Basics of JAVA

1. Rules for Java Class
2. A class can have only public or default (no modifier) access specifier.
3. It can be either abstract, final or concrete (normal class).
4. It must have the class keyword and class must be followed by a legal identifier.
5. It may optionally extend one parent class. By default, it will extend java.lang.Object.
6. It may optionally implement any number of comma-separated interfaces.
7. Each **.java** source file may contain only one public class. A source file may contain any number of default visible classes and source file name must match with public class name.
8. There are two ways to pass an argument to a method
9. **call-by-value:** In this approach, copy of an argument value is pass to a method. Changes made to the argument value inside the method will have no effect on the arguments.

public class Test

{

public void callByValue(int x)

{

x=100;

}

public static void main(String[] args)

{

int x=50;

Test t = new Test();

t.callByValue(x); //function call

System.out.println(x);

}

}

Output: 50

1. **call-by-reference:** In this reference of an argument is pass to a method. Any changes made inside the method will affect the argument value.

public class Test

{

int x=10;

int y=20;

public void callByReference(Test t)

{

t.x=100;

t.y=50;

}

public static void main(String[] args)

{

Test ts = new Test();

System.out.println("Before "+ts.x+" "+ts.y);

ts.callByReference(ts);

System.out.println("After "+ts.x+" "+ts.y);

}

}

**Output:** Before 10 20

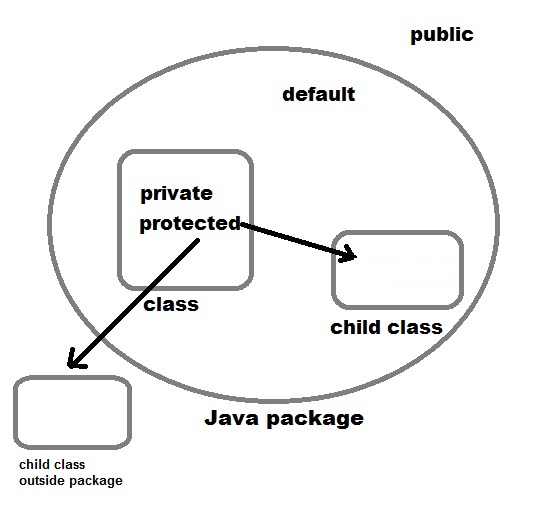
**After 100 50**

1. **In java, when you pass a primitive type to a method, it is passed by value and when you pass an object of any type to a method it is passed as reference.**
2. Two or more method have same name and same parameter list **but differ in return type** are not said to be overloaded methods.
3. There are 2 ways of method overloading
4. Method overloading by changing **data** **type** of Arguments
5. Method overloading by changing **number** of Arguments
6. When an overloaded method is called java look for match between the arguments to call the method and the method's parameters. This match need not always be exact, sometime when exact match is not found, Java automatic type conversion plays a vital role.

Type promotion- byte||char🡪short🡪int🡪long🡪double

1. Constructor:-
2. A constructor does not have any return type. Yes, constructors return current instant of a class. But yet constructor signature cannot have any return type.
3. Constructor in Java cannot be abstract, static, final or synchronized.
4. Constructors must have the same name as the class and cannot return a value. They are only called once while regular methods could be called many times and it can return a value or can be void
5. Constructor chaining is a phenomena of calling one constructor from another constructor of same class
6. This keyword:-
7. **this** keyword is used to refer to current object.
8. **this** is always a reference to the object on which method was invoked.
9. **this** can be used to invoke current class constructor.
10. **this** can be passed as an argument to another method.
11. Garbage Collection:-
12. In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called **Garbage Collection**. This is accomplished by the JVM
13. Garbage collection cannot be forced explicitly. We may request JVM for garbage collection by calling **System.gc()** method. But this not guarantee that JVM will perform the garbage collection. gc() method is present in **System** and **Runtime** class.
14. Advantages: Programmer doesn’t need to worry about dereferencing an object. Increases memory efficiency and decreases the chances for memory leak.
15. finalize() method is available to all classes because it is defined in java.lang.Object class. It is declared as **protected.** It is called only once by GC threads
16. Modifiers are the keywords that are added to change meaning of a definition.
17. Access control modifier – Java language has 4 access modifier to control access levels for classes, variable methods and constructor

* Default: Default has scope only inside the same package
* Public : Public scope is visible everywhere
* Protected: Protected has scope within the package and all sub classes
* Private: Private has scope only within the classes



1. Non-access Modifier – Non-access modifiers do not change the accessibility of variables and methods, but they do provide them special properties. These are of 5 types.

* Final- final modifier is used to declare a field as **final** i.e. it prevents its content from being modified. Final field must be initialized when it is declared/

1. A class declared as final cannot be inherited. Ex – String class is final
2. Method declared as final can be inherited but cannot be overridden.

* Static- static modifier is used to create class variable and class methods which can be accessed without instance of a class. Static variable are initialized only once. And it has only one single storage.

**Usage of Static:-** Static variable are used to represent common property of a class. It saves memory. Suppose there are 100 employee in a company. All employee have its unique name and employee id but company name will be same all 100 employee. Here company name is the common property. So if you create a class to store employee detail, company name field will be mark as static.

* Transient
* Synchronized
* Volatile

String Handling

1. String, Integer, Byte, Short, Float, Double and all other wrapper classes’ objects are immutable (whose state cannot be changed). So, lot of changes with String objects with end up with a lot of memory leak.
2. Creating an Immutable Class

public final class MyString{

final String str;

MyString(String s){

this.str=s;

}

public String get(){return str;}

}

In above example, MyString class is an immutable class. MyString’s state cannot be changed once it is created.

1. String comparison can be done in 3 ways.
2. Using **equals()** method—equals() method compares two strings for equality. It compares **the content** of the strings
3. Using == operator 🡪 == operator compares two object **references** to check whether they refer to same instance
4. By compareTo() method—compareTo() method compares values and returns an int which tells if the string compared is less than, equal to or greater than the other string.

int compareTo(String str)

to use this function, class must implements the **Comparable Interface**

1. **String class Methods: -** chatAt(int index), equalsIgnoreCase(), length(), trim(), replace(), substring(), toLowerCase(), toUpperCase(), valueOf() – **used to convert primitive data types into String**. But for objects, valueOf() method calls toString() function.

**toString()** method of Object Class returns the string representation of the object used to invoke this method

1. StringBuffer class is used to create **mutable String** object. StringBuffer class is used when we’ve to make lot of modifications to out string. It is also **Thread Safe**

StringBuffer(), StringBuffer(int size), **StringBuffer(String str), StringBuffer(charSequence[] ch)** are 4 constructors. Default buffer size is 16 characters

1. **StringBuffer class Methods: -** append(), insert()- insert one string into another, reverse(), replace(), capacity(), ensureCapacity()- used to ensure minimum capacity of StringBuffer obj

**For ex- str.ensureCapacity(10);**

1. StringBuilder class is **not** Thread Safe i.e. it is not Synchronized
2. StringBuilder()- default room for 16 characters, StringBuilder(int size), StringBuilder(String str) are three constructors.

Exception Handling

1. A Java Exception is an Object that describes the exception that occurs in a program. When an exceptional event occur in java, an exception is said to be thrown
2. Checked Exception: - The exception that can be predicted by the programmer. Ex- File not found. These type of exceptions must be checked at compile time. Some checked exceptions are: SQLException,IOException,DataAccessException,CloneNotSupportedException,InterruptedException
3. Unchecked Exception: - are the classes that extends RuntimeException. Unchecked Exceptions are ignored at compile time and are checked at runtime. Ex- AirthmeticException, NullPointerException, ArrayIndexOutOfBoundException, IllegalArgumentException
4. Error: - Errors are ignored in code because we cannot do anything about the error. Ex- if stack overflow occurs, an error will arise.
5. Exception handling is done by transferring the execution of a program to an appropriate exception handler when exception occurs
6. Example for Unreachable Catch block: - While using multiple catch statements, it is **important to remember** that exception sub-classes inside **catch** must come **before any of their super classes** otherwise it’ll lead to Compile Time Error

class Excep

{

public static void main(String[] args)

{

try

{

int arr[]={1,2};

arr[2]=3/0;

}

catch(Exception e) //This block handles all Exception

{

System.out.println("Generic exception");

}

catch(ArrayIndexOutOfBoundsException e) //This block is unreachable

{

System.out.println("array index out of bound exception");

}

}

}

1. Nested try block is used when a part of a block may cause one error while entire block may cause another error. In case, if inner **try** block does not have a **catch** handler for a particular exception then the outer **try** is checked for match
2. **Important points to Remember**
3. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.
4. Super class **Throwable** overrides **toString()** function, to display error message in form of string.
5. While using multiple catch block, always make sure that exception subclasses comes before any of their super classes. Else you will get compile time error.
6. In nested try catch, the inner try block, uses its own catch block as well as catch block of the outer try, if required.
7. Only the object of Throwable class or its subclasses can be thrown.
8. Throw keyword: - It is used to explicitly throw exception as well as custom exception. We can throw **either checked or unchecked** exception. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception. After throw statement we cannot write any code below this, otherwise we’ll get CTError UnreachableStatement.
9. Throws keyword: - Throws is mainly used for propagate **checked** exception and used to specify the method level exception. Means, it is used to propagate the checked exception to the caller of the method that use throws keyword. It is used to declare the exception, it provides information to the programmer that there may occur an exception so during call of that method, programmer must use exception handling mechanism.

If a method A() can throw any checked exception, then we declare this by using throws keyword.

A() throws Exception

If we need the method A() anywhere in the program then, while calling this method we need to use try/catch block or using throws keyword to calling method(here main) to handle this exception.

For Example:-

Class MyException extends Exception{ // Exception is checked exception

MyException(String s){

super(s); // Creating custom exception

}

}

Class ex2{

Static void valid(int i) throws MyException{

if(i<18){

throw new MyException(“Not Valid”);

}

}

public static void main(String args[]){

try{

valid(12);

}catch(MyException e){System.out.println(e);}

}

}

1. **Points to remember** when using Method Overriding with Exception Handling
2. If super class method does not declare any exception, then sub class overridden method **cannot** declare **checked exception** but it **can** declare **unchecked exception**

class Super

{

void show() { System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws IOException //Compile time error

{ System.out.println("parent class"); }

public static void main( String[] args )

{

Super s=new Sub();

s.show();

}

}

As the method show() doesn’t throws any exception while in Super class, hence its overridden version can also **not throw any checked exception**

import java.io.\*;

class Super

{

void show(){ System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws ArrayIndexOutOfBoundsException //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

Super s=new Sub();

s.show();

}

}

**Because ArrayIndexOutOfBoundException** is an unchecked exception, hence overrided show() method can throw it.

1. If super class method throws an exception, then subclass overridden method can throw the **same** exception or **no** exception, but **must not throw parent exception** of the exception thrown by Super class method.

It means, if Super class method throws object of NullPointerException class, then Subclass method can either throw same exception or can throw no exception, but it can never throw object of Exception class(Parent of NullPointerException class).

import java.io.\*;

class Super

{

void show() throws ArithmeticException

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() throws Exception //Cmpile time Error

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

1. List of Common Checked Exceptions in Java

Common checked exceptions defined in the java.lang package:

* ReflectiveOperationException
  + ClassNotFoundException
  + InstantiationException
  + IllegalAccessException
  + InvocationTargetException
  + NoSuchFieldException
  + NoSuchMethodException
* CloneNotSupportedException
* InterruptedException

Common checked exceptions defined in the java.io package:

* IOException
  + EOFException
  + FileNotFoundException
  + InterruptedIOException
  + UnsupportedEncodingException
  + UTFDataFormatException
  + ObjectStreamException
* InvalidClassException
* InvalidObjectException
* NotSerializableException
* StreamCorruptedException
* WriteAbortedException

Common checked exceptions defined in the java.net package (almost are subtypes of IOException):

* SocketException
  + BindException
  + ConnectException
* HttpRetryException
* MalformedURLException
* ProtocolException
* UnknownHostException
* UnknownServiceException

Common checked exceptions defined in the java.sql package:

* SQLException
  + BatchUpdateException
  + SQLClientInfoException
  + SQLNonTransientException
* SQLDataException
* SQLFeatureNotSupportedException
* SQLIntegrityConstraintViolationException
* SQLSyntaxErrorException
  + SQLTransientException
* SQLTimeoutException
* SQLTransactionRollbackException
* SQLTransientConnectionException
  + SQLRecoverableException
  + SQLWarning

1. List of common Unchecked Exception in Java

Common unchecked exceptions in the java.lang package:

* ArithmeticException
* IndexOutOfBoundsException
  + ArrayIndexOutOfBoundsException
  + StringIndexOutOfBoundsException
* ArrayStoreException
* ClassCastException
* EnumConstantNotPresentException
* IllegalArgumentException
  + IllegalThreadStateException
  + NumberFormatException
* IllegalMonitorStateException
* IllegalStateException
* NegativeArraySizeException
* NullPointerException
* SecurityException
* TypeNotPresentException
* UnsupportedOperationException

Common unchecked exceptions in the java.util package:

* ConcurrentModificationException
* EmptyStackException
* NoSuchElementException
  + InputMismatchException
* MissingResourceException

Multi-Threading

* 1. Thread Class: - Thread class is the main class on which Java’s Multithreading system is based. Thread class & Runnable interface is used to create and run threads for utilizing multithreading feature of java
  2. Constructors of Thread class :-
     1. Thread()
     2. Thread(String str)
     3. Thread(Runnable r)
     4. Thread(Runnable r, String str)
  3. Method of Thread Class: -
     1. setName() – to give thread a name
     2. getName() – return thread’s name
     3. getPriority() – return thread’s priority
     4. isAlive() – checks if thread is still running or not
     5. join() –Using this method, we tell our thread to wait until the thread on which it is called completes its execution
     6. join(long milliseconds)- It allows us specific time for which we want to wait for the specified thread to terminate.
     7. run() – Entry point for a thread
     8. sleep() – suspend thread for a specified time
     9. start() – start a thread by calling run() method

#### Some Important points to Remember

1. When we extend Thread class, we cannot override setName() and getName() functions, because they are declared final in Thread class.
2. While using sleep(), always handle the exception it throws.

*static* void **sleep**(long *milliseconds*) throws **InterruptedException**

* 1. Interthread Communication: -The **wait**(), **notify**(), **notifyaAll**() of Object class are implemented as final . All three methods can be called only from within **synchronized** context.
     1. wait() tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify
     2. notify() wakes up a thread that called wait() on same object.
     3. notifyAll() wakes up all the thread that called wait() on same object.
  2. **Difference between wait() and sleep()**

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| called from synchronised block | no such requirement |
| monitor is released | monitor is not released |
| awake when notify() or notifyAll() method is called. | not awake when notify() or notifyAll() method is called |
| not a static method | static method |
| wait() is generaly used on condition | sleep() method is simply used to put your thread on sleep. |

Collections

**Why Collections were made Generic ?**

Generics added type safety to Collection framework. Earlier collections stored Object class references. Which means any collection could store any type of object. Hence there were chances of storing incompatible types in a collection, which could result in run time mismatch. Hence Generics was introduced, now you can explicitly state the type of object being stored.

**Collections and Autoboxing**

We have studied that Autoboxing converts primitive types into Wrapper class Objects. As collections doesn't store primitive data types(stores only refrences), hence Autoboxing facilitates the storing of primitive data types in collection by boxing it into its wrapper type.

**Most Commonly thrown Exceptions in Collection Framework**

|  |  |
| --- | --- |
| **Exception Name** | **Description** |
| UnSupportedOperationException | occurs if a Collection cannot be modified |
| ClassCastException | occurs when one object is incompatible with another |
| NullPointerException | occurs when you try to store null object in Collection |
| IllegalArgumentException | thrown if an invalid argument is used |
| IllegalStateException | thrown if you try to add an element to an already full Collection |

**The Collection Interface**

1. It is at the top of collection heirarchy and must be implemented by any class that defines a collection. Its general declaration is,
2. *interface* **Collection** < *E* >
3. Following are some of the commonly used methods in this interface.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| add( E obj ) | Used to add objects to a collection. Doesn't add duplicate elements to the collection. |
| addAll( Collection C ) | Add all elements of collection C to the invoking collection |
| remove( Object obj ) | To remove an object from collection |
| removeAll( Collection C ) | Removes all element of collection C from the invoking collection |
| contains( Object obj ) | To determine whether an object is present in collection or not |
| isEmpty() | Returns true if collection is empty, else returns false |
| size() | returns number of elements present in collection |

**The List Interface**

1. It extends the **Collection** Interface, and defines storage as sequence of elements. Following is its general declaration,

*interface* **List** < *E* >

1. Allows random access and insertion, based on position.
2. It allows Duplicate elements. And insertion order is preserved
3. Apart from methods of Collection Interface, it adds following methods of its own.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| get( int index ) | Returns object stored at the specified index |
| set( int index, E obj) | Stores object at the specified index in the calling collection |
| indexOf( Object obj ) | Returns index of first occurence of obj in the collection |
| lastIndexOf( Object obj ) | Returns index of last occurence of obj in the collection |
| subList( int start, int end ) | Returns a list containing elements between start and end index in the collection |

**The Set Interface**

1. This interface defines a Set. It extends **Collection** interface and doesn't allow insertion of duplicate elements. Insertion order is not preserved It's general declaration is,

*interface* **Set** < *E* >

1. It doesn't define any method of its own. It has two sub interfaces, **SortedSet** and **NavigabeSet** .
2. **SortedSet** interface extends **Set** interface and arranges added elements in an ascending order.
3. **NavigabeSet** interface extends **SortedSet** interface, and allows retrieval of elements based on the closest match to a given value or values.

**The Queue Interface**

1. It extends **collection** interface and defines behavior of queue that is first-in, first-out. It's general declaration is,

*interface* **Queue** < *E* >

1. There are couple of new and interesting methods added by this interface. Some of them are mentioned in below table.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| poll() | removes element at the head of the queue and returns **null** if queue is empty |
| remove() | removes element at the head of the queue and throws **NoSuchElementException** if queue is empty |
| peek() | returns the element at the head of the queue without removing it. Returns **null** if queue is empty |
| element() | same as peek(), but throws **NoSuchElementException** if queue is empty |
| offer( E obj ) | Adds object to queue. |

**The Dequeue Interface**

1. It extends **Queue** interface and declares behaviour of a double-ended queue. Its general declaration is,

*interface* **Dequeue** < *E* >

1. Double ended queues can function as simple queues as well as like standard Stacks.

**The Collection classes**

Java provides a set of Collection classes that implements Collection interface. Some of these classes provide full implementations that can be used as it is and other abstract classes provides skeletal implementations that can be used as starting points for creating concrete collections.

**ArrayList class**

1. ArrayList class extends **AbstractList** class and implements the **List** interface.
2. ArrayList supports dynamic array that can grow as needed. ArrayList has three constructors.
3. **ArrayList**()
4. **ArrayList**( *Collection* C )
5. **ArrayList**( int *capacity* )
6. ArrayLists are created with an initial size, when this size is exceeded, it gets enlarged automatically.
7. It can contain Duplicate elements and maintains the insertion order.
8. ArrayLists are not synchronized.

**Example of ArrayList**

import java.util.\*

class Test

{

public static void main(String[] args)

{

**ArrayList< String> al = new ArrayList< String>()**;

al.add("ab");

al.add("bc");

al.add("cd");

system.out.println(al);

}

}

**Output :**

[ab,bc,cd]

**Getting Array from an ArrayList**

toArray() method is used to get an araay containing all the contents of the list. Following are the reasons why you must obtain array from your ArrayList whenever required.

* To obtain faster processing.
* To pass array to methods who do not accept Collectionn as arguments.
* To integrate and use collections with legacy code.

**Storing User-Defined classes**

In the above example we are storing only string object in ArrayList collection. But You can store any type of object, including object of class that you create in Collection classes.

**Example of storing User-Defined object**

**Contact class**

class Contact

{

String first\_name;

String last\_name;

String phone\_no;

public Contact(String fn,String ln,String pn)

{

first\_name = fn;

last\_name = ln;

phone\_no = pn;

}

public String toString()

{

return first\_name+" "+last\_name+"("+phone\_no+")";

}

}

**Storing Contact class**

public class PhoneBook

{

public static void main(String[] args)

{

Contact c1 = new Contact("Ricky", "Pointing","999100091");

Contact c2 = new Contact("David", "Beckham","998392819");

Contact c3 = new Contact("Virat", "Kohli","998131319");

**ArrayList< Contact> al = new ArrayList< Contact>();**

al.add(c1);

al.add(c2);

al.add(c3);

System.out.println(al);

}

}

**Output:**

[Ricky Pointing(999100091), David Beckham(998392819), Virat Kohli(998131319)]

**LinkedList class**

1. LinkedList class extends **AbstractSequentialList** and implements **List**,**Deque** and **Queue** inteface.
2. It can be used as List, stack or Queue as it implements all the related interfaces.
3. It can contain duplicate elements and is not synchronized.

**Example of LinkedList class**

import java.util.\* ;

class Test

{

public static void main(String[] args)

{

**LinkedList< String> ll = new LinkedList< String>();**

ll.add("a");

ll.add("b");

ll.add("c");

ll.addLast("z");

ll.addFirst("A");

System.out.println(ll);

}

}

**Output:**

[A, a, b,c, z]

**HashSet class**

1. HashSet extends **AbstractSet** class and implements the **Set** interface.
2. It creates a collection that uses hash table for storage.
3. HashSet does not maintain any order of elements.

**Example of HashSet class**

import java.util.\*;

class HashSetDemo

{

public static void main(String args[])

{

**HashSet< String> hs = new HashSet< String>();**

hs.add("B");

hs.add("A");

hs.add("D");

hs.add("E");

hs.add("C");

hs.add("F");

System.out.println(hs);

}

}

**Output:**

[D, E, F, A, B, C]

**LinkedHashSet Class**

1. LinkedHashSet class extends **HashSet** class
2. LinkedHashSet maintains a linked list of entries in the set.
3. LinkedHashSet stores elements in the order in which elements are inserted.

**Example of LinkedHashSet class**

import java.util.\*;

class LinkedHashSetDemo

{

public static void main(String args[])

{

**LinkedHashSet< String> hs = new LinkedHashSet< String>()**;

hs.add("B");

hs.add("A");

hs.add("D");

hs.add("E");

hs.add("C");

hs.add("F");

System.out.println(hs);

}

}

**Output :**

[B, A, D, E, C, F]

**TreeSet Class**

1. It extends **AbstractSet** class and implements the **NavigableSet** interface.
2. It stores elements sorted ascending order.
3. Uses a Tree structure to store elements.
4. Access and retrieval times are quite fast.
5. It has four Constructors.

**TreeSet**()

**TreeSet**( Collection *C* )

**TreeSet**( Comparator *comp* )

**TreeSet**( SortedSet *ss* )

**Accessing a Collection**

To access, modify or remove any element from any collection we need to first find the element, for which we have to cycle throught the elements of the collection. There are three possible ways to cycle through the elements of any collection.

1. Using Iterator interface
2. Using ListIterator interface
3. Using for-each loop

**Accessing elements using Iterator**

Iterator Interface is used to traverse a list in forward direction, enabling you to remove or modify the elements of the collection. Each collection classes provide **iterator()** method to return an iterator.

import java.util.\*;

class Test\_Iterator

{

public static void main(String[] args)

{

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

ar.add("ab");

ar.add("bc");

ar.add("cd");

ar.add("de");

**Iterator it = ar.iterator()**; //Declaring Iterator

while(it.hasNext())

{

System.out.print(it.next()+" ");

}

}

}

**Output :**

ab bc cd de

**Accessing element using ListIterator**

ListIterator Interface is used to traverse a list in both forward and backward direction. It is available to only those collections that implement the **List** Interface.

import java.util.\*;

class Test\_Iterator

{

public static void main(String[] args)

{

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

ar.add("ab");

ar.add("bc");

ar.add("cd");

ar.add("de");

**ListIterator litr = ar.listIterator()**;

while(litr.hasNext()) //In forward direction

{

System.out.print(litr.next()+" ");

}

while(litr.hasPrevious()) //In backward direction

{

System.out.print(litr.next()+" ");

}

}

}

**Output :**

ab bc cd de

de cd bc ab

**Using for-each loop**

for-each version of for loop can also be used for traversing each element of a collection. But this can only be used if we don't want to modify the contents of a collection and we don't want any reverse access. for-each loop can cycle through any collection of object that implements Iterable interface.

import java.util.\*;

class ForEachDemo

{

public static void main(String[] args)

{

LinkedList< String> ls = new LinkedList< String>();

ls.add("a");

ls.add("b");

ls.add("c");

ls.add("d");

**for(String str : ls)**

{

System.out.print(str+" ");

}

}

}

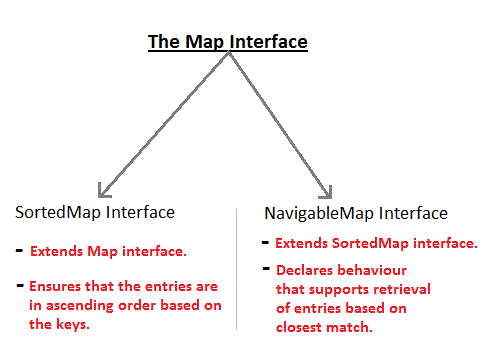
**Output :**

a b c d

**Map Interface**

A Map stores data in key and value association. Both key and values are objects. The key must be **unique** but the values **can** **be** **duplicate**. Although Maps are a part of Collection Framework, they cannot actually be called as collections because of some properties that they possess. However we can obtain a **collection-view** of maps.

|  |  |
| --- | --- |
| **Interface** | **Description** |
| **Map** | Maps unique key to value. |
| **Map.Entry** | Describe an element in key and value pair in a map. This is an inner class of map. |
| **NavigableMap** | Extends SortedMap to handle the retrienal of entries based on closest match searches |
| **SortedMap** | Extends Map so that key are maintained in an ascending order. |



**Commonly used Methods defined by Map**

* boolean **containsKey**(Object *k*): returns true if map contain *k* as key. Otherwise false.
* Object **get**(Object *k*) : returns values associated with the key *k*.
* Object **put**(Object *k*, Object *v*) : stores an entry in map.
* Object **putAll**(Map *m*) : put all entries from *m* in this map.
* Set **keySet**() : returns **Set** that contains the key in a map.
* Set **entrySet**() : returns **Set** that contains the entries in a map.

**HashMap class**

1. HashMap class extends **AbstractMap** and implements **Map** interface.
2. It uses a **hashtable** to store the map. This allows the execution time of get() and put() to remain same.
3. HashMap has four constructor.
   * 1. HashMap()
     2. HashMap(Map< ? extends k, ? extends V> m)
     3. HashMap(int capacity)
     4. HashMap(int capacity, float fillratio)
     5. HashMap does not maintain order of its element.

**Example**

import java.util.\*;

class HashMapDemo

{

public static void main(String args[])

{

**HashMap< String,Integer> hm = new HashMap< String,Integer>()**;

hm.put("a",new Integer(100));

hm.put("b",new Integer(200));

hm.put("c",new Integer(300));

hm.put("d",new Integer(400));

**Set< Map.Entry< String,Integer> > st = hm.entrySet()**; //returns Set view

for(Map.Entry< String,Integer> me:st)

{

System.out.print(me.getKey()+":");

System.out.println(me.getValue());

}

}

}

**Output :**

c 300

a 100

d 400

b 200

**TreeMap class**

1. TreeMap class extends **AbstractMap** and implements **NavigableMap** interface.
2. It creates Map, stored in a tree structure.
3. A **TreeMap** provides an efficient means of storing key/value pair in efficient order.
4. It provides key/value pairs in sorted order and allows rapid retrieval.

**Example**

import java.util.\*;

class TreeMapDemo

{

public static void main(String args[])

{

**TreeMap< String,Integer> tm = new TreeMap< String,Integer>()**;

tm.put("a",new Integer(100));

tm.put("b",new Integer(200));

tm.put("c",new Integer(300));

tm.put("d",new Integer(400));

**Set< Map.Entry< String,Integer> > st = tm.entrySet()**;

for(Map.Entry me:st)

{

System.out.print(me.getKey()+":");

System.out.println(me.getValue());

}

}

}

**Output :**

a 100

b 200

c 300

d 400

**LinkedHashMap class**

1. **LinkedHashMap** extends **HashMap** class.
2. It maintains a linked list of entries in map in order in which they are inserted.
3. **LinkedHashMap** defines the following constructor
4. LinkedHashMap()
5. LinkedHashMap(Map< ? extends k, ? extends V> m)
6. LinkedHashMap(int capacity)
7. LinkedHashMap(int capacity, float fillratio)
8. LinkedHashMap(int capacity, float fillratio, boolean order)
9. It adds one new method removeEldestEntry(). This method is called by put() and putAll() By default this method does nothing. However we can override this method to remove oldest element in the map. **Syntax**
10. protected boolean removeEldestEntry(Map.Entry e)

**EnumMap class**

1. **EnumMap** extends **AbstractMap** and implements **Map** interface.
2. It is used for key as enum

### Comparator Interface

In Java, Comparator interface is used to order the object in your own way. It gives you ability to decide how element are stored within sorted collection and map.

Comparator Interface defines compare() method. This method compare two object and return 0 if two object are equal. It returns a positive value if object1 is greater than object2. Otherwise a negative value is return. The method can throw a **ClassCastException** if the type of object are not compatible for comparison.

#### Example

**Student class**

class Student

int roll;

String name;

Student(int r,String n)

{

roll = r;

name = n;

}

public String toString()

{

return roll+" "+name;

}

**MyComparator class**

This class defines the comparison logic for Student class based on their roll. Student object will be sotred in ascending order of their roll.

class MyComparator implements Comparator

{

public int compare(Student s1,Student s2)

{

if(s1.roll == s2.roll) return 0;

else if(s1.roll > s2.roll) return 1;

else return -1;

}

}

public class Test

{

public static void main(String[] args)

{

TreeSet< Student> ts = new TreeSet< Student>(**new MyComparator()**);

ts.add(new Student(45, "Rahul"));

ts.add(new Student(11, "Adam"));

ts.add(new Student(19, "Alex"));

System.out.println(ts);

}

}

**Output :**

[ 11 Adam, 19 Alex, 45 Rahul ]

As you can see in the ouput Student object are stored in ascending order of their **roll**.

**Legacy Classes**

Early version of java did not include the **Collection** framework. It only defined several classes and interface that provide method for storing objects. When **Collection** framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as Legacy classes. All legacy claases and interface were redesign by JDK 5 to support Generics.

The following are the legacy classes defined by **java.util** package

1. Dictionary
2. HashTable
3. Properties
4. Stack
5. Vector

There is only one legacy interface called **Enumeration**

**NOTE:** All the legacy classes are syncronized

**Enumeration interface**

1. **Enumeration** interface defines method to enumerate through collection of object.
2. This interface is suspended by **Iterator** interface.
3. However some legacy classes such as **Vector** and **Properties** defines several method in which **Enumeration** interface is used.
4. It specifies the following two methods

boolean hasMoreElements()

Object nextElement()

**Vector class**

1. **Vector** is similar to **ArrayList** which represents a dynamic array.
2. The only difference between **Vector** and **ArrayList** is that Vector is synchronised while Array is not.
3. Vector class has following four constructor
   1. Vector()
   2. Vector(int size)
4. Vector(int size, int incr)
5. Vector(Collection< ? extends E> c)

Vector defines several legacy method. Lets see some important legacy method define by **Vector** class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| addElement() | add element to the Vector |
| elementAt() | return the element at specified index |
| elements | return an enumeration of element in vector |
| firstElement() | return first element in the Vector |
| lastElement() | return last element in the Vector |
| removeAllElement() | remove all element of the Vector |

**Example of Vector**

import java.util.\*;

public class Test

{

public static void main(String[] args)

{

Vector ve = new Vector();

ve.add(10);

ve.add(20);

ve.add(30);

ve.add(40);

ve.add(50);

ve.add(60);

Enumeration en = ve.elements();

while(en.hasMoreElements())

{

System.out.println(en.nextElement());

}

}

}

**Output :**

10

20

30

40

50

60

**Hashtable class**

1. Like HashMap, Hashtable also stores key/value pair in hashtable. However neither **keys** nor **values** can be **null**.
2. There is one more difference between **HashMap** and **Hashtable** that is Hashtable is synchronized while HashMap is not.
3. Hashtable has following four constructor
4. Hashtable()
5. Hashtable(int size)
6. Hashtable(int size, float fillratio)
7. Hashtable(Map< ? extends K, ? extends V> m)

**Example of Hashtable**

import java.util.\*;

class HashTableDemo

{

public static void main(String args[])

{

**Hashtable< String,Integer> ht = new Hashtable< String,Integer>()**;

ht.put("a",new Integer(100));

ht.put("b",new Integer(200));

ht.put("c",new Integer(300));

ht.put("d",new Integer(400));

Set st = ht.entrySet();

Iterator itr=st.iterator();

while(itr.hasNext())

{

Map.Entry m=(Map.Entry)itr.next();

System.out.println(itr.getKey()+" "+itr.getValue());

}

}

}

**Output:**

a 100

b 200

c 300

d 400

**Difference between HashMap and Hashtable**

|  |  |
| --- | --- |
| **Hashtable** | **HashMap** |
| Hashtable class is synchronized. | HastMap is not synchronize. |
| Because of Thread-safe, Hashtable is slower than HashMap | HashMap works faster. |
| Neither **key** nor **values** can be null | Both **key** and **values** can be null |
| Order of table remain constant over time. | does not guarantee that order of map remain constant over time. |

**Properties class**

1. **Properties** class extends **Hashtable** class.
2. It is used to maintain list of value in which both key and value are **String**
3. **Properties** class define two constructor
4. Properties()
5. Properties(Properties default)
6. One advantage of **Properties** over **Hashtable** is that we can specify a default property that will be useful when no value is associated with a certain key.

**Example of Properties class**

import java.util.\*;

public class Test

{

public static void main(String[] args)

{

Properties pr = new Properties();

pr.put("Java", "James Ghosling");

pr.put("C++", "Bjarne Stroustrup");

pr.put("C", "Dennis Ritchie");

pr.put("C#", "Microsoft Inc.");

Set< ?> creator = pr.keySet();

for(Object ob: creator)

{

System.out.println(ob+" was created by "+ pr.getProperty((String)ob) );

}

}

}

**Output :**

Java was created by James Ghosling

C++ was created by Bjarne Stroustrup

C was created by Dennis Ritchie

C# was created by Microsoft Inc