Thread life cycle in java

1. NEW state  or Born State

2. RUNNABLE state

3. RUNNING state(This state doesn’t exist)

4. TIMED WAITING state

5. WAITING state

6. BLOCKED state

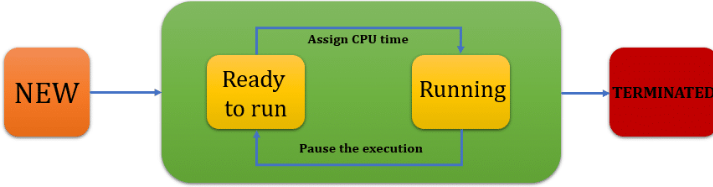
7. TERMINATED state  
  
**1. NEW state**

It is the first state of the thread after the creation of a thread. When we create a thread object by use of the **new operator**, the thread enters in **NEW state**. In this state, the thread is not considered alive. A thread keeps the **NEW state**until **JVM**calls the **start() method**. Once **JVM**calls the **start() method**, the thread leaves the **NEW state** and moves to another state. A thread can’t move back to the **NEW state** after the call of the **start() method**.  
  
  
  
  
**2. RUNNABLE** **state**

When the **JVM** invokes the **start() method** on the thread object, the thread leaves the **NEW state** and moves to the **RUNNABLE state**. Meanwhile, the control transfers to the **Thread scheduler** to finish its execution. The **thread scheduler** decides whether to run the thread instantly or keep it in a runnable **thread pool**before running. Ifthe thread is in the **RUNNABLE state** it means the CPU can run it at any time. When the thread gets a chance to run its states will be changed accordingly. It is the responsibility of the **thread scheduler** to give the chance to thread**.**  
The thread in a **RUNNABLE state** is considered to be **alive** and it can be returned back to this state from the  **WAITING, BLOCKED state**.

**NOTE:**In the **RUNNABLE state** a thread can be either**running**or **ready to run.**

In multi-threaded program can have multiple threads.  The **thread scheduler** allocates a fixed amount of time to each individual thread**. Each thread gets CPU for a short-fixed time and then pauses so that other threads can get a chance to run**. When the execution of a thread is paused by a thread scheduler, the thread is **ready to run**, and it is waiting for the CPU. It means the thread lies in a **runnable state**.

  
  
**3. RUNNING state**

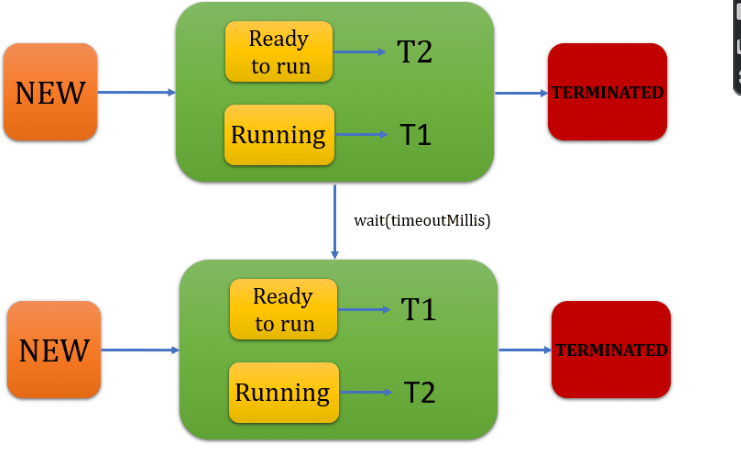
The **RUNNING state** doesn’t exist in reality, but it’s considered a part of the RUNNABLE state. As we know in a **RUNNABLE state**, the control transfers to the **thread scheduler**and the thread scheduler pick one of the thread from the runnable **thread pool** and change its state to Running. It is the state in which the CPU executes the code of the thread.   
**Run method is executing**

**4. TIMED WAITING**

A **RUNNABLE** thread can move to the**TIMED WAITING** state for a specific time interval. Meantime the *CPU picks another thread that* is ready to run and waiting for the CPU cycle. When a thread is in the **TIMED WAITING** **state** it means that the thread lies in this state for a specified time period. It will remain in this state until the given time interval expires, or a notification is received. Any thread can be put in a **TIMED WAITING state** when it calls a method with a time-out parameter.  
There are some methods that are used to move the thread in the **TIMED WAITING state**like **sleep(time)**, **wait(timeout)**, **join(timeout)**, **parkNanos(), parkUntil()**.

 If there are two threads **T1** and **T2. The**T1 thread will take one hour to complete its corresponding job and the **T2**thread will take 10 seconds. Let’s saythe **T1**thread is in the running state and the **T2**thread is in the RUNNABLE state(Ready to run and to wait for the CPU cycle). In this scenario, the **T2**thread must wait 1 hour to complete the execution of 10 seconds. To execute the **T2**thread in between we can call the **waiting(timeOut) method** and move the **T1** thread to the **TIMED WAITING** **state** because **T2** needs to execute only for 10 seconds on priority. Now **T2** will move to the **running state** and get the CPU.

The thread **T1** is in the **TIMED WAITING state** only for 10 seconds. The **T1**thread will be again in the **RUNNABLE**state only if the given time will be expired or any thread gives the notification. It is the responsibility of the thread scheduler to determine which thread should be run.



**5. WAITING state**

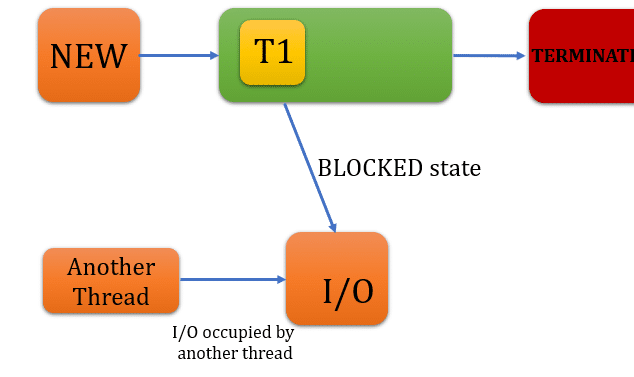
When a thread is in the **WAITING** **state**, it means some other threads are on priority. A thread can be in the **state**for various reasons. We can put a thread in the **WAITING** **state**by calling its **wait() method**, **join() method** or **park() method**.

**For example:**If there are two threads **T1** and **T2. T1**will take one hour to complete their corresponding job and T2 will takes 10 seconds. Let’s say**T1**is in running state and **T2** is in the **RUNNABLE state**(Ready to run and to wait for the CPU cycle). So, we call the waiting() method and move the **T1**thread to the **WAITING** state, because **T2**needs to execute only for 10 seconds on priority. Now T2 moved to the **RUNNABLE state** and get the CPU. The thread **T1** in the **WAITING state** is scheduled by the thread scheduler. Once the thread wait state is over its state is changed to a runnable state and it’s moved back to the thread pool. It is the responsibility of the thread scheduler to determine which thread to run.  
  
**6. BLOCKED state**

When a thread is in the **BLOCKED state** it is not eligible to run. Although the thread is considered to be alive. A **RUNNABLE** thread can move to the **BLOCKED state** if a thread wants to perform some operation but can’t complete it immediately so it must temporarily wait until that task completes. After the **BLOCKED state**, the thread moves to the **RUNNABLE state** and waits for the CPU cycle.

**For example,** A thread is waiting for I/O operations, but some other thread already occupies the I/O operations. So now the thread must wait for I/O operation, and it lies in the **blocked state**. It’s the responsibility of the **thread scheduler** to reactivate and schedule a blocked thread. A blocked thread can’t be executed further until it is moved to a **RUNNABLE state**. The threads which are in **blocked states** do not consume any CPU cycle.

If there are two threads **T1** and **T2. T1**is in the **RUNNABLE** **state**(Ready to run and to wait for the CPU cycle) and **T2**is running by CPU. If T2 (running thread) is moved to the blocked state, then another thread(T1) in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of the thread scheduler to determine which thread to run.

**  
  
 Terminated state**

It is a dead state of the thread. In this state, the thread does not consume any cycles of CPU. A thread terminates because of some reasons:

* The first one is when a thread is finished with its execution completely. It means it completes the execution of the **run() method**.
* Second is when there occurred some unusual erroneous event, like a segmentation fault or an **unhandled exception.**
* Synchronization:

=================

In a multi threaded program often come to a

situation where multiple threads try to access

the same resources and finally produce unexpected result.

Synchronization in java is the capability to control the access of multiple threads to any shared resources.

so it needs to be made sure by some synchronization method that only one thread can access the resources at given point of time.

Synchronization in java is a process where we want to allow only one thread to access the shared resource at a time. So that multiple threads can work together without creating any problems. The synchronization process can achieve by synchronized keyword in java. The Synchronized keyword makes the code thread-safe.

Synchronization Block:

It synchronised on some objects . All Synchronised block synchronized on the same object can only have one thread executing inside them at time.

Why do we use Synchronization in Java?

=======================================

Suppose you have two resources and there may be a chance when multiple threads try to attempt the same resource and produce errors. So synchronization is used to handle the following errors:

1. Thread Interference Error: The thread interference error occurs when we have more than one thread and they are running simultaneously. Because they want to access the same piece of data and perform different operations on the same data. The operation of one thread could overlap with another thread and leads to data inconsistency.

2. Memory Consistency Error: The memory Consistency Error occurs when the changes made by one thread may not be visible to the other threads. The other threads have inconsistent views of the same shared data.

There are two types of Synchronization:

1:Process Synchronization

2:Thread Synchronization

Thread Synchronization:

Types of Synchronization

There are 2 types of synchronization in java as shown below:

1. Process Synchronization: The simultaneous execution of multiple threads or processes to reach a state such that they commit to a certain sequence of actions.

2. Thread Synchronization: Thread Synchronization allows to access the shared space only one thread at a time.

1. Mutual Exclusive: Mutual exclusion is the simplest type of thread synchronization. This synchronization allows only one thread can execute the shared resource at a time. When a thread accesses the shared resource, it takes a lock and all other threads wait to release the lock. Let’s read how we can achieve it. There are two ways

a. Synchronized method.

b. Synchronized block.

2. Cooperation (Inter-thread communication in java): Interthread communication in java is used to avoid polling and is important when you want to develop an application where two or more threads exchange some information. Java provides three methods that are used for inter-thread communication. Those methods are, wait(), notify() and notifyAll() and these methods belong to the object class.

3: Static Synchronization:

If we use static method as synchronized , the lock will be on the class not on object

The Concept of Lock:

===================

Synchronization is built around an internal entity which is known as lock

===========================================================

Important points about java synchronized

1. Synchronized keyword in Java is used to synchronize the shared resource when multiple threads use it. Synchronization in Java is used to remove thread interference and memory inconstancy.

2. Java allows us to use the synchronized keyword with a method or block.

3. The concept of synchronization works based on lock. Whenever a thread enters into a java synchronized method or blocks it acquires a lock and releases the lock after completion of execution.

4. When a thread enters into a non-static synchronized method, it acquires an object-level lock. But when the thread enters into a static synchronized java method it acquires a class-level lock.

5. when a java synchronized method calls another synchronized method then it requires the same lock. But the current thread can enter without a lock because it already holding the lock.

6. Synchronized block throws NullPointerEception, if the object used in the java synchronized block is null.

7. A synchronized method is slow and can degrade performance because threads have to wait for their turn.

8. It is always better to use the Java synchronized block instead of the synchronized method because the using synchronized block only locks critical section of code and avoid locking the whole method.

9. The static synchronized and non-static synchronized method can run simultaneously because they lock on different objects.

===========================================================

synchronized Method

Multiple threads used the shared resources and access the fields and objects reference fields. But it can create a problem with data inconsistency or thread interference. We prevent these errors by use of synchronized keyword in java

To achieve the synchronization in java we can use the java synchronized keyword with the method. A synchronized method is used to ensure that only one thread can execute it at a time. each object has a lock or monitor, so when any thread accesses it, the thread performs certain operations:

1. A thread checks the lock of the synchronized method. If there no other thread is executing it then the lock will be available.

2. If the lock is available, the thread takes the lock and performs the operation as per requirements. Meanwhile, all the other thread waits for the lock.

3. Multiple threads can’t take a lock at the same time.

The synchronized keyword can be used with a static method and a non-static method(instance methods). Here we will see the static and non-static synchronized method in java.

* Lambda Expression:

===================

It is basically instance of functionalInterfaces.

functionalInterfaces: An interface with single abstract method is called functional Interface.

Lambda Expressions implemented the only abstract functional interface.

In java 8 Given more functionalities.

It is a new and important feature of java which included in java SE 8.

1: Treat functionality as method argument.

2: A function that can be created without belonging any class.

Uses Of Lambda Expression:

1: Less Coding.

2: Implementing functional Interface

Syntax:

interfacename reference =(argument-list)->

{

Method Body

};

1: Argument-list: It is optional;

2: ->(Arrow-Token): it is used to link argument-list and body of expression.

3: Method Body:

interfacename reference =(argument-list)-> Statements;

Example

interface abc

{

void disp();

}

public class arraylist

{

public void disp1()

{

abc xy =()->

{

System.out.println("Welcome in java1");

};

xy.disp();

}

public static void main(String args[])

{

abc ab=()->

{

System.out.println("Welcome in java");

};

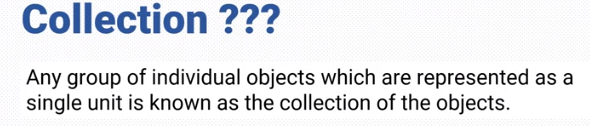
arraylist ll =new arraylist();

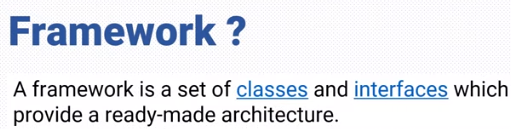
ll.disp1();

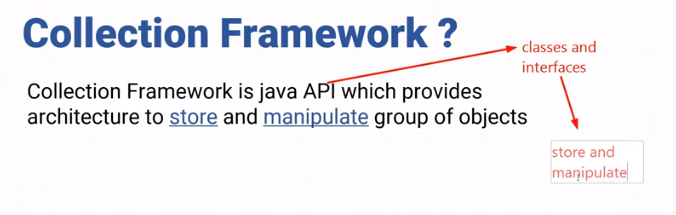
ab.disp();

}

* **Collection Frameworks Java:**





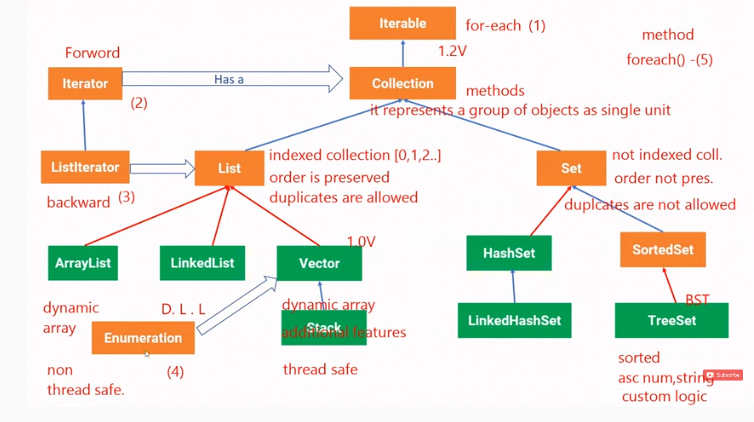
  
  
What Is a Collections Framework?

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

I**nterfaces**: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

**Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

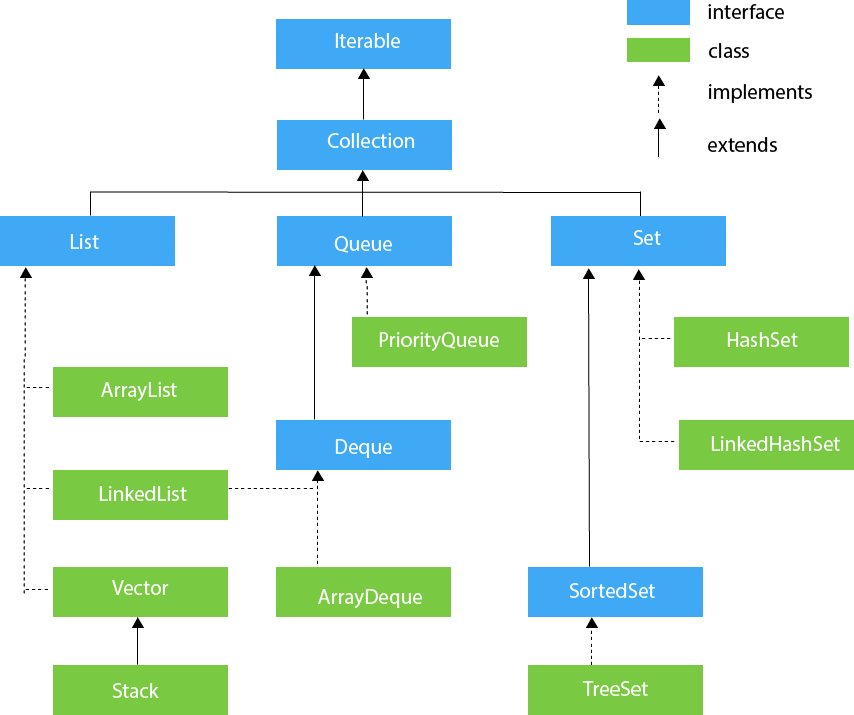
**Algorithms: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces**. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.



The Java Collections API provide Java developers with a set of classes and interfaces that makes it easier to work with collections of objects, e.g. lists, maps, stacks etc.

Rather than having to write your own collection classes, Java provides these ready-to-use collection classes for you. This tutorial will look closer at the Java Collections, as they are also sometimes referred to, and more specifically the Java Collections available in Java 8 and later.

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



|  |  |
| --- | --- |
| **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| **2** | **public boolean addAll(Collection<? extends E> c)** | **It is used to insert the specified collection elements in the invoking collection.** |
| 3 | - | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |
| 20 | Void sort() |  |

Parallel Stream::

   It is a feature of Java 8: It is used for utilizing cores of the processors.

Normally  java code has one stream of processing,  it is executed sequenctially.

 Parallel Stream: we can divide the code into multiple stream that are executed in parallel or separate cores.

In parallel stream order of execution is not controlled.

parallelStream():

Sequential   Stream Execution

Prallel Stream

idle

**Java List**

The *Java List* interface, java.util.List, represents an ordered sequence of objects. The objects contained in a Java List can be inserted, accessed, iterated and removed according to the order in which they appear internally in the Java List. The ordering of the elements is why this data structure is called a *List*.

Each object in a Java List has an index. The first element in the List has index 0, the second element has index 1 etc. The index means "how many elements away from the beginning of the list". The first element is thus 0 elements away from the beginning of the list - because it is at the beginning of the list.

**List vs. Set**

The Java List and **Java Set** interfaces are quite similar in that they both represents a collection of elements. However, there are some significant differences. These differences are reflected in the methods the List and Set interfaces offer.

The first difference between the Java List and Java Set interface is, that the same object can occur more than once in a Java List. This is different from a Java Set where each object can occur only once.

The second difference between a Java List and Java Set interfaces is, that the objects in a List has an order, and the elements can be iterated in that order. A Java Set does not make any promises about the order of the objects kept internally.

**List Implementations**

Being a Collection subtype all methods in the Collection interface are also available in the List interface.

Since List is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following List implementations in the Java Collections API:

* **java.util.ArrayList**
* **java.util.LinkedList**
* **java.util.Vector**
* **java.util.S**tack

Of these implementations, the ArrayList is the most commonly used.

Create a List

You create a List instance by creating an instance of one of the classes that implements the List interface. Here are a few examples of how to create a List instance:

List listA = new ArrayList();

List listB = new LinkedList();

List listC = new Vector();

List listD = new Stack();

**Generic Lists**

By default you can put any Object into a List, but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a List. Here is an example:

List<MyObject> list = new ArrayList<MyObject>();

Iterator Interface :

  It provides the facility of iterating  the elements in forward direction only.

There are three methods :

  1: public Boolean hasNext():  It return true if the iterator has more element otherwise it returns false.

  2: public object next(): it returns the element and moves the cursor pointer to the next element.

Iterator   obj = Al.iterator();

  While(obj. hasNext())

     {

   System.out.println(obj.next());

}

ArrayLiist:

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| boolean | [**add**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#add-E-)([**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) e)  Appends the specified element to the end of this list. |
| void | [**add**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#add-int-E-)(int index, [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) element)  Inserts the specified element at the specified position in this list. |
| boolean | [**addAll**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#addAll-java.util.Collection-)([**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<? extends [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> c)  Appends 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. |
| Boolean- | [**addAll**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#addAll-int-java.util.Collection-)(int index, [**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<? extends [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> c)  Inserts all of the elements in the specified collection into this list, starting at the specified position. |
| void | [**clear**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#clear--)()  Removes all of the elements from this list. |
| [**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) | [**clone**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#clone--)()  Returns a shallow copy of this ArrayList instance. |
| boolean | [**contains**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#contains-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) o)  Returns true if this list contains the specified element. |
| void | [**ensureCapacity**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ensureCapacity-int-)(int minCapacity)  Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
|  |  |
| void | [**forEach**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#forEach-java.util.function.Consumer-)([**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> action)  Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) | [**get**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#get-int-)(int index)  Returns the element at the specified position in this list. |
| int | [**indexOf**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#indexOf-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) o)  Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| boolean | [**isEmpty**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#isEmpty--)()  Returns true if this list contains no elements. |
| [**Iterator**](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)<[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> | [**iterator**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#iterator--)()  Returns an iterator over the elements in this list in proper sequence. |
| int | [**lastIndexOf**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#lastIndexOf-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) o)  Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| [**ListIterator**](https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html)<[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> | [**listIterator**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#listIterator--)()  Returns a list iterator over the elements in this list (in proper sequence). |
| [**ListIterator**](https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html)<[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> | [**listIterator**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#listIterator-int-)(int index)  Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list. |
| [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) | [**remove**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#remove-int-)(int index)  Removes the element at the specified position in this list. |
| boolean | [**remove**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#remove-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) o)  Removes the first occurrence of the specified element from this list, if it is present. |
| boolean | [**removeAll**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#removeAll-java.util.Collection-)([**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<?> c)  Removes from this list all of its elements that are contained in the specified collection. |
| boolean | [**removeIf**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#removeIf-java.util.function.Predicate-)([**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<? super [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> filter)  Removes all of the elements of this collection that satisfy the given predicate. |
| protected void | [**removeRange**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#removeRange-int-int-)(int fromIndex, int toIndex)  Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. |
| void | [**replaceAll**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#replaceAll-java.util.function.UnaryOperator-)(**[UnaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html)**<[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> operator)  Replaces each element of this list with the result of applying the operator to that element. |
| boolean | [**retainAll**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#retainAll-java.util.Collection-)([**Collection**](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html)<?> c)  Retains only the elements in this list that are contained in the specified collection. |
| [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) | [**set**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#set-int-E-)(int index, [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) element)  Replaces the element at the specified position in this list with the specified element. |
| int | [**size**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#size--)()  Returns the number of elements in this list. |
| void | [**sort**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#sort-java.util.Comparator-)([**Comparator**](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html)<? super [**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> c)  Sorts this list according to the order induced by the specified [**Comparator**](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html). |
| [**Spliterator**](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)<[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> | [**spliterator**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#spliterator--)()  Creates a [***late-binding***](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#binding) and *fail-fast* **[Spliterator](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)** over the elements in this list. |
| [**List**](https://docs.oracle.com/javase/8/docs/api/java/util/List.html)<[**E**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)> | [**subList**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#subList-int-int-)(int fromIndex, int toIndex)  Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. |
| [**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)[] | [**toArray**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#toArray--)()  Returns an array containing all of the elements in this list in proper sequence (from first to last element). |
| <T> T[] | [**toArray**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#toArray-T:A-)(T[] a)  Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array. |
| void | [**trimToSize**](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#trimToSize--)()  Trims the capacity of this ArrayList instance to be the list's current size. |

Java LinkedList class uses a 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.

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 occur.

Java LinkedList class can be used as a list, stack or queue.

 Java linkedList class acts as a dynamic array and we donot have  to specify the size while creating it. It dynamically created and removed.

It internally uses the Doubly linked list data structure.

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(E e) | It is used to append the specified element to the end of a list. |
| void add(int index, E element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(Collection<? extends E> 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. |
| boolean addAll(Collection<? extends E> 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. |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| void addFirst(E e) | It is used to insert the given element at the beginning of a list. |
| void addLast(E e) | It is used to append the given element to the end of a list. |
| void clear() | It is used to remove all the elements from a list. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| boolean contains(Object o) | It is used to return true if a list contains a specified element. |
| Iterator<E> descendingIterator() | It is used to return an iterator over the elements in a deque in reverse sequential order. |
| E element() | It is used to retrieve the first element of a list. |
| E get(int index) | It is used to return the element at the specified position in a list. |
| E getFirst() | It is used to return the first element in a list. |
| E 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. |
| ListIterator<E> listIterator(int index) | It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list. |
| boolean offer(E e) | It adds the specified element as the last element of a list. |
| boolean offerFirst(E e) | It inserts the specified element at the front of a list. |
| boolean offerLast(E e) | It inserts the specified element at the end of a list. |
| E peek() | It retrieves the first element of a list |
| E peekFirst() | It retrieves the first element of a list or returns null if a list is empty. |
| E peekLast() | It retrieves the last element of a list or returns null if a list is empty. |
| E poll() | It retrieves and removes the first element of a list. |
| E pollFirst() | It retrieves and removes the first element of a list, or returns null if a list is empty. |
| E pollLast() | It retrieves and removes the last element of a list, or returns null if a list is empty. |
| E pop() | It pops an element from the stack represented by a list. |
| void push(E e) | It pushes an element onto the stack represented by a list. |
| E remove() | It is used to retrieve and removes the first element of a list. |
| E remove(int index) | It is used to remove the element at the specified position in a list. |
| boolean remove(Object o) | It is used to remove the first occurrence of the specified element in a list. |
| E removeFirst() | It removes and returns the first element from a list. |
| boolean removeFirstOccurrence(Object o) | It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail). |
| E removeLast() | It removes and returns the last element from a list. |
| boolean removeLastOccurrence(Object o) | It removes the last occurrence of the specified element in a list (when traversing the list from head to tail). |
| E set(int index, E element) | It replaces the element at the specified position in a list with the specified element. |
| Object[] toArray() | It is used to return an array containing all the elements in a list in proper sequence (from first to the last element). |
| <T> T[] toArray(T[] a) | It returns an array containing all the elements in the proper sequence (from first to the last element); the runtime type of the returned array is that of the specified array. |
| int size() | It is used to return the number of elements in a list. |

**Set Interface**:  A set is collection  that cannot contain  duplicate elements. It models  the mathematical set abstraction. The Set Interface  contains only methods inherited from collection and add the restriction that duplicate elements are prohibited.

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.

he important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Modifier & Type** | **Method** | **Description** |
| 1) | boolean | add(E e) | It is used to add the specified element to this set if it is not already present. |
| 2) | void | clear() | It is used to remove all of the elements from the set. |
| 3) | object | clone() | It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| 4) | boolean | contains(Object o) | It is used to return true if this set contains the specified element. |
| 5) | boolean | isEmpty() | It is used to return true if this set contains no elements. |
| 6) | Iterator<E> | iterator() | It is used to return an iterator over the elements in this set. |
| 7) | boolean | remove(Object o) | It is used to remove the specified element from this set if it is present. |
| 8) | int | size() | It is used to return the number of elements in the set. |
|  |  |  |  |

* **Wrapper Class:**

**Introduction**:

 Java is not fully object oriented programming language.  **The object oriented programming language every data describe as object**. We are creating variables on the help of primitive type which are not object.

 Java is not a purely object-oriented programming language, the reason being it works on primitive data types. These eight primitive data **types int, short, byte, long, float, double, char and, boolean are not objects.**  
  
**Some important points about Wrapper Class.**

* We use wrapper classes to use these data types in the form of objects. **Wrapper class in Java makes the Java code fully object-oriented.**
* Sometimes in the process of development, we come across situations where there is a need for objects instead of primitive data types. To achieve this, Java provides a concept of Wrapper classes.
* Wrapper classes are used for converting **primitive data types into objects,** like **int to Integer**
* **Definition: A Wrapper class in Java is the type of class that provides a mechanism to convert the primitive data types into the objects and vice-versa**.
* When a wrapper class is created, there is a creation of a new field in which we store the primitive data types. The object of the wrapper class wraps or holds its respective primitive data type.

* **The process of converting primitive data types into an object is called Autoboxing**. While using a wrapper class, you just have to pass the value of the primitive data type to the constructor of the Wrapper class.

* All the wrapper classes **Byte, Short, Integer, Long, Double and, Float**, are subclasses of the abstract **class Number**. While Character **and Boolean** wrapper classes are the subclasses of class Objec t

* **Why don't Java wrapper classes have no-arg constructors?**

Wrapper objects are immutable. This means that once a wrapper object has a value assigned to it, that value cannot be changed. It doesn't make much sense to have a default value for an object whose value can't be changed. You wouldn't *want* to get a new Instance() of a wrapper class, because then you'd be stuck with the default value

**Need for Wrapper class in Jav**a  
  
Wrapper classes are used to provide a mechanism to ‘wrap’ or bind the values of primitive data types into an object. This helps primitives types act like objects and do the activities reserved for objects like we can add these converted types to the collections like **ArrayList, HashSet, HashMap, etc**.

Wrapper classes are also used to provide a variety of utility functions for primitives data types like converting primitive types to string objects and vice-versa, converting to various bases like binary, octal or hexadecimal, or comparing various objects.

We can not provide null values to Primitive types but wrapper classes can be null. So wrapper classes can be used in such cases we want to assign a null value to primitive data types.

**Advantages of using Wrapper class in Java**

**1. Serialization:** In Serialization, We need to convert the objects into streams. If we have a primitive value and we want to serialize them then we can do this by converting them with the help of wrapper classes.

**2. Synchronization:** In Multithreading, Java synchronization works with objects.

**3. java.util package:** The package java.util provides many utility classes to deal with objects rather than values.

**4. Collection Framework:** The Collection Framework in Java works only with objects. All classes of the collection framework like ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc, work only with objects.

**5. Changing the value inside a Method:** So, if we pass a primitive value using call by value, it will not change the original value. But, it will change the original value if we convert the primitive value into an object.

**6. Polymorphism:** Wrapper classes also help in achieving Polymorphism in Java.  
  
  
Autoboxing and Unboxing in Java

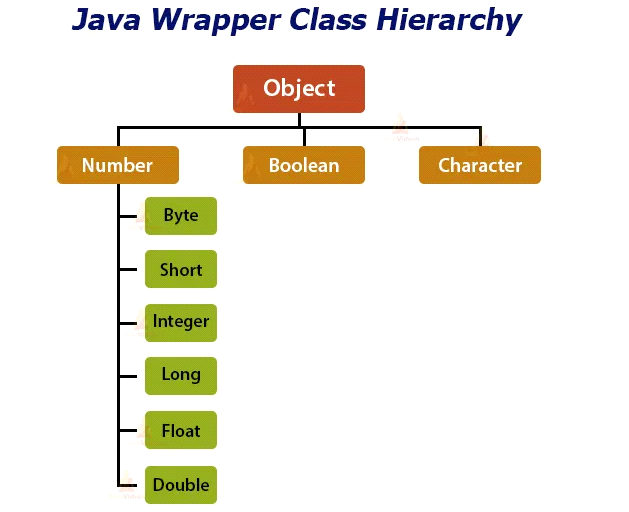
**1. Autoboxing**

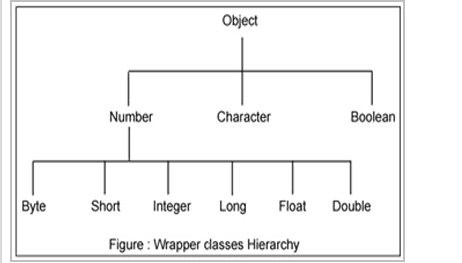
The process to automatically convert the primitive data types into corresponding wrapper class objects is called Autoboxing in Java. This is Autoboxing because this is done automatically by the Java compiler.

For example, char to Character, int to Integer, long to Long, double to Double, float to Float, boolean to Boolean, byte to Byte, and short to Short.

**2. Unboxing**

Java Unboxing is the reverse process of Autoboxing. The process to convert the wrapper class object into its corresponding primitive data type is called Java Unboxing.





|  |  |
| --- | --- |
| **Primitive Type** | **Wrapper class** |
| boolean | Boolean |
| char | Character |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

public abstract class Number  extends Object  implements Serializable

Java Integer Class

The Java Integer class comes under the Java.lang.Number package. This class wraps a value of the primitive type int in an object. An object of Integer class contains a single field of type int value.

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Modifier & Type** | **Method** | **Discription** |
|  |  |  |  |
|  | byte | byteValue() | It converts the given number into a primitive byte type and returns the value of integer object as byte. |
|  | Static int | compare() | It compares two int values numerically and returns the result in integer equivalent. |
|  | Int | compareTo() | It compares two integer objects numerically and returns the result as -1, 0 or 1. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | double | doubleValue() | It converts the given Integer value and returns the result as a double equivalent. |
|  | boolean | equals() | It compares the value of the parameter to the value of the current Integer object and returns boolean ( True or False ). |
|  | float | floatValue() | It converts the given Integer value and returns the result as a float equivalent. |
|  |  |  |  |
|  | int | intValue() | It returns the value of the specified number as an int. |
|  | long | [longValue HYPERLINK "https://www.javatpoint.com/java-integer-longvalue-method"()](https://www.javatpoint.com/java-integer-longvalue-method) | It returns the value of the specified long object as long equivalent. |
|  | static int | parseInt() | It parses the String argument as a signed decimal Integer object. |
|  |  |  |  |
|  |  |  |  |
|  | short | shortValue() | It returns the value of this Integer as a short type after a primitive conversion. |
|  | static String | toBinaryString() | It returns a string representation of the integer argument as an unsigned integer in binary base 2. |
|  | static String | toHexString() | It returns a string representation of the integer argument as an unsigned integer in binary base 16. |
|  | static String | toOctalString() | It returns a string representation of the integer argument as an unsigned integer in binary base 8. |
|  | static Integer | valueOf() | It returns the relevant Integer Object holding the value of the argument passed. |
|  |  |  |  |

static int remainderUnsigned(int dividend, int divisor)

Returns the unsigned remainder from dividing the first argument by the second where each argument and the result is interpreted as an unsigned value.

static    int compare(int x, int y)

Compares two int values numerically.  
  
public static int compare(int x, int y)

Compares two int values numerically. The value returned is identical to what would be returned by:

    Integer.valueOf(x).compareTo(Integer.valueOf(y))

Parameters:

x - the first int to compare

y - the second int to compare

Returns:

the value 0 if x == y; a value less than 0 if x < y; and a value greater than 0 if x > y

int compareTo(Integer anotherInteger)

Compares two Integer objects numerically.  
  
**public int compareTo(Integer anotherInteger)**

Compares two Integer objects numerically.

Specified by:

compareTo in interface Comparable<Integer>

Parameters:

anotherInteger - the Integer to be compared.

Returns:

the value 0 if this Integer is equal to the argument Integer; a value less than 0 if this Integer is numerically less than the argument Integer; and a value greater than 0 if this Integer is numerically greater than the argument Integer (signed comparison).  
  
\* Java File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

import java.io.\*;

public class FileDemo {

    public static void main(String[] args) {

        try {

            File file = new File("javaFile123.txt");

            if (file.createNewFile()) {

                System.out.println("New File is created!");

            } else {

                System.out.println("File already exists.");

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}

public boolean createNewFile()

                      throws IOException

Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Note: this method should not be used for file-locking, as the resulting protocol cannot be made to work reliably. The FileLock facility should be used instead.

Returns:

true if the named file does not exist and was successfully created; false if the named file already exists

Throws:

IOException - If an I/O error occurred

SecurityException - If a security manager exists and its SecurityManager.checkWrite(java.lang.String) method denies write access to the file

Since:

1.2

import java.io.File;

import java.io.IOException;

public class CreateFileDemo

{

   public static void main( String[] args )

   {

      try {

    File file = new File("C:\\newfile.txt");

    /\*If file gets created then the createNewFile()

      \* method would return true or if the file is

      \* already present it would return false

      \*/

             boolean fvar = file.createNewFile();

    if (fvar){

          System.out.println("File has been created successfully");

    }

    else{

          System.out.println("File already present at the specified location");

    }

     } catch (IOException e) {

     System.out.println("Exception Occurred:");

        e.printStackTrace();

  }

   }

}

How to read file in Java – BufferedInputStream

In this example we will see how to read a file in Java using FileInputStream and BufferedInputStream. Here are the detailed steps that we have taken in the below code:

1) Created a File instance by providing the full path of the file(which we will read) during File Object creation.

2) Passed the file instance to the FileInputStream which opens a connection to the actual file, the file named by the File object file in the file system.

3) Passed the FileInputStream instance to BufferedInputStream which creates a BufferedInputStream and saves its argument, the input stream in, for later use. An internal buffer array is created and stored in buf using which the read operation gives good performance as the content is readily available in the buffer.

4) Used while loop to read the file. Method available() is used for checking the end of the file as it returns 0 when the pointer reaches to the end of the file. Read the file content using read() method of FileInputStream.

public class ReadFileDemo {

   public static void main(String[] args) {

      //Specify the path of the file here

      File file = new File("C://myfile.txt");

      BufferedInputStream bis = null;

      FileInputStream  fis= null;

      try

      {

          //FileInputStream to read the file

          fis = new FileInputStream(file);

          /\*Passed the FileInputStream to BufferedInputStream

           \*For Fast read using the buffer array.\*/

          bis = new BufferedInputStream(fis);

          /\*available() method of BufferedInputStream

           \* returns 0 when there are no more bytes

           \* present in the file to be read\*/

          while( bis.available() > 0 ){

              System.out.print((char)bis.read());

          }

       }catch(FileNotFoundException fnfe)

        {

            System.out.println("The specified file not found" + fnfe);

        }

        catch(IOException ioe)

        {

            System.out.println("I/O Exception: " + ioe);

        }

        finally

        {

            try{

               if(bis != null && fis!=null)

               {

                  fis.close();

                  bis.close();

               }

             }catch(IOException ioe)

              {

                  System.out.println("Error in InputStream close(): " + ioe);

              }

        }

   }

}

public class BufferedReader

extends Reader

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

 BufferedReader in

   = new BufferedReader(new FileReader("foo.in"));

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.

Since:

How to read file in Java using BufferedReader

 Using readLine() method of BufferedReader class.

public String readLine() throws IOException

It reads a line of text.

Method 2: Using read() method

public int read() throws IOException

It reads a character of text. Since it returns an integer value, it needs to be explicitly cast as char for reading the content of file.

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class ReadFileDemo {

   public static void main(String[] args) {

       BufferedReader br = null;

       BufferedReader br2 = null;

       try{

           br = new BufferedReader(new FileReader("B:\\myfile.txt"));

           //One way of reading the file

  System.out.println("Reading the file using readLine() method:");

  String contentLine = br.readLine();

  while (contentLine != null) {

      System.out.println(contentLine);

      contentLine = br.readLine();

  }

  br2 = new BufferedReader(new FileReader("B:\\myfile2.txt"));

  //Second way of reading the file

  System.out.println("Reading the file using read() method:");

  int num=0;

  char ch;

  while((num=br2.read()) != -1)

  {

               ch=(char)num;

      System.out.print(ch);

  }

       }

       catch (IOException ioe)

       {

  ioe.printStackTrace();

       }

       finally

       {

  try {

      if (br != null)

br.close();

      if (br2 != null)

br2.close();

  }

  catch (IOException ioe)

           {

System.out.println("Error in closing the BufferedReader");

  }

}

   }

}

How to write to a file in java using FileOutputStream

public class FileOutputStream

extends OutputStream

A file output stream is an output stream for writing data to a File or to a FileDescriptor. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one FileOutputStream (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.

FileOutputStream is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using FileWriter.

Since:

JDK1.0

public void write(byte[] b) throws IOException

It writes b.length bytes from the specified byte array to this file output stream. As you can see this method needs array of bytes in order to write them into a file. Hence we would need to convert our content into array of bytes before writing it into the file.

import java.io.File;

import java.io.FileOutputStream;

import java.io.IOException;

public class WriteFileDemo {

   public static void main(String[] args) {

      FileOutputStream fos = null;

      File file;

      String mycontent = "This is my Data which needs" +

    " to be written into the file";

      try {

          //Specify the file path here

  file = new File("C:/myfile.txt");

  fos = new FileOutputStream(file);

          /\* This logic will check whether the file

  \* exists or not. If the file is not found

  \* at the specified location it would create

  \* a new file\*/

  if (!file.exists()) {

    file.createNewFile();

  }

  /\*String content cannot be directly written into

  \* a file. It needs to be converted into bytes

  \*/

  byte[] bytesArray = mycontent.getBytes();

  fos.write(bytesArray);

  fos.flush();

  System.out.println("File Written Successfully");

       }

       catch (IOException ioe) {

  ioe.printStackTrace();

       }

       finally {

  try {

    if (fos != null)

    {

fos.close();

    }

          }

  catch (IOException ioe) {

    System.out.println("Error in closing the Stream");

  }

       }

   }

}

How to write to file in Java using BufferedWriter

public class FileWriter

extends OutputStreamWriter

Convenience class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.

Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one FileWriter (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.

FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.

Since:

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

public class WriteFileDemo {

   public static void main(String[] args) {

      BufferedWriter bw = null;

      try {

String mycontent = "This String would be written" +

    " to the specified File";

         //Specify the file name and path here

File file = new File("C:/myfile.txt");

/\* This logic will make sure that the file

  \* gets created if it is not present at the

  \* specified location\*/

  if (!file.exists()) {

    file.createNewFile();

  }

  FileWriter fw = new FileWriter(file);

  bw = new BufferedWriter(fw);

  bw.write(mycontent);

          System.out.println("File written Successfully");

      } catch (IOException ioe) {

  ioe.printStackTrace();

}

finally

{

  try{

      if(bw!=null)

bw.close();

  }catch(Exception ex){

      System.out.println("Error in closing the BufferedWriter"+ex);

    }

}

   }

}

How to delete file in Java – delete() Method

import java.io.File;

public class DeleteFileJavaDemo

{

   public static void main(String[] args)

   {

      try{

         //Specify the file name and path

     File file = new File("C:\\myfile.txt");

         /\*the delete() method returns true if the file is

          \* deleted successfully else it returns false

          \*/

     if(file.delete()){

         System.out.println(file.getName() + " is deleted!");

         }else{

         System.out.println("Delete failed: File didn't delete");

       }

       }catch(Exception e){

           System.out.println("Exception occurred");

       e.printStackTrace();

     }

    }

}

\*\* File handling in Java is defined as reading and writing data to a file. The particular file class from the package called java.io allows us to handle and work with different formats of files. Thus,  if we want to use a file class, we need to create an object of that particular class and should specify the filename or directory name.

n Java, a File is an abstract data type. A named location used to store related information is known as a File. There are several File Operations like creating a new File, getting information about File, writing into a File, reading from a File and deleting a File.

.No. Method Return Type Description

1. canRead() Boolean The canRead() method is used to check whether we can read the data of the file or not.

2. createNewFile() Boolean The createNewFile() method is used to create a new empty file.

3. canWrite() Boolean The canWrite() method is used to check whether we can write the data into the file or not.

4. exists() Boolean The exists() method is used to check whether the specified file is present or not.

5. delete() Boolean The delete() method is used to delete a file.

6. getName() String The getName() method is used to find the file name.

7. getAbsolutePath() String The getAbsolutePath() method is used to get the absolute pathname of the file.

8. length() Long The length() method is used to get the size of the file in bytes.

9. list() String[] The list() method is used to get an array of the files available in the directory.

10. mkdir() Boolean The mkdir() method is used for creating a new directory.

File Operations

We can perform the following operation on a file:

Create a File

Get File Information

Write to a File

Read from a File

Delete a File

Create a File

Create a File operation is performed to create a new file. We use the createNewFile() method of file. The createNewFile() method returns true when it successfully creates a new file and returns false when the file already exists.

// Importing File class

import java.io.File;

// Importing the IOException class for handling errors

import java.io.IOException;

 class CreateFile {

               public static void main(String args[])

               {

               try {

                       // Creating an object of a file

                       File fl = new File("D:\\FileOperationExample.txt");

                       if (fl.createNewFile()) {

                                  System.out.println("File " + fl.getName() + " is created successfully.");

                       } else {

                                  System.out.println("File is already exist in the directory.");

                       }

                     } catch (IOException exception) {

                              System.out.println("An unexpected error is occurred.");

                              exception.printStackTrace();

                  }

        }

}

Get File Information

The operation is performed to get the file information. We use several methods to get the information about the file like name, absolute path, is readable, is writable and length.

package aug2;

//Import the File class

import java.io.File;

class FileInfo {

 public static void main(String[] args) {

     // Creating file object

     File f0 = new File("D:\\FileOperationExample.txt");

     if (f0.exists()) {

         // Getting file name

         System.out.println("The name of the file is: " + f0.getName());

         // Getting path of the file

         System.out.println("The absolute path of the file is: " + f0.getAbsolutePath());

         // Checking whether the file is writable or not

         System.out.println("Is file writeable?: " + f0.canWrite());

         // Checking whether the file is readable or not

         System.out.println("Is file readable " + f0.canRead());

         // Getting the length of the file in bytes

         System.out.println("The size of the file in bytes is: " + f0.length());

     } else {

         System.out.println("The file does not exist.");

     }

 }

}

Write to a File

===============

The next operation which we can perform on a file is "writing into a file". In order to write data into a file, we will use the FileWriter class and its write() method together. We need to close the stream using the close() method to retrieve the allocated resources.

//Importing the FileWriter class

import java.io.FileWriter;

//Importing the IOException class for handling errors

import java.io.IOException;

class WriteToFile {

 public static void main(String[] args) {

 try {

     FileWriter fwrite = new FileWriter("D:\\FileOperationExample.txt");

     // writing the content into the FileOperationExample.txt file

     fwrite.write("A named location used to store related information is referred to as a File.");

     // Closing the stream

     fwrite.close();

     System.out.println("Content is successfully wrote to the file.");

 } catch (IOException e) {

     System.out.println("Unexpected error occurred");

     e.printStackTrace();

     }

 }

}

Read from a File

The next operation which we can perform on a file is "read from a file". In order to write data into a file, we will use the Scanner class. Here, we need to close the stream using the close() method. We will create an instance of the Scanner class and use the hasNextLine() method nextLine() method to get data from the file.

//Importing the File class

import java.io.File;

//Importing FileNotFoundException class for handling errors

import java.io.FileNotFoundException;

//Importing the Scanner class for reading text files

import java.util.Scanner;

class ReadFromFile {

 public static void main(String[] args) {

     try {

         // Create f1 object of the file to read data

         File f1 = new File("D:\\FileOperationExample.txt");

         Scanner dataReader = new Scanner(f1);

         while (dataReader.hasNextLine()) {

             String fileData = dataReader.nextLine();

             System.out.println(fileData);

         }

         dataReader.close();

     } catch (FileNotFoundException exception) {

         System.out.println("Unexcpected error occurred!");

         exception.printStackTrace();

     }

 }

}

Delete a File

The next operation which we can perform on a file is "deleting a file". In order to delete a file, we will use the delete() method of the file. We don't need to close the stream using the close() method because for deleting a file, we neither use the FileWriter class nor the Scanner class.

// Importing the File class

import java.io.File;

class DeleteFile {

  public static void main(String[] args) {

    File f0 = new File("D:\\FileOperationExample.txt");

    if (f0.delete()) {

      System.out.println(f0.getName()+ " file is deleted successfully.");

    } else {

      System.out.println("Unexpected error found in deletion of the file.");

    }

  }

}