CORE JAVA IMPORTANT QUESTIONS

1.What is java

Java is a programming language and a platform.

Java is a high level, robust, secured and object-oriented programming language.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.

2.Internal compilation of program?

What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.

compilation of simple java program

What happens at runtime?

|  |
| --- |
| At runtime, following steps are performed: |
| what happens at runtime when simple java program runs |

|  |
| --- |
| Classloader: is the subsystem of JVM that is used to load class files. |
| Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects. |
| Interpreter: read bytecode stream then execute the instructions. |

3.How to set path?

The path is required to be set for using tools such as javac, java etc.

If you are saving the java source file inside the jdk/bin directory, path is not required to be set because all the tools will be available in the current directory.

But If you are having your java file outside the jdk/bin folder, it is necessary to set path of JDK.

There are 2 ways to set java path:

1. temporary
2. permanent

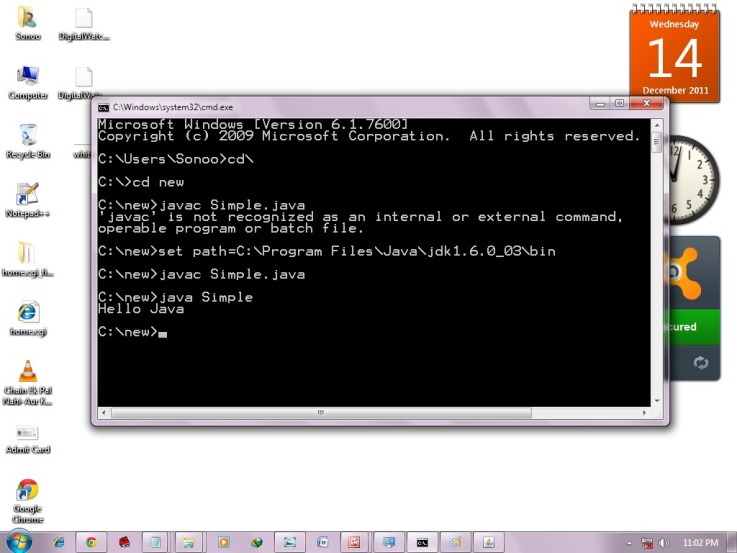
1) How to set Temporary Path of JDK in Windows

To set the temporary path of JDK, you need to follow following steps:

1. Open command prompt
2. copy the path of jdk/bin directory
3. write in command prompt: set path=copied\_path

For Example:

set path=C:\Program Files\Java\jdk1.6.0\_23\bin



2) How to set Permanent Path of JDK in Windows

For setting the permanent path of JDK, you need to follow these steps:

* Go to My Computer properties -> advanced tab -> environment variables -> new tab of user variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok

For Example:

4.Explain about JVM, JRE, JDK

1. **JVM**

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

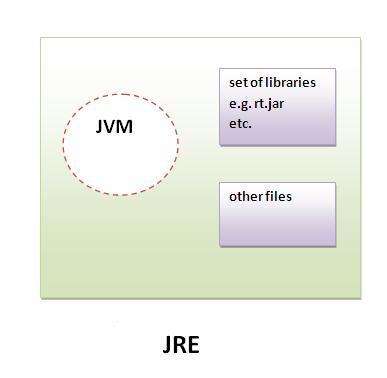
The JVM performs following main tasks:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

1. **JRE**

JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



1. **JDK**

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.



5.Internal details of jvm

JVM (Java Virtual Machine)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

* What is JVM

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

* What it does

The JVM performs following operation:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

JVM provides definitions for the:

* Memory area
* Class file format
* Register set
* Garbage-collected heap
* Fatal error reporting etc.
* Internal Architecture of JVM

Let's understand the internal architecture of JVM. It contains class loader, memory area, execution engine etc.



1) Classloader

Classloader is a subsystem of JVM that is used to load class files.

2) Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3) Heap

It is the runtime data area in which objects are allocated.

4) Stack

|  |
| --- |
| Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. |
| Each thread has a private JVM stack, created at the same time as thread. |
| A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes. |

5) Program Counter Register

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack

It contains all the native methods used in the application.

7) Execution Engine

|  |
| --- |
| It contains: |
| **1) A virtual processor** |
| **2) Interpreter:** Read bytecode stream then execute the instructions. |
| **3) Just-In-Time(JIT) compiler:** It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term ?compiler? refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU. |

6)Elaborate public static void main (String args [])

* **Public** : is an [Access Modifier](http://javabeginnerstutorial.com/core-java-tutorial/access-modifier-in-java/), which defines who can access this Method. Public means that this Method will be accessible by any Class (If other Classes are able to access this Class.).
* **Static:**is a keyword which identifies the class related thing. This means the given Method or variable is not instance related but Class related. It can be accessed without creating the instance of a Class.
* **Void:** is used to define the Return Type of the Method. It defines what the method can return. Void means the Method will not return any value.
* **main:**is the **n**ame of the Method**.**This Method name is searched by JVM as a starting point for an application with a particular signature only.
* **String args []:** is the parameter to the main Method.

Variables and datatypes?

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in java: primitive and non-primitive.

**Variable** is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*. It is a combination of "vary + able" that means its value can be changed.

variables in java

**Ex:** int data=50;//Here data is variable

**Types of Variable**

There are three types of variables in java:

* local variable
* instance variable
* static variable

types of variables in java

1) Local Variable

A variable which is declared inside the method is called local variable.

2) Instance Variable

A variable which is declared inside the class but outside the method, is called instance variable. It is not declared as static.

3) Static variable

A variable that is declared as static is called static variable. It cannot be local.

We will have detailed learning of these variables in next chapters.

**class** A{

**int** data=50;//instance variable

**static** **int** m=100;//static variable

**void** method(){

**int** n=90;//local variable

}

}//end of class

what is Hashcode?

1. Hashcode is an integer which represents internal address of an object.
2. As Hashcode represents internal address of an object.,
3. Hashcode differs from one object to other object.
4. Hashcode acts like an identity of an object

**Purpose of object or when Hashcode of an object is used:**

Hashcode is used to store, remove and search an object in Set and Map Collection Objects

Ex: HashSet and HashMap. Both of these will work in the principle of Hashcode.

**How can we retrieve Hashcode of an Object?**

In Object class we have a method called hashcode() which returns an integer value.

Ex: int hashcode(); //method defined in Object class which is super class for all classes in java

**How to generate hashcode of an object:**

By default, our java will generate hashcode of an object but if we want to generate hashcode of an object we need to override the hashcode of an object

Diff bw == operator and. equals () method

* == operator meant for reference/address comparison.

Example:

String s1=new String(“Hello”);

String s2=new String(“Hello”);

/\*false bcoz s1 and s2 are pointing to 2 different objects and if both s1 and s2 are pointing to same object then it returns true…\*/

System.out.println(s1==s2); //false

System.out.println(s1. equals(s2)); //true bcoz both contents are same i.e., “Hello”

* . equals () method meant for content comparison

what is Object and Class

class is a dummy thing or blueprint which doesn’t exists physically Ex: Person,Bird

Object is instance of a class which exists physically Ex: Hari, Lalitha, sparrow

oops concepts

**Object** means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

* Object
* Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

**Object**

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

**Class**

Collection of objects is called class. It is a logical entity.

**Inheritance:**

**Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

Why use inheritance in java

* For Method Overriding (so runtime polymorphism can be achieved).
* For Code Reusability.

**Syntax of Java Inheritance**

class Subclass-name extends Superclass-name

{

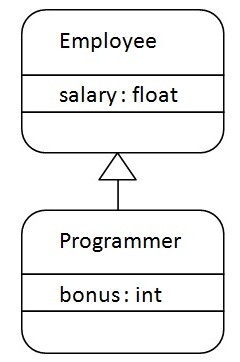
//methods and fields

}

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child or subclass.

**Java Inheritance Example**



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**.It means that Programmer is a type of Employee.

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **class** Programmer **extends** Employee{
5. **int** bonus=10000;
6. **public** **static** **void** main(String args[]){
7. Programmer p=**new** Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }  }

**Output:**

Programmer salary is:40000.0

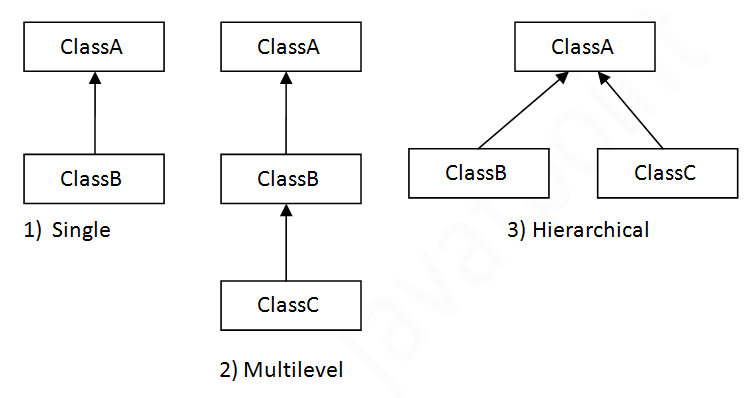
Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

**Types of inheritance in java**

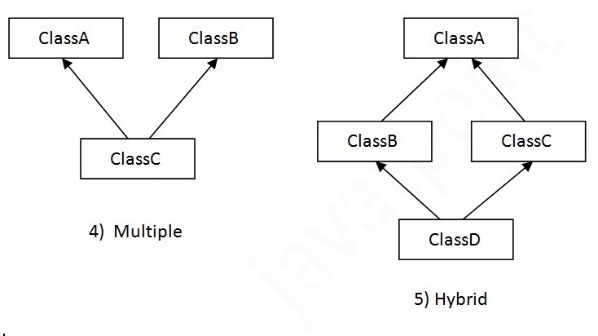
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note**: Multiple inheritance is not supported in java through class.**

**When a class extends multiple classes i.e. known as multiple inheritance. For Example:**



**Single Inheritance Example**

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** TestInheritance{
8. **public** **static** **void** main(String args[]){
9. Dog d=**new** Dog();
10. d.bark();
11. d.eat();
12. }}

**Output:**

barking...

eating...

**Multilevel Inheritance Example**

*File: TestInheritance2.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** weep(){System.out.println("weeping...");}
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** main(String args[]){
12. BabyDog d=**new** BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}

**Output:**

weeping...

barking...

eating...

**Hierarchical Inheritance Example**

File: TestInheritance3.java

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** meow(){System.out.println("meowing...");}
9. }
10. **class** TestInheritance3{
11. **public** **static** **void** main(String args[]){
12. Cat c=**new** Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}

**Output:**

meowing...

eating...

what is encapsulation

**Encapsulation in java** is a *process of wrapping code and data together into a single unit*, for example capsule i.e. mixed of several medicines.



We can create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of fully encapsulated class.

**Advantage of Encapsulation in java**

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

**Simple example of encapsulation in java**

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

**//save as Student.java**

1. **package** com.javatpoint;
2. **public** **class** Student{
3. **private** String name;
5. **public** String getName(){
6. **return** name;
7. }
8. **public** **void** setName(String name){
9. **this**.name=name
10. }
11. }

**//save as Test.java**

1. package com.javatpoint;
2. class Test{
3. public static void main(String[] args){
4. Student s=new Student();
5. s.setName("vijay");
6. System.out.println(s.getName());
7. }
8. }

Output: vijay

what is polymorphism and static polymorphism and dynamic polymorphism

In greek, Poly means many and morph means shapes or forms. So. Polymorphism refers to any entity which takes many form.

Polymorphism in java refers to any entity whether it is an operator or a constructor or any method which takes many forms or can be used for multiple tasks either while compiling or while running a java program.

There are two types of polymorphism in Java.   **1) Static Polymorphism       2) Dynamic Polymorphism**

**1) Static Polymorphism**

Any entity which shows polymorphism during compile time is called static polymorphism. Operator Overloading, Constructor Overloading and [method overloading](http://javaconceptoftheday.com/method-overloading-in-java/) are best examples of static polymorphism. Because, they show polymorphism during compilation.

In static polymorphism, object used is determined during compilation itself. So, it is called **static binding or Early Binding**.

* **Operator Overloading :**For example, Operator ‘+’ can be used to add two numbers and also can be used to concatenate two strings. It is called operator overloading. ‘+’ is the only operator in java which is used for operator overloading.
* **Constructor Overloading :**We can include multiple constructors in a class. This is called constructor overloading. Through [constructor overloading](http://javaconceptoftheday.com/constructors-in-java/), we can create objects to the class in multiple ways. This shows the polymorphism.
* **Method Overloading :**We can have different forms of same method in the same class. This is called [method overloading](http://javaconceptoftheday.com/method-overloading-in-java/). Through method overloading we can perform different tasks through different forms of the same method. This shows the polymorphism.

In [casting](http://javaconceptoftheday.com/type-casting-in-java/), we have seen super class reference variable can refer to objects of its sub class. This also shows polymorphism. For example,

class A

{

     //Some Statements

}

class B extends A

{

     //Some Statements

}

class C extends B

{

    //Some Statements

}

public class D

{

    public static void main(String[] args)

    {

        A a = new A();  //A-Class variable refers to A-Class object

        a = new B();    //A-Class variable refers to B-Class object

        a = new C();    //A-Class variable refers to C-Class object

    }

}

In the above example, ‘a’ is Class A-type reference variable which can be used to refer objects of A-type, B-type or C-type. Because, B-type and C-type are sub class of A-type. This also shows the polymorphism.

**2) Dynamic Polymorphism**

Any entity which shows polymorphism during run time is called dynamic polymorphism. [Method Overriding](http://javaconceptoftheday.com/method-overriding-java/) is the best example of dynamic polymorphism. It is also called **dynamic binding or late binding**, because type of the object used will be determined at run time only.

Consider the following example,

class A

{

    void methodA()

    {

        System.out.println("From Super Class");

    }

}

class B extends A

{

    //Super Class Method Overrided

    void methodA()

    {

        System.out.println("From Sub Class");

    }

}

public class D

{

    static void util(A a)

    {

        a.methodA();

        //For each execution of this method, different objects will be passed to it.

        //which Object will be used is determined during run time only.

        //This shows dynamic polymorphism.

    }

    public static void main(String[] args)

    {

        A a = new A();

        B b = new B();

        A a1 = new B();

        util(a);  //SuperClass object is passes to util()

        util(b);     //SubClass object is passed to util()

        util(a1); //SubClass object is passed to util()

    }

}

what is method overloading and method overriding

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

**Advantage of method overloading**

Method overloading *increases the readability of the program*.

**Different ways to overload the method**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

**Note:In java, Method Overloading is not possible by changing the return type of the method only.**

**1) Method Overloading: changing no. of arguments**

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

1. **class** Adder{
2. **static** **int** add(**int** a,**int** b){**return** a+b;}
3. **static** **int** add(**int** a,**int** b,**int** c){**return** a+b+c;}
4. }
5. **class** TestOverloading1{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}

Output:

22

33

**2) Method Overloading: changing data type of arguments**

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

1. **class** Adder{
2. **static** **int** add(**int** a, **int** b){**return** a+b;}
3. **static** **double** add(**double** a, **double** b){**return** a+b;}
4. }
5. **class** TestOverloading2{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}

Output:

22

24.9

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

1. **class** Adder{
2. **static** **int** add(**int** a,**int** b){**return** a+b;}
3. **static** **double** add(**int** a,**int** b){**return** a+b;}
4. }
5. **class** TestOverloading3{
6. **public** **static** **void** main(String[] args){
7. System.out.println(Adder.add(11,11));//ambiguity
8. }}

**Output**:

Compile Time Error: method add(int,int) is already defined in class Adder

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

**Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.**

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

1. **class** TestOverloading4{
2. **public** **static** **void** main(String[] args){System.out.println("main with String[]");}
3. **public** **static** **void** main(String args){System.out.println("main with String");}
4. **public** **static** **void** main(){System.out.println("main without args");}
5. }

Output:

main with String[]

can java support multiple inheritance if not why??

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

1. **class** A{
2. **void** msg(){System.out.println("Hello");}
3. }
4. **class** B{
5. **void** msg(){System.out.println("Welcome");}
6. }
7. **class** C **extends** A,B{//suppose if it were
9. Public Static **void** main(String args[]){
10. C obj=**new** C();
11. obj.msg();//Now which msg() method would be invoked?
12. }
13. }

**Output**:

Compile Time Error

what is constructor..default and parameterized constructor??

what is static keyword

what is static Initializer block,static variables and static methods

what are non static members and their memory management

what is this keyword

super keyword

Abstract class and Interface and their differences

access modifiers

what is exception and exception hierarchy??

method overriding with throws clause

about try and catch blocks

about multiple catch blocks and nested try block

what is finally block

what is the use of throw keyword

diff bw throw, throws and throwable??

diff bw final,finally and finalize

Return value from try, catch and finally block??

Diff bw Error and Exception

diff bw ClassNotFoundException and NoClassDefFoundError??

what are custom exceptions

what are checked and unchecked

what is multithreading

In thread-based multitasking or multithreading, multiple threads in a process are executed simultaneously. For example, MS word can print a document using background thread, at the same another thread can accept the user input so that user can create a new document.

Now come to Threads in java. Consider this example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class ThreadsInJava  {      //Main Thread      public static void main(String[] args)      {          for (int i = 0; i <= 100; i++)          {              System.out.println("From Main Thread");          }      }  } |

When you run this program, java command creates a primary thread or main thread which is responsible for executing main method. That’s why, execution of all java programs start with main() method.

This is an example of single thread execution. Java supports multi thread execution. i.e Java program can have more than one threads that can run simultaneously. This is called multithreaded programming.

For example, in the below program, main thread creates two threads. The task of first thread is to print the numbers from 0 to 1000. The task of second thread is to print the numbers from 1001 to 2000. These two threads perform their task simulataneously not one after the other.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | //Defining first thread with task  //The task of this thread is to print the numbers from 0 to 1000 times  class Thread1 extends Thread  {      @Override      public void run()      {          for(int i = 0; i <= 1000; i++)          {              System.out.println(i);          }      }  }    //Defining second thread with task  //The task of this thread is to print the numbers from 1001 to 2000  class Thread2 extends Thread  {      @Override      public void run()      {          for(int i = 1001; i<= 2000; i++)          {              System.out.println(i);          }      }  }    public class ThreadsInJava  {      //Main Thread      public static void main(String[] args)      {          //Creating first thread          Thread1 t1 = new Thread1();          t1.start();            //Creating second thread          Thread2 t2 = new Thread2();          t2.start();      }  } |

life cycle of a thread

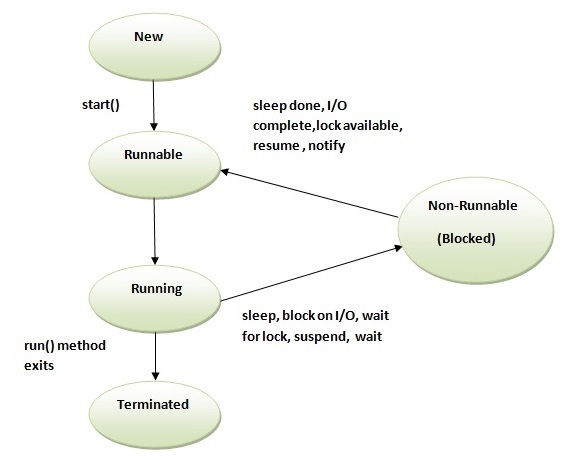
**Life cycle of a Thread (Thread States)**

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



|  |
| --- |
| 1) New  The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

**Commonly used methods of Thread class:**

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. 2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():**waits for a thread to die. 5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds. 6. **public int getPriority():**returns the priority of the thread. 7. **public int setPriority(int priority):**changes the priority of the thread. 8. **public String getName():**returns the name of the thread. 9. **public void setName(String name):**changes the name of the thread. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted. |

in how many ways we can create threads??

There are two ways to create threads in java language.

1) By extending **java.lang.Thread** class.

2) By implementing **java.lang.Runnable** interface.

1) By Extending java.lang.Thread Class

You can create your own thread by extending **Thread class of java.lang** package. You have to override run() method of Thread class and keep the task which you want your thread to perform in this run() method. Here is the syntax of creating a thread by extending Thread class.

[?](http://javaconceptoftheday.com/creating-threads-in-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class MyThread extends Thread  {      @Override      public void run()      {          //Keep the task to be performed here      }  } |

After defining your thread, create an object of your thread and call start() method where ever you want this task to be performed. Like this,

|  |  |
| --- | --- |
| 1  2 | MyThread myThread = new MyThread();  myThread.start(); |

The following example shows how to create a thread by extending Thread class and how to start it

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class MyThread extends Thread  {      @Override      public void run()      {          //Task of this thread is to print multiplication of successive numbers up to 1000 numbers          for(int i = 0; i < 1000; i++)          {              System.out.println(i+" \* "+(i+1)+" = " + i \* (i+1));          }      }  }    public class ThreadsInJava  {      //Main Thread      public static void main(String[] args)      {          //Creating and starting MyThread.          MyThread myThread = new MyThread();          myThread.start();      }  } |

2) By Implementing java.lang.Runnable Interface.

Another method of creating a thread is to implement **Runnable interface of java.lang** package. Runnable interface has only one abstract method i.e run(). You have to implement this method and keep the task to be performed in this method. Here is the syntax for creating a thread by implementing Runnable interface.

|  |
| --- |
| class MyThread implements Runnable  {      @Override      public void run()      {          //Keep the task to be performed here      }  } |

After defining the thread, create an object to java.lang.Thread class through a constructor which takes Runnable type as an argument and pass the object of your thread that implements Runnable interface as an argument to it and call the start() method.Like this,

[?](http://javaconceptoftheday.com/creating-threads-in-java/)

|  |  |
| --- | --- |
| 1  2  3 | MyThread myThread = new MyThread();    //Creating object of your thread that implements Runnable interface  Thread t = new Thread(myThread);       //passing your thread object to the constructor of Thread class  t.start();                             //Starting the thread |

The following example shows how to create a thread by implementing Runnable interface and how to start it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class MyThread implements Runnable  {      @Override      public void run()      {          //Task of this thread is to print multiplication of successive numbers up to 1000 numbers          for(int i = 0; i < 1000; i++)          {              System.out.println(i+" \* "+(i+1)+" = " + i \* (i+1));          }      }  }    public class ThreadsInJava  {      //Main Thread      public static void main(String[] args)      {          MyThread myThread = new MyThread();    //instantiating your thread class that implements Runnable interface            Thread t = new Thread(myThread);       //Creating an object to Thread class by passing your thread as an argument            t.start();                            //Starting the thread      }  } |

what is Thread Scheduler

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

sleep() method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

**Syntax of sleep() method in java**

The Thread class provides two methods for sleeping a thread:

* public static void sleep(long miliseconds)throws InterruptedException
* public static void sleep(long miliseconds, int nanos)throws InterruptedException

**Example of sleep method in java**

1. **class** TestSleepMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestSleepMethod1 t1=**new** TestSleepMethod1();
10. TestSleepMethod1 t2=**new** TestSleepMethod1();
12. t1.start();
13. t2.start();
14. }
15. }

Output:

1

1

2

2

3

3

4

4

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

how we can start thread and can we start thread twice??

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

1. **public** **class** TestThreadTwice1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestThreadTwice1 t1=**new** TestThreadTwice1();
7. t1.start();
8. t1.start();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThreadTwice1)

running

Exception in thread "main" java.lang.IllegalThreadStateException

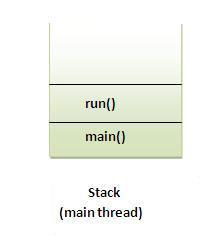
what is the use of run() method??

|  |
| --- |
| * Each thread starts in a separate call stack. * Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack. |

1. **class** TestCallRun1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestCallRun1 t1=**new** TestCallRun1();
7. t1.run();//fine, but does not start a separate call stack
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun1)

Output:running...

 ***Problem if you direct call run() method***

1. **class** TestCallRun2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestCallRun2 t1=**new** TestCallRun2();
10. TestCallRun2 t2=**new** TestCallRun2();
12. t1.run();
13. t2.run();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun2)

Output:1

2

3

4

5

1

2

3

4

5

|  |
| --- |
| As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object. |

can we call run() method instead of start() method

join() method in threads

The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:

|  |
| --- |
| public void join()throws InterruptedException |
| public void join(long milliseconds)throws InterruptedException |

***Example of join() method***

1. **class** TestJoinMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod1 t1=**new** TestJoinMethod1();
12. TestJoinMethod1 t2=**new** TestJoinMethod1();
13. TestJoinMethod1 t3=**new** TestJoinMethod1();
14. t1.start();
15. **try**{
16. t1.join();
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod1)

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

|  |
| --- |
| As you can see in the above example,when t1 completes its task then t2 and t3 starts executing. |

***Example of join(long miliseconds) method***

1. **class** TestJoinMethod2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod2 t1=**new** TestJoinMethod2();
12. TestJoinMethod2 t2=**new** TestJoinMethod2();
13. TestJoinMethod2 t3=**new** TestJoinMethod2();
14. t1.start();
15. **try**{
16. t1.join(1500);
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod2)

Output:1

2

3

1

4

1

2

5

2

3

3

4

4

5

5

|  |
| --- |
| In the above example,when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing. |

getName(),setName(String) and getId() method:

|  |
| --- |
| public String getName() |
| public void setName(String name) |
| public long getId() |

1. **class** TestJoinMethod3 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestJoinMethod3 t1=**new** TestJoinMethod3();
7. TestJoinMethod3 t2=**new** TestJoinMethod3();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
10. System.out.println("id of t1:"+t1.getId());
12. t1.start();
13. t2.start();
15. t1.setName("Sonoo Jaiswal");
16. System.out.println("After changing name of t1:"+t1.getName());
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod3)

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changling name of t1:Sonoo Jaiswal

running...

The currentThread() method:

|  |
| --- |
| The currentThread() method returns a reference to the currently executing thread object. |

Syntax:

|  |
| --- |
| public static Thread currentThread() |

***Example of currentThread() method***

1. **class** TestJoinMethod4 **extends** Thread{
2. **public** **void** run(){
3. System.out.println(Thread.currentThread().getName());
4. }
5. }
6. **public** **static** **void** main(String args[]){
7. TestJoinMethod4 t1=**new** TestJoinMethod4();
8. TestJoinMethod4 t2=**new** TestJoinMethod4();
10. t1.start();
11. t2.start();
12. }
13. }

**Output:**

Thread-0

Thread-1

Thread Priority

Priority of a Thread (Thread Priority):

|  |
| --- |
| Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. |

3 constants defiend in Thread class:

|  |
| --- |
| 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_PRIORITY |

|  |
| --- |
| Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10. |

Example of priority of a Thread:

1. **class** TestMultiPriority1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running thread name is:"+Thread.currentThread().getName());
4. System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
6. }
7. **public** **static** **void** main(String args[]){
8. TestMultiPriority1 m1=**new** TestMultiPriority1();
9. TestMultiPriority1 m2=**new** TestMultiPriority1();
10. m1.setPriority(Thread.MIN\_PRIORITY);
11. m2.setPriority(Thread.MAX\_PRIORITY);
12. m1.start();
13. m2.start();
15. }
16. }

Output:

running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

what is Daemon Thread and User Thread??

There are two types of Threads in java.

1) User Thread

2) Daemon Thread

1) User Thread :

User threads are threads which are created by the application or user. They are high priority threads. JVM (Java Virtual Machine) will not exit until all user threads finish their execution. JVM wait for these threads to finish their task. These threads are foreground threads.

2)Daemon Thread :

Daemon threads are threads which are mostly created by the JVM. These threads always run in background. These threads are used to perform some background tasks like garbage collection and house-keeping tasks. These threads are less priority threads. JVM will not wait for these threads to finish their execution. JVM will exit as soon as all user threads finish their execution. JVM doesn’t wait for daemon threads to finish their task.

Thread Pool??

What is Thread Synchronization

what is Synchronized block and static Synchronization

What is deadlock explain??

what is Inter Thread Communication??

Differnce bw Collection and Collections

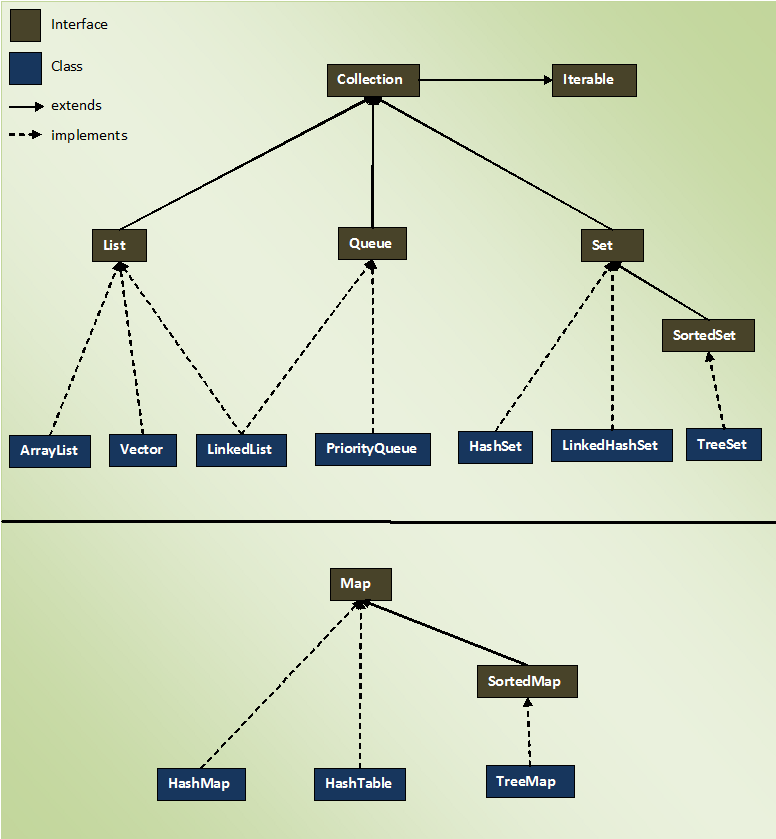
* **Collections in java** is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

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

* Collection in java

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



The entire collection framework is divided into four interfaces.

**1) List**  —> It handles sequential list of objects. **ArrayList**, **Vector** and **LinkedList** classes implement this interface.

**2) Queue**  —> It handles special list of objects in which elements are removed only from the head. **LinkedList** and **PriorityQueue** classes implement this interface.

**3) Set**  —> It handles list of objects which must contain unique element. This interface is implemented by **HashSet** and **LinkedHashSet** classes and extended by **SortedSet** interface which in turn, is implemented by **TreeSet**.

**4) Map**  —> This is the one interface in Collection Framework which is not inherited from Collection interface. It handles group of objects as Key/Value pairs. It is implemented by **HashMap** and **HashTable** classes and extended by **SortedMap** interface which in turn is implemented by **TreeMap**.

Three of above interfaces (List, Queue and Set) inherit from Collection interface. Although, Map is included in collection framework it does not inherit from Collection interface.

**Methods of Collection interface**

There are many methods declared in the Collection interface. They are as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(Object element) | is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection c) | is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | is used to delete an element from this collection. |
| 4 | public boolean removeAll(Collection c) | is used to delete all the elements of specified collection from the invoking collection. |
| 5 | public boolean retainAll(Collection c) | is used to delete all the elements of invoking collection except the specified collection. |
| 6 | public int size() | return the total number of elements in the collection. |
| 7 | public void clear() | removes the total no of element from the collection. |
| 8 | public boolean contains(Object element) | is used to search an element. |
| 9 | public boolean containsAll(Collection c) | is used to search the specified collection in this collection. |
| 10 | public Iterator iterator() | returns an iterator. |
| 11 | public Object[] toArray() | converts collection into array. |
| 12 | public boolean isEmpty() | checks if collection is empty. |
| 13 | public boolean equals(Object element) | matches two collection. |
| 14 | public int hashCode() | returns the hashcode number for collection. |

**Iterator interface**

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in forward direction only. |

**Methods of Iterator interface**

There are only three methods in the Iterator interface. They are:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean hasNext() | It returns true if iterator has more elements. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is rarely used. |

diff bw list and set

* **List** **Interface**

It represents an ordered or sequential collection of objects. This interface has some methods which can be used to store and manipulate the ordered collection of objects. The classes which implement the List interface are called as **Lists**. ArrayList, Vector and LinkedList are some examples of lists. You have the control over where to insert an element and from where to remove an element in the list.

Here are some properties of lists.

* Elements of the lists are ordered using Zero based index.
* You can access the elements of lists using an integer index.
* Elements can be inserted at a specific position using integer index. Any pre-existing elements at or beyond that position are shifted right.
* Elements can be removed from a specific position. The elements beyond that position are shifted left.
* A list may contain duplicate elements.
* A list may contain multiple null elements.

**Methods of Java List Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index,Object element) | It is used to insert element into the invoking list at the index passed in the index. |
| boolean addAll(int index,Collection c) | It is used to insert all elements of c into the invoking list at the index passed in the index. |
| object get(int index) | It is used to return the object stored at the specified index within the invoking collection. |
| object set(int index,Object element) | It is used to assign element to the location specified by index within the invoking list. |
| object remove(int index) | It is used to remove the element at position index from the invoking list and return the deleted element. |
| ListIterator listIterator() | It is used to return an iterator to the start of the invoking list. |
| ListIterator listIterator(int index) | It is used to return an iterator to the invoking list that begins at the specified index. |

**Java List Example**

1. **import** java.util.\*;
2. **public** **class** ListExample{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Amit");
6. al.add("Vijay");
7. al.add("Kumar");
8. al.add(1,"Sachin");
9. System.out.println("Element at 2nd position: "+al.get(2));
10. **for**(String s:al){
11. System.out.println(s);
12. }
13. }
14. }

**Output**:

Element at 2nd position: Vijay

Amit

Sachin

Vijay

Kumar

* **Set interface**

The Set interface defines a set. The **set** is a linear collection of objects with no duplicates. Duplicate elements are not allowed in a set. The **Set interface** extends Collection interface. Set interface does not have it’s own methods. All it’s methods are inherited from Collection interface. The only change that has been made to Set interface is that add() method will return false if you try to insert an element which is already present in the set.

**Properties Of Set :**

* Set contains only unique elements. It does not allow **duplicates**.
* Set can contain only one **null** element.
* **Random access** of elements is not possible.
* **Order of elements** in a set is implementation dependent. **HashSet** elements are ordered on hash code of elements. **TreeSet** elements are ordered according to supplied Comparator (If no Comparator is supplied, elements will be placed in ascending order) and **LinkedHashSet** maintains insertion order.
* Set interface contains only methods inherited from Collection interface. It does not have it’s own methods. But, applies restriction on methods so that duplicate elements are always avoided.
* One more good thing about Set interface is that the **stronger contract** between equals() and hashCode() methods. According to this contract, you can compare two set instances of different implementation types (HashSet, TreeSet and LinkedHashSet).
* Two set instances, irrespective of their implementation types, are said to be equal if they contain same elements.

**Note**: Set interface contains only methods inherited from Collection interface. It does not have it’s own methods.

diff bw ArrayList and LinkedList

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because array works at the index basis.
* In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

**Constructors of Java ArrayList**

|  |  |
| --- | --- |
| Constructor | **Description** |
| ArrayList() | It is used to build an empty array list. |
| ArrayList(Collection c) | It is used to build an array list that is initialized with the elements of the collection c. |
| ArrayList(int capacity) | It is used to build an array list that has the specified initial capacity. |

**Methods of Java ArrayList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(Collection c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| void clear() | It is used to remove all of the elements from this list. |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |
| Object[] toArray(Object[] a) | It is used to return an array containing all of the elements in this list in the correct order. |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean addAll(int index, Collection c) | It is used to insert all of the elements in the specified collection into this list, starting at the specified position. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the list's current size. |

* **Java LinkedList class**

Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
* Java LinkedList class can be used as list, stack or queue.

**Doubly Linked List**

In case of doubly linked list, we can add or remove elements from both side.

java LinkedList class using doubly linked list

**Constructors of Java LinkedList**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedList() | It is used to construct an empty list. |
| LinkedList(Collection c) | It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. |

**Methods of Java LinkedList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| void addFirst(Object o) | It is used to insert the given element at the beginning of a list. |
| void addLast(Object o) | It is used to append the given element to the end of a list. |
| int size() | It is used to return the number of elements in a list |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean contains(Object o) | It is used to return true if the list contains a specified element. |
| boolean remove(Object o) | It is used to remove the first occurence of the specified element in a list. |
| Object getFirst() | It is used to return the first element in a list. |
| Object getLast() | It is used to return the last element in a list. |
| int indexOf(Object o) | It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element. |
| int lastIndexOf(Object o) | It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element. |

**Java LinkedList Example**

1. **import** java.util.\*;
2. **public** **class** TestCollection7{
3. **public** **static** **void** main(String args[]){
5. LinkedList<String> al=**new** LinkedList<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Ravi

Vijay

Ravi

Ajay

**Java LinkedList Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** LinkedListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** LinkedList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

**Output**:

Let us C Yashwant Kanetkar BPB 8

Data Communications & Networking Forouzan Mc Graw Hill 4

Operating System Galvin Wiley 6

**Difference between ArrayList and LinkedList**

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

But there are many differences between ArrayList and LinkedList classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

**Example of ArrayList and LinkedList in Java**

Let's see a simple example where we are using ArrayList and LinkedList both.

1. **import** java.util.\*;
2. **class** TestArrayLinked{
3. **public** **static** **void** main(String args[]){
5. List<String> al=**new** ArrayList<String>();//creating arraylist
6. al.add("Ravi");//adding object in arraylist
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. List<String> al2=**new** LinkedList<String>();//creating linkedlist
12. al2.add("James");//adding object in linkedlist
13. al2.add("Serena");
14. al2.add("Swati");
15. al2.add("Junaid");
17. System.out.println("arraylist: "+al);
18. System.out.println("linkedlist: "+al2);
19. }
20. }

**Output**:

arraylist: [Ravi,Vijay,Ravi,Ajay]

linkedlist: [James,Serena,Swati,Junaid]

what is ListIterator Interface?

**Java ListIterator Interface**

ListIterator Interface is used to traverse the element in backward and forward direction.

**Methods of Java ListIterator Interface:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean hasNext() | This method return true if the list iterator has more elements when traversing the list in the forward direction. |
| Object next() | This method return the next element in the list and advances the cursor position. |
| boolean hasPrevious() | This method return true if this list iterator has more elements when traversing the list in the reverse direction. |
| Object previous() | This method return the previous element in the list and moves the cursor position backwards. |

**Example of ListIterator Interface**

1. **import** java.util.\*;
2. **public** **class** TestCollection8{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Amit");
6. al.add("Vijay");
7. al.add("Kumar");
8. al.add(1,"Sachin");
9. System.out.println("element at 2nd position: "+al.get(2));
10. ListIterator<String> itr=al.listIterator();
11. System.out.println("traversing elements in forward direction...");
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. System.out.println("traversing elements in backward direction...");
16. **while**(itr.hasPrevious()){
17. System.out.println(itr.previous());
18. }
19. }
20. }

Output:

element at 2nd position: Vijay

traversing elements in forward direction...

Amit

Sachin

Vijay

Kumar

traversing elements in backward direction...

Kumar

Vijay

Sachin

Amit

**Example of ListIterator Interface: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

**Output**:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

What is HashSet?

**Java HashSet class**

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.

**Constructors of Java HashSet class:**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashSet() | It is used to construct a default HashSet. |
| HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |
| HashSet(int capacity) | It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet. |

**Methods of Java HashSet class:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the elements from this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean add(Object o) | It is used to adds the specified element to this set if it is not already present. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| Object clone() | It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| Iterator iterator() | It is used to return an iterator over the elements in this set. |
| int size() | It is used to return the number of elements in this set. |

**Java HashSet Example**

1. **import** java.util.\*;
2. **class** TestCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**output**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection9)

Ajay

Vijay

Ravi

what is LinkedhashSet??

**Java LinkedHashSet class**

Java LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Contains unique elements only like HashSet.
* Provides all optional set operations, and permits null elements.
* Maintains insertion order.

**Constructors of Java LinkedHashSet class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashSet() | It is used to construct a default HashSet. |
| HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |
| LinkedHashSet(int capacity) | It is used initialize the capacity of the linkedhashset to the given integer value capacity. |
| LinkedHashSet(int capacity, float fillRatio) | It is used to initialize both the capacity and the fill ratio (also called load capacity) of the hash set from its argument. |

Example of LinkedHashSet class:

1. **import** java.util.\*;
2. **class** TestCollection10{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> al=**new** LinkedHashSet<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

[**Output:**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection10)

Ravi

Vijay

Ajay

What is TreeSet

**Java TreeSet class**

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Contains unique elements only like HashSet.
* Access and retrieval times are quiet fast.
* Maintains ascending order.

**Constructors of Java TreeSet class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeSet() | It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set. |
| TreeSet(Collection c) | It is used to build a new tree set that contains the elements of the collection c. |
| TreeSet(Comparator comp) | It is used to construct an empty tree set that will be sorted according to given comparator. |
| TreeSet(SortedSet ss) | It is used to build a TreeSet that contains the elements of the given SortedSet. |

**Methods of Java TreeSet class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean addAll(Collection c) | It is used to add all of the elements in the specified collection to this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| void add(Object o) | It is used to add the specified element to this set if it is not already present. |
| void clear() | It is used to remove all of the elements from this set. |
| Object clone() | It is used to return a shallow copy of this TreeSet instance. |
| Object first() | It is used to return the first (lowest) element currently in this sorted set. |
| Object last() | It is used to return the last (highest) element currently in this sorted set. |
| int size() | It is used to return the number of elements in this set. |

Java TreeSet Example

1. **import** java.util.\*;
2. **class** TestCollection11{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> al=**new** TreeSet<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**output:**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection11)

Output:

Ajay

Ravi

Vijay

What is Map Interface

**Java Map Interface**

A map contains values on the basis of key i.e. key and value pair. Each key and value pair is known as an entry. Map contains only unique keys.

Map is useful if you have to search, update or delete elements on the basis of key.

**Useful methods of Map interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object put(Object key, Object value) | It is used to insert an entry in this map. |
| void putAll(Map map) | It is used to insert the specified map in this map. |
| Object remove(Object key) | It is used to delete an entry for the specified key. |
| Object get(Object key) | It is used to return the value for the specified key. |
| boolean containsKey(Object key) | It is used to search the specified key from this map. |
| Set keySet() | It is used to return the Set view containing all the keys. |
| Set entrySet() | It is used to return the Set view containing all the keys and values. |

**Map.Entry Interface**

Entry is the sub interface of Map. So we will be accessed it by Map.Entry name. It provides methods to get key and value.

**Methods of Map.Entry interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| **Object getKey()** | **It is used to obtain key.** |
| **Object getValue()** | **It is used to obtain value.** |

**Java Map Example: Generic (New Style)**

1. **import java.util.\*;**
2. **class MapInterfaceExample{**
3. **public static void main(String args[]){**
4. **Map<Integer,String> map=new HashMap<Integer,String>();**
5. **map.put(100,"Amit");**
6. **map.put(101,"Vijay");**
7. **map.put(102,"Rahul");**
8. **for(Map.Entry m:map.entrySet()){**
9. **System.out.println(m.getKey()+" "+m.getValue());**
10. **}**
11. **}**
12. **}**

**Output:**

**102 Rahul**

**100 Amit**

**101 Vijay**

**Java Map Example: Non-Generic (Old Style)**

1. **//Non-generic**
2. **import java.util.\*;**
3. **public class MapExample1 {**
4. **public static void main(String[] args) {**
5. **Map map=new HashMap();**
6. **//Adding elements to map**
7. **map.put(1,"Amit");**
8. **map.put(5,"Rahul");**
9. **map.put(2,"Jai");**
10. **map.put(6,"Amit");**
11. **//Traversing Map**
12. **Set set=map.entrySet();//Converting to Set so that we can traverse**
13. **Iterator itr=set.iterator();**
14. **while(itr.hasNext()){**
15. **//Converting to Map.Entry so that we can get key and value separately**
16. **Map.Entry entry=(Map.Entry)itr.next();**
17. **System.out.println(entry.getKey()+" "+entry.getValue());**
18. **}**
19. **}**
20. **}**

**Output:**

**1 Amit**

**2 Jai**

**5 Rahul**

**6 Amit**

what is HashMap

**Java HashMap class**

Java HashMap class implements the map interface by using a hashtable. It inherits AbstractMap class and implements Map interface.

The important points about Java HashMap class are:

* A HashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains no order.

**HashMap class Parameters**

Let's see the Parameters for java.util.HashMap class.

* K: It is the type of keys maintained by this map.
* V: It is the type of mapped values.

**Constructors of Java HashMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashMap() | It is used to construct a default HashMap. |
| HashMap(Map m) | It is used to initializes the hash map by using the elements of the given Map object m. |
| HashMap(int capacity) | It is used to initializes the capacity of the hash map to the given integer value, capacity. |
| HashMap(int capacity, float fillRatio) | It is used to initialize both the capacity and fill ratio of the hash map by using its arguments. |

**Methods of Java HashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the mappings from this map. |
| boolean containsKey(Object key) | It is used to return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It is used to return true if this map maps one or more keys to the specified value. |
| boolean isEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| Object put(Object key, Object value) | It is used to associate the specified value with the specified key in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |
| Collection values() | It is used to return a collection view of the values contained in this map. |

**Java HashMap Example**

1. **import** java.util.\*;
2. **class** TestCollection13{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>();
5. hm.put(100,"Amit");
6. hm.put(101,"Vijay");
7. hm.put(102,"Rahul");
8. **for**(Map.Entry m:hm.entrySet()){
9. System.out.println(m.getKey()+" "+m.getValue());
10. }
11. }
12. }

[**output**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection13)

Output:102 Rahul

100 Amit

101 Vijay

**Java HashMap Example: remove()**

1. **import** java.util.\*;
2. **public** **class** HashMapExample {
3. **public** **static** **void** main(String args[]) {
4. // create and populate hash map
5. HashMap<Integer, String> map = **new** HashMap<Integer, String>();
6. map.put(101,"Let us C");
7. map.put(102, "Operating System");
8. map.put(103, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

**Output:**

Values before remove: {102=Operating System, 103=Data Communication and Networking, 101=Let us C}

Values after remove: {103=Data Communication and Networking, 101=Let us C}

**Difference between HashSet and HashMap**

HashSet contains only values whereas HashMap contains entry(key and value).

**Java HashMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** HashMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(1,b1);
24. map.put(2,b2);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

**Output**:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

What is LinkedHashMap??

**Java LinkedHashMap class**

Java LinkedHashMap class is Hash table and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

The important points about Java LinkedHashMap class are:

* A LinkedHashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead maintains insertion order.

**LinkedHashMap class Parameters**

Let's see the Parameters for java.util.LinkedHashMap class.

* K: It is the type of keys maintained by this map.
* V: It is the type of mapped values.

**Constructors of Java LinkedHashMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedHashMap() | It is used to construct a default LinkedHashMap. |
| LinkedHashMap(int capacity) | It is used to initialize a LinkedHashMap with the given capacity. |
| LinkedHashMap(int capacity, float fillRatio) | It is used to initialize both the capacity and the fillRatio. |
| LinkedHashMap(Map m) | It is used to initialize the LinkedHashMap with the elements from the given Map class m. |

**Methods of Java LinkedHashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object get(Object key) | It is used to return the value to which this map maps the specified key. |
| void clear() | It is used to remove all mappings from this map. |
| boolean containsKey(Object key) | It is used to return true if this map maps one or more keys to the specified value. |

**Java LinkedHashMap Example**

1. **import** java.util.\*;
2. **class** TestCollection14{
3. **public** **static** **void** main(String args[]){
5. LinkedHashMap<Integer,String> hm=**new** LinkedHashMap<Integer,String>();
7. hm.put(100,"Amit");
8. hm.put(101,"Vijay");
9. hm.put(102,"Rahul");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

[**output**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection14)

Output:100 Amit

101 Vijay

102 Rahul

**Java LinkedHashMap Example:remove()**

1. **import** java.util.\*;
2. **public** **class** LinkedHashMapExample {
3. **public** **static** **void** main(String args[]) {
4. // Create and populate linked hash map
5. Map<Integer, String> map = **new** LinkedHashMap<Integer, String>();
6. map.put(101,"Let us C");
7. map.put(102, "Operating System");
8. map.put(103, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

**Output:**

Values before remove: {101=Let us C, 102=Operating System, 103=Data Communication and Networking}

Values after remove: {101=Let us C, 103=Data Communication and Networking}

**Java LinkedHashMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** LinkedHashMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(2,b2);
24. map.put(1,b1);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

**Output:**

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

3 Details:

103 Operating System Galvin Wiley 6

what is TreeMap??

**Java TreeMap class**

Java TreeMap class implements the Map interface by using a tree. It provides an efficient means of storing key/value pairs in sorted order.

The important points about Java TreeMap class are:

* A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* It contains only unique elements.
* It cannot have null key but can have multiple null values.
* It is same as HashMap instead maintains ascending order.

**TreeMap class Parameters**

Let's see the Parameters for java.util.TreeMap class.

* K: It is the type of keys maintained by this map.
* V: It is the type of mapped values.

**Constructors of Java TreeMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeMap() | It is used to construct an empty tree map that will be sorted using the natural order of its key. |
| TreeMap(Comparator comp) | It is used to construct an empty tree-based map that will be sorted using the comparator comp. |
| TreeMap(Map m) | It is used to initialize a tree map with the entries from **m**, which will be sorted using the natural order of the keys. |
| TreeMap(SortedMap sm) | It is used to initialize a tree map with the entries from the SortedMap **sm**, which will be sorted in the same order as **sm.** |

Methods of Java TreeMap class

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean containsKey(Object key) | It is used to return true if this map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It is used to return true if this map maps one or more keys to the specified value. |
| Object firstKey() | It is used to return the first (lowest) key currently in this sorted map. |
| Object get(Object key) | It is used to return the value to which this map maps the specified key. |
| Object lastKey() | It is used to return the last (highest) key currently in this sorted map. |
| Object remove(Object key) | It is used to remove the mapping for this key from this TreeMap if present. |
| void putAll(Map map) | It is used to copy all of the mappings from the specified map to this map. |
| Set entrySet() | It is used to return a set view of the mappings contained in this map. |
| int size() | It is used to return the number of key-value mappings in this map. |
| Collection values() | It is used to return a collection view of the values contained in this map. |

**Java TreeMap Example:**

1. **import** java.util.\*;
2. **class** TestCollection15{
3. **public** **static** **void** main(String args[]){
4. TreeMap<Integer,String> hm=**new** TreeMap<Integer,String>();
5. hm.put(100,"Amit");
6. hm.put(102,"Ravi");
7. hm.put(101,"Vijay");
8. hm.put(103,"Rahul");
9. **for**(Map.Entry m:hm.entrySet()){
10. System.out.println(m.getKey()+" "+m.getValue());
11. }
12. }
13. }

[**OUTPUT**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection15)

**Output**:

100 Amit

101 Vijay

102 Ravi

103 Rahul

**Java TreeMap Example: remove()**

1. **import** java.util.\*;
2. **public** **class** TreeMapExample {
3. **public** **static** **void** main(String args[]) {
4. // Create and populate tree map
5. Map<Integer, String> map = **new** TreeMap<Integer, String>();
6. map.put(102,"Let us C");
7. map.put(103, "Operating System");
8. map.put(101, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

**Output**:

Values before remove: {101=Data Communication and Networking, 102=Let us C, 103=Operating System}

Values after remove: {101=Data Communication and Networking, 103=Operating System}

**What is difference between HashMap and TreeMap?**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1) HashMap can contain one null key. | TreeMap can not contain any null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. |

**Java TreeMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** TreeMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(2,b2);
24. map.put(1,b1);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

**Output:**

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

what is Hashtable??

**Java Hashtable class**

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

The important points about Java Hashtable class are:

* A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* It contains only unique elements.
* It may have not have any null key or value.
* It is synchronized.

**Hashtable class Parameters**

Let's see the Parameters for java.util.Hashtable class.

* K: It is the type of keys maintained by this map.
* V: It is the type of mapped values.

**Constructors of Java Hashtable class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| Hashtable() | It is the default constructor of hash table it instantiates the Hashtable class. |
| Hashtable(int size) | It is used to accept an integer parameter and creates a hash table that has an initial size specified by integer value size. |
| Hashtable(int size, float fillRatio) | It is used to create a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. |

**Methods of Java Hashtable class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to reset the hash table. |
| boolean contains(Object value) | This method return true if some value equal to the value exist within the hash table, else return false. |
| boolean containsValue(Object value) | This method return true if some value equal to the value exists within the hash table, else return false. |
| boolean containsKey(Object key) | This method return true if some key equal to the key exists within the hash table, else return false. |
| boolean isEmpty() | This method return true if the hash table is empty; returns false if it contains at least one key. |
| void rehash() | It is used to increase the size of the hash table and rehashes all of its keys. |
| Object get(Object key) | This method return the object that contains the value associated with the key. |
| Object remove(Object key) | It is used to remove the key and its value. This method return the value associated with the key. |
| int size() | This method return the number of entries in the hash table. |

**Java Hashtable Example**

1. **import** java.util.\*;
2. **class** TestCollection16{
3. **public** **static** **void** main(String args[]){
4. Hashtable<Integer,String> hm=**new** Hashtable<Integer,String>();
6. hm.put(100,"Amit");
7. hm.put(102,"Ravi");
8. hm.put(101,"Vijay");
9. hm.put(103,"Rahul");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

**Output:**

103 Rahul

102 Ravi

101 Vijay

100 Amit

**Java Hashtable Example: remove()**

1. **import** java.util.\*;
2. **public** **class** HashtableExample {
3. **public** **static** **void** main(String args[]) {
4. // create and populate hash table
5. Hashtable<Integer, String> map = **new** Hashtable<Integer, String>();
6. map.put(102,"Let us C");
7. map.put(103, "Operating System");
8. map.put(101, "Data Communication and Networking");
9. System.out.println("Values before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("Values after remove: "+ map);
13. }
14. }

**Output:**

Values before remove: {103=Operating System, 102=Let us C, 101=Data Communication and Networking}

Values after remove: {103=Operating System, 101=Data Communication and Networking}

**Java Hashtable Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** HashtableExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** Hashtable<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(1,b1);
24. map.put(2,b2);
25. map.put(3,b3);
26. //Traversing map
27. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
28. **int** key=entry.getKey();
29. Book b=entry.getValue();
30. System.out.println(key+" Details:");
31. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
32. }
33. }
34. }

**Output:**

3 Details:

103 Operating System Galvin Wiley 6

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

Difference bw HashMap and Hashtable??

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

Difference between ArrayList and Vector??

ArrayList and Vector both implements List interface and maintains insertion order.

But there are many differences between ArrayList and Vector classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses **Iterator** interface to traverse the elements. | Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also. |

**Example of Java ArrayList**

Let's see a simple example where we are using ArrayList to store and traverse the elements.

1. **import** java.util.\*;
2. **class** TestArrayList21{
3. **public** **static** **void** main(String args[]){
5. List<String> al=**new** ArrayList<String>();//creating arraylist
6. al.add("Sonoo");//adding object in arraylist
7. al.add("Michael");
8. al.add("James");
9. al.add("Andy");
10. //traversing elements using Iterator
11. Iterator itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

**Output:**

Sonoo

Michael

James

Andy

**Example of Java Vector**

Let's see a simple example of java Vector class that uses Enumeration interface.

1. **import** java.util.\*;
2. **class** TestVector1{
3. **public** **static** **void** main(String args[]){
4. Vector<String> v=**new** Vector<String>();//creating vector
5. v.add("umesh");//method of Collection
6. v.addElement("irfan");//method of Vector
7. v.addElement("kumar");
8. //traversing elements using Enumeration
9. Enumeration e=v.elements();
10. **while**(e.hasMoreElements()){
11. System.out.println(e.nextElement());
12. }
13. }
14. }

**Output:**

umesh

irfan

kumar

what is Comparable and Comparator??

**Java Comparable interface**

Java Comparable interface is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object). It provide single sorting sequence only i.e. you can sort the elements on based on single data member only.

For example it may be rollno, name, age or anything else.

**compareTo(Object obj) method**

**public int compareTo(Object obj):** is used to compare the current object with the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

**Collections class**

Collections class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But We cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

**Method of Collections class for sorting List elements**

**public void sort(List list):** is used to sort the elements of List. List elements must be of Comparable type.

**Note**: String class and Wrapper classes implements Comparable interface by default. So if you store the objects of string or wrapper classes in list, set or map, it will be Comparable by default.

Java Comparable Example

Let's see the example of Comparable interface that sorts the list elements on the basis of age.

*File: Student.java*

1. **class** Student **implements** Comparable<Student>{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
11. **public** **int** compareTo(Student st){
12. **if**(age==st.age)
13. **return** 0;
14. **else** **if**(age>st.age)
15. **return** 1;
16. **else**
17. **return** -1;
18. }
19. }

*File: TestSort3.java*

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** TestSort3{
4. **public** **static** **void** main(String args[]){
5. ArrayList<Student> al=**new** ArrayList<Student>();
6. al.add(**new** Student(101,"Vijay",23));
7. al.add(**new** Student(106,"Ajay",27));
8. al.add(**new** Student(105,"Jai",21));
10. Collections.sort(al);
11. **for**(Student st:al){
12. System.out.println(st.rollno+" "+st.name+" "+st.age);
13. }
14. }
15. }

**Output:**

105 Jai 21

101 Vijay 23

106 Ajay 27

**Java Comparator interface**

**Java Comparator interface** is used to order the objects of user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequence i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

**compare() method**

public int compare(Object obj1,Object obj2): compares the first object with second object.

**Collections class**

**Collections** class provides static methods for sorting the elements of collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

**Method of Collections class for sorting List elements**

public void sort(List list, Comparator c): is used to sort the elements of List by the given Comparator.

**Java Comparator Example (Non-generic Old Style)**

Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:

1. Student.java
2. AgeComparator.java
3. NameComparator.java
4. Simple.java

**Student.java**

This class contains three fields rollno, name and age and a parameterized constructor.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }

AgeComparator.java

This class defines comparison logic based on the age. If age of first object is greater than the second, we are returning positive value, it can be any one such as 1, 2 , 10 etc. If age of first object is less than the second object, we are returning negative value, it can be any negative value and if age of both objects are equal, we are returning 0.

1. **import** java.util.\*;
2. **class** AgeComparator **implements** Comparator{
3. **public** **int** compare(Object o1,Object o2){
4. Student s1=(Student)o1;
5. Student s2=(Student)o2;
7. **if**(s1.age==s2.age)
8. **return** 0;
9. **else** **if**(s1.age>s2.age)
10. **return** 1;
11. **else**
12. **return** -1;
13. }
14. }

NameComparator.java

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

1. **import** java.util.\*;
2. **class** NameComparator **implements** Comparator{
3. **public** **int** compare(Object o1,Object o2){
4. Student s1=(Student)o1;
5. Student s2=(Student)o2;
7. **return** s1.name.compareTo(s2.name);
8. }
9. }

Simple.java

In this class, we are printing the objects values by sorting on the basis of name and age.

1. **import** java.util.\*;
2. **import** java.io.\*;
4. **class** Simple{
5. **public** **static** **void** main(String args[]){
7. ArrayList al=**new** ArrayList();
8. al.add(**new** Student(101,"Vijay",23));
9. al.add(**new** Student(106,"Ajay",27));
10. al.add(**new** Student(105,"Jai",21));
12. System.out.println("Sorting by Name...");
14. Collections.sort(al,**new** NameComparator());
15. Iterator itr=al.iterator();
16. **while**(itr.hasNext()){
17. Student st=(Student)itr.next();
18. System.out.println(st.rollno+" "+st.name+" "+st.age);
19. }
21. System.out.println("sorting by age...");
23. Collections.sort(al,**new** AgeComparator());
24. Iterator itr2=al.iterator();
25. **while**(itr2.hasNext()){
26. Student st=(Student)itr2.next();
27. System.out.println(st.rollno+" "+st.name+" "+st.age);
28. }

31. }
32. }

Sorting by Name...

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by age...

105 Jai 21

101 Vijay 23

106 Ajay 27

**Java Comparator Example (Generic)**

**Student.java**

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }

**AgeComparator.java**

1. **import** java.util.\*;
2. **class** AgeComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **if**(s1.age==s2.age)
5. **return** 0;
6. **else** **if**(s1.age>s2.age)
7. **return** 1;
8. **else**
9. **return** -1;
10. }
11. }

NameComparator.java

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

1. **import** java.util.\*;
2. **class** NameComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **return** s1.name.compareTo(s2.name);
5. }
6. }

Simple.java

In this class, we are printing the objects values by sorting on the basis of name and age.

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **class** Simple{
4. **public** **static** **void** main(String args[]){
6. ArrayList<Student> al=**new** ArrayList<Student>();
7. al.add(**new** Student(101,"Vijay",23));
8. al.add(**new** Student(106,"Ajay",27));
9. al.add(**new** Student(105,"Jai",21));
11. System.out.println("Sorting by Name...");
13. Collections.sort(al,**new** NameComparator());
14. **for**(Student st: al){
15. System.out.println(st.rollno+" "+st.name+" "+st.age);
16. }
18. System.out.println("sorting by age...");
20. Collections.sort(al,**new** AgeComparator());
21. **for**(Student st: al){
22. System.out.println(st.rollno+" "+st.name+" "+st.age);
23. }
25. }
26. }

**Output:**

Sorting by Name...

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by age...

105 Jai 21

101 Vijay 23

106 Ajay 27

what are the cursors and how many types of cursors are there and for what they are used for??

Diff bw ArrayList and Vector??

ArrayList and Vector both implements List interface and maintains insertion order.

But there are many differences between ArrayList and Vector classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses **Iterator** interface to traverse the elements. | Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also. |

What is String,StringBuffer and StringBuilder Classes

Strings are immutable or not??if not why explain??