CHAPTER – 1
INTRODUCTION TO CORE JAVA

HISTORY OF JAVA:

James Gosling

The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape. Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc.

James Gosling, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**. Originally designed for small, embedded systems in electronic appliances like set-top boxes.

Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt. After that, it was called **Oak** and was developed as a part of the Green project. Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.

In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies. The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. According to James Gosling "Java was one of the top choices along with Silk". Since java was so unique, most of the team members preferred java.

Java is an island of Indonesia where first coffee was produced (called java coffee). Notice that Java is just a name not an acronym. Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995. In 1995, Time magazine called Java one of the Ten Best Products of 1995. JDK 1.0 released in (January 23, 1996).

JAVA VERSION HISTORY:

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)

10. Java SE 8 (18th March, 2014)

FEATURES OF JAVA:

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

1. SIMPLE:

According to Sun, Java language is simple because syntax is based on C++ (so easier for programmers to learn it after C++). It removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc. No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

2. OBJECT-ORIENTED:

Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior. Object-oriented programming (OOPs) is a methodology that simplify software development and maintenance by providing some rules.
Basic concepts of OOPs are:

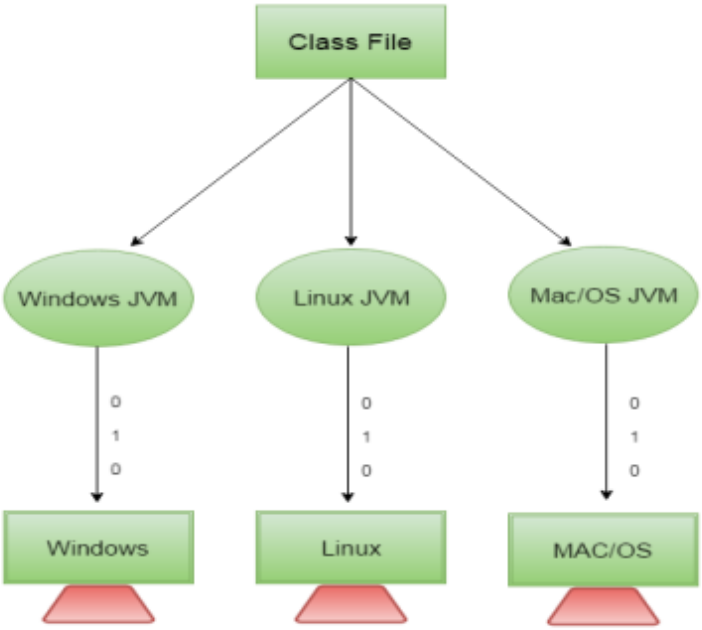
- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

3. PLATFORM INDEPENDENT:

Java is platform independent. A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides software-based platform. The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

a. Runtime Environment
b. API(Application Programming Interface)

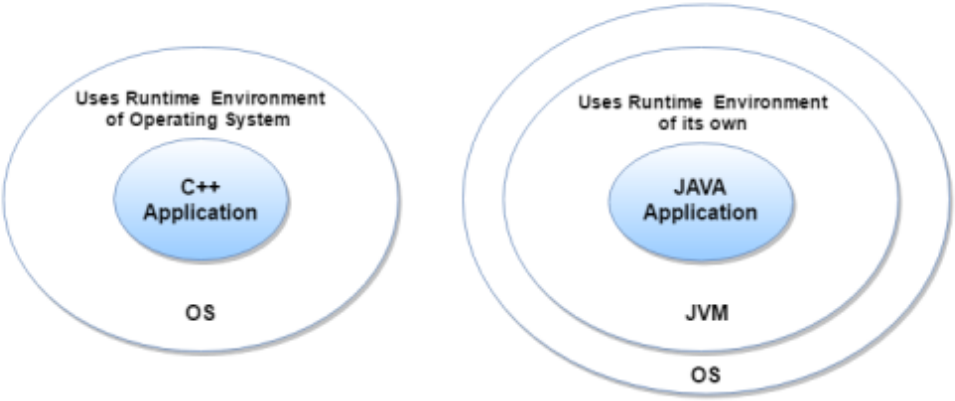
Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere (WORA).



4. Secured:

Java is secured because:

- **No explicit pointer**
- **Java Programs run inside virtual machine sandbox**



- **Classloader:** adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, and Cryptography etc.

5. Robust:

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

6. Architecture-Neutral:

There is no implementation dependent features e.g. size of primitive types is fixed. In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

7. Portable:

We may carry the java bytecode to any platform.

8. High-Performance:

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

9. Distributed:

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

10. Multi-Threaded:

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications etc.

11. Dynamic:

Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

12. Interpreted:

Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

INTRODUCTION TO JVM ARCHITECTURE:

JVM stands for Java Virtual Machine and is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

It is a specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies. Its implementation is known as JRE (Java Runtime Environment). Whenever we write java command on the command prompt to run the java class, an instance of JVM is created.

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

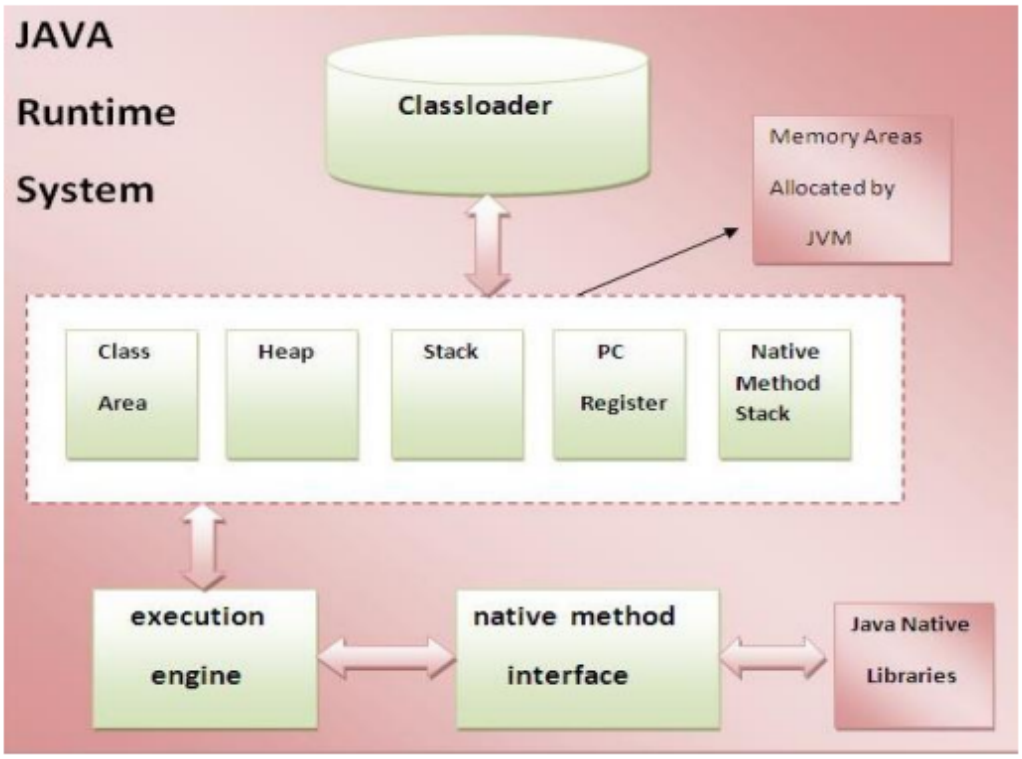


Fig: Internal Architecture of JVM

1. **Classloader:**

Classloader is a subsystem of JVM that is used to load class files.

2. **Class(Method) Area:**

Class (Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3. **Heap:**

It is the runtime data area in which objects are allocated.

4. **Stack:**

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5. **Program Counter Register:**

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

6. **Native Method Stack:**

It contains all the native methods used in the application.

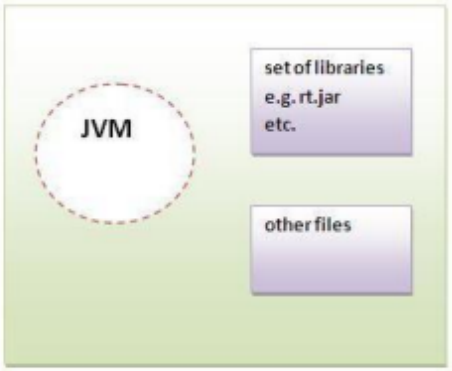
7. **Execution Engine:**

It contains:

a. A virtual processor
b. Interpreter: Read bytecode stream then execute the instructions.
c. Just-In-Time (JIT) compiler: It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term Compiler refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

INTRODUCTION TO JRE:

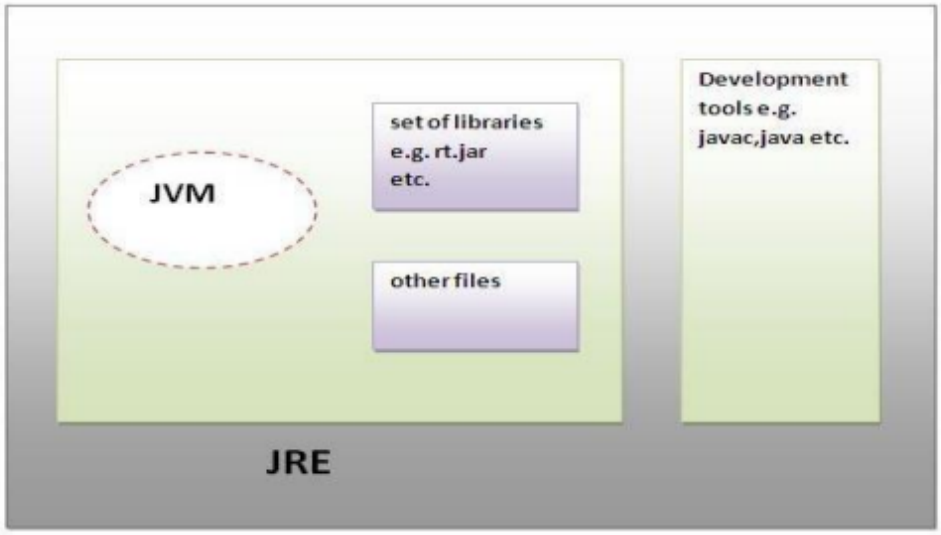
JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime. Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JRE

INTRODUCTION TO JDK:

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.



JDK

OBJECT ORIENTED FEATURES:

1. ENCAPSULATION:

Encapsulation means putting together all the variables (instance variables) and the methods into a single unit called Class. It also means hiding data and methods within an Object. Encapsulation provides the security that keeps data and methods safe from inadvertent changes. Programmers sometimes refer to encapsulation as using a "black box," or a device that you can use without regard to the internal mechanisms. A programmer can access and use the methods and data contained in the black box but cannot change them. Below example shows Mobile class with properties, which can be set once while creating object using constructor arguments. Properties can be accessed using getXXX() methods which are having public access modifiers.

Example:

```
public class Mobile {
    private String manufacturer;
    private String operatingSystem;
    public String model;
    private int cost;
    //Constructor to set properties/characteristics of object
    Mobile(String man, String o,String m, int c){
            this.manufacturer = man;
            this.operatingSystem=o;
            this.model=m;
            this.cost=c;
    }
    //Method to get access Model property of Object
    public String getModel(){
            return this.model;
```

```
    }
    public String getManufacturer(){
        return this.manufacturer;
    }
    public String getOperatingSystem(){
        return this.operatingSystem;
    }
    public int getCost(){
        return this.cost;
    }
    public static void main(String [] args){
        Mobile obj = new Mobile("Samsung","Lolipop","J5",18000);
        System.out.println("Mobile Specification");
        System.out.println("Manufacturer: "+obj.getManufacturer());
        System.out.println("Operating System: "+obj.getOperatingSystem());
        System.out.println("Model: "+obj.getModel());
        System.out.println("Cost: "+obj.getCost());
    }
}
```

2. INHERITANCE:

An important feature of object-oriented programs is inheritance. Inheritance is the ability to create classes that share the attributes and methods of existing classes, but with more specific features. Inheritance is mainly used for code reusability. So we are making use of already written the classes and further extending on that. That why we discussed the code reusability the concept. In general one line definition, we can tell that deriving a new class from existing class, it's called as Inheritance. We can look into the following example for inheritance concept.

```
public class Mobile {
    private String manufacturer;
    private String operatingSystem;
    private String model;
    private int cost;
    public Mobile(String man, String o,String m, int c){
        manufacturer = man;
        operatingSystem=o;
        model=m;
        cost=c;
    }
    public String getModel(){
        return model;
    }
    public String getManufacturer(){
```

```
            return manufacturer;
        }
        public String getOperatingSystem(){
            return operatingSystem;
        }
        public int getCost(){
            return cost;
        }
    }
    public class Android extends Mobile{
        private int number;
        Android(int no){
            super("Samsung","Lolipop","J5",18000);
            this.number = no;
        }
        public int getNumber(){
            return this.number;
        }
        public static void main(String [] args){
            Android obj = new Android(15000);
            System.out.println("\nMobile Specification");
            System.out.println("Manufacturer: "+obj.getManufacturer());
            System.out.println("Operating System: "+obj.getOperatingSystem());
            System.out.println("Model: "+obj.getModel());
            System.out.println("Cost: "+obj.getCost());
            System.out.println("Sale Set: "+obj.getNumber());
        }
    }
```

3. Polymorphism:

Polymorphism definition is that **Poly** means **many** and **morphos** means **forms**. It describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. For example, in English, the verb "run" means different things if we use it with "a footrace," "a business," or "a computer." We understand the meaning of "run" based on the other words used with it. Object-oriented programs are written so that the methods having the same name works differently in different context. Java provides two ways to implement polymorphism.

a. Static Polymorphism (Compile Time Polymorphism/ Method Overloading):

The ability to execute different method implementations by altering the argument used with the method name is known as method overloading. In below program, we have three print methods each with different arguments. When we properly overload a method, we can call it providing different argument lists, and the appropriate version of the method executes.

```
class Overload{
    public void print(String s){
            System.out.println("First Method with only String: "+ s);
    }
    public void print (int i){
            System.out.println("Second Method with only int: "+ i);
    }
    public void print (String s, int i){
            System.out.println("Third Method with both String and Integer: "+ s + " and " + i);
    }
    public static void main(String[] args) {
            Overload obj = new Overload();
            obj.print(10);
            obj.print("Amit");
            obj.print("Hello", 100);
    }
}
```

b. <u>Dynamic Polymorphism (Run Time Polymorphism/ Method Overriding):</u>

When we create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. In other words, any child class object has all the attributes of its parent. Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, we want to override the parent class members. Let's take the example used in inheritance explanation.

```
class Parent{
            public void function(){
            System.out.println("This is parent class");
            }
    }
    class Child extends Parent{
            public void function(){
            System.out.println("This is child class");
            }
    }
    public class Help{
            public static void main(String [] args){
            Child obj = new Child();
            Parent obj1 = new Parent();
            obj.function();
            obj1.function();
            }
    }
```

4. Abstraction:

An essential element of object-oriented programming is an abstraction. Hiding internal details and showing functionality is known as abstraction. Humans manage complexity through abstraction. When we drive our car we do not have to be concerned with the exact internal working of our car (unless we are a mechanic). What we are concerned with is interacting with our car via its interfaces like steering wheel, brake pedal, accelerator pedal etc. Various manufacturers of car have different implementation of the car working but its basic interface has not changed (i.e. we still use the steering wheel, brake pedal, accelerator pedal etc. to interact with our car). Hence the knowledge we have of our car is abstract.

In java, we use abstract class and interface to achieve abstraction. An abstract class is something which is incomplete and we cannot create an instance of the abstract class. If we want to use it we need to make it complete or concrete by extending it. A class is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended. By the way, Java has a concept of abstract classes, abstract method but a variable cannot be abstract in Java.

```
abstract class VehicleAbstract {
    public abstract void start();
    public abstract void stop();
}
class TwoWheeler extends VehicleAbstract{
    public void start() {
            System.out.println("Starting Two Wheeler");
    }
    public void stop(){
            System.out.println("Stopping Two Wheeler");
    }
}
class FourWheeler extends VehicleAbstract{
    public void start() {
            System.out.println("Starting Four Wheeler");
    }
    public void stop(){
            System.out.println("Stopping Four Wheeler");
    }
}
public class VehicleTesting {
    public static void main(String[] args) {
            TwoWheeler my2Wheeler = new TwoWheeler();
            FourWheeler my4Wheeler = new FourWheeler();
            my2Wheeler.start();
            my2Wheeler.stop();
            my4Wheeler.start();
```

```
            my4Wheeler.stop();
        }
    }
```

CLASS AND OBJECT:

A class is a template, blueprint, or contract that defines what an object's data fields and methods will be. An object is an instance of a class. We can create many instances of a class. A Java class uses variables to define data fields and methods to define actions. Additionally, a class provides methods of a special type, known as constructors, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects.
Objects are made up of attributes and methods. Attributes are the characteristics that define an object; the values contained in attributes differentiate objects of the same class from one another. To understand this better let's take the example of Mobile as an object. Mobile has characteristics like a model, manufacturer, cost, operating system etc. So if we create "Samsung" mobile object and "IPhone" mobile object we can distinguish them from characteristics. The values of the attributes of an object are also referred to as the object's state.

OPERATORS:

Operators are used to manipulate primitive data types. Java operators can be classified as unary, binary, or ternary that means they can take one, two, or three arguments, respectively. A unary operator may appear before (prefix) its argument or after (postfix) its argument. A binary or ternary operator appears between its arguments. Operators in java fall into 8 different categories and are explained one by one:

1. ARITHMETIC OPERATOR:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Description
+ (Addition)	Adds values on either side of the operator.
- (Subtraction)	Subtracts right-hand operand from left-hand operand.
* (Multiplication)	Multiplies values on either side of the operator.
/ (Division)	Divides left-hand operand by right-hand operand.
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.
++ (Increment)	Increases the value of operand by 1.
-- (Decrement)	Decreases the value of operand by 1.

Example:

```
public class Test {
  public static void main(String args[]) {
    int a = 10;
```

```
    int b = 20;
    System.out.println("a + b = " + (a + b) );
    System.out.println("b - a = " + (b - a) );
    System.out.println("a * b = " + (a * b) );
    System.out.println("b / a = " + (b / a) );
    System.out.println("b % a = " + (b % a) );
    System.out.println("a++   = " +  (a++) );
    System.out.println("a--   = " +  (a--) );
    System.out.println("++b   = " +  (++b) );
    System.out.println("--b   = " +  (--b) );
  }
}
```

2. Relational Operators:

Relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. We just have to keep in mind that we must use "==", not "=", while testing if two primitive values are equal. The outcome of these operations is a Boolean value. The following table lists the relational operators:

Operator	Description
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Example:

```
public class Test {
  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    System.out.println("a == b = " + (a == b) );
    System.out.println("a != b = " + (a != b) );
    System.out.println("a > b = " + (a > b) );
    System.out.println("a < b = " + (a < b) );
    System.out.println("b >= a = " + (b >= a) );
```

```
      System.out.println("b <= a = " + (b <= a) );
    }
  }
```

3. Bitwise Operators:

Java provides Bitwise operators to manipulate the contents of variables at the bit level. These variables must be of numeric data type (byte, char, short, int, long). These operators act upon the individual bits of the operators. These operators are less commonly used. Java provides seven bitwise operators.

Operator	Description
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.

Example:

```
public class Test {
  public static void main(String args[]) {
    int a = 60;  /* 60 = 0011 1100 */
    int b = 13;  /* 13 = 0000 1101 */
    int c = 0;
    c = a & b;       /* 12 = 0000 1100 */
    System.out.println("a & b = " + c );
    c = a | b;       /* 61 = 0011 1101 */
    System.out.println("a | b = " + c );
    c = a ^ b;       /* 49 = 0011 0001 */
    System.out.println("a ^ b = " + c );
    c = ~a;          /*-61 = 1100 0011 */
    System.out.println("~a = " + c );
    c = a << 2;      /* 240 = 1111 0000 */
    System.out.println("a << 2 = " + c );
    c = a >> 2;      /* 15 = 1111 */
```

```
      System.out.println("a >> 2  = " + c );
      c = a >>> 2;     /* 15 = 0000 1111 */
      System.out.println("a >>> 2 = " + c );
   }
}
```

4. Logical Operators:

Logical operators return a true or false value based on the state of the Variables. Each argument to a logical operator must be a Boolean data type, and the result is always a Boolean data type. Logical operators are known as Boolean operators or bitwise logical operators. The Boolean operator operates on Boolean values to create a new Boolean value. The following table lists the logical operators:

Operator	Description
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
|| (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

Example:

```
public class Test {
   public static void main(String args[]) {
      boolean a = true;
      boolean b = false;
      System.out.println("a && b = " + (a&&b));
      System.out.println("a || b = " + (a||b) );
      System.out.println("!(a && b) = " + !(a && b));
   }
}
```

5. Assignment Operator:

The java assignment operator statement has the following syntax: <variable> = <expression>. The values generated by the right hand side expression will be assign to the variable in the left side, If the value already exists in the variable it is overwritten by the assignment operator (=). Following are the assignment operators supported by Java language:

Operator	Description
=	Simple assignment operator. Assigns values from right side operands to left side operand.
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.

-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.
<<=	Left shift AND assignment operator.
>>=	Right shift AND assignment operator.
&=	Bitwise AND assignment operator.
^=	Bitwise exclusive OR and assignment operator.
|=	Bitwise inclusive OR and assignment operator.

Example:

```
public class Test {
  public static void main(String args[]) {
    int a = 10;
    int b = 20;
    int c = 0;
    c = a + b;
    System.out.println("c = a + b = " + c );
    c += a ;
    System.out.println("c += a  = " + c );
    c -= a ;
    System.out.println("c -= a = " + c );
    c *= a ;
    System.out.println("c *= a = " + c );
    a = 10;
    c = 15;
    c /= a ;
    System.out.println("c /= a = " + c );
    a = 10;
    c = 15;
    c %= a ;
    System.out.println("c %= a  = " + c );
    c <<= 2 ;
    System.out.println("c <<= 2 = " + c );
    c >>= 2 ;
    System.out.println("c >>= 2 = " + c );
    c >>= 2 ;
    System.out.println("c >>= 2 = " + c );
    c &= a ;
```

```
    System.out.println("c &= a  = " + c );
    c ^= a ;
    System.out.println("c ^= a   = " + c );
    c |= a ;
    System.out.println("c |= a   = " + c );
  }
}
```

6. Conditional Operator (? :)

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator evaluates the first argument and, if true, evaluates the second argument. If the first argument evaluates to false, then the third argument is evaluated. The conditional operator is the expression equivalent of the if-else statement. The operator is written as:

variable x = (expression) ? value if true : value if false

Example:

```
public class Test {
  public static void main(String args[]) {
    int a, b;
    a = 10;
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is : " +  b );
    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
  }
}
```

7. Instanceof Operator:

The instanceof operator is use to test whether the object is an instance of the specified type (class or sub class or interface). instanceof operator is written as:

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example:

```
class Vehicle {}
public class Car extends Vehicle {
  public static void main(String args[]) {
    Vehicle a = new Car();
    boolean result =  a instanceof Car;
    System.out.println( result );
  }
}
```

DATA TYPES:

Variables are nothing but reserved memory locations to store values. This means that when we create a variable we reserve some space in the memory. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, we can store integers, decimals, or characters in these variables. There are two data types available in Java:

1. PRIMITIVE DATA TYPES:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword.

a. Byte:

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

b. Short:

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000

c. Int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648 (-2^31)
- Maximum value is 2,147,483,647(inclusive) (2^31 -1)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

d. Long:

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive)(2^63 -1)
- This type is used when a wider range than int is needed

- Default value is 0L
- Example: long a = 100000L, long b = -200000L

e. Float:

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

f. Double:

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

g. Boolean:

- Boolean data type represents one bit of information
- There are only two possible values: true and false
- This data type is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

h. Char:

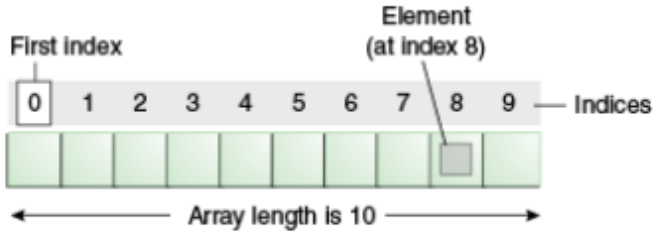
- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

2. REFERENCE DATA TYPES:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various type of array variables come under reference datatype.
- Default value of any reference variable is null.
- A reference variable can be used to refer any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

JAVA ARRAY:

Normally, array is a collection of similar type of elements that have contiguous memory location. **Java array** is an object that contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array. Array in java is index based, first element of the array is stored at 0 index.



Advantage of Java Array

- Code Optimization: It makes the code optimized, we can retrieve or sort the data easily.
- Random access: We can get any data located at any index position.

Disadvantage of Java Array

- Size Limit: We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

PROGRAMMING WITH ARRAYS

1. **Zero-Based Indexing:**

We always refer to the first element of an array a[] as a[0], the second as a[1], and so forth. It might seem more natural to us to refer to the first element as a[1], the second value as a[2], and so forth, but starting the indexing with 0 has some advantages and has emerged as the convention used in most modern programming languages.

2. **Array Length:**

Once we create an array, its length is fixed. We can refer to the length of an a[] in our program with the code a.length.

3. **Default Array Initialization:**

For economy in code, we often take advantage of Java's default array initialization convention. For example

```
double[] a = new double[n];
```

The default initial value is 0 for all numeric primitive types and false for type Boolean.

4. **Memory Representation:**

When we use new to create an array, Java reserves space in memory for it (and initializes the values). This process is called memory allocation.

5. **Bounds Checking:**

When programming with arrays, we must be careful. It is our responsibility to use legal indices when accessing an array element.

6. **Setting Array Values At Compile Time:**

When we have a small number of literal values that we want to keep in array, we can initialize it by listing the values between curly braces, separated by a comma. For example, we might use the following code in a program that processes playing cards.

```
String[] SUITS = {"Clubs", "Diamonds", "Hearts", "Spades"};
String[] RANKS = {"2", "3", "4", "5", "6", "7", "8", "9", "10","Jack", "Queen", "King", "Ace"};
```

After creating the two arrays, we might use them to print a random card name such as Queen of Clubs, as follows.

```
int i = (int) (Math.random() * RANKS.length);
int j = (int) (Math.random() * SUITS.length);
System.out.println(RANKS[i] + " of " + SUITS[j]);
```

7. **Setting Array Values At Run Time:**

A more typical situation is when we wish to compute the values to be stored in an array For example, we might use the following code to initialize an array of length 52 that represents a deck of playing cards, using the arrays RANKS[] and SUITS[] just defined.

```
String[] deck = new String[RANKS.length * SUITS.length];
for (int i = 0; i < RANKS.length; i++)
   for (int j = 0; j < SUITS.length; j++)
      deck[SUITS.length*i + j] = RANKS[i] + " of " + SUITS[j];
System.out.println(RANKS[i] + " of " + SUITS[j]);
```

TYPES OF ARRAY IN JAVA:

1. **One Dimensional Array:**

One Dimensional Array has only a single subscript or index. The one dimensional array is also known as vector. It stores data only row wise or column wise.

Syntax:

```
datatype [] arrayRefVar=new datatype[size];
```

Example 1:

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Example 2:

```
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

2. **Multi-Dimensional Array:**

When we declare array more than one dimensional it is known as multi-dimensional array. It consists of two subscripts in which first given number is of row size and second given number is of column size.

Syntax:

```
datatype [][] arrayRefVar=new datatype[row][column];
```

Example 1:

```
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

PASSING ARRAY TO METHOD IN JAVA:

We can pass the java array to method so that we can reuse the same logic on any array.

Example:

```
class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};
```

```
min(a);//passing array to method
}}
```

INHERITANCE IN JAVA:

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object. The idea behind inheritance in java is that we can create new classes that are built upon existing classes. When we inherit from an existing class, we can reuse methods and fields of parent class, and we can add new methods and fields also. Inheritance represents the **IS-A relationship**, also known as parent-child relationship.

The **extends keyword** indicates that we are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality. In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child or subclass. We perform inheritance for:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

TYPES OF INHERITANCE IN JAVA:

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical. In java programming, multiple and hybrid inheritance is supported through interface only.

1. Single Inheritance:

Inheritance is a property of OOP in which the characteristics of one class comes into derived class. In such case a derived class and only one base class are used to share some properties of one another classes.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

2. Multilevel Inheritance:

Multilevel inheritances can be achieved when we place super class A serves as a base class for the derived class B which again behaves as the base class for the derived class C. The B is known as intermediate base class since it provides a link for the inheritance between A and B. The chain

ABC is known as inheritance path. This process can be extended to any number of levels depending upon the requirement.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

3. Hierarchical Inheritance:

In hierarchical inheritance there is single parents from which child or sibling are derived.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
}}
```

INTERFACE IN JAVA:

An interface in java is a blueprint of a class. It has static constants and abstract methods. The interface in java is a mechanism to achieve abstraction. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in

Java. Java Interface also represents IS-A relationship. It cannot be instantiated just like abstract class.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

An interface is different from a class in following ways:

- We cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

DECLARING INTERFACES:

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface:

Example

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements
public interface NameOfInterface {
  // Any number of final, static fields
  // Any number of abstract method declarations\
}
```

Interfaces have the following properties:

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
/* File name : Animal.java */
interface Animal {
  public void eat();
  public void travel();
}
```

IMPLEMENTING INTERFACES:

When a class implements an interface, we can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {
  public void eat() {
    System.out.println("Mammal eats");
  }
  public void travel() {
    System.out.println("Mammal travels");
  }
  public int noOfLegs() {
    return 0;
  }
  public static void main(String args[]) {
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

When overriding methods defined in interfaces, there are several rules to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules:

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces:

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

Example

```
// Filename: Sports.java
public interface Sports {
  public void setHomeTeam(String name);
  public void setVisitingTeam(String name);
}
// Filename: Football.java
public interface Football extends Sports {
  public void homeTeamScored(int points);
  public void visitingTeamScored(int points);
  public void endOfQuarter(int quarter);
}
// Filename: Hockey.java
public interface Hockey extends Sports {
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
  public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces:

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface. The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

Example

```
public interface Hockey extends Sports, Event
```

Tagging Interfaces:

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

Example

```
package java.util;
public interface EventListener
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces:

- **Creates a common parent**:

As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

- **Adds a data type to a class**:

This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

JAVA PACKAGE:

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package:

1. Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2. Java package provides access protection.
3. Java package removes naming collision.

How to run java package program:

We need to use fully qualified name e.g. mypack.Simple etc to run the class.

- To Compile: javac -d . Simple.java
- To Run: java mypack.Simple

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The (.) represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. **import package.*;**

If we use package.* then all the classes and interfaces of this package will be accessible but not subpackages. The import keyword is used to make the classes and interface of another package accessible to the current package.

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

2. **import package.classname;**

If we import package.classname then only declared class of this package will be accessible.

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();
 }
}
```

3. **fully qualified name;**

If we use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But we need to use fully qualified name every time when we are

accessing the class or interface. It is generally used when two packages have same class name e.g. 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();
  }
}
```

If we import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, we need to import the subpackage as well.

SUBPACKAGE IN JAVA:

Package inside the package is called the subpackage. It should be created to categorize the package further.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

Example:

```
package com.javatpoint.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

- To Compile: javac -d . Simple.java
- To Run: java com.javatpoint.core.Simple

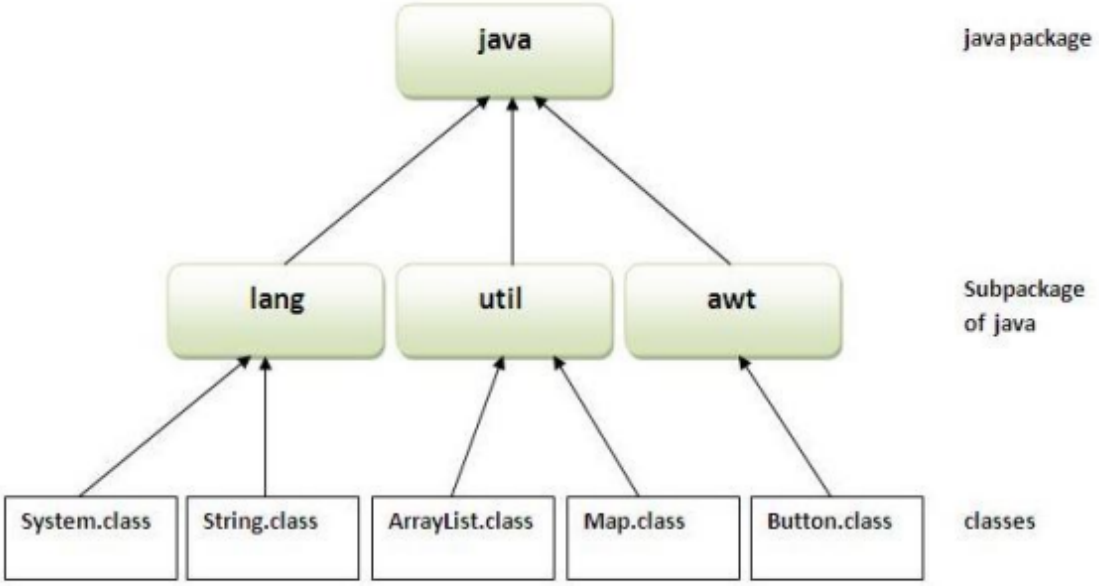


Fig: Java Package

EXCEPTION HANDLING:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner. Based on these, we have three categories of Exceptions:

1. CHECKED EXCEPTIONS:

A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

For example, if we use **FileReader** class in our program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

```
import java.io.File;
import java.io.FileReader;
public class FilenotFound_Demo {
```

```
    public static void main(String args[]) {
      File file = new File("E://file.txt");
      FileReader fr = new FileReader(file);
    }
  }
```

Note: Since the methods read() and close() of FileReader class throws IOException, you can observe that the compiler notifies to handle IOException, along with FileNotFoundException.

2. Unchecked exceptions:

An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
For example, if we have declared an array of size 5 in our program, and trying to call the 6th element of the array then an ArrayIndexOutOfBoundsExceptionexception occurs.

Example

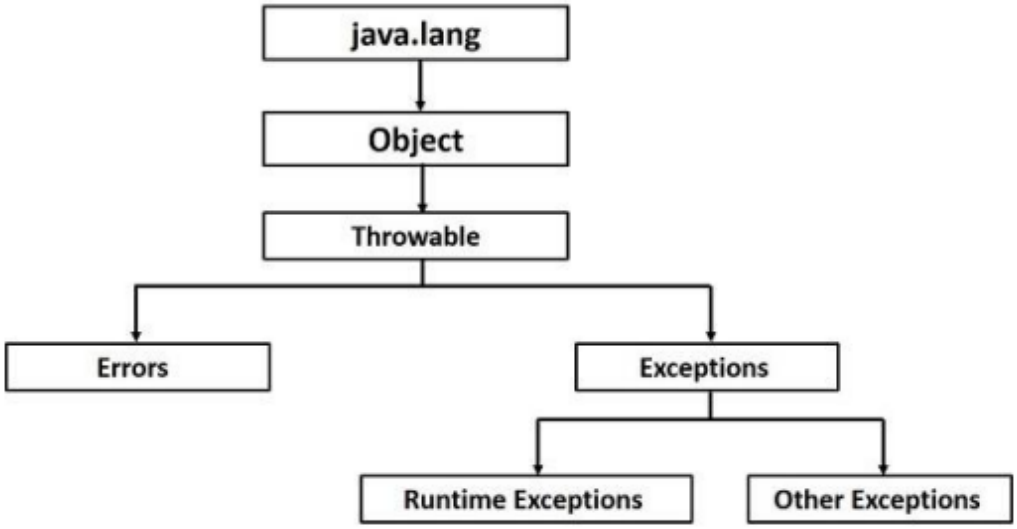
```
  public class Unchecked_Demo {
    public static void main(String args[]) {
      int num[] = {1, 2, 3, 4};
      System.out.println(num[5]);
    }
  }
```

3. Errors:

These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in our code because we can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.
Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.
The Exception class has two main subclasses: IOException class and RuntimeException Class.



Exceptions Methods:

Following is the list of important methods available in the Throwable class.

S.N.	Method & Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage().
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

Syntax

```
try {
  // Protected code
```

```
}catch(ExceptionName e1) {
  // Catch block
}
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest {
  public static void main(String args[]) {
    try {
      int a[] = new int[2];
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

MULTIPLE CATCH BLOCKS:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

Syntax

```
try {
  // Protected code
}catch(ExceptionType1 e1) {
  // Catch block
}catch(ExceptionType2 e2) {
  // Catch block
}catch(ExceptionType3 e3) {
  // Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first

catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example

Here is code segment showing how to use multiple try/catch statements.

```
try {
  file = new FileInputStream(fileName);
  x = (byte) file.read();
}catch(IOException i) {
  i.printStackTrace();
  return -1;
}catch(FileNotFoundException f) // Not valid! {
  f.printStackTrace();
  return -1;
}
```

The Throws/Throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature. We can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly. The following method declares that it throws a RemoteException:

Example

```
import java.io.*;
public class className {
  public void deposit(double amount) throws RemoteException {
    // Method implementation
    throw new RemoteException();
  }
  // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException.

Example

```
import java.io.*;
public class className {
  public void withdraw(double amount) throws RemoteException,
    InsufficientFundsException {
```

```
    // Method implementation
  }
  // Remainder of class definition
}
```

The Finally Block:

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

Syntax

```
try {
  // Protected code
}catch(ExceptionType1 e1) {
  // Catch block
}catch(ExceptionType2 e2) {
  // Catch block
}catch(ExceptionType3 e3) {
  // Catch block
}finally {
  // The finally block always executes.
}
```

Example

```
public class ExcepTest {
  public static void main(String args[]) {
    int a[] = new int[2];
    try {
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }finally {
      a[0] = 6;
      System.out.println("First element value: " + a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

Note:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

THE TRY-WITH-RESOURCES:

Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block. In the following program, we are reading data from a file using **FileReader** and we are closing it using finally block.

Example

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class ReadData_Demo {
  public static void main(String args[]) {
    FileReader fr = null;
    try {
      File file = new File("file.txt");
      fr = new FileReader(file); char [] a = new char[50];
      fr.read(a);   // reads the content to the array
      for(char c : a)
      System.out.print(c);   // prints the characters one by one
    }catch(IOException e) {
      e.printStackTrace();
    }finally {
      try {
        fr.close();
      }catch(IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

try-with-resources, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.

To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block. Following is the syntax of try-with-resources statement.

Syntax

```
try(FileReader fr = new FileReader("file path")) {
  // use the resource
  }catch() {
    // body of catch
  }
}
```

Following is the program that reads the data in a file using try-with-resources statement.
Example

```
import java.io.FileReader;
import java.io.IOException;
public class Try_withDemo {
  public static void main(String args[]) {
    try(FileReader fr = new FileReader("E://file.txt")) {
      char [] a = new char[50];
      fr.read(a);   // reads the contentto the array
      for(char c : a)
      System.out.print(c);   // prints the characters one by one
    }catch(IOException e) {
      e.printStackTrace();
    }
  }
}
```

Following points are to be kept in mind while working with try-with-resources statement.

- To use a class with try-with-resources statement it should implement **AutoCloseable** interface and the **close()** method of it gets invoked automatically at runtime.
- We can declare more than one class in try-with-resources statement.
- While you declare multiple classes in the try block of try-with-resources statement these classes are closed in reverse order.
- Except the declaration of resources within the parenthesis everything is the same as normal try/catch block of a try block.
- The resource declared in try gets instantiated just before the start of the try-block.
- The resource declared at the try block is implicitly declared as final.

User-defined Exceptions:

We can create our own exceptions in Java. Keep the following points in mind when writing our own exception classes:

- All exceptions must be a child of Throwable.
- If we want to write a checked exception that is automatically enforced by the Handle or Declare Rule, we need to extend the Exception class.
- If we want to write a runtime exception, we need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception {
}
```

We just need to extend the predefined **Exception** class to create your own Exception. These are considered to be checked exceptions. The following **InsufficientFundsException** class is a user-

defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example

```
// File Name InsufficientFundsException.java
import java.io.*;
public class InsufficientFundsException extends Exception {
  private double amount;
  public InsufficientFundsException(double amount) {
    this.amount = amount;
  }
  public double getAmount() {
    return amount;
  }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;
public class CheckingAccount {
  private double balance;
  private int number;
  public CheckingAccount(int number) {
    this.number = number;
  }
  public void deposit(double amount) {
    balance += amount;
  }
  public void withdraw(double amount) throws InsufficientFundsException {
    if(amount <= balance) {
      balance -= amount;
    }else {
      double needs = amount - balance;
      throw new InsufficientFundsException(needs);
    }
  }
  public double getBalance() {
    return balance;
  }
  public int getNumber() {
    return number;
  }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
public class BankDemo {

  public static void main(String [] args) {
    CheckingAccount c = new CheckingAccount(101);
    System.out.println("Depositing $500...");
    c.deposit(500.00);
    try {
      System.out.println("\nWithdrawing $100...");
      c.withdraw(100.00);
      System.out.println("\nWithdrawing $600...");
      c.withdraw(600.00);
    }catch(InsufficientFundsException e) {
      System.out.println("Sorry, but you are short $" + e.getAmount());
      e.printStackTrace();
    }
  }
}
```

Common Exceptions:

In Java, it is possible to define two categories of Exceptions and Errors.

- **JVM Exceptions**:

These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException.

- **Programmatic Exceptions**:

These exceptions are thrown explicitly by the application or the API programmers. Examples: IllegalArgumentException, IllegalStateException.

CHAPTER – 2

APPLET AS JAVA APPLICATION

INTRODUCTION TO APPLET:

An applet is a java program that runs in a web browser. Applet is a container class like frame. An applet is a java class that extends the java.applet.Applet class. Applets are designed to be embedded within HTML page. Whenever a user views in HTML page that contains an applet, the code for the applet is downloaded to the user machine. A JVM is required to view an applet. The JVM on the user machine creates an instance of the applet and invokes various method during applet life time.

In order to create a java applet we have to import a package called applet.Applet, for example:

```
import java.applet.Applet;
public class MyApplet extends Applet{
        statements;
}
```

<applet code = "MyApplet.class" height = 300 width = 300></applet>

Save this file as: MyApplet.html or MyApplet.txt or We can write the applet code in the java file as well but should provide in a comment section as below:

```
//<applet code = "MyApplet.class" height = 300 width = 300></applet>
import java.applet.Applet;
public class MyApplet extends Applet{
        statements;
}
```

JVM on the user machine creates an instance of the applet class and invokes various methods during the applet lifetime.

CREATING AN APPLET:

```
import java.awt.*;
import java.applet.Applet;
public class DemoApplet extends Applet{
        public void paint (Graphics g){
                g.drawString("First Applet Program",80,40);
        }
}
```

In order to run this java applet program, we must create a file that should contain a tag called <applet></applet> as mention below:

- <applet code = "DemoApplet.class" height = 300 width = 300></applet>
- File should be saved as DemoApplet.html or DemoApplet.txt

- To run the program, we have a tag called appletviewer such as appletviewer DemoApplet.html or appletviewer DemoApplet.txt.

BASIC METHODS OF APPLET CLASS:

1. void **init**()

This method is called once by the browser or applet viewer when the applet that it has been loaded into the system. It performs the initialization of an applet. Typical examples are initialization of instance variables and GUI components of the applet.

2. void **start**()

This method is called after the **init** method completes execution and every time the user of the browser returns to the HTML on which the applet resides (after browsing to another HTML page). Typical actions include starting an animation or other execution threads.

3. void **paint**(Graphics g)

This method is called automatically every time the applet needs to be repainted. For example, if the user of the browser covers the applet with another open window on the screen, then uncovers the applet, the **paint** method is called. (Inherited from Container)

4. void **stop**()

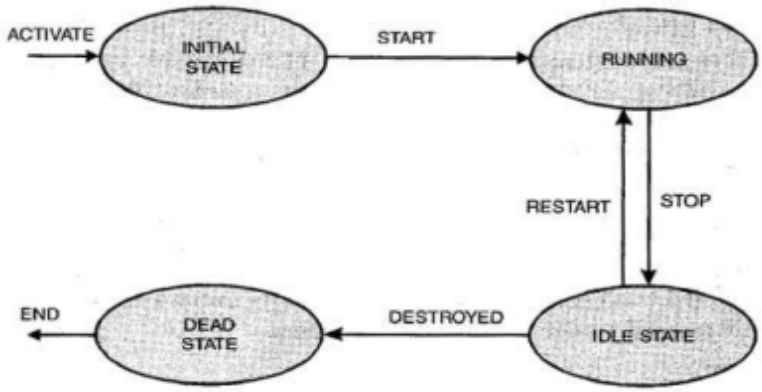
This method is called by the browser or applet viewer when the applet should stop its execution, normally when the user of the browser leaves the HTML page on which the applet resides. Typical actions performed here are to stop execution of animation and threads.

5. void **destroy**()

This method is called by the browser or applet viewer when the applet is being removed from memory, normally when the user of the browser exits the browsing session. This method performs any tasks that are required to destroy any resources that it has allocated.

EXPLAIN APPLET LIFE CYCLE:

Java applet inherits features from the class Applet. Thus, whenever an applet is created, it undergoes a series of changes from initialization to destruction. Various stages of an applet life cycle are depicted in the figure below:



Life Cycle of an Applet

1. **Initial State:**

When a new applet is born or created, it is activated by calling init () method. At this stage, new objects to the applet are created, initial values are set, images are loaded and the colors of the images are set. An applet is initialized only once in its lifetime. It's general form is:

```
public void init( ) {
        //Action to be performed
}
```

2. **Running State:**

An applet achieves the running state when the system calls the start () method. This occurs as soon as the applet is initialized. An applet may also start when it is in idle state. At that time, the start () method is overridden. It's general form is:

```
public void start( ) {
        //Action to be performed
}
```

3. **Idle State:**

An applet comes in idle state when its execution has been stopped either implicitly or explicitly. An applet is implicitly stopped when we leave the page containing the currently running applet. An applet is explicitly stopped when we call stop () method to stop its execution. It's general form is:

```
public void stope
{
        //Action to be performed
}
```

4. **Dead State:**

An applet is in dead state when it has been removed from the memory. This can be done by using destroy () method. It's general form is:

```
public void destroy( ) {
```

```
        //Action to be performed
    }
```

DIFFERENCE BETWEEN APPLET AND APPLICATION:

Applet	Application
❖ Small Program	❖ Large Program
❖ Used to run a program on client Browser	❖ Can be executed on standalone computer system
❖ Applet is portable and can be executed by any JAVA supported browser.	❖ Need JDK, JRE, JVM installed on client machine.
❖ Applet applications are executed in a Restricted Environment	❖ Application can access all the resources of the computer
❖ Applets are created by extending the java.applet.Applet	❖ Applications are created by writing public static void main (String [] s) method.
❖ Applet application has 5 methods which will be automatically invoked on occurrence of specific event	❖ Application has a single start point which is main method

❖ Example:
import java.awt.*;
import java.applet.*;
public class Myclass extends Applet
{
public void init() { }
public void start() { }
public void stop() {}
public void destroy() {}
public void paint(Graphics g) {}
}
	❖ Example:
	public class MyClass
	{
	public static void main(String args[]) {}
	}

EVENT HANDLING IN APPLET:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
//<applet code = "EventApplet.java" height = 300 width = 300></applet>

public class EventApplet extends Applet implements ActionListener{
        Button b;
        TextField tf;
        public void init(){
                tf=new TextField();
                tf.setBounds(30,40,150,20);
                b=new Button("Click");
                b.setBounds(80,150,60,50);
                add(b);
```

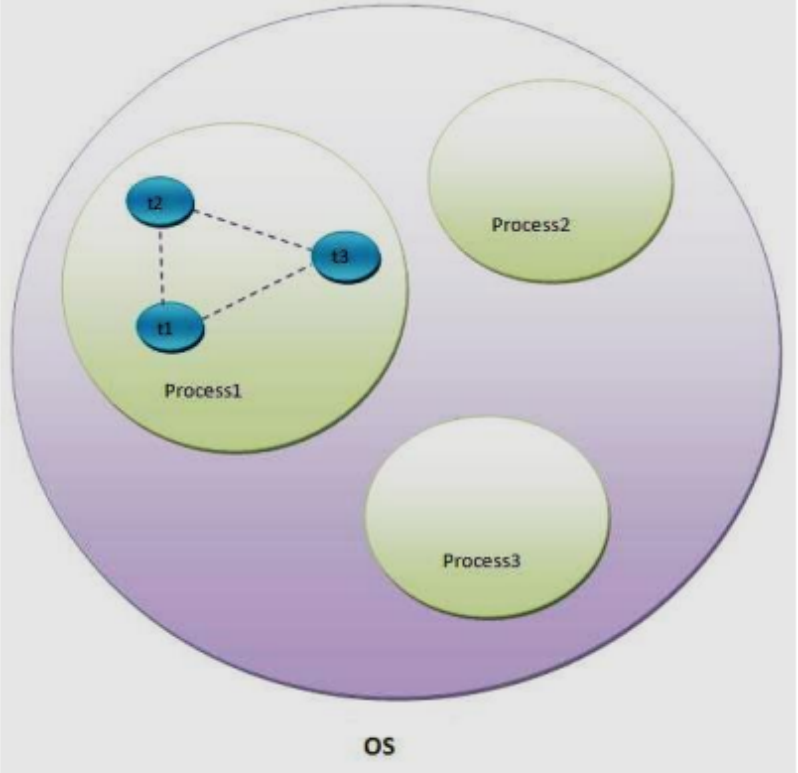
```
                add(tf);
                b.addActionListener(this);
                setLayout(null);
        }
        public void actionPerformed(ActionEvent e){
                tf.setText("Welcome");
        }
}
```

CHAPTER - 3

MULTITHREADING

THREAD:

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution. Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

MULTITHREADING:

Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading:

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
2. We **can perform many operations together so it saves time**.
3. Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

MULTITASKING:

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

1. **Process-based Multitasking (Multiprocessing):**

- ❖ Each process have its own address in memory i.e. each process allocates separate memory area.
- ❖ Process is heavyweight.
- ❖ Cost of communication between the processes is high.
- ❖ Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2. **Thread-based Multitasking (Multithreading):**

- ❖ Threads share the same address space.
- ❖ Thread is lightweight.
- ❖ Cost of communication between the thread is low.

THREAD LIFE CYCLE:

A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state. But for better understanding the threads are explained in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. **New:**

The thread is in new state if we create an instance of Thread class but before the invocation of start() method. It is also referred to as a **born thread.**

2. **Runnable:**

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3. **Running:**

The thread is in running state if the thread scheduler has selected it.

4. **Non-Runnable (Blocked):**

This is the state when the thread is still alive, but is currently not eligible to run.

5. **Terminated:**

A thread is in terminated or dead state when its run() method exits.

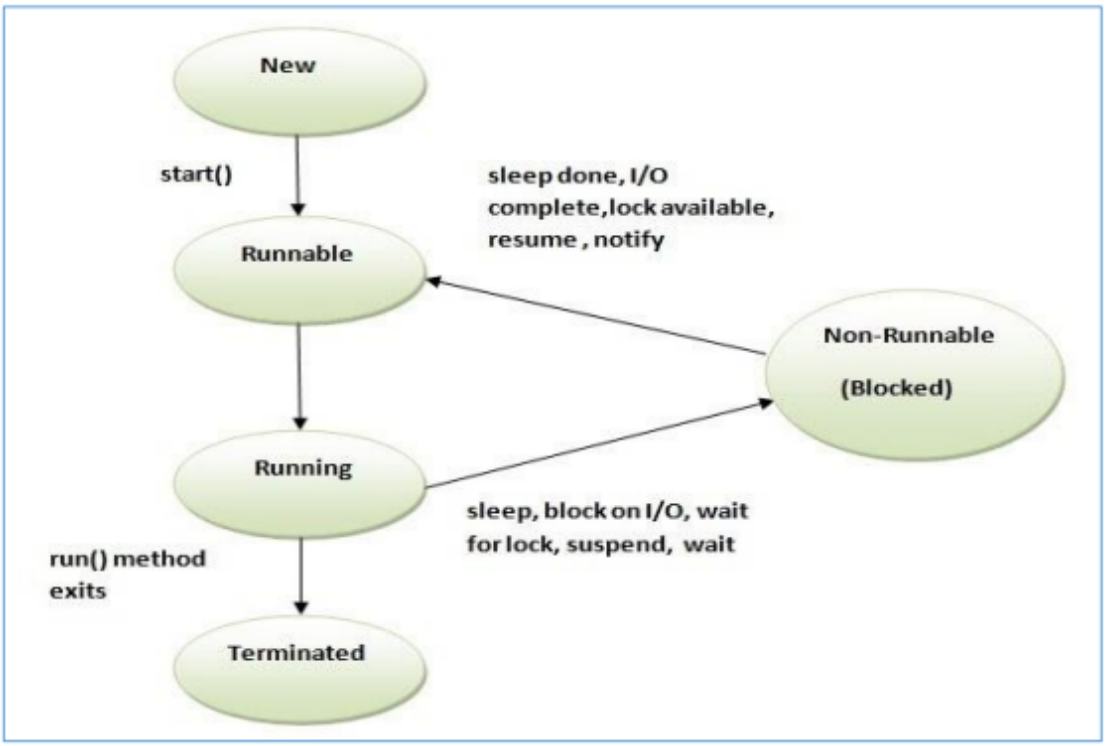


Fig: Life Cycle of Thread

HOW TO CREATE THREAD:

There are two ways to create a thread:

1. **Create a Thread by Implementing a Runnable Interface:**

If our class is intended to be executed as a thread then we can achieve this by implementing a Runnable interface. We need to follow three basic steps:

STEP 1:

As a first step, we need to implement a run() method provided by a Runnable interface. This method provides an entry point for the thread and we put our complete business logic inside this method. Following is a simple syntax of the run() method:

```
public void run( )
```

STEP 2:

As a second step, we will instantiate a Thread object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, ***threadObj*** is an instance of a class that implements the Runnable interface and ***threadName*** is the name given to the new thread.

Step 3:

Once a Thread object is created, we can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of start() method:

```
void start();
```

Example

```
class RunnableDemo implements Runnable {
  private Thread t;
  private String threadName;

  RunnableDemo( String name) {
    threadName = name;
    System.out.println("Creating " +  threadName );
  }
  public void run() {
    System.out.println("Running " +  threadName );
    try {
      for(int i = 4; i > 0; i--) {
        System.out.println("Thread: " + threadName + ", " + i);
        // Let the thread sleep for a while.
        Thread.sleep(50);
      }
    }catch (InterruptedException e) {
      System.out.println("Thread " +  threadName + " interrupted.");
    }
    System.out.println("Thread " +  threadName + " exiting.");
  }
  public void start () {
    System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
    }
  }
}
public class TestThread {
  public static void main(String args[]) {
    RunnableDemo R1 = new RunnableDemo( "Thread-1");
    R1.start();
    RunnableDemo R2 = new RunnableDemo( "Thread-2");
    R2.start();
  }
}
```

2. Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1:

We need to override run( ) method available in Thread class. This method provides an entry point for the thread and we put our complete business logic inside this method. Following is a simple syntax of run() method:

```
public void run( )
```

Step 2:

Once Thread object is created, we can start it by calling start() method, which executes a call to run() method. Following is a simple syntax of start() method:

```
void start( );
```

Example

```
class ThreadDemo extends Thread {
  private Thread t;
  private String threadName;
  ThreadDemo( String name) {
    threadName = name;
    System.out.println("Creating " +  threadName );
  }
  public void run() {
    System.out.println("Running " +  threadName );
    try {
      for(int i = 4; i > 0; i--) {
        System.out.println("Thread: " + threadName + ", " + i);
        // Let the thread sleep for a while.
        Thread.sleep(50);
      }
    }catch (InterruptedException e) {
      System.out.println("Thread " +  threadName + " interrupted.");
    }
    System.out.println("Thread " +  threadName + " exiting.");
  }
  public void start () {
    System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
    }
  }
}
```

```
public class TestThread {
  public static void main(String args[]) {
    ThreadDemo T1 = new ThreadDemo( "Thread-1");
    T1.start();
    ThreadDemo T2 = new ThreadDemo( "Thread-2");
    T2.start();
  }
}
```

THREAD METHODS:

Following is the list of important methods available in the Thread class.

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.

20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.
23. **public static boolean holdsLock(Object x):** Returns true if the current thread holds the lock on the given Object.
24. **public static void dumpStack():** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

PRIORITY OF A THREAD (THREAD PRIORITY):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads.

3 Constants Defined In Thread Class:

```
public static int MIN_PRIORITY
public static int NORM_PRIORITY
public static int MAX_PRIORITY
```

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of Priority of a Thread:

```
class TestMultiPriority extends Thread{
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getName());
  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
 }
 public static void main(String args[]){
  TestMultiPriority m1=new TestMultiPriority();
  TestMultiPriority m2=new TestMultiPriority();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
 }
}
```

THREAD SYNCHRONIZATION:

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. Java programming language provides a very handy way of creating threads and synchronizing their task.

Example:

```
import java.util.Scanner;
class Account{
        private int balance;
        public Account(int bal){
                this.balance = bal;
        }
        public boolean isSufficentBalance(int amount){
                if(balance > amount){
                        return(true);
                }
                else{
                        return(false);
                }
        }
        public void withdraw(int wd){
                balance = balance - wd;
                System.out.println("Withdrawl Amount = " + wd);
                System.out.println("Current Balance = " + balance);
        }
}

class Customer implements Runnable{
        private final Account account;
    private String name;
        public Customer(Account account, String n){
                this.account = account;
          name = n;
        }
    @Override
        public void run(){
                Scanner scan = new Scanner(System.in);
                synchronized(account){
                        System.out.println(name + ",Enter amount: ");
                        int amount = scan.nextInt();
```

```
                    if(account.isSufficentBalance(amount)){
                        System.out.println(name);
                        account.withdraw(amount);
                    }
                    else{
                        System.out.println("Insufficent Balance");
                    }
                }
            }
        }
        public class ThreadExample{
            public static void main(String [] args){
                Account a1 = new Account(1000);
                Customer c1 = new Customer(a1,"Ramesh");
                Customer c2 = new Customer(a1,"Samir");
                Thread t1 = new Thread(c1);
                Thread t2 = new Thread(c2);
                t1.start();
                t2.start();
            }
        }
```

- **Synchronized Method:**

If we declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example:

```
class Table{
    synchronized void printTable(int n){//synchronized method
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }
                catch(Exception e){
                    System.out.println(e);
                }
        }
    }
}
class MyThread1 extends Thread{
  Table t;
  MyThread1(Table t){
      this.t=t;
```

```
    }
    public void run(){
        t.printTable(5);
    }

}
class MyThread2 extends Thread{
    Table t;
    MyThread2(Table t){
        this.t=t;
    }
    public void run(){
        t.printTable(100);
    }
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

- **Synchronized Block:**

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, then we can use synchronized block. If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax:

```
synchronized (object reference expression) {
 //code block
}
```

Example:

```
    class Table{
```

```
 void printTable(int n){
  synchronized(this){//synchronized block
   for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
   }
  }
 }//end of the method
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

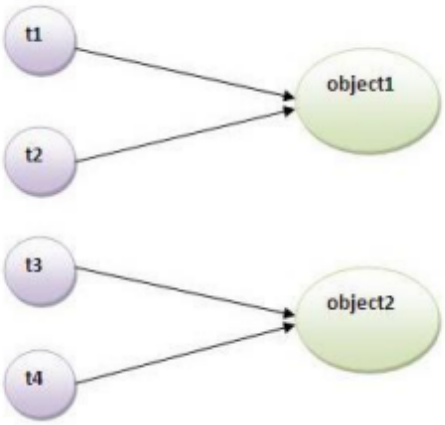
- **Static Synchronization:**

If we make any static method as synchronized, the lock will be on the class not on object.

Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there

can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.



Example:

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table{
 synchronized static void printTable(int n){
  for(int i=1;i<=10;i++){
    System.out.println(n*i);
    try{
      Thread.sleep(400);
    }catch(Exception e){}
  }
 }
}
class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}
class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}
class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}
class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}
public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
```

```
t3.start();
t4.start();
}
}
```

CHAPTER – 4

JAVA INPUT OUTPUT

JAVA INPUT/OUTPUT PACKAGE:

All the input/output operations are handled by java.io package. All the classes that are needed to perform input/output operations are contained inside java.io package. A stream can be defined as a sequence of data and it produce or consumes information. A stream is linked to a physical device by java input/output system. The InputStream is used to read from a source and OutputStream is used for writing data to a destination. Streams represent the source and destination.

BYTE STREAMS:

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream. Following is an example which makes use of these two classes to copy an input file into an output file.

Example:

```
import java.io.*;
public class CopyFile {
  public static void main(String args[]) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;
      in = new FileInputStream("input.txt");
      out = new FileOutputStream("output.txt");
      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
         System.out.print((char)c);
      }
        in.close();
        out.close();
  }
}
```

CHARACTER STREAMS:

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit Unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but

here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file:

Example:

```
import java.io.*;
public class CopyFile {
  public static void main(String args[]) throws IOException {
    FileReader in = null;
    FileWriter out = null;
      in = new FileReader("input.txt");
      out = new FileWriter("output.txt");
      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
         System.out.print((char)c);
      }
        in.close();
        out.close();
  }
}
```

JAVA BUFFERED OUTPUT STREAM CLASS:

Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast. For adding the buffer in an OutputStream, use the BufferedOutputStream class.

Syntax:

```
OutputStream os= new BufferedOutputStream(new FileOutputStream("test.txt"));
```

Declaration:

```
public class BufferedOutputStream extends FilterOutputStream
```

Class Constructors:

Constructor	Description
BufferedOutputStream(OutputStream os)	It creates the new buffered output stream which is used for writing the data to the specified output stream.
BufferedOutputStream(OutputStream os, int size)	It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

Class Methods:

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.
void write(byte[ ] b, int off, int len)	It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset
void flush()	It flushes the buffered output stream.

Example:

```
import java.io.*;
public class BufferedOutputStreamExample{
public static void main(String args[])throws IOException{
    FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
    BufferedOutputStream bout=new BufferedOutputStream(fout);
    String s="Welcome to javaTpoint.";
    byte b[ ]=s.getBytes();
    bout.write(b);
    bout.flush();
    bout.close();
    fout.close();
    System.out.println("success");
}
}
```

JAVA BUFFERED INPUT STREAM CLASS:

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

The important points about BufferedInputStream are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.

Syntax:

```
InputStream in= new BufferedIutputStream(new FileInputStream("test.txt"));
```

Class Declaration:

```
public class BufferedInputStream extends FilterInputStream
```

Class Constructors:

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the BufferedInputStream and saves it argument, the input stream IS, for later use.
BufferedInputStream(InputStream IS, int size)	It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use.

Class Methods:

Method	Description
int available()	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
int read()	It read the next byte of data from the input stream.
int read(byte[] b, int off, int ln)	It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
void close()	It closes the input stream and releases any of the system resources associated with the stream.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readlimit)	It sees the general contract of the mark method for the input stream.
long skip(long x)	It skips over and discards x bytes of data from the input stream.
boolean markSupported()	It tests for the input stream to support the mark and reset methods.

Example:

```
import java.io.*;
public class BufferedInputStreamExample{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("D:\\testout.txt");
    BufferedInputStream bin=new BufferedInputStream(fin);
    int i;
    while((i=bin.read())!=-1){
     System.out.print((char)i);
    }
    bin.close();
    fin.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

JAVA BUFFERED READER CLASS:

Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class.

Class Declaration:

public class BufferedReader **extends** Reader

Class Constructors:

Constructor	Description
BufferedReader(Reader rd)	It is used to create a buffered character input stream that uses the default size for an input buffer.
BufferedReader(Reader rd, int size)	It is used to create a buffered character input stream that uses the specified size for an input buffer.

Class Methods:

Method	Description
int read()	It is used for reading a single character.
int read(char[] cbuf, int off, int len)	It is used for reading characters into a portion of an array.
boolean markSupported()	It is used to test the input stream support for the mark and reset method.
String readLine()	It is used for reading a line of text.
boolean ready()	It is used to test whether the input stream is ready to be read.
long skip(long n)	It is used for skipping the characters.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readAheadLimit)	It is used for marking the present position in a stream.
void close()	It closes the input stream and releases any of the system resources associated with the stream.

Example 1:

```
import java.io.*;
public class BufferedReaderExample {
   public static void main(String args[])throws Exception{
      FileReader fr=new FileReader("D:\\testout.txt");
      BufferedReader br=new BufferedReader(fr);
      int i;
      while((i=br.read())!=-1){
```

```
        System.out.print((char)i);
        }
        br.close();
        fr.close();
    }
}
```

Example 2:

```
import java.io.*;
public class BufferedReaderExample{
public static void main(String args[])throws Exception{
   InputStreamReader r=new InputStreamReader(System.in);
   BufferedReader br=new BufferedReader(r);
   System.out.println("Enter your name");
   String name=br.readLine();
   System.out.println("Welcome "+name);
}
}
```

JAVA BUFFERED WRITER CLASS:

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.

Class Declaration:

public class BufferedWriter **extends** Writer

Class constructors:

Constructor	Description
BufferedWriter(Writer wrt)	It is used to create a buffered character output stream that uses the default size for an output buffer.
BufferedWriter(Writer wrt, int size)	It is used to create a buffered character output stream that uses the specified size for an output buffer.

Class methods:

Method	Description
void newLine()	It is used to add a new line by writing a line separator.
void write(int c)	It is used to write a single character.

void write(char[] cbuf, int off, int len)	It is used to write a portion of an array of characters.
void write(String s, int off, int len)	It is used to write a portion of a string.
void flush()	It is used to flushes the input stream.
void close()	It is used to closes the input stream

Example:

```
import java.io.*;
public class BufferedWriterExample {
public static void main(String[] args) throws Exception {
   FileWriter writer = new FileWriter("D:\\testout.txt");
   BufferedWriter buffer = new BufferedWriter(writer);
   buffer.write("Welcome to javaTpoint.");
   buffer.close();
   System.out.println("Success");
   }
}
```

JAVA FILE READER CLASS:

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class. It is character-oriented class which is used for file handling in java.

Class Declaration:

public class FileReader **extends** InputStreamReader

Class Constructor:

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Class Method:

Method	Description
int read()	It is used to return a character in ASCII form. It returns -1 at the end of file.
void close()	It is used to close the FileReader class.

Example:

```
import java.io.FileReader;
public class FileReaderExample {
```

```
    public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("D:\\testout.txt");
        int i;
        while((i=fr.read())!=-1)
        System.out.print((char)i);
        fr.close();
    }
}
```

JAVA PRINT WRITER CLASS:

Java PrintWriter class is the implementation of Writer class. It is used to print the formatted representation of objects to the text-output stream.

Class Declaration:

public class PrintWriter **extends** Writer

Methods of PrintWriter class:

Method	Description
void println(boolean x)	It is used to print the boolean value.
void println(char[] x)	It is used to print an array of characters.
void println(int x)	It is used to print an integer.
PrintWriter append(char c)	It is used to append the specified character to the writer.
PrintWriter append(CharSequence ch)	It is used to append the specified character sequence to the writer.
PrintWriter append(CharSequence ch, int start, int end)	It is used to append a subsequence of specified character to the writer.
boolean checkError()	It is used to flushes the stream and check its error state.
protected void setError()	It is used to indicate that an error occurs.
protected void clearError()	It is used to clear the error state of a stream.
PrintWriter format(String format, Object... args)	It is used to write a formatted string to the writer using specified arguments and format string.
void print(Object obj)	It is used to print an object.
void flush()	It is used to flushes the stream.
void close()	It is used to close the stream.

Example:

```
import java.io.File;
import java.io.PrintWriter;
public class PrintWriterExample {
```

```
  public static void main(String[] args) throws Exception {
        //Data to write on Console using PrintWriter
    PrintWriter writer = new PrintWriter(System.out);
    writer.write("Javatpoint provides tutorials of all technology.");
 writer.flush();
    writer.close();
//Data to write in File using PrintWriter
    PrintWriter writer1 =null;
      writer1 = new PrintWriter(new File("D:\\testout.txt"));
      writer1.write("Like Java, Spring, Hibernate, Android, PHP etc.");
              writer1.flush();
      writer1.close();
  }
}
```

FILE SEQUENTIAL/RANDOM:

Sequential file access allows data to be read from a file or written to a file from beginning to end. It is not possible to read data starting in the middle of the file, nor is it possible to write data to the file starting in the middle using sequential methods. The input and output streams in Java are sequential.

Random file access allows the programmer to read or write a data at a random position in the file. The programmer specifies a position in the file with a special SEEK operation and then read or write a data at that position. Thus, a random access file has the logical behavior of array and support much faster access then sequential access file.

The main point is that the programmer must be able to specify the position of a data in terms of a numbers of bytes between it and the beginning of the file. To do this the programmer must also know the type of each data and the number of bytes required to store it.

The random file access supports random access file processing. The most convenient constructor expect a file name and a mode as a parameter. The mode 'rw' and 'r' specify read/write and read only access prospectively. We use 'rw' for output files and 'r' for input files.

Example:

```
import java.io.*;
public class RandomAccessDemo{
    public static void main(String [] args) throws IOException{
            char a[] = {'a','e','i','o','u'};
            RandomAccessFile rand = new RandomAccessFile("rand.txt","rw");
            System.out.println("Writing vowel letters to file");
            for(int i=0;i<a.length;i++){
                    rand.writeChar(a[i]);
            }
            System.out.println("Finished Writing");
            rand.close();
```

```
    }
}
```

UNIT-5: JAVA GUI COMPONENTS

1.Window Fundamentals

The most common windows are those that are derived from Panel and Frame. The window derived from Panel is used by an Applet and from Frame is used to create standard window. The class hierarchy for Panel and Frame is given below.

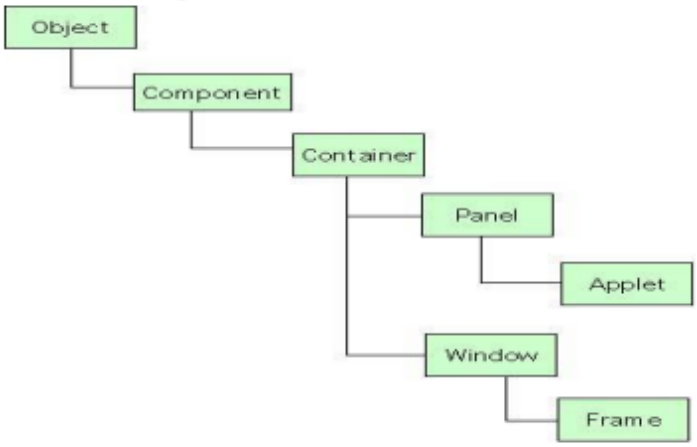


Fig: the class hierarchy for Panel and Frame

1.1 Component

Component is an abstract class that encapsulates all of the attributes of all visual components. All user interface elements that are displayed on the screen and that interact with the user are subclasses of component. It defines hundreds of public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. The component object is also responsible for remembering the current foreground and background colors and the currently selected text font.

1.2 Container

This is a subclass of Component. It has additional methods that allow other Component object to be nested within it. A container is responsible for laying out i.e. positioning any components that it contains. It does this through the use of various managers.

1.3 Panel

Panel is a concrete subclass of Container. It does not add any new methods, it simply implements Container. Panel is a superclass of Applet. When screen output is directed to an Applet, it is drawn on the surface of a Panel object. A Panel is a window that does not contain a title bar, menu bar or border.

Other components can be added to a Panel object by its add() method which is inherited from Container. Once these components are added, we can position and resize them manually using the setLocation(), setSize() or setBound() methods defined by Component.

1.4 Window

This class creates top-level class window. A top level window is not contained within any other object. It sits directly on the desktop. We cannot create window object directly but can use from the subclass of the Window called Frame.

1.5 Frame

It is the subclass of the window and has a title bar, menu bar, borders and resizing corners. The most common method of creating a frame is by using single argument constructor of the Frame class that contains the single string argument which is the title of the window or frame. Then we can add user interface by constructing and adding different components to the container one by one.

The following program is used to create frame with header and label inside the frame

```
import java.awt.*;
public class AwtFrame extends Frame
{
public static void main(String[] args)
{
Frame frm = new Frame("Java AWT Frame");
Label lbl = new Label("Welcome Dhangadhi Engineering College.", Label.CENTER);
frm.add(lbl);
frm.setSize(400,400);
frm.setVisible(true);
}
}
```

In this program we are constructing a label to display "Welcome to Dhangadhi Engineering College." message on the frame. The center alignment of the label has been defined by the **Label.**CENTER and it is auxillary. The program will work fine with just one argument on the Label() constructor. The frame is initially invisible, so after creating the frame it need to visualize the frame by setVisible(true) method.

Few methods used in above programs

add(lbl): This method has been used to add the label to the frame. Method add() adds a component to its container.

setSize (width, height): This is the method of the Frame class that sets the size of the frame or window. This method takes two arguments width (int), height (int).

setVisible(boolean): This is also a method of the Frame class sets the visibility of the frame. The frame will be invisible if we pass the boolean value false otherwise frame will be visible.

2. Layout Manager

Layout managers arrange GUI components in a container for presentation purposes. We can use the layout managers for basic layout capabilities. All layout managers implement the interface LayoutManager (in package java.awt).

2.1 FlowLayout

This is the default layout manager. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right. Class FlowLayout allows GUI components to be left aligned, centered (the default) and right aligned. Its constructors are:

FlowLayout(); It creates the default layout, which centers components and leaves five pixels of space between each component.

FlowLayout(int how); It specify how each line is aligned. And its values are:

FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT

FlowLayout(int how, int horz, int vert); It specify the horizontal and vertical space left between components in horz and vert.

```
//Simple example of flow layout
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class layOut extends JFrame
{
public static void main( String[] args )
{
JFrame fr1=new JFrame("flowlayout demo");
FlowLayout layout = new FlowLayout();
fr1.setLayout( layout );
JLabel floatLabel = new JLabel( "Being Human" );
fr1.add( floatLabel );
layout.setAlignment( FlowLayout.RIGHT );
layout.layoutContainer( fr1 );
fr1.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
fr1.setSize( 400, 100 );
fr1.setVisible( true );
}
}
```

2.2 BorderLayout

The BorderLayout class implements a common layout style for top-level windows. It has four narrow, fixed-width components as the edges and one large area in the center. The four sides are referred to as north, south, east and west. The middle area is called center.

The components placed in the NORTH and SOUTH regions extend horizontally to the sides of the container and are as tall as the components placed in those regions. The EAST and WEST regions expand vertically between the NORTH and SOUTH regions and are as wide as the components placed in those regions. The component placed in the CENTER region expands to fill all remaining space in the layout.

Its constructors are:

BorderLayout(): It creates a default border layout.

BorderLayout(int horz, int vert): It specify the horizontal and vertical space left between components in horz and vert. The BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER BorderLayout.SOUTH
BorderLayout.EAST BorderLayout.WEST
BorderLayout.NORTH

While adding component to the applet the following form of add() method is used void add(Component compobj, Object region)

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class demoBorderlayout extends JFrame
{
private JButton button1;
```

```
private JButton button2;
private JButton button3;
private JButton button4;
private JButton button5;
private BorderLayout layout; // borderlayout object
// set up GUI and event handling
public demoBorderlayout ()
{
super( "BorderLayout Demo" );
layout = new BorderLayout( 5, 5 ); // 5 pixel gaps
setLayout( layout ); // set frame layout
button1 = new JButton( "North" );
button2 = new JButton( "South" );
button3 = new JButton( "East" );
button4 = new JButton( "West" );
button5 = new JButton( "Center" );
add( button1, BorderLayout.NORTH ); // add button to north
add( button2, BorderLayout.SOUTH ); // add button to SOUTH
add( button3, BorderLayout.EAST ); // add button to east
add( button4, BorderLayout.WEST ); // add button to west
add( button5, BorderLayout.CENTER ); // add button to center
}
public static void main( String[] args )
{
demoBorderlayout      borderLayoutFrame = new demoBorderlayout ();
borderLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
borderLayoutFrame.setSize( 300, 200 ); // set frame size
borderLayoutFrame.setVisible( true ); // display frame
}
}
```

2.3 GridLayout

GridLayout lays out components in a two-dimensional grid. While instantiating a GridLayout, it define number of rows and columns. Every Component in a GridLayout has the same width and height. Components are added to a GridLayout starting at the top-left cell of the grid and proceeding left to right until the row is full. Then the process continues left to right on the next row of the grid, and so on.

The constructors supported by GridLayout are:

GridLayout(): It creates a single-column grid layout.

GridLayout(int numrows, int numcols): It creates a grid layout with the specified number of rows and columns.

GridLayout(int numrows, int numcols, int horz, int vert): It specifies the horizontal and vertical space left between components in horz and vert.

Example of gridlayout:

```
import java.awt.GridLayout;
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JButton;
```

```
public class GridLayoutFrame extends JFrame
{
private JButton button1;
private JButton button2;
private JButton button3;
private JButton button4;
private Container container; // frame container
private GridLayout gLayout; // first gridlayout

// no-argument constructor
public GridLayoutFrame()
{
super( "GridLayout Demo" );
gLayout = new GridLayout( 2, 3, 5, 5 ); // 2 by 3; gaps of 5

container = getContentPane(); // get content pane
setLayout( gLayout ); // set JFrame layout

button1= new JButton("One");
button2= new JButton("Two");
button3= new JButton("Three");
button4= new JButton("Four");
add( button1 );
add( button2 );
add( button3 );
add( button4 );

} // end GridLayoutFrame constructor
public static void main( String[] args )
{
GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
gridLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
gridLayoutFrame.setSize( 300, 200 ); // set frame size
gridLayoutFrame.setVisible( true ); // display frame
} // end main
} // end class GridLayoutFrame
```

3. Introduction To Netbeans IDE

It is a free and open source IDE (Integrated Development Environment). It is easy to use and code in netbeans IDE.

Creating GUI components in NetBeans 8.0.1 using drag and drop

Step 1: Create a New Project

Step 2: Choose Java -> Java Application

Step 3: Set a Project Name "demoProject". We should make sure that we deselect the "Create Main Class" checkbox; leaving this option selected generates a new class as the main entry point for the application, but our main GUI window (created in the next step) will serve that purpose, so

checking this box is not necessary. Click the "Finish" button when we are done.

Step 4: Add a JFrame Form-Now right-click the **demoProject** name and choose New -> JFrame Form (JFrame is the Swing class responsible for the main frame for our application.)

Step 5: Name the GUI Class-Next type **demoProjectGUI** as the class name, and **demoPackage** as the package name. The remainder of the fields should automatically be filled in, as shown above. Click the Finish button when we are done.

When the IDE finishes loading, the right pane will display a design-time, graphical view of the **demoProjectGUI**. It is on this screen that we will visually drag, drop, and manipulate the various Swing components like text fields, labels, password fields,checkbox, radio button, etc.

NetBeans IDE Basics

It is not necessary to learn every feature of the NetBeans IDE before exploring its GUI creation capabilities. In fact, the only features that we really need to understand are **the Palette, the Design Area, the Property Editor, and the Inspector.**

The Palette

The Palette contains all of the components offered by the Swing API(JLabel is a text label, JList is a drop-down list, etc.).

The Design Area

The Design Area is where we will visually construct our GUI. It has two views: **source view, and design view.** Design view is the default, as shown below. We can toggle between views at any time by clicking their respective tabs.

The Property Editor

The Property Editor does what its name implies: it allows us to edit the properties of each component. The Property Editor is intuitive to use; in it we will see a series of rows — one row per property — that we can click and edit without entering the source code directly.

The Inspector

The Inspector provides a graphical representation of our application's components. We will use the Inspector only once, to change a few variable names to something other than their defaults.

4. Event Delegation Model, Event Source and Handler, Event Categories, Listeners Interfaces, Adaptor Classes

4.1 Event:

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, software or hardware failure occurs, or an operation is completed. Most of the events relating to the GUI for a program are represented by classes defined in the package java.awt.event. This package also defines the listener interfaces for the various kinds of events that it defines. The package javax.swing.event defines classes for events that are specific to Swing components.

4.2 Types of Event (Event Categories):

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user are known as foreground events. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

- **Background Events** - Those events that do not require the interaction of end user are known as background events. For example: Operating system interrupts, hardware or software failure, if timer expires, an operation completion, etc.

The table below shows most of the important event classes and their description:

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or losses keyboard focus.
ItemEvent	Generated when a checkbox or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.

4.3 The Event Delegation Model

The modern approach that is used to handle an event is delegation event model. It is a standard and consistent mechanism to generate and process event. It implements the following concept:

Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. A source generates an event and sends it to one or more listeners.

Listener (Event Handler) - It is also known as event handler. Listener is an object that is responsible for generating response to an event. Listener waits until it receives an event. Once the event is received, the listeners process the event and then returns.

The advantage of this design is that the application logic that processes events is clearly separated from the user interface logic that generates those events. This is an efficient way of handling the event because the event notifications are sent only to those listeners that want to receive them.

4.4 Event source

A source is an object that generates an event. This occurs when the internal state of that object changes. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.

The general form of registration method is:

public void addTypeListener(TypeListener el)

Here type is the name of event and el is the reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener(KyeListener el). The method that register a mouse motion listener is called addMouseMotionListener(MouseMotionListener el). When an event occurs,

all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event.

The table below shows most of the important event sources and their description:

Event Sources	Description
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the checkbox is selected or deselected.
Choice	Generates item events when the choice is changed.
Menu Item	Generates action events when the menu is selected; generates item events when a checkable menu item is selected or deselected.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text Components	Generates text events when the user enters the character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

4.5 Event Listener Interfaces

Listeners are created by implementing one or more of the interfaces defined by the java.awt.event package. To react to an event, we implement appropriate event listener interfaces. A listener is an object that is notified when an event occurs. It has two requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. We use addTypeListener() method to register the event listener with the source. Similarly, we can also unregister a listener by using removeTypeListener() method. The table below shows most of the important event listener and their description:

Interface	Description
ActionListener	Declares one method to receive action events. void actionPerformed(ActionEvent e)
AdjustmentListener	Declares one method when a scrollbar is manipulated. void adjustmentValueChanged (AdjustmentEvent e)
ComponentListener	Declares four methods to recognize when a component is hidden, moved, resized, or shown. void componentResized(ComponentEvent e) void componentMoved(ComponentEvent e) void componentShown(ComponentEvent e) void componentHidden(ComponentEvent e)
ContainerListener	Declares two methods to recognize when a component is added to or removed from a container. void componentAdded(ContainerEvent e) void

	componentRemoved(ContainerEvent e)
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus. void focusGained(FocusEvent e) void focusLost(FocusEvent e)
ItemListener	Defines one method to recognize when a check box or list item is clicked, when a choice selection is made, or when a checkable menu item is selected or deselected. void itemStateChanged(ItemEvent e)
KeyListener	Defines three method to recognize when a key is pressed, released, or typed. void keyPressed(KeyEvent e) void keyReleased(KeyEvent e) void keyTyped(KeyEvent e)
MouseListener	Defines five method to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released. void mouseClicked(MouseEvent e) void mouseEntered(MouseEvent e) void mouseExited(MouseEvent e) void mousePressed(MouseEvent e) void mouseReleased(MouseEvent e)
MouseMotionListener	Defines two method to recognize when the mouse is dragged or moved. void mouseDragged(MouseEvent e) void mouseMoved(MouseEvent e)
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved. void mouseWheelMoved(MouseWheelEvent e)
TextListener	Defines one method to recognize when a text value changes. void textChanged(TextEvent e)
WindowFocusListener	Defines two method to recognize when a window gains or loses input focus. void windowGainedFocus(WindowEvent e) void windowLostFocus(WindowEvent e)
WindowListener	Defines seven method to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. void windowActivated(WindowEvent e) void windowDeactivated(WindowEvent e) void windowClosed(WindowEvent e) void windowClosing(WindowEvent e) void windowOpened(WindowEvent e) void windowIconified(WindowEvent e) void windowDeiconified(WindowEvent e)

Example of event handling

Example1:

```
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
public class EventExample extends JFrame implements ActionListener
{
public static void main(String args[])
{
EventExample e1=new EventExample();
JFrame f1=new JFrame("Event handling");
JButton b1=new JButton("Click me");
f1.add(b1);
b1.addActionListener(e1);
f1.setVisible(true);
f1.setSize(400,450);
}
public void actionPerformed(ActionEvent e)
{
JOptionPane.showMessageDialog(null, "The button is clicked");
}
}
```

Example 2:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class EventExample extends JFrame
{
public static void main(String args[])
{
JFrame f1=new JFrame("Event handling");
JButton b1=new JButton("Click me");
f1.add(b1);
f1.setVisible(true);
f1.setSize(400,450);
b1.addActionListener(new ActionListener()
{
public void actionPerformed(ActionEvent e)
{
String firstNumber=JOptionPane.showInputDialog( "Enter first integer" );
String secondNumber=JOptionPane.showInputDialog( "Enter second integer" );
// convert String inputs to int values for use in a calculation
int number1 = Integer.parseInt( firstNumber );
int number2 = Integer.parseInt( secondNumber );
int sum = number1 + number2; // add numbers
// display result in a JOptionPane message dialog
JOptionPane.showMessageDialog( null, "The sum is " + sum, "Sum of Two Integers",
JOptionPane.PLAIN_MESSAGE );
}
}
);
}
}
```

4.6 Adaptor Classes

Java provides a special feature, called an adapter class that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface i.e. this class itself write definition for methods which are present in particular event listener interface. Adapter classes are useful when we want to receive and process only some of the events that are handled by a particular event listener interface. We can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events which we want.

For example: Suppose we want to use MouseClicked Event or method from MouseListener, if we do not use adapter class then unnecessarily we have to define all other methods from MouseListener such as MouseReleased, MousePressed ec. But if we use adapter class then we can only define MouseClicked method.

The different adapter classes are;

- ComponentAdapter class
- ContainerAdapter class
- FocusAdapter class
- KeyAdapter class
- MouseAdapter class
- MouseMotionAdapter class
- WindowAdapter class

Example:

```
import java.awt.*;
import java.awt.event.*;
public class WindowAdapterEx extends Frame
{
public WindowAdapterEx()
{
WindowAdapterClose clsme=new WindowAdapterClose();
addWindowListener(clsme);
setTitle("WindowAdapter  frame closing");
setSize(400,400);
setVisible(true);
}
public static void main(String args[])
{
new WindowAdapterEx();
}
public class WindowAdapterClose extends WindowAdapter
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
}
}
```

Advantages Of Adapter Classes

- Assists unrelated classes to work together.
- Provides a way to use classes in multiple ways.

- Increase the transparency of classes.
- Makes a class highly reusable.

5. Swing Libraries, Model View Controller Design Pattern, Different Layout and All Swing Components

5.1 Swing Libraries

Swing is described as a set of customizable graphical components whose look-and-feel (L&F) can be dictated at runtime. Swing is the next-generation GUI toolkit that Sun Microsystems created to enable enterprise development in Java. By enterprise development, we mean that programmers can use Swing to create large-scale Java applications with a wide array of powerful components. In addition, we can easily extend or modify these components to control their appearance and behavior.

Swing is a package (library) in java API that contains everything we need when we are doing GUI. It is actually part of a larger family of Java products known as the Java Foundation Classes (JFC). The main package of swing is javax.swing.

5.2 The Model/View/Controller (MVC) Design Pattern

Swing uses the model-view-controller architecture (MVC) as the fundamental design behind each of its components. MVC breaks GUI components into three elements. Each of these elements plays an important role in how the component behaves. It is a design pattern for the architecture of web applications whose purpose is to achieve a clean separation between three components of most any web application.

- **Model**

The model represents the state data for each component. Model represents knowledge. A model stores data that is retrieved to the controller and displayed in the view. Whenever there is change in the data it is updated by the controller.

There are different models for different types of components. For example, the model of a menu may contain a list of the menu items the user can select from. This information remains the same no matter how the component is painted on the screen; model data is always independent of the component's visual representation.

- **View**

The view refers to how we see the component on the screen. It is a visual representation of its model. A view is attached to its model and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages. A view requests information from the model that it uses to generate an output representation to the user.

As an example we can look at an application window on two different GUI platforms. Almost all window frames have a title bar spanning the top of the window. However, the title bar may have a close box on the left side (like the Mac OS platform), or it may have the close box on the right side (as in the Windows platform). These are examples of different types of views for the same window object.

- **Controller**

The controller is the portion of the user interface that dictates how the component interacts with events. The controller decides how each component reacts to the event. It is a link between the user and the system. It provides the user with input by arranging relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. It receives such user output, translates it into the appropriate messages and passes these messages on to one or more of the views.

A controller can send commands to the model to update the model's state (e.g., editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).

5.3 Different Layouts and Swing Components

Layout means the arrangement of components within the container i.e. placing the components at a particular position within the container. The task of layouting the control is done automatically by the layout manager. The different layout managers are FlowLayout, BorderLayout, Grid Layout, etc.

Swing Components

GUIs are built from GUI components. These are also called controls or widgets. A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition. Swing components are the basic building blocks of an application. The Swing GUI components are defined in the javax.swing package. Swing has a wide range of various components, including buttons, check boxes, labels, panels, sliders, etc.

Component	Description
`JLabel`	Displays uneditable text and/or icons.
`JTextField`	Typically receives input from the user.
`JButton`	Triggers an event when clicked with the mouse.
`JCheckBox`	Specifies an option that can be selected or not selected.
`JComboBox`	A drop-down list of items from which the user can make a selection.
`JList`	A list of items from which the user can make a selection by clicking on any one of them. Multiple elements can be selected.
`JPanel`	An area in which components can be placed and organized.

fig. Some basic swing GUI components

```
import javax.swing.*;
public class SwingComponent
{
public static void main(String args[])
{
JFrame  fr=new JFrame("Swingcomponentdemo");
JPanel panel=new JPanel();
fr.add(panel);
JTextField txt=new JTextField(20);
panel.add(txt);
JButton btn=new JButton("Click");
panel.add(btn);
JCheckBox chk=new JCheckBox();
panel.add(chk);
JComboBox  combo= new JComboBox();
panel.add(combo);
fr.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
fr.setSize( 400, 100 );
fr.setVisible( true );
}
}
```

CHAPTER – 6

NETWORKING IN JAVA

NETWORKING BASICS:

Java Networking is a concept of connecting two or more computing devices together so that we can share resources. Java socket programming provides facility to share data between different computing devices.

Advantage of Java Networking

- Sharing resources
- Centralize software management

NETWORKING TERMINOLOGY

1. **IP Address:**

IP address is a unique number assigned to a node of a network example: 192.168.0.1. It is composed of octets that range from 0 to 255. It is a logical address that can be changed.

2. **Protocol:**

A protocol is a set of rules basically that is followed for communication. For example:

- TCP
- FTP
- Telnet
- SMTP
- POP etc.

3. **Port Number:**

The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications. The port number is associated with the IP address for communication between two applications.

4. **MAC Address:**

MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC.

5. **Connection-Oriented And Connection-Less Protocol:**

In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP. But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is UDP.

6. **Socket:**

A socket is an endpoint between two way communications.

SOCKETS:

Socket is an object which establish a communication link between two ports. A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

The working mechanism of socket is shown in the figure below:

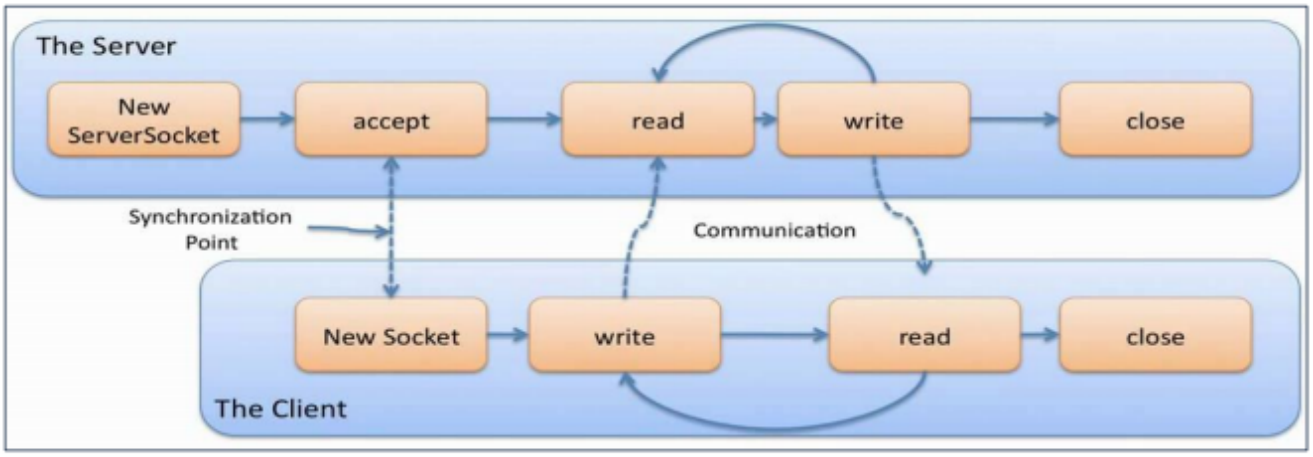


Fig: Overview of Java Socket

PORTS:

Client connect to server through object known as ports. A port server as a channel through which several client can exchange data with the same server or with different servers. Ports are usually specified by numbers. Some ports are dedicated to special server or task. For example: many computers reserve port number 13 for day/time server which allow client to obtain date and time. Port number 80 is reserved for a web server.

Most computers also have 100's or even 1000 of free ports available for use by network application.

PROXY SERVER:

It is an intermediate sever computer that offers a computer network services to allow client to connect with different server. A client connect to the proxy server and then request various services such as a file, connection, web page or other resources available on different server.

The proxy provides the resource either by connecting to a specified server or by serving it from a cache. If the resources have been cached before then the proxy server return them to the client computer. However, if it is not cached it will connect to the relevant servers and request the resources on the behalf of the client computers. Then, it cached resources from the remote servers which in term response to the client.

The advantages of proxy servers are:

a. **Improve Performance:** It can improve performance for group of users because it saves the results of all request for certain amount of time.
b. **Filter Request:** It can filter the content also so, it can be used as a filter or firewall.

c. **Increased Browsing Speed:** It can increase browsing speed due to the concept of caching.
d. **Security:** It acts as an intermediate between user's computer and the internet to prevent from attack and unexpected access. It can also hide IP Address of the client computer.

INTERNET ADDRESS URL:

The URL allow uniquely identifying or address information on the internet. Every browser uses them to identify information on the web. Within java network class library, the URL class provides a simple, concise API to access information across the internet using URL. All URL share the same basic format although some variation is allows.

A URL specification is based on four components:

- ☑ The first is protocol to use, separate from rest of the locator by a colon. Common protocols are http, ftp, gopher, etc.
- ☑ The second component is the host name or IP Address of the host to use. This is delimited on the left by double slashed (://) and on the right by a (/) or a colon (:).
- ☑ The third component is the port number and is an optional parameter. It default to port 80, the predefine http port.
- ☑ The fourth part actual file path.

Example:

```
import java.net.*;
class URLDemo{
    public static void main(String [] args) throws MalformedURLException{
            URL hp = new URL ("http://localhost/project");
            System.out.println("Protocol: "+ hp.getProtocol());
            System.out.println("Port: "+ hp.getPort());
            System.out.println("Host: "+ hp.getHost());
            System.out.println("File: "+ hp.getFile());
    }
}
```

Implementing TCP/IP based Server and Client:

TCP/IP sockets are used to implement reliable bidirectional persistent, point to point, stream based connection between hosts on the internet. There are two kinds of TCP sockets in java: one is for server and the other is for the client.

The following steps occurs when establishing TCP connection between two computer using sockets:

1. The server initialize ServerSocket object, denoting in which port number communication is to start.
2. The server invokes the accept () method of the ServerSocket class this method waits until a client connects to the server on the given port.

3. After the server is waiting a client instantiate a socket object specifying the server name and port number to connect to.
4. The constructor of the socket class attempt to connect the client to the specified server and the port number, if communication is established the client now has socket object capable of communicating with the server.
5. On the serer side the accept () method returns a reference to a new socket on the server that is connected to the client socket.

After the connections are established communication can occur using input/output stream. Each socket has both an output stream and input stream. The client output stream is connected to the server input stream and the client input stream is connected to the server output stream.

TCP is a two way communication protocol so data can be send across stream at the same time.

DATAGRAM (DATAGRAM PACKET, DATAGRAM SERVER AND DATAGRAM CLIENT):

Datagram packet class represents a datagram packet. They are used to implement a connectionless packet delivery service. Each message is route from one machine to another based on solely on information contained within the packet. Multiple packet sent from one machine to another might route differently and might arrival in any order and the packet delivery is not granted.

Datagrams are bundle of information passed between machines once the datagram have been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received there is no assurance that it has not been damage in transmit or that whoever sent it still there to receive a response.

Java implements a datagram on top of the UDP protocol by using two classes as; the Datagram packet object is the data container and the datagram socket is the mechanism used to send and receive datagram packet.

URL CONNECTION:

URL Connection is a general purpose class for accessing the attribute of a remote resources. Once we make a connection to a remote server, we can use URL Connection to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the http protocol specification and also make sense for URL object that are using the http protocol.

To access the actual bits or content information of URL, we create a URL Connection object from URL and using it open connection method is invoke.

Example:

```
import java.net.*;
class URLDemo1{
        public static void main(String [] args) throws MalformedURLException{
                URL hp = new URL("https://www.facebook.com");
                URLConnection hpCon = hp.openConnection();
```

```
                int len = hpCon.getContentLength();
                System.out.println("Content Length: "+len);
        }
}
```

1. Database Connectivity (JDBC):

A database is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A database management system (DBMS) provides mechanisms for storing, organizing, retrieving and modifying data from any users. DBMS allows for the access and storage of data without concern for the internal representation of data.

Java Database Connectivity (JDBC) is an API (Application Programming Interface) for java that allows the Java programmer to access the database. The JDBC API consists of a numbers of classes and interfaces, written in java programming language, which provides a numbers of methods for updating and querying a data in a database. It is a relational database oriented driver. It makes a bridge between java application and database. Java application utilizes the features of database using JDBC.

JDBC helps to write Java applications that manage these three programming activities:

- Connect to a data source, like a database.
- Send queries and update statements to the database.
- Retrieve and process the results received from the database in answer to our query.

ODBC (Open DataBase Connectivity)

ODBC is a standard database access method developed by the SQL Access group. The goal of ODBC is to make it possible to access any data from any application, regardless of which database management system (DBMS) is handling the data. ODBC manages this by inserting a middle layer, called a database driver, between an application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS understands. For this to work, both the application and the DBMS must be ODBC-compliant i.e. the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.

1.1 JDBC Components:

- **The JDBC API:**

  The JDBC API provides programmatic access to relational data from the Java programming language. Using JDBC API, front end java applications can execute query and fetch data from connected database. JDBC API can also connect with multiple applications with same database or same application with multiple databases which can resides in different computers (distributed environment). The JDBC API is part of the Java platform, which includes the Java Standard Edition (Java SE) and the Java Enterprise Edition (Java EE). The JDBC 4.0 API is divided into two packages: java.sql and javax.sql. Both packages are included in the Java SE and Java EE platforms.

- **JDBC Driver Manager**:

  The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. Driver Manager manages all the JDBC Driver loaded in client's system. Driver Manager loads the most appropriate driver among all the Drivers for creating a connection.

- **JDBC Test Suite**:

  The JDBC driver test suite helps us to determine that JDBC drivers will run our program. JDBC Test Suite is used to check compatibility of a JDBC driver with J2EE platform. It also check whether a Driver follow all the standard and requirements of J2EE Environment.

- **JDBC-ODBC Bridge:**

  The Java Software Bridge provides JDBC access via ODBC Bridge (or drivers). JDBC Bridge is used to access ODBC drivers installed on each client machine. The JDBC Driver contacts the ODBC Driver for connecting to the database. When Java first came out, ODBC was a useful driver because

most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available. The ODBC Driver is already installed or come as default driver in windows. In Windows 'Datasource' name can be created using control panel >administrative tools>Data Sources (ODBC). After creating 'data source', connectivity of 'data source' to the 'database' can be checked. Using this 'data source' , we can connect JDBC to ODBC.

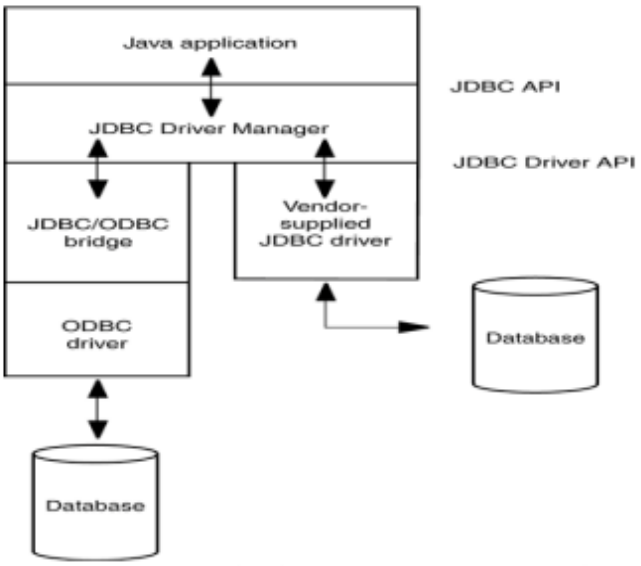


Fig: JDBC-to-database communication path

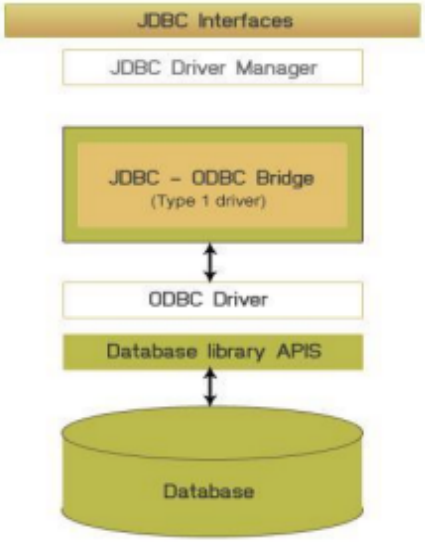
1.2 JDBC Driver Types

JDBC drivers are classified into the following four categories:

❖ JDBC-ODBC BRIDGE DRIVER(TYPE 1)

Features

- Convert the query of JDBC Driver into the ODBC query, which in return pass the data.
- The bridge requires deployment and proper configuration of an ODBC driver.
- JDBC-ODBC is native code not written in java.
- Nowadays in most cases it is only being used for the educational purposes.
- The connection occurs as follows -- Client -> JDBC Driver -> ODBC Driver -> Database.



Pros

A type-1 driver is easy to install and handle.

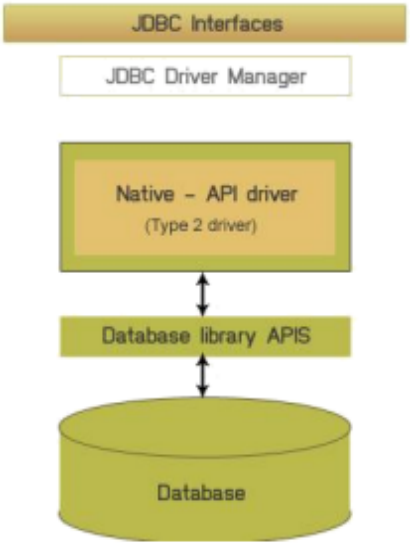
Cons

- Extra channels in between database and application made performance overhead.
- Needs to be installed on client machine.
- Not suitable for applet, due to the installation at clients end.

❖ Native-API (Type-2) Driver

Features

The type 2 driver need libraries installed at client site. For example, we need "mysqlconnector.jar" to be copied in library of java kit. It is not written in java entirely because the non-java interfaces have the direct access to database. It is written partly in Java and partly in native code. When we use such a driver, we must install some platform-specific code in addition to a Java library. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.



Pros

- Type 2 driver has additional functionality and better performance than Type 1.
- Has faster performance than type 1,3 and 4,since it has separate code for native APIS.

Cons

- Library needs to be installed on the client machine.
- Due to the client side software demand, it can't be used for web based application.
- Platform dependent.
- It doesn't support "Applets".

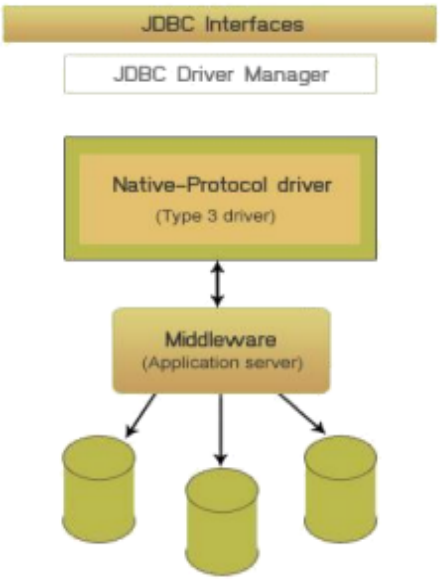
❖ Network-Protocol (Type 3) driver

Features

- It is also known as JDBC-Net pure Java.
- It has 3-tier architecture.
- It can interact with multiple database of different environment.
- The JDBC Client driver written in java communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database

commands for that database.

- The connection occurs as follows--Client -> JDBC Driver -> Middleware-Net Server -> Any Database.



Pros

- The client driver to middleware communication is database independent.
- Can be used in internet since there is no client side software needed.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

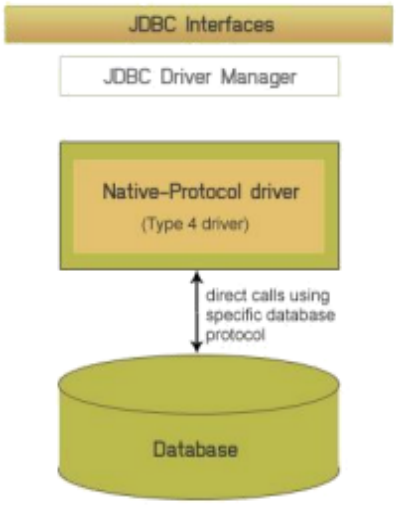
Cons

- Needs specific coding for different database at middleware.
- Due to extra layer in middle can result in time-delay.

❖ Native Protocol (Type 4) Driver

Features

Also known as Direct to Database Pure Java Driver .It is entirely in java. It interacts directly with database generally through socket connection. It is platform independent. It directly converts driver calls into database protocol calls. MySQL's Connector/J driver is an example of Type 4 driver.



Pros

- Improved performance because no intermediate translator like JDBC or middleware server.
- All the connection is managed by JVM, so debugging is easier.

2. Creating JDBC Application:

There are following six steps involved in building a JDBC application:

- **Import the packages:** We should include the packages containing the JDBC classes needed for database programming. We use,

  import java.sql.*

- **Register (Load) the JDBC driver:** We should initialize a driver so that we can open a communications channel with the database. We can load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself.

  Syntax: Class.forName (String ClassName)
  Example:Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

- **Creating a jdbc Connection:** We can use the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with the database. The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager class manages the JDBC drivers that are installed on the system and getConnection() method is used to establish a connection to a database. This method uses a jdbc url and username, password (not compulsory in every DBMS) to establish a connection to the database and returns a connection object.

  Syntax: Connection con=DriverManager.getConnection(url);

- **Creating a jdbc Statement object:** Once a connection is obtained we can interact with the database. To execute SQL statements, we need to instantiate a Statement object from our connection object by using the createStatement() method.
  Syntax: Statement statement = con.createStatement();
  A statement object is used to send and execute SQL statements to a database.

- **Executing a SQL statement with the Statement object, and returning a jdbc resultSet:** Object of type Statement is used for building and submitting an SQL statement to the database. The Statement class has three methods for executing statements i.e. executeQuery(), executeUpdate(), and execute(). For a SELECT statement, executeQuery() method is used . For statements that create or modify tables, executeUpdate() method is used. execute() executes an SQL statement that is written as String object.

  ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

- **Clean up the environment:** We should explicitly close all database resources after the task is complete.

Sample Code:

```
//STEP 1. Import required packages
import java.sql.*;
public class FirstExample {
  static final String JDBC_DRIVER = " sun.jdbc.odbc.JdbcOdbcDriver ";
  static final String DB_URL = "jdbc:odbc:EMP";

  public static void main(String[] args) {
  Connection conn = null;
  Statement stmt = null;
  try{
    //STEP 2: Register JDBC driver
    Class.forName(JDBC_DRIVER);
    //STEP 3: Open a connection
    System.out.println("Connecting to database...");
    conn = DriverManager.getConnection(DB_URL);
    //STEP 4: create a statement object
    System.out.println("Creating statement...");
    stmt = conn.createStatement();
   //STEP 5:Execute a SQL statement with the Statement object, and returning a jdbc resultSet
    String sql = "SELECT id, first, last, age FROM Employees";
    ResultSet rs = stmt.executeQuery(sql);
    //Extract data from result set
    while(rs.next()){
      //Retrieve by column name
      int id  = rs.getInt("id");
      int age = rs.getInt("age");
      String first = rs.getString("first");
      String last = rs.getString("last");
      //Display values
      System.out.print("ID: " + id);
      System.out.print(", Age: " + age);
      System.out.print(", First: " + first);
      System.out.println(", Last: " + last);
    }
    //STEP 6: Clean-up environment
    rs.close();
    stmt.close();
    conn.close();
  }catch(SQLException se){
        JOptionPane.showMessageDialog(null, “SQL exception caught”);
  }catch(Exception e){
        JOptionPane.showMessageDialog(null, “exception caught”);
  }
  }//end main
}//end FirstExample
```

OUTPUT:

```
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Dipendra, Last: Pandey
ID: 101, Age: 25, First: Madan, Last: Pandey
ID: 102, Age: 30, First: Ram, Last: Bhatt
ID: 103, Age: 28, First: Hari, Last: Pant
```

3. Types of Statement Objects

Generally there are three types of statement objects. They are: Statement, PreparedStatement and CallableStatement. These statement interfaces define the methods and properties that enable us to send SQL or PL/SQL commands and receive data from our database. They also define methods that help bridge data type differences between Java and SQL data types used in a database.

Generally, three types of parameters exist: IN, OUT, and INOUT.

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. We bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. We retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. We bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

3.1 The Statement:

The Statement object is used for general-purpose access to our database. It represents the base statements interface. It is suitable to use the Statement only when we know that we will not need to execute the SQL query multiple times. The Statement interface cannot accept parameters. In contrast to PreparedStatment, the Statement does not offer support for the parameterized SQL queries. Before we can use a Statement object to execute a SQL statement, we need to create the object using the Connection object's createStatement( ) method. Statements would be suitable for the execution of the DDL statements such as CREATE, ALTER, DROP.

3.2 PreparedStatements

The PreparedStatement is derived from the more general class, Statement. The PreparedStatement interface extends the Statement interface which gives us added functionality over a generic Statement object. This statement gives us the flexibility of supplying arguments dynamically. It is more efficient to use the PreparedStatement because the SQL statement that is sent gets pre-compiled in the DBMS. We can also use PreparedStatement to safely provide values to the SQL parameters, through a range of setter methods (i.e. setInt (int, int), setString (int, String), etc.). We can execute the same query repeatedly with different parameter values. The PreparedStatement interface accepts input parameters at runtime i.e. the PreparedStatement object only uses the IN parameter. Just as we close a Statement object, we should also close the PreparedStatement object.

Example:

```
try {
        connection = DriverManager.getConnection(
          DATABASE_URL);
   PreparedStatement pstmt = connection.prepareStatement(
                      "INSERT INTO authors VALUES(?, ?, ?)");
pstmt.setInt(1,19);
pstmt.setString(2, "Navin");
pstmt.setString(3,"Sharma");

pstmt.execute();

}catch ( SQLException e )  {
       e.printStackTrace();
    } // end catch
```

The three question marks (?) in the the preceding SQL statement's last line are placeholders for values that will be passed as part of the query to the database. Before executing a PreparedStatement, the program must specify the parameter values by using the Prepared- Statement interface's set methods.
For the preceding query, parameters are int and strings that can be set with Prepared- Statement method setInt and setString.
Method setInt's and setString's first argument represents the parameter number being set, and the second argument is that parameter's value. Parameter numbers are counted from 1, starting with the first question mark (?).
Interface PreparedStatement provides set methods for each supported SQL type. It's important to use the set method that is appropriate for the parameter's SQL type in the database—SQLExceptions occur when a program attempts to convert a parameter value to an incorrect type.

3.3 CallableStatement

The CallableStatement extends PreparedStatement interface. It is used when we want to access database stored procedures. The main advantage of this statement is that it adds a level of abstraction, so the execution of stored procedures does not have to be DBMS-specific. The CallableStatement interface can also accept runtime input parameters. The CallableStatement object can use all three parameters IN, OUT, and INOUT. The output parameters needs to be explicitly defined through the corresponding registerOutParameter() method, whereas the input parameters are provided in the same manner as with the PreparedStatement. Just as we close other Statement object, we should also close the CallableStatement object.

4. Types of ResultSet, ResultSetMetadata

4.1 ResultSet

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. A ResultSet object can be created through any object that implements the Statement interface, including PreparedStatement, CallableStatement, and RowSet.

We access the data in a ResultSet object through a cursor. This cursor is a pointer that points to one row of data in the ResultSet. Initially, the cursor is positioned before the first row. The method **ResultSet.next()** moves the cursor to the next row. This method returns false if the cursor is positioned after the last row. This method repeatedly calls the ResultSet.next() method with a while loop to iterate through all the data in the ResultSet.

ResultSet Interface

The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics. These characteristics are **type, concurrency, and cursor holdability.**

4.2 Types of ResultSet

The type of a ResultSet object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the ResultSet object. There are generally three different ResultSet types:

- **TYPE_FORWARD_ONLY:** The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- **TYPE_SCROLL_INSENSITIVE:** The result set can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- **TYPE_SCROLL_SENSITIVE:** The result set can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE_FORWARD_ONLY.

Note: Not all databases and JDBC drivers support all ResultSet types. The method DatabaseMetaData.supportsResultSetType returns true if the specified ResultSet type is supported and false otherwise.

4.3 ResultSetMetadata

The metadata describes the ResultSet's contents. Programs can use metadata programmatically to obtain information about the ResultSet's column names and types. ResultSetMetaData method getColumnCount is used to retrieve the number of columns in the ResultSet.

5. CRUD Operations In Database:

The operations such as insertion, creation, deletion, updating and selection of the tabular data are known as CRUD operation in database. Following are the CRUD operations with the appropriate examples;

INSERT OPERATION:

Example1: Program that insert data in the student table through java

```
import java.sql.*;
class TestJdbc
{
public static void main(String arg[])
{
        try{
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                Connection cn = DriverManager.getConnection("jdbc:odbc:ram");
                 String str = "insert into student values(12,'Ram', 'dhangadi')";
                Statement stat = cn.createStatement();
                stat.executeUpdate(str);
                cn.close();
        }
        catch(ClassNotFoundException e)
        {
                System.out.println("Cannot insert into the table");
        }
        catch(SQLException e) {}
}
}
```

Example2: Java program that is used to insert multiple records in a table Customers

```
import java.sql.*;
class JdbcInsertMultipleRows {
  public static void main (String[] args) { try
    {
      String url = "jdbc:odbc:ram";
      Connection conn = DriverManager.getConnection(url,"","");
      Statement st = conn.createStatement();
      st.executeUpdate("INSERT INTO Customers " + "VALUES (1001, 'Ram','Kathmandu', 2001)");
      st.executeUpdate("INSERT INTO Customers " + "VALUES (1002,'Shyam','Pokhara', 2004)");
      st.executeUpdate("INSERT INTO Customers " + "VALUES (1003, 'Hari','Nepalgung', 2003)");
      st.executeUpdate("INSERT INTO Customers " + "VALUES(1004,'Krishna','Dhangadhi',2001)");
      conn.close();
    }
      catch (SQLException e)
      {
      System.out.println("Cannot insert into the table");
       }

  }
```

```
}
```

SELECT OPERATION

Example: Program that is used to select multiple rows from the table of SQL database

```
import java.sql.*;
class Query {
   public static void main (String[] args) { try
      {
         String url = "jdbc:odbc:ram";
         Connection conn = DriverManager.getConnection(url,"","");
         Statement stmt = conn.createStatement();
         ResultSet rs;
         rs = stmt.executeQuery("SELECT name FROM student WHERE id = 1001");
         while ( rs.next() ) {
            String lastName = rs.getString("name");
                  System.out.println(lastName);
         }
         conn.close();
      } catch (SQLException e) {
      System.out.println("Cannot select from the table");

      }
   }
}
```

UPDATE OPERATION:

Example: Program that is used to modify the record in the table of SQL database

```
import java.sql.*;
 class UpdateQuery {
   public static void main (String[] args) { try {
                  String url = "jdbc:odbc:ram";
                  Connection conn = DriverManager.getConnection(url,"","");
                   Statement stmt = conn.createStatement();
                  ResultSet rs;
                  stmt.executeUpdate("UPDATE student SET name = 'Krishna' WHERE id = 1001");
                  rs = stmt.executeQuery("SELECT name from student where id = 1001");
                  while ( rs.next() )
                  {
                           String lastName = rs.getString("name");
                           System.out.println(lastName);
                  }
                  conn.close();
                           } catch (SQLException e) {
                           System.out.println("Cannot update the table");
```

```
        }
        }
        }
```

DELETE OPERATION

Example: Program that is used to delete record from the table of SQL database

```
import java.sql.*;
class DeleteQuery {
   public static void main (String[] args) {
    try {
                String url = "jdbc:odbc:ram";
                Connection conn = DriverManager.getConnection(url,"","");
                 Statement stmt = conn.createStatement();
                ResultSet rs;
                stmt.executeUpdate("DELETE from student WHERE id = 1001");
                conn.close();
        } catch (SQLException e) {
                        System.out.println("Cannot delete the data");

     }
   }
}
```

CREATE OPERATION

Example: Program that is used to create table in the given database

```
import java.sql.*;
class CreateTableQuery {
   public static void main (String[] args) {
    try {
                String url = "jdbc:odbc:ram";
                Connection conn = DriverManager.getConnection(url,"","");
                 Statement stmt = conn.createStatement();
                String sql = "create table college(cname varchar(255), address varchar(255))";
                stmt.executeUpdate(sql);
                System.out.println("Table is created in the given database");
                conn.close();
     } catch (SQLException e) {
                        System.out.println("Cannot create the table");

     }
   }
}
```

6. JDBC and AWT

AWT is responsible for creating the GUI components and JDBC is responsible for database connectivity. java.awt package has all the classes that helps in creation of the GUI components.

//Example for AWT, JDBC

```
import java.sql.*;
import java.awt.*;
import java.awt.event.*;

class AwtJdbcTest extends Frame implements ActionListener {
   Connection con;
   PreparedStatement ps;
   ResultSet rs;
   Label lno, lna, ladd;
   TextField tno, tna, tadd;
   Button bins;
   AwtJdbcTest() {
      FlowLayout fl;
      fl = new FlowLayout();
      setLayout(fl);
      setSize(500, 400);
      setBackground(Color.GREEN);
      lno = new Label("RollNO");
      add(lno);
      tno = new TextField(20);
      add(tno);
      lna = new Label("Name");
      add(lna);
      tna = new TextField(20);
      add(tna);
      ladd = new Label("Address");
      add(ladd);
      tadd = new TextField(20);
      add(tadd);
      bins = new Button("INSERT");
      bins.addActionListener(this);
      add(bins);
      setVisible(true);
      dbConnect();
          addWindowListener(new WindowAdapter(){
                        public void windowClosing(WindowEvent e)
                         {
                           System.exit(0);
```

```
                        }});

    } // constructor closing
    void dbConnect() {
       try{
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
          con = DriverManager.getConnection("jdbc:odbc:school");
          String q = "INSERT INTO student VALUES(?, ?, ?)";
          ps = con.prepareStatement(q);
         } // try
       catch(Exception e) {
          e.printStackTrace();
       } //catch
    } // dbConnect() closing

    public void actionPerformed(ActionEvent ae) {
       try{
          int no;
          String name, add;
          no = Integer.parseInt(tno.getText());
          name = tna.getText();
          add = tadd.getText();
            ps.setInt(1, no);
             ps.setString(2, name);
             ps.setString(3, add);
             ps.executeUpdate();
          JOptionPane.showMessageDialog(null,"data inserted successfully");
       } // try
       catch(Exception e)    {
          e.printStackTrace();
       } //catch
    } // actionPerformed
    public static void main(String[] args) {
       new AwtJdbcTest();
    } // main
} // class
```

7. Connection Pooling

Object pooling is used as a technique to improve application performance. Object pooling is effective for two simple reasons. First, the run time creation of new software objects is often more expensive in terms of performance and memory than the reuse of previously created objects. Second, garbage collection is an expensive process and when we reduce the number of objects to clean up, we generally reduce the garbage collection load. Objects which do not have short lifetimes are used for pooling. Pooling is the concept that contains various pooled objects such as database connections, process threads, server sockets or any other kind of object that may be expensive to create from scratch. When we create any application, it first asks the pool for objects. If there are no objects in the pool, then they will be newly created otherwise they are obtained from the pool. And when the application has finished with the object it is returned to the pool rather than destroyed. So the pooled object can be reused in the future and hence are less expensive in contrast to create it from scratch.

Database connections are often expensive to create because of the overhead of establishing a network connection and initializing a database connection. JDBC connection pooling is conceptually similar to any other form of object pooling. It setup the pool of database connection objects and instead of creating the database object many times, it simply use the object from the pool when the database object is not being used. JDBC connections are both expensive to initially create and then maintain over time. Therefore, they are an ideal resource to pool.

CHAPTER – 8

GENERICS

INTRODUCTION TO GENERICS:

Generics was first introduced in jdk 5 and generic in simple term refers general. By using generics, it is possible to create a single class that automatically works with different types of data. A class, interface or method that operates on a parameterized type is called generic or generic class or generic method.

Parameterized types are important because they enables us to create classes, interface and methods in which the type of data upon which they operates is specified as a parameter.

It is important to understand that java has always given us the ability to create generalized classes, interface and methods by operating through references of type object. Because object is the super class of all other classes, and object reference can refer to any type of object. Thus, in pre-generics code, generalized classes, interface and methods used object reference to operate on various type of object. Generics expands our ability to reuse code.

Generics programs have the following properties:

1. **Generics Works Only with Object:**

When declaring an instance of a generics type, the type of argument passed to the type of parameter must be a class type. We cannot use a primitive type such as int, char, double, float.

2. **Generics Type Differ Based on Their Type Argument:**

A reference of one specific version of generics type is not type compatible with another version of the same generic type.

3. **Stronger Type Check at Compile Time:**

A java compiler applies strong type checking to generics code and issue errors if the code violets type safety. Fixing compile time error is easier than fixing runtime error, which can be difficult to find.

4. **Enabling Programmers to Implement Generics Algorithms:**

By using generics programmer implement generics algorithms that works on collection of different types, can be customized and are type safe and easier to read.

GENERICS CLASS WITH PARAMETERS:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example:

```
class Gen <T> {
   T obj;
   Gen(T o){
      obj = o;
   }
   T getObj(){
      return obj;
   }
   void showType(){
      System.out.println("Type of T is: "+obj.getClass().getName());
   }
}
public class GenericsDemo{
   public static void main(String [] args){
      Gen <Integer> obj1 = new Gen <Integer>(88);
      obj1.showType();
     System.out.println("Input Data: "+obj1.getObj());
     System.out.println();
     Gen <String> obj2 = new Gen <String>("Ramesh");
      obj2.showType();
     System.out.println("Input Data: "+obj2.getObj());
   }
}
```

GENERAL FORM OF A GENERIC CLASS:

Syntax:

```
class classname <type parameter list>{
    statements;
}
```

Syntax for declaring a reference to a generic classes:

classname <type-argument-list> variable = new classname <type-argument-list>(constructor-argument-list);

CREATING A GENERIC METHOD, CONSTRUCTOR, AND INTERFACE:

1. GENERIC METHOD:

Methods inside a generic class can make use of class type parameter and are automatically generic relative to the type parameter. It is possible to declare a generic method that uses one or

more type parameters of its own. It is also possible to create a generic method that is enclosed with non-generic class.

Example:

```
public class GenericMethodTest {
  // generic method printArray
  public static < E > void printArray( E[] inputArray ) {
    // Display array elements
      int i;
    for(i=0;i<inputArray.length;i++) {
      System.out.print(inputArray[i]+" ");
    }
    System.out.println();
  }

  public static void main(String args[]) {
    // Create arrays of Integer, Double and Character
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println("Array integerArray contains:");
    printArray(intArray);   // pass an Integer array

    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // pass a Double array

    System.out.println("\nArray characterArray contains:");
    printArray(charArray);   // pass a Character array
  }
}
```

2. Generic Constructor:

A constructor can be declared as generic, independently of whether the class that the constructor is declared in is itself generic. A constructor is generic if it declares one or more type variables. These type variables are known as the formal type parameters of the constructor. The form of the formal type parameter list is identical to a type parameter list of a generic class or interface.

Example:

```
class Test {
        //Generics constructor
        public <T> Test(T item){
                System.out.println("Value of the item: " + item);
                System.out.println("Type of the item: " + item.getClass().getName());
```

```
                System.out.println();
        }
    }
    public class GenericsTest {
        public static void main(String args[]){
                Test test1 = new Test("Test String.");
                Test test2 = new Test(100);
                Test test3 = new Test(100.45f);
                Test test4 = new Test(100.45d);
        }
    }
```

3. Generic Interface:

Generic interfaces are specified just like generic classes. It offers two benefits:

a. It can be implemented for different types of data.
b. It allows us to put constraints on the type of data for which interface can be implemented.

General syntax for generic interface:

interface interface-name <type-parameter-list>

Here, type-parameter-list is a comma separated list of type parameters.

When generic interface is implemented we must specify the type arguments as shown below:

class class-name <type-parameter-list> implements interface-name <type-argument-list>

Example:

```
interface Exmaple<T>{
  public void add(T t);
}
public class Box implements Example <Integer> {

  int t;
  public void add(Integer t) {
    this.t = t;
  }

  public int get() {
    return t;
  }

   public static void main(String[] args) {
    Box a = new Box();
    a.add(10);
    System.out.print(a.get());
```

```
    }
}
```

POLYMORPHISM IN GENERICS:

To understand generics programming we need to understand about the base type and generics type. We can make polymorphic in base type while it is not supported in generic type. The general form is:

List<String> strings = new ArrayList<String>(); // This is valid
List<Object> objects = new ArrayList<String>(); // This is not valid

- List is the base type.
- String is the generic type
- ArrayList is the base type.

There is a simple rule. The type of declaration must match the type of object we are creating. We can make polymorphic references for the base type NOT for the generic type. Hence the first statement is valid while second is not.

- ArrayList<String> list = new ArrayList<String>(); // Valid. Same base type, same generic type
- List<String> list1 = new ArrayList<String>(); // Valid. Polymorphic base type but same generic type
- ArrayList<Object> list2 = new ArrayList<String>(); // Invalid. Same base type but different generic type.
- List<Object> list3 = new ArrayList<String>(); // Invalid. Polymorphic base type different generic type

Example:

Bike extends Vehicle
Car extends Vehicle
Every class has service() method

Vehicle class

```
public class Vehicle {
    public void service() {
            System.out.println("Generic vehicle servicing");
    }
}
```

Bike Class

```
public class Bike extends Vehicle {
    @Override
    public void service(){
            System.out.println("Bike specific servicing");
    }
```

```
}
```

Car Class

```
public class Car extends Vehicle {
    @Override
    public void service() {
            System.out.println("Car specific servicing");
    }
}
```

Main Class

```
import java.util.ArrayList;
import java.util.List;

public class Mechanic {
    public void serviceVehicles(List<Vehicle> vehicles){
      for(Vehicle vehicle: vehicles){
          vehicle.service();
      }
    }

    public static void main(String[] args) {

     List<Vehicle> vehicles = new ArrayList<Vehicle>();
     vehicles.add(new Vehicle());
     vehicles.add(new Vehicle());

     List<Bike> bikes = new ArrayList<Bike>();
     bikes.add(new Bike());
     bikes.add(new Bike());

     List<Car> cars = new ArrayList<Car>();
     cars.add(new Car());
     cars.add(new Car());

     Mechanic mechanic = new Mechanic();
     mechanic.serviceVehicles(vehicles); // This works fine.
     mechanic.serviceVehicles(bikes); // Compiler error.
     mechanic.serviceVehicles(cars); //Compiler error.
   }
}
```

mechanic.serviceVehicles(vehicles) works fine because we are passing ArrayList <Vehicle> to the method that takes List <Vehicle>. mechanic.serviceVehicles (bikes) and

mechanic.serviceVehicles (cars) does not compile because we are passing ArrayList <Bike> and ArrayList<Car> to a method that takes List<Vehicle>.