**Theory Question:**

**1. What is JVM and explain me the Java memory allocation**

Ans)

**JVM** is short for ***J***ava ***V***irtual ***M***achine. JVM is an abstract computing machine, or virtual machine. It is a platform-independent execution environment that converts Java bytecode into machine language and executes it.

A runtime instance of the Java virtual machine has a clear mission in life: to run one Java application. When a Java application starts, a runtime instance is born. When the application completes, the instance dies. If you start three Java applications at the same time, on the same computer, using the same concrete implementation, you'll get three Java virtual machine instances. Each Java application runs inside its own Java virtual machine.

The Java Virtual Machine exists only in the memory of our computer. Reproducing a machine within our computer's memory requires seven key objects: a set of registers, a stack, an execution environment, a garbage-collected heap, a constant pool, a method storage area, and a mechanism to tie it all together. This mechanism is the byte code instruction set.

Each time an object is created in Java it goes into the area of memory known as heap. The primitive variables like int and double are allocated in the stack, if they are local method variables and in the heap if they are member variables (i.e. fields of a class). In Java methods local variables are pushed into stack when a method is invoked and stack pointer is decremented when a method call is completed.

In a multi-threaded application each thread will have its own stack but will share the same heap. This is why care should be taken in your code to avoid any concurrent access issues in the heap space. The stack is thread-safe (each thread will have its own stack) but the heap is not thread-safe unless guarded with synchronization through your code.

**2. What is Polymorphism and encapsulation?**

Ans)

**Polymorphism:**

**Polymorphism in java** is a concept by which we can perform a single action by different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms. Polymorphism allows you define one interface and have multiple implementations.

There are two types of polymorphism in java: compile time polymorphism **(static polymorphism)** and runtime polymorphism **(Dynamic polymorphism)**. We can perform polymorphism in java by method overloading and method overriding.

**Runtime Polymorphism (or Dynamic polymorphism)**

*Method overriding* is a perfect example of runtime polymorphism. In this kind of polymorphism, reference of class X can hold object of class X or an object of any sub classes of class X. For e.g. if class Y extends class X then both of the following statements are valid:

Y obj = new Y ();

//Parent class reference can be assigned to child object

X obj = new Y ();

Since in method overriding both the classes (base class and child class) have same method, compile doesn’t figure out which method to call at compile-time. In this case JVM (java virtual machine) decides which method to call at runtime that’s why it is known as runtime or dynamic polymorphism.

**Compile time Polymorphism (or Static polymorphism)**

Compile time polymorphism is nothing but the method overloading in java. In simple terms we can say that a class can have more than one methods with same name but with different number of arguments or different types of arguments or both. In such scenario, compiler is able to figure out the method call at compile-time that’s the reason it is known as compile time polymorphism.

Polymorphism is extensively used in implementing inheritance.

**Difference Between Compile Time and Run Time Polymorphism**

**Compile time polymorphism:**

1. In compile time polymorphism, call is resolved by the compiler.
2. It is also known as static or early binding.
3. It is achieved by function overloading and [operator](http://abhiandroid.com/java/operators-in-java) overloading.
4. It provides fast execution because known early at compile time.
5. Compile time polymorphism is less flexible as all things execute at compile time.

**Run time polymorphism :**

1. In run time polymorphism, call is not resolved by the compiler.
2. It is also known as dynamic binding or late binding.
3. It is achieved by virtual functions.
4. It provides slow execution as compare to early binding because it is known as runtime.
5. Run time polymorphism is more flexible as all things execute at run time.

**Importance of Polymorphism**

* It reduces the complexity of the object.
* Through polymorphism complete implementation can be replaced by using same method signatures.
* It reduces the volume of work in terms of handling various objects.

**Polymorphism Important Points To Remember**

* Java does not support operator overloading directly.
* Java does not support compile time polymorphism.
* Access specifiers and access modifiers do not play any role in case of function overloading.
* If you want to keep number of arguments in each function same and still you want to overload them, then change the datatype of their arguments.
* If a function is returning the value, then it is not mandatory to catch the return value while calling it.
* Return type of a function does not play any role in case of function overloading.
* Function overloading can only be achieved by changing only in arguments.

### Encapsulation

**Encapsulation** means the localization of the information or knowledge within an object.  
**Encapsulation is also called as “Information Hiding”**.  
1) Objects encapsulate data and implementation details. To the outside world, an object is a black box that exhibits a certain behavior.  
2) The behavior of this object is what which is useful for the external world or other objects.  
3) An object exposes its behavior by means of methods or functions.  
4) The set of functions an object exposes to other objects or external world acts as the interface of the object.

**Benefits of Encapsulation**  
1) The functionality wherein we can change the implementation code without breaking the code of others who use our code is the biggest benefit of Encapsulation.  
2) Here in encapsulation we hide the implementation details behind a public programming interface. By interface, we mean the set of accessible methods our code makes available for other code to call—in other words, our code’s API.

3) By hiding implementation details, we can rework on our method code at a later point of time, each time we change out implementation this should not affect the code which has a reference to our code, as our API still remains the same.

4)The fields of a class can be made read-only or write-only.

5)A class can have total control over what is stored in its fields.

6)The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

**How to bring in Encapsulation**  
1) Make the instance variables protected.  
2) Create public accessor methods and use these methods from within the calling code.  
3) Use the JavaBeans naming convention of **getter and setter**.  
Eg: **getPropertyName**, **setPropertyName**.

**3. What is method overloading and Method over riding?**

Ans)

**Method Definition:**  
A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method’s name.

1)**Method Overloading:**  
In Java, it is possible to define two or more methods of same name in a class, provided that their argument list or parameters are different. This concept is known as Method Overloading.

1. To call an overloaded method in Java, it is must to use the type and/or number of arguments to determine which version of the overloaded method to actually call.
2. Overloaded methods may have different return types; the return type alone is insufficient to distinguish two versions of a method.
3. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
4. It allows the user to achieve compile time polymorphism.
5. An overloaded method can throw different exceptions.
6. It can have different access modifiers.

**Rules for Method Overloading**

1. Overloading can take place in the same class or in its sub-class.
2. Constructor in Java can be overloaded
3. Overloaded methods must have a different argument list.
4. Overloaded method should always be the part of the same class (can also take place in sub class), with same name but different parameters.
5. The parameters may differ in their type or number, or in both.
6. They may have the same or different return types.
7. It is also known as compile time polymorphism.

**2) Method Overriding**

Child class has the same method as of base class. In such cases child class overrides the parent class method without even touching the source code of the base class. This feature is known as method overriding.

**Rules for Method Overriding:**

1. applies only to inherited methods
2. object type (NOT reference variable type) determines which overridden method will be used at runtime
3. Overriding method can have different return type
4. Overriding method must not have more restrictive access modifier
5. Abstract methods must be overridden
6. Static and final methods cannot be overridden
7. Constructors cannot be overridden
8. It is also known as Runtime polymorphism.

### super keyword in Overriding:

When invoking a superclass version of an overridden method the super keyword is used.

**4. Why string is Immutable?**

Ans)

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable. Once string object is created its data or state can't be changed but a new string object is created.

**String Pool:**

If you simply assign a Value to String using double quotes, it is stored in area called string literal pool and one string can be referenced by many reference variables and if String Is mutable, then it will affect all reference variables.

### Thread safe:

### Immutable objects are safe when shared between multiple threads in multi-threaded applications. If something can’t be changed, then even thread cannot change it.

**Caching Hashcode**

The hashcode of string is frequently used in Java. For example, in a HashMap. Being immutable guarantees that hashcode will always the same, so that it can be cashed without worrying the changes. That means, there is no need to calculate hashcode every time it is used. This is more efficient.

**Security**

The Most important reason is security. String is widely used in java for parameters as file names or on the network to send data. If String were not immutable then anybody can modify it easily and it cause to a serious problem.

### Class Loading Mechanism

### Java class loading mechanism works on class names passed as parameters, then these classes are searched in class path. If String were mutable then any one can easily modify the mechanism.

**5. What is the difference between String and String buffer?**

Ans)

String & StringBuffer, both are used to represent the sequence of characters. However there is some difference between them

**Strings:**

* **Immutable:-** WhenYou perform any write operation on existing String object, a new object will be created, the modification is done there and that object will be returned to you. The old object will lose the reference and hence garbage collected later.
* **String Pool:-**It supports String Pool concepts. So you can write String str =”Hello”;
* **Performance :-** Low performance due to unnecessary object created
* **Overriding:-** Strings class has overridden equals() from object class, equals() checks for content equality

**StringBuffers:**

* **Mutable**:-When you perform any write operation on existing StringBuffer object, new object will not be  created and same object will be modified
* **String Pool**:-It does not support String pool concepts. So you cannot write StringBuffer strBuff =”Hello”;
* **Performance:-** High Performance
* **Overriding:-**This class has not overridden equals().so equals() will still compare for reference equality

**When to use String and StringBuffer:**

If your code demands you to write the logic, where string object will be modified continuously, then better use StringBuffer, to avoid unnecessary object creation  
If you want to enjoy string pool, and are sure string won’t be modified much, use String.

**6.What is the difference between array and array list?**

Ans)

**Difference between Array and ArrayList in Java:**

1. **Resizable :**

* Array is static in size that is fixed length data structure, one cannot change the length after creating the Array object.
* ArrayList is dynamic in size. Each ArrayList object has instance variable *capacity* which indicates the size of the ArrayList. As elements are added to an ArrayList its capacity grows automatically.

1. **Performance :** Performance of Array and ArrayList depends on the operation you are performing :  
     
   *resize() opertation :* Automatic resize of ArrayList will slow down the performance as it will use temporary array to copy elements from the old array to new array.  
   ArrayList is internally backed by Array during resizing  as it calls the native implemented method System.arrayCopy(src,srcPos,dest,destPos,length) .  
     
   *add() or get() operation :* adding an element or retrieving an element from the array or arraylist object has almost same  performance , as for ArrayList object these operations  run in constant time.

**3. Primitives :**

ArrayList can not contains primitive data types (like int , float , double) it can only contains Object while Array can contain both primitive data types as well as objects.  
One get a misconception that we can store primitives(int,float,double) in ArrayList , but it is not true    
  
Suppose we have ArrayList object ,

ArrayList  arraylistobject = new ArrayList();  
arraylistobject.add(**23**);  // try to add 23 (primitive)

JVM through Auto-boxing (converting primitives to equivalent objects internally) ensures that only objects are added to the arraylist object.   
thus, above step internally works like this:

arraylistobject.add (**new Integer(23)**);         
// Converted int primitive to Integer object and added to arraylistobject

4. **Iterating the values:**

* We can use iterator to iterate through ArrayList. The iterators returned by the ArrayList class's iterator and listiterator method are [fail-fast](http://javahungry.blogspot.ca/2014/04/fail-fast-iterator-vs-fail-safe-iterator-difference-with-example-in-java.html).
* We can use for loop or for each loop to iterate through array .

**5. Type-Safety:**

In Java , one can ensure Type Safety through Generics. while Array is a homogeneous data structure , thus it will contain objects of specific class or primitives of specific  data type. In array if one try to store the different data type other than the specified while creating the array object, ArrayStoreException is thrown.

for example :

String temp[] =  new String[2];  // creates a string array of size 2  
temp[0] = new Integer(12); // throws ArrayStoreException, trying to add Integer object in String[]

**6. Length :** Length of the ArrayList is provided by the size() method while Each array object has the length variable which returns the length of the array.

for example :

* Integer arrayobject[] = new Integer[3];  
  arraylength= arrayobject.length   ;  //uses arrayobject length variable
* ArrayList  arraylistobject = new ArrayList();  
  arraylistobject.add(12);   
  arraylistobject.size();   //uses arraylistobject size method

**7. Adding elements :** We can insert elements into the arraylist object using the add() method while  in array we insert elements using the assignment operator.

for example :

* Integer addarrayobject[] = new Integer[3];  
  addarrayobject[0]= new Integer(8)   ;  //new object is added to the array object

**8.Multi-dimensional:**

Array can be multidimensional, while ArrayList is always single dimensional.

example of multidimensional array:

* Integer addarrayobject[][] = new Integer[3][2];  
  addarrayobject[0][0]= new Integer(8)

**Similarities Between Array and ArrayList**  
  
**1. add and get method:** Performance of Array and ArrayList are similar for the add and get operations. Both operations run in constant time.  
  
**2. Duplicate elements:** Both array and arraylist can contain duplicate elements.  
  
**3. Null Values:** Both can store null values and uses index to refer to their elements.  
  
**4. Unordered:**  Both does not guarantee ordered elements.

**Recap: Difference between Array and ArrayList in Java**

|  |  |  |
| --- | --- | --- |
|  | **Array** | **ArrayList** |
|  |  |  |
| Resizable | No | Yes |
|  |  |  |
| Primitives | Yes | No |
|  |  |  |
| Iterating values | for, for each | Iterator, for each |
|  |  |  |
| Length | length variable | size method |
|  |  |  |
| Performance | Fast | Slow in comparision |
|  |  |  |
| Multidimensional | Yes | No |
| Add Elements | Assignment operator | add method |

**7. What is the difference between hash map and Hash table?**

Ans)

**Difference between HashMap and HashTable / HashMap vs HashTable** :

**1. Synchronization or Thread-Safe:**

* This is the most important difference between two. HashMap is non synchronized and not thread safe. On the other hand, HashTable is thread safe and synchronized.
* When to use HashMap: If your application does not require any multi-threading task, in other words hashmap is better for non-threading applications. HashTable should be used in multithreading applications.

**2. Null keys and null values:**

Hashmap allows one null key and any number of null values, while Hashtable do not allow null keys and null values in the HashTable object.

**3. Iterating the values:**  Hashmap object values are iterated by using iterator. HashTable is the only class other than vector which uses enumerator to iterate the values of HashTable object.

**4.  Fail-fast iterator**:

The iterator in Hashmap is fail-fast iterator while the enumerator for Hashtable is not.  
According to Oracle Docs, if the Hashtable is structurally modified at any time after the iterator is created in any way except the iterator's own remove method, then the iterator will throw ConcurrentModification Exception.

Structural modification means adding or removing elements from the Collection object (here hashmap or hashtable). Thus the enumerations returned by the Hashtable keys and elements methods are not fail fast.We have already explained the difference between iterator and enumeration.

**5. Performance:**  Hashmap is much faster and uses less memory than Hashtable as former is unsynchronized. Unsynchronized objects are often much better in performance in compare to synchronized object like Hashtable in single threaded environment.

**6. Superclass and Legacy:**  Hashtable is a subclass of Dictionary class which is now obsolete in Jdk 1.7 ,so ,it is not used anymore. It is better off externally synchronizing a HashMap or using a ConcurrentMap implementation (e.g ConcurrentHashMap).HashMap is the subclass of the AbstractMap class. Although Hashtable and HashMap has different superclasses but they both are implementations of the *"Map"* abstract data type.

**Similarities Between HashMap and Hashtable**  
  
**1. Insertion Order:**   Both HashMap and Hashtable  does not guarantee that  the order of the map will remain constant over time. Instead use LinkedHashMap, as the order remains constant over time.  
  
**2. Map interface:**   Both HashMap and Hashtable implements Map interface.  
  
**3. Put and get method:**  Both HashMap and Hashtable provides constant time performance for put and get methods assuming that the objects are distributed uniformly across the bucket.  
  
**4. Internal working:**  Both HashMap and Hashtable works on the Principle of Hashing.

**When to use HashMap and Hashtable?**  
  
*1. Single Threaded Application*  
  
HashMap should be preferred over Hashtable for the non-threaded applications. In simple words, use HashMap in unsynchronized or single threaded applications .  
  
*2. Multi- Threaded Application*  
  
We should avoid using Hashtable, as the class is now obsolete in latest Jdk 1.8. Oracle has provided a better replacement of Hashtable named ConcurrentHashMap. For multithreaded application prefer ConcurrentHashMap instead of Hashtable.

**Recap: Difference between HashMap and Hashtable in Java**

|  |  |  |
| --- | --- | --- |
|  | **HashMap** | **Hashtable** |
|  |  |  |
| Synchronized | No | Yes |
|  |  |  |
| Thread-Safe | No | Yes |
|  |  |  |
| Null Keys and  Null values | One null key,  Any null values | Not permit null keys  and values |
|  |  |  |
| Iterator type | Fail fast iterator | Fail safe iterator |
|  |  |  |
| Performance | Fast | Slow in comparison |
|  |  |  |
| Superclass and Legacy | AbstractMap, No | Dictionary, Yes |

**8. What is a vector in Java?**

Ans)

Vectors are dynamically-allocated. They aren't declared to contain a type of variable; instead, each Vector contains a dynamic list of references to other objects. The Vector class is found in the java.util package, and extends java.util.Abstractlist.   
  
The big advantage of using Vectors is that the size of the vector can change as needed.

Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

**Three ways to create vector class object:**

**Method 1:**  
Vector vec = new Vector ();  
It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

**Method 2:**  
Syntax: Vector object= new Vector (int initialCapacity)  
Vector vec = new Vector (3);  
It will create a Vector of initial capacity of 3.

**Method 3:**  
Syntax:  
Vector object= new vector (int initialcapacity, capacityIncrement)  
Vector vec= new Vector (4, 6)  
Here we have provided two arguments. The initial capacity is 4 and capacityIncrement is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

**Important methods of Vector Class:**

1. **void addElement (Object element):** It inserts the element at the end of the Vector.
2. **int capacity ():** This method returns the current capacity of the vector.
3. **int size ():** It returns the current size of the vector.
4. **void setSize (int size):** It changes the existing size with the specified size.
5. **boolean contains (Object element):** This method checks whether the specified element is present in the Vector. If the element has been found it returns true else false.
6. **boolean containsAll (Collection c):** It returns true if all the elements of collection c are present in the Vector.
7. **Object elementAt (int index):** It returns the element present at the specified location in Vector.
8. **Object firstElement ():** It is used for getting the first element of the vector.
9. **Object lastElement ():** Returns the last element of the array.
10. **Object get (int index):** Returns the element at the specified index.
11. **boolean isEmpty ():** This method returns true if Vector doesn’t have any element.
12. **boolean removeElement (Object element):** Removes the specifed element from vector.
13. **boolean removeAll (Collection c):** It Removes all those elements from vector which are present in the Collection c.
14. **void setElementAt (Object element, int index):** It updates the element of specifed index with the given element.

**9. What is set in java?**

Ans)

A collection in which an object may be "present or not" (but not present multiple times) is called a **set**.

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

Set in Java doesn't maintain any order. Though Set provide another alternative called SortedSet which can store Set elements in specific Sorting order defined by Comparable and Comparator methods of Objects stored in Set.

**Set Implementations**

**1.HashSet:**

HashSet is the implementation of Set Interface which uses Hash table for storing the data. Hash table internally uses a phenomena known as hashing, Hash set does not maintain the insertion order, that is when we retrieve values from it we do not get that values in the same order as we have entered in it.

**2.LinkedHashSet:**

LinkedHashSet is also the implementation of Set Interface. It is almost similar to hashset, but it maintains insertion order i.e. values are retrieved in the same order in which they are added. It uses doubly linked list to obtain this functionality.

1. **TreeSet:**

TreeSet also implements Set Interface, it is also similar to hashset , but it stores all the elements in their natural order , like all integer values are stored in ascending order   and strings are stored according to Dictionary values.

Apart from adding this functionality of maintaining natural ordering, TreeSet do not allow null values.

TreeSet is best choice for storing large amount of data, as its retrieval and access time is very fast, which makes data to be found in no time.

**Importance of Set Interface**

* Set has its own importance like in case we just want to remove all the duplicate elements from list, we can convert that list into set, and can very easily remove all the duplicate elements, and convert back this set to list object.
* Set has various implementations like HashSet, LinkedHashSet and TreeSet. These implementations are very helpful in storing the elements according to the need, like if we want our data to be stored in natural order than we can use TreeSet, or we want data as it is we have entered than we can use LinkedHashSet.

The methods declared by Set are summarized in the following table:

**1. boolean add(object)**:

This methods adds an element to the set, if and only if that element is not already present in the set, It will return true if element is added, or false if given element is already present and hence it is not added

**2. void clear():**

This method removes all elements from the Set.

**3. boolean contains(Object)**

This method returns true if the Set contains the given element in the set, else false if there is no such element.

**4. boolean isEmpty()**

This method returns true if the Set contains no elements or false if there are elements present in the set.

**5. Iterator iterator()**

This method gives an object of Iterator class which can be used to traverse through the Set.

**6. boolean remove(Element e)**

This method removes the given element in the argument from the Set and returns true if the argument was removed, else false if the given element is not present.

**7. int size()**

This method gives the number of elements in the Set.

**8. Object[] toArray()**

This method returns the set in the form of array of type “Object

**Important Points About Set in JAVA**

* Set has three implementation classes HashSet, LinkedHashSet and TreeSet. If we have requirement of storing elements in natural order than we should use TreeSet and if we have requirement of retrieving the elements as it is entered than we should use LinkedHashSet, and if no such requirements are there we can use HashSet.
* Duplicate values are not allowed
* Only one null element is allowed as duplicity is not allowed

**Set Quick Revision points**

* Duplicates are not allowed
* Only one Null value is allowed
* TreeSet is preferred where we want to store data in natural order
* LinkedHashSet is preferred where we want to maintain insertion order.
* HashSet is preferred where we just want to store data without any requirement of maintaining any special ordering

**10. What is an abstract class?**

Ans)

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods. The purpose of an abstract class is to function as a base for subclasses.

**Declaring an Abstract Class in Java**

In Java you declare that a class is abstract by adding the abstract keyword to the class declaration. Here is a Java abstract class example:

public abstract class MyAbstractClass {

}

That is all there is to declaring an abstract class in Java. Now you cannot create instances of MyAbstractClass. Thus, the following Java code is no longer valid:

MyAbstractClass myClassInstance = new MyAbstractClass(); //not valid

If you try to compile the code above the Java compiler will generate an error, saying that you cannot instantiate MyAbstractClass because it is an abstract class.

**Abstract Methods**

An abstract class can have abstract methods. You declare a method abstract by adding the abstract keyword in front of the method declaration. Here is a Java abstract method example:

public abstract class MyAbstractClass {

public abstract void abstractMethod();

}

An abstract method has no implementation. It just has a method signature. Just like methods in a Java interface.

If a class has an abstract method, the whole class must be declared abstract. Not all methods in an abstract class have to be abstract methods. An abstract class can have a mixture of abstract and non-abstract methods.

Subclasses of an abstract class must implement (override) all abstract methods of its abstract superclass. The non-abstract methods of the superclass are just inherited as they are. They can also be overridden, if needed.

Here is an example subclass of the abstract class MyAbstractClass:

public class MySubClass extends MyAbstractClass {

public void abstractMethod() {

System.out.println("My method implementation");

}

}

Notice how MySubClass has to implement the abstract method abstractMethod() from its abstract superclass MyAbstractClass.

The only time a subclass of an abstract class is not forced to implement all abstract methods of its superclass, is if the subclass is also an abstract class.

**The Purpose of Abstract Classes**

The purpose of abstract classes is to function as base classes which can be extended by subclasses to create a full implementation.

You can use abstract class as parent class for sub classes

* When you expect to implement same method in all sub classes with different implementation detail.
* When you expect to implement methods in super class (abstract class) which can be used by its sub classes if needed.
* When you expect to implement independent method in sub class which do not have any relation or use with its super class.

### Abstraction in Java

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

### Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

**Points to Remember About Abstract Class**

1. If class is declared with abstract keyword, then it is called abstract class in java software development language.
2. If class has abstract method, you must have to declare class as abstract class.
3. In abstract class, you can only declare abstract methods but you cannot define abstract methods.
4. Sub classes of abstract class must have to implement all abstract methods of super abstract class.
5. You cannot instantiate (cannot create object of) abstract class.
6. Abstract classes can have concrete methods, constructors and Member variables too.

**11. What is an interface?**

Ans)

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

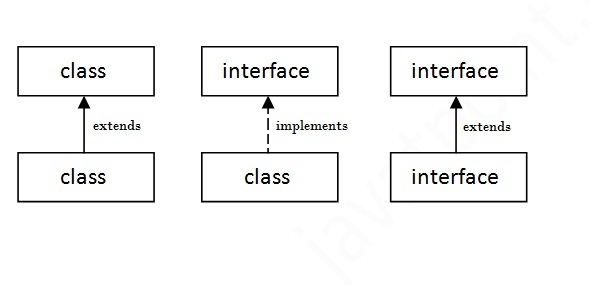
There are mainly three reasons to use interface. They are given below.

* It is used to achieve fully abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.

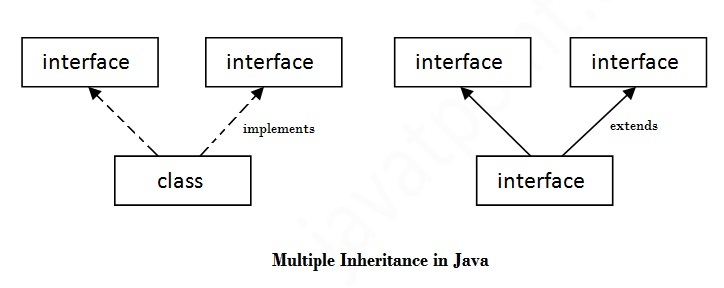
**Understanding relationship between classes and interfaces**

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



**Multiple inheritance in Java by interface**

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple inheritance is not supported through class in java but it is possible by interface, why?

|  |
| --- |
| Multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.  **Interface inheritance**  A class implements interface but one interface extends another interface.  What is marker or tagged interface?  An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.  //How Serializable interface is written?  public interface Serializable{  }  **Nested Interface in Java**  An interface can have another interface i.e. known as nested interface.  **Points to Remember About Interface:**   * Interface can have **only abstract** methods. * Interface **supports multiple inheritance**. * Interface has **only static and final variables**. * Interface **can't have static methods, main method or constructor**. * Interface **can't provide the implementation of abstract class**. * The **interface keyword** is used to declare interface. * **Example:** public interface Drawable{ void draw(); } * interface achieves fully abstraction (100%).   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.   However, an interface is different from a class in several ways, including:   * You 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:**   * 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.   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, similarly to the way that a class can extend another class. |

**12. Why Java is Platform independent?**

Ans)

Java has been termed as a ‘Platform Independent Language’ as it primarily works on the notion of ‘compile once, run everywhere’. Here’s a sequential step establishing the Platform independence feature in Java:

* The Java Compiler outputs Non-Executable Codes called ‘Bytecode’.
* Bytecode is a highly optimized set of computer instruction which could be executed by the Java Virtual Machine (JVM).
* The translation into Bytecode makes a program easier to be executed across a wide range of platforms, since all we need is a JVM designed for that particular platform.
* JVMs for various platforms might vary in configuration, those they would all understand the same set of Bytecode, thereby making the Java Program ‘Platform Independent’.

**13. What are access modifiers? Give me an example?**

Ans)

**Access Modifiers in Java with Examples**

Java provides us a many number of *access modifiers* to set access levels *for class, variables, methods and constructor*. It means the access modifiers in java specify scope of a data member, method, constructor or a class. The four access modifiers in JAVA are private, default, protected and public.

**Types of Access Modifiers**

There are 4 types of java access modifiers:

* **private –** accessible within class only
* **default –** accessible available to the package only
* **protected –** accessible within package and outside the package but through *inheritance* only
* **public –** accessible everywhere

**Private Access Modifier:**

The private access modifier is accessible only within class. It means the methods, variables and constructors that are declared as private can only be accessed within the declared class itself. It is very restrictive access modifier.

Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

**Important Note:** We cannot make class and interfaces private.

**Program example of private access modifier:**

Let us take an example to show the use of private access modifier.

**Step 1:** First we create class **PrivateClass**in which we declare one private data member and one private method:

class PrivateClass

{

private int x= 10;

private void show()

{

System.out.println("Private class");

}

}

**Step 2:** Second we create a class **PrivateAccess** in which we call the **PrivateClass** class data member and method:

class PrivateAccess

{

public [static](http://abhiandroid.com/java/static-keyword.html) void main(String args[])

{

PrivateClass obj=new PrivateClass();

System.out.println(obj.x);  //Compile Time Error

obj.show();  //Compile Time Error

}

}

**Explanation of Example:**

In **PrivateAccess** class when we are trying to call the private data member and method of a **PrivateClass** class it gives us compile time error because private data members and methods have a access level to **PrivateClass** class only.

**Default Access Modifier:**

If we don’t use any modifier, it is treated as default access modifier by default. In other words, we can say it id default if no access modifier for a class, method, field, etc is explicitly declared. The default modifier is accessible only within package.

A variable or method which is declared without any access modifier is available to any other class in the same package.

**Program Example of default access modifier**

Let us take an example to show the use of default access modifier.

**Step 1:** First we create a class **DefaultClass** under a package **pack** in which we declare default method:

package pack;

class DefaultClass

{

void show()

{

System.out.println("Default class");

}

}

**Step 2:** Second we create a class **DefaultAccess** under a package **mypack** and imports the above package **pack:**

package mypack;

import pack.\*;

class DefaultAccess

{

public static void main(String args[])

{

DefaultClass obj = new DefaultClass(); //Compile Time Error

obj.show(); //Compile Time Error

}

}

**Explanation of Example:**

In the above example, the scope of class **DefaultClass** and its method **show()** is default so it cannot be accessed from outside the package. As a result, it shows compile time error.

**Protected Access Modifier:**

We can access protected access modifier either within package and outside the package but through inheritance only. It means variables, methods and constructors which are declared as protected in a base class can be accessed only by the child classes in other packages.

The protected access modifier can be applied on the data member, method and constructor. It cannot be applied on the class and interfaces. Protected access modifier gives a chance to the child class to use the helper method or variable, while preventing from trying to use a class i.e. non related.

**Program Example of protected access modifier**

Let us take an example to show use of protected access modifier:

**Step 1:** First we create a class **ProtectedClass** under a package **pack1** in which we declare a protected method show():

package pack1;

class ProtectedClass

{

protected void show()

{

System.out.println("Protected class");

}

}

**Step 2:** Second we create a class **ProtectedAccess** under a package **mypack1** and import a package **pack1:**

package mypack1;

import pack1.\*;

class ProtectedAccess extends ProtectedClass

{

public static void main(String args[])

{

ProtectedAccess obj = new ProtectedAccess();

obj.show();

}

}

**Output:**

Protected class

**Explanation of Example:**

The ProtectedClass class of pack1 package is public, so it can be accessed from outside the package. But show() method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

**Public Access Modifier:**

The public access modifier is accessible everywhere. It means a class, method, constructor, interface etc declared as public can be accessed from any other class. It has the widest scope among all other modifiers.

**Program Example of public access modifier**

Let us take an example to show the use of public access modifier.

**Step 1:** First we create a class **PublicClass** in which we declare the public method **show()**:

class PublicClass

{

public void show()

{

System.out.println("Public class");

}

}

**Step 2:** Second we create a class **PublicAccess** in which we call the method of **PublicClass** class:

class PublicAccess

{

public static void main(String args[])

{

PublicClass obj = new PublicClass();

obj.show();

}

}

**Output:**

Public class

**Understanding all access modifiers in java**

Let’s understand the access modifiers with the help of following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Access Modifier | within a class | within a package | outside a package by subclass only | outside a package |
| Private | Yes | No | No | No |
| Default | Yes | Yes | No | No |
| Protected | Yes | Yes | Yes | No |
| Public | Yes | Yes | Yes | Yes |

Top of Form

Bottom of Form

Top of Form

Bottom of Form

Top of Form

Bottom of Form

**14. What are java exceptions? Give me an example**

Ans)

**Exception:**

An Exception can be anything which interrupts the normal flow of the program. When an exception occurs program processing gets terminated and doesn’t continue further. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled.

**When an exception can occur?**  
Exception can occur at runtime (known as runtime exceptions) as well as at compile-time (known Compile-time exceptions).

**Reasons for Exceptions**  
There can be several reasons for an exception. For example, following situations can cause an exception – Opening a non-existing file, Network connection problem, Operands being manipulated are out of prescribed ranges, class file missing which was supposed to be loaded and so on.

**Java’s exception handling consists of three operations:**

1. Declaring exceptions;
2. Throwing an exception; and
3. Catching an exception.

**Advantage of Exception Handling**

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5;//exception occurs
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

**Types of Exception**

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

## **Difference between checked and unchecked exceptions**

### 1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

### 2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

### 3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

### Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

### 1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

1. int a=50/0;//ArithmeticException

### 2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

1. String s=null;
2. System.out.println(s.length());//NullPointerException

### 3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

String s="abc";

1. int i=Integer.parseInt(s);//NumberFormatException

### 4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

## **Java Exception Handling Keywords**

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

**try-catch block:**

* **try block:**

Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block.

**Syntax of java try-catch**

1. try{
2. //code that may throw exception
3. }catch(Exception\_class\_Name ref){}

**Syntax of try-finally block**

1. try{
2. //code that may throw exception
3. }finally{}

**catch block:**

Java catch block is used to handle the Exception. It must be used after the try block only. You can use multiple catch block with a single try.

1. public class Testtrycatch2{
2. public static void main(String args[]){
3. try{
4. int data=50/0;
5. }catch(ArithmeticException e){System.out.println(e);}
6. System.out.println("rest of the code...");
7. }
8. }

**Multi catch block:**

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

1. public class TestMultipleCatchBlock{
2. public static void main(String args[]){
3. try{
4. int a[]=new int[5];
5. a[5]=30/0;
6. }
7. catch(ArithmeticException e){System.out.println("task1 is completed");}
8. catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
9. catch(Exception e){System.out.println("common task completed");}
11. System.out.println("rest of the code...");
12. }
13. }

Output: task1 completed

rest of the code...

**Rules:**

* At a time only one Exception is occured and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

class TestMultipleCatchBlock1{

  public static void main(String args[]){

   try{

    int a[]=new int[5];

    a[5]=30/0;

  }

   catch(Exception e){System.out.println("common task completed");}

   catch(ArithmeticException e){System.out.println("task1 is completed");}

   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

   System.out.println("rest of the code...");

 }

}

Output: Compile-time error

**Nested try block:**

The try block within a try block is known as nested try block in java.

### Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

**Syntax:**

....

try

{

    statement 1;

    statement 2;

    try

   {

        statement 1;

        statement 2;

    }

 catch(Exception e)

   {

    }

}

catch(Exception e)

{

}

....

**finally block:**

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not. Java finally block must be followed by try or catch block.

**Why use java finally**

* Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

## **Usage of Java finally**

Let's see the different cases where java finally block can be used.

### Case 1

Let's see the java finally example where **exception doesn't occur**.

1. class TestFinallyBlock{
2. public static void main(String args[]){
3. try{
4. int data=25/5;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

Output:5

finally block is always executed

rest of the code...

### Case 2

Let's see the java finally example where **exception occurs and not handled**.

1. class TestFinallyBlock1{
2. public static void main(String args[]){
3. try{
4. int data=25/0;
5. System.out.println(data);
6. }
7. catch(NullPointerException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

Output: finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

### Case 3

Let's see the java finally example where **exception occurs and handled**.

1. public class TestFinallyBlock2{
2. public static void main(String args[]){
3. try{
4. int data=25/0;
5. System.out.println(data);
6. }
7. catch(ArithmeticException e){System.out.println(e);}
8. finally{System.out.println("finally block is always executed");}
9. System.out.println("rest of the code...");
10. }
11. }

Output: Exception in thread main java.lang.ArithmeticException:/ by zero

finally block is always executed

rest of the code...

**Rules:**

* If you don't handle exception, before terminating the program, JVM executes finally block(if any).
* For each try block there can be zero or more catch blocks, but only one finally block.
* The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

**throw keyword:**

The Java throw keyword is used to explicitly throw an exception. We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

The syntax of java throw keyword is given below.

1. throw exception;

Let's see the example of throw IOException.

throw new IOException("sorry device error);

**throw keyword example:**

1. public class TestThrow1{
2. static void validate(int age){
3. if(age<18)
4. throw new ArithmeticException("not valid");
5. else
6. System.out.println("welcome to vote");
7. }
8. public static void main(String args[]){
9. validate(13);
10. System.out.println("rest of the code...");
11. }
12. }

Output:

Exception in thread main java.lang.ArithmeticException:not valid

**Java Exception propagation:**

|  |
| --- |
| An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation. |

**Rules:**

* By default Unchecked Exceptions are forwarded in calling chain (propagated).
* By default, Checked Exceptions are not forwarded in calling chain (propagated).

**throws keyword:**

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

**Syntax of java throws**

1. return\_type method\_name() throws exception\_class\_name{
2. //method code
3. }

### Which exception should be declared

**Ans)** checked exception only, because:

* **unchecked Exception:** under your control so correct your code.
* **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

### Advantage of Java throws keyword

* Now Checked Exception can be propagated (forwarded in call stack).
* It provides information to the caller of the method about the exception.

**Java throws example:**

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

1. import java.io.IOException;
2. class Testthrows1{
3. void m()throws IOException{
4. throw new IOException("device error");//checked exception
5. }
6. void n()throws IOException{
7. m();
8. }
9. void p(){
10. try{
11. n();
12. }catch(Exception e){System.out.println("exception handled");}
13. }
14. public static void main(String args[]){
15. Testthrows1 obj=new Testthrows1();
16. obj.p();
17. System.out.println("normal flow...");
18. }
19. }

Output:

exception handled

normal flow...

|  |
| --- |
| **There are two cases:**   1. Case1:You caught the exception i.e. handle the exception using try/catch. 2. Case2:You declare the exception i.e. specifying throws with the method. |

**Case1: You handle the exception**

* In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

import java.io.\*;

class M{

 void method()throws IOException{

  throw new IOException("device error");

 }

}

public class Testthrows2{

   public static void main(String args[]){

   try{

M m=new M();

     m.method();

   }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");

  }

}

**Case2: You declare the exception**

1. In case you declare the exception, if exception does not occur, the code will be executed fine.
2. In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.
3. ***Program if exception does not occur***
4. import java.io.\*;
5. class M{
6. void method()throws IOException{
7. System.out.println("device operation performed");
8. }
9. }
10. class Testthrows3{
11. public static void main(String args[])throws IOException{//declare exception
12. M m=new M();
13. m.method();
15. System.out.println("normal flow...");
16. }
17. }

Output:device operation performed

normal flow...

***B) Program if exception occurs***

1. import java.io.\*;
2. class M{
3. void method()throws IOException{
4. throw new IOException("device error");
5. }
6. }
7. class Testthrows4{
8. public static void main(String args[])throws IOException{//declare exception
9. M m=new M();
10. m.method();
12. System.out.println("normal flow...");
13. }
14. }

Output: Runtime Exception

### Can we rethrow an exception?

Yes, by throwing same exception in catch block.

**List of checked exception**

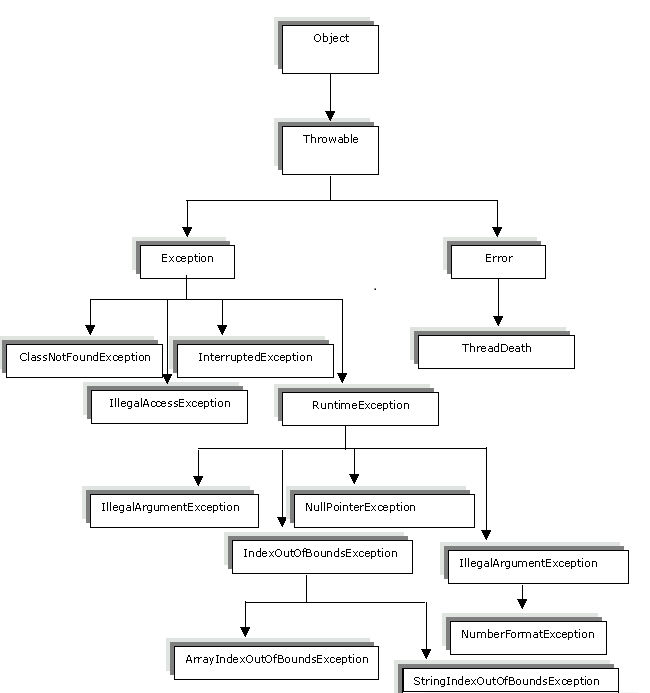
ClassNotFoundException: Class not found.  
CloneNotSupportedException: Attempt to clone an object that does not implement the Cloneable interface  
IllegalAccessException: Access to a class is denied.  
InstantiationException: Attempt to create an object of an abstract class or interface.  
InterruptedException: One thread has been interrupted by another thread.  
NoSuchFieldException: A requested field does not exist.  
NoSuchMethodException: A requested method does not exist.

**List of unchecked exception**

ArithmeticException: Arithmetic error, such as divide-by-zero.  
ArrayIndexOutOfBoundsException: Array index is out-of-bounds.  
ArrayStoreException: Assignment to an array element of an incompatible type.  
ClassCastException: Invalid cast.  
IllegalArgumentException: Illegal argument used to invoke a method.  
IllegalMonitorStateException: Illegal monitor operation, such as waiting on an unlocked thread.  
IllegalStateException: Environment or application is in incorrect state.  
IllegalThreadStateException: Requested operation not compatible with current thread state.  
IndexOutOfBoundsException: Some type of index is out-of-bounds.  
NegativeArraySizeException: Array created with a negative size.  
NullPointerException: Invalid use of a null reference.  
NumberFormatException: Invalid conversion of a string to a numeric format.  
SecurityException: Attempt to violate security.  
StringIndexOutOfBounds: Attempt to index outside the bounds of a string.  
UnsupportedOperationException: An unsupported operation was encountered.

**ExceptionHandling with MethodOverriding in Java**

|  |
| --- |
| There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:   * **If the superclass method does not declare an exception**   + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception. * **If the superclass method declares an exception**   + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception. |



**Exception hierarchy**

**Points to Remember:**

* + Both catch and finally blocks are optional, but at least one must follow a try.
  + The try-catch-finally structure can be nested in try, catch, or finally blocks.
  + The finally block is used to clean up resources, particularly in the context of I/O.
  + If you omit the catch block, the finally block is executed before the exception is propagated.
  + Exceptions can be caught at any level.
  + If they are not caught, they are said to *propagate* to the next method.
  + Exceptions can be caught based on their generic type.
  + catch statements are like repeated else-ifs.
  + At most one catch bock is executed.
  + Be sure to catch generic exceptions after the specific ones.

**15. What is the difference between throws and throwable?**

Ans)

**throws:**

* throws is also keyword in java
* throws keyword is used in conjunction with method signature, to signify what kind of exception the method can throw.

**Throwable:**

* Throwable is super class of all exceptions and errors.
* All kind of exceptions and error are kind of throwable type.

You can use the following methods to retrieve the type of the exception and the state of the program from the Throwable object:

printStackTrace():

Prints this Throwable and its call stack trace to the standard error stream System.err. The first line of the outputs contains the result of toString(), and the remaining lines are the stack trace. This is the most common handler, if there is nothing better that you can do. For example,

try {

Scanner in = new Scanner(new File("test.in"));

// process the file here

......

} catch (FileNotFoundException ex) {

ex.printStackTrace();

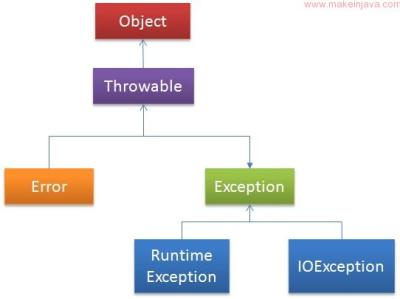
}

You can also use printStackTrace(PrintStream s) or printStackTrace(PrintWriter s).

getMessage(): Returns the message specified if the object is constructed using constructor Throwable(String message).

toString(): Returns a short description of this Throwable object, consists of the name of the class, a colon ':', and a message from getMessage().

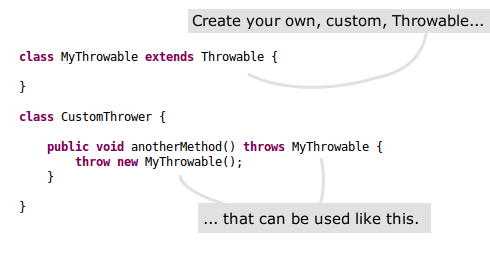
The **class hierarchy of Throwable** class is as follows:

[](http://i1.wp.com/www.makeinjava.com/wp-content/uploads/2016/04/ExceptionClassHierarchy.jpg)

Exception class hierarchy

* A class which you must extend in order to create your own, custom, throwable.

**Example:**

[](http://i.stack.imgur.com/jXAmN.png)

A throwable contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error. Finally, it can contain a cause: another throwable that caused this throwable to get thrown. The cause facility was added in release 1.4. It is also known as the chained exception facility, as the cause can, itself, have a cause, and so on, leading to a "chain" of exceptions, each caused by another.

* An object to Throwable or to it’s sub classes can be explicitly created and thrown by using **throw** keyword.
* Such explicitly thrown exception must be handled some where in the program, otherwise program will be terminated.
* It is not compulsory that explicitly thrown exception must be handled by immediately following try-catch block. It can be handled by any one of it’s enclosing try-catch blocks.

**16. What is the difference between Error and exception?**

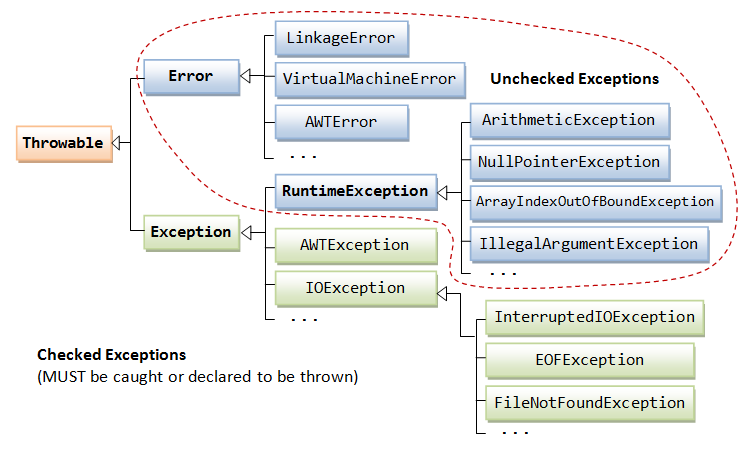
Ans)

|  |  |
| --- | --- |
| **Errors** | **Exceptions** |
| 1. Errors in java are of type java.lang.Error. | 1. Exceptions in java are of type java.lang.Exception. |
| 2. All errors in java are unchecked type. | 2. Exceptions include both checked as well as unchecked type. |
| 3. Errors happen at run time. They will not be known to compiler. | 3. Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time. |
| 4. It is impossible to recover from errors. | 4. You can recover from exceptions by handling them through try-catch blocks. |
| 5. Errors are mostly caused by the environment in which application is running. | 5. Exceptions are mainly caused by the application itself. |
| 6. Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError | 6. Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException |

**17. What is the difference between Error, throwable and exception?**

Ans)

**Exception Classes - Throwable, Error, Exception & RuntimeException**



The figure ‎above shows the hierarchy of the Exception classes. The base class for all Exception objects is java.lang.Throwable, together with its two subclasses java.lang.Exception and java.lang.Error.

* The Error class describes internal system errors (e.g., VirtualMachineError, LinkageError) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.
* The Exception class describes the error caused by your program (e.g. FileNotFoundException, IOException). These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

**18. What are collection APIs, give me an example**

Ans)

Java Collections framework API is a unified architecture for representing and manipulating collections. The API contains Interfaces, Implementations & Algorithm to help java programmer in everyday programming. In nutshell, this API does 6 things at high level

* Reduces programming efforts. - Increases program speed and quality.
* Allows interoperability among unrelated APIs.
* Reduces effort to learn and to use new APIs.
* Reduces effort to design new APIs.
* Encourages & Fosters software reuse.

To be specific, there are six collection java interfaces. The most basic interface is Collection. Three interfaces extend Collection: Set, List, and SortedSet. The other two collection interfaces, Map and SortedMap, do not extend Collection, as they represent mappings rather than true collections.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

* **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations, i.e., Classes:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

## **The Collection Interfaces:**

The collections framework defines several interfaces. This section provides an overview of each interface:

|  |  |
| --- | --- |
| **SN** | **Interfaces with Description** |
| 1 | **The Collection Interface:**  This enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| 2 | **The List Interface:**  This extends **Collection** and an instance of List stores an ordered collection of elements. |
| 3 | **The Set:**  This extends Collection to handle sets, which must contain unique elements |
| 4 | **The SortedSet**  This extends Set to handle sorted sets |
| 5 | **The Map**  This maps unique keys to values. |
| 6 | **The Map.Entry**  This describes an element (a key/value pair) in a map. This is an inner class of Map. |
| 7 | **The SortedMap**  This extends Map so that the keys are maintained in ascending order. |
| 8 | **The Enumeration**  This is legacy interface and defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. |

**The Collection Classes:**

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table:

|  |  |
| --- | --- |
| **SN** | **Classes with Description** |
| 1 | **AbstractCollection:**  Implements most of the Collection interface. |
| 2 | **AbstractList**  Extends AbstractCollection and implements most of the List interface. |
| 3 | **AbstractSequentialList:**  Extends AbstractList for use by a collection that uses sequential rather than random access of its elements. |
| 4 | **LinkedList:**  Implements a linked list by extending AbstractSequentialList.  **ArrayList:** |
| 5 | Implements a dynamic array by extending AbstractList. |
| 6 | **AbstractSet**  Extends AbstractCollection and implements most of the Set interface.  **HashSet:** |
| 7 | Extends AbstractSet for use with a hash table.  **LinkedHashSet:** |
| 8 | Extends HashSet to allow insertion-order iterations.  **TreeSet:** |
| 9 | Implements a set stored in a tree. Extends AbstractSet. |
| 10 | **AbstractMap**  Implements most of the Map interface.  **HashMap:** |
| 11 | Extends AbstractMap to use a hash table.  **TreeMap:** |
| 12 | Extends AbstractMap to use a tree.  **WeakHashMap:** |
| 13 | Extends AbstractMap to use a hash table with weak keys.  **LinkedHashMap:** |
| 14 | Extends HashMap to allow insertion-order iterations  **IdentityHashMap:** |
| 15 | Extends AbstractMap and uses reference equality when comparing documents. |

The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

**The Collection Algorithms:**

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

Collections define three static variables: EMPTY\_SET, EMPTY\_LIST, and EMPTY\_MAP. All are immutable.

**How to use an Iterator?**

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.

The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements.

**How to use a Comparator?**

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

This interface lets us sort a given collection any number of different ways. Also this interface can be used to sort any instances of any class (even classes we cannot modify).

**19. What is the difference between final and finally?**

Ans)

**final** - final keyword can be used with a class, variable or a method.

* A variable declared as final acts as constant, which means once a variable is declared and assigned, the value cannot be changed. An object can also be final, which means that the once the object is created it cannot be assigned a different object, although the properties or fields of the object can be changed.
* A final class is immutable, which means that no other class can extend from it. E.g String, Integer.
* A final method in a class cannot be overridden in the child class.

**finally** - finally keyword is used with try-catch block for handling exception. The finally block is optional in try-catch block. The finally code block is always executed after try or catch block is completed. The general use case for finally block is to close the resources or clean up objects used in try block. For e.g. Closing a FileStream, I/O stream objects, Database connections, HTTP connections are generally closed in a finally block.

**Points to remember:**

final ----> Keyword in java that can be used for classes, methods and variables.  
\* If a class is declared as final, it cannot be sub-classed/inherited  
\* If a method is declared as final, it cannot be overriden  
\* if a variable is declared as final, it's value cannot be changed

finally ---------> Its a block in java that is used in exception hadling.  
\* The code written in finally block will be executed even when an exception in thrown or not.

finalize --------> Method that is used in the concept of Garbage Collection.  
\* Before JVM performs the Garbage Collection, if you want to perform important tasks like closing the connection object and so on, this kind of code can be written in finalize method

**20. Will java supports multiple inheritance?**

Ans)

### Multiple inheritance:

* The concept of Getting the properties from multiple class objects to sub class object with same priorities is known as multiple inheritance.
* Java Doesn't Support multiple Inheritance.

**Diamond problem:**

* In multiple inheritance there is every chance of multiple properties of multiple objects with  the same name available to the sub class object with same priorities leads for the ambiguity.

//Multiple inheritance program

Class A{

}

Class B extends A{

public void show(){

}

}

Class C extends A{

public void show(){

}

}

Class D extends B,C{  // not supported by java leads to syntax error.

}

* We have two classes B and c which are inheriting A class properties.
* Here Class D inheriting B class and C class So properties present in those classes will be available in java.
* But both classes are in same level with same priority.
* If we want to use show() method that leads to ambiguity
* This is called diamond problem.
* Because of multiple inheritance there is chance of the root object getting created more than once.
* Always the root object i.e object of object class hast to be created only once.
* Because of above mentioned reasons multiple inheritance would not be supported by java.
* Thus in java a class can not extend more than one class simultaneously. At most a class can extend only one class.

So these are the reasons that java does not supports multiple inheritance.

The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritance. Java does not support multiple inheritance but the multiple inheritance can be achieved by using the interface.



In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class.

**21. What are the different types of interface? (Ans List, set, Queue)**

Ans)

There is no direct implementation for the java.util.Collection interface. The Collection interface has five sub interfaces.

|  |
| --- |
| **Figure 1:** The five sub interfaces of the java.util.Collection interface. |
| [Java collection interfaces.svg](https://commons.wikimedia.org/wiki/File:Java_collection_interfaces.svg) |

**Set**

A set collection contains unique elements, so duplicates are not allowed. It is similar to a mathematical Set. When adding a new item to a set, the set calls the method int hashCode() of the item and compares its result to the hash code of all the already inserted items. If the hash code is not found, the item is added. If the hash code is found, the set calls the boolean equals(Object obj); method for all the set items with the same hashcode as the new item. If all equal-calls return false, the new item is inserted in the set. If an equal-call returns true, the new item is not inserted in the set.

|  |
| --- |
| [Java collection set implementations.jpg](https://en.wikibooks.org/wiki/File:Java_collection_set_implementations.jpg) |

Set class diagram.

java.util.HashSet<E>

This is the basic implementation of the Set interface. Not synchronized. Allows the **null** elements

java.util.TreeSet<E>

Elements are sorted, not synchronized. **null** not allowed

java.util.CopyOnWriteArraySet<E>

Thread safe, a fresh copy is created during modification operation. Add, update, delete are expensive.

java.util.EnumSet<E extends Enum<E>>

All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. Enum sets are represented internally as bit vectors.

java.util.LinkedHashSet<E>

Same as HashSet, plus defines the iteration ordering, which is the order in which elements were inserted into the set.

**Detecting duplicate objects in Sets**

Set cannot have duplicates in it. You may wonder how duplicates are detected when we are adding an object to the Set. We have to see if that object exists in the Set or not. It is not enough to check the object references, the objects' values have to be checked as well.

To do that, fortunately, each java object has the **boolean equals(Object obj),** method available inherited from Object. You need to override it. That method will be called by the Set implementation to compare the two objects to see if they are equal or not.

There is a problem, though. What if I put two different type of objects to the Set. I put an Apple and an Orange. They cannot be compared. Calling the **equals()** method would cause a ClassCastException. There are two solutions to this:

* **Solution one** : Override the **int hashCode()** method and return the same values for the same type of objects and return different values for different type of objects. The **equals()** method is used to compare objects only with the same value of hashCode. So before an object is added, the Set implementation needs to:
  + find all the objects in the Set that have the same hashCode as the candidate object hashCode
  + and for those, call the equals() methods passing in the candidate object
  + if any of them returns true, the object is not added to the Set.
* **Solution two** : Create a super class for the Apple and Orange, let's call it Fruit class. Put Fruits in the Set. You need to do the following:
  + Do not override the equals() and hashCode() methods in the Apple and Orange classes
  + Create appleEquals() method in the Apple class, and create orangeEquals() method in the Orange class
  + Override the hashCode() method in the Fruit class and return the same value, so the equals() is called by the Set implementation
  + Override the equals() method in the Fruit class for something like this.

//equals method implementation:

public boolean equals(Object obj) {

boolean ret = false;

if (this instanceof Apple &&

obj instanceof Apple) {

ret = this.appleEquals(obj);

} else if (this instanceof Orange &&

obj instanceof Orange) {

ret = this.orangeEquals(obj);

} else {

// Can not compare Orange to Apple

ret = false;

}

return ret;

}

**Note:**

* Only the objects that have the same hashCode will be compared.
* You are responsible to override the equals() and hashCode() methods. The default implementations in Object won't work.
* Only override the hashCode() method if you want to eliminate value duplicates.
* Do not override the hashCode() method if you know that the values of your objects are different, or if you only want to prevent adding the exactly same object.
* Beware that the hashCode() may be used in other collection implementations, like in a Hashtable to find an object fast. Overriding the default hashCode() method may affect performance there.
* The default hashCodes are unique for each object created, so if you decide not to override the hashCode() method, there is no point overriding the equals() method, as it won't be called.

**SortedSet**

The SortedSet interface is the same as the Set interface plus the elements in the SortedSet are sorted. It extends the Set Interface. All elements in the SortedSet must implement the Comparable Interface, furthermore all elements must be mutually comparable.

Note that the ordering maintained by a sorted set must be consistent with equals if the sorted set is to correctly implement the Set interface. This is so because the Set interface is defined in terms of the equals operation, but a sorted set performs all element comparisons using its compare method, so two elements that are deemed equal by this method are, from the standpoint of the sorted set, equal.

The SortedSet interface has additional methods due to the sorted nature of the 'Set'. Those are:

|  |  |
| --- | --- |
| E first(); | returns the first element |
| E last(); | returns the last element |
| SortedSet headSet(E toElement); | returns from the first, to the exclusive toElement |
| SortedSet tailSet(E fromElement); | returns from the inclusive fromElement to the end |
| SortedSet subSet(E fromElement, E toElement); | returns elements range from fromElement, inclusive, to toElement, exclusive. (If fromElement and toElement are equal, the returned sorted set is empty.) |

**List**

In a list collection, the elements are put in a certain order, and can be accessed by an index. Duplicates are allowed, the same element can be added twice to a list. It has the following implementations:

List class diagram

|  |
| --- |
| [Java collection list implementations.jpg](https://en.wikibooks.org/wiki/File:Java_collection_list_implementations.jpg) |

java.util.Vector<E>

Synchronized, use in multiple thread access, otherwise use [ArrayList](https://en.wikibooks.org/wiki/Java_Programming/ArrayList).

java.util.Stack<E>

It extends class Vector with five operations that allow a vector to be treated as a stack. It represents a last-in-first-out (LIFO) stack of objects.

java.util.ArrayList<E>

The basic implementation of the List interface is the ArrayList. The ArrayList is not synchronized, not thread safe. Vector is synchronized, and thread safe. Vector is slower, because of the extra overhead to make it thread safe. When only one thread is accessing the list, use the ArrayList. Whenever you insert or remove an element from the list, there are extra overhead to reindex the list. When you have a large list, and you have lots of insert and remove, consider using the LinkedList.

java.util.LinkedList<E>

Non-synchronized, update operation is faster than other lists, easy to use for stacks, queues, double-ended queues. The name LinkedList implies a special data structure where the elements/nodes are connected by pointers.

Head Node 1 Node 2 Node n

\_\_\_\_\_\_

| Size | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_

|\_\_\_\_\_\_| | | point | | | point | | | |

| First |-------->| Data | to next |------>| Data | to next|-- ... -->| Data | null |

| elem | |\_\_\_\_\_\_|\_\_\_\_\_\_ | |\_\_\_\_\_\_ |\_\_\_\_\_\_ | |\_\_\_\_\_\_|\_\_\_\_\_\_|

|\_\_\_\_\_\_| ^

| Last | |

| elem |-----------------------------------------------------------------

|\_\_\_\_\_\_|

Each node is related to an item of the linked list. To remove an element from the linked list the pointers need to be rearranged. After removing Node 2:

Head Node 1 Node 2 Node n

\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| Size | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | \_\_\_\_\_\_\_\_\_\_\_\_\_\_

|\_- 1\_ \_| | | point | | | | point | | | | |

| First |-------->| Data | to next |---- | Data | to next | -...-->| Data | null |

| elem | |\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_|\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_|\_\_\_\_\_\_|

|\_\_\_\_\_\_| ^

| Last | |

| elem |----- ----- ----- --- --- ---- --- ---- ------ ---- --- ------- -- ----- -- -

|\_\_\_\_\_\_|

javax.management.AtributeList<E>

Represents a list of values for attributes of an MBean. The methods used for the insertion of Attribute objects in the AttributeList overrides the corresponding methods in the superclass ArrayList. This is needed in order to insure that the objects contained in the AttributeList are only Attribute objects.

javax.management.relation.RoleList<E>

A RoleList represents a list of roles (Role objects). It is used as parameter when creating a relation, and when trying to set several roles in a relation (via 'setRoles()' method). It is returned as part of a RoleResult, to provide roles successfully retrieved.

javax.management.relation.RoleUnresolvedList<E>

A RoleUnresolvedList represents a list of RoleUnresolved objects, representing roles not retrieved from a relation due to a problem encountered when trying to access (read or write to roles).

**Queue**

The Queue interface provides additional insertion, extraction, and inspection operations. There are FIFO (first in, first out) and LIFO (last in, first out) queues. This interface adds the following operations to the Collection interface:

|  |  |
| --- | --- |
| E element() | Retrieves, but does not remove, the head of this queue. This method differs from the peek method only in that it throws an exception if this queue is empty |
| boolean offer(E o) | Inserts the specified element into this queue, if possible. |
| E peek() | Retrieves, but does not remove, the head of this queue, returning null if this queue is empty |
| E poll() | Retrieves and removes the head of this queue, or null if this queue is empty |
| E remove() | Retrieves and removes the head of this queue. This method differs from the poll method in that it throws an exception if this queue is empty. |

|  |
| --- |
| [Java collection queue implementations.jpg](https://en.wikibooks.org/wiki/File:Java_collection_queue_implementations.jpg) |

Queue class diagram

java.util.BlockingQueue<E>

waits for the queue to become non-empty when retrieving an element, and waits for space to become available in the queue when storing an element. Best used for producer-consumer queues.

java.util.PriorityQueue<E>

orders elements according to an order/priority specified at construction time, null element is not allowed.

java.util.concurrent.ArrayBlockingQueue<E>

orders elements FIFO; synchronized, thread safe.

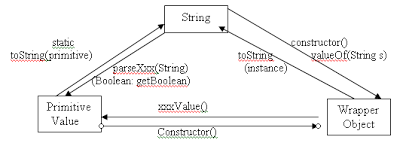
java.util.concurrent.SynchronousQueue<E>

each put must wait for a take, and vice versa, does not have any internal capacity, not even a capacity of one, an element is only present when you try to take it; you cannot add an element (using any method) unless another thread is trying to remove it.

**22. What are wrapper class? Give me an example**

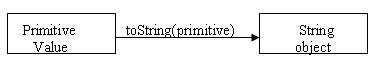
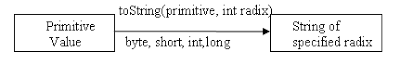
Ans)

The Wrapper classes have introduced for the following two purposes by SUN people.  
1. To wrap primitives into object form so that we can handle primitives also just like objects.  
2. We can get several utility functions for primitives.

• The following are the wrapper classes.  
**Type** --------- **Class**  
byte ---------> Byte  
short ---------> Short  
int ---------> Integer  
long ----------> Long  
float ----------> Float  
double ----------> Double  
char ----------> Character  
boolean ----------> Boolean  
  
[](http://2.bp.blogspot.com/_Pc8VSCmbaAo/SalJbYJRSCI/AAAAAAAAAIk/j4iFl_6Qe80/s1600-h/wrapper+class.png)  
  
**Constructors of the Wrapper classes :**  
• Integer class contains the following two constructors.  
Integer i= new Integer(int i);  
Integer i= new Integer(String s);  
**Ex:** Integer i= new Integer(10);  
Integer i= new Integer(“10”);  
Integer i=new Integer(“ten”); x given  
RTE: NumberFormatException  
I.e., If the String is unable to convert into a number then we will get a RTE saying “NumberFormantException”.  
---> Byte b=new Byte(byte b);  
---> Byte b=new Byte(String s);

**Wrapper Class** --------- **Constructor Arguments**  
1. Byte --------------------> byte or String  
2. Short --------------------> short or String  
3. Integer ---------------------> int or String  
4. Long ---------------------> long or String  
5. Float ---------------------> float or String or double  
Ex: float f= new Float(float f); //10.0f //valid  
float f= new Float(String s); //”10.0f” //valid  
float f= new Float(double d);//10.0 //valid  
6. Double ----------------------> double  
7. Character ------------------> char  
**Ex:** Character c=new Character(char c);  
Character c=new Character(‘d’); //valid  
Character c=new Character(“d”); //invalid  
8. Boolean ---------------------> Boolean or String  
**Ex:** Boolean b=new Boolean(true); //valid  
(false); //valid  
(TRUE); //invalid  
(FALSE); //invalid  
Boolean b=new Boolean(“viswan”); //false  
(“yes”); //false  
(“no”); //false  
Note: Other than “true” everything should return ‘false’.

**Which of the following are valid Boolean declarations?**  
1. boolean b=true; //valid  
2. boolean b=false; //valid  
3. boolean b=TRUE; //invalid  
4. boolean b=FALSE; //invalid  
5. Boolean b=new Boolean(“true”); //valid  
6. Boolean b=new Boolean(“false”); //valid  
7. Boolean b=new Boolean(“yes”); //valid //false  
8. Boolean b=new Boolean(“NO”); //valid // false  
9. boolean b=yes; //invalid //Compile time error  
10. boolean b=NO; //invalid // Compiletime error  
**valueOf() method:**  
  
[http://2.bp.blogspot.com/_Pc8VSCmbaAo/SZgXip1xceI/AAAAAAAAAFw/Zxru6itj-44/s400/valueOf%28string%29.png](http://2.bp.blogspot.com/_Pc8VSCmbaAo/SZgXip1xceI/AAAAAAAAAFw/Zxru6itj-44/s1600-h/valueOf(string).png)  
• Except character class every wrapper class contain a static valueOf() which can be used for converting a String object to corresponding wrapper object.  
**Ex:** Integer i=Integer.valueOf(“10”); //valid  
Integer i=Integer.valueOf(“ten”); //invalid // RTE :NumberFormatException  
Boolean b= Boolen.valueOf(“xxx”); //false  
**valueOf(String)**:  
String object ----------to------------> Wrapper Object  
• This is a static method.  
• The argument is String only.  
• Throws NumberFormatException, if we are unable to convert the String to number.  
**Ex:** Integer i= Integer.valueOf(10); //invalid  
Argument is not String ,the **Second version of valueOf() method:**  
• Byte. Short, Integer, Long classes contain second version of valueOf() method which can take a String and radix as arguments.  
• First the String converted into decimal form and then that decimal value will be stored in the wrapper object.  
**Ex1:** Integer i=Integer.valueOf(”101100”,2) ;  
System.out.println(i);//44 base=radix  
**Signatures:**  
1. public static wrapper valueOf(String s)  
object  
2. public static wrapper valueOf(String s, int radix)  
object  
**Ex2:** Long l=Long.valueOf(“12”,5);  
System.out.println(l); //7  
\* totally “36” bases or radixes are valid  
**xxxValue() methods:-**  
• All the numeric wrapper classes contains the fallowing methods for “converting wrapperclass object to primitive”.  
intVlaue(); byteVlaue(); shortVlaue(); longVlaue(); floatVlaue(); doubleVlaue();  
• Each wrapper class contains the above six methods, in total 36 xxxValue() methods are possible.  
• These xxxValue() methods are instance methods and no arg methods.  
Ex: Double d =new(150,263);  
System.out.println(d.byteValue()); // -106  
System.out.println(d.shortValue()); //150  
System.out.println(d.intValue()); //150  
System.out.println(d.longValue()); // 150  
System.out.println(d.floatValue()); // 150.263  
System.out.println(d.doubleValue()); // 150.263  
  
[http://1.bp.blogspot.com/_Pc8VSCmbaAo/SZgX6r-wceI/AAAAAAAAAF4/DJ0X5aKIxGk/s400/xxxValue%28%29.png](http://1.bp.blogspot.com/_Pc8VSCmbaAo/SZgX6r-wceI/AAAAAAAAAF4/DJ0X5aKIxGk/s1600-h/xxxValue().png)  
  
• Character class contains charValue() method for converting character object to primitive.  
**Ex:** Character ch= new Character(‘a’) ;  
Char c=ch.charValue();  
System.out.println(c); // a  
• Boolean class contain booleaValue() method for converting a Boolean object to Boolean primitive.  
**Ex:** Boolean B=new Boolen(“viswan”);  
boolean b=B.charValue();  
System.out.println(b); //false  
**parseXxx() methods:**  
  
[http://1.bp.blogspot.com/_Pc8VSCmbaAo/SZgYMqgwwfI/AAAAAAAAAGA/zxEpr_RxQrA/s400/parsexxx%28%29.png](http://1.bp.blogspot.com/_Pc8VSCmbaAo/SZgYMqgwwfI/AAAAAAAAAGA/zxEpr_RxQrA/s1600-h/parsexxx().png)  
• Every wrapper class except Character class contain parseXxx() for converting a String object to corresponding primitive.  
**Ex**: String s=”10”;  
int i=Integer.parseInt(s);  
double d=Double.parseDouble(s); //10.0  
long l=Long.parseLong(s);

**Wrapperclass**--------- **parseXxx()**  
Byte ------------> parseByte(String s)  
Short -------------> parseShort(String s)  
Integer -------------> parseInt(String s)  
Long -------------> parseLong(String s)  
Float -----------> parseFloat(String s)  
Double ------------> parseDouble(String s)  
• Boolean class contain a static method getBoolean() method for converting a string to Boolean primitive.. -----> 1.4 version  
String s=”123”;  
int i=Integer.parseInt(s);  
double d=Double.parseDouble(s);  
System.out.println(i); //123  
System.out.println(d); //123.0  
Boolean b=getBoolean(“viswan”);  
System.out.println(b); //false  
**Second version of parseXxx() :-**  
• All the integral wrapper classes (byte, short, integer, long) contains second version of parseXxx().  
public static primitive parseXxx(String s,int radix)  
**Ex:** int i=Integer.parseInt(“10101100”,2);  
System.out.println(i); //172  
long l=Long.parseLong(“12”,8);  
System.out.println(l); //10  
• java support max – radix is 36(base)  
System.out.println(Character.MAX-RADIX); //036  
**toString() method:**  
**1st version:**-  
• All the wrapper classes contains an instance toString() method for converting wrapper class object to corresponding String object.  
• public String toString();  
Integer i=new Integer(10);  
String s= i.toString();  
System.out.println(s); // 10  
Boolen B=new Boolean(“surya”);  
String s=B.toString();  
System.out.println(s);//false  
  
[http://2.bp.blogspot.com/_Pc8VSCmbaAo/SZgYi74cPoI/AAAAAAAAAGQ/xw0ayVExako/s400/tostring%28%29.png](http://2.bp.blogspot.com/_Pc8VSCmbaAo/SZgYi74cPoI/AAAAAAAAAGQ/xw0ayVExako/s1600-h/tostring().png)  
**2nd Version:**  
  
[](http://3.bp.blogspot.com/_Pc8VSCmbaAo/SZgYxJFEiVI/AAAAAAAAAGY/CJAp90eqLPk/s1600-h/tostring2.png)  
• Every wrapper class contain a static toString() method for converting a primitive to String object.  
• This is available in all wrapper classes and object class also includes Boolean and Character classes.  
• public static String toString(10);  
**Ex:** String s1=Integer.toString(10);  
System.out.println(s);  
String s1=Boolean.toString(true);  
String s1=Integer.toString(‘a’);  
System.out.println(s1);  
System.out.println(s2);  
**3rd Version:**  
  
[](http://4.bp.blogspot.com/_Pc8VSCmbaAo/SZgYxIP9S8I/AAAAAAAAAGg/ecjKROJnMzA/s1600-h/tostring3.png)  
• Integer and Long classes contain the 3rd version of the toString() method for converting given primitive to String of sprcified radix.  
• public satic String toString(30,2);  
Ex: String s= Integer.toString(30,2);  
System.out.println(s); //”11110”  
toString(primitive, int radix)  
**4th version:**  
• public static String to xxxString(primitive) //int/long  
• The version of toString() is available in the Integer and Long classes only. The possible to xxxString() methods are, toBinaryString, toHexString and tooctalString for converting the given primitive to the corresponding String form.  
String s=Integr.toBinaryString(100); //1100100  
String s=Integr.toOctalString(100); //144  
String s=Integr.toHexString(100); //64  
• All the wrapper class objects are immutable and all the wrapper classes are final classes.  
• ‘void’ is also one type of wrapper class.  
• String class and all the wrapper classes are immutable.  
• The following are the final classes.  
-->String  
--> StrringBuffer  
--->Math

**23. What is boxing and unboxing in Java? Explain with an example**

Ans)

**Autoboxing and Unboxing:**

* Autoboxing and Unboxing features was added in Java5.
* **Autoboxing** is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper
* **Auto-Unboxing** is a process by which the value of object is automatically extracted from a type wrapper.

**Example of Autoboxing and Unboxing:**

class Test

{

public static void main(String[] args)

{

Integer iob = 100; //Autoboxing of int

int i = iob; //unboxing of Integer

System.out.println(i+" "+iob);

Character cob = 'a'; /Autoboxing of char

char ch = cob; //Auto-unboxing of Character

System.out.println(cob+" "+ch);

}

}

**Output :**

100 100

a a

**Autoboxing / Unboxing in Expressions:**

Whenever we use object of Wrapper class in an expression, automatic unboxing and boxing is done by JVM.

Integer iOb;

iOb = 100; //Autoboxing of int

**++iOb**;

When we perform increment operation on Integer object, it is first unboxed, then incremented and then again reboxed into Integer type object.

This will happen always, when we will use Wrapper class objects in expressions or conditions etc.

**Benefits of Autoboxing / Unboxing**

1. Autoboxing / Unboxing lets us use primitive types and Wrapper class objects interchangeably.
2. We don't have to perform Explicit **typecasting**.
3. It helps prevent errors, but may lead to unexpected results sometimes. Hence must be used with care.

**24. Explain for each loop**

Ans)

**For-each loop (Advanced or Enhanced For loop):**

The for-each loop introduced in Java5. It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.

**Advantage of for-each loop:**

* It makes the code more readable.
* It eliminates the possibility of programming errors.

**Syntax of for-each loop:**

1. for(data\_type variable : array | collection){}

### Simple Example of for-each loop for traversing the array elements:

class ForEachExample1{

  public static void main(String args[]){

   int arr[]={12,13,14,44};

   for(int i:arr){

     System.out.println(i);

   }

 }

}

Output:12

13

14

44

### Simple Example of for-each loop for traversing the collection elements:

import java.util.\*;

class ForEachExample2{

  public static void main(String args[]){

   ArrayList<String> list=new ArrayList<String>();

   list.add("vimal");

   list.add("sonoo");

   list.add("ratan");

   for(String s:list){

     System.out.println(s);

   }

 }

### }

Output:

vimal

sonoo

ratan

**25. What are iterators, explain with an example**

Ans)

In Java, an i**terator** is an interface and is implemented by all collection classes. The Java collections framework is a group of classes and interfaces that implement reusable collection of data structures. The iterator method returns an object that implements the Iterator interface. An object of an iterator interface can be used to traverse through all elements of a collection. Elements can also be modified and removed from the collection while traversal.

**Characteristics of a Java Iterator:**

* It works similar to enumeration.
* The **iterator** interface is in the Java Collections Framework.
* The Iterator object traverses through all elements of the collection.
* All collection framework Interfaces have the iterator() method.
* Its main difference from other iterators (such as C++ iterator) is that is does not require incrementing an array index to traverse through the elements. It uses the next() method to look up an element and the loop automatically advances after that.

**How to Use Java Iterator in Your Java Code?**

An **iterator** must first be obtained before it can be used for traversing. Each class in the collections framework includes an iterator() method that returns an object of the **iterator** interface. This method can be invoked to obtain an iterator object before traversing the collection. The object can provide you access to each element of the collection. Using an iterator object typically requires going through the following cycle:

* Call the collection’s iterator() method
* Create a loop that calls the hasNext() — which returns a boolean variable. If the returned variable is true, the iterator object accesses the collection; otherwise the looping stops.
* Call the next() method to obtain each item of the collection.

**Important Methods Used by the Java Iterator:**

1) boolean hasNext(): this method returns true when there is another element to be traversed in the collection.

2) Object next(): It provides the next object to be traversed. This method throws an exception if there are no more elements to be visited in the collection.

3) Object remove(): This method removes and returns the last object that was visited. If the objects in the collection have been modified since the last visited object, an **IllegalStateException** is thrown by the method.

**Using the Java Iterator remove() Method**

The remove() when called, eliminates the element that was returned by the call to next(). The remove method was added to java iterator because the iterator knows the exact position of all the elements in the collection. It is a lot easier to remove an object from the collection if you know where it is.

**The following example code removes the first element in a collection:**

ArrayList  abc = new ArrayList(); // create an arraylist abc

Iterator itr = abc.iterator();

itr.next(); // skip over the first element

itr.remove; // remove the first element

**How to use the java iterator hasNext() and next() methods?**

Every element in the collection can be visited one by one by calling next(). The method throws an exception when there are no more elements in the collection. In order to avoid the exception, you should call  hasnext() before next(); this is to make sure the iterator advances only if the next element/object exist in the collection.

**Important points about Java Iterators:**

**1)** Basically List and set collection provides the iterator. You can get Iterator from ArrayList, LinkedList and TreeSet etc. Map implementation such as HashMap doesn’t provide Iterator directory but you can get there keySet or Value Set and can iterator through that collection.

2) Although traversal actions can be achieved using the common looping structures such as a **for** or **while** loop, using an iterator has a few advantages

* It will allow you to scan forward through any collection.
* If you remove elements from an array using a traditional looping structure, you will encounter an **IndexOutofBoundsException**. This is because the size of the array changes as you iterate through it and you may have set the size of the array before the iteration. That problem does not exist with iterators because the hasnext() and next() does not rely on a pre-defined index.

3) For collections that implement the **List** interface, you can also obtain an iterator by calling the ListIterator. The list iterator allows you to access the objects in the collection in either forwards or backward direction; it also lets you modify an element.

4) Iterator in Java support generics so always use Generic version of Iterator rather than using Iterator with raw type.

5) Iterating over collection using Iterator is subject to ConcurrentModificationException if Collection is modified after Iteration started, but this only happens in case of fail-fast Iterators.

6) There are two types of Iterators in Java, fail-fast and fail-safe.

7) The list of elements can be displayed in reverse order using the hasPrevious() and previous() of the ListIterator interface.

**26. How do you access Private variables in different class?**

Ans)

For example,

In Class1, there are two private variables one is String type and another is int type. Since these variables are private so we can’t access these directly into another class. So we will create getter and setter method of these variable

s. In Class2, create object of Class1 as Class1 class1 = new Class1(); Now you can use its methods and variable through by calling Class1 getter methods. If there is public variable in Class1 then you can access them directly by Class1 object as class1.name.

public class Class1 {                       //Class1  
    private String name = "Class1";  
    private int IdNo = 20;  
  
    public void class1Method() {  
        System.out.println("Welcome in " + name + " Its id number : " + IdNo);  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getIdNo() {  
        return IdNo;  
    }  
  
    public void setIdNo(int idNo) {  
        IdNo = idNo;  
    }  
  
}

//Another class Class2.  
  
public class Class2 {     
    public static void main(String[] args) {  
        Class1 class1 = new Class1(); // Creating object of Class1  
        String name = "Class2";  
        System.out.println("Welcome in " + name);  
        System.out.println("Another Class variables value:" + class1.getIdNo()  
                + " and " + class1.getName());  
           class1.class1Method();  
    }  
}

**27. Prepare for one java program to write on the board**

Ans)

**Java program to reverse a number** :

**import** java.util.Scanner;

**public** **class** ReverseNumberExample {  
  
    **public** **static** **void** main(**String** args[]) {  
       *//input number to reverse*  
        **System**.out.println("Please enter number to be reversed using Java program: ");  
        **int** number = **new Scanner**(**System**.in).nextInt();

**int** reverse = reverse(number);  
        **System**.out.println("Reverse of number: " + number + " is " + reverse(number));

}

**public** **static** **int** reverse(**int** number){  
        **int** reverse = 0;  
        **int** remainder = 0;

do{  
            remainder = number%10;  
            reverse = reverse\*10 + remainder;  
            number = number/10;  
            
        }while(number > 0);

**return** reverse;  
    }  
  
}

Output:  
Please enter number to be reversed using Java program:  
1234  
Reverse of number: 1234 is 4321

**28. What is Constructor Over loading?**

Ans)

Constructors is a special method in Java which is used to initialize the instance variables of the class.A constructor has the same name of the class and does not have any return type(not even void) .A constructor may or may not have parameters.

Whenever a new object is created JVM first allocates memory for the object and then invokes the matching constructor.If class does not have any constructor JVM provides default constructor with no arguments.

Constructors in Java can be overloaded. Two ore more constructors with difference in the parameters is called constructor overloading. Each constructor is used to perform different task.

Compiler differentiates which constructor is to be called depending upon the number of parameters and their sequence of data types.

Example:

public class Perimeter

{

  public Perimeter()                                                     // I

  {

    System.out.println("From default");

  }

  public Perimeter(int x)                                                // II

  {

    System.out.println("Circle perimeter: " + 2\*Math.PI\*x);

  }

  public Perimeter(int x, int y)                                         // III

  {

    System.out.println("Rectangle perimeter: " +2\*(x+y));

  }

  public static void main(String args[])

  {

    Perimeter p1 = new Perimeter();                     // I

    Perimeter p2 = new Perimeter(10);                  // II

    Perimeter p3 = new Perimeter(10, 20);             // III

  }

}

Perimeter constructor is overloaded three times. As per the parameters, the appropriate constructor is called. To call all the three constructors three objects are created. Using **this()**, all the three constructors can be called with a single constructor.

**this() with Constructors**

Suppose by accessing one constructor, the programmer may require the functionality of other constructors also but by creating one object only. For this, Java comes with this(). "**this()**" is used to access one constructor from another "within the same class". Depending on the parameters supplied, the suitable constructor is accessed.

public class Perimeter

{

  public Perimeter()                                                      // I

  {

    System.out.println("From default");

  }

  public Perimeter(int x)                                                 // II

  {

    this();

    System.out.println("Circle perimeter: " + 2\*Math.PI\*x);

  }

  public Perimeter(int x, int y)                                          // III

  {

    this(100);

    System.out.println("Rectangle perimeter: " +2\*(x+y));

  }

  public static void main(String args[])

  {

    Perimeter p3 = new Perimeter(10, 20);                                 // III

  }

}

In the code, creating object **p3**, the III constructor is accessed. From III, with "this(100)" statement, the II constructor is accessed. Again from II, the I is accessed without the statement "this()". As per the parameter supplied to this(), the appropriate or corresponding constructor is accessed.

**Rules of using this()**

A few restrictions exist for the usage of this().

1. If included, this() statement must be the first one in the constructor. You cannot write anything before this() in the constructor.
2. With the above rule, there cannot be two this() statements in the same constructor (because both cannot be the first).
3. this() must be used with constructors only, that too to call the same class constructor (but not super class constructor).

29. Without using sync key word how do you perform synchronization?

Ans)

1. No need for synchronization at all if you don't have mutable state.
2. No need for synchronization if the mutable state is confined to a single thread. This can be done by using local variables or java.lang.ThreadLocal
3. You can also use built-in synchronizers. Java.util.concurrent.locks.ReentrantLock has the same functionality as the lock you access when using synchronized blocks and methods, and it is even more powerful.

30. What is Super keyword ? when and where do you use it ?

Ans)

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

**Usage of java super Keyword**

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

Calling *exactly* super() is **always** redundant. It's explicitly doing what would be implicitly done otherwise. That's because if you omit a call to the super constructor, the no-argument super constructor will be invoked automatically anyway.

However, where it becomes useful is when the super constructor takes arguments that you want to pass in from the subclass.

**What if a class don't extends another class, still uses 'super' in his constructor.**

If a class dont extends another class and still uses a super() call, this is perfectly ok. No error will be shown because every independent class extends 'Object' class by default. And in this case a dafult constructor of 'Object' class is being called.

**What if there is a chain of extended classes and 'super' keyword is used**

Now the question comes, what if there is a chain of extending classes and a subclass in the end of hierarchy calls a 'super()'. Yes, this is perfectly true in this case the default constructors of all classes in the hierarchy will be called implicitly.

**Note:**  
1) super() must be the first statement in constructor otherwise we will get the compilation error message: “Constructor call must be the first statement in a constructor”

2) We can also invoke parameterized constructor of parent class by providing arguments while calling super. For e.g. super(10) would invoke the parametrized constructor [having one integer argument] of parent class. Similarly super(“hi”) would invoke constructor having String argument.

Programing Questions:

**1. Find out the number of days in between two given dates?**

Sol)

**package** javaTest1;

**import** java.util.Calendar;

**import** java.util.Date;

**import** java.util.GregorianCalendar;

**public** **class** DateDifference {

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

DateDifference difference = **new** DateDifference();

}

DateDifference() {

Calendar cal1 = **new** GregorianCalendar();

Calendar cal2 = **new** GregorianCalendar();

cal1.set(2009, 9, 1);

cal2.set(2011, 9, 31);

System.***out***.println("Days= "+daysBetween(cal1.getTime(),cal2.getTime()));

}

**public** **int** daysBetween(Date d1, Date d2){

**return** (**int**)( (d2.getTime() - d1.getTime()) / (1000 \* 60 \* 60 \* 24));

}

}

**Output:** Days= 760

**2. How to divide a number by 2 without using / operator?**

Ans)

**public** **class** DivisionWithoutUsingDivisionOperator {

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

**int** num = 16;

**int** dv = *devideByTwo* (num);

System.***out***.println ("Result of " + num + "/2 = " + dv);

}

**public** **static** **int** devideByTwo (**int** num) {

**return** (num >> 1);

}

}

Output:

Result of 16/2 = 8

**3. How to multiply a number by 2 without using \* operator?**

Ans)

**public** **class** MultiplyWithoutUsingMultiplyOperator {

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

**int** num = 12;

**int** mul = *multiplayByTwo* (num);

System.***out***.println ("Result of " + num + "\*2 = " + mul);

}

**public** **static** **int** multiplayByTwo (**int** num) {

**return** (num << 1);

}

}

**4.How to swap two variables, by using pass by reference method ?**

Sol)

**public** **class** SwapUsingPassByReference {

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

Animal a = **new** Animal("Lion");

Animal b = **new** Animal("Tiger");

System.***out***.println("Before Swap: " + a);

System.***out***.println("Before Swap: " + b);

*swapNames*(a,b);

System.***out***.println("After Swap: " + a);

System.***out***.println("After Swap: " + b);

}

**public** **static** **void** swapNames(Animal animal1,Animal animal2) {

String firstname = animal1.getName();

String secondName = animal2.getName();

animal1.setName(secondName);

animal2.setName(firstname);

}

}

**class** Animal {

String name;

**public** Animal(String name) {

**this**.name = name;

}

**public** String toString() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getName(){

**return** **this**.name;

}

}

**Output:**

Before Swap: Lion

Before Swap: Tiger

After Swap: Tiger

After Swap: Lion

**6. Write a sample code to reverse Singly Linked List by iterating through it only once.**

Sol)

**public** **class** ReverseSinglyLinkedList<T> {

**private** Node<T> head;

**public** **void** add(T element){

Node<T> nd = **new** Node<T>();

nd.setValue(element);

System.***out***.println("Adding: "+element);

Node<T> tmp = head;

**while**(**true**){

**if**(tmp == **null**){

//since there is only one element, both head and

//tail points to the same object.

head = nd;

**break**;

} **else** **if**(tmp.getNextRef() == **null**){

tmp.setNextRef(nd);

**break**;

} **else** {

tmp = tmp.getNextRef();

}

}

}

**public** **void** traverse(){

Node<T> tmp = head;

**while**(**true**){

**if**(tmp == **null**){

**break**;

}

System.***out***.print(tmp.getValue()+"\t");

tmp = tmp.getNextRef();

}

}

**public** **void** reverse(){

System.***out***.println("\nreversing the linked list\n");

Node<T> prev = **null**;

Node<T> current = head;

Node<T> next = **null**;

**while**(current != **null**){

next = current.getNextRef();

current.setNextRef(prev);

prev = current;

current = next;

}

head = prev;

}

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

ReverseSinglyLinkedList<Integer> sl = **new** ReverseSinglyLinkedList<Integer>();

sl.add(3);

sl.add(32);

sl.add(54);

sl.add(89);

System.***out***.println();

sl.traverse();

System.***out***.println();

sl.reverse();

sl.traverse();

}

}

**class** Node<T> **implements** Comparable<T> {

**private** T value;

**private** Node<T> nextRef;

**public** T getValue() {

**return** value;

}

**public** **void** setValue(T value) {

**this**.value = value;

}

**public** Node<T> getNextRef() {

**return** nextRef;

}

**public** **void** setNextRef(Node<T> ref) {

**this**.nextRef = ref;

}

@Override

**public** **int** compareTo(T arg) {

**if**(arg == **this**.value){

**return** 0;

} **else** {

**return** 1;

}

}

}

Output:

Adding: 3

Adding: 32

Adding: 54

Adding: 89

3 32 54 89

reversing the linked list

89 54 32 3

**7. Write a program to implement ArrayList and Linked list**

Sol)

import java.util.ArrayList;

import java.util.Collection;

import java.util.LinkedList;

import java.util.List;

public class ArrayListLinkedList {

public static void main(String[] args) {

// create a LinkedList

LinkedList list = new LinkedList();

// add some elements

list.add("Hello");

list.add(2);

list.add("Chocolate");

list.add("10");

// print the list

System.out.println("LinkedList:" + list);

// create a new collection and add some elements

Collection collection = new ArrayList();

collection.add("One");

collection.add("Two");

collection.add("Three");

// add the collection in the LinkedList at index 2

list.addAll(2, collection);

// print the new list

System.out.println("LinkedList:" + list);

}

}

**Output:**

LinkedList:[Hello, 2, Chocolate, 10]

LinkedList:[Hello, 2, One, Two, Three, Chocolate, 10]

**8. Write a program for Insertion Sort in java.**

Sol)

**package** javaTest1;

**public** **class** InsertionSort {

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

**int** i;

**int** array[] = {12,9,4,99,120,1,3,10};

System.***out***.println(" Selection Sort\n\n");

System.***out***.println("Values Before the sort:\n");

**for**(i = 0; i < array.length; i++)

System.***out***.print( array[i]+" ");

System.***out***.println();

*insertion\_srt*(array, array.length);

System.***out***.print("Values after the sort:\n");

**for**(i = 0; i <array.length; i++)

System.***out***.print(array[i]+" ");

System.***out***.println();

}

**public** **static** **void** insertion\_srt(**int** array[], **int** n){

**for** (**int** i = 1; i < n; i++){

**int** j = i;

**int** B = array[i];

**while** ((j > 0) && (array[j-1] > B)){

array[j] = array[j-1];

j--;

}

array[j] = B;

}

}

}

**Output:**

Selection Sort

Values Before the sort:

12 9 4 99 120 1 3 10

Values after the sort:

1 3 4 9 10 12 99 120

**9. Write a program to get distinct word list from the given file.**

Sol)

**import** java.io.BufferedReader;

**import** java.io.DataInputStream;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.StringTokenizer;

**public** **class** DistinctWordListFromGivenFile {

**public** List<String> getDistinctWordList(String fileName){

FileInputStream fis = **null**;

DataInputStream dis = **null**;

BufferedReader br = **null**;

List<String> wordList = **new** ArrayList<String>();

**try** {

fis = **new** FileInputStream(fileName);

dis = **new** DataInputStream(fis);

br = **new** BufferedReader(**new** InputStreamReader(dis));

String line = **null**;

**while**((line = br.readLine()) != **null**){

StringTokenizer st = **new** StringTokenizer(line, " ,.;:\"");

**while**(st.hasMoreTokens()){

String tmp = st.nextToken().toLowerCase();

**if**(!wordList.contains(tmp)){

wordList.add(tmp);

}

}

}

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

} **finally**{

**try**{**if**(br != **null**) br.close();}**catch**(Exception ex){}

}

**return** wordList;

}

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

DistinctWordListFromGivenFile distFw = **new** DistinctWordListFromGivenFile();

List<String> wordList = distFw.getDistinctWordList("C:/Users/smeda/Desktop/Testing/javafilewordlist.txt");

**for**(String str:wordList){

System.***out***.println(str);

}

}

}

**Output:**

the

synchronized

keyword

may

be

applied

to

statement

block

or

a

method

**10. Find longest substring without repeating characters.**

Sol)

import java.util.HashSet;

import java.util.Scanner;

import java.util.Set;

public class LongestSubstringWithoutRepeatingCharacters {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Please enter the longest Substring");

String s = sc.nextLine();

System.out.println(lenLongestSubstr(s));

}

public static int lenLongestSubstr(String s)

{

Set<Character> charSet = new HashSet<Character>();

int j = 0, i = 0, maxLength = 0;

while (j < s.length()) {

charSet.add(s.charAt(j));

i = j + charSet.size();

while (i < s.length() && !charSet.contains(s.charAt(i))) {

charSet.add(s.charAt(i));

i++;

}

maxLength = Math.max(charSet.size(), maxLength);

if (i == s.length())

break;

while (s.charAt(j) != s.charAt(i)) {

charSet.remove(s.charAt(j));

j++;

}

j++;

}

return maxLength;

}

}

**Output:**

Please enter the longest Substring

NiagaraFalls

4

**11. Write a program to remove duplicates from sorted array**

Sol)

**public** **class** RemoveDuplicatesFromSortedArray {

**public** **static** **int**[] removeDuplicates(**int**[] input){

**int** j = 0;

**int** i = 1;

//return if the array length is less than 2

**if**(input.length < 2){

**return** input;

}

**while**(i < input.length){

**if**(input[i] == input[j]){

i++;

}**else**{

input[++j] = input[i++];

}

}

**int**[] output = **new** **int**[j+1];

**for**(**int** k=0; k<output.length; k++){

output[k] = input[k];

}

**return** output;

}

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

**int**[] input1 = {3,4,4,5,5,5,5,6,7,7,8,8,8,9};

**int**[] output = *removeDuplicates*(input1);

**for**(**int** i:output){

System.***out***.print(i+" ");

}

}

}

**Output:**

3 4 5 6 7 8 9

**12. Write a program to print fibonacci series.**

Sol)

**package** javaTest1;

**import** java.util.Scanner;

**public** **class** RecursiveFibonacciSeries {

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

Scanner s = **new** Scanner(System.***in***);

System.***out***.print("Enter the value of n: ");

**int** n = s.nextInt();

**for** (**int** i = 0; i <= n; i++) {

System.***out***.print(fibonacci(i) + " ");

}

}

**public** **static** **int fibonacci**(**int** n) {

**if** (n == 0) {

**return** 0;

} **else** **if** (n == 1) {

**return** 1;

} **else** {

**return fibonacci**(n - 1) + fibonacci(n - 2);

}

}

}

**Output:**

Enter the value of n: 9

0 1 1 2 3 5 8 13 21 34

**13. Write a program to find out duplicate characters in a string**

Sol)

**import** java.util.HashMap;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** DuplicateCharactersInString{

**public** **void** findDuplicateChars(String str){

Map<Character, Integer> dupMap = **new** HashMap<Character, Integer>();

**char**[] chrs = str.toCharArray();

**for**(Character ch:chrs){

**if**(dupMap.containsKey(ch)){

dupMap.put(ch, dupMap.get(ch)+1);

} **else** {

dupMap.put(ch, 1);

}

}

Set<Character> keys = dupMap.keySet();

**for**(Character ch:keys){

**if**(dupMap.get(ch) > 1){

System.***out***.println(ch+"--->"+dupMap.get(ch));

}

}

}

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

DuplicateCharactersInString dcs = **new** DuplicateCharactersInString();

dcs.findDuplicateChars("Iwanttolearnjava");

}

}

**Output:**

a--->4

t--->2

n--->2

**14. Write a program to create deadlock between two threads**

Sol)

**public** **class** DeadlockBetweenTwoThreads {

**public** **static** Object *Lock1* = **new** Object();

**public** **static** Object *Lock2* = **new** Object();

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

Thread1 T1 = **new** Thread1();

Thread2 T2 = **new** Thread2();

T1.start();

T2.start();

}

**private** **static** **class** Thread1 **extends** Thread {

**public** **void** run() {

**synchronized** (*Lock1*) {

System.***out***.println("Thread 1: Holding lock 1...");

**try** { Thread.*sleep*(10); }

**catch** (InterruptedException e) {}

System.***out***.println("Thread 1: Waiting for lock 2...");

**synchronized** (*Lock2*) {

System.***out***.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

**private** **static** **class** Thread2 **extends** Thread {

**public** **void** run() {

**synchronized** (*Lock2*) {

System.***out***.println("Thread 2: Holding lock 2...");

**try** { Thread.*sleep*(10); }

**catch** (InterruptedException e) {}

System.***out***.println("Thread 2: Waiting for lock 1...");

**synchronized** (*Lock1*) {

System.***out***.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

}

**Output:**

Thread 2: Holding lock 2...

Thread 1: Holding lock 1...

Thread 2: Waiting for lock 1...

Thread 1: Waiting for lock 2...

**15. Find out middle index where sum of both ends are equal**

**Sol)**

**public** **class** FindMiddleIndexWhereSumOfBothEndsEqual {

**public** **static** **int** findMiddleIndex(**int**[] numbers) **throws** Exception {

**int** endIndex = numbers.length - 1;

**int** startIndex = 0;

**int** sumLeft = 0;

**int** sumRight = 0;

**while** (**true**) {

**if** (sumLeft > sumRight) {

sumRight += numbers[endIndex--];

} **else** {

sumLeft += numbers[startIndex++];

}

**if** (startIndex > endIndex) {

**if** (sumLeft == sumRight) {

**break**;

} **else** {

**throw** **new** Exception("Please pass proper array to match the requirement");

}

}

}

**return** endIndex;

}

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

**int**[] num = { 2, 4, 4, 5, 4, 1 };

**try** {

System.***out***.println("Starting from index 0, adding numbers till index "

+ *findMiddleIndex*(num) + " and");

System.***out***.println("adding rest of the numbers can be equal");

} **catch** (Exception ex) {

System.***out***.println(ex.getMessage());

}

}

}

Output:

Starting from index 0, adding numbers till index 2 and

adding rest of the numbers can be equal

**16. Write a program to find the given number is Armstrong number or not?**

Ans)

**import** java.util.Scanner;

**public** **class** ArmstrongNumber {

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

Scanner s = **new** Scanner(System.***in***);

System.***out***

.println("Please enter any number to check whether it is Armstrong or not: ");

**int** n = s.nextInt();

**int** r, sum = 0, temp = n;

**while** (n > 0) {

r = n % 10;

n = n / 10;

sum = sum + (r \* r \* r);

}

**if** (sum == temp){

System.***out***.println("Given number " + temp + " is Armstrong.");

}

**else**{

System.***out***.println("Given number " + temp + " is not Armstrong.");

}

}

}

**Output:**

Please enter any number to check whether it is Armstrong or not:

34

Given number 34 is not Armstrong.