
    static void display()
    {
        System.out.println("hello");
    }
}

// A class that implements interface.
class testClass implements in1
{
    // Driver Code
    public static void main (String[] args)
    {
        in1.display();
    }
}
```

3. Output :

```
4. hello
```

Access specifier of methods in interfaces

In Java, all methods in an interface are *public* even if we do not specify *public* with method names. Also, data fields are *public static final* even if we do not mention it with fields names. Therefore, data fields must be initialized.

Consider the following example, *x* is by default *public static final* and *foo()* is *public* even if there are no specifiers.

```
interface Test {
  int x = 10;  // x is public static final and must be initialized here
  void foo();  // foo() is public
}
```

Access specifiers for classes or interfaces in Java

In Java, methods and data members of a class/interface can have one of the following four access specifiers. The access specifiers are listed according to their restrictiveness order.

1) private
2) default (when no access specifier is specified)
3) protected
4) public

But, the classes and interfaces themselves can have only two access specifiers when declared outside any other class.
1) public
2) default (when no access specifier is specified)

We cannot declare class/interface with private or protected access specifiers. For example, following program fails in compilation.

```
//filename: Main.java
protected class Test {}

public class Main {
  public static void main(String args[]) {

  }
}
```

Note : Nested interfaces and classes can have all access specifiers.

Abstract Classes in Java

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract.

```
// An example abstract class in Java
abstract class Shape {
    int color;

    // An abstract function (like a pure virtual function in C++)
    abstract void draw();
}
```

Following are some important observations about abstract classes in Java.

1) Like C++, in Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.

```
abstract class Base {
```

```
    abstract void fun();
}
class Derived extends Base {
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {

        // Uncommenting the following line will cause compiler error as the
        // line tries to create an instance of abstract class.
        // Base b = new Base();

        // We can have references of Base type.
        Base b = new Derived();
        b.fun();
    }
}
```

Output:

```
Derived fun() called
```

2) Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created. For example, the following is a valid Java program.

```
// An abstract class with constructor
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
       Derived d = new Derived();
    }
}
```

Output:

```
Base Constructor Called
Derived Constructor Called
```

3) In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
    void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base { }

class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.fun();
    }
}
```

Output:

```
Base fun() called
```

4) Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

```
// An abstract class with a final method
abstract class Base {
    final void fun() { System.out.println("Derived fun() called"); }
}

class Derived extends Base {}

class Main {
    public static void main(String args[]) {
       Base b = new Derived();
       b.fun();
    }
}
```

Output:

```
Derived fun() called
```

Exercise:
1. Is it possible to create abstract and final class in Java?
2. Is it possible to have an abstract method in a final class?
3. Is it possible to inherit from multiple abstract classes in Java?

Comparator Interface in Java

Comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of two different classes. Following function compare obj1 with obj2

Syntax:

```
public int compare(Object obj1, Object obj2):
```

Suppose we have an array/arraylist of our own class type, containing fields like rollno, name, address, DOB etc and we need to sort the array based on Roll no or name?

Method 1: One obvious approach is to write our own sort() function using one of the standard algorithms. This solution requires rewriting the whole sorting code for different criterion like Roll No. and Name.

Method 2: Using comparator interface- Comparator interface is used to order the objects of user-defined class. This interface is present java.util package and contains 2 methods compare(Object obj1, Object obj2) and equals(Object element). Using comparator, we can sort the elements based on data members. For instance it may be on rollno, name, age or anything else.

Method of Collections class for sorting List elements is used to sort the elements of List by the given comparator.

```
// To sort a given list. ComparatorClass must implement
// Comparator interface.
public void sort(List list, ComparatorClass c)
```

How does Collections.Sort() work?
Internally the Sort method does call Compare method of the classes it is sorting. To compare two elements, it asks "Which is greater?" Compare method returns -1, 0 or 1 to say if it is less than, equal, or greater to the other. It uses this result to then determine if they should be swapped for its sort.

Working Program:

```
// Java program to demonstrate working of Comparator
// interface
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a student.
class Student
{
    int rollno;
    String name, address;

    // Constructor
    public Student(int rollno, String name,
```

```
                                String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +
                            " " + this.address;
    }
}

class Sortbyroll implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}

class Sortbyname implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll name
    public int compare(Student a, Student b)
    {
        return a.name.compareTo(b.name);
    }
}

// Driver class
class Main
{
    public static void main (String[] args)
    {
        ArrayList<Student> ar = new ArrayList<Student>();
        ar.add(new Student(111, "bbbb", "london"));
        ar.add(new Student(131, "aaaa", "nyc"));
        ar.add(new Student(121, "cccc", "jaipur"));

        System.out.println("Unsorted");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyroll());

        System.out.println("\nSorted by rollno");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));
```

```
        Collections.sort(ar, new Sortbyname());

        System.out.println("\nSorted by name");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));
    }
}
```

Output:

```
Unsorted
111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollno
111 bbbb london
121 cccc jaipur
131 aaaa nyc

Sorted by name
131 aaaa nyc
111 bbbb london
121 cccc jaipu
```

By changing the return value in inside compare method you can sort in any order you want. eg.for descending order just change the positions of a and b in above compare method.

(Java Interface methods)

There is a rule that every member of interface is only and only public whether you define or not. So when we define the method of the interface in a class implementing the interface, we have to give it public access as child class can't assign the weaker access to the methods.

```
// A Simple Java program to demonstrate that
// interface methods must be public in
// implementing class
interface A
{
    void fun();
}

class B implements A
{
    // If we change public to anything else,
    // we get compiler error
    public void fun()
    {
        System.out.println("fun()");
    }
```

```
}

class C
{
    public static void main(String[] args)
    {
        B b = new B();
        b.fun();
    }
}
```

Output:

```
fun()
```

If we change fun() to anything other than public in class B, we get compiler error “attempting to assign weaker access privileges; was public”

Nested Interface in Java

We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface.

Interface in a class

Interfaces (or classes) can have only public and default access specifiers when declared outside any other class (Refer this for details). This interface declared in a class can either be default, public, private, protected. While implementing the interface, we mention the interface as **c_name.i_name** where **c_name** is the name of the class in which it is nested and **i_name** is the name of the interface itself.
Let us have a look at the following code:-

```
// Java program to demonstrate working of
// interface inside a class.
import java.util.*;
class Test
{
    interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
```

```
}

class A
{
    public static void main(String[] args)
    {
        Test.Yes obj;
        Testing t = new Testing();
        obj=t;
        obj.show();
    }
}
show method of interface
```

The access specifier in above example is default. We can assign public, protected or private also. Below is an example of protected. In this particular example, if we change access specifier to private, we get compiler error because a derived class tries to access it.

```
// Java program to demonstrate protected
// specifier for nested interface.
import java.util.*;
class Test
{
    protected interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
}

class A
{
    public static void main(String[] args)
    {
        Test.Yes obj;
        Testing t = new Testing();
        obj=t;
        obj.show();
    }
}
show method of interface
```

Interface in another Interface

An interface can be declared inside another interface also. We mention the interface as

i_name1.i_name2 where **i_name1** is the name of the interface in which it is nested and **i_name2** is the name of the interface to be implemented.

```
// Java program to demonstrate working of
// interface inside another interface.
import java.util.*;
interface Test
{
   interface Yes
   {
      void show();
   }
}

class Testing implements Test.Yes
{
   public void show()
   {
      System.out.println("show method of interface");
   }
}

class A
{
   public static void main(String[] args)
   {
     Test.Yes obj;
     Testing t = new Testing();
     obj = t;
     obj.show();
   }
}
show method of interface
```

Note: In the above example, access specifier is public even if we have not written public. If we try to change access specifier of interface to anything other than public, we get compiler error. Remember, interface members can only be public..

```
// Java program to demonstrate an interface cannot
// have non-public member interface.
import java.util.*;
interface Test
{
    protected interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
```

```
}

class A
{
    public static void main(String[] args)
    {
        Test.Yes obj;
        Testing t = new Testing();
        obj = t;
        obj.show();
    }
}
illegal combination of modifiers: public and protected
   protected interface Yes
```

Comparable vs Comparator in Java

Java provides two interfaces to sort objects using data members of the class:

1. Comparable
2. Comparator

Using Comparable Interface

A comparable object is capable of comparing itself with another object. The class itself must implements the **java.lang.Comparable** interface to compare its instances.

Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method compareTo() of Comparable interface.

```
// A Java program to demonstrate use of Comparable
import java.io.*;
import java.util.*;

// A class 'Movie' that implements Comparable
class Movie implements Comparable<Movie>
{
    private double rating;
    private String name;
    private int year;

    // Used to sort movies by year
    public int compareTo(Movie m)
    {
        return this.year - m.year;
    }

    // Constructor
    public Movie(String nm, double rt, int yr)
```

```
    {
        this.name = nm;
        this.rating = rt;
        this.year = yr;
    }

    // Getter methods for accessing private data
    public double getRating() { return rating; }
    public String getName()   {  return name; }
    public int getYear()      {  return year;  }
}

// Driver class
class Main
{
    public static void main(String[] args)
    {
        ArrayList<Movie> list = new ArrayList<Movie>();
        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));
        list.add(new Movie("Return of the Jedi", 8.4, 1983));

        Collections.sort(list);

        System.out.println("Movies after sorting : ");
        for (Movie movie: list)
        {
            System.out.println(movie.getName() + " " +
                               movie.getRating() + " " +
                               movie.getYear());
        }
    }
}
```

Output:

```
Movies after sorting :
Star Wars 8.7 1977
Empire Strikes Back 8.8 1980
Return of the Jedi 8.4 1983
Force Awakens 8.3 2015
```

Now, suppose we want sort movies by their rating and names also. When we make a collection element comparable(by having it implement Comparable), we get only one chance to implement the compareTo() method. The solution is using Comparator.

Using Comparator

Unlike Comparable, Comparator is external to the element type we are comparing. It's a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.

Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.

To compare movies by Rating, we need to do 3 things :

1. Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).
2. Make an instance of the Comparator class.
3. Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator.

```
//A Java program to demonstrate Comparator interface
import java.io.*;
import java.util.*;

// A class 'Movie' that implements Comparable
class Movie implements Comparable<Movie>
{
    private double rating;
    private String name;
    private int year;

    // Used to sort movies by year
    public int compareTo(Movie m)
    {
        return this.year - m.year;
    }

    // Constructor
    public Movie(String nm, double rt, int yr)
    {
        this.name = nm;
        this.rating = rt;
        this.year = yr;
    }

    // Getter methods for accessing private data
    public double getRating() { return rating; }
    public String getName()   {  return name; }
    public int getYear()      {  return year;  }
}

// Class to compare Movies by ratings
class RatingCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        if (m1.getRating() < m2.getRating()) return -1;
```

```
        if (m1.getRating() > m2.getRating()) return 1;
        else return 0;
    }
}

// Class to compare Movies by name
class NameCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        return m1.getName().compareTo(m2.getName());
    }
}

// Driver class
class Main
{
    public static void main(String[] args)
    {
        ArrayList<Movie> list = new ArrayList<Movie>();
        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));
        list.add(new Movie("Return of the Jedi", 8.4, 1983));

        // Sort by rating : (1) Create an object of ratingCompare
        //                  (2) Call Collections.sort
        //                  (3) Print Sorted list
        System.out.println("Sorted by rating");
        RatingCompare ratingCompare = new RatingCompare();
        Collections.sort(list, ratingCompare);
        for (Movie movie: list)
            System.out.println(movie.getRating() + " " +
                               movie.getName() + " " +
                               movie.getYear());


        // Call overloaded sort method with RatingCompare
        // (Same three steps as above)
        System.out.println("\nSorted by name");
        NameCompare nameCompare = new NameCompare();
        Collections.sort(list, nameCompare);
        for (Movie movie: list)
            System.out.println(movie.getName() + " " +
                               movie.getRating() + " " +
                               movie.getYear());

        // Uses Comparable to sort by year
        System.out.println("\nSorted by year");
        Collections.sort(list);
        for (Movie movie: list)
            System.out.println(movie.getYear() + " " +
                               movie.getRating() + " " +
                               movie.getName()+" ");
    }
```

```
}
```

Output :

```
Sorted by rating
8.3 Force Awakens 2015
8.4 Return of the Jedi 1983
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980

Sorted by name
Empire Strikes Back 8.8 1980
Force Awakens 8.3 2015
Return of the Jedi 8.4 1983
Star Wars 8.7 1977

Sorted by year
1977 8.7 Star Wars
1980 8.8 Empire Strikes Back
1983 8.4 Return of the Jedi
2015 8.3 Force Awakens
```

- Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.
- Logically, Comparable interface compares “this” reference with the object specified and Comparator in Java compares two different class objects provided.
- If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on there natural order defined by CompareTo method.

To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.

Multithreading in Java

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :
1. Extending the Thread class
2. Implementing the Runnable Interface

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                  Thread.currentThread().getId() +
                  " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
public class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            MultithreadingDemo object = new MultithreadingDemo();
```

```
            object.start();
        }
    }
}
```

Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

Thread creation by implementing the Runnable Interface

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                                Thread.currentThread().getId() +
                                " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            Thread object = new Thread(new MultithreadingDemo());
            object.start();
```

```
        }
    }
}
```

Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

Thread Class vs Runnable Interface

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

Synchronized in Java

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Following is the general form of a synchronized block:

```
// Only one thread can execute at a time.
// sync_object is a reference to an object
// whose lock associates with the monitor.
// The code is said to be synchronized on
```

```
// the monitor object
synchronized(sync_object)
{
   // Access shared variables and other
   // shared resources
}
```

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Following is an example of multi threading with synchronized.

```
// A Java program to demonstrate working of
// synchronized.
import java.io.*;
import java.util.*;

// A Class used to send a message
class Sender
{
    public void send(String msg)
    {
        System.out.println("Sending\t" + msg );
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
            System.out.println("Thread  interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}

// Class for send a message using Threads
class ThreadedSend extends Thread
{
    private String msg;
    private Thread t;
    Sender  sender;

    // Recieves a message object and a string
    // message to be sent
    ThreadedSend(String m,  Sender obj)
    {
        msg = m;
        sender = obj;
    }

    public void run()
    {
```

```
        // Only one thread can send a message
        // at a time.
        synchronized(sender)
        {
            // synchronizing the snd object
            sender.send(msg);
        }
    }
}

// Driver class
class SyncDemo
{
    public static void main(String args[])
    {
        Sender snd = new Sender();
        ThreadedSend S1 =
            new ThreadedSend( " Hi " , snd );
        ThreadedSend S2 =
            new ThreadedSend( " Bye " , snd );

        // Start two threads of ThreadedSend type
        S1.start();
        S2.start();

        // wait for threads to end
        try
        {
            S1.join();
            S2.join();
        }
        catch(Exception e)
        {
            System.out.println("Interrupted");
        }
    }
}
```

Output:

```
Sending  Hi

 Hi Sent
Sending  Bye

 Bye Sent
```

The output is same every-time we run the program.

In the above example, we chose to synchronize the Sender object inside the run() method of the ThreadedSend class. Alternately, we could define the **whole send() block as synchronized** and it would produce the same result. Then we don't have to synchronize the Message object inside the run() method in ThreadedSend class.

```
// An alternate implementation to demonstrate
// that we can use synchronized with method also.
class Sender
{
    public synchronized void send(String msg)
    {
        System.out.println("Sending\t" + msg );
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
            System.out.println("Thread interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}
```

We do not always have to synchronize a whole method. Sometimes it is preferable to **synchronize only part of a method**. Java synchronized blocks inside methods makes this possible.

```
// One more alternate implementation to demonstrate
// that synchronized can be used with only a part of
// method
class Sender
{
    public void send(String msg)
    {
        synchronized(this)
        {
            System.out.println("Sending\t" + msg );
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e)
            {
                System.out.println("Thread interrupted.");
            }
            System.out.println("\n" + msg + "Sent");
        }
    }
}
```

Does Java support goto?

Unlike C/C++, Java does not have *goto* statement, but java supports label. The only place where a label is useful in Java is right before nested loop statements. We can specify label name with break to break out a specific outer loop. Similarly, label name can be specified with continue.

See following program for example.

```
// file name: Main.java
public class Main {
  public static void main(String[] args) {
    outer: //label for outer loop
    for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10; j++) {
        if (j == 1)
          break outer;
        System.out.println(" value of j = " + j);
      }
    } //end of outer loop
  } // end of main()
} //end of class Main
```

Output:
value of j = 0

Inner class in java

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

1) Nested Inner class
2) Method Local inner classes
3) Anonymous inner classes
4) Static nested classes

Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier. Like class, interface can also be nested and can have access specifiers.

Following example demonstrates a nested class.

```
class Outer {
   // Simple nested inner class
   class Inner {
      public void show() {
```

```
            System.out.println("In a nested class method");
        }
    }
}
class Main {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.show();
    }
}
```

Output:

```
In a nested class method
```

As a side note, we can't have static method in a nested inner class because an inner class is implicitly associated with an object of its outer class so it cannot define any static method for itself. For example the following program doesn't compile.

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");
    }
    class Inner {
        public static void main(String[] args){
            System.out.println("inside inner class Method");
        }
    }
}
```

Output:

```
Error illegal static declaration in inner class
Outer.Inner public static void main(String[] args)
modifier 'static' is only allowed in constant
variable declaration
```

An interface can also be nested and nested interfaces have some interesting properties. We will be covering nested interfaces in the next post.

Method Local inner classes

Inner class can be declared within a method of an outer class. In the following example, Inner is an inner class in outerMethod().

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");
        // Inner class is local to outerMethod()
        class Inner {
            void innerMethod() {
```

```
                System.out.println("inside innerMethod");
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}
class MethodDemo {
    public static void main(String[] args) {
        Outer x = new Outer();
        x.outerMethod();
    }
}
```

Output

```
Inside outerMethod
Inside innerMethod
```

Method Local inner classes can't use local variable of outer method until that local variable is not declared as final. For example, the following code generates compiler error (Note that x is not final in outerMethod() and innerMethod() tries to access it)

```
class Outer {
   void outerMethod() {
      int x = 98;
      System.out.println("inside outerMethod");
      class Inner {
         void innerMethod() {
            System.out.println("x= "+x);
         }
      }
      Inner y = new Inner();
      y.innerMethod();
   }
}
class MethodLocalVariableDemo {
   public static void main(String[] args) {
      Outer x=new Outer();
      x.outerMethod();
   }
}
```

Output:

```
local variable x is accessed from within inner class;
needs to be declared final
```

But the following code compiles and runs fine (Note that x is final this time)

```
class Outer {
   void outerMethod() {
      final int x=98;
      System.out.println("inside outerMethod");
```

```
      class Inner {
         void innerMethod() {
            System.out.println("x = "+x);
         }
      }
      Inner y = new Inner();
      y.innerMethod();
   }
}
class MethodLocalVariableDemo {
    public static void main(String[] args){
      Outer x = new Outer();
      x.outerMethod();
    }
}
```

Output-:

```
Inside outerMethod
X = 98
```

The main reason we need to declare a local variable as a final is that local variable lives on stack till method is on the stack but there might be a case the object of inner class still lives on the heap.
Method local inner class can't be marked as private, protected, static and transient but can be marked as abstract and final, but not both at the same time.

Static nested classes
Static nested classes are not technically an inner class. They are like a static member of outer class.

```
class Outer {
   private static void outerMethod() {
     System.out.println("inside outerMethod");
   }

   // A static inner class
   static class Inner {
     public static void main(String[] args) {
        System.out.println("inside inner class Method");
        outerMethod();
     }
   }

}
```

Output

```
inside inner class Method
inside outerMethod
```

Anonymous inner classes

Anonymous inner classes are declared without any name at all. They are created in two ways.

a) *As subclass of specified type*

```
class Demo {
   void show() {
      System.out.println("i am in show method of super class");
   }
}
class Flavor1Demo {

   //  An anonymous class with Demo as base class
   static Demo d = new Demo() {
       void show() {
           super.show();
           System.out.println("i am in Flavor1Demo class");
       }
   };
   public static void main(String[] args){
       d.show();
   }
}
```

Output

```
i am in show method of super class
i am in Flavor1Demo class
```

In the above code, we have two class Demo and Flavor1Demo. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous class show() method is overridden.

a) *As implementer of the specified interface*

```
class Flavor2Demo {

    // An anonymous class that implements Hello interface
    static Hello h = new Hello() {
        public void show() {
            System.out.println("i am in anonymous class");
        }
    };

    public static void main(String[] args) {
        h.show();
    }
}

interface Hello {
    void show();
```

```
}
```

Output:

```
i am in anonymous class
```

In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello. Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement interface at a time.

(The Initializer Block in Java)

Initializer block contains the code that is always executed whenever an instance is created. It is used to declare/initialize the common part of various constructors of a class. For example,

```
import java.io.*;
public class GFG
{
    // Initializer block starts..
    {
        // This code is executed before every constructor.
        System.out.println("Common part of constructors invoked !!");
    }
    // Initializer block ends

    public GFG()
    {
        System.out.println("Default Constructor invoked");
    }
    public GFG(int x)
    {
        System.out.println("Parametrized constructor invoked");
    }
    public static void main(String arr[])
    {
        GFG obj1, obj2;
        obj1 = new GFG();
        obj2 = new GFG(0);
    }
}
```

Output:

```
Common part of constructors invoked!!
Default Constructor invoked
Common part of constructors invoked!!
Parametrized constructor invoked
```

We can note that the contents of initializer block are executed whenever any constructor is invoked (before the constructor's contents)

The order of initialization constructors and initializer block doesn't matter, initializer block is always executed before constructor. See this for example.

What if we want to execute some code once for all objects of a class?
We use Static Block in Java

(String vs StringBuilder vs StringBuffer in Java)

Consider below code with three concatenation functions with three different types of parameters, String, StringBuffer and StringBuilder.

```
// Java program to demonstrate difference between String,
// StringBuilder and StringBuffer
class HelloWorld
{
    // Concatenates to String
    public static void concat1(String s1)
    {
        s1 = s1 + "World";
    }

    // Concatenates to StringBuilder
    public static void concat2(StringBuilder s2)
    {
        s2.append("World");
    }

    // Concatenates to StringBuffer
    public static void concat3(StringBuffer s3)
    {
        s3.append("World");
    }

    public static void main(String[] args)
    {
        String s1 = "Hello";
        concat1(s1);  // s1 is not changed
        System.out.println("String: " + s1);

        StringBuilder s2 = new StringBuilder("Geeks");
        concat2(s2); // s2 is changed
        System.out.println("StringBuilder: " + s2);
```

```
        StringBuffer s3 = new StringBuffer("Geeks");
        concat3(s3); // s3 is changed
        System.out.println("StringBuffer: " + s3);
    }
}
```

Output:

```
String: Hello
StringBuilder: HelloWorld
StringBuffer: HelloWorld
```

Explanation:
1. Concat1 : In this method, we pass a string “Geeks” and perform “s1 = s1 + ”forgeeks”. The string passed from main() is not changed, this is due to the fact that String is **immutable**. Altering the value of string creates another object and s1 in concat1() stores reference of new string. References s1 in main() and cocat1() refer to different strings.

2. Concat2 : In this method, we pass a string “Geeks” and perform “s2.append(“forgeeks”)” which changes the actual value of the string (in main) to “HelloWorld”. This is due to the simple fact that StringBuilder is **mutable** and hence changes its value.

2. Concat3 : StringBuffer is similar to StringBuilder except one difference that StringBuffer is thread safe, i.e., multiple threads can use it without any issue. The thread safety brings a penalty of performance.

Conclusion:

- Objects of String are immutable, and objects of StringBuffer and StringBuilder are mutable.
- StringBuffer and StringBuilder are similar, but StringBuilder is faster and preferred over StringBuffer for single threaded program. If thread safety is needed, then StringBuffer is used.

(final, finally and finalize in Java)

final: The final keyword in java has different meaning depending upon it is applied to variable, class or method.

- *Variables:* The value cannot be changed once initialized.
- *Method:* The method cannot be overridden by a subclass.
- *Class:* The class cannot be subclassed.

See final keyword in Java for Code examples of above three.

finally: The finally keyword is used in association with a try/catch block and guarantees that a section of code will be executed, even if an exception is thrown. The finally block will be executed after the try and catch blocks, but before control transfers back to its origin.

```
// A Java program to demonstrate finally.
class Geek
{
    // A method that throws an exception and has finally.
    // This method will be called inside try-catch.
    static void A()
    {
        try
        {
            System.out.println("inside A");
            throw new RuntimeException("demo");
        }
        finally
        {
            System.out.println("A's finally");
        }
    }

    // This method also calls finally. This method
    // will be called outside try-catch.
    static void B()
    {
        try
        {
            System.out.println("inside B");
            return;
        }
        finally
        {
            System.out.println("B's finally");
        }
    }

    public static void main(String args[])
    {
        try
        {
            A();
        }
        catch (Exception e)
        {
            System.out.println("Exception caught");
        }
        B();
    }
}
```

Output:

```
inside A
```

```
A's finally
Exception caught
inside B
B's finally
```

finalize: The automatic garbage collector call the finalize() method just before actually destroying the object. A class can therefore override the finalize() method from the Object class in order to define custom behavior during garbage collection.Syntax:

```
protected void finalize() throws Throwable
{
    /* close open files, release resources, etc */
}
```

Note that there is no guarantee about the time when finalize is called. It may be called any time after the object is not being referred anywhere (cab be garbage collected).

BigInteger Class in Java

BigInteger class is used for mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types.

For example factorial of 100 contains 158 digits in it so we can't store it in any primitive data type available. We can store as large Integer as we want in it. There is no theoretical limit on the upper bound of the range because memory is allocated dynamically but practically as memory is limited you can store a number which has Integer.MAX_VALUE number of bits in it which should be sufficient to store mostly all large values.

Below is an example Java program that uses BigInteger to compute Factorial.

```
// Java program to find large factorials using BigInteger
import java.math.BigInteger;
import java.util.Scanner;

public class Example
{
    // Returns Factorial of N
    static BigInteger factorial(int N)
    {
        // Initialize result
        BigInteger f = new BigInteger("1"); // Or BigInteger.ONE

        // Multiply f with 2, 3, ...N
        for (int i = 2; i <= N; i++)
            f = f.multiply(BigInteger.valueOf(i));

        return f;
```

```
    }

    // Driver method
    public static void main(String args[]) throws Exception
    {
        int N = 20;
        System.out.println(factorial(N));
    }
}
```

Output:

```
2432902008176640000
```

If we have to write above program in C++, that would be too large and complex, we can look at Factorail of Large Number.
In this way BigInteger class is very handy to use because of its large method library and it is also used a lot in competitive programming.

Now below is given a list of simple statements in primitive arithmetic and its analogous statement in terms of BigInteger objects.

Declaration

```
int a, b;
BigInteger A, B;
```

Initialization:

```
a = 54;
b = 23;
A  = BigInteger.valueOf(54);
B  = BigInteger.valueOf(37);
```

And for Integers available as string you can initialize them as:

```
A  = new BigInteger("54");
B  = new BigInteger("123456789123456789");
```

Some constant are also defined in BigInteger class for ease of initialization :

```
A = BigInteger.ONE;
// Other than this, available constant are BigInteger.ZERO
// and BigInteger.TEN
```

Mathematical operations:

```
int c = a + b;
BigInteger C = A.add(B);
```

Other similar function are subtract() , multiply(), divide(), remainder()
But all these function take BigInteger as their argument so if we want these operation with integers or string convert them to BigInteger before passing them to functions as shown below:

```
String str = "123456789";
BigInteger C = A.add(new BigInteger(str));
int val  = 123456789;
BigInteger C = A.add(BigIntger.valueOf(val));
```

Extraction of value from BigInteger:

```
int x   =  A.intValue();   // value should be in limit of int x
long y   = A.longValue();  // value should be in limit of long y
String z = A.toString();
```

Comparison:

```
if (a < b) {}           // For primitive int
if (A.compareTo(B) < 0)  {} // For BigInteger
```

Actually compareTo returns -1(less than), 0(Equal), 1(greater than) according to values.

For equality we can also use:

```
if (A.equals(B)) {}  // A is equal to B
```

BigInteger class also provides quick methods for prime numbers.

SPOJ Problems:
So after above knowledge of function of BigInteger class, we can solve many complex problem easily, but remember as BigInteger class internally uses array of integers for processing, the operation on object of BigIntegers are not as fast as on primitives that is add function on BigIntgers doesn't take constant time it takes time proportional to length of BigInteger, so complexity of program will change accordingly.

'this' reference in Java

'this' is a reference variable that refers to the current object.

Following are the ways to use 'this' keyword in java :

1. Using 'this' keyword to refer current class instance variables

```
//Java code for using 'this' keyword to
//refer current class instance variables
class Test
{
    int a;
    int b;

    // Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    void display()
    {
        //Displaying value of variables a and b
        System.out.println("a = " + a + "  b = " + b);
    }

    public static void main(String[] args)
    {
        Test object = new Test(10, 20);
        object.display();
    }
}
```

Output:

```
a = 10  b = 20
```

2. Using this() to invoke current class constructor

```
// Java code for using this() to
// invoke current class constructor
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        this(10, 20);
        System.out.println("Inside  default constructor \n");
    }

    //Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
        System.out.println("Inside parameterized constructor");
```

```
    }

    public static void main(String[] args)
    {
        Test object = new Test();
    }
}
```

Output:

```
Inside parameterized constructor
Inside  default constructor
```

3. Using ‘this’ keyword to return the current class instance

```
//Java code for using 'this' keyword
//to return the current class instance
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

    //Method that returns current class instance
    Test get()
    {
        return this;
    }

    //Displaying value of variables a and b
    void display()
    {
        System.out.println("a = " + a + "  b = " + b);
    }

    public static void main(String[] args)
    {
        Test object = new Test();
        object.get().display();
    }
}
```

Output:

```
a = 10  b = 20
```

4. Using ‘this’ keyword as method parameter

```
// Java code for using 'this'
// keyword as method parameter
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

    //Method that receives 'this' keyword as parameter
    void display(Test obj)
    {
        System.out.println("a = " + a + "  b = " + b);
    }

    //Method that returns current class instance
    void get()
    {
        display(this);
    }

    public static void main(String[] args)
    {
        Test object = new Test();
        object.get();
    }
}
```

Output:

```
a = 10  b = 20
```

Serialization in Java

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the **java.io.Serializable** interface.

The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

```
private void writeObject(ObjectOutputStream stream)
                        throws IOException;
```

The ObjectInputStream class contains **readObject()** method for deserializing an object.

```
private void readObject(ObjectInputStream stream)
                        throws IOException, ClassNotFoundException;
```

```
// Java code for serialization and deserialization of a Java object

import java.io.*;

class Demo implements java.io.Serializable
{
    public int a;
    public String b;

    // Default constructor
    public Demo(int a, String b)
    {
        this.a = a;
        this.b = b;
    }

}

class Test
{
    public static void main(String[] args)
    {
        Demo object = new Demo(1, "HelloWorld");
        String filename = "file.ser";

        // Serialization
        try
        {
            //Saving of object in a file
            FileOutputStream file = new FileOutputStream(filename);
            ObjectOutputStream out = new ObjectOutputStream(file);

            // Method for serialization of object
            out.writeObject(object);

            out.close();
            file.close();
```

```
            System.out.println("Object has been serialized");

        }

        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }


        Demo object1 = null;

        // Deserialization
        try
        {
            // Reading the object from a file
            FileInputStream file = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(file);

            // Method for deserialization of object
            object1 = (Demo)in.readObject();

            in.close();
            file.close();

            System.out.println("Object has been deserialized ");
            System.out.println("a = " + object1.a);
            System.out.println("b = " + object1.b);
        }

        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }

        catch(ClassNotFoundException ex)
        {
            System.out.println("ClassNotFoundException is caught");
        }

    }
}
```

Output :

```
Object has been serialized
Object has been deserialized
a = 1
b = HelloWorld
```

Few Tricky Programs in Java

1. **Comments that execute :**

   Till now, we were always taught “Comments do not Execute”. Let us see today “The comments that execute”

   Following is the code snippet:

```
public class Testing {
    public static void main(String[] args)
     {
         // the line below this gives an output
         // \u000d System.out.println("comment executed");
     }
}
```

Output:

```
comment executed
```

The reason for this is that the Java compiler parses the unicode character \u000d as a new line and gets transformed into:

```
public class Testing {
    public static void main(String[] args)
    {
        // the line below this gives an output
        // \u000d
        System.out.println("comment executed");
    }
}
```

- **Named loops :**

```
// A Java program to demonstrate working of named loops.
public class Testing
{
    public static void main(String[] args)
    {
    loop1:
    for (int i = 0; i < 5; i++)
     {
        for (int j = 0; j < 5; j++)
        {
            if (i == 3)
                break loop1;
            System.out.println("i = " + i + " j = " + j);
        }
    }
   }
}
```

Output:

```
i = 0 j = 0
i = 0 j = 1
i = 0 j = 2
i = 0 j = 3
i = 0 j = 4
i = 1 j = 0
i = 1 j = 1
i = 1 j = 2
i = 1 j = 3
i = 1 j = 4
i = 2 j = 0
i = 2 j = 1
i = 2 j = 2
i = 2 j = 3
i = 2 j = 4
```

You can also use continue to jump to start of the named loop.

We can also use break (or continue) in a nested if-else with for loops in order to break several loops with if-else, so one can avoid setting lot of flags and testing them in the if-else in order to continue or not in this nested level.

(Foreach in C++ and Java)

Foreach loop is used to access elements of an array quickly without performing initialization, testing and increment/decrement. The working of foreach loops is to do something for every element rather than doing something n times.

There is no foreach loop in C, but both C++ and Jave support foreach type of loop. In C++, it was introduced in C++ 11 and Java in JDK 1.5.0

The keyword used for foreach loop is "**for**" in both C++ and Java.

C++ Program:

```
// C++ program to demonstrate use of foreach
#include <iostream>
using namespace std;

int main()
{
  int arr[] = {10, 20, 30, 40};

  // Printing elements of an array using
  // foreach loop
  for (int x : arr)
```

```
     cout << x << endl;
}
```

Output:

```
10
20
30
40
```

Java program

```
// Java program to demonstrate use of foreach
public class Main
{
    public static void main(String[] args)
    {
       // Declaring 1-D array with size 4
       int arr[] = {10, 20, 30, 40};

       // Printing elements of an array using
       // foreach loop
       for (int x : arr)
          System.out.println(x);
   }
}
```

Output:

```
10
20
30
40
```

Advantages of Foreach loop :-
1) Makes code more readable.
2) Eliminates the possibility of programming errors.

Flexible nature of java.lang.Object

We all love the mechanism of python, where we don't have to bother about data types of the variables (don't we!)

Interestingly we have one class in Java too, which is pretty similar !

Yes, you guessed it right! It's java.lang.Object

For example,

```
// A Java program to demonstrate flexible nature of
// java.lang.Object
public class GFG
{
    public static void main(String arr[])
    {
        Object y;

        y = 'A';
        System.out.println(y.getClass().getName());

        y = 1;
        System.out.println(y.getClass().getName());

        y = "Hi";
        System.out.println(y.getClass().getName());

        y = 1.222;
        System.out.println(y.getClass().getName());

        y = false;
        System.out.println(y.getClass().getName());
    }
}
```

Output:

```
java.lang.Character
java.lang.Integer
java.lang.String
java.lang.Double
java.lang.Boolean
```

Such a behaviour can be attributed to the fact that java.lang.Object is super class to all other classes. Hence, a reference variable of type Object can be practically used to refer objects of any class. So, we could also assign y = new InputStreamReader(System.in) in the above code!

How to swap or exchange objects in Java?

How to swap objects in Java?
Let's say we have a class called "Car" with some attributes. And we create two objects of Car, say car1 and car2, how to exchange the data of car1 and car2?

A Simple Solution is to swap members. For example, if the class Car has only one integer attribute say "no" (car number), we can swap cars by simply swapping the members of two cars.

```
// A Java program to demonstrate that we can swap two
// objects be swapping members.
```

```
// A car with number class Car
class Car
{
    int no;
    Car(int no) { this.no = no; }
}

// A class that uses Car objects
class Main
{
    // To swap c1 and c2
    public static void swap(Car c1, Car c2)
    {
        int temp = c1.no;
        c1.no = c2.no;
        c2.no = temp;
    }

    // Driver method
    public static void main(String[] args)
    {
        Car c1 = new Car(1);
        Car c2 = new Car(2);
        swap(c1, c2);
        System.out.println("c1.no = " + c1.no);
        System.out.println("c2.no = " + c2.no);
    }
}
```

Output:

```
c1.no = 2
c2.no = 1
```

What if we don't know members of Car?

The above solution worked as we knew that there is one member "no" in Car. What if we don't know members of Car or the member list is too big. T*his is a very common situation as a class that uses some other class may not access members of other class*. Does below solution work?

```
// A Java program to demonstrate that simply swapping
// object references doesn't work

// A car with number and name
class Car
{
    int model, no;

    // Constructor
    Car(int model, int no)
    {
        this.model = model;
        this.no = no;
    }
```

```
    // Utility method to print Car
    void print()
    {
        System.out.println("no = " + no +
                           ", model = " + model);
    }
}

// A class that uses Car
class Main
{
    // swap() doesn't swap c1 and c2
    public static void swap(Car c1, Car c2)
    {
        Car temp = c1;
        c1 = c2;
        c2 = temp;
    }

    // Driver method
    public static void main(String[] args)
    {
        Car c1 = new Car(101, 1);
        Car c2 = new Car(202, 2);
        swap(c1, c2);
        c1.print();
        c2.print();
    }
}
```

Output:

```
no = 1, model = 101
no = 2, model = 202
```

As we can see from above output, the objects are not swapped. We have discussed in a previous post that parameters are passed by value in Java. So when we pass c1 and c2 to swap(), the function swap() creates a copy of these references.

Solution is to use Wrapper Class If we create a wrapper class that contains references of Car, we can swap cars by swapping references of wrapper class.

```
// A Java program to demonstrate that we can use wrapper
// classes to swap to objects

// A car with model and no.
class Car
{
    int model, no;

    // Constructor
    Car(int model, int no)
    {
```

```
        this.model = model;
        this.no = no;
    }

    // Utility method to print object details
    void print()
    {
        System.out.println("no = " + no +
                           ", model = " + model);
    }
}

// A Wrapper over class that is used for swapping
class CarWrapper
{
   Car c;

   // Constructor
   CarWrapper(Car c)   {this.c = c;}
}

// A Class that use Car and swaps objects of Car
// using CarWrapper
class Main
{
    // This method swaps car objects in wrappers
    // cw1 and cw2
    public static void swap(CarWrapper cw1,
                            CarWrapper cw2)
    {
        Car temp = cw1.c;
        cw1.c = cw2.c;
        cw2.c = temp;
    }

    // Driver method
    public static void main(String[] args)
    {
        Car c1 = new Car(101, 1);
        Car c2 = new Car(202, 2);
        CarWrapper cw1 = new CarWrapper(c1);
        CarWrapper cw2 = new CarWrapper(c2);
        swap(cw1, cw2);
        cw1.c.print();
        cw2.c.print();
    }
}
```

Output:

```
no = 2, model = 202
no = 1, model = 101
```

So a wrapper class solution works even if the user class doesn't have access to members of the class whose objects are to be swapped.

How to swap two variables in one line?

We have discussed different approaches to swap two integers without the temporary variable. How to swap in a single line without using library function?

Python: In Python, there is a simple and syntactically neat construct to swap variables, we just need to write "x, y = y, x".

C/C++: Below is one generally provided classical solution

```
// Swap using bitwise XOR (Wrong Solution in C/C++)
x ^= y ^= x ^= y;
```

The above solution is wrong in C/C++ as it causes undefined behaviour (compiler is free to behave in any way). The reason is, modifying a variable more than once in an expression causes undefined behaviour if there is no sequence point between the modifications.

However, we can use comma to introduce sequence points. So the modified solution is

```
// Swap using bitwise XOR (Correct Solution in C/C++)
// sequence point introduced using comma.
(x ^= y), (y ^= x), (x ^= y);
```

Java: In Java, rules for subexpression evaluations are clearly defined. The left hand operand is always evaluated before right hand operand (See this for more details). In Java, the expression "x ^= y ^= x ^= y;" doesn't produce the correct result according to Java rules. It makes x = 0. However, we can use "x = x ^ y ^ (y = x);" Note the expressions are evaluated from left to right. If x = 5 and y = 10 initially, the expression is equivalent to "x = 5 ^ 10 ^ (y = 5);". Note that we can't use this in C/C++ as in C/C++, it is not defined whether left operand or right operand is executed for any operator (See this for more details)

```
// Java program to swap two variables in single line
class GFG
{
   public static void main (String[] args)
   {
      int x = 5, y = 10;
      x = x ^ y ^ (y = x);
      System.out.println("After Swapping values of x and y are "
                         + x + " " + y);
   }
}
```

Output:

```
After Swapping values of x and y are 10 5
```

Private Constructors and Singleton Classes in Java

Let's first analyze the following question:

Can we have private constructors ?

As you can easily guess, like any method we can provide access specifier to the constructor. If it's made private, then it can only be accessed inside the class.

Do we need such 'private constructors ' ?

There are various scenarios where we can use private constructors. The major ones are

1. Internal Constructor chaining
2. Singleton class design pattern

What is a Singleton class?

As the name implies, a class is said to be singleton if it limits the number of objects of that class to one.

We can't have more than a single object for such classes.

Singleton classes are employed extensively in concepts like Networking and Database Connectivity.

Design Pattern of Singleton classes:

The constructor of singleton class would be private so there must be another way to get the instance of that class. This problem is resolved using a class member instance and a factory method to return the class member.

Below is an example in java illustrating the same:

```
// Java program to demonstrate implementation of Singleton
// pattern using private constructors.
import java.io.*;

class MySingleton
```

```
{
    static MySingleton instance = null;
    public int x = 10;

    // private constructor can't be accessed outside the class
    private MySingleton() {  }

    // Factory method to provide the users with instances
    static public MySingleton getInstance()
    {
        if (instance == null)
            instance = new MySingleton();

        return instance;
    }
}

// Driver Class
class Main
{
   public static void main(String args[])
   {
       MySingleton a = MySingleton.getInstance();
       MySingleton b = MySingleton.getInstance();
       a.x = a.x + 10;
       System.out.println("Value of a.x = " + a.x);
       System.out.println("Value of b.x = " + b.x);
   }
}
```

Output:

```
Value of a.x = 20
Value of b.x = 20
```

We changed value of a.x, value of b.x also got updated because both ‘a’ and ‘b’ refer to same object, i.e., they are objects of a singleton class.

Generics in Java

Generics in Java is similar to templates in C++. The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()
```

Note: In Parameter type we can not use primitives like
'int','char' or 'double'.

```
// A Simple Java program to show working of user defined
// Generic classes

// We use < > to specify Parameter type
class Test<T>
{
    // An object of type T is declared
    T obj;
    Test(T obj) {  this.obj = obj;  }  // constructor
    public T getObject()  { return this.obj; }
}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        // instance of Integer type
        Test <Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test <String> sObj =
                           new Test<String>("HelloWorld");
        System.out.println(sObj.getObject());
    }
}
```

Output:

```
15
HelloWorld
```

We can also pass multiple Type parameters in Generic classes.

```
// A Simple Java program to show multiple
// type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1;  // An object of type T
    U obj2;  // An object of type U
```

```
    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =
            new Test<String, Integer>("GfG", 15);

        obj.print();
    }
}
```

Output:

```
GfG
15
```

Generic Functions:

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

```
// A Simple Java program to show working of user defined
// Generic functions

class Test
{
    // A Generic method example
    static <T> void genericDisplay (T element)
    {
        System.out.println(element.getClass().getName() +
                           " = " + element);
    }

    // Driver method
    public static void main(String[] args)
```

```
    {
         // Calling generic method with Integer argument
        genericDisplay(11);

        // Calling generic method with String argument
        genericDisplay("HelloWorld");

        // Calling generic method with double argument
        genericDisplay(1.0);
    }
}
```

Output :

```
java.lang.Integer = 11
java.lang.String = HelloWorld
java.lang.Double = 1.0
```

Advantages of Generics:

Programs that uses Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use for any type we want.

   .

2. Type Safety : Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

   ```
   // A Simple Java program to demonstrate that NOT using
   // generics can cause run time exceptions
   import java.util.*;

   class Test
   {
       public static void main(String[] args)
       {
           // Creatinga an ArrayList without any type specified
           ArrayList al = new ArrayList();

           al.add("Sachin");
           al.add("Rahul");
           al.add(10); // Compiler allows this

           String s1 = (String)al.get(0);
           String s2 = (String)al.get(1);
   ```

```
            // Causes Runtime Exception
            String s3 = (String)al.get(2);
        }
    }
```

Output :

```
Exception in thread "main" java.lang.ClassCastException:
   java.lang.Integer cannot be cast to java.lang.String
        at Test.main(Test.java:19)
```

How generics solve this problem?
At the time of defining ArrayList, we can specify that this list can take only String objects.

```
// Using generics converts run time exceptions into
// compile time exception.
import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Sachin");
        al.add("Rahul");

        // Now Compiler doesn't allow this
        al.add(10);

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);
        String s3 = (String)al.get(2);
    }
}
```

- Output:

```
15: error: no suitable method found for add(int)
        al.add(10);
          ^
```

.

- Individual Type Casting is not needed: If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.

```
// We don't need to typecast individual members of ArrayList
import java.util.*;
```

```
class Test
{
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Sachin");
        al.add("Rahul");

        // Typecasting is not needed
        String s1 = al.get(0);
        String s2 = al.get(1);
    }
}
```

- .

- Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.s

Reflection in Java

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

- The required classes for reflection are provided under java.lang.reflect package.
- Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
- Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

Reflection can be used to get information about –

1. **Class** The getClass() method is used to get the name of the class to which an object belongs.
2. **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
3. **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

```
// A simple Java program to demonstrate the use of reflection
import java.lang.reflect.Method;
import java.lang.reflect.Field;
import java.lang.reflect.Constructor;

// class whose object is to be created
```

```
class Test
{
    // creating a private field
    private String s;

    // creating a public constructor
    public Test()  {  s = "HelloWorld"; }

    // Creating a public method with no arguments
    public void method1()  {
        System.out.println("The string is " + s);
    }

    // Creating a public method with int as argument
    public void method2(int n)  {
        System.out.println("The number is " + n);
    }

    // creating a private method
    private void method3() {
        System.out.println("Private method invoked");
    }
}

class Demo
{
    public static void main(String args[]) throws Exception
    {
        // Creating object whose property is to be checked
        Test obj = new Test();

        // Creating class object from the object using
        // getclass method
        Class cls = obj.getClass();
        System.out.println("The name of class is " +
                           cls.getName());

        // Getting the constructor of the class through the
        // object of the class
        Constructor constructor = cls.getConstructor();
        System.out.println("The name of constructor is " +
                           constructor.getName());

        System.out.println("The public methods of class are : ");

        // Getting methods of the class through the object
        // of the class by using getMethods
        Method[] methods = cls.getMethods();

        // Printing method names
        for (Method method:methods)
            System.out.println(method.getName());

        // creates object of desired method by providing the
```

```
        // method name and parameter class as arguments to
        // the getDeclaredMethod
        Method methodcall1 = cls.getDeclaredMethod("method2",
                                                   int.class);

        // invokes the method at runtime
        methodcall1.invoke(obj, 19);

        // creates object of the desired field by providing
        // the name of field as argument to the
        // getDeclaredField method
        Field field = cls.getDeclaredField("s");

        // allows the object to access the field irrespective
        // of the access specifier used with the field
        field.setAccessible(true);

        // takes object and the new value to be assigned
        // to the field as arguments
        field.set(obj, "JAVA");

        // Creates object of desired method by providing the
        // method name as argument to the getDeclaredMethod
        Method methodcall2 = cls.getDeclaredMethod("method1");

        // invokes the method at runtime
        methodcall2.invoke(obj);

        // Creates object of the desired method by providing
        // the name of method as argument to the
        // getDeclaredMethod method
        Method methodcall3 = cls.getDeclaredMethod("method3");

        // allows the object to access the method irrespective
        // of the access specifier used with the method
        methodcall3.setAccessible(true);

        // invokes the method at runtime
        methodcall3.invoke(obj);
    }
}
```

Output :

```
The name of class is Test
The name of constructor is Test
The public methods of class are :
method2
method1
wait
wait
wait
equals
toString
```

```
hashCode
getClass
notify
notifyAll
The number is 19
The string is JAVA
Private method invoked
```

Important observations :

1. We can invoke an method through reflection if we know its name and parameter types. We use below two methods for this purpose
   getDeclaredMethod() : To create an object of method to be invoked. The syntax for this method is
2. `Class.getDeclaredMethod(name, parametertype)`
3. `name- the name of method whose object is to be created`
   `parametertype - parameter is an array of Class objects`

   invoke() : To invoke a method of the class at runtime we use following method–

   ```
   Method.invoke(Object, parameter)
   If the method of the class doesn't accepts any
   parameter then null is passed as argument.
   ```

4. Through reflection we can **access the private variables and methods** of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.

   Class.getDeclaredField(FieldName) : Used to get the private field. Returns an object of type Field for specified field name.
   Field.setAccessible(true) : Allows to access the field irrespective of the access modifier used with the field.

   Advantages of Using Reflection:

   - **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
   - **Debugging and testing tools**: Debuggers use the property of reflection to examine private members on classes.

   Drawbacks:

- **Performance Overhead:** Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Assertions in Java

An assertion allows testing the correctness of any assumptions that have been made in the program.

Assertion is achieved using the **assert** statement in Java. While executing assertion, it is believed to be true. If it fails, JVM throws an error named **AssertionError.** It is mainly used for testing purposes during development.

The **assert** statement is used with a Boolean expression and can be written in two different ways.

First way :

```
assert expression;
```

Second way :

```
assert expression1 : expression2;
```

Example of Assertion:-

```
// Java program to demonstrate syntax of assertion
import java.util.Scanner;

class Test
{
    public static void main( String args[] )
    {
        int value = 15;
        assert value >= 20 : " Underweight";
        System.out.println("value is "+value);
    }
}
```

Output:

```
value is 15
```

After enabling assertions
Output:

```
Exception in thread "main" java.lang.AssertionError: Underweight
```

Enabling Assertions

By default, assertions are disabled. We need to run the code as given. The syntax for enabling assertion statement in Java source code is:

```
java –ea Test
```

Or

```
java –enableassertions Test
```

Here, Test is the file name.

Disabling Assertions

The syntax for disabling assertions in java are:

```
java –da Test
```

Or

```
java –disableassertions Test
```

Here, Test is the file name.

Why to use Assertions
Wherever a programmer wants to see if his/her assumptions are wrong or not.

- To make sure that an unreachable looking code is actually unreachable.
- To make sure that assumptions written in comments are right.
- ```
  if ((x & 1) == 1)
  {  }
  else // x must be even
  { assert (x % 2 == 0); }
  ```
- To make sure default switch case is not reached.
- To check object's state.
- In the beginning of the method
- After method invocation.

Assertion Vs Normal Exception Handling
Assertions are mainly used to check logically impossible situations. For example, they can be used to check the state a code expects before it starts running or state after it finishes running. Unlike normal exception/error handling, assertions are generally disabled at run-time.
  ```

Where to use Assertions

- Arguments to private methods. Private arguments are provided by developer's code only and developer may want to check his/her assumptions about arguments.
- Conditional cases.
- Conditions at the beginning of any method.

Where not to use Assertions

- Assertions should not be used to replace error messages
- Assertions should not be used to check arguments in the public methods as they may be provided by user. Error handling should be used to handle errors provided by user.
- Assertions should not be used on command line arguments.

Pair Class in Java

In C++, we have std::pair in the utility library which is of immense use if we want to keep a pair of values together. We were looking for an equivalent class for pair in Java but Pair class did not come into existence till Java 7. JavaFX 2.2 has the javafx.util.Pair class which can be used to store a pair. We need to store the values into Pair using the parameterized constructor provided by the javafx.util.Pair class.

Note : Note that the <Key, Value> pair used in HashMap/TreeMap. Here, <Key, Value> simply refers to a pair of values that are stored together.

Methods provided by the javafx.util.Pair class

- **Pair (K key, V value) :** Creates a new pair
- **boolean equals() :** It is used to compare two pair objects. It does a deep comparison, i.e., it compares on the basic of the values (<Key, Value>) which are stored in the pair objects.

  Example:

```
Pair p1 = new Pair(3,4);
Pair p2 = new Pair(3,4);
Pair p3 = new Pair(4,4);
System.out.println(p1.equals(p2) + " " + p2.equals(p3));
```

- **Output:**

true false

- **String toString() :** This method will return the String representation of the Pair.
- **K getKey() :** It returns key for the pair.
- **V getValue() :** It returns value for the pair.
- **int hashCode() :** Generate a hash code for the Pair.

Let us have a look at the following problem.
Problem Statement : We are given names of n students with their corresponding scores obtained in a quiz. We need to find the student with maximum score in the class.

Note : You need to have Java 8 installed on your machine in order to run the below program.

```
/* Java program to find a Pair which has maximum score*/
import javafx.util.Pair;
import java.util.ArrayList;

class Test
{
    /* This method returns a Pair which hasmaximum score*/
    public static Pair <String,Integer>
                  getMaximum(ArrayList < Pair <String,Integer> > l)
    {
        // Assign minimum value initially
        int max = Integer.MIN_VALUE;

        // Pair to store the maximum marks of a student with its name
        Pair <String, Integer> ans = new Pair <String, Integer> ("", 0);

        // Using for each loop to iterate array of Pair Objects
        for (Pair <String,Integer> temp : l)
        {
            // Get the score of Student
            int val = temp.getValue();

            // Check if it is greater than the previous maximum marks
            if (val > max)
            {
                max = val;  // update maximum
                ans = temp; // update the Pair
            }
        }
        return ans;
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int n = 5;//Number of Students

        //Create an Array List
```

```
        ArrayList <Pair <String,Integer> > l =
                        new ArrayList <Pair <String,Integer> > ();

        /*  Create pair of name of student  with their
            corresponding score and insert into the
            Arraylist */
        l.add(new Pair <String,Integer> ("Student A", 90));
        l.add(new Pair <String,Integer> ("Student B", 54));
        l.add(new Pair <String,Integer> ("Student C", 99));
        l.add(new Pair <String,Integer> ("Student D", 88));
        l.add(new Pair <String,Integer> ("Student E", 89));

        // get the Pair which has maximum value
        Pair <String,Integer> ans = getMaximum(l);

        System.out.println(ans.getKey() + " is top scorer " +
                           "with score of " + ans.getValue());
    }
}
```

Output :

```
Student C is top scorer with score of 99
```

Commonly Asked Java Programming Interview Questions

Why is Java called the 'Platform Independent Programming Language'?

Platform independence means that execution of your program does not dependent on type of operating system(it could be any : Linux, windows, Mac ..etc). So compile code only once and run it on any System (In C/C++, we need to compile the code for every machine on which we run it). Java is both compiler(javac) and interpreter(jvm) based lauguage. Your java source code is first compiled into byte code using javac compiler. This byte code can be easily converted to equivalent machine code using JVM. JVM(Java Virtual Machine) is available in all operating systems we install. Hence, byte code generated by javac is universal and can be converted to machine code on any operating system, this is the reason why java is platform independent.

Explain Final keyword in java?

Final keyword in java is used to restrict usage of variable, class and method.

Variable: Value of Final variable is constant, you can not change it.
Method: you can't override a Final method.
Class: you can't inherit from Final class.

When is the super keyword used?

super keyword is used to refer:

- immediate parent class constructor,
- immediate parent class variable,
- immediate parent class method.

What is the difference between StringBuffer and String?

String is an Immutable class, i.e. you can not modify its content once created. While StringBuffer is a mutable class, means you can change its content later. Whenever we alter content of String object, it creates a new string and refer to that,it does not modify the existing one. This is the reason that the performance with StringBuffer is better than with String.
Refer this for details.

Why multiple inheritance is not supported in java?

Java supports multiple inheritance but not through classes, it supports only through its interfaces. The reason for not supporting multiple inheritance is to avoid the conflict and complexity arises due to it and keep Java a Simple Object Oriented Language. If we recall this in C++, there is a special case of multiple inheritance (diamond problem) where you have a multiple inheritance with two classes which have methods in conflicts. So, Java developers decided to avoid such conflicts and didn't allow multiple inheritance through classes at all.

Can a top level class be private or protected?

Top level classes in java can't be private or protected, but inner classes in java can. The reason for not making a top level class as private is very obvious, because nobody can see a private class and thus they can not use it. Declaring a class as protected also doesn't make any sense. The only difference between default visibility and protected visibility is that we can use it in any package by inheriting it. Since in java there is no such concept of package inheritance, defining a class as protected is no different from default.

What is the difference between 'throw' and 'throws' in Java Exception Handling?

Following are the differences between two:

- throw keyword is used to throw Exception from any method or static block whereas throws is used to indicate that which Exception can possibly be thrown by this method
- If any method throws checked Exception, then caller can either handle this exception(using try catch block)or can re throw it by declaring another 'throws' clause in method declaration.
- throw clause can be used in any part of code where you feel a specific exception needs to be thrown to the calling method

E.g.
throw
throw new Exception("You have some exception")
throw new IOException("Connection failed!!")
throws
throws IOException, NullPointerException, ArithmeticException

What is finalize() method?

Unlike c++ , we don't need to destroy objects explicitly in Java. 'Garbage Collector' does that automatically for us. Garbage Collector checks if no references to an object exist, that object is assumed to be no longer required, and the memory occupied by the object can be freed. Sometimes an object can hold non-java resources such as file handle or database connection, then you want to make sure these resources are also released before object is destroyed. To perform such operation Java provide protected void finalize() in object class. You can override this method in your class and do the required tasks. Right before an object is freed, the java run time calls the finalize() method on that object. Refer this for more details.

Difference in Set and List interface?

Set and List both are child interface of Collection interface. There are following two main differences between them

- List can hold duplicate values but Set doesn't allow this.
- In List interface data is present in the order you inserted but in the case of Set insertion order is not preserved.

What will happen if you put System.exit(0) on try or catch block? Will finally block execute?

By Calling System.exit(0) in try or catch block, we can skip the finally block. System.exit(int) method can throw a SecurityException. If Sysytem.exit(0) exits the JVM without throwing that exception then finally block will not execute. But, if System.exit(0) does throw security exception then finally block will be executed.

Annotations in Java

Annotations are used to provide supplement information about a program.

- Annotations start with '**@**'.
- Annotations do not change action of a compiled program.
- Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by compiler. See below code for example.

```
/* Java program to demonstrate that annotations are
   not barely comments (This program throws compiler
   error because we have mentioned override, but not
   overridden, we haver overloaded display) */
class Base
{
     public void display()
     {
         System.out.println("Base display()");
     }
}
class Derived extends Base
{
     @Override
     public void display(int x)
     {
         System.out.println("Derived display(int )");
     }

     public static void main(String args[])
     {
         Derived obj = new Derived();
         obj.display();
     }
}
```

- Output :
- ```
  10: error: method does not override or implement
      a method from a supertype
  ```

If we remove parameter (int x) or we remove @override, the program compiles fine.

Categories of Annotations

There are 3 categories of Annotations:-

1. Marker Annotations:

The only purpose is to mark a declaration. These annotations contain no members and do not consist any data. Thus, its presence as an annotation is sufficient. Since, marker interface contains no members, simply determining whether it is present or absent is sufficient. **@Override** is an example of Marker Annotation.

```
Example: - @TestAnnotation()
```

2. Single value Annotations:

These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don't need to specify the name of the member. However in order to use this shorthand, the name of the member must be **value.**

```
Example: - @TestAnnotation("testing");
```

3. Full Annotations:

These annotations consist of multiple data members/ name, value, pairs.

```
Example:- @TestAnnotation(owner="Rahul", value="Class Geeks")
```

Predefined/ Standard Annotations

Java defines seven built-in annotations.

- Four are imported from java.lang.annotation: **@Retention**, **@Documented**, **@Target**, and **@Inherited**.
- Three are included in java.lang: **@Deprecated, @Override** and **@SuppressWarnings**

@Deprecated Annotation

- It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.
- The Javadoc @deprecated tag should be used when an element has been deprecated.
- @deprecated tag is for documentation and @Deprecated annotation is for runtime reflection.
```

- @deprecated tag has higher priority than @Deprecated annotation when both are together used.

```
public class DeprecatedTest
{
    @Deprecated
    public void Display()
    {
        System.out.println("Deprecatedtest display()");
    }

    public static void main(String args[])
    {
        DeprecatedTest d1 = new DeprecatedTest();
        d1.Display();
    }
}
```

Output:

```
Deprecatedtest display()
```

@Override Annotation
It is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result (see this for example). It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

Example:-

```
class Base
{
     public void Display()
     {
         System.out.println("Base display()");
     }

     public static void main(String args[])
     {
         Base t1 = new Derived();
         t1.Display();
     }
}
class Derived extends Base
{
     @Override
     public void Display()
     {
         System.out.println("Derived display()");
     }
}
```

Output:

```
Derived display()
```

@SuppressWarnings
It is used to inform the compiler to suppress specified compiler warnings. The warnings to suppress are specified by name, in string form. This type of annotation can be applied to any type of declaration.

Java groups warnings under two categories. They are : **deprecation** and **unchecked.**. Any unchecked warning is generated when a legacy code interfaces with a code that use generics.

```
class DeprecatedTest
{
    @Deprecated
    public void Display()
    {
        System.out.println("Deprecatedtest display()");
    }
}

public class SuppressWarningTest
{
    // If we comment below annotation, program generates
    // warning
    @SuppressWarnings({"checked", "deprecation"})
    public static void main(String args[])
    {
        DeprecatedTest d1 = new DeprecatedTest();
        d1.Display();
    }
}
```

Output:

```
Deprecatedtest display()
```

@Documented Annotations
It is a marker interface that tells a tool that an annotation is to be documented. Annotations are not included by Javadoc comments. Use of @Documented annotation in the code enables tools like Javadoc to process it and include the annotation type information in the generated document.

@Target
It is designed to be used only as an annotation to another annotation. **@Target** takes one argument, which must be constant from the **ElementType** enumeration. This argument specifies the type of declarations to which the annotation can be applied. The constants are shown below along with the type of declaration to which they correspond.

Target Constant	Annotations Can be Applied To
ANNOTATION_TYPE	Another annotation

CONSTRUCTOR	Constructor
FIELD	Field
LOCAL_VARIABLE	Local variable
METHOD	Method
PACKAGE	Package
PARAMETER	Parameter
TYPE	Class, Interface, or enumeration

We can specify one or more of these values in a **@Target**annotation. To specify multiple values, we must specify them within a braces-delimited list. For example, to specify that an annotation applies only to fields and local variables, you can use this @Target annotation: @Target({ElementType.FIELD, ElementType.LOCAL_VARIABLE}) **@Retention Annotation** It determines where and how long the annotation is retent. The 3 values that the @Retention annotation can have:

- **SOURCE:** Annotations will be retained at the source level and ignored by the compiler.
- **CLASS:** Annotations will be retained at compile time and ignored by the JVM.
- **RUNTIME:** These will be retained at runtime.

@Inherited
@Inherited is a marker annotation that can be used only on annotation declaration. It affects only annotations that will be used on class declarations. **@Inherited** causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with **@Inherited,** then that annotation will be returned.

User-defined/ Custom Annotations

User-defined annotations can be used to annotate program elements, i.e. variables, constructors, methods, etc. These annotations can be applied just before declaration of an element (constructor, method, classes, etc).

Syntax of Declaration:-

```
[Access Specifier] @interface<AnnotationName>
{
   DataType <Method Name>() [default value];
}
```

- **AnnotationName** is an identifier.

- Parameter should not be associated with method declarations and **throws** clause should not be used with method declaration.
- Parameters will not have a null value but can have a default value.
- ***default value*** is optional.
- Return type of method should be either primitive, enum, string, class name or array of primitive, enum, string or class name type.

```
package source;
// A Java program to demonstrate user defined annotations
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

// user-defined annotation
@Documented
@Retention(RetentionPolicy.RUNTIME)
@ interface TestAnnotation
{
    String Developer() default "Rahul";
    String Expirydate();
} // will be retained at runtime

// Driver class that uses @TestAnnotation
public class Test
{
    @TestAnnotation(Developer="Rahul", Expirydate="01-10-2020")
    void fun1()
    {
        System.out.println("Test method 1");
    }

    @TestAnnotation(Developer="Anil", Expirydate="01-10-2021")
    void fun2()
    {
        System.out.println("Test method 2");
    }

    public static void main(String args[])
    {
        System.out.println("Hello");
    }
}
```

Output :

```
Hello
```

Implementing our Own Hash Table with Separate Chaining in Java

Every data structure has its own special characteristics for example a BST is used when quick searching of an element (in log(n)) is required. A heap or a priority queue is used when the minimum or maximum element needs to be fetched in constant time. Similarly a hash table is used to fetch, add and remove an element in constant time. It is necessary for anyone to be clear with the working of a hash table before moving on to the implementation aspect. So here is a brief background on the working of hash table and also it should be noted that we will be using Hash Map and Hash Table terminology interchangeably though in Java HashTables are thread safe while HashMaps are not.

The code we are going to implement is available at Link 1 and Link2

But it is strongly recommended that one must read this blog completely and try and decipher the nitty gritty of what goes into implementing a hash map and then try to write the code yourself.

Background

Every hash-table stores data in the form of (key, value) combination. Interestingly every key is unique in a Hash Table but values can repeat which means values can be same for different keys present in it. Now as we observe in an array to fetch a value we provide the position/index corresponding to the value in that array. In a Hash Table, instead of an index we use a key to fetch the value corresponding to that key. Now the entire process is described below

Every time a key is generated. The key is passed to a hash function. Every hash function has two parts a **Hash code and a Compressor**.

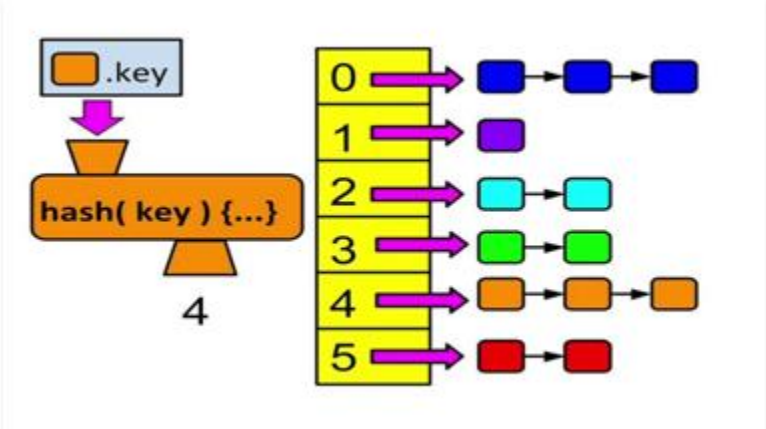
Hash code is an Integer number (random or nonrandom). In Java every Object has its own hash code. We will use the hash code generated by JVM in our hash function and to compress the hash code we modulo(%) the hash code by size of the hash table. *So modulo operator is compressor in our implementation.*

The entire process ensures that for any key, we get an integer position within the size of the Hash Table to insert the corresponding value.

So the process is simple, user gives a (key, value) pair set as input and based on the value generated by hash function an index is generated to where the value corresponding to the particular key is stored. So whenever we need to fetch a value corresponding to a key that is just O(1).

This picture stops being so rosy and perfect when the concept of hash collision is introduced. Imagine for different key values same block of hash table is allocated now where do the previously stored values corresponding to some other previous key go. We certainly can't replace it .That will be disastrous! To resolve this issue we will use Separate Chaining Technique, Please note there are other open addressing techniques like double hashing and linear probing whose efficiency is almost same as to that of separate chaining and you can read more about them at Link 1 Link 2 Link3

Now what we do is make a linked list corresponding to the particular bucket of the Hash Table, to accommodate all the values corresponding to different keys who map to the same bucket.



Now there may be a scenario that all the keys get mapped to the same bucket and we have a linked list of n(size of hash table) size from one single bucket, with all the other buckets empty and this is the worst case where a hash table acts a linked list and searching is O(n).So what do we do ?

Load Factor

If n be the total number of buckets we decided to fill initially say 10 and let's say 7 of them got filled now, so the load factor is 7/10=0.7.

In our implementation whenever we add a key value pair to the Hash Table we check the load factor if it is greater than 0.7 we double the size of our hash table.

Implementation

Hash Node Data Type

We will try to make a generic map without putting any restrictions on the data type of the key and the value . Also every hash node needs to know the next node it is pointing to in the linked list so a next pointer is also required.

The functions we plan to keep in our hash map are

- **get(K key)** : returns the value corresponding to the key if the key is present in **HT** (**H**ast **T**able)
- **getSize()** : return the size of the HT
- **add()** : adds new valid key, value pair to the HT, if already present updates the value
- **remove()** : removes the key, value pair
- **isEmpty()** : returns true if size is zero

Every Hash Map must have an array list/linked list with an initial size and a bucket size which gets increased by unity every time a key, value pair is added and decreased by unity every time a node is deleted

```
ArrayList<HashNode<K, V>> bucket = new ArrayList<>();
```

A **Helper Function** is implemented to get the index of the key, to avoid redundancy in other functions like get, add and remove. This function uses the in built java function to generate a hash code and we compress the hash code by the size of the HT so that the index is within the range of the size of the HT

get()
The get function just takes a key as an input and returns the corresponding value if the key is present in the table otherwise returns null. Steps are:

- Retrieve the input key to find the index in the HT
- Traverse the liked list corresponding to the HT, if you find the value then return it else if you fully traverse the list without returning it means the value is not present in the table and can't be fetched so return null

remove()

- Fetch the index corresponding to the input key using the helper function
- The traversal of linked list similar like in get() but what is special here is that one needs to remove the key along with finding it and two cases arise
- If the key to be removed is present at the head of the linked list
- If the key to be removed is not present at head but somewhere else

add()
Now to the most interesting and challenging function of this entire implementation.It is interesting because we need to dynamically increase the size of our list when load factor is above the value we specified.

- Just like remove steps till traversal and adding and two cases (addition at head spot or non-head spot) remain the same.
- Towards the end if load factor is greater than 0.7
- We double the size of the array list and then recursively call add function on existing keys because in our case hash value generated uses the size of the array to compress the inbuilt JVM hash code we use ,so we need to fetch new indices for the existing keys. This is very important to understand please re read this paragraph till you get a hang of what is happening in the add function.

Java does in its own implementation of Hash Table uses Binary Search Tree if linked list corresponding to a particular bucket tend to get too long.

```
// Java program to demonstrate implementation of our
// own hash table with chaining for collision detection
```

```
import java.util.ArrayList;

// A node of chains
class HashNode<K, V>
{
    K key;
    V value;

    // Reference to next node
    HashNode<K, V> next;

    // Constructor
    public HashNode(K key, V value)
    {
        this.key = key;
        this.value = value;
    }
}

// Class to represent entire hash table
class Map<K, V>
{
    // bucketArray is used to store array of chains
    private ArrayList<HashNode<K, V>> bucketArray;

    // Current capacity of array list
    private int numBuckets;

    // Current size of array list
    private int size;

    // Constructor (Initializes capacity, size and
    // empty chains.
    public Map()
    {
        bucketArray = new ArrayList<>();
        numBuckets = 10;
        size = 0;

        // Create empty chains
        for (int i = 0; i < numBuckets; i++)
            bucketArray.add(null);
    }

    public int size() { return size; }
    public boolean isEmpty() { return size() == 0; }

    // This implements hash function to find index
    // for a key
    private int getBucketIndex(K key)
    {
        int hashCode = key.hashCode();
        int index = hashCode % numBuckets;
        return index;
    }
```

```
// Method to remove a given key
public V remove(K key)
{
    // Apply hash function to find index for given key
    int bucketIndex = getBucketIndex(key);

    // Get head of chain
    HashNode<K, V> head = bucketArray.get(bucketIndex);

    // Search for key in its chain
    HashNode<K, V> prev = null;
    while (head != null)
    {
        // If Key found
        if (head.key.equals(key))
            break;

        // Else keep moving in chain
        prev = head;
        head = head.next;
    }

    // If key was not there
    if (head == null)
        return null;

    // Reduce size
    size--;

    // Remove key
    if (prev != null)
        prev.next = head.next;
    else
        bucketArray.set(bucketIndex, head.next);

    return head.value;
}

// Returns value for a key
public V get(K key)
{
    // Find head of chain for given key
    int bucketIndex = getBucketIndex(key);
    HashNode<K, V> head = bucketArray.get(bucketIndex);

    // Search key in chain
    while (head != null)
    {
        if (head.key.equals(key))
            return head.value;
        head = head.next;
    }
```

```
    // If key not found
    return null;
}

// Adds a key value pair to hash
public void add(K key, V value)
{
    // Find head of chain for given key
    int bucketIndex = getBucketIndex(key);
    HashNode<K, V> head = bucketArray.get(bucketIndex);

    // Check if key is already present
    while (head != null)
    {
        if (head.key.equals(key))
        {
            head.value = value;
            return;
        }
        head = head.next;
    }

    // Insert key in chain
    size++;
    head = bucketArray.get(bucketIndex);
    HashNode<K, V> newNode = new HashNode<K, V>(key, value);
    newNode.next = head;
    bucketArray.set(bucketIndex, newNode);

    // If load factor goes beyond threshold, then
    // double hash table size
    if ((1.0*size)/numBuckets >= 0.7)
    {
        ArrayList<HashNode<K, V>> temp = bucketArray;
        bucketArray = new ArrayList<>();
        numBuckets = 2 * numBuckets;
        size = 0;
        for (int i = 0; i < numBuckets; i++)
            bucketArray.add(null);

        for (HashNode<K, V> headNode : temp)
        {
            while (headNode != null)
            {
                add(headNode.key, headNode.value);
                headNode = headNode.next;
            }
        }
    }
}

// Driver method to test Map class
public static void main(String[] args)
{
    Map<String, Integer>map = new Map<>();
```

```
        map.add("this",1 );
        map.add("coder",2 );
        map.add("this",4 );
        map.add("hi",5 );
        System.out.println(map.size());
        System.out.println(map.remove("this"));
        System.out.println(map.remove("this"));
        System.out.println(map.size());
        System.out.println(map.isEmpty());
    }
}
```

Output :

```
3
4
null
2
false
```

Quick ways to check for Prime and find next Prime in Java

Many programming contest problems are somehow related Prime Numbers. Either we are required to check Prime Numbers, or we are asked to perform certain functions for all prime number between 1 to N. Example: Calculate the sum of all prime numbers between 1 and 1000000.

Java provides two function under java.math.BigInteger to deal with Prime Numbers.

- **isProbablePrime(int certainty):** A method in BigInteger class to check if a given number is prime. For certainty = 1, it return true if BigInteger is prime and false if BigInteger is composite.

Below is Java program to demonstrate above function.

```
// A Java program to check if a number is prime using
// inbuilt function
import java.util.*;
import java.math.*;

class CheckPrimeTest
{
    //Function to check and return prime numbers
    static boolean checkPrime(long n)
    {
        // Converting long to BigInteger
        BigInteger b = new BigInteger(String.valueOf(n));
```

```
        return b.isProbablePrime(1);
    }

    // Driver method
    public static void main (String[] args)
                        throws java.lang.Exception
    {
       long n = 13;
       System.out.println(checkPrime(n));
    }
}
```

- Output:

```
true
```

- **nextProbablePrime()** : Another method present in BigInteger class. This functions returns the next Prime Number greater than current BigInteger.

Below is Java program to demonstrate above function.

```
// Java program to find prime number greater than a
// given number using built in method
import java.util.*;
import java.math.*;

class NextPrimeTest
{
    // Function to get nextPrimeNumber
    static long nextPrime(long n)
    {
        BigInteger b = new BigInteger(String.valueOf(n));
        return Long.parseLong(b.nextProbablePrime().toString());
    }

    // Driver method
    public static void main (String[] args)
                    throws java.lang.Exception
    {
        long n = 14;
        System.out.println(nextPrime(n));
    }
}
```

- Output:

```
17
```

Myth about the file name and class name in Java

The first lecture note given during java class is “In java file name and class name should be the same”. When the above law is violated a compiler error message will appear as below

```
/***** File name: Trial.java ******/
public class Geeks
{
   public static void main(String[] args) {
        System.out.println("Hello world");
   }
}
```

Output:

```
javac Trial.java
Trial.java:9: error: class Geeks is public, should be
                    declared in a file named Geeks.java
public class Geeks
^
1 error
```

But the myth can be violated in such a way to compile the above file.

```
/***** File name: Trial.java ******/
class Geeks
{
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
Step 1: javac Trial.java
```

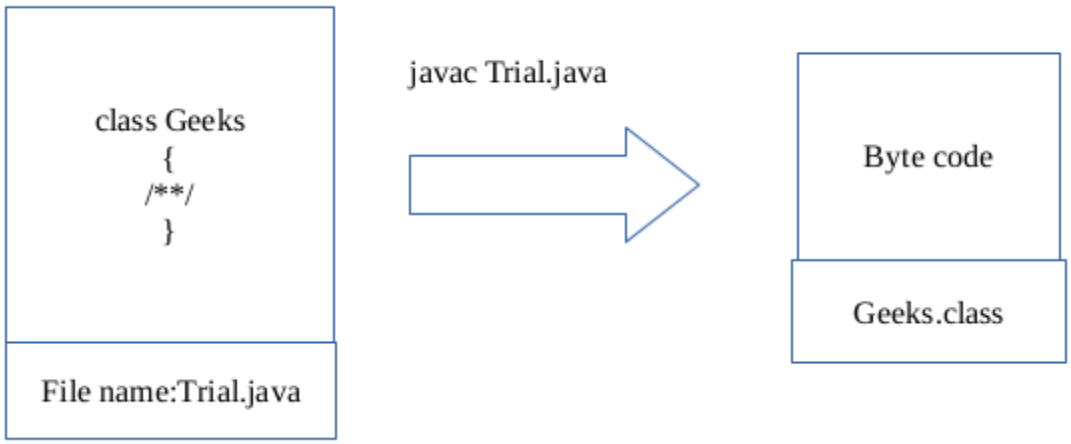
Step1 will create a Geeks.class (byte code) without any error message since the class is not public.

```
Step 2: java Geeks
```

Now the output will be **Hello world**

The myth about the file name and class name should be same only when the class is declared in **public**.

The above program works as follows :



Now this .class file can be executed. By the above features some more miracles can be done. It is possible to have many classes in a java file. For debugging purposes this approach can be used. Each class can be executed separately to test their functionalities(only on one condition: Inheritance concept should not be used).

But in general it is good to follow the myth.

For example:

```
/*** File name: Trial.java ***/
class ForGeeks
{
   public static void main(String[] args){
      System.out.println("For Geeks class");
   }
}

class GeeksTest
{
   public static void main(String[] args){
      System.out.println("Geeks Test class");
   }
}
```

When the above file is compiled as **javac Trial.java** will create two .class files as **ForGeeks.class and GeeksTest.class** .
Since each class has separate main() stub they can be tested individually.
When **java ForGeeks** is executed the output is **For Geeks class**.
When **java GeeksTest** is executed the output is **Geeks Test class**.

Mark-and-Sweep: Garbage Collection Algorithm

Background

All the objects which are created dynamically (using new in C++ and Java) are allocated memory in the heap. If we go on creating objects we might get Out Of Memory error, since it is not possible to allocate heap memory to objects. So we need to clear heap memory by releasing memory for all those objects which are no longer referenced by the program (or the unreachable objects) so that the space is made available for subsequent new objects. This memory can be released by the programmer itself but it seems to be an overhead for the programmer, here garbage collection comes to our rescue, and it automatically releases the heap memory for all the unreferenced objects.
There are many garbage collection algorithms which run in the background. One of them is mark and sweep.

Mark and Sweep Algorithm

Any garbage collection algorithm must perform 2 basic operations. One, it should be able to detect all the unreachable objects and secondly, it must reclaim the heap space used by the garbage objects and make the space available again to the program.
The above operations are performed by Mark and Sweep Algorithm in two phases:
1) Mark phase
2) Sweep phase

Mark Phase
When an object is created, its mark bit is set to 0(false). In the Mark phase, we set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true). Now to perform this operation we simply need to do a graph traversal, a depth first search approach would work for us. Here we can consider every object as a node and then all the nodes (objects) that are reachable from this node (object) are visited and it goes on till we have visited all the reachable nodes.

- Root is a variable that refer to an object and is directly accessible by local variable. We will assume that we have one root only.
- We can access the mark bit for an object by: markedBit(obj).

Algorithm -Mark phase:

```
Mark(root)
    If markedBit(root) = false then
        markedBit(root) = true
        For each v referenced by root
             Mark(v)
```

Note: If we have more than one root, then we simply have to call Mark() for all the root variables.

Sweep Phase

As the name suggests it "sweeps" the unreachable objects i.e. it clears the heap memory for all the unreachable objects. All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to false.
Now the mark value for all the reachable objects is set to false, since we will run the algorithm (if required) and again we will go through the mark phase to mark all the reachable objects.

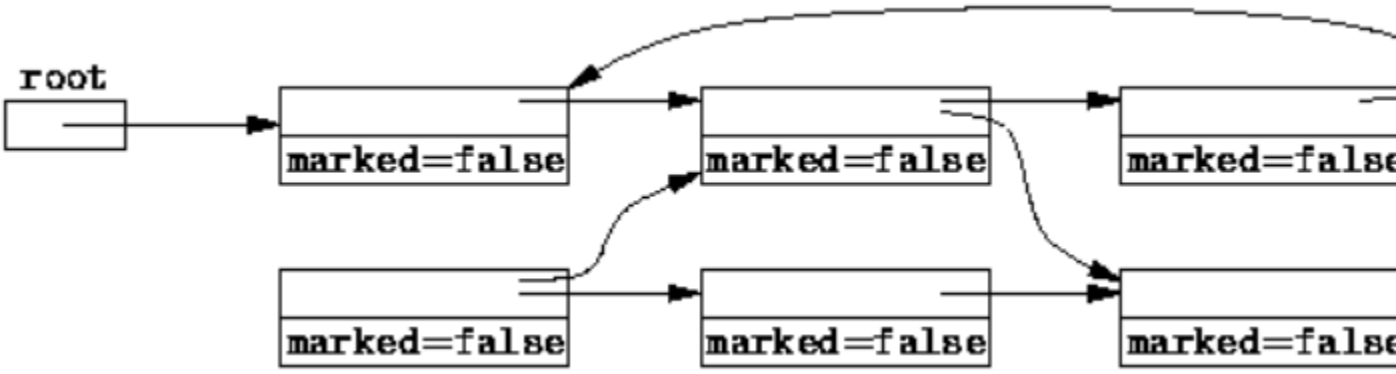
Algorithm – Sweep Phase

```
Sweep()
For each object p in heap
    If markedBit(p) = true then
        markedBit(p) = false
    else
        heap.release(p)
```

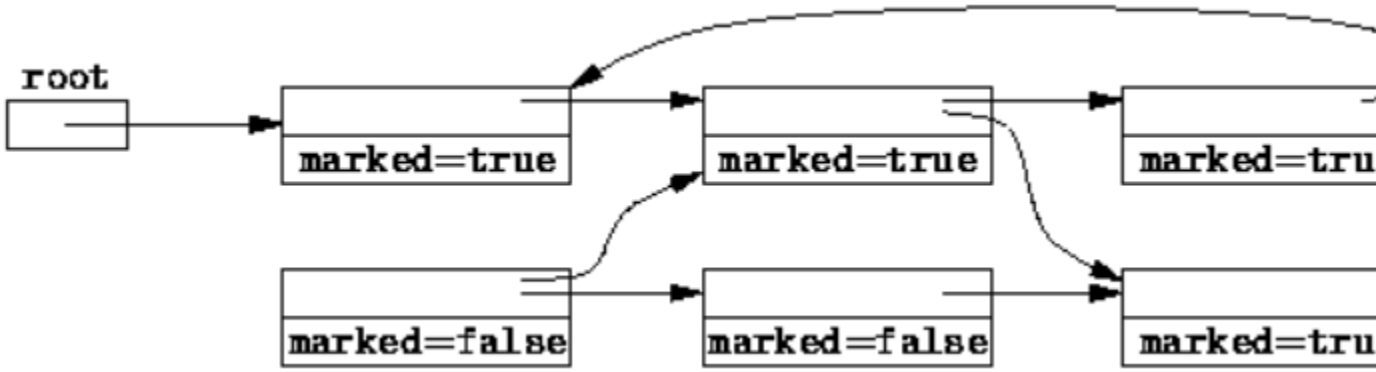
The mark-and-sweep algorithm is called a tracing garbage collector because is traces out the entire collection of objects that are directly or indirectly accessible by the program.

Example:

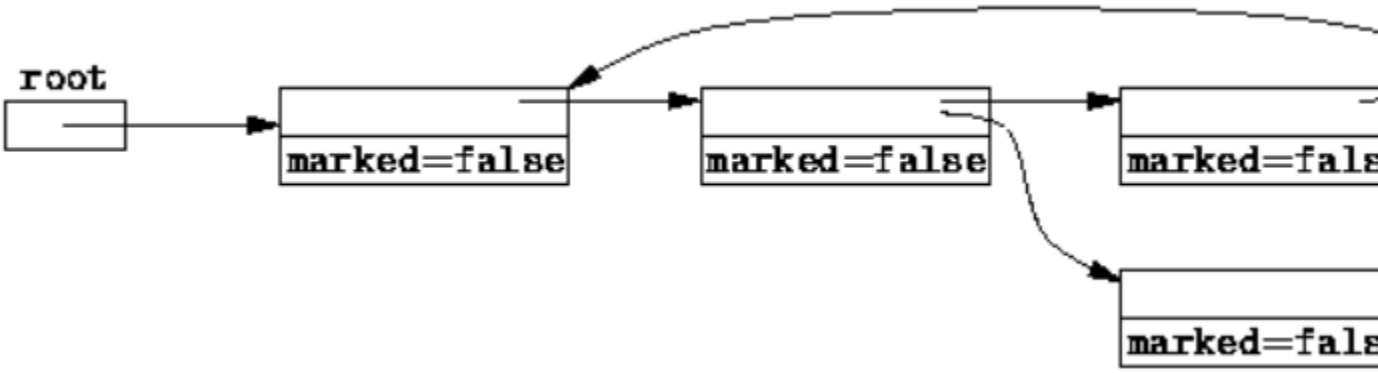
a) All the objects have their marked bits set to false.



b) Reachable objects are marked true



c) Non reachable objects are cleared from the heap.



Advantages of Mark and Sweep Algorithm

- It handles the case with cyclic references, even in case of a cycle, this algorithm never ends up in an infinite loop.
- There are no additional overheads incurred during the execution of the algorithm.

Disadvantages of Mark and Sweep Algorithm

- The main disadvantage of the mark-and-sweep approach is the fact that that normal program execution is suspended while the garbage collection algorithm runs.
- Other disadvantage is that, after the Mark and Sweep Algorithm is run several times on a program, reachable objects end up being separated by many, small unused memory regions. Look at the below figure for better understanding.

Figure:

Heap Memory

Here white blocks denote the free memory, while the grey blocks denote the memory taken by all the reachable objects.
Now the free segments (which are denoted by white color) are of varying size let's say the 5 free segments are of size 1, 1, 2, 3, 5 (size in units).
Now we need to create an object which takes 10 units of memory, now assuming that memory can be allocated only in contiguous form of blocks, the creation of object isn't possible although we have an available memory space of 12 units and it will cause OutOfMemory error. This problem is termed as "Fragmentation". We have memory available in "fragments" but we are unable to utilize that memory space.
We can reduce the fragmentation by compaction; we shuffle the memory content to place all the free memory blocks together to form one large block. Now consider the above example, after compaction we have a continuous block of free memory of size 12 units so now we can allocate memory to an object of size 10 units.

Static vs Dynamic Binding in Java

Static Binding: The binding which can be resolved at compile time by compiler is known as static or early binding. Binding of all the static, private and final methods is done at compile-time .

Why binding of static, final and private methods is always a static binding?
Static binding is better performance wise (no extra overhead is required). Compiler knows that all such methods **cannot be overridden** and will always be accessed by object of local class. Hence compiler doesn't have any difficulty to determine object of class (local class for sure). That's the reason binding for such methods is static.
Let's see by an example

```
public class NewClass
{
    public static class superclass
    {
        static void print()
        {
            System.out.println("print in superclass.");
        }
    }
    public static class subclass extends superclass
    {
```

```
        static void print()
        {
            System.out.println("print in subclass.");
        }
    }

    public static void main(String[] args)
    {
        superclass A = new superclass();
        superclass B = new subclass();
        A.print();
        B.print();
    }
}
```

Before scrolling further down, Guess the output of the above program?

Output:

```
print in superclass.
print in superclass.
```

As you can see, in both cases print method of superclass is called. Lets see how this happens

- We have created one object of subclass and one object of superclass with the reference of the superclass.
- Since the print method of superclass is static, compiler knows that it will not be overridden in subclasses and hence compiler knows during compile time which print method to call and hence no ambiguity.

As an exercise, reader can change the reference of object B to subclass and then check the output.

Dynamic Binding: In Dynamic binding compiler doesn't decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have same method . Let's see by an example

```
public class NewClass
{
    public static class superclass
    {
        void print()
        {
            System.out.println("print in superclass.");
        }
    }

    public static class subclass extends superclass
    {
        @Override
        void print()
        {
```

```
            System.out.println("print in subclass.");
        }
    }

    public static void main(String[] args)
    {
        superclass A = new superclass();
        superclass B = new subclass();
        A.print();
        B.print();
    }
}
```

Output:

```
print in superclass.
print in subclass.
```

Here the output differs. But why? Let's break down the code and understand it thoroughly.

- Methods are not static in this code.
- During compilation, the compiler has no idea as to which print has to be called since compiler goes only by referencing variable not by type of object and therefore the binding would be delayed to runtime and therefore the corresponding version of print will be called based on type on object.

Important Points

- private, final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon run time object.
- Static binding uses Type information for binding while Dynamic binding uses Objects to resolve binding.
- Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e, at run time.

Collections in Java

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are two main root interfaces of Java collection classes.

Need for Collection Framework :

Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were array or Vector or Hashtable. All three of these collections had no common interface.
For example, if we want to access elements of array, vector or Hashtable. All these three have different methods and syntax for accessing members:

```
// Java program to show whey collection framework was needed
import java.io.*;
import java.util.*;

class Test
{
    public static void main (String[] args)
    {
        // Creating instances of array, vector and hashtable
        int arr[] = new int[] {1, 2, 3, 4};
        Vector<Integer> v = new Vector();
        Hashtable<Integer, String> h = new Hashtable();
        v.addElement(1);
        v.addElement(2);
        h.put(1,"geeks");
        h.put(2,"4geeks");

        // Array instance creation requires [], while Vector
        // and hastable require ()
        // Vector element insertion requires addElement(), but
        // hashtable element insertion requires put()

        // Accessing first element of array, vector and hashtable
        System.out.println(arr[0]);
        System.out.println(v.elementAt(0));
        System.out.println(h.get(1));

        // Array elements are accessed using [], vector elements
        // using elementAt() and hashtable elements using get()
    }
}
```

Output:

```
1
```

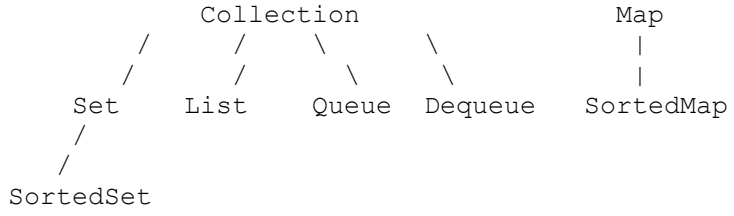
```
1
geek
```

As we can see, none of the collections (Array, Vector or Hashtable) implements a standard member access interface. So, it was very difficult for programmers to write algorithm that can work for all kind of collections.
Another drawback is that, most of the 'Vector' methods are final. So, we cannot extend 'Vector' class to implement a similar kind of collection.
Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced Collection Framework, they introduced collections in JDK 1.2 and changed the legacy Vector and Hashtable to conform to the collection framework.

Advantages of Collection Framework:

1. Consistent API : The API has basic set of interfaces like Collection, Set, List, or Map. All those classes (such as ArrayList, LinkedList, Vector etc) which implements, these interfaces have some common set of methods.
2. Reduces programming effort: The programmer need not to worry about design of Collection rather than he can focus on its best use in his program.
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.

Hierarchy of Collection Framework



Core Interfaces in Collections

Note that this diagram shows only core interfaces.
Collection : Root interface with basic methods like add(), remove(), contains(), isEmpty(), addAll(), ... etc.

Set : Doesn't allow duplicates. Example implementations of Set interface are HashSet (Hashing based) and TreeSet (balanced BST based). Note that TreeSet implements **SortedSet**.

List : Can contain duplicates and elements are ordered. Example implementations are LinkedList (linked list based) and ArrayList (dynamic array based)

Queue : Typically order elements in FIFO order except exceptions like PriorityQueue.

Deque : Elements can be inserted and removed at both ends. Allows both LIFO and FIFO.

Map : Contains Key value pairs. Doesn't allow duplicates. Example implementation are HashMap and TreeMap.

TreeMap implements **SortedMap.**

The difference between Set and Map interface is, in Set we have only keys, but in Map, we have key value pairs.

We will soon be discussing examples of interface implementations.

How to use Iterator in Java?

'Iterator' is an interface which belongs to collection framework. It allows us to traverse the collection, access the data element and remove the data elements of the collection.

java.util package has **public interface Iterator** and contains three methods:

1. **boolean hasNext()**: It returns true if Iterator has more element to iterate.

2. **Object next()**: It returns the next element in the collection until the hasNext()method return true. This method throws 'NoSuchElementException' if there is no next element.

3.**void remove()**: It removes the current element in the collection. This method throws 'IllegalStateException' if this function is called before next( ) is invoked.

```
// Java code to illustrate the use of iterator
import java.io.*;
import java.util.*;

class Test
{
    public static void main (String[] args)
    {
        ArrayList<String> list = new ArrayList<String>();

        list.add("A");
        list.add("B");
        list.add("C");
        list.add("D");
        list.add("E");

        // Iterator to traverse the list
        Iterator iterator = list.iterator();

        System.out.println("List elements : ");

        while (iterator.hasNext())
            System.out.print(iterator.next()+ " ");

        System.out.println();
    }
```

```
}
```

Output:

```
List elements :
A B C D E
```

'ListIterator' in Java is an Iterator which allows users to traverse Collection in both direction. It contains the following methods:

1. **void add(Object object)**: It inserts object in front of the element that is returned by the next( ) function.

2. **boolean hasNext( )**: It returns true if the list has a next element.

3. **boolean hasPrevious( )**: It returns true if the list has a previous element.

4. **Object next( )**: It returns the next element of the list. It throws 'NoSuchElementException' if there is no next element in the list.

5. **Object previous( )**: It returns the previous element of the list. It throws 'NoSuchElementException' if there is no previous element.

6. **void remove( )**: It removes the current element from the list. It throws 'IllegalStateException' if this function is called before next( ) or previous( ) is invoked.

```
// Java code to illustrate the use of ListIterator
import java.io.*;
import java.util.*;

class Test
{
    public static void main (String[] args)
    {
        ArrayList<String> list = new ArrayList<String>();

        list.add("A");
        list.add("B");
        list.add("C");
        list.add("D");
        list.add("E");

        // ListIterator to traverse the list
        ListIterator iterator = list.listIterator();

        // Traversing the list in forward direction
        System.out.println("Displaying list elements in forward direction :
");
```

```
        while (iterator.hasNext())
            System.out.print(iterator.next()+ " ");

        System.out.println();

        // Traversing the list in backward direction
        System.out.println("Displaying list elements in backward direction :
");

        while(iterator.hasPrevious())
            System.out.print(iterator.previous()+ " ");

        System.out.println();
    }
}
```

Output:

```
Displaying list elements in forward direction :
A B C D E
Displaying list elements in backward direction :
E D C B A
```

Iterator vs Foreach In Java

Background :

Iterator is an interface provided by collection framework to traverse a collection and for a sequential access of items in the collection.

```
// Iterating over collection 'c' using terator
for (Iterator i = c.iterator(); i.hasNext(); )
    System.out.println(i.next());
```

For each loop is meant for traversing items in a collection.

```
// Iterating over collection 'c' using for-each
for (Element e: c)
    System.out.println(e);
```

We read the ':' used in for-each loop as "in". So loop reads as "for each element e in elements", here elements is the collection which stores Element type items.

Note : In Java 8 using lambda expressions we can simply replace for-each loop with

```
elements.forEach (e -> System.out.println(e) );
```

Difference between the two traversals

In for-each loop, we can't modify collection, it will throw a ConcurrentModificationException on the other hand with iterator we can modify collection.

Modifying a collection simply means removing an element or changing content of an item stored in the collection. This occurs because for-each loop implicitly creates an iterator but it is not exposed to the user thus we can't modify the items in the collections.

When to use which traversal?

- If we have to modify collection, we must use an Iterator.
- While using nested for loops it is better to use for-each loop, consider the below code for better understanding.

```
// Java program to demonstrate working of nested iterators
// may not work as expected and throw exception.
import java.util.*;

public class Main
{
    public static void main(String args[])
    {
        // Create a link list which stores integer elements
        List<Integer> l = new LinkedList<Integer>();

        // Now add elements to the Link List
        l.add(2);
        l.add(3);
        l.add(4);

        // Make another Link List which stores integer elements
        List<Integer> s=new LinkedList<Integer>();
        s.add(7);
        s.add(8);
        s.add(9);

        // Iterator to iterate over a Link List
        for (Iterator<Integer> itr1=l.iterator(); itr1.hasNext(); )
        {
            for (Iterator<Integer> itr2=s.iterator(); itr2.hasNext(); )
            {
                if (itr1.next() < itr2.next())
                {
                    System.out.println(itr1.next());
                }
            }
        }
    }
}
```

Output:

```
Exception in thread "main" java.util.NoSuchElementException
        at java.util.LinkedList$ListItr.next(LinkedList.java:888)
        at Main.main(Main.java:29)
```

The above code throws java.util.NoSuchElementException.

In the above code we are calling the next() method again and again for itr1 (i.e., for List l). Now we are advancing the iterator without even checking if it has any more elements left in the collection(in the inner loop), thus we are advancing the iterator more than the number of elements in the collection which leads to NoSuchElementException.

for-each loops are tailor made for nested loops. Replace the iterator code with the below code.

```
// Java program to demonstrate working of nested for-each
import java.util.*;
public class Main
{
    public static void main(String args[])
    {
        // Create a link list which stores integer elements
        List<Integer> l=new LinkedList<Integer>();

        // Now add elements to the Link List
        l.add(2);
        l.add(3);
        l.add(4);

        // Make another Link List which stores integer elements
        List<Integer> s=new LinkedList<Integer>();
        s.add(2);
        s.add(4);
        s.add(5);
        s.add(6);

        // Iterator to iterate over a Link List
        for (int a:l)
        {
            for (int b:s)
            {
                if (a<b)
                    System.out.print(a + " ");
            }
        }
    }
}
```

Output:

```
2 2 2 3 3 3 4 4
```

Performance Analysis

Traversing a collection using for-each loops or iterators give the same performance. Here, by performance we mean the time complexity of both these traversals.

If you iterate using the old styled C for loop then we might increase the time complexity drastically.

// Here l is List ,it can be ArrayList /LinkedList and n is size of the List

```
for (i=0;i<n;i++)
   System.out.println(l.get(i));
```

Here if the list l is an ArrayList then we can access it in O(1) time since it is allocated contiguous memory blocks (just like an array) i.e random access is possible. But if the collection is LinkedList, then random access is not possible since it is not allocated contiguous memory blocks, so in order to access a element we will have to traverse the link list till you get to the required index, thus the time taken in worst case to access an element will be O(n).

Iterator and for-each loop are faster than simple for loop for collections with no random access, while in collections which allows random access there is no performance change with for-each loop/for loop/iterator.

Retrieving Elements from Collection in Java (For-each, Iterator, ListIterator & EnumerationIterator)

Prerequisite : Collection in Java

Following are the 4 ways to retrieve any elements from a collection object:

For-each

For each loop is meant for traversing items in a collection.

```
// Iterating over collection 'c' using for-each
for (Element e: c)
    System.out.println(e);
```

We read the ':' used in for-each loop as "in". So loop reads as "for each element e in elements", here elements is the collection which stores Element type items.

Note : In Java 8 using lambda expressions we can simply replace for-each loop with

```
elements.forEach (e -> System.out.println(e) );
```

Iterator Interface

Iterator is an interface provided by collection framework to traverse a collection and for a sequential access of items in the collection.

```
// Iterating over collection 'c' using terator
for (Iterator i = c.iterator(); i.hasNext(); )
    System.out.println(i.next());
```

It has 3 methods:

- *boolean hasNext():* This method returns true if the iterator has more elements.
- *elements next()*: This method returns the next elements in the iterator.
- *void remove()*: This method removes from the collection the last elements returned by the iterator.

```
// Java program to demonstrate working of iterators
import java.util.*;
public class IteratorDemo
{
    public static void main(String args[])
    {
        //create a Hashset to store strings
        HashSet<String> hs = new HashSet<String>() ;

        // store some string elements
        hs.add("India");
        hs.add ("America");
        hs.add("Japan");

        // Add an Iterator to hs.
        Iterator it = hs.iterator();

        // Display element by element using Iterator
        while (it.hasNext())
            System.out.println(it.next());
    }
}
```

Output:

```
America
Japan
India
```

Refer Iterator vs For-each for differences between iterator and for-each.

ListIterator Interface

*ListIterator*is an interface that contains methods to retrieve the elements from a collection object, both in **forward and reverse directions**. This iterator is for list based collections.

It has following important methods:

- *booleanhasNext()*: This returns true if the ListIterator has more elements when traversing the list in the forward direction.
- *booleanhasPerivous()*: This returns true if the ListIterator has more elements when traversing the list in the reverse direction.
- *element next()*: This returns the next element in the list.
- *element previous()*:This returns the previous element in the list.
- *void remove()*: This removes from the list the last elements that was returned by the next() or previous() methods.
- *int nextIndex()* Returns the index of the element that would be returned by a subsequent call to next(). (Returns list size if the list iterator is at the end of the list.)
- *int previousIndex()* Returns the index of the element that would be returned by a subsequent call to previous(). (Returns -1 if the list iterator is at the beginning of the list.)

Source: ListIterator Oracle Docs

How is Iterator different from ListIterator?

Iterator can retrieve the elements only in forward direction. But ListIterator can retrieve the elements in forward and reverse direction also. So ListIterator is preferred to Iterator.

Using ListItarator, we can get iterator's current position

Since ListIterator can access elements in both directions and supports additional operators, ListIterator cannot be applied on Set (e.g., HashSet and TreeSet. See this). However, we can use LisIterator with vector and list (e.g. ArrayList).

```
// Java program to demonstrate working of ListIterator
import java. util.* ;

class Test
{
    public static void main(String args[])
    {
        // take a vector to store Integer objects
        Vector<Integer> v = new Vector<Integer>();

        // Adding Elements to Vector
        v.add(10);
```

```
        v.add(20);
        v.add(30);

        // Creating a ListIterator
        ListIterator lit = v.listIterator();
        System.out.println("In Forward direction:");

        while (lit.hasNext())
            System.out.print(lit.next()+" ") ;

        System.out.print("\n\nIn backward direction:\n") ;
        while (lit.hasPrevious())
            System.out.print(lit.previous()+" ");
    }
}
```

Output :

```
In Forward direction:
10 20 30

In backward direction:
30 20 10
```

EnumerationIterator Interface

The interface is useful to retrieve one by one the element. This iterator is based on data from Enumeration and has methods:

- *booleanhasMoreElements()*: This method tests if the Enumeration has any more elements or not .
- *element nextElement()*: This returns the next element that is available in elements that is available in Enumeration

What is the difference between Iterator and Enumeration?

Iterator has an option to remove elements from the collection which is not available in Enumeration. Also, Iterator has methods whose names are easy to follow and Enumeration methods are difficult to remember.

```
import java.util.Vector;
import java.util.Enumeration;
public class Test
{
    public static void main(String args[])
    {
        Vector dayNames = new Vector();
        dayNames.add("Sunday");
```

```
        dayNames.add("Monday");
        dayNames.add("Tuesday");
        dayNames.add("Wednesday");
        dayNames.add("Thursday");
        dayNames.add("Friday");
        dayNames.add("Saturday");

        // Creating enumeration
        Enumeration days = dayNames.elements();

        // Retrieving elements of enumeration
        while (days.hasMoreElements())
            System.out.println(days.nextElement());
    }
}
```

Output:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Set in Java

- Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored.
- Basically, Set is implemented by HashSet, LinkedSet or TreeSet (sorted representation).
- Set has various methods to add, remove clear, size, etc to enhance the usage of this interface

```
// Java code for adding elements in Set
import java.util.*;
public class Set_example
{
    public static void main(String[] args)
    {
        // Set deonstration using HashSet
        Set<String> hash_Set = new HashSet<String>();
        hash_Set.add("Geeks");
        hash_Set.add("For");
        hash_Set.add("Geeks");
        hash_Set.add("Example");
        hash_Set.add("Set");
        System.out.print("Set output without the duplicates");

        System.out.println(hash_Set);

        // Set deonstration using TreeSet
```

```
        System.out.print("Sorted Set after passing into TreeSet");
        Set<String> tree_Set = new TreeSet<String>(hash_Set);
        System.out.println(tree_Set);
    }
}
```

(Please note that we have entered a duplicate entity but it is not displayed in the output. Also, we can directly sort the entries by passing the unordered Set in as the parameter of TreeSet).

Output:

```
Set output without the duplicates[Geeks, Example, For, Set]
Sorted Set after passing into TreeSet[Example, For, Geeks, Set]
```

Note: As we can see the duplicate entry “Geeks” is ignored in the final output, Set interface doesn’t allow duplicate entries.

Now we will see some of the basic operations on the Set i.e. Union, Intersection and Difference.

Let’s take an example of two integer Sets:

- [1, 3, 2, 4, 8, 9, 0]
- [1, 3, 7, 5, 4, 0, 7, 5]

Union
In this, we could simply add one Set with other. Since the Set will itself not allow any duplicate entries, we need not take care of the common values.

Expected Output:

```
Union : [0, 1, 2, 3, 4, 5, 7, 8, 9]
```

Intersection
We just need to retain the common values from both Sets.

Expected Output:

```
Intersection : [0, 1, 3, 4]
```

Difference
We just need to remove tall the values of one Set from the other.

Expected Output:

```
Difference : [2, 8, 9]
```

```
// Java code for demonstrating union, intersection and difference
// on Set
import java.util.*;
public class Set_example
{
    public static void main(String args[])
    {
        Set<Integer> a = new HashSet<Integer>();
        a.addAll(Arrays.asList(new Integer[] {1, 3, 2, 4, 8, 9, 0}));
        Set<Integer> b = new HashSet<Integer>();
        b.addAll(Arrays.asList(new Integer[] {1, 3, 7, 5, 4, 0, 7, 5}));

        // To find union
        Set<Integer> union = new HashSet<Integer>(a);
        union.addAll(b);
        System.out.print("Union of the two Set");
        System.out.println(union);

        // To find intersection
        Set<Integer> intersection = new HashSet<Integer>(a);
        intersection.retainAll(b);
        System.out.print("Intersection of the two Set");
        System.out.println(intersection);

        // To find the symmetric difference
        Set<Integer> difference = new HashSet<Integer>(a);
        difference.removeAll(b);
        System.out.print("Difference of the two Set");
        System.out.println(difference);
    }
}
```

Output:

```
Union of the two Set[0, 1, 2, 3, 4, 5, 7, 8, 9]
Intersection of the two Set[0, 1, 3, 4]
Difference of the two Set[2, 8, 9]
```

HashSet in Java

HashSet:

- Implements Set Interface.

- Underlying data structure for HashSet is hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in same order. Objects are inserted based on their hash code.
- NULL elements are allowed in HashSet.
- HashSet also implements Searlizable and Cloneable interfaces.

Constructors in HashSet:

```
HashSet h = new HashSet();
Default initial capacity is 16 and default load factor is 0.75.

HashSet h = new HashSet(int initialCapacity);
default loadFactor of 0.75

HashSet h = new HashSet(int initialCapacity, float loadFactor);

HashSet h = new HashSet(Collection C);
```

What is initial capacity and load factor?

The initial capacity means the number of buckets when hashtable (HashSet internally uses hashtable data structure) is created. Number of buckets will be automatically increased if the current size gets full.
The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

$$\text{load factor} = \frac{\text{Number of stored elements in the table}}{\text{Size of the hash table}}$$

E.g. If internal capacity is 16 and load factor is 0.75 then, number of buckets will automatically get increased when table has 12 elements in it.

Effect on performance:
Load factor and initial capacity are two main factors that affect the performance of HashSet operations. Load factor of 0.75 provides very effective performance as respect to time and space complexity. If we increase the load factor value more than that then memory overhead will be reduced (because it will decrease internal rebuilding operation) but, it will affect the add and search operation in hashtable. To reduce the rehashing operation we should choose initial capacity wisely. If initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operation will ever occur.

Important Methods in HashSet:

1. **boolean add(E e)** : add the specified element if it is not present, if it is present then return false.
2. **void clear()** : removes all the elements from set.
3. **boolean contains(Object o)** : return true if element is present in set.
4. **boolean remove(Object o)** : remove the element if it is present in set.
5. **Iterator iterator()** : return an iterator over the element in the set.

Sample Program:

```
// Java program to demonstrate working of HashSet
import java.util.*;

class Test
{
    public static void main(String[]args)
    {
        HashSet<String> h = new HashSet<String>();

        // adding into HashSet
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add("India");// adding duplicate elements

        // printing HashSet
        System.out.println(h);
        System.out.println("List contains India or not:" +
                           h.contains("India"));

        // Removing an item
        h.remove("Australia");
        System.out.println("List after removing Australia:"+h);

        // Iterating over hash set items
        System.out.println("Iterating over list:");
        Iterator<String> i = h.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

Output of the above program:

```
[Australia, South Africa, India]
List contains India or not:true
List after removing Australia:[South Africa, India]
Iterating over list:
South Africa
India
```

How HashSet internally work?

All the classes of Set interface internally backed up by Map. HashSet uses HashMap for storing

its object internally. You must be wondering that to enter a value in HashMap we need a key-value pair, but in HashSet we are passing only one value.

Then how is it storing in HashMap?
Actually the value we insert in HashSet acts as key to the map Object and for its value java uses a constant variable. So in key-value pair all the keys will have same value.

If we look at the implementation of HashSet in java doc, it is something like this;

```
private transient HashMap map;

// Constructor - 1
// All the constructors are internally creating HashMap Object.
public HashSet()
{
    // Creating internally backing HashMap object
    map = new HashMap<>();
}

// Constructor - 2
public HashSet(int initialCapacity)
{
    // Creating internally backing HashMap object
    map = new HashMap<>(initialCapacity);
}

// Dummy value to associate with an Object in Map
private static final Object PRESENT = new Object();
```

If we look at add() method of HashSet class:

```
public boolean add(E e)
{
   return map.put(e, PRESENT) == null;
}
```

We can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant “PRESENT” as its value.

remove() method also works in the same manner. It internally calls remove method of Map interface.

```
public boolean remove(Object o)
{
  return map.remove(o) == PRESENT;
}
```

Time Complexity of HashSet Operations:
The underlying data structure for HashSet is hashtable. So amortize (average or usual case) time

complexity for add, remove and look-up (contains method) operation of HashSet takes **O(1)** time.

Linked List in java

In Java, LinkedList class implements the list interface.

This class consists of the following methods :

1. **boolean add(Object element)** : It appends the element to the end of the list.

2. **void add(int index, Object element)**: It inserts the element at the position 'index' in the list.

3. **void addFirst(Object element)** : It inserts the element at the beginning of the list.

4. **void addLast(Object element)** : It appends the element at the end of the list.

5. **boolean contains(Object element)** : It returns true if the element is present in the list.

6. **Object get(int index)** : It returns the element at the position 'index' in the list. It throws 'IndexOutOfBoundsException' if the index is out of range of the list.

7. **int indexOf(Object element)** : If element is found, it returns the index of the first occurrence of the element. Else, it returns -1.

8. **Object remove(int index)** : It removes the element at the position 'index' in this list. It throws 'NoSuchElementException' if the list is empty.

9. **int size()** : It returns the number of elements in this list.

10. **void clear()** : It removes all of the elements from the list.

```
// Java code for Linked List implementation


import java.util.*;


public class Test

{
```

```
public static void main(String args[])
{
    // Creating object of class linked list
    LinkedList<String> object = new LinkedList<String>();

    // Adding elements to the linked list
    object.add("A");
    object.add("B");
    object.addLast("C");
    object.addFirst("D");
    object.add(2, "E");
    object.add("F");
    object.add("G");
    System.out.println("Linked list : " + object);

    // Removing elements from the linked list
    object.remove("B");
    object.remove(3);
    object.removeFirst();
    object.removeLast();
    System.out.println("Linked list after deletion: " + object);

    // Finding elements in the linked list
    boolean status = object.contains("E");

    if(status)
        System.out.println("List contains the element 'E' ");
    else
        System.out.println("List doesn't contain the element 'E'");

    // Number of elements in the linked list
```

```
        int size = object.size();
        System.out.println("Size of linked list = " + size);

        // Get and set elements from linked list
        Object element = object.get(2);
        System.out.println("Element returned by get() : " + element);
        object.set(2, "Y");
        System.out.println("Linked list after change : " + object);
    }
}
```

Output :

```
Linked list : [D, A, E, B, C, F, G]
Linked list after deletion: [A, E, F]
List contains the element 'E'
Size of linked list = 3
Element returned by get() : F
Linked list after change : [A, E, Y]
```

Stack Class in Java

Java provides an inbuilt object type called **Stack**. It is a collection that is based on the last in first out (LIFO) principle. On Creation, a stack is empty.

It extends **Vector** class with five methods that allow a vector to be treated as a stack. The five methods are:

1. **Object push(Object element)** : Pushes an element on the top of the stack.

2. **Object pop()** : Removes and returns the top element of the stack. An 'EmptyStackException' exception is thrown if we call pop() when the invoking stack is empty.

3. **Object peek( )** : Returns the element on the top of the stack, but does not remove it.

4. **boolean empty()** : It returns true if nothing is on the top of the stack. Else, returns false.

5. **int search(Object element)** : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

```
// Java code for stack implementation

import java.io.*;
import java.util.*;

class Test
{
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for(int i = 0; i < 5; i++)
        {
            stack.push(i);
        }
    }

    // Popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop :");

        for(int i = 0; i < 5; i++)
        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);
        }
    }

    // Displaying element on the top of the stack
    static void stack_peek(Stack<Integer> stack)
    {
        Integer element = (Integer) stack.peek();
        System.out.println("Element on stack top : " + element);
    }

    // Searching element in the stack
    static void stack_search(Stack<Integer> stack, int element)
    {
        Integer pos = (Integer) stack.search(element);

        if(pos == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element is found at position " + pos);
    }


    public static void main (String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();
```

```
        stack_push(stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);
    }
}
```

Output :

```
Pop :
4
3
2
1
0
Element on stack top : 4
Element is found at position 3
Element not found
```

Differences between HashMap and HashTable in Java

HashMap and Hashtable store key/value pairs in a hash table. When using a Hashtable or HashMap, we specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Sample Java code.

```
// A sample Java program to demonstrate HashMap and HashTable
import java.util.*;
import java.lang.*;
import java.io.*;

/* Name of the class has to be "Main" only if the class is public. */
class Ideone
{
    public static void main(String args[])
    {
        //----------hashtable -------------------------
        Hashtable<Integer,String> ht=new Hashtable<Integer,String>();
        ht.put(101," ajay");
        ht.put(101,"Vijay");
        ht.put(102,"Ravi");
        ht.put(103,"Rahul");
        System.out.println("-------------Hash table--------------");
        for (Map.Entry m:ht.entrySet()) {
            System.out.println(m.getKey()+" "+m.getValue());
```

```
        }

        //----------------hashmap---------------------------------
        HashMap<Integer,String> hm=new HashMap<Integer,String>();
        hm.put(100,"Amit");
        hm.put(104,"Amit");  // hash map allows duplicate values
        hm.put(101,"Vijay");
        hm.put(102,"Rahul");
        System.out.println("-----------Hash map-----------");
        for (Map.Entry m:hm.entrySet()) {
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Output:

```
-------------Hash table--------------
103 Rahul
102 Ravi
101 Vijay
-----------Hash map-----------
100 Amit
101 Vijay
102 Rahul
104 Amit
```

Hashmap vs Hashtable
1. HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
2. HashMap allows one null key and multiple null values whereas Hashtable doesn't allow any null key or value.
3. HashMap is generally preferred over HashTable if thread synchronization is not needed

Why HashTable doesn't allow null and HashMap does?
To successfully store and retrieve objects from a HashTable, the objects used as keys must implement the hashCode method and the equals method. Since null is not an object, it can't implement these methods. HashMap is an advanced version and improvement on the Hashtable. HashMap was created later.

HashMap and TreeMap in Java

HashMap and TreeMap are part of collection framework.

HashMap

java.util.HashMap class is a Hashing based implementation. In HashMap, we have a key and a value pair<Key, Value>.

```
 HashMap<K, V> hmap = new HashMap<K, V>();
```

Let us consider below example where we have to count occurrences of each integer in given array of integers.

```
Input: arr[] = {10, 3, 5, 10, 3, 5, 10};
Output: Frequency of 10 is 3
        Frequency of 3 is 2
        Frequency of 5 is 2
/* Java program to print frequencies of all elements using
   HashMap */
import java.util.*;

class Main
{
    // This function prints frequencies of all elements
    static void printFreq(int arr[])
    {
        // Creates an empty HashMap
        HashMap<Integer, Integer> hmap =
                    new HashMap<Integer, Integer>();

        // Traverse through the given array
        for (int i = 0; i < arr.length; i++)
        {
            Integer c = hmap.get(arr[i]);

            // If this is first occurrence of element
            if (hmap.get(arr[i]) == null)
               hmap.put(arr[i], 1);

            // If elements already exists in hash map
            else
              hmap.put(arr[i], ++c);
        }

        // Print result
        for (Map.Entry m:hmap.entrySet())
          System.out.println("Frequency of " + m.getKey() +
                             " is " + m.getValue());
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int arr[] = {10, 34, 5, 10, 3, 5, 10};
        printFreq(arr);
    }
}
```

Output:

```
Frequency of 34 is 1
Frequency of 3 is 1
Frequency of 5 is 2
Frequency of 10 is 3
```

Key Points

- HashMap does not maintain any order neither based on key nor on basis of value, If we want the keys to be maintained in a sorted order, we need to use TreeMap.
- **Complexity**: get/put/containsKey() operations are O(1) in average case but we can't guarantee that since it all depends on how much time does it take to compute the hash.

Application:
HashMap is basically an implementation of hashing. So wherever we need hashing with key value pairs, we can use HashMap. For example, in Web Applications username is stored as a key and user data is stored as a value in the HashMap, for faster retrieval of user data corresponding to a username.

TreeMap

TreeMap can be a bit handy when we only need to store unique elements in a sorted order. Java.util.TreeMap uses a red-black tree in the background which makes sure that there are no duplicates; additionally it also maintains the elements in a sorted order.

```
TreeMap<K, V> hmap = new TreeMap<K, V>();
```

Below is TreeMap based implementation of same problem. This solution has more time complexity O(nLogn) compared to previous one which has O(n). The advantage of this method is, we get elements in sorted order.

```
/* Java program to print frequencies of all elements using
   TreeMap */
import java.util.*;

class Main
{
    // This function prints frequencies of all elements
    static void printFreq(int arr[])
    {
        // Creates an empty TreeMap
        TreeMap<Integer, Integer> tmap =
                    new TreeMap<Integer, Integer>();

        // Traverse through the given array
        for (int i = 0; i < arr.length; i++)
        {
            Integer c = tmap.get(arr[i]);
```

```
            // If this is first occurrence of element
            if (tmap.get(arr[i]) == null)
               tmap.put(arr[i], 1);

            // If elements already exists in hash map
            else
              tmap.put(arr[i], ++c);
        }

        // Print result
        for (Map.Entry m:tmap.entrySet())
          System.out.println("Frequency of " + m.getKey() +
                             " is " + m.getValue());
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int arr[] = {10, 34, 5, 10, 3, 5, 10};
        printFreq(arr);
    }
}
```

Output:

```
Frequency of 3 is 1
Frequency of 5 is 2
Frequency of 10 is 3
Frequency of 34 is 1
```

Key Points

- For operations like add, remove, containsKey, time complexity is O(log n where n is number of elements present in TreeMap.
- TreeMap always keeps the elements in a sorted(increasing) order, while the elements in a HashMap have no order. TreeMap also provides some cool methods for first, last, floor and ceiling of keys.

Overview:

1. HashMap implements Map interface while TreeMap implements SortedMap interface. A Sorted Map interface is a child of Map.
2. HashMap implements Hashing, while TreeMap implements Red-Black Tree(a Self Balancing Binary Search Tree). Therefore all differences between Hashing and Balanced Binary Search Tree apply here.
3. Both HashMap and TreeMap have their counterparts HashSet and TreeSet. HashSet and TreeSet implement Set interface. In HashSet and TreeSet, we have only key, no value, these are mainly used to see presence/absence in a set. For above problem, we can't use HashSet (or TreeSet) as we can't store counts. An example problem where we would

prefer HashSet (or TreeSet) over HashMap (or TreeMap) is to print all distinct elements in an array.

Array vs ArrayList in Java

In Java, following are two different ways to create an array.

1. **Array:** Simple fixed sized arrays that we create in Java, like below

```
int arr[] = new int[10]
```

2. **ArrayList :** Dynamic sized arrays in Java that implement List interface.

```
ArrayList<Type> arrL = new ArrayList<Type>();

Here Type is the type of elements in ArrayList to
be created
```

Differences between Array and ArrayList

- An array is basic functionality provided by Java. ArrayList is part of collection framework in Java. Therefore array members are accessed using [], while ArrayList has a set of methods to access elements and modify them.

```
// A Java program to demonstrate differences between array
// and ArrayList
import java.util.ArrayList;
import java.util.Arrays;

class Test
{
    public static void main(String args[])
    {
        /* ........... Normal Array............. */
        int[] arr = new int[3];
        arr[0] = 1;
        arr[1] = 2;
        System.out.println(arr[0]);

        /*...........ArrayList.............*/
        // Create an arrayList with initial capacity 2
        ArrayList<Integer> arrL = new ArrayList<Integer>(2);

        // Add elements to ArrayList
        arrL.add(1);
        arrL.add(2);

        // Access elements of ArrayList
```

```
            System.out.println(arrL.get(0));
        }
    }
```

- Output:

```
1
1
```

.

- Array is a fixed size data structure while ArrayList is not. One need not to mention the size of Arraylist while creating its object. Even if we specify some initial capacity, we can add more elements.

```
// A Java program to demonstrate differences between array
// and ArrayList
import java.util.ArrayList;
import java.util.Arrays;
class Test
{
    public static void main(String args[])
    {
        /* ........... Normal Array............. */
        // Need to specify the size for array
        int[] arr = new int[3];
        arr[0] = 1;
        arr[1] = 2;
        arr[2] = 3;
        // We cannot add more elements to array arr[]

        /*............ArrayList.............*/
        // Need not to specify size
        ArrayList<Integer> arrL = new ArrayList<Integer>();
        arrL.add(1);
        arrL.add(2);
        arrL.add(3);
        arrL.add(4);
        // We can add more elements to arrL

        System.out.println(arrL);
        System.out.println(Arrays.toString(arr));
    }
}
```

- Output:

```
[1, 2, 3, 4]
[1, 2, 3]
```

.

- Array can contain both primitive data types as well as objects of a class depending on the definition of the array. However, ArrayList only supports object entries, not the primitive data types.

Note: When we do arraylist.add(1); : it converts the primitive int data type into an Integer object.
Sample Code:

```
import java.util.ArrayList;
class Test
{
    public static void main(String args[])
    {
       // allowed
        int[] array = new int[3];

        // allowed, however, need to be intialized
        Test[] array1 = new Test[3];

        // not allowed (Uncommenting below line causes
        // compiler error)
        // ArrayList<char> arrL = new ArrayList<char>();

        // Allowed
        ArrayList<Integer> arrL1 = new ArrayList<>();
        ArrayList<String> arrL2 = new ArrayList<>();
        ArrayList<object> arrL3 = new ArrayList<>();
    }
}
```

-

  .

- Since ArrayList can't be created for primitive data types, members of ArrayList are always references to objects at different memory locations (See this for details). Therefore in ArrayList, the actual objects are never stored at contiguous locations. References of the actual objects are stored at contiguous locations.
  In array, it depends whether the arrays is of primitive type or object type. In case of primitive types, actual values are contiguous locations, but in case of objects, allocation is similar to ArrayList.

As a side note, ArrayList in Java can be seen as similar to vector in C++.

PriorityQueue Class in Java

To process the objects in the queue based on the priority, we tend to use Priority Queue.

Important points about Priority Queue:

- PriorityQueue doesn't allow **null**
- We can't create PriorityQueue of Objects that are non-comparable
- The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
- The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements — ties are broken arbitrarily.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
- It inherits methods from AbstractQueue, AbstractCollection, Collection and Object class.

Constructor: PriorityQueue()

This creates a PriorityQueue with the default initial capacity that orders its elements according to their natural ordering.

Methods:

1. booleanadd(E element): This method inserts the specified element into this priority queue.

2. public remove(): This method removes a single instance of the specified element from this queue, if it is present

3. public poll(): This method retrieves and removes the head of this queue, or returns null if this queue is empty.

4. public peek(): This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

5. iterator(): Returns an iterator over the elements in this queue.

6. booleancontains(Object o): This method returns true if this queue contains the specified element

Sample code Snippet to show usage of Priority Queue Class:

```
// Java progrm to demonstrate working of priority queue in Java
import java.util.*;

class Example
{
    public static void main(String args[])
    {
        // Creating empty priority queue
        PriorityQueue<String> pQueue =
                           new PriorityQueue<String>();

        // Adding items to the pQueue
```

```
        pQueue.add("C");
        pQueue.add("C++");
        pQueue.add("Java");
        pQueue.add("Python");

        // Printing the most priority element
        System.out.println("Head value using peek function:"
                                           + pQueue.peek());

        // Printing all elements
        System.out.println("The queue elements:");
        Iterator itr = pQueue.iterator();
        while (itr.hasNext())
            System.out.println(itr.next());

        // Removing the top priority element (or head) and
        // printing the modified pQueue
        pQueue.poll();
        System.out.println("After removing an element" +
                           "with poll function:");
        Iterator<String> itr2 = pQueue.iterator();
        while (itr2.hasNext())
            System.out.println(itr2.next());

        // Removing Java
        pQueue.remove("Java");
        System.out.println("after removing Java with" +
                           " remove function:");
        Iterator<String> itr3 = pQueue.iterator();
        while (itr3.hasNext())
            System.out.println(itr3.next());

        // Check if an element is present
        boolean b = pQueue.contains("C");
        System.out.println ( "Priority queue contains C" +
                             "ot not?: " + b);

        // get objects from the queue in an array and
        // print the array
        Object[] arr = pQueue.toArray();
        System.out.println ( "Value in array: ");
        for (int i = 0; i<arr.length; i++)
          System.out.println ( "Value: " + arr[i].toString()) ;
    }
}
```

Output:

```
Head value using peek function:C
The queue elements:
C
C++
Java
Python
After removing an elementwith poll function:
```

```
C++
Python
Java
after removing Java with remove function:
C++
Python
Priority queue contains Cot not?: false
Value in array:
Value: C++
Value: Python
```

Applications :
Implementing Dijkstra's and Prim's algorithms.