**Explain the usage of interfaces / abstract classes.**

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 Interfaces

* 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.

**Which features were added to interfaces in Java 8 and Java 9? Why? (like default methods, private static methods)**

**Java 8:- public abstract** methods, you can have **public static** methods and **public default** methods.

**Java 9:-** we will be able to add **private methods** and **private static method** in interfaces.

These private methods will improve code re-usability inside interfaces. Foe example, if two default methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to it’s implementing classes.

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

An immutable class is good for caching purpose because you don’t need to worry about the value changes. Other benefit of immutable class is that it is inherently [thread-safe](https://www.journaldev.com/1061/thread-safety-in-java), so you don’t need to worry about thread safety in case of multi-threaded environment.

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

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

**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**. You can do this just by making variables private and not providing setter methods to modify them.

**Why is String class in Java immutable?**

The **string** is **Immutable** in **Java** because **String** objects are cached in **String** pool. ... Another reason of why **String class** is **immutable** could die due to HashMap. Since **Strings** are very popular as HashMap key, it's important for them to be **immutable** so that they can retrieve the value object which was stored in HashMap.

**Explain the marker interfaces in Java with example. Explain Serializable and Cloneable.**

Marker interface in Java is interfaces with no field or methods or in simple word empty interface in java is called marker interface. Example of market interface is Serializable, Cloneable and Remote interface.

**Cloneable interface** : Cloneable interface is present in java.lang package. There is a method clone() in [Object](https://www.geeksforgeeks.org/object-class-in-java/) 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.

**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](http://quiz.geeksforgeeks.org/serialization-in-java/).  
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.

**What is the difference between checked and unchecked exception? Give examples for each from the JDK.**

There are two types of exceptions: checked exception and unchecked exception. In this guide, we will discuss them. The main difference between checked and unchecked exception is that the checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime.

class Example {

public static void main(String args[])

{

FileInputStream fis = null;

fis = new FileInputStream("B:/myfile.txt");

int k;

while(( k = fis.read() ) != -1)

{

System.out.print((char)k); }

fis.close();

}

}

Why this compilation error? As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn’t handled/declared the exceptions, our program gave the compilation error.

class Example {

public static void main(String args[])

{

int num1=10;

int num2=0;

int res=num1/num2;

System.out.println(res);

}

}

It doesn’t mean that compiler is not checking these exceptions so we shouldn’t handle them. In fact we should handle them more carefully. For e.g. In the above example there should be a exception message to user that they are trying to display a value which doesn’t exist in array so that user would be able to correct the issue.

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

|  |  |  |
| --- | --- | --- |
| **Parameter** | Serializable | **Externalizable** |
| Marker interface | It is marker interface. You don’t have to provide implementation of any method. | Externalizable is not marker interface, you have to override writeExternal and readExternal method. |
| Control | Serializable interface has less control over serialization process and it is optional to override readObject and writeObject. | Externalizable interface has more control over serialization process and it is mandatory to override writeExternal and readExternal. |
| Performance | JVM uses reflection to perform serialization in the case of Serializable interface which is quite slow. | Programmer have to implement readExternal and writeExternal methods but it relatively results in better performance |
| Supersedes | NA | If you implement Externalizable interface and provide implementation of readExternal and writeExternal then it supersedes readObject and writeObject methods in that class. It is due to the fact that Externalizable extends Serializable interface. |

**What is the difference between Exception and Error? Give examples of Errors.**

|  |  |
| --- | --- |
| **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 environment in which program is running. | Program itself is responsible for causing 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. |

**What is the difference between ClassNotFoundException and NoClassDefFoundError?**

|  |  |
| --- | --- |
| **ClassNotFoundException** | **NoClassDefFoundError** |
| It is an exception. It is of type java.lang.Exception. | It is an error. It is of type java.lang.Error. |
| It occurs when an application tries to load a class at run time which is not updated in the classpath. | It occurs when java runtime system doesn’t find a class definition, which is present at compile time, but missing at run time. |
| It is thrown by the application itself. It is thrown by the methods like Class.forName(), loadClass() and findSystemClass(). | It is thrown by the Java Runtime System. |
| It occurs when classpath is not updated with required JAR files. | It occurs when required class definition is missing at runtime. |

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

ClassLoader in Java is a class which is used to load [class files in Java](http://javarevisited.blogspot.ca/2012/05/10-points-about-class-file-in-java.html). Java code is compiled into class file by [javac](http://javarevisited.blogspot.sg/2012/12/javac-is-not-recognized-as-internal-or-external-command.html)compiler and [JVM](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html)executes Java program, by executing byte codes written in class file. ClassLoader is responsible for loading class files from file system, network or any other source. There are three default class loader used in Java, Bootstrap , Extension and System or Application class loader. 

**Explain the difference between Comparable and Comparator.**

**Comparator Interface :**A comparison function, which is used to impose ordering on some collection of objects. To allow precisely control over the sort order , Comparators can be passed to a sort method (e.g Collections.sort()). Certain type of data structures such as TreeSet or TreeMap can also be sorted using Comparator.

**Comparable Interface :** Comparable is an public interfaces which is used to impose an natural ordering (if numbers then 1,2,3 or in alphabetical order   'a','b','c' ) of the class that implements it.

|  |  |
| --- | --- |
| Comparable | Comparator |
| Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.  Comparable **affects the original class**, i.e., the actual class is modified.  Comparable provides **compareTo() method** to sort elements.  Comparable is present in **java.lang** package.  We can sort the list elements of Comparable type by **Collections.sort(List)** method. | The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.  Comparator **doesn't affect the original class**, i.e., the actual class is not modified.  Comparator provides **compare() method** to sort elements.  A Comparator is present in the **java.util** package.  We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method. |

**Compare String, StringBuffer and StringBuilder.**

1. String is immutable whereas StringBuffer and StringBuider are mutable classes.
2. StringBuffer is thread safe and synchronized whereas StringBuilder is not, thats why [StringBuilder is more faster than StringBuffer](https://www.journaldev.com/137/stringbuffer-vs-stringbuilder).
3. String concat + operator internally uses StringBuffer or StringBuilder class.
4. For String manipulations in non-multi threaded environment, we should use StringBuilder else use StringBuffer class.

**Explain the concept of constructor chaining.**

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

**What is serialVersionUID?**

The **serialVersionUID** is a unique identifier for Serializable classes. This is used during the deserialization of an object, to ensure that a loaded class is compatible with the serialized object. If no matching class is found, an InvalidClassException is thrown.

**What is the difference between Iterator and ListIterator.**

|  |  |
| --- | --- |
| **Iterator** | **ListIterator** |
| Iterator is used for traversing List and Set both.  We can traverse in only forward direction using Iterator.   We cannot obtain indexes while using Iterator  We cannot add element to collection while traversing it using Iterator, it throws ConcurrentModificationException when you try to do it.  We cannot replace the existing element value when using Iterator.  It has 3 methods | ListIterator to traverse List only, we cannot traverse Set using ListIterator.  we can traverse a List in both the directions (forward and Backward).  We can obtain indexes at any point of time while traversing a list using ListIterator. The methods nextIndex() and previousIndex() are used for this purpose.  We can add element at any point of time while traversing a list using ListIterator.  By using set(E e) method of ListIterator we can replace the last element returned by next() or previous() methods.  It has 9 Methods. |

**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.

**What is the difference between fail-fast and fail-safe iterator? Explain ConcurrentModificationException with an example.**

|  |  |
| --- | --- |
| **Fail-Fast Iterators** | **Fail-Safe Iterators** |
| Fail-Fast iterators doesn’t allow modifications of a collection while iterating over it.  These iterators throw ConcurrentModificationException if a collection is modified while iterating over it.  They use original collection to traverse over the elements of the collection.  These iterators don’t require extra memory.  Ex : Iterators returned by ArrayList, Vector, HashMap. | Fail-Safe iterators allow modifications of a collection while iterating over it.  These iterators don’t throw any exceptions if a collection is modified while iterating over it.  They use copy of the original collection to traverse over the elements of the collection.  These iterators require extra memory to clone the collection.  Ex : Iterator returned by ConcurrentHashMap. |

ConcurrentHashMap in Java is introduced as an alternative of Hashtable in Java 1.5 as part of Java concurrency package. Prior to Java 1.5 if you need a Map implementation, which can be safely used in a concurrent and multi-threaded Java program, then, you only have [Hashtable](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html) or [synchronized Map](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html) because HashMap is not [thread-safe](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html). With ConcurrentHashMap, now you have a better choice; because not only it can be safely used in the concurrent multi-threaded environment but also provides better performance over Hashtable and synchronizedMap. ConcurrentHashMap performs better than earlier two because it only locks a portion of Map, instead of whole Map, which is the case with [Hashtable and synchronized Map](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html). CHM allows concurred read operations and the same time maintains integrity by synchronizing write operations.

**What are the rules for overriding methods in Java.**

* In java, a method can only be written in [Subclass](https://crunchify.com/top-10-java-interview-questions-answers-must-read-before-appearing-for-any-java-interview/), not in same class.
* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the [super class](https://crunchify.com/create-simple-pojo-and-multiple-java-reflection-examples/).
* The access level cannot be more restrictive than the overridden method’s access level.
  + For example: if the super class method is declared public then the over-ridding method in the sub class cannot be either private or [protected](https://crunchify.com/java-how-to-create-your-own-logging-level-in-log4j-configuring-log4j/).
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A [method declared static](https://crunchify.com/fundamentals-of-java-static-methods-and-variables/) cannot be overridden but can be re-declared.
* If a method cannot be inherited then it cannot be overridden.
* A subclass within the same package as the instance’s superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws [exceptions](https://crunchify.com/how-to-fix-exception-in-thread-main-java-lang-illegalmonitorstateexception-error-on-thread-wait/) or not.
  + However the overriding method should not throw [checked exceptions](https://crunchify.com/better-understanding-on-checked-vs-unchecked-exceptions-how-to-handle-exception-better-way-in-java/) that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* [Constructors](https://crunchify.com/how-to-implement-simple-circulararraylist-in-java/) cannot be overridden

**How to implement a Singleton in Java?**

To implement a Singleton pattern, we have different approaches but all of them have the following common concepts.

Private constructor to restrict instantiation of the class from other classes.

Private static variable of the same class that is the only instance of the class.

Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

In further sections, we will learn different approaches of Singleton pattern implementation and design concerns with the implementation.

1. [Eager initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#eager-initialization)
2. [Static block initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#static-block-initialization)
3. [Lazy Initialization](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#lazy-initialization)
4. [Thread Safe Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#thread-safe-singleton)
5. [Bill Pugh Singleton Implementation](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#bill-pugh-singleton)
6. [Using Reflection to destroy Singleton Pattern](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#reflection-and-singleton)
7. [Enum Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#enum-singleton)
8. [Serialization and Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples#serialization-and-singleton)