

a. Discuss 3 OOPs Principles?

1. Encapsulation:

Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. Typically, only the object's own methods can directly inspect or manipulate its fields.

Encapsulation is the hiding of data implementation by restricting access to accessors and mutators.

An accessor is a method that is used to ask an object about itself. In OOP, these are usually in the form of properties, which have a get method, which is an accessor method. However, accessor methods are not restricted to properties and can be any public method that gives information about the state of the object.

A *Mutator* is public method that is used to modify the state of an object, while hiding the implementation of exactly how the data gets modified. It's the set method that lets the caller modify the member data behind the scenes.

Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. This type of data protection and implementation protection is called Encapsulation.

A benefit of encapsulation is that it can reduce system complexity.

2. Abstraction

Data abstraction and encapsulation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details. Abstraction denotes a model, a view, or some other focused representation for an actual item.

“An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.” — G. Booch

In short, data abstraction is nothing more than the implementation of an object that contains the same essential properties and actions we can find in the original object we are representing.

3. Inheritance

Inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support. In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, superclasses, parent classes or ancestor classes. The resulting classes are known as derived classes, subclasses or child classes. The relationships of classes through inheritance gives rise to a hierarchy.

Subclasses and Superclasses A subclass is a modular, derivative class that inherits one or more properties from another class (called the superclass). The properties commonly include class data variables, properties, and methods or functions. The superclass establishes a common interface and foundational functionality, which specialized subclasses can inherit, modify, and supplement. The software inherited by a subclass is considered reused in the subclass. In some cases, a subclass may customize or redefine a method inherited from the superclass. A superclass method which can be redefined in this way is called a virtual method.

b. Explain Java Application Development Step and JVM?

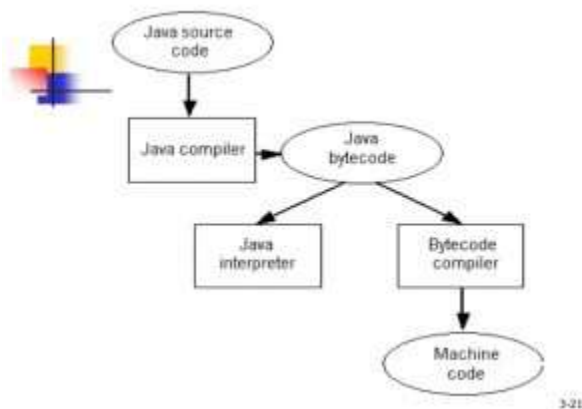
The meaning of platform independent is that, the java source code can run on all operating systems.

A program is written in a language which is a human readable language. It may contain words, phrases etc which the

machine does not understand. For the source code to be understood by the machine, it needs to be in a language understood by machines, typically a machine-level language. So, here comes the role of a compiler. The compiler converts the high-level language (human language) into a format understood by the machines. Therefore, a compiler is a program that translates the source code for another program from a programming language into executable code. This executable code may be a sequence of machine instructions that can be executed by the CPU directly, or it may be an intermediate representation that is interpreted by a virtual machine. This intermediate representation in Java is the Java Byte Code.

Step by step Execution of Java Program:

- Whenever, a program is written in JAVA, the javac compiles it.
- The result of the JAVA compiler is the .class file or the bytecode and not the machine native code (unlike C compiler).
- The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.
- And finally program runs to give the desired output.



In case of C or C++ (language that are not platform independent), the compiler generates an .exe file which is OS dependent. When we try to run this .exe file on another OS it does not run, since it is OS dependent and hence is not compatible with the other OS.

Java is platform independent but JVM is platform dependent

In Java, the main point here is that the JVM depends on the operating system – so if you are running Mac OS X you will have a different JVM than if you are running Windows or some other operating system. This fact can be verified by trying to download the JVM for your particular machine – when trying to download it, you will be given a list of JVM's corresponding to different operating systems, and you will obviously pick whichever JVM is targeted for the operating system that you are running. So we can conclude that JVM is platform dependent and it is the reason why Java is able to become “Platform Independent”.

Important Points:

- In the case of Java, it is the magic of Bytecode that makes it platform independent.
- This adds to an important feature in the JAVA language termed as portability. Every system has its own JVM which gets installed automatically when the jdk software is installed. For every operating system separate JVM is available which is capable to read the .class file or byte code.
- An important point to be noted is that while JAVA is platform-independent language, the JVM is platform-

dependent. Different JVM is designed for different OS and byte code is able to run on different OS.

1.b Find the sum of n numbers using for each statement?

```
class SumOfNumbers
{
    public static void main(String arg[])
    {
        int n = 5;
        int sum = 0;
        int input[];
        for(int i=1;i<=n;i++)
        input[i]=i;
        for(int x: input)
        {
            sum = sum + x; // LINE A
        }

        System.out.println("Sum of numbers till " + input + " is " + sum); // LINE B
    }
}
```

C. Explain Different Access Specifiers?

There are 4 types of java access modifiers: 1. private 2. default 3. protected 4. public There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

2.

a. Explain the scope and life time of a variables with example?

Scope and Lifetime of Variables

The scope of a variable defines the section of the code in which the variable is visible. As a general rule, variables that are defined within a block are not accessible outside that block. The lifetime of a variable refers to how long the variable exists before it is destroyed. Destroying variables refers to deallocating the memory that was allotted to the variables when declaring it. We have written a few classes till now. You might have observed that not all variables are the same. The ones declared in the body of a method were different from those that were declared in the class itself. There are three types of variables: instance variables, formal parameters or local variables and local variables.

```
class Sample
```

```
{
```

```
    int x, y; //instance variables
    static int result;
    void add(int a, int b) //a and b are local variables
    {
        x = a;
        y = b;
        int sum = x+y; //Sum
        System.out.println("Sum = "+sum);
    }
```

Scope of
x and y

```
    public static void main(String[] args)
```

```
    {
```

```
        Sample obj = new Sample();
        obj.add(10,20);
```

```
    }
```

```
}
```

Startertutorials.com

Summary of scope and lifetime of variables

Variable Type	Scope	Lifetime
Instance variable	Throughout the class except in static methods	Until the object is available in the memory
Class variable	Throughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Startertutorials.com

b. What is narrowing and Widening explain with example?

Type conversion in Java with Examples

When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. For example, assigning an int value to a long variable.

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.

Byte → Short → Int → Long → Float → Double

Widening or Automatic Conversion

Example:

```
class Test
{
    public static void main(String[] args)
    {
        int i = 100;

        //automatic type conversion
        long l = i;

        //automatic type conversion
        float f = l;
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

Output:

```
Int value 100
Long value 100
Float value 100.0
```

Narrowing or Explicit Conversion

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

Double → Float → Long → Int → Short → Byte

Narrowing or Explicit Conversion

char and number are not compatible with each other. Let's see when we try to convert one into other.

```
//Java program to illustrate incompatible data
// type for explicit type conversion
public class Test
{
    public static void main(String[] argv)
    {
        char ch = 'c';
        int num = 88;
        ch = num;
    }
}
```

Error:

7: error: incompatible types: possible lossy conversion from int to char

```
    ch = num;
      ^
```

1 error

How to do Explicit Conversion?

Example:

```
//Java program to illustrate explicit type conversion
class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;

        //explicit type casting
        long l = (long)d;

        //explicit type casting
        int i = (int)l;
        System.out.println("Double value "+d);

        //fractional part lost
        System.out.println("Long value "+l);

        //fractional part lost
        System.out.println("Int value "+i);
    }
}
```

Output:

Double value 100.04

Long value 100
Int value 100

While assigning value to byte type the fractional part is lost and is reduced to modulo 256(range of byte).

Example:

```
//Java program to illustrate Conversion of int and double to byte
class Test
{
    public static void main(String args[])
    {
        byte b;
        int i = 257;
        double d = 323.142;
        System.out.println("Conversion of int to byte.");

        //i%256
        b = (byte) i;
        System.out.println("i = b " + i + " b = " + b);
        System.out.println("\nConversion of double to byte.");

        //d%256
        b = (byte) d;
        System.out.println("d = " + d + " b = " + b);
    }
}
```

Output:

Conversion of int to byte.
i = 257 b = 1

Conversion of double to byte.
d = 323.142 b = 67

Type promotion in Expressions

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

1. Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
2. If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

Example:

```
//Java program to illustrate Type promotion in Expressions
class Test
{
    public static void main(String args[])
    {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
    }
}
```

```

double d = .1234;

// The Expression
double result = (f * b) + (i / c) - (d * s);

//Result after all the promotions are done
System.out.println("result = " + result);
    }
}

```

Output:

Result = 626.7784146484375

C .How array in java work differently than c/c++.Write a java program to display

0 1 2 3 4

5 6 7 8 9

10 11 12 13 14

15 16 17 18 19

In C++, when we declare an array, storage for the array is allocated. In Java, when we declare an array, we really only declare a pointer or reference to an array; storage for the array itself is not allocated until we use the "new" keyword. This difference is elaborated below:

C++

```

int A[10]; // A is an array of length 10
A[0] = 5; // set the 1st element of array A

```

JAVA

```

int [ ] A; // A is a reference / pointer to an array
A = new int [10]; // now A points to an array of length 10
A[0] = 5; // set the 1st element of the array pointed to by A

```

In both C++ and Java we can initialize an array using values in curly braces.

```

class Test
{
    public static void main(String args[])
    {
        int b = 0,i,j;
        for(j=0;j<5;j++)
        for(i=0;i<5;i++)
            System.out.print(b++);
        System.out.println();
    }
}

```


3.a.Explain the following operator with example.

i. Logical operator

A *logical operator* (sometimes called a “Boolean operator”) in Java programming is an operator that returns a Boolean result that’s based on the Boolean result of one or two other expressions.

ometimes, expressions that use logical operators are called “compound expressions” because the effect of the logical operators is to let you combine two or more condition tests into a single expression.

Operator	Name	Type	Description
!	Not	Unary	Returns true if the operand to the right evaluates to false. Returns false if the operand to the right is true.
&	And	Binary	Returns true if both of the operands evaluate to true. Both operands are evaluated before the And operator is applied.
	Or	Binary	Returns true if at least one of the operands evaluates to true. Both operands are evaluated before the Or operator is applied.
^	Xor	Binary	Returns true if one — and only one — of the operands evaluates to true. Returns false if both operands evaluate to true or if both operands evaluate to false.
&&	Conditional And	Binary	Same as &, but if the operand on the left returns false, it returns false without evaluating the operand on the right.
	Conditional Or	Binary	Same as , but if the operand on the left returns true, it returns true without evaluating the operand on the right.

b.Java Bitwise

Bitwise and bit shift operators are used on integral types (byte, short, int and long) to perform bit-level operations.

These operators are not commonly used. You will learn about a few use cases of bitwise operators in *Java enum type* chapter. This article will only focus on how these operators work.

There are 7 operators to perform bit-level operations in Java (4 bitwise and 3 bit shift).

Java Bitwise and Bit Shift Operators

Operator	Description
	Bitwise OR
&	Bitwise AND
~	Bitwise Complement
^	Bitwise XOR
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift

Bitwise OR

Bitwise OR is a binary operator (operates on two operands). It's denoted by |.

The | operator compares corresponding bits of two operands. If either of the bits is 1, it gives 1. If not, it gives 0. For example,

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

00001100

| 00011001

00011101 = 29 (In decimal)

Example 1: Bitwise OR

```
class BitwiseOR {  
    public static void main(String[] args) {  
  
        int number1 = 12, number2 = 25, result;  
  
        result = number1 | number2;  
        System.out.println(result);  
    }  
}
```

When you run the program, the output will be:

29

b. With Example explain about ternary operator.?find largest of 3 numbers using ternary operator?

At its most basic, the ternary operator, also known as the *conditional operator*, can be used as an alternative to the Java if/then/else syntax, but it goes beyond that, and can even be used on the right hand side of Java statements.

One use of the Java ternary operator is to assign the minimum (or maximum) value of two variables to a third variable, essentially replacing a `Math.min(a,b)` or `Math.max(a,b)` method call. Here's an example that assigns the minimum of two variables, a and b, to a third variable named `minVal`:

```
minVal = (a < b) ? a : b;
```

In this code, if the variable a is less than b, `minVal` is assigned the value of a; otherwise, `minVal` is assigned the value of b.

Enter any three integer numbers as an input from which you want to find the largest integer. After that we use ternary operator(?, >, :) to find largest number as the output.

Here is the source code of the Java Program to Find Largest Between Three Numbers Using Ternary Operator. The Java program is successfully compiled and run on a Windows system. The program output is also shown below.

```
1. import java.util.Scanner;
2. public class Largest_Ternary
3. {
4.     public static void main(String[] args)
5.     {
6.         int a, b, c, d;
7.         Scanner s = new Scanner(System.in);
8.         System.out.println("Enter all three numbers:");
9.         a = s.nextInt();
10.        b = s.nextInt();
11.        c = s.nextInt();
```

1. `big = a > b ? (a > c ? a : c) : (b > c ? b : c);`

```
12. ;
13.     System.out.println("Largest Number:"+big);
14. }
15. }
```

c. Explain for each version for loop? Write a program to display 2D student data:Name and USN?

The for-each loop introduced in Java5. It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

Advantage of for-each loop:

- It makes the code more readable.
- It eliminates the possibility of programming errors.

Syntax of for-each loop:

1. `for(data_type variable : array | collection){}`

```
class PrintStudentDetailsUsingClasses
{
    public static void main(String s[])
    {
        Student students[] = new Student[5];

        students[0] = new Student();
        students[0].name = "Rajesh";
        students[0].usn = 45;
```

```
students[1] = new Student();
students[1].name = "Suresh";
students[1].usn = 78;
```

```
students[2] = new Student();
students[2].name = "Ramesh";
students[2].usn = 83;
```

```
for(Student i : students)
{
    System.out.println(" name= " + i.name + " with USN= " + i.usn);
}
```

```
}
```

```
class Student
{
    String name;
    int usn;
}
```

4 a.

Demonstrate use of

- Continue in while loop
- break in do while loop?

- Consider this example:

```
while ( i != 3 ) // continue jumps here
{
    if ( i > num )
        continue ; // Jumps to condition, i != 3
    System.out.println( "Here I am!" );
}
```

If the continue statement runs, then it causes control flow to jump to the condition of the enclosing while loop.

○

```
int i=0;
```

```
# do
{
    bool test1success = false;
    if (test1success==true)
```

```

    {
        i=0;
        break; // move on to next iteration
    }
System.out.println(i++);
if (i==8)
{
    test1success==true; }
}
while (i<9);

```

b. Write a java program to find the prime numbers between from 1 to 100.

```

class PrimeNumbers
{
    public static void main (String[] args)
    {
        int i =0;
        int num =0;
        //Empty String
        String primeNumbers = "";

        for (i = 1; i <= 100; i++)
        {
            int counter=0;
            for(num =i; num>=1; num--)
            {
                if(i%num==0)
                {
                    counter = counter + 1;
                }
            }
            if (counter ==2)
            {
                //Appended the Prime number to the String
                primeNumbers = primeNumbers + i + " ";
            }
        }
        System.out.println("Prime numbers from 1 to 100 are :");
        System.out.println(primeNumbers);
    }
}

```

Output:

```

Prime numbers from 1 to 100 are :
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

```

c. Write a java program to perform simple calculator operations?

```

import java.util.Scanner;

```

```
public class Calculator {

    public static void main(String[] args) {

        Scanner reader = new Scanner(System.in);
        System.out.print("Enter two numbers: ");

        // nextDouble() reads the next double from the keyboard
        double first = reader.nextDouble();
        double second = reader.nextDouble();

        System.out.print("Enter an operator (+, -, *, /): ");
        char operator = reader.next().charAt(0);

        double result;

        switch(operator)
        {
            case '+':
                result = first + second;
                break;

            case '-':
                result = first - second;
                break;

            case '*':
                result = first * second;
                break;

            case '/':
                result = first / second;
                break;

            // operator doesn't match any case constant (+, -, *, /)
            default:
                System.out.printf("Error! operator is not correct");
                return;
        }
    }
}
```

```

    }

    System.out.printf("%.1f%c %.1f = %.1f", first, operator, second, result);
}
}

```

5.

- a. What are the salient features of constructor? Write a java program to show this?

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```

1. class Bike1 {
2. Bike1(){System.out.println("Bike is created");}
3. public static void main(String args[]){
4. Bike1 b=new Bike1();
5. }
6. }

```

b.

1. this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.

2. Garbage collection
 - this can be used to return the current class instance from the method

Whenever you run a java program, JVM creates three threads. 1) main thread 2) Thread Scheduler 3) Garbage Collector Thread. In these three threads, main thread is a user thread and remaining two are daemon threads which run in background.

The task of main thread is to execute the main() method. The task of thread scheduler is to schedule the threads. The task of garbage collector thread is to sweep out abandoned objects from the heap memory. Abandoned objects or dead objects are those objects which does not have live references. Garbage collector thread before sweeping out an abandoned object, it calls finalize() method of that object. After finalize() method is executed, object is destroyed from the memory. That means clean up operations which you have kept in the finalize() method are executed before an object is destroyed from the memory.

Garbage collector thread does not come to heap memory whenever an object becomes abandoned. It comes once in a while to the heap memory and at that time if it sees any abandoned objects, it sweeps out those objects after calling finalize() method on them. Garbage collector thread calls finalize() method only once for one object.

3. Finalize()

finalize() method is a protected and non-static method of java.lang.Object class. This method will be available in all objects you create in java. This method is used to perform some final operations or clean up operations on an object before it is removed from the memory. you can override the finalize() method to keep those operations you want to perform before an object is destroyed. Here is the general form of finalize() method.?

```
protected void finalize() throws Throwable
{
    //Keep some resource closing operations here
}
```

C With Java program show how final Keyword is used to prevent inheritance and overriding?

When a class is declared as final then it cannot be subclassed i.e. no any other class can extend it. This is particularly useful, for example, when creating an immutable class like the predefined String class. The following fragment illustrates final keyword with a class:

```
final class A
{
    // methods and fields
}
// The following class is illegal.
class B extends A
{
    // ERROR! Can't subclass A
}
```

Note :

- Declaring a class as final implicitly declares all of its methods as final, too.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. For more on abstract classes, refer abstract classes in java

Using final to Prevent Overriding

When a method is declared as final then it cannot be overridden by subclasses. The Object class does this—a number of its methods are final. The following fragment illustrates final keyword with a method:

```
class A
{
    final void m1()
    {
        System.out.println("This is a final method.");
    }
}

class B extends A
{
    void m1()
    {
        // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

Normally, Java resolves calls to methods dynamically, at run time. This is called late or dynamic binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early or static binding.

6. Create a Java class called Student with the following details as variables within it.

- (i) USN
- (ii) Name
- (iii) Branch
- (iv) Phone

Write a Java program to create nStudent objects and print the USN, Name, Branch, and Phone of these objects with suitable messages

```
import java.util.Scanner;
import java.io.*;
public class student
{ String USN,name , branch;
  int phoneno;
  public void getinfo() throws Exception
  { InputStreamReader r=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(r);
    System.out.println("Enter USN");
    USN=br.readLine();
    System.out.println("Enter Name");
    name=br.readLine();
    System.out.println("Enter Branch");
    branch=br.readLine();
    Scanner integer=new Scanner(System.in);
    System.out.println("Enter phone number");
```

```

phoneno=integer.nextInt();
}
public void display()
{ System.out.printf("%s\t%s\t%s\t%d\n",USN,name,branch,phoneno);
}
public static void main(String[] args) throws Exception
{ int n,i;
Scanner integer=new Scanner(System.in);
System.out.println("Enter Number of student");
n=integer.nextInt();
//declare object with array
student[] obj=new student[n];
//object creation
for(i=0;i<n;i++)
{ obj[i]=new student();
System.out.printf("Student : %d\n",i+1);
obj[i].getinfo();
}
System.out.println("USN\tName\tBranch\tPhone Number");
for(i=0;i<n;i++)
obj[i].display();
}
}

```

b. difference between c++ and java with respect to inheritance.Mention the use of super and this parameter in java with example?

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));
    }
}

```

Run on IDE

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);
    }
}

```

Run on IDE

4) Java uses *extends* keyword for inheritance. 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 separate 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");
    }
}

```

Run on IDE

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");
}
}

```

Run on IDE

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);
    }
}

```

Run on IDE

Output:

```

Base constructor called
Derived constructor called

```

c. What is innerclass demonstrate with example?

non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

1. class Outer {
2. //code

```
3. class Inner {
4. //code
5. }
6. }
```

Java Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
1. class TestMemberOuter1 {
2. private int data=30;
3. class Inner {
4. void msg(){System.out.println("data is "+data);}
5. }
6. public static void main(String args[]){
7. TestMemberOuter1 obj=new TestMemberOuter1();
8. TestMemberOuter1.Inner in=obj.new Inner();
9. in.msg();
10. }
11. }
```

Output:

```
data is 30
```

7 a. Define interface? explain how to define implement assign variables in interface to perform "one interface multiple method"?

Java includes a concept called *interfaces*. A Java interface is a bit like a class, except a Java interface can only contain method signatures and fields. An Java interface cannot contain an implementation of the methods, only the signature (name, parameters and exceptions) of the method.

Implementing an Interface

Before you can really use an interface, you must implement that interface in some Java class. Here is a class that implements the MyInterface interface shown above:

```
public class MyInterfaceImpl implements MyInterface {
    public void sayHello() {
        System.out.println(MyInterface.hello);
    }
}
```

Notice the implements MyInterface part of the above class declaration. This signals to the Java compiler that the MyInterfaceImpl class implements the MyInterface interface.

A class that implements an interface must implement all the methods declared in the interface. The methods must have the exact same signature (name + parameters) as declared in the interface. The class does not need to implement (declare) the variables of an interface. Only the methods.

Interface Instances

Once a Java class implements an Java interface you can use an instance of that class as an instance of that interface. Here is an example:

```
MyInterface myInterface = new MyInterfaceImpl();  
  
myInterface.sayHello();
```

Notice how the variable is declared to be of the interface type MyInterface while the object created is of type MyInterfaceImpl. Java allows this because the class MyInterfaceImpl implements the MyInterfaceinterface. You can then reference instances of the MyInterfaceImpl class as instances of the MyInterfaceinterface.

You cannot create instances of a Java interface by itself. You must always create an instance of some class that implements the interface, and reference that instance as an instance of the interface.

```
import java.io.*;  
  
interface Vehicle {  
  
    // all are the abstract methods.  
    void changeGear(int a);  
    void speedUp(int a);  
    void applyBrakes(int a);  
}  
  
class Bicycle implements Vehicle{  
  
    int speed;  
    int gear;  
  
    // to change gear  
    @Override  
    public void changeGear(int newGear){  
  
        gear = newGear;  
    }  
  
    // to increase speed  
    @Override  
    public void speedUp(int increment){  
  
        speed = speed + increment;  
    }  
  
    // to decrease speed  
    @Override  
    public void applyBrakes(int decrement){  
  
        speed = speed - decrement;
```

```

    }

    public void printStates() {
        System.out.println("speed: " + speed
            + " gear: " + gear);
    }
}

class Bike implements Vehicle {

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed
            + " gear: " + gear);
    }
}

class GFG {

    public static void main (String[] args) {

        // creating an inatnce of Bicycle
        // doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");
        bicycle.printStates();

        // creating instance of bike.
        Bike bike = new Bike();
    }
}

```



```
bike.changeGear(1);
bike.speedUp(4);
bike.applyBrakes(3);
```

```
System.out.println("Bike present state :");
bike.printStates();
}
```

```
}
```

- b. Role of interface in multiple inheritance?
- c. Inheritance is when an object or class is based on another object or class, using the same implementation specifying implementation to maintain the same behavior. It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a hierarchy. Multiple Inheritance allows a class to have more than one super class and to inherit features from all parent class. it is achieved using interface.

```
d. interface vehicleone{
e.     int speed=90;
f.     public void distance();
g. }
h.
i. interface vehicletwo{
j.     int distance=100;
k.     public void speed();
l. }
m.
n. class Vehicle implements vehicleone,vehicletwo{
o.     public void distance(){
p.         int distance=speed*100;
q.         System.out.println("distance travelled is "+distance);
r.     }
s.     public void speed(){
t.         int speed=distance/100;
u.     }
v. }
w.
x. class MultipleInheritanceUsingInterface{
```

```
y.      public static void main(String args[]){
z.          System.out.println("Vehicle");
aa.          obj.distance();
bb.          obj.speed();
cc.      }
dd. }
```

c.short note on

1.importing packages

2.access protection

1. import keyword

import keyword is used to import built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to any class that is present in a different package:

1. Using fully qualified name (But this is not a good practice.)

If you use fully qualified name to import any class into your program, then only that particular class of the package will be accessible in your program, other classes in the same package will not be accessible. For this approach, there is no need to use the import statement. But you will have to use the fully qualified name every time you are accessing the class or the interface, which can look a little untidy if the package name is long.

This is generally used when two packages have classes with same names. For example: java.util and java.sql packages contain Date class.

Example :

```
//save by A.java
package pack;
public class A {
    public void msg() {
        System.out.println("Hello");
    }
}

//save by B.java
package mypack;
class B {
    public static void main(String args[]) {
```

```
pack.A obj = new pack.A(); //using fully qualified name
obj.msg();
}
}
```

Output:

Hello

2. To import only the class/classes you want to use

If you import `packagename.classname` then only the class with name `classname` in the package with name `packagename` will be available for use.

Example :

```
//save by A.java
package pack;
public class A {
    public void msg() {
        System.out.println("Hello");
    }
}

//save by B.java
package mypack;
import pack.A;
class B {
    public static void main(String args[]) {
        A obj = new A();
        obj.msg();
    }
}
```

Output:

Hello

3. To import all the classes from a particular package

If you use `packagename.*`, then all the classes and interfaces of this package will be accessible but the classes and interface inside the subpackages will not be available for use.

The `import` keyword is used to make the classes and interface of another package accessible to the current package.

Example :

```
//save by First.java
package learnjava;
public class First {
    public void msg() {
        System.out.println("Hello");
    }
}

//save by Second.java
package Java;
import learnjava.*;
class Second {
    public static void main(String args[]) {
        First obj = new First();
        obj.msg();
    }
}
```

2.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

8.a.Explain exception Demonstrate working of nested try with example?

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
1. ....
2. try
3. {
4.     statement 1;
5.     statement 2;
6.     try
7.     {
8.         statement 1;
9.         statement 2;
10.    }
11. catch(Exception e)
12. {
13. }
14. }
15. catch(Exception e)
16. {
17. }
18. ....
```

Java nested try example

Let's see a simple example of java nested try block.

```
1. class Excep6{
2.     public static void main(String args[]){
3.         try{
4.             try{
5.                 System.out.println("going to divide");
6.                 int b =39/0;
7.             }catch(ArithmeticException e){System.out.println(e);}
8.
9.             try{
10.                int a[]=new int[5];
11.                a[5]=4;
12.            }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
13.
14.            System.out.println("other statement");
15.        }catch(Exception e){System.out.println("handed");}
16.
17.        System.out.println("normal flow..");
18.    }
19. }
```

b. write a program w which will throw illegalacessexception and use proper exception should be printed?

```

public class ClassNewInstanceExample {
    public static void main(String[] args) throws
        InstantiationException {
        Class<?> classVar = ExampleClass.class;
        ExampleClass t = (ExampleClass) classVar.newInstance();
    try{
        t.testMethod();
    }
    Catch(IllegalAccessException e)
    System.out.println("exception caught");
    }
}
ExampleClass.java

```

```

package javabeat.net.lang;

```

```

public class ExampleClass {
    private ExampleClass(){
    }
    public void testMethod(){
        System.out.println("Method 'testMethod' Called");
    }
}

```

Advertisement

c.Java's builtin Exception

Java defines several exception classes inside the standard package java.lang.

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

Sr.No.	Exception & Description
1	<p>ArithmeticException</p> <p>Arithmetic error, such as divide-by-zero.</p>
2	<p>ArrayIndexOutOfBoundsException</p> <p>Array index is out-of-bounds.</p>
3	<p>ArrayStoreException</p> <p>Assignment to an array element of an incompatible type.</p>
4	<p>ClassCastException</p>

	Invalid cast.
5	IllegalArgumentException Illegal argument used to invoke a method.
6	IllegalMonitorStateException Illegal monitor operation, such as waiting on an unlocked thread.
7	IllegalStateException Environment or application is in incorrect state.
8	IllegalThreadStateException Requested operation not compatible with the current thread state.
9	IndexOutOfBoundsException Some type of index is out-of-bounds.
10	NegativeArraySizeException Array created with a negative size.
11	NullPointerException Invalid use of a null reference.
12	NumberFormatException Invalid conversion of a string to a numeric format.
13	SecurityException Attempt to violate security.
14	StringIndexOutOfBoundsException Attempt to index outside the bounds of a string.
15	UnsupportedOperationException An unsupported operation was encountered.

Following is the list of Java Checked Exceptions Defined in java.lang.

Sr.No.	Exception & Description
--------	-------------------------

1	ClassNotFoundException Class not found.
2	CloneNotSupportedException Attempt to clone an object that does not implement the Cloneable interface.
3	IllegalAccessException Access to a class is denied.
4	InstantiationException Attempt to create an object of an abstract class or interface.
5	InterruptedException One thread has been interrupted by another thread.
6	NoSuchFieldException A requested field does not exist.
7	NoSuchMethodException A requested method does not exist.

Un caught Exception

Java actually handles uncaught exceptions according to the thread in which they occur. When an uncaught exception occurs in a particular thread, Java looks for what is called an uncaught exception handler, actually an implementation of the interface `UncaughtExceptionHandler`. The latter interface has a method `handleException()`, which the implementer overrides to take appropriate action, such as printing the stack trace to the console. As we'll see in a moment, we can actually install our own instance of `UncaughtExceptionHandler` to handle uncaught exceptions of a particular thread, or even for the whole system.

The specific procedure is as follows. When an uncaught exception occurs, the JVM does the following:

- it calls a special private method, `dispatchUncaughtException()`, on the `Thread` class in which the exception occurs;
- it then terminates the thread in which the exception occurred¹.

The `dispatchUncaughtException` method, in turn, calls the thread's `getUncaughtExceptionHandler()` method to find out the appropriate uncaught exception handler to use. Normally, this will actually be the thread's parent `ThreadGroup`, whose `handleException()` method by default will print the stack trace.

However, we can actually override this process for an individual thread, for a `ThreadGroup`, or for all threads.

9. a), What is an Applet? Explain different stages in Applet lifecycle?

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

There are many advantages of applet. They are as follows:

- o It works at client side so less response time.
- o Secured
- o It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- o Plugin is required at client browser to execute applet.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. public void init(): is used to initialize the Applet. It is invoked only once.
2. public void start(): is invoked after the init() method or browser is maximized. It is used to start the Applet.

java.awt.Component class

The Component class provides 1 life cycle method of applet.

3. public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.
4. public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
5. public void destroy(): is used to destroy the Applet. It is invoked only once.

1 b). Explain applet with the help of eg?

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser

and works at client side.

Advantage of Applet

There are many advantages of applet. They are as follows:

- o It works at client side so less response time.
- o Secured
- o It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- o Plugin is required at client browser to execute applet.

Sample program

```
o //First.java
o import java.applet.Applet;
o import java.awt.Graphics;
o public class First extends Applet{
o
o public void paint(Graphics g){
o g.drawString("welcome to applet",150,150);
o }
o
o }
o /*
o <applet code="First.class" width="300" height="300">
o </applet>
o */
```

To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java
c:\>appletviewer First.java
```

b.Type wrappers

Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object (with `java.io.File`), an address of a system can be seen as an object (with `java.util.URL`), an image can be treated as an object (with `java.awt.Image`) and a simple data type can be converted into an object (with wrapper classes). This tutorial discusses wrapper classes. Wrapper classes are used to convert any data type into an object.

As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used. Wrapper classes include methods to unwrap the object and give back the data type. It can be compared with a chocolate. The manufacturer wraps the chocolate with some foil or paper to prevent from pollution. The user takes the chocolate, removes and throws the wrapper and eats it.

Observe the following conversion.

```
int k = 100;
```

```
Integer it1 = new Integer(k);
```

The int data type k is converted into an object, it1 using Integer class. The it1 object can be used in Java programming wherever k is required an object.

The following code can be used to unwrap (getting back int from Integer object) the object it1.

```
int m = it1.intValue();
```

```
System.out.println(m*m); // prints 10000
```

intValue() is a method of Integer class that returns an int data type.

Transient and volatile Modifiers

The transient modifier tells the Java object serialization subsystem to exclude the field when serializing an instance of the class. When the object is then deserialized, the field will be initialized to the default value; i.e. null for a reference type, and zero or false for a primitive type.

The volatile modifier tells the JVM that writes to the field should always be synchronously flushed to memory, and that reads of the field should always read from memory. This means that fields marked as volatile can be safely accessed and updated in a multi-thread application without using native or standard library-based synchronization.

Similarly, reads and writes to volatile fields are atomic. (This does not apply to >>non-volatile<< long or double fields, which may be subject to "word tearing" on some JVMs.)

Enlist Applet Tag?

```
<APPLET
  [CODEBASE = codebaseURL]
  CODE = appletFile
  [ALT = alternateText]
  [NAME = appletInstanceName]
  WIDTH = pixels
  HEIGHT = pixels
  [ALIGN = alignment]
  [VSPACE = pixels]
  [HSPACE = pixels]
>
[< PARAM NAME = appletParameter1 VALUE = value >]
[< PARAM NAME = appletParameter2 VALUE = value >]
...
[alternateHTML]
</APPLET>
```

CODEBASE = *codebaseURL*

This optional attribute specifies the base URL of the applet -- the directory or folder that contains the applet's code. If this attribute is not specified, then the document's URL is used.

CODE = *appletFile*

This required attribute gives the name of the file that contains the applet's compiled Applet subclass. This file is relative to the base URL of the applet. It cannot be absolute.

ALT = *alternateText*

This optional attribute specifies any text that should be displayed if the browser understands the APPLET tag but can't run Java applets.

NAME = *appletInstanceName*

This optional attribute specifies a name for the applet instance, which makes it possible for applets on the same page to find (and communicate with) each other.

WIDTH = *pixels*

HEIGHT = *pixels*

These required attributes give the initial width and height (in pixels) of the applet display area, not counting any windows or dialogs that the applet brings up.

ALIGN = *alignment*


This required attribute specifies the alignment of the applet. The possible values of this attribute are the same (and have the same effects) as those for the IMG tag: left, right, top, texttop, middle, absmiddle, baseline, bottom, absbottom.

VSPACE = *pixels*

HSPACE = *pixels*

These optional attributes specify the number of pixels above and below the applet (VSPACE) and on each side of the applet (HSPACE). They're treated the same way as the IMG tag's VSPACE and HSPACE attributes.

< PARAM NAME = *appletParameter1* VALUE = *value* >

<PARAM> tags are the only way to specify applet-specific parameters. Applets read user-specified values for parameters with the `getParameter()` method. See [Defining and Using Applet Parameters](#)  for information about the `getParameter()` method.

10. Explain the following with example?

a. String Comparison

b. Searching strings

c. Modifying Strings

d. Overloading constructors?

a.

Concatenating Strings

Concatenating Strings means appending one string to another. Strings in Java are immutable meaning they cannot be changed once created. Therefore, when concatenating two Java String objects to each other, the result is actually put into a third String object.

Here is a Java String concatenation example:

```
String one = "Hello";
String two = "World";

String three = one + " " + two;
```

The content of the String referenced by the variable three will be Hello World; The two other Strings objects are untouched.

String Concatenation Performance

When concatenating Strings you have to watch out for possible performance problems. Concatenating two Strings in Java will be translated by the Java compiler to something like this:

```
String one = "Hello";
String two = " World";

String three = new StringBuilder(one).append(two).toString();
```

As you can see, a new `StringBuilder` is created, passing along the first `String` to its constructor, and the second `String` to its `append()` method, before finally calling the `toString()` method. This code actually creates two objects: A `StringBuilder` instance and a new `String` instance returned from the `toString()` method.

When executed by itself as a single statement, this extra object creation overhead is insignificant. When executed inside a loop, however, it is a different story.

Here is a loop containing the above type of `String` concatenation:

```
String[] strings = new String[]{"one", "two", "three", "four", "five" };

String result = null;
for(String string : strings) {
    result = result + string;
}
```

This code will be compiled into something similar to this:

```
String[] strings = new String[]{"one", "two", "three", "four", "five" };

String result = null;
for(String string : strings) {
    result = new StringBuilder(result).append(string).toString();
}
```

Now, for every iteration in this loop a new `StringBuilder` is created. Additionally, a `String` object is created by the `toString()` method. This results in a small object instantiation overhead per iteration: One `StringBuilder` object and one `String` object. This by itself is not the real performance killer though. But something else related to the creation of these objects is.

Every time the new `StringBuilder(result)` code is executed, the `StringBuilder` constructor copies all characters from the `result` `String` into the `StringBuilder`. The more iterations the loop has, the bigger the `result` `String` grows. The bigger the `result` `String` grows, the longer it takes to copy the characters from it into a new `StringBuilder`, and again copy the characters from the `StringBuilder` into the temporary `String` created by the `toString()` method. In other words, the more iterations the slower each iteration becomes.

The fastest way of concatenating `Strings` is to create a `StringBuilder` once, and reuse the same instance inside the loop. Here is how that looks:

```
String[] strings = new String[]{"one", "two", "three", "four", "five" };
```

```
StringBuilder temp = new StringBuilder();
for(String string : strings) {
    temp.append(string);
}
String result = temp.toString();
```

Searching in Strings With indexOf()

You can search for substrings in Strings using the indexOf() method. Here is an example:

```
String string1 = "Hello World";
int index = string1.indexOf("World");
```

The index variable will contain the value 6 after this code is executed. The indexOf() method returns the index of where the first character in the first matching substring is found. In this case the W of the matched substring World was found at index 6.

If the substring is not found within the string, the indexOf() method returns -1;

There is a version of the indexOf() method that takes an index from which the search is to start. That way you can search through a string to find more than one occurrence of a substring. Here is an example:

```
String theString = "is this good or is this bad?";
String substring = "is";

int index = theString.indexOf(substring);
while(index != -1) {
    System.out.println(index);
    index = theString.indexOf(substring, index + 1);
}
```

This code searches through the string "is this good or is this bad?" for occurrences of the substring "is". It does so using the indexOf(substring, index) method. The index parameter tells what character index in the String to start the search from. In this example the search is to start 1 character after the index where the previous occurrence was found. This makes sure that you do not just keep finding the same occurrence.

The output printed from this code would be:

```
0
5
16
21
```

The substring "is" is found in four places. Two times in the words "is", and two times inside the word "this".

The Java String class also has a lastIndexOf() method which finds the last occurrence of a substring. Here is an example:

```
String theString = "is this good or is this bad?";
String substring = "is";

int index = theString.lastIndexOf(substring);
System.out.println(index);
```

The output printed from this code would be 21 which is the index of the last occurrence of the substring "is".

Matching a String Against a Regular Expression With matches()

The Java String matches() method takes a regular expression as parameter, and returns true if the regular expression matches the string, and false if not.

Here is a matches() example:

```
String text = "one two three two one";
boolean matches = text.matches(".*two.*");
```

Comparing Strings

Java Strings also have a set of methods used to compare Strings. These methods are:

- equals()
- equalsIgnoreCase()
- startsWith()
- endsWith()
- compareTo()

equals()

The equals() method tests if two Strings are exactly equal to each other. If they are, the equals() method returns true. If not, it returns false. Here is an example:

```
String one = "abc";
String two = "def";
String three = "abc";
String four = "ABC";

System.out.println( one.equals(two) );
System.out.println( one.equals(three) );
System.out.println( one.equals(four) );
```

The two strings one and three are equal, but one is not equal to two or to four. The case of the characters must match exactly too, so lowercase characters are not equal to uppercase characters.

The output printed from the code above would be:

```
false
```

```
true  
false
```

`equalsIgnoreCase()`

The `String` class also has a method called `equalsIgnoreCase()` which compares two strings but ignores the case of the characters. Thus, uppercase characters are considered to be equal to their lowercase equivalents.

`startsWith()` and `endsWith()`

The `startsWith()` and `endsWith()` methods check if the `String` starts with a certain substring. Here are a few examples:

```
String one = "This is a good day to code";  
  
System.out.println( one.startsWith("This") );  
System.out.println( one.startsWith("This", 5) );  
  
System.out.println( one.endsWith ("code") );  
System.out.println( one.endsWith ("shower") );
```

This example creates a `String` and checks if it starts and ends with various substrings.

The first line (after the `String` declaration) checks if the `String` starts with the substring "This". Since it does, the `startsWith()` method returns `true`.

The second line checks if the `String` starts with the substring "This" when starting the comparison from the character with index 5. The result is `false`, since the character at index 5 is "i".

The third line checks if the `String` ends with the substring "code". Since it does, the `endsWith()` method returns `true`.

The fourth line checks if the `String` ends with the substring "shower". Since it does not, the `endsWith()` method returns `false`.

`compareTo()`

The `compareTo()` method compares the `String` to another `String` and returns an `int` telling whether this `String` is smaller, equal to or larger than the other `String`. If the `String` is earlier in sorting order than the other `String`, `compareTo()` returns a negative number. If the `String` is equal in sorting order to the other `String`, `compareTo()` returns 0. If the `String` is after the other `String` in sorting order, the `compareTo()` method returns a positive number.

Here is an example:

```
String one = "abc";  
String two = "def";  
String three = "abd";  
  
System.out.println( one.compareTo(two) );  
System.out.println( one.compareTo(three) );
```

This example compares the one String to two other Strings. The output printed from this code would be:

```
-3  
-1
```

The numbers are negative because the one String is earlier in sorting order than the two other Strings.

The compareTo() method actually belongs to the Comparable interface. This interface is described in more detail in my tutorial about Sorting.

c. Trimming Strings With trim()

The Java String class contains a method called trim() which can trim a string object. By *trim* is meant to remove white space characters at the beginning and end of the string. White space characters include space, tab and new lines. Here is a Java String trim() example:

```
String text = " And he ran across the field  ";  
String trimmed = text.trim();
```

After executing this code the trimmed variable will point to a String instance with the value

```
"And he ran across the field"
```

The white space characters at the beginning and end of the String object have been removed. The white space character inside the String have not been touched. By *inside* is meant between the first and last non-white-space character.

The trim() method does not modify the String instance. Instead it returns a new Java String object which is equal to the String object it was created from, but with the white space in the beginning and end of the String removed.

The trim() method can be very useful to trim text typed into input fields by a user. For instance, the user may type in his or her name and accidentally put an extra space after the last word, or before the first word. The trim() method is an easy way to remove such extra white space characters.

Replacing Characters in Strings With replace()

The Java String class contains a method named replace() which can replace characters in a String.

The replace() method does not actually replace characters in the existing String. Rather, it returns a new String instance which is equal to the String instance it was created from, but with the given characters replaced. Here is a Java String replace() example:

```
String source = "123abc";  
String replaced = source.replace('a', '@');
```

After executing this code the replaced variable will point to a String with the text:

```
123@bc
```


The `replace()` method will replace all character matching the character passed as first parameter to the method, with the second character passed as parameter to the `replace()` method.

`replaceFirst()`

The Java String `replaceFirst()` method returns a new String with the first match of the regular expression passed as first parameter with the string value of the second parameter.

Here is a `replaceFirst()` example:

```
String text = "one two three two one";  
String s = text.replaceFirst("two", "five");
```

This example will return the string "one five three two one".

`replaceAll()`

The Java String `replaceAll()` method returns a new String with all matches of the regular expression passed as first parameter with the string value of the second parameter.

Here is a `replaceAll()` example:

```
String text = "one two three two one";  
String t = text.replaceAll("two", "five");
```

This example will return the string "one five three five one".

Splitting Strings With `split()`

The Java String class contains a `split()` method which can be used to split a String into an array of String objects. Here is a Java String `split()` example:

```
String source = "A man drove with a car.";  
String[] occurrences = source.split("a");
```

After executing this Java code the occurrences array would contain the String instances:

```
"A m"  
"n drove with "  
" c"  
".r."
```

The source String has been split on the a characters. The Strings returned do not contain the a characters. The a characters are considered delimiters to split the String by, and the delimiters are not returned in the resulting String array.

The parameter passed to the `split()` method is actually a Java regular expression. Regular expressions can be quite advanced. The regular expression above just matched all a characters. It even only matched lowercase a characters.

The `String split()` method exists in a version that takes a limit as a second parameter. Here is a Java `String split()` example using the limit parameter:

```
String source = "A man drove with a car.";
int limit = 2;
String[] occurrences = source.split("a", limit);
```

The limit parameter sets the maximum number of elements that can be in the returned array. If there are more matches of the regular expression in the `String` than the given limit, then the array will contain `limit - 1` matches, and the last element will be the rest of the `String` from the last of the `limit - 1` matches. So, in the example above the returned array would contain these two `String`s:

```
"A m"
"n drove with a car."
```

The first `String` is a match of the a regular expression. The second `String` is the rest of the `String` after the first match.

Running the example with a limit of 3 instead of 2 would result in these `String`s being returned in the resulting `String` array:

```
"A m"
"n drove with "
" car."
```

Notice how the last `String` still contains the a character in the middle. That is because this `String` represents the rest of the `String` after the last match (the a after 'n drove with ').

Running the example above with a limit of 4 or higher would result in only the `Split` strings being returned, since there are only 4 matches of the regular expression a in the `String`.

Converting Numbers to Strings With `valueOf()`

The Java `String` class contains a set of overloaded static methods named `valueOf()` which can be used to convert a number to a `String`. Here are some simple Java `String valueOf()` examples:

```
String intStr = String.valueOf(10);
System.out.println("intStr = " + intStr);

String flStr = String.valueOf(9.99);
System.out.println("flStr = " + flStr);
```

The output printed from this code would be:

```
intStr = 10
flStr = 9.99
```

d. Like method overloading, Java also supports constructor overloading. Writing multiple constructors, with different parameters, in the same class is known as constructor overloading. Depending upon the parameter list, the appropriate constructor is called when an object is created.

Let us see the Constructor Overloading concept programmatically.

```
public class Student
1  {
2  public Student()           // I, default constructor
3  {
4  System.out.println("Hello 1");
5  }
6  public Student(String name) // II, parameterized constructor with single
7  parameter
8  {
9  System.out.println("Student name is " + name);
10 }
11
12 public Student(String name, int marks) // III, parameterized constructor with
two parameters
13 {
14 System.out.println("Student name is " + name + " and marks are " + marks);
15 }
16
17 public static void main(String args[])
18 {
19 Student std1 = new Student();           // calls I
20 Student std2 = new Student("Mr.Reddy"); // calls II
21 Student std3 = new Student("Mr.Raju", 56); // calls III
22 }
23 }
```

