Socket Programming:

Network: Interconnection between any two systems with in local or remote is called network.

Types of network:

1) LAN
2) MAN
3) WAN

LAN: Stands for Local Area Network.

The systems which are connected with in the organization level or building level is called LAN.

MAN: Stands for Metropolitan Area Network.

The systems which are connected with in the city level are called MAN.

WAN: Stands for Wide Area Network.

The systems which are connected throughout the world level are called WAN.

Request: The data (input) which is need to processing is called request.

Response: The data (output) which has processed is called response.

Client Program: The program which is able to read the input values and send to another program is called client program.

Client Machine: The machine which has client program is called Client Machine.

Server program: The program which is able to send output values to client program is called server program.

Server Machine: The machine which has server program is called Server machine.

IP Address: Every system is identified in the network with one unique identification number that identification number is called IP Address. No two systems have same IP Address.

IP Address starts with 0.0.0.0 to 255.255.255.255

Hostname: The alternative of IP Address is called Hostname.

We can identify the system either through IP Address or hostname.

Port: Every services running on unique number, that number is called port. No two services have same port number, if have those two services not running at a time.

Socket: It is a listener, which is used to send and read the request and response.

Client side listener is called Socket.

Server side listener is called ServlerSocket

Protocol: It is a set of instructions. It is used to transform the data from client machine to server machine.

Types of protocols:

TCP/IP: Transfer Control Protocol/Internet Protocol

UDP: User Datagram protocol.

Under the TCP we have different sub protocols. They are

1) http

2) https

3) smtp

4) ftp

URL: Unified Resource Locator. It is used indentify the resource location. It is provide absolute path.

Absolute path means combination of

Protocol name + hostname/IP address + port number + resource + query string

URI: Unified Resource Identifier. It is directly represent the resource and query string.

It provides relative path.

```
Socket s = new Socket("localhost",7777);
 DataOutputStream dos =   new DataOutputStream(s.getOutputStream());
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
BufferedReader br1 = new BufferedReader(new InputStreamReader(s.getInputStream()));
 String str,str1;
        while(!(str=br.readLine()).equals("exit")){
            dos.writeBytes(str+"\n");str1=br1.readLine();System.out.println(str1);
        }
    }
==================================================
ServerSocket ss = new ServerSocket(7777);Socket s=ss.accept();System.out.println("connection
established");
        PrintStream ps = new PrintStream(s.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BufferedReader br1 = new BufferedReader(   new InputStreamReader(s.getInputStream()));
        String str,str1;
        while(true){
            while((str=br1.readLine())!=null){
                System.out.println(str);str1=br.readLine();
                ps.println(str1);
            }
```

ClientDemo.java:

import java.io.BufferedReader;

import java.io.DataOutputStream;

import java.io.IOException;

import java.io.InputStreamReader;

```
import java.net.Socket;

import java.net.UnknownHostException;

public class ClientDemo {

	public static void main(String[] args) throws
UnknownHostException, IOException {

		Socket s = new Socket("localhost",7777);

		DataOutputStream dos= new
DataOutputStream(s.getOutputStream());

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

		BufferedReader br1 = new BufferedReader(new
InputStreamReader (s.getInputStream()));

		String str,str1;

		while(!(str=br.readLine()).equals("exit")){

			dos.writeBytes(str+"\n");

			str1=br1.readLine();

			System.out.println(str1);

		}

	}

}
```

ServerDemo.java

```
import java.io.BufferedReader;

import java.io.IOException;
```

```
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;
public class ServerDemo {
	public static void main(String[] args) throws IOException {
		ServerSocket ss = new ServerSocket(7777);
		Socket s=ss.accept();
		System.out.println("connection established");
		PrintStream ps = new
PrintStream(s.getOutputStream());
	BufferedReader br = new BufferedReader(new
InputStreamReader (System.in));
		BufferedReader br1 = new BufferedReader(new
InputStreamReader (s.getInputStream()));
		String str,str1;
		while(true){
			while((str=br1.readLine())!=null){
				System.out.println(str);
				str1=br.readLine();
				ps.println(str1);
			}
		}}}
```

Reflection API:

If we want know source code information from ".class" file(byte code) then we can go for Reflection API.

With reflection API we can read information from runtime loaded class(.class).

Reflection API provides mirror information about variables, methods, constructors, annotations, exception class information.

With the help of reflection API we can access both private and public data.

This Reflection API is coming under

java.lang.reflect package.

Important classes under java.lang.reflect packages are

1)Filed

2)Constructor

3)Method

4)Modifer

Filed: It is used to store the variable information. Variables may public or private.

Method: It is used to store the methods information (public or private)

Constructor: It is used to store the information about constructors.

Modifier: It is used to store information about access modifier and modifier.

This class provides the modifier information in the form of int.

If want convert from int to string then Modifier class itself provides some predefine method (toString()).

ABSTRACT	1024
FINAL	16
INTERFACE	512
NATIVE	256
PRIVATE	2
PROTECTED	4
PUBLIC	1
STATIC	8
STRICT	2048
SYNCHRONIZED	32
TRANSIENT	128
VOLATILE	64

java.lang.Class: This class having capability to hold any class bytecode information.

Important method under java.lang.Class:

1)getDeclaredFields()

2)getFields()

3)getDeclaredMethods()

4)getMethods()

5)getDeclaredConstructors()

6)getConstructors()

forName():

It is an one static factory method, is used to loads the byte code of any class.

forName() always needs bytecode not source code.

Syntax : Class cls = Class.forName("class_name");

This method has been throwing one compile time exception i.e

Java.lang.ClassNotFoundException.

forName() is throwing exception like ClassNotFoundException if we are not passing exsisting ".class" file name(classname).

newInstance():

This is one instance factory methods, which is used to create object with the help of class reference variable.

Object obj = cls.newInstance().

newInstance() has been throwing two compile time excepitons.

1) IllegalAccessException

2) InstantiationException

Jvm will throwing Instantiation Exception, if given ".class " file(class) doesnot contains zero argument constructor.

Private variables by defaultly having one background method like setAccessible(false), so this method doesnot

provide permission to access the data out side of the class.

If we want to access private data out side of the class we need to convert from setAccessible(false) to setAccessible(true), otherwise we will IllegalAccessException

```
import java.io.FileNotFoundException;
import java.io.IOException;
public class A {
	public int b=2000;
	A(){	//getDeclatedConstructor or Constructor
		System.out.println("A class construc");
	}
	A(int x){
		System.out.println("A class paramconstruc");
	}
	private static int c =3000;
	int m1()throws IOException,FileNotFoundException{
		return 10;
	}
	public void m2(int x){
		System.out.println("hi");
	}
}
```

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
public class B  extends A{
	public static void main(String[] args)
			throws ClassNotFoundException,
InstantiationException, IllegalAccessException {
		java.lang.Class cls = java.lang.Class.forName("A");
				java.lang.Object obj =
cls.newInstance();
		Field f[]= cls.getDeclaredFields();
		for(Field f1:f){
			String modi = Modifier.toString(f1.getModifiers());
			if(modi.contains("private")){
				System.out.println(f1);
				f1.setAccessible(true);
				System.out.println(f1.get(obj));
				Object type = f1.getType();
				System.out.println(type);
			}
```

```
            System.out.println(f1);
        }
        System.out.println("*****************");
        Constructor[] c = cls.getDeclaredConstructors();
        for(Constructor c1:c){
            System.out.println(c1);
        }
        System.out.println("*****************");
        Method[] m = cls.getDeclaredMethods();
        for(Method a:m){
            System.out.println(a);
            System.out.println("*************");
            Class[] e = a.getExceptionTypes();
            for(Class e1:e){
                System.out.println(e1);
            }
        }
    }
}
```

Annotations:

Annotation provides explanatory information about subject.

It is one programming element comes under referenced datatype.

It one type of interface.

It will provide metadata(data about data or more meaningfull information)on java programming elements(class,enum,interface,method,constructors,variable).

It main usage of annotation is

to develop doccuments comments

to develop xml configuration files.

annotations are always force to prorammer to send values at usage time.

syntax:

<accessmodifier> @interface annotation_name{

}

<accessmodifier> can be replaced with public,strictfp,abstract

we cannot apply private, protected, static.

Annotation always starts with @ symbol

If we are not writing @ symbol it will be treated as interface.

<accessmodifier> @interface annotation_name{

public static final int a = 111;

public abstract non-void and non-parameterized methods

}

```
public @interface Test{
	int a=111;
	int m1();
}
```

javac Test.java

compile added funcationalities:

```
public @interface Test extends java.lang.annotation.Annotation{
	public static final int a=111;
	public abstract int m1();
}
```

invalid:

1.
```
	@interface Test{
		int a;
	}
```
2.
```
	@interface Test{
		void m1();
	}
```
3.
```
	@interface Test{
```

```
        int m1(int x);
    }
```

4.
```
    public @interface Test{
    //private int a=111;
    //protected int a=111;

    //private abstract int m1();
    //protected int m1();
    //static abstract int m1();
    //final abstract int m1();
    }
```

infront of interface keyword if we are not writing @ that is comes under normal interface or compiler treat that element as interface.

```
public interface Test{
}
```

javac Test.java

javap Test

after Test we are not getting extends of java.lang.annotation.Annotation.

we cannot extends keyword after annotation.

ex: public @interface Test extends java.lang.annotation.Annotation{

}

above code is invalid.

Why is the use of '@' ?

A) it is acommunication channel between programmer and compiler to identify whether programming element is annotation or not.Once we write @ symbol compile will compile over program with the help annotation rules otherwise interface rules.

---

annotation for doccument comment:

```
@interface Test{
    String author();
    String version();
}
//@Test
//@Test(Author="ram",version="1.8")
//@Test(author="ram",version="1.8");
```

```
//@Test(author="ram",version=1.8)
//@Test(author="ram" version="1.8")
@Test(author="ram", version="1.8")
class A{
	public static void main(String[] s){
		System.out.println("main method");
	}
}
```

1.annotation starts with '@'.

2. values must be write in "()".

3. member and value are separated with '='.

4. members separated with ','.

5. members must be in same case.(case sensitive)

6. valuemust be compatable with returntype.

7. annotation should not ended with ';'

Annotation rules:

- starts with @
- we can apply public , strictfp, abstract
- extends not allowed
- variable are public static final
- public abstract non void and non-param methods

Return Types:

---

primitive

String

java.lang.Class

enum

annotation

array of predefine

Dont write void and Wrapper class.

Dont use same annotation name in annotation as method parameter.

error:

```
@interface Example{
    Example m1();
}
enum Color{
}
@interface Test1{}
@interface Test{
    int author();
```

```
    String version();
    Class m2();
    Test1 m3();
    int[] m4();
    Color m5();
    //Integer m6();
    //Test m7();
}
```

Why method are non-void?

Annotation methods always treated as a variables in class level

if we treat method as variable, we need to assign some value

that value type specified by the return type.thatswhy annotation methods always non-void.

we cannot apply throws keyword on annotation methods

Why method are non-param?

Annoataion method doesnot required any parameter the reason

is in class level there is no implementation for annoatiaon methods.

How can we make anntation member name as optional?

by using value word.

```
@interface Test{
	String value();
}
-----
//@Test(value="ram")    //here value word and = is optional
@Test("ram")
class A{
	public static void main(String[] s){
		System.out.println("main method");
	}
}
```

```
	with default
@interface Rank{
	String name();
	int rank() default 1;
}
@Rank(name="ram")
class A{
@Rank(name="ram",rank=2)
```

```
	void m1(){
	}

}
@Rank(name="ram")
class A{
@Rank(rank=2,name="ram")
	void m1(){
	}

}we can change the order of members also


types of annotatons:

	marker annotation:no methods
	sinlge value: single member
	@interface Rank{
	String value();

	}
	@Rank("ram")
```

```
class A{

	void m1(){
	}

}

/*@Rank(value="ram")
class A{

	void m1(){
	}

}*/
```

---

array return type:

```
@interface Rank{
	String[] value();

}

/*@Rank({"php"})
```

```
class A{

	void m1(){
	}

}*/

/*@Rank({"php","oracle"})
class A{

	void m1(){
	}

}*/
@Rank("php")
class A{

	void m1(){
	}

}
```

multi value: multiple members

what is difference between marker interface and marker annotation?

A) Marker interface will treat our class as a special class and provide the permission to partispated in the special process.

Marker annotation will treat our programming element as a special element.

Predefine annotation given by sun:

--> meta annotations

--> standard annotations.

meta:java.lang.annotation

we can not appply on class,method,variable these are apply on other annotations

import java.lang.annotation package.

Target

Retention

Inherited

Documented

Repeatable

Native

Standard Annotation: Defined in java.lang

- Deprecated
- Override
- SuppressWarnings
- SafeVarargs
- FunctionalInterface

@Deprecated:

```
class A{
	@Deprecated
	void m1(){
	}

}

class B{
	public static void main(String[] s){
		A obj = new A();
		obj.m1();
	}
```

}

Override: This annotation will check whether the the following method is

overrided method or not by appling 5 rules.

If any one of the rule failure, then we will get compile time error.

syntax:

@Target(value=METHOD)

@Retention(value=SOURCE)

public @interface Override

@Override is only applicable for methods.

@Override is only available in source code, after checking the method is override or not simple compiler will neglect this annotation converting into bytecode.

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

class B extends A{
```

```
	@Override
	void m1(){
		System.out.println("B class m1 method");
	}
}
public class TestDemo {
	public static void main(String[] args) {
		A obj = new B();
		obj.m1();
	}
}
```

---

java.lang.Deprecated:

@Deprecated

this annotaion we can apply on top of the following programming elements.

CONSTRUCTOR

FIELD

LOCAL_VARIABLE

METHOD

PACKAGE

PARAMETER

TYPE

This annotation will talk about the above particular element programatically not performance or not good.

when ever any programmer using any deprected programming elements, then compiler will give some warning message by using @Deprecated annotation.

```
	class A{
	@Deprecated
	A(){
	}
	//@Deprecated
	int  add(int a, int b ){
		int c = a + b;
		return c;
	}
}
class B{
	int add(int a, int b ){
		return a+b;
	}
}
public class Demo1 {
	public static void main(String[] args) {
```

```
            System.out.println(new A().add(10, 20));
            System.out.println(new B().add(10,20));
      }
}
-----------------
SuppressWarnings
import java.util.ArrayList;


public class A {
      @SuppressWarnings("rawtypes")

      public static void main(String[] args) {
            ArrayList al = new ArrayList();
      }

}
-------------
/*import java.util.*;
import java.lang.*;

class SafeVarargsEx{
```

```
 static void call(List<String>... stringLists) {
  String s = stringLists[0].get(0);
  System.out.println(s);
 }

 public static void main(String args[]){
  List<String> myList1 = new ArrayList<> ();
  List<String> myList2 = new ArrayList<> ();

  myList1.add("Hi");
  myList2.add("Hi");

  call(myList1, myList2);
 }
}*/
import java.util.*;
import java.lang.*;

class SafeVarargsEx{
@SafeVarargs
 static void call(List<String>... stringLists) {
  String s = stringLists[0].get(0);
```

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

 public static void main(String args[]){
  List<String> myList1 = new ArrayList<> ();
  List<String> myList2 = new ArrayList<> ();

  myList1.add("Hi");
  myList2.add("Hi");

  call(myList1, myList2);
 }
}
import java.util.*;
import java.lang.*;

class SafeVarargsEx{
@SafeVarargs
 static void call(List<String>  stringLists) {
  String s = stringLists[0].get(0);
  System.out.println(s);
 }
```

```
 public static void main(String args[]){
  List<String> myList1 = new ArrayList<> ();
  List<String> myList2 = new ArrayList<> ();

  myList1.add("Hi");
  myList2.add("Hi");

  call(myList1, myList2);
 }
}
import java.util.*;
import java.lang.*;

class SafeVarargsEx{
@SafeVarargs
void call(List<String>... stringLists) {
  String s = stringLists[0].get(0);
  System.out.println(s);
 }

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

```
  List<String> myList1 = new ArrayList<> ();
  List<String> myList2 = new ArrayList<> ();

  myList1.add("Hi");
  myList2.add("Hi");

  call(myList1, myList2);
 }
}
```

```
import java.util.*;
import java.lang.*;

class SafeVarargsEx{
@SafeVarargs
final void call(List<String>... stringLists) {
  String s = stringLists[0].get(0);
  System.out.println(s);
 }

 public static void main(String args[]){
  List<String> myList1 = new ArrayList<> ();
  List<String> myList2 = new ArrayList<> ();
```

```
  myList1.add("Hi");
  myList2.add("Hi");

 // call(myList1, myList2);
 }
}
@FunctionalInterface
Interface I{
}
```

@Target: it is a sigle level annotation.

it talks about the place we can use.

it have method like value() its retuntype is enum[]

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

  /** Class, interface (including annotation type), or enum
declaration */
    TYPE,
```

```
    /** Field declaration (includes enum constants) */
    FIELD,

    /** Method declaration */
    METHOD,

    /** Parameter declaration */
    PARAMETER,

    /** Constructor declaration */
    CONSTRUCTOR,

    /** Local variable declaration */
    LOCAL_VARIABLE,

    /** Annotation type declaration */
    ANNOTATION_TYPE,

    /** Package declaration */
    PACKAGE
}
```

```
@Target(ElementType.TYPE)
public @interface Rank {
	String value();

}
```

@Retention:

	it is single value annotation

SOURCE: only available in .java

CLASS: available in .class but jvm not recognize

RUNTIME: available .class and jvm can recognize

```
@Rank("ram")
public class A {

	A(){

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

	}
```

```
}
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Course {
	String faculty();
	String courseName();
}
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Inherited
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.TYPE)
public @interface Rank {

	int rank();
}




@Rank(rank=1)
@Course(faculty="ram",courseName="advanced java")
public class L {

}
import java.lang.annotation.Annotation;


public class M extends L{
	public static void main(String[] args)
	throws ClassNotFoundException,
	InstantiationException,IllegalAccessException{
		Class cls  = Class.forName("L");
```

```
            Object o = cls.newInstance();
            Annotation[] a = cls.getDeclaredAnnotations();
            //Course(faculty,courseName) Rank(rank)
            for(Annotation a1 : a){
                System.out.println(a1);
                if(a1 instanceof Course){
                    Course c = (Course)a1;
                    System.out.println(c.courseName());
                    System.out.println(c.faculty());
                }
                else if(a1 instanceof Rank){
                    Rank r = (Rank)a1;
                    System.out.println(r.rank());
                }
            }
        }
}
```

Optional class :

```
import java.util.Optional;
public class Demo {
    public static void main(String[] args){

                String[] words = new String[10];
                Optional opt =
```

```
Optional.ofNullable(words[5]);
                    System.out.println(opt.isPresent());
                    if (opt.isPresent()) {
                        String word = words[5].toLowerCase();
                        System.out.print(word);
                    } else
                        System.out.println("word is null");
                }
}
import java.util.Optional;

public class Test{
     public static void main(String[] args){
          /*String[] s = new String[3];
          s[2]="ramchandra";
          Optional<String> o = Optional.ofNullable(s[2]);
          o.ifPresent(System.out::println);
          System.out.println(o.get());*/
          /*String[] s = new String[3];

          Optional o = Optional.ofNullable(s[2]);
          if(o.isPresent()){
               System.out.println(s[2].toUpperCase());
          }
          else{
               System.out.println("there is no string");
          }
          System.out.println("hi");*/
          /*String[] s = new String[3];
          //System.out.println(s[2].toUpperCase());
          Optional o = Optional.ofNullable(s[2]);
          if(o.isPresent()){
               System.out.println(s[2].toUpperCase());
          }
          else{
               System.out.println("there is no string");
          }
          System.out.println("hi");*/
```

```
	}
}
```

How to create icon for executing our program:

First create one .java file with some package like bellow.

```
package com.ram;

import java.io.*;


class Test{

	public static void main(String[] s) throws Exception{

		FileOutputStream fos = new
FileOutputStream("check.txt");

		PrintStream ps = new PrintStream(fos);

		ps.println("good");

	}
}
```

Compilation: javac –d . Test.java

Later : right click on current location and goto new and click on shortcut

And give location of your .class file like bellow

Javaw com.ram.Test

And give shortcut icon name.

Later right click on shortcut give location in start in textfield like

C:\Users\Ramchandar\Desktop

And click on apply and ok.

Then double click on that icon our program automatically executes.