Practice Set 3

Q.Explain the usage of interfaces / abstract classes.
What is Interface?

- The interface is a blueprint that can be used to implement a class. The interface does not contain any concrete methods (methods that have code). All the methods of an interface are abstract methods.
- An interface cannot be instantiated. However, classes that implement interfaces can be instantiated. Interfaces never contain instance variables but, they can contain public static final variables (i.e., constant class variables)

What Is Abstract Class?

- A class which has the abstract keyword in its declaration is called abstract class. Abstract classes should have at least one abstract method. , i.e., methods without a body. It can have multiple concrete methods.
- Abstract classes allow you to create blueprints for concrete classes. But the inheriting class should implement the abstract method.
- Abstract classes cannot be instantiated.

Important Reasons For Using Interfaces

- Interfaces are used to achieve abstraction.
- Designed to support dynamic method resolution at run time
- It helps you to achieve loose coupling.
- Allows you to separate the definition of a method from the inheritance hierarchy

Important Reasons For Using Abstract Class

- Abstract classes offer default functionality for the subclasses.
- Provides a template for future specific classes
- Helps you to define a common interface for its subclasses
- Abstract class allows code reusability.
- **Interface Vs. Abstract Class**
- An abstract class permits you to make functionality that subclasses can implement or override whereas an interface only permits you to state functionality but not to implement it. A class can extend only one abstract class while a class can implement multiple interfaces.

Parameters	Interface	Abstract class
Speed	Slow	Fast
Multiple Inheritances	Implement several Interfaces	Only one abstract class
Structure	Abstract methods	Abstract & concrete methods
When to use	Future enhancement	To avoid independence
Inheritance/ Implementation	A Class can implement multiple interfaces	The class can inherit only one Abstract Class
Default Implementation	While adding new stuff to the interface, it is a nightmare to find all the implementors and	In case of Abstract Class, you can take advantage of the default

Parameters	Interface	Abstract class
	implement newly defined stuff.	implementation.
Access Modifiers	The interface does not have access modifiers. Everything defined inside the interface is assumed public modifier.	Abstract Class can have an access modifier.
When to use	It is better to use interface when various implementations share only method signature. Polymorphic hierarchy of value types.	It should be used when various implementations of the same kind share a common behavior.
Data fields	the interface cannot contain data fields.	the class can have data fields.
Multiple Inheritance Default	A class may implement numerous interfaces.	A class inherits only one abstract class.
Implementation	An interface is abstract so that it can't provide any code.	An abstract class can give complete, default code which should be overridden.
Use of Access modifiers	You cannot use access modifiers for the method, properties, etc.	You can use an abstract class which contains access modifiers.
Usage	Interfaces help to define the peripheral abilities of a class.	An abstract class defines the identity of a class.
Defined fields	No fields can be defined	An abstract class allows you to define both fields and constants
Inheritance	An interface can inherit multiple interfaces but cannot inherit a class.	An abstract class can inherit a class and multiple interfaces.
Constructor or destructors	An interface cannot declare constructors or destructors.	An abstract class can declare constructors and destructors.
Limit of Extensions	It can extend any number of interfaces.	It can extend only one class or one abstract class at a time.
Abstract keyword	In an abstract interface keyword, is optional for declaring a method as an abstract.	In an abstract class, the abstract keyword is compulsory for declaring a method as an abstract.
Class type	An interface can have only public abstract methods.	An abstract class has protected and public abstract methods.

Q. Which features were added to interfaces in Java 8 and Java 9?
Java 8 provides following features for Java Programming:

- Lambda expressions,
- Method references,
- Functional interfaces,
- Stream API,
- Default methods,
- Base64 Encode Decode,
- Static methods in interface,
- Optional class,
- Collectors class,

- ForEach() method,
- Parallel array sorting,
- Nashorn JavaScript Engine,

Java 9 provides following features for Java Programming:

- Platform Module System (Project Jigsaw)
- Interface Private Methods
- Try-With Resources
- Anonymous Classes
- @SafeVarargs Annotation
- Collection Factory Methods
- Process API Improvement
- New Version-String Scheme
- JShell: The Java Shell (REPL)
- Process API Improvement
- Control Panel
- Stream API Improvement
- Installer Enhancement for Microsoft windows and many more

Q. What is an immutable class? Explain the steps involved in creating an immutable class.

To create an immutable class in java, you have to do following steps.

- Declare the class as final so it can't be extended.
- Make all fields private so that direct access is not allowed.
- Don't provide setter methods for variables
- Make all **mutable fields final** so that it's value can be assigned only once.
- Initialize all the fields via a constructor performing deep copy.
- Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

Q. Is it necessary that all properties of immutable class be final?
No, it is not mandatory to have all properties final to create an immutable object. In immutable objects you should not allow users to modify the variables of the class.

- No. If those properties/attributes are private and there's no getter for the same, i.e. if those aren't exposed to outer world, they need not be final.

- The idea of immutability is to maintain the state of the object, that is created during the object creation.
  If you are satisfying this condition, you're well and good.

- Also, just by having final on all attributes, you won't achieve immutability. You can put final on mutable attributes only, what if you have an attribute i.e. mutable?

- For example, you have an attribute of Date type, that you are exposing using a getter. Any one can call date.someDateMethod(), even if it was final. So, in these cases you have to clone the value of Date attribute, and expose the cloned Date attribute, so no one cam change the state of your immutable class.

Q.Explain the marker interfaces in Java with example. Explain Serializable and Cloneable.
Marker interface in Java

It is an empty interface (no field or methods). Examples of marker interface are Serializable, Clonnable and Remote interface. All these interfaces are empty interfaces.

```
public interface Serializable
{
 // nothing here
}
```

Examples of Marker Interface which are used in real-time applications :

- **Cloneable interface** : Cloneable interface is present in java.lang package. There is a method clone() in Object class. A class that implements the Cloneable interface indicates that it is legal for clone() method to make a field-for-field copy of instances of that class.
  Invoking Object's clone method on an instance of the class that does not implement the Cloneable interface results in an exception CloneNotSupportedException being thrown. By convention, classes that implement this interface should override Object.clone() method.

- **Serializable interface** : Serializable interface is present in java.io package. It is used to make an object eligible for saving its state into a file. This is called Serialization.
  Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable.
- **Remote interface** : Remote interface is present in java.rmi package. A remote object is an object which is stored at one machine and accessed from another machine. So, to make an object a remote object, we need to flag it with Remote interface. Here, Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine.Any object that is a remote object must directly or indirectly implement this interface. RMI (Remote Method Invocation) provides some convenience classes that remote object implementations can extend which facilitate remote object creation

Q.Is Externalizable a marker interface? Difference between Serializable and Externalizable

Externalization serves the purpose of custom Serialization, where we can decide what to store in stream.

Externalizable interface present in java.io, is used for Externalization which extends Serializable interface. It consist of two methods which we have to override to write/read object into/from stream which are-

```
// to read object from stream
void readExternal(ObjectInput in)


// to write object into stream
void writeExternal(ObjectOutput out)
```

Key differences between Serializable and Externalizable

- **Implementation :** Unlike Serializable interface which will serialize the variables in object with just by implementing interface, here we have to explicitly mention what fields or variables you want to serialize.

- **Methods :** Serializable is marker interface without any methods. Externalizable interface contains two methods: writeExternal() and readExternal().
- **Process:** Default Serialization process will take place for classes implementing Serializable interface. Programmer defined Serialization process for classes implementing Externalizable interface.
- **Backward Compatibility and Control:** If you have to support multiple versions, you can have full control with Externalizable interface. You can support different versions of your object. If you implement Externalizable, it's your responsibility to serialize super class.
- **public No-arg constructor:** Serializable uses reflection to construct object and does not require no arg constructor. But Externalizable requires public no-arg constructor.
  - When an Externalizable object is reconstructed, an instance is created first using the public no-argument constructor, then the readExternal method is called. So, it is mandatory to provide a no-argument constructor.
  - When an object implements Serializable interface, is serialized or deserialized, no constructor of object is called and hence any initialization which is implemented in constructor can't be done.

Q.What is the difference between checked and unchecked exception

In Java, there are two types of exceptions:

- **Checked:** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.
- For example, consider the following Java program that opens file at location "C:\test\a.txt" and prints the first three lines of it. The program doesn't compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*
- To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block. We have used throws in the below program. Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.
- **2) Unchecked** are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.
  In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

Checked vs. Unchecked Exceptions

Checked Exceptions	Unchecked Exceptions
Not subclass of RuntimeException	Subclass of RuntimeException
if not caught, method *must* specify it to be thrown	if not caught, method *may* specify it to be thrown
for errors that the programmer *cannot* directly prevent from occurring	For errors that the programmer *can* directly prevent from occurring
`IOException`, `FileNotFoundException`, `SocketException`, etc.	`NullPointerException`, `IllegalArgumentException`, `IllegalStateException`, etc.

Q.What is the difference between Exception and Error ?

Error : An Error "indicates serious problems that a reasonable application should not try to catch."
Both Errors and Exceptions are the subclasses of java.lang.Throwable class. Errors are the conditions which cannot get recovered by any handling techniques. It surely cause termination of the program abnormally. Errors belong to unchecked type and mostly occur at runtime. Some of the examples of errors are Out of memory error or a System crash error.

Exceptions : An Exception "indicates conditions that a reasonable application might want to catch."
Exceptions are the conditions that occur at runtime and may cause the termination of program. But they are recoverable using try, catch and throw keywords. Exceptions are divided into two catagories : checked and unchecked exceptions. Checked exceptions like IOException known to the compiler at compile time while unchecked exceptions like ArrayIndexOutOfBoundException known to the compiler at runtime. It is mostly caused by the program written by the programmer.

Summary of Differences

ERRORS	EXCEPTIONS
Recovering from Error is not possible.	We can recover from exceptions by either using try-catch block or throwing exceptions back to caller.
All errors in java are unchecked type.	Exceptions include both checked as well as unchecked type.
Errors are mostly caused by the	Program itself is responsible for causing

environment in which program is running.	exceptions.
Errors occur at runtime and not known to the compiler.	All exceptions occurs at runtime but checked exceptions are known to compiler while unchecked are not.
They are defined in java.lang.Error package.	They are defined in java.lang.Exception package
Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError	Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, NullPointerException, ArithmeticException.

Q.What is the difference between ClassNotFoundException and NoClassDefFoundError?

ClassNotFoundException and **NoClassDefFoundError** both occur when class is not found at runtime. They are related to Java classpath.

ClassNotFoundException

ClassNotFoundException occurs when you try to load a class at runtime using **Class.forName()** or **loadClass()** methods and requested classes are not found in classpath. Most of the time this exception will occur when you try to run application without updating classpath with JAR files. This exception is a **checked Exception** derived from **java.lang.Exception** class and you need to provide **explicit handling** for it. This exception also occurs when you have two class loaders and if a ClassLoader tries to access a class which is loaded by another classloader in Java. You must be wondering that what actually is classloader in Java. **Java ClassLoader** is a part of Java Runtime Environment that dynamically loads Java classes in JVM(Java Virtual Machine). The Java Runtime System does not need to know about files and files system because of classloaders.

ClassNotFoundException is raised in below program as class “GeeksForGeeks” is not found in classpath.

NoClassDefFoundError

NoClassDefFoundError occurs when class was present during compile time and program was compiled and linked successfully but class was **not** present during runtime. It is error which is derived from **LinkageError**. Linkage error occurs when a class has some dependencies on another class and latter class changes after compilation of former class. NoClassFoundError is the result of **implicit loading** of class because of calling a

method or accessing a variable from that class. This error is more difficult to debug and find the reason why this error occurred. So in this case you should always check the classes which are dependent on this class.

ClassNotFoundException Vs NoClassDefFoundError

- As the name suggests, ClassNotFoundException is an exception while NoClassDefFoundError is an error.
- ClassNotFoundException occurs when classpath is does not get updated with required JAR files while error occurs when required class definition is not present at runtime.

Q.Explain how class loader works in Java. Can one class be loaded by two class loaders in Java.

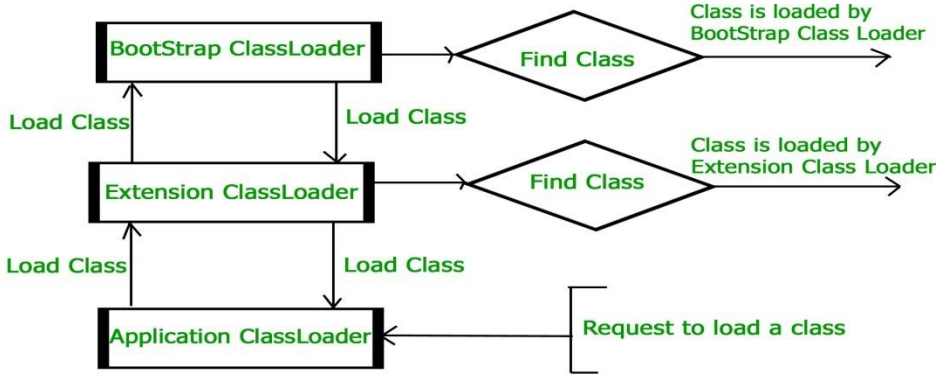
The **Java ClassLoader** is a part of the **Java Runtime Environment** that dynamically loads Java classes into the **Java Virtual Machine**. The Java run time system does not need to know about files and file systems because of classloaders.
Java classes aren't loaded into memory all at once, but when required by an application. At this point, the **Java ClassLoader** is called by the **JRE** and these ClassLoaders load classes into memory dynamically.

Types of ClassLoaders in Java

Not all classes are loaded by a single ClassLoader. Depending on the type of class and the path of class, the ClassLoader that loads that particular class is decided. To know the ClassLoader that loads a class the ***getClassLoader()*** method is used. All classes are loaded based on their names and if any of these classes are not found then it returns a **NoClassDefFoundError** or **ClassNotFoundException**.
A Java Classloader is of **three types**:

1. **BootStrap ClassLoader:** A Bootstrap Classloader is a Machine code which kickstarts the operation when the JVM calls it. It is not a java class. Its job is to load the first pure Java ClassLoader. Bootstrap ClassLoader loads classes from the location ***rt.jar***. Bootstrap ClassLoader doesn't have any parent ClassLoaders. It is also called as the **Primodial ClassLoader**.
2. **Extension ClassLoader:** The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from ***jre/lib/ext*** directory or any other directory pointed by the system property ***java.ext.dirs***.
3. **System ClassLoader:** An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable ***CLASSPATH, -classpath or -cp command line option***. The Application ClassLoader is a child class of Extension ClassLoader.

Note: The ClassLoader Delegation Hierarchy Model always functions in the order Application ClassLoader->Extension ClassLoader->Bootstrap ClassLoader. The Bootstrap ClassLoader is always given the higher priority, next is Extensi
on ClassLoader and then Application ClassLoader.

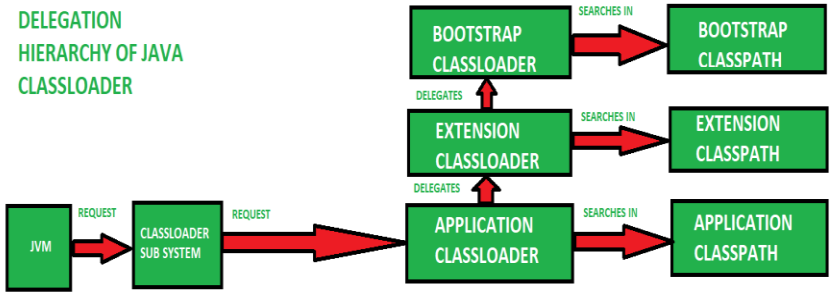


Principles of functionality of a Java ClassLoader

Principles of functionality are the **set of rules** or features on which a Java ClassLoader works. There are **three principles of functionality**, they are:

1. **Delegation Model**: The Java Virtual Machine and the Java ClassLoader use an algorithm called the **Delegation Hierarchy Algorithm** to Load the classes into the Java file.
   The ClassLoader works based on a set of operations given by the delegation model. They are:

   - ClassLoader always follows the **Delegation Hierarchy Principle**.
   - Whenever JVM comes across a class, it checks whether that class is already loaded or not.
   - If the Class is already loaded in the method area then the JVM proceeds with execution.
   - If the class is not present in the method area then the JVM asks the Java ClassLoader Sub-System to load that particular class, then ClassLoader sub-system hands over the control to **Application ClassLoader**.
   - Application ClassLoader then delegates the request to Extension ClassLoader and the **Extension ClassLoader** in turn delegates the request to **Bootstrap ClassLoader**.
   - Bootstrap ClassLoader will search in the Bootstrap classpath(JDK/JRE/LIB). If the class is available then it is loaded, if not the request is delegated to Extension ClassLoader.
   - Extension ClassLoader searches for the class in the Extension Classpath(JDK/JRE/LIB/EXT). If the class is available then it is loaded, if not the request is delegated to the Application ClassLoader.
   - Application ClassLoader searches for the class in the Application Classpath. If the class is available then it is loaded, if not then a **ClassNotFoundException** exception is generated.



1. **Visibility Principle**: The **Visibility Principle** states that a class loaded by a parent ClassLoader is visible to the child ClassLoaders but a class loaded by a child ClassLoader is not visible to the parent ClassLoaders. Suppose a class GEEKS.class has been loaded by the Extension ClassLoader, then that class is only visible to the Extension ClassLoader and Application ClassLoader but not to the Bootstrap ClassLoader. If that class is again tried to load using Bootstrap ClassLoader it gives an exception ***java.lang.ClassNotFoundException***.
2. **Uniqueness Property**: The **Uniquesness Property** ensures that the classes are unique and there is no repetition of classes. This also ensures that the classes loaded by parent classloaders are not loaded by the child classloaders. If the parent class loader isn't able to find the class, only then the current instance would attempt to do so itself.

Methods of Java.lang.ClassLoader

After the JVM requests for the class, a few steps are to be followed in order to load a class. The Classes are loaded as per the delegation model but there are a few important Methods or Functions that play a vital role in loading a Class.

1. **loadClass(String name, boolean resolve)**: This method is used to load the classes which are referenced by the JVM. It takes the name of the class as a parameter. This is of type loadClass(String, boolean).
2. **defineClass()**: The defineClass() method is a *final* method and cannot be overriden. This method is used to define a array of bytes as an instance of class. If the class is invalid then it throws **ClassFormatError**.
3. **findClass(String name)**: This method is used to find a specified class. This method only finds but doesn't load the class.
4. **findLoadedClass(String name)**: This method is used to verify whether the Class referenced by the JVM was previously loaded or not.
5. **Class.forName(String name, boolean initialize, ClassLoader loader)**: This method is used to load the class as well as initialize the class. This method also gives the option to choose any one of the ClassLoaders. If the ClassLoader parameter is NULL then Bootstrap ClassLoader is used.

   Note: If a class has already been loaded, it returns it. Otherwise, it delegates the search for the new class to the parent class loader. If the parent class loader doesn't find the class, **loadClass()** calls the method **findClass()** to find and load the class. The **finalClass()** method searches for the class in the current **ClassLoader** if the class wasn't found by the parent **ClassLoader**.

Q.Explain the difference between Comparable and Comparator

Using Comparable Interface

A comparable object is capable of comparing itself with another object. The class itself must implements the **java.lang.Comparable** interface to compare its instances.

Using Comparator

Unlike Comparable, Comparator is external to the element type we are comparing. It's a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.

Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.

To compare movies by Rating, we need to do 3 things :

1. Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).
2. Make an instance of the Comparator class.
3. Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator
4. Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.
5. Logically, Comparable interface compares "this" reference with the object specified and Comparator in Java compares two different class objects provided.
6. If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on there natural order defined by CompareTo method.

To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.

.

Q.Explain the concept of constructor chaining.

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

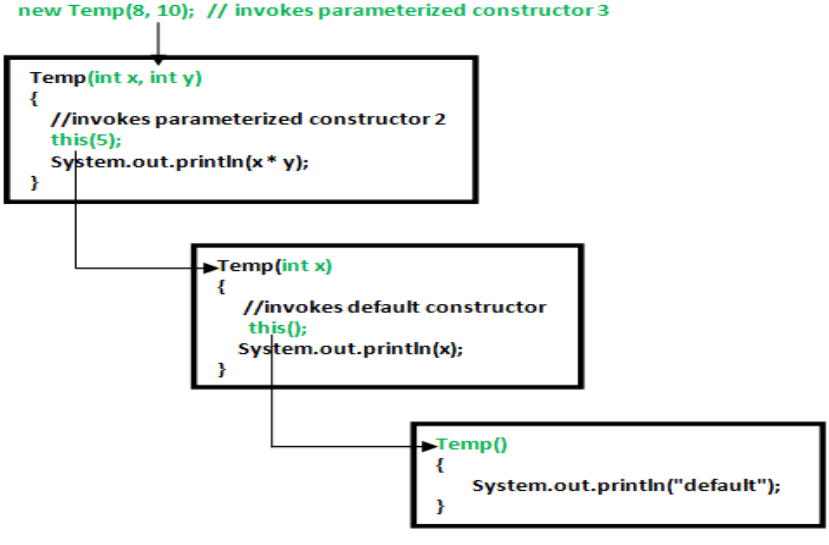
Constructor chaining can be done in two ways:

- **Within same class**: It can be done using **this()** keyword for constructors in same class
- **From base class:** by using **super()** keyword to call constructor from the base class.

Constructor chaining occurs through **inheritance**. A sub class constructor's task is to call super class's constructor first. This ensures that creation of sub class's object starts with the initialization of the data members of the super class. There could be any numbers of classes in inheritance chain. Every constructor calls up the chain till class at the top is reached.

Why do we need constructor chaining ?

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.



Rules of constructor chaining :

1. The **this()** expression should always be the first line of the constructor.
2. There should be at-least be one constructor without the this() keyword (constructor 3 in above example).
3. Constructor chaining can be achieved in any order.

Q.What is serialVersionUID?

The serialization at runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

Why so we use SerialVersionUID : SerialVersionUID is used to ensure that during deserialization the same class (that was used during serialize process) is loaded.
Example: Suppose a person who is in UK and another person who is in India, both are going to perform serialization and deserialization respectively. In this case to authenticate that the receiver who is in India is the authenticated person, JVM creates an Unique ID which is known as **SerialVersionUID**.
In most of the cases, serialization and deserialization both activities are done by a single person with the same system and same location. But in serialization, sender and receiver are not the same person i.e. the persons may be different, machine or system may be different and location must be different then SerialVersionUID comes in the picture. In serialization, both sender and receiver should have .class file at the time of beginning only i.e. the person who is going to do serialization and the person who is ready for deserialization should contain same .class file at the beginning time only.

Serialization : At the time of serialization, with every object sender side JVM will save a **Unique Identifier**. JVM is responsible to generate that unique ID based on the corresponding .class file which is present in the sender system.
Deserialization: At the time of deserialization, receiver side JVM will compare the unique ID associated with the Object with local class Unique ID i.e. JVM will also create a Unique ID based on the corresponding .class file which is present in the receiver system. If both unique ID matched then only deserialization will be performed. Otherwise we will get Runtime Exception saying **InvalidClassException**. This unique Identifier is nothing but **SerialVersionUID**.
Problem of depending on default SerialVersionUID generated by JVM :

- Both sender and receiver should use the same JVM with respect to platform and version also. Otherwise receiver unable to deserialize because of different SerialVersionUID.
- Both sender and receiver should use same .class file version. After serialization if there is any change in .class file at receiver side then receiver unable to deserialize.
- To generate SerialVersionUID internally JVM may use complex algorithm which may create performance problem.

We can solve the above problem by configuring our own SerialVersionUID. We can configure our own SerialVersionUID as follows:

```
private static final long SerialVersionUID=10l;
```

Q.What is the difference between Iterator and ListIterator?
Iterator
Iterators are used in Collection framework in Java to retrieve elements one by one. It can be applied to any Collection object. By using Iterator, we can perform both read and remove operations. Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the only cursor available for entire collection framework.
Iterator object can be created by calling iterator() method present in Collection interface.

```
// Here "c" is any Collection object. itr is of

// type Iterator interface and refers to "c"

Iterator itr = c.iterator();
```

ListIterator
It is only applicable for List collection implemented classes like arraylist, linkedlist etc. It provides bi-directional iteration. ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

ListIterator object can be created by calling listIterator() method present in List interface.

```
// Here "l" is any List object, ltr is of type
// ListIterator interface and refers to "l"
ListIterator ltr = l.listIterator();
// Here "c" is any Collection object. itr is of
// type Iterator interface and refers to "c"
Iterator itr = c.iterator();
```

ListIterator

It is only applicable for List collection implemented classes like arraylist, linkedlist etc. It provides bi-directional iteration. ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.
ListIterator object can be created by calling listIterator() method present in List interface.

```
// Here "l" is any List object, ltr is of type
// ListIterator interface and refers to "l"
ListIterator ltr = l.listIterator();
```

Differences between Iterator and ListIterator:

1. Iterator can traverse only in forward direction whereas ListIterator traverses both in forward and backward directions.

2.ListIterator can help to replace an element whereas Iterator cannot

Table showing Difference between Iterator and ListIterator

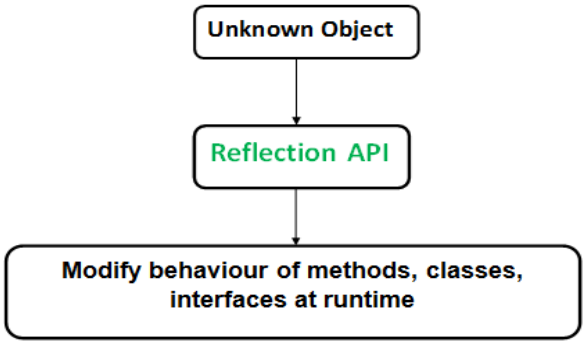
ITERATOR	LISTITERATOR
Can traverse elements present in Collection only in the forward direction.	Can traverse elements present in Collection both in forward and backward directions.
Helps to traverse Map, List and Set.	Can only traverse List and not the other two.
Indexes cannot be obtained by using Iterator.	It has methods like nextIndex() and previousIndex() to obtain indexes of elements at any time while traversing List.
Cannot modify or replace elements present in Collection	We can modify or replace elements with the help of set(E e)

ITERATOR	LISTITERATOR
Cannot add elements and it throws ConcurrentModificationException.	Can easily add elements to a collection at any time.
Certain methods of Iterator are next(), remove() and hasNext().	Certain methods of ListIterator are next(), previous(), hasNext(), hasPrevious(), add(E e).

Q.What is reflection? Where is it used?

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

- The required classes for reflection are provided under java.lang.reflect package.
- Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
- Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.



Reflection can be used to get information about –

1. **Class** The getClass() method is used to get the name of the class to which an object belongs.
2. **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
3. **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

Important observations :

1. We can invoke an method through reflection if we know its name and parameter types. We use below two methods for this purpose

getDeclaredMethod() : To create an object of method to be invoked. The syntax for this method is

2. Class.getDeclaredMethod(name, parametertype)
3. name- the name of method whose object is to be created

parametertype - parameter is an array of Class objects

invoke() : To invoke a method of the class at runtime we use following method–

Method.invoke(Object, parameter)

If the method of the class doesn't accepts any

parameter then null is passed as argument.

4. Through reflection we can **access the private variables and methods** of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.
Class.getDeclaredField(FieldName) : Used to get the private field. Returns an object of type Field for specified field name.
Field.setAccessible(true) : Allows to access the field irrespective of the access modifier used with the field.

Advantages of Using Reflection:

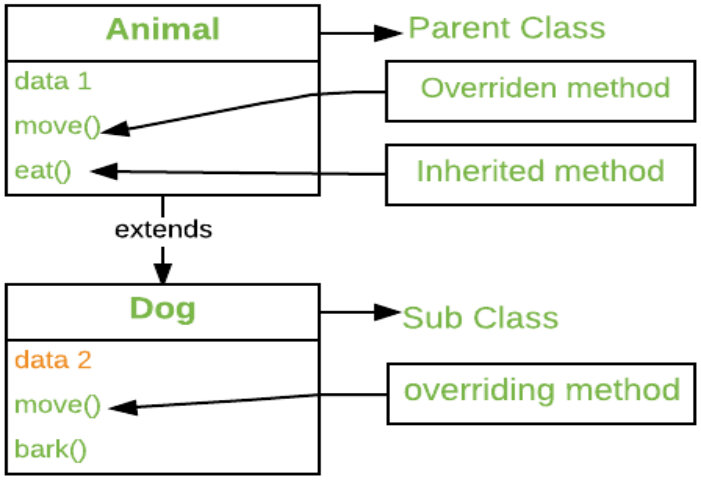
- **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Debugging and testing tools**: Debuggers use the property of reflection to examine private members on classes.

Drawbacks:

- **Performance Overhead:** Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

Q.What are the rules for overriding methods in Java?

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.



Method overriding is one of the way by which java achieve Run Time Polymorphism.The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

1.Overriding and Access-Modifiers : The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

2.Final methods can not be overridden : If we don't want a method to be overridden, we declare it as final. Please see Using final with Inheritance .

3.Static methods can not be overridden(Method Overriding vs Method Hiding) : When you defines a static method with same signature as a static method in base class, it is known as method hiding.
The following table summarizes what happens when you define a method with the same signature as a method in a super-class.

	SUPERCLASS INSTANCE METHOD	SUPERCLASS STATIC METHOD
SUBCLASS INSTANCE METHOD	Overrides	Generates a compile-time error
SUBCLASS STATIC METHOD	Generates a compile-time error	Hides

4.Private methods can not be overridden : Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in a subclass.

5.The overriding method must have same return type (or subtype) : From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomena is known as **covariant return type**.

6.Invoking overridden method from sub-class : We can call parent class method in overriding method using super keyword.

7.Overriding and constructor : We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).

8.Overriding and Exception-Handling : Below are two rules to note when overriding methods related to exception-handling.

- **Rule#1 :** If the super-class overridden method does not throws an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.

Rule#2 : If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error.Also there is no issue if subclass overridden method is not throwing any exception.

9.Overriding and abstract method: Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.

10.Overriding and synchronized/strictfp method : The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding, i.e. it's possible that a synchronized/strictfp

Note :

1. In C++, we need virtual keyword to achieve overriding or Run Time Polymorphism. In Java, methods are virtual by default.
2. We can have multilevel method-overriding.

- **Overriding vs Overloading :**

- Overloading is about same method have different signatures. Overriding is about same method, same signature but different classes connected through inheritance
  Overloading is an example of compiler-time polymorphism and overriding is an example of run time polymorphism.

Why Method Overriding ?

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.
Dynamic Method Dispatch is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability to exist code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.
Overridden methods allow us to call methods of any of the derived classes without even knowing the type of derived class object.

When to apply Method Overriding ?(with example)

Overriding and Inheritance : Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its methods, yet still enforces a consistent interface. **Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.**

Q.How to implement a Singleton in Java?

Singleton Class in Java

In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.
After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.
To design a singleton class:

1. Make constructor as private.
2. Write a static method that has return type object of this singleton class. Here, the concept of Lazy initialization is used to write this static method.

Normal class vs Singleton class: Difference in normal and singleton class in terms of instantiation is that, For normal class we use constructor, whereas for singleton class we use getInstance() method (Example code:I). In general, to avoid confusion we may also use the class name as method name while defining this method (Example code:II).

Implementing Singleton class with getInstance() method

```
// Java program implementing Singleton class
// with getInstance() method
class Singleton
{
	// static variable single_instance of type Singleton
	private static Singleton single_instance = null;

	// variable of type String
	public String s;

	// private constructor restricted to this class itself
	private Singleton()
	{
		s = "Hello I am a string part of Singleton class";
	}

	// static method to create instance of Singleton class
	public static Singleton getInstance()
	{
		if (single_instance == null)
			single_instance = new Singleton();

		return single_instance;
	}
}

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

```
        // instantiating Singleton class with variable x
        Singleton x = Singleton.getInstance();

        // instantiating Singleton class with variable y
        Singleton y = Singleton.getInstance();

        // instantiating Singleton class with variable z
        Singleton z = Singleton.getInstance();

        // changing variable of instance x
        x.s = (x.s).toUpperCase();

        System.out.println("String from x is " + x.s);
        System.out.println("String from y is " + y.s);
        System.out.println("String from z is " + z.s);
        System.out.println("\n");

        // changing variable of instance z
        z.s = (z.s).toLowerCase();

        System.out.println("String from x is " + x.s);
        System.out.println("String from y is " + y.s);
        System.out.println("String from z is " + z.s);
    }
}
```

Explanation: In the Singleton class, when we first time call getInstance() method, it creates an object of the class with name single_instance and return it to the variable. Since single_instance is static, it is changed from null to some object. Next time, if we try to call getInstance() method, since single_instance is not null, it is returned to the variable, instead of instantiating the Singleton class again. This part is done by if condition.
In the main class, we instantiate the singleton class with 3 objects x, y, z by calling static method getInstance(). But actually after creation of object x, variables y and z are pointed to object x as shown in the diagram. Hence, if we change the variables of object x, that is reflected when we access the variables of objects y and z. Also if we change the variables of object z, that is reflected when we access the variables of objects x and y.