**1.**Java's I/O classes are part of the java.io package and provide a wide range of functionalities for reading from and writing to different data sources such as files, network connections, and memory buffers.

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Java FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class.

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data.

**2.**Serialization in Java is a mechanism of writing the state of an object into a byte-stream. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called deserialization where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform.

It is mainly used to travel object's state on the network (that is known as marshalling).

To save/persist state of an object.

To travel an object across a network

For serializing the object, call the writeObject() method of ObjectOutputStream class

for deserialization call the readObject() method of ObjectInputStream class.

 To implement the Serializable interface for serializing the object

3.Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

 Built-In Java Annotations used in Java code

@Override

@SuppressWarnings

@Deprecated

Built-In Java Annotations used in other annotations

@Target

@Retention

@Inherited

@Documented

**@Override** annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

class Animal{

   void eatSomething() { System.out.println("eating something"); }

}

 class Dog extends Animal{

  @Override

  void eatsomething() { System.out.println("eating foods"); }//should be eatSomething

}

 class TestAnnotation1{

     public static void main(String args[]) {

       Animal a=new Dog();

       a.eatSomething();

    }

}

**@SuppressWarnings** annotation: is used to suppress warnings issued by the compiler.

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

import java.util.\*;

class TestAnnotation2{

   @SuppressWarnings("unchecked")

   public static void main(String args[]){

       ArrayList list=new ArrayList();

       list.add("sonoo");

       list.add("vimal");

       list.add("ratan");

      for(Object obj:list)

          System.out.println(obj);

  }

}

**@Deprecated** annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

class A{

   void m(){ System.out.println("hello m"); }

   @Deprecated

   void n(){System.out.println("hello n");}

}

class TestAnnotation3{

   public static void main(String args[]){

       A a=new A();

       a.n();

   }

}

 An annotation that has no method, is called **marker annotation**. For example:

@interface MyAnnotation{}

The @Override and @Deprecated are marker annotations.

An annotation that has one method, is called **single-value annotation**. For example

@interface MyAnnotation{

int value();

}

We can provide the default value also. For example:

@interface MyAnnotation{

int value() default 0;

}

How to apply Single-Value Annotation

Let's see the code to apply the single value annotation.

@MyAnnotation(value=10)

The value can be anything.

 An annotation that has more than one method, is called **Multi-Value annotation**. For example:

@interface MyAnnotation {

   int value1();

   String value2();

   String value3();

}

 We can provide the default value also. For example:

@interface MyAnnotation {

   int value1() default 1;

   String value2() default "";

   String value3() default "xyz";

}

How to apply Multi-Value Annotation

Let's see the code to apply the multi-value annotation.

java annotations used in other anotations

**@Target** tag is used to specify at which type, the annotation is used.

The java.lang.annotation. ElementType enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc

@Target(ElementType.TYPE)

@interface MyAnnotation{

    int value1();

    String value2();

}

**@Retention**annotation is used to specify to what level annotation will be available.

RetentionPolicy.SOURCE------refers to the source code, discarded during compilation.

RetentionPolicy.CLASS----------refers to the .class file, available to java compiler but not to JVM . It is included in the class file.

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

@interface MyAnnotation{

int value1();

String value2();

}

**@Inherited**

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

@interface ForEveryone { }//Now it will be available to subclass also

@interface ForEveryone { }

class Superclass{}

class Subclass extends Superclass{}

The **@Documented** Marks the annotation for inclusion in the documentation.

**4.**The purpose of **upper bounded** wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.

Syntax

List<? extends Number>

? is a wildcard character.

extends, is a keyword.

Number, is a class present in java.lang package

Suppose, we want to write the method for the list of Number and its subtypes (like Integer, Double). Using List<? extends Number> is suitable for a list of type Number or any of its subclasses whereas List<Number> works with the list of type Number only. So, List<? extends Number> is less restrictive than List<Number>.

The **unbounded wildcard** type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -

When the given method is implemented by using the functionality provided in the Object class.

When the generic class contains the methods that don't depend on the type parameter.

Example of Unbounded Wildcards

import java.util.Arrays;

import java.util.List;

public class UnboundedWildcard {

  public static void display(List<?> list)  {

    for(Object o:list)  {

      System.out.println(o);

    }

  }

  public static void main(String[] args) {

  List<Integer> l1=Arrays.asList(1,2,3);

  System.out.println("displaying the Integer values");

  display(l1);

  List<String> l2=Arrays.asList("One","Two","Three");

   System.out.println("displaying the String values");

    display(l2);

  }

}

The purpose of **lower bounded** wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.

Syntax

List<? super Integer>

Here,

? is a wildcard character.

super, is a keyword.

Integer, is a wrapper class.

Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using List<? super Integer> is suitable for a list of type Integer or any of its superclasses whereas List<Integer> works with the list of type Integer only. So, List<? super Integer> is less restrictive than List<Integer>.

**5.Lambda expression** is a new and important feature of Java which was included in Java SE 8.

 It provides a clear and concise way to represent one method interface using an expression.

It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code.

In case of lambda expression, we don't need to define the method again for providing the implementation.

Java lambda expression is treated as a function, so compiler does not create .class file.

To provide the implementation of Functional interface.

Less coding.

Java Lambda Expression Syntax

     (argument-list) -> {body}

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

**6.** JDBC Driver is a software component that enables java application to interact with the database.

There are 4 types of JDBC drivers:

JDBC-ODBC bridge driver

Native-API driver (partially java driver)

Network Protocol driver (fully java driver)

Thin driver (fully java driver)

 The **JDBC-ODBC bridge driver** uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

easy to use.

can be easily connected to any database.

Disadvantages:

Performance degraded because JDBC method call is converted into the ODBC function calls.

The ODBC driver needs to be installed on the client machine.

 The **Native API driver** uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

Advantage:

performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

The Native driver needs to be installed on the each client machine.

The Vendor client library needs to be installed on client machine.

The **Network Protocol driver** uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

Advantage:

No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

Network support is required on client machine.

Requires database-specific coding to be done in the middle tier.

Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

The **thin driver** converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.  
Advantage:

Better performance than all other drivers.

No software is required at client side or server side.

Disadvantage:

Drivers depend on the Database.

**7.**A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Creating a generic class:

class MyGen<T>{

     T obj;

     void set(T obj){this.obj=obj;}

     T get(){return obj;}

}

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

T - Type

E - Element

K - Key

N - Number

V - Value

Like the generic class, we can create a **generic method** that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

Let's see a simple example of java generic method to print array elements. We are using here E to denote the element.

public class TestGenerics4{

  public static < E > void printArray(E[] elements) {

    for ( E element : elements){

      System.out.println(element );

     }

     System.out.println();

  }

  public static void main( String args[] ) {

    Integer[] intArray = { 10, 20, 30, 40, 50 };

    Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

    System.out.println( "Printing Integer Array" );

    printArray( intArray  );

    System.out.println( "Printing Character Array" );

    printArray( charArray );

  }

}

**8.Collection Framework**

provides an architecture to store and manipulate the group of objects.

achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector,LinkedList,PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Collection represents a single unit of objects, i.e., a group.

framework - provides readymade architecture.

            - represents a set of classes and interfaces.

The Collection framework represents a unified architecture for storing and manipulating a group of objects.

It has:

Interfaces and its implementations, i.e., classes

 Interfaces of the Collection Framework

List Interface

The List interface extends the Collection interface and represents an ordered collection (also known as a sequence). It allows duplicate elements.

Implementations: ArrayList, LinkedList, Vector, Stack.

Set Interface

The Set interface extends the Collection interface and represents a collection that does not allow duplicate elements.

Implementations: HashSet, LinkedHashSet, TreeSet.

Queue Interface

The Queue interface extends the Collection interface and represents a collection used to hold multiple elements prior to processing. It typically orders elements in a FIFO (first-in-first-out) manner.

Implementations: LinkedList, PriorityQueue.

3. Map Interface

The Map interface is part of the Collection Framework but does not extend the Collection interface. It represents a collection of key-value pairs, where each key maps to exactly one value.

Implementations: HashMap, LinkedHashMap, TreeMap, Hashtable.

Algorithm

The java.util package contains all the classes and interfaces for the Collection framework.

Algorithms

The Collection Framework also provides algorithms that operate on collections, such as:

Sorting: Collections.sort()

Searching: Collections.binarySearch()

Shuffling: Collections.shuffle()

Reverse: Collections.reverse()