Java Array

Normally, an array is a collection of similar type of elements that have a contiguous memory location.

Java array is an object which contains elements of a similar data type. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in java is index-based, the first element of the array is stored at the 0 index.

Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in Java

1. arrayRefVar=**new** datatype[size];

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1. //Java Program to illustrate how to declare, instantiate, initialize
2. //and traverse the Java array.
3. class Testarray{
4. public static void main(String args[]){
5. int a[]=new int[5];//declaration and instantiation
6. a[0]=10;//initialization
7. a[1]=20;
8. a[2]=70;
9. a[3]=40;
10. a[4]=50;
11. //traversing array
12. for(int i=0;i<a.length;i++)//length is the property of array
13. System.out.println(a[i]);
14. }}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
1. int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
1. //Java Program to illustrate the use of declaration, instantiation
2. //and initialization of Java array in a single line
3. class Testarray1{
4. public static void main(String args[]){
5. int a[]={33,3,4,5};//declaration, instantiation and initialization
6. //printing array
7. for(int i=0;i<a.length;i++)//length is the property of array
8. System.out.println(a[i]);
9. }}
```

Output:

```
33
3
4
5
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in Java

```
1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
1. int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
1. arr[0][0]=1;
2. arr[0][1]=2;
3. arr[0][2]=3;
4. arr[1][0]=4;
5. arr[1][1]=5;
6. arr[1][2]=6;
7. arr[2][0]=7;
8. arr[2][1]=8;
9. arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
1. //Java Program to illustrate the use of multidimensional array
2. class Testarray3{
3. public static void main(String args[]){
4. //declaring and initializing 2D array
5. int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6. //printing 2D array
7. for(int i=0;i<3;i++){
8.  for(int j=0;j<3;j++){
9.    System.out.print(arr[i][j]+" ");
```

```
10. }
11. System.out.println();
12.}
13.}}
```

Output:

```
1 2 3
2 4 5
4 4 5
```

Java - The Vector Class

Vector implements a dynamic array. It is similar to ArrayList, but with two differences −

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

Sr.No.	Constructor & Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10.
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.
3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.
4	**Vector(Collection c)**

	This constructor creates a vector that contains the elements of collection c.

Apart from the methods inherited from its parent classes, Vector defines the following methods –

Sr.No.	Method & Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position in this Vector.
2	**boolean add(Object o)** Appends the specified element to the end of this Vector.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in in the specified Collection into this Vector at the specified position.
5	**void addElement(Object obj)** Adds the specified component to the end of this vector, increasing its size by one.
6	**int capacity()** Returns the current capacity of this vector.
7	**void clear()** Removes all of the elements from this vector.
8	**Object clone()**

	Returns a clone of this vector.
9	**boolean contains(Object elem)** Tests if the specified object is a component in this vector.
10	**boolean containsAll(Collection c)** Returns true if this vector contains all of the elements in the specified Collection.
11	**void copyInto(Object[] anArray)** Copies the components of this vector into the specified array.
12	**Object elementAt(int index)** Returns the component at the specified index.
13	**Enumeration elements()** Returns an enumeration of the components of this vector.
14	**void ensureCapacity(int minCapacity)** Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
15	**boolean equals(Object o)** Compares the specified Object with this vector for equality.
16	**Object firstElement()** Returns the first component (the item at index 0) of this vector.
17	**Object get(int index)** Returns the element at the specified position in this vector.

18	**int hashCode()** Returns the hash code value for this vector.
19	**int indexOf(Object elem)** Searches for the first occurence of the given argument, testing for equality using the equals method.
20	**int indexOf(Object elem, int index)** Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
21	**void insertElementAt(Object obj, int index)** Inserts the specified object as a component in this vector at the specified index.
22	**boolean isEmpty()** Tests if this vector has no components.
23	**Object lastElement()** Returns the last component of the vector.
24	**int lastIndexOf(Object elem)** Returns the index of the last occurrence of the specified object in this vector.
25	**int lastIndexOf(Object elem, int index)** Searches backwards for the specified object, starting from the specified index, and returns an index to it.
26	**Object remove(int index)** Removes the element at the specified position in this vector.

27	**boolean remove(Object o)** Removes the first occurrence of the specified element in this vector, If the vector does not contain the element, it is unchanged.
28	**boolean removeAll(Collection c)** Removes from this vector all of its elements that are contained in the specified Collection.
29	**void removeAllElements()** Removes all components from this vector and sets its size to zero.
30	**boolean removeElement(Object obj)** Removes the first (lowest-indexed) occurrence of the argument from this vector.
31	**void removeElementAt(int index)** removeElementAt(int index).
32	**protected void removeRange(int fromIndex, int toIndex)** Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
33	**boolean retainAll(Collection c)** Retains only the elements in this vector that are contained in the specified Collection.
34	**Object set(int index, Object element)** Replaces the element at the specified position in this vector with the specified element.
35	**void setElementAt(Object obj, int index)** Sets the component at the specified index of this vector to be the

	specified object.
36	**void setSize(int newSize)** Sets the size of this vector.
37	**int size()** Returns the number of components in this vector.
38	**List subList(int fromIndex, int toIndex)** Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
39	**Object[] toArray()** Returns an array containing all of the elements in this vector in the correct order.
40	**Object[] toArray(Object[] a)** Returns an array containing all of the elements in this vector in the correct order; the runtime type of the returned array is that of the specified array.
41	**String toString()** Returns a string representation of this vector, containing the String representation of each element.
42	**void trimToSize()** Trims the capacity of this vector to be the vector's current size.

Example

The following program illustrates several of the methods supported by this collection –

```
import java.util.*;
```

```
publicclassVectorDemo{


publicstaticvoid main(String args[]){

// initial size is 3, increment is 2

Vector v =newVector(3,2);

System.out.println("Initial size: "+ v.size());

System.out.println("Initial capacity: "+ v.capacity());


      v.addElement(newInteger(1));

      v.addElement(newInteger(2));

      v.addElement(newInteger(3));

      v.addElement(newInteger(4));

System.out.println("Capacity after four additions: "+ v.capacity());


      v.addElement(newDouble(5.45));

System.out.println("Current capacity: "+ v.capacity());


      v.addElement(newDouble(6.08));

      v.addElement(newInteger(7));

System.out.println("Current capacity: "+ v.capacity());


      v.addElement(newFloat(9.4));

      v.addElement(newInteger(10));

System.out.println("Current capacity: "+ v.capacity());


      v.addElement(newInteger(11));

      v.addElement(newInteger(12));

System.out.println("First element: "+(Integer)v.firstElement());

System.out.println("Last element: "+(Integer)v.lastElement());
```

```
if(v.contains(newInteger(3)))

System.out.println("Vector contains 3.");



// enumerate the elements in the vector.

Enumeration vEnum = v.elements();

System.out.println("\nElements in vector:");



while(vEnum.hasMoreElements())

System.out.print(vEnum.nextElement()+" ");

System.out.println();

}

}
```

This will produce the following result –

Output

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

Exception Handling in Java

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

What is exception

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is exception handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

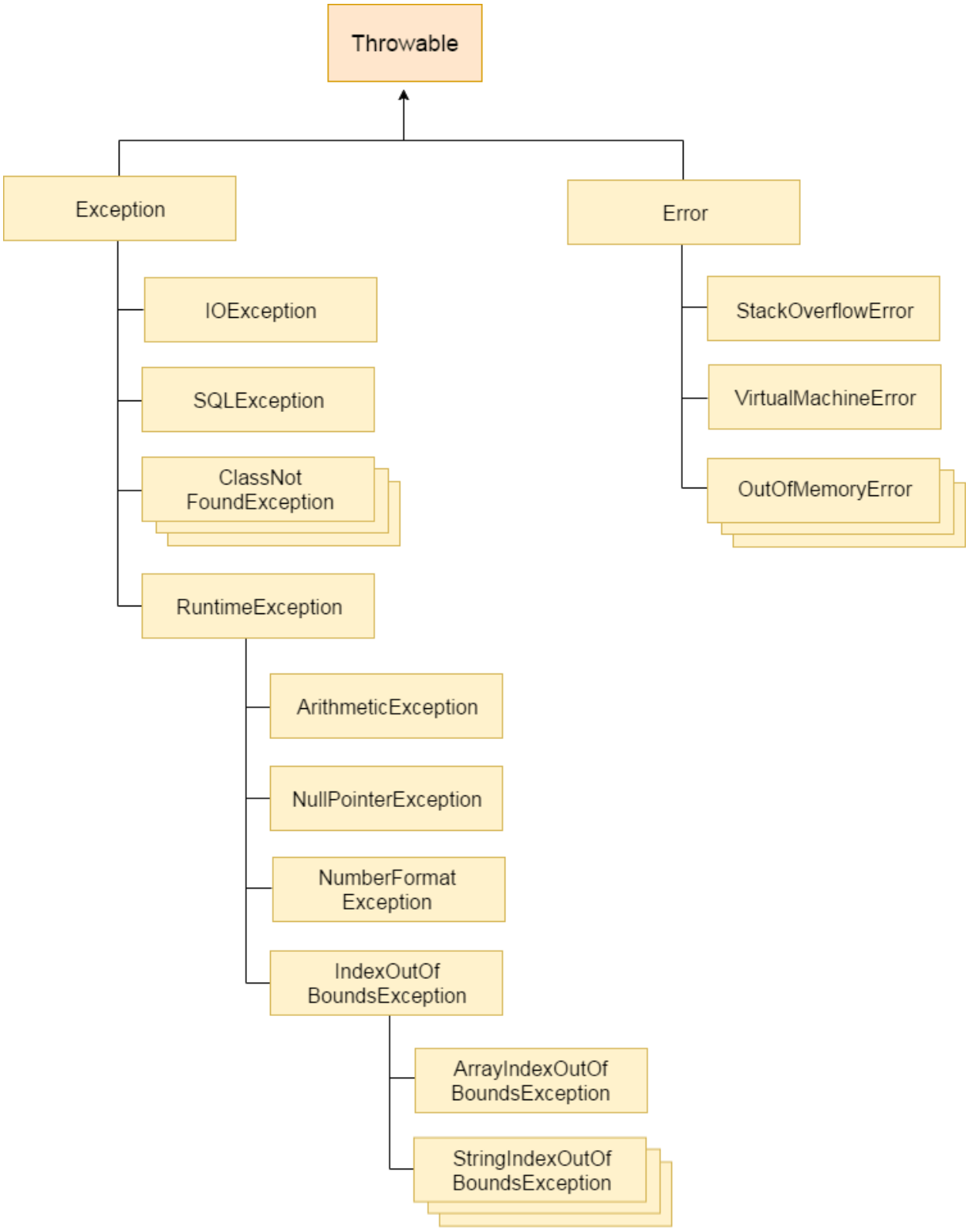
Advantage of Exception Handling

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

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

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

Hierarchy of Java Exception classes



Types of Exception

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

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

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

2) Unchecked Exception

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

3) Error

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

Common scenarios where exceptions may occur

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

1) Scenario where ArithmeticException occurs

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

```
1. int a=50/0;//ArithmeticException
```

---

2) Scenario where NullPointerException occurs

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

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

---

3) Scenario where NumberFormatException occurs

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

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

---

4) Scenario where ArrayIndexOutOfBoundsException occurs

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

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

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

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

Java try-catch

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch

1. **try**{
2. //code that may throw exception
3. }**catch**(Exception_class_Name ref){}

Syntax of try-finally block

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

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
1. public class Testtrycatch1{
2.   public static void main(String args[]){
3.       int data=50/0;//may throw exception
4.       System.out.println("rest of the code...");
5. }
6. }
```

Output:

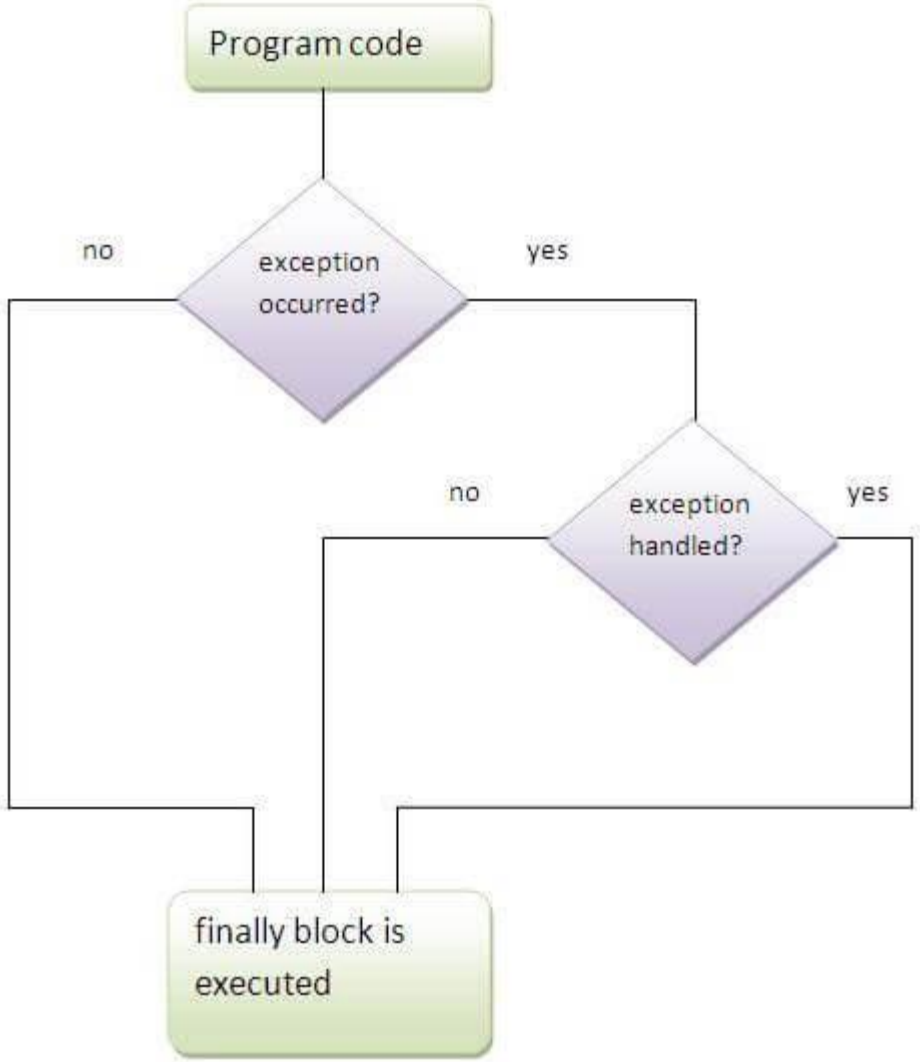
```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



Usage of Java finally

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

Case 1

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

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

```
Output:5
       finally block is always executed
       rest of the code...
```

Java throw exception

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
1. throw exception;
```

Let's see the example of throw IOException.

```
1. throw new IOException("sorry device error);
```

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

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

Output:

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

Java throws keyword

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

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

Syntax of java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
3. }
```

Which exception should be declared

Ans) checked exception only, because:

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

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception

Java throws example

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

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

Output:

```
exception handled
normal flow...
```

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

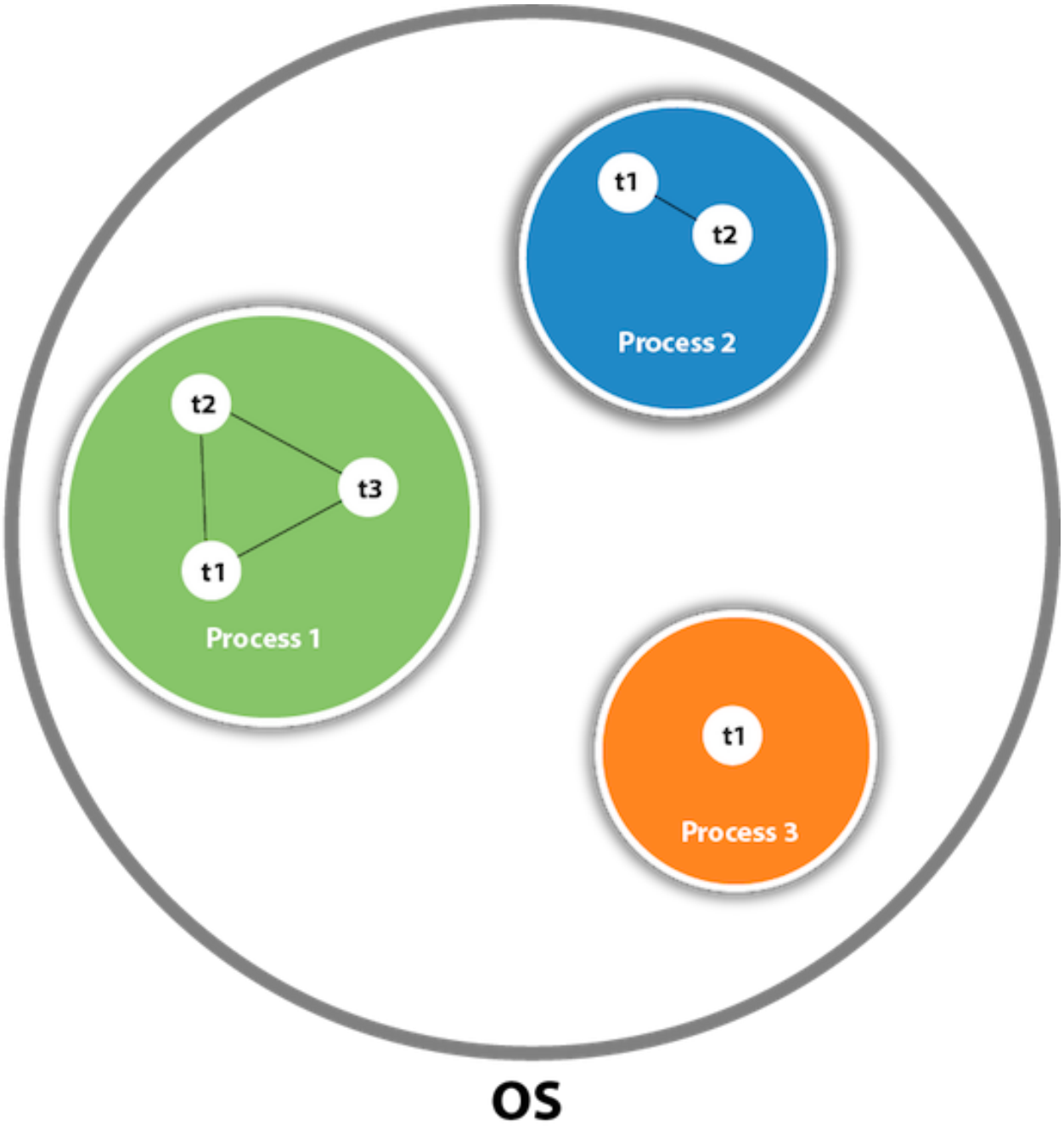
2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



Java Thread class

Thread class is the main class on which java's multithreading system is based. Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Java Thread Methods

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to perform action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to temporarily pause and allow

			other threads to execute.
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.
17)	boolean	isDaemon()	It tests if the thread is a daemon thread.
18)	void	setDaemon()	It marks the thread as daemon or user thread.
19)	void	interrupt()	It interrupts the thread.
20)	boolean	isinterrupted()	It tests whether the thread has been interrupted.
21)	static boolean	interrupted()	It tests whether the current thread has been interrupted.
22)	static int	activeCount()	It returns the number of active threads in the current thread's thread group.
23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.

24)	static boolean	holdLock()	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	dumpStack()	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	getStackTrace()	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	enumerate()	It is used to copy every active thread's thread group and its subgroup into the specified array.
28)	Thread.State	getState()	It is used to return the state of the thread.
29)	ThreadGroup	getThreadGroup()	It is used to return the thread group to which this thread belongs
30)	String	toString()	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	notify()	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	notifyAll()	It is used to give the notification to all waiting threads of a particular object.

33)	void	setContextClassLoader()	It sets the context ClassLoader for the Thread.
34)	ClassLoader	getContextClassLoader()	It returns the context ClassLoader for the thread.
35)	static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler()	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	setDefaultUncaughtExceptionHandler()	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

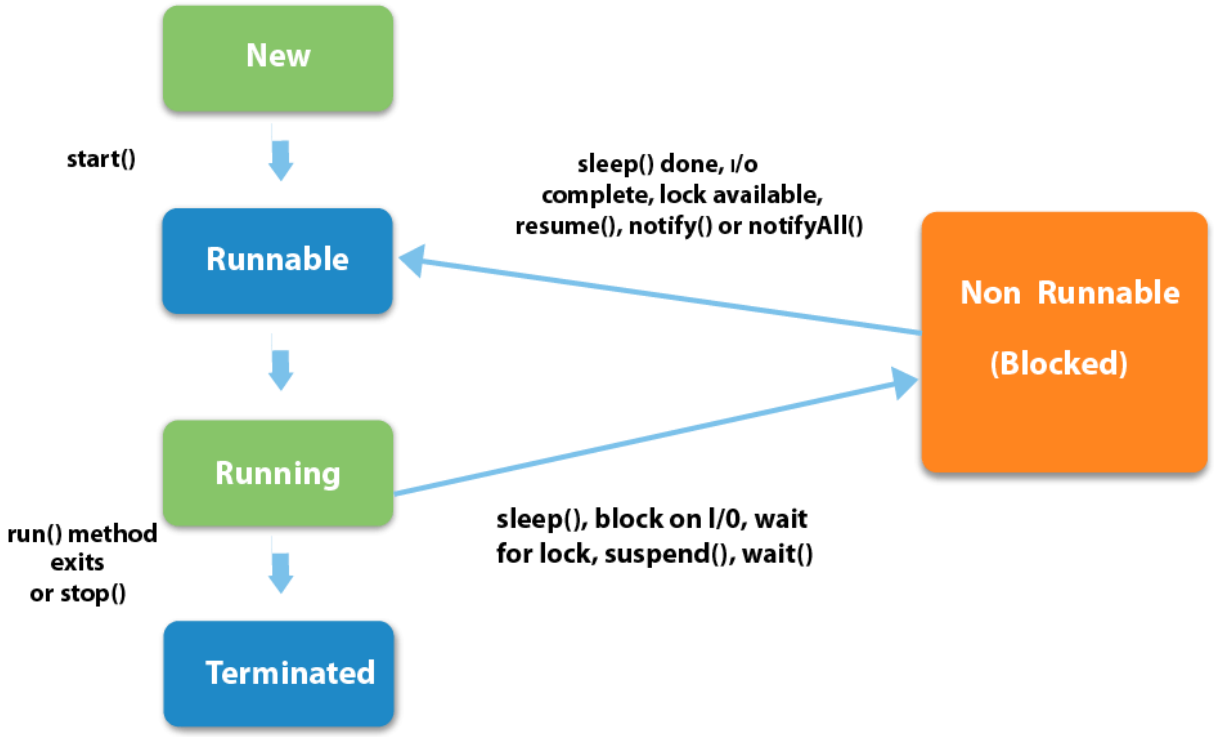
Life cycle of a Thread (Thread States)

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.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

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.

1) Java Thread Example by extending Thread class

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

```
Output:thread is running...
```

Synchronization in Java

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

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

```
4. class Table{
5. void printTable(int n){//method not synchronized
6.     for(int i=1;i<=5;i++){
7.       System.out.println(n*i);
8.       try{
9.        Thread.sleep(400);
10.     }catch(Exception e){System.out.println(e);}
11.   }
12.
13. }
14.}
15.
16.class MyThread1 extends Thread{
17.Table t;
18.MyThread1(Table t){
19.this.t=t;
20.}
21.public void run(){
22.t.printTable(5);
23.}
24.
25.}
26.class MyThread2 extends Thread{
27.Table t;
28.MyThread2(Table t){
29.this.t=t;
30.}
31.public void run(){
32.t.printTable(100);
33.}
```

```
}

class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
        100
        10
        200
        15
        300
        20
        400
        25
        500
```

Java I/O Tutorial

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and an error** message to the console.

```
1. System.out.println("simple message");
2. System.err.println("error message");
```

Let's see the code to get **input** from console.

```
1. int i=System.in.read();//returns ASCII code of 1st character
2. System.out.println((char)i);//will print the character
```

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.

OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws	is used to write a byte to the current output

IOException	stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy

InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy

Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

FileOutputStream class declaration

Let's see the declaration for Java.io.FileOutputStream class:

1. **public class** FileOutputStream **extends** OutputStream

---

FileOutputStream class methods

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write **ary.length** bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

Java FileOutputStream Example 1: write byte

```
1. import java.io.FileOutputStream;
2. public class FileOutputStreamExample {
3.     public static void main(String args[]){
4.           try{
5.             FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
6.             fout.write(65);
7.             fout.close();
8.             System.out.println("success...");
9.            }catch(Exception e){System.out.println(e);}
10.      }
11. }
```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **A**.

testout.txt

```
A
```

Java FileOutputStream example 2: write string

```
1. import java.io.FileOutputStream;
2. public class FileOutputStreamExample {
3.     public static void main(String args[]){
4.           try{
5.             FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
6.             String s="Welcome to javaTpoint.";
7.             byte b[]=s.getBytes();//converting string into byte array
8.             fout.write(b);
9.             fout.close();
10.            System.out.println("success...");
11.           }catch(Exception e){System.out.println(e);}
12.      }
13. }
```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **Welcome to javaTpoint.**

testout.txt

Welcome to javaTpoint.

Java FileInputStream Class

Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

Java FileInputStream class declaration

Let's see the declaration for java.io.FileInputStream class:

1. **public class** FileInputStream **extends** InputStream

Java FileInputStream class methods

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to **b.length** bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to **len** bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel	It is used to return the unique FileChannel object

getChannel()	associated with the file input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the stream.

Java ByteArrayOutputStream Class

Java ByteArrayOutputStream class is used to **write common data** into multiple files. In this stream, the data is written into a byte array which can be written to multiple streams later.

The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.

The buffer of ByteArrayOutputStream automatically grows according to data.

Java ByteArrayOutputStream class declaration

Let's see the declaration for Java.io.ByteArrayOutputStream class:

1. **public class** ByteArrayOutputStream **extends** OutputStream

---

Java ByteArrayOutputStream class constructors

Constructor	Description
ByteArrayOutputStream()	Creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
ByteArrayOutputStream(int	Creates a new byte array output stream, with a

size)	buffer capacity of the specified size, in bytes.

Java ByteArrayOutputStream class methods

Method	Description
int size()	It is used to returns the current size of a buffer.
byte[] toByteArray()	It is used to create a newly allocated byte array.
String toString()	It is used for converting the content into a string decoding bytes using a platform default character set.
String toString(String charsetName)	It is used for converting the content into a string decoding bytes using a specified charsetName.
void write(int b)	It is used for writing the byte specified to the byte array output stream.
void write(byte[] b, int off, int len	It is used for writing **len** bytes from specified byte array starting from the offset **off** to the byte array output stream.
void writeTo(OutputStream out)	It is used for writing the complete content of a byte array output stream to the specified output stream.
void reset()	It is used to reset the count field of a byte array output stream to zero value.
void close()	It is used to close the ByteArrayOutputStream.

Example of Java ByteArrayOutputStream

Let's see a simple example of java ByteArrayOutputStream class to write common data into 2 files: f1.txt and f2.txt.

```
1. package com.javatpoint;
2. import java.io.*;
3. public class DataStreamExample {
4. public static void main(String args[])throws Exception{
5.       FileOutputStream fout1=new FileOutputStream("D:\\f1.txt");
6.       FileOutputStream fout2=new FileOutputStream("D:\\f2.txt");
7.
8.       ByteArrayOutputStream bout=new ByteArrayOutputStream();
9.       bout.write(65);
10.      bout.writeTo(fout1);
11.      bout.writeTo(fout2);
12.
13.      bout.flush();
14.      bout.close();//has no effect
15.      System.out.println("Success...");
16.     }
17.    }
```

Output:

```
Success...
```

f1.txt:

```
A
```

f2.txt:

```
A
```

Java ByteArrayInputStream Class

The ByteArrayInputStream is composed of two words: ByteArray and InputStream. As the name suggests, it can be used to read byte array as input stream.

Java ByteArrayInputStream class contains an internal buffer which is used to **read byte array** as stream. In this stream, the data is read from a byte array.

The buffer of ByteArrayInputStream automatically grows according to data.

Java ByteArrayInputStream class declaration

Let's see the declaration for Java.io.ByteArrayInputStream class:

1. **public class** ByteArrayInputStream **extends** InputStream

---

Java ByteArrayInputStream class constructors

Constructor	Description
ByteArrayInputStream(byte[] ary)	Creates a new byte array input stream which uses **ary** as its buffer array.
ByteArrayInputStream(byte[] ary, int offset, int len)	Creates a new byte array input stream which uses **ary** as its buffer array that can read up to specified **len** bytes of data from an array.

Java ByteArrayInputStream class methods

Methods	Description
int available()	It is used to return the number of remaining bytes that can be read from the input stream.
int read()	It is used to read the next byte of data from the input stream.
int read(byte[] ary, int off, int len)	It is used to read up to len bytes of data from an array of bytes in the input stream.
boolean markSupported()	It is used to test the input stream for mark and reset method.
long skip(long x)	It is used to skip the x bytes of input from the input stream.
void mark(int	It is used to set the current marked position in the

readAheadLimit)	stream.
void reset()	It is used to reset the buffer of a byte array.
void close()	It is used for closing a ByteArrayInputStream.

Example of Java ByteArrayInputStream

Let's see a simple example of java ByteArrayInputStream class to read byte array as input stream.

```
1. package com.javatpoint;
2. import java.io.*;
3. public class ReadExample {
4.   public static void main(String[] args) throws IOException {
5.     byte[] buf = { 35, 36, 37, 38 };
6.     // Create the new byte array input stream
7.     ByteArrayInputStream byt = new ByteArrayInputStream(buf);
8.     int k = 0;
9.     while ((k = byt.read()) != -1) {
10.      //Conversion of a byte into character
11.      char ch = (char) k;
12.      System.out.println("ASCII value of Character is:" + k + "; Special character is: " + ch);
13.    }
14.  }
15. }
```

Output:

```
ASCII value of Character is:35; Special character is: #
ASCII value of Character is:36; Special character is: $
ASCII value of Character is:37; Special character is: %
ASCII value of Character is:38; Special character is: &
```

Java CharArrayReader Class

The CharArrayReader is composed of two words: CharArray and Reader. The CharArrayReader class is used to read character array as a reader (stream). It inherits Reader class.

Java CharArrayReader class declaration

Let's see the declaration for Java.io.CharArrayReader class:

```
1. public class CharArrayReader extends Reader
```

Java CharArrayReader class methods

Method	Description
int read()	It is used to read a single character
int read(char[] b, int off, int len)	It is used to read characters into the portion of an array.
boolean ready()	It is used to tell whether the stream is ready to read.
boolean markSupported()	It is used to tell whether the stream supports mark() operation.
long skip(long n)	It is used to skip the character in the input stream.
void mark(int readAheadLimit)	It is used to mark the present position in the stream.
void reset()	It is used to reset the stream to a most recent mark.
void close()	It is used to closes the stream.

Example of CharArrayReader Class:

Let's see the simple example to read a character using Java CharArrayReader class.

```
1. package com.javatpoint;
2.
```

```
3. import java.io.CharArrayReader;
4. public class CharArrayExample{
5.   public static void main(String[] ag) throws Exception {
6.     char[] ary = { 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't' };
7.     CharArrayReader reader = new CharArrayReader(ary);
8.     int k = 0;
9.     // Read until the end of a file
10.    while ((k = reader.read()) != -1) {
11.      char ch = (char) k;
12.      System.out.print(ch + " : ");
13.      System.out.println(k);
14.    }
15.  }
16. }
```

Output

```
j : 106
a : 97
v : 118
a : 97
t : 116
p : 112
o : 111
i : 105
n : 110
t : 116
```

Java CharArrayWriter Class

The CharArrayWriter class can be used to write common data to multiple files. This class inherits Writerclass. Its buffer automatically grows when data is written in this stream. Calling the close() method on this object has no effect.

Java CharArrayWriter class declaration

Let's see the declaration for Java.io.CharArrayWriter class:

```
1. public class CharArrayWriter extends Writer
```

Java CharArrayWriter class Methods

Method	Description
int size()	It is used to return the current size of the buffer.
char[] toCharArray()	It is used to return the copy of an input data.
String toString()	It is used for converting an input data to a string.
CharArrayWriter append(char c)	It is used to append the specified character to the writer.
CharArrayWriter append(CharSequence csq)	It is used to append the specified character sequence to the writer.
CharArrayWriter append(CharSequence csq, int start, int end)	It is used to append the subsequence of a specified character to the writer.
void write(int c)	It is used to write a character to the buffer.
void write(char[] c, int off, int len)	It is used to write a character to the buffer.
void write(String str, int off, int len)	It is used to write a portion of string to the buffer.
void writeTo(Writer out)	It is used to write the content of buffer to different character stream.
void flush()	It is used to flush the stream.
void reset()	It is used to reset the buffer.

void close()	It is used to close the stream.

Example of CharArrayWriter Class:

In this example, we are writing a common data to 4 files a.txt, b.txt, c.txt and d.txt.

```
1.  package com.javatpoint;
2.
3.  import java.io.CharArrayWriter;
4.  import java.io.FileWriter;
5.  public class CharArrayWriterExample {
6.  public static void main(String args[])throws Exception{
7.          CharArrayWriter out=new CharArrayWriter();
8.          out.write("Welcome to javaTpoint");
9.          FileWriter f1=new FileWriter("D:\\a.txt");
10.         FileWriter f2=new FileWriter("D:\\b.txt");
11.         FileWriter f3=new FileWriter("D:\\c.txt");
12.         FileWriter f4=new FileWriter("D:\\d.txt");
13.         out.writeTo(f1);
14.         out.writeTo(f2);
15.         out.writeTo(f3);
16.         out.writeTo(f4);
17.         f1.close();
18.         f2.close();
19.         f3.close();
20.         f4.close();
21.         System.out.println("Success...");
22.        }
23.       }
```

Output

```
Success...
```

After executing the program, you can see that all files have common data: Welcome to javaTpoint.

a.txt:

```
Welcome to javaTpoint
```

b.txt:

```
Welcome to javaTpoint
```

c.txt:

```
Welcome to javaTpoint
```

d.txt:

```
Welcome to javaTpoint
```

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.

Fields

Modifier	Type	Field	Description
static	String	pathSeparator	It is system-dependent path-separator character, represented as a string for convenience.
static	char	pathSeparatorChar	It is system-dependent path-separator character.
static	String	separator	It is system-dependent default name-separator character, represented as a string for convenience.
static	char	separatorChar	It is system-dependent default name-separator character.

Constructors

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

Useful Methods

Modifier and Type	Method	Description
static File	createTempFile(String prefix, String suffix)	It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can

		read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.
boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
Path	toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
URI	toURI()	It constructs a file: URI that represents this abstract pathname.
File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified

		filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.

Java File Example 1

```
1. import java.io.*;
2. public class FileDemo {
3.     public static void main(String[] args) {
4.
5.         try {
6.             File file = new File("javaFile123.txt");
7.             if (file.createNewFile()) {
8.                 System.out.println("New File is created!");
9.             } else {
10.                System.out.println("File already exists.");
11.            }
12.        } catch (IOException e) {
13.            e.printStackTrace();
14.        }
15.
16.    }
17. }
```

Output:

```
New File is created!
```

Java File Example 2

```
1. import java.io.*;
2. public class FileDemo2 {
3.     public static void main(String[] args) {
4.
5.         String path = "";
6.         boolean bool = false;
7.         try {
8.             // createing  new files
9.             File file  = new File("testFile1.txt");
10.            file.createNewFile();
11.            System.out.println(file);
12.            // createing new canonical from file object
13.            File file2 = file.getCanonicalFile();
14.            // returns true if the file exists
```

```
            System.out.println(file2);
            bool = file2.exists();
            // returns absolute pathname
            path = file2.getAbsolutePath();
            System.out.println(bool);
            // if file exists
            if (bool) {
                // prints
                System.out.print(path + " Exists? " + bool);
            }
        } catch (Exception e) {
            // if any error occurs
            e.printStackTrace();
        }
    }
}
```

Output:

```
testFile1.txt
/home/Work/Project/File/testFile1.txt
true
/home/Work/Project/File/testFile1.txt Exists? true
```

Java File Example 3

```
import java.io.*;
public class FileExample {
public static void main(String[] args) {
    File f=new File("/Users/sonoojaiswal/Documents");
    String filenames[]=f.list();
    for(String filename:filenames){
        System.out.println(filename);
    }
}
}
```

Output:

```
"info.properties"
"info.properties".rtf
.DS_Store
.localized
Alok news
apache-tomcat-9.0.0.M19
apache-tomcat-9.0.0.M19.tar
```

```
bestreturn_org.rtf
BIODATA.pages
BIODATA.pdf
BIODATA.png
struts2jars.zip
workspace
```

Java File Example 4

```
1. import java.io.*;
2. public class FileExample {
3. public static void main(String[] args) {
4.     File dir=new File("/Users/sonoojaiswal/Documents");
5.     File files[]=dir.listFiles();
6.     for(File file:files){
7.         System.out.println(file.getName()+" Can Write: "+file.canWrite()+"
8.         Is Hidden: "+file.isHidden()+" Length: "+file.length()+" bytes");
9.     }
10. }
11. }
```

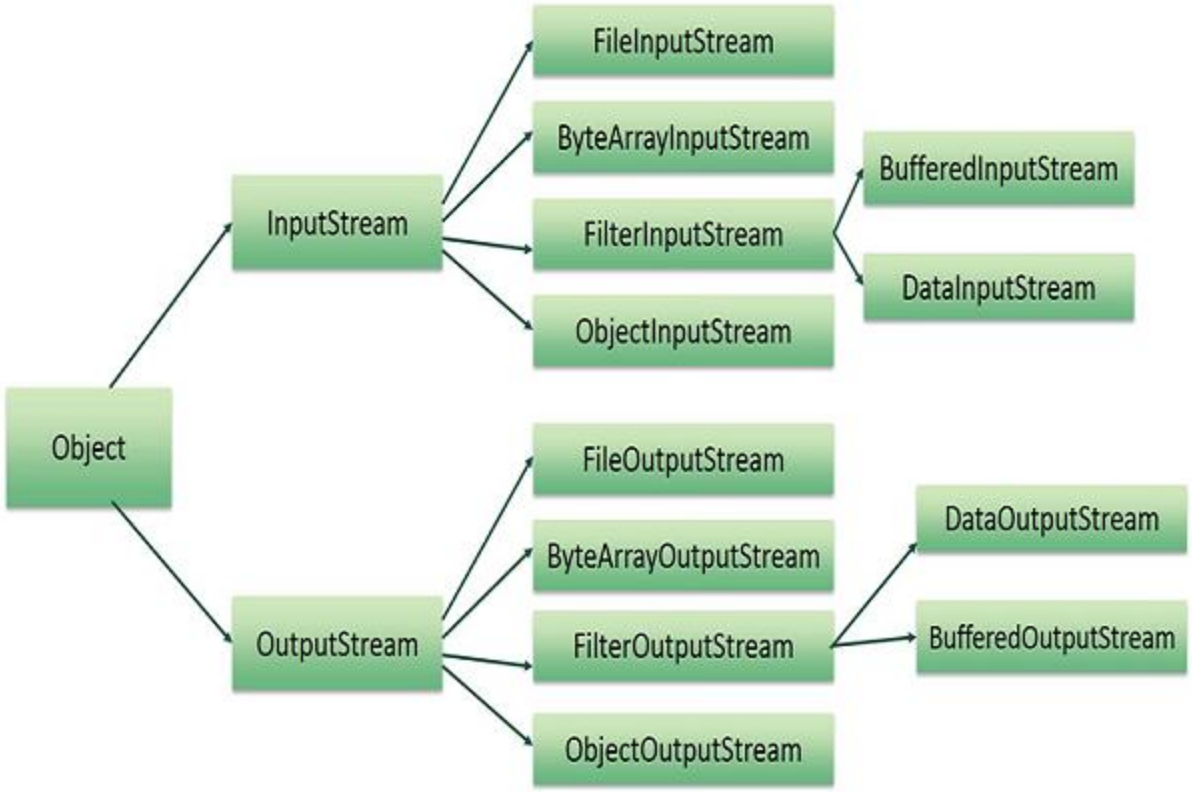
Output:

```
"info.properties" Can Write: true Is Hidden: false Length: 15 bytes
"info.properties".rtf Can Write: true Is Hidden: false Length: 385 bytes
.DS_Store Can Write: true Is Hidden: true Length: 36868 bytes
.localized Can Write: true Is Hidden: true Length: 0 bytes
Alok news Can Write: true Is Hidden: false Length: 850 bytes
apache-tomcat-9.0.0.M19 Can Write: true Is Hidden: false Length: 476
bytes
apache-tomcat-9.0.0.M19.tar Can Write: true Is Hidden: false Length:
13711360 bytes
bestreturn_org.rtf Can Write: true Is Hidden: false Length: 389 bytes
BIODATA.pages Can Write: true Is Hidden: false Length: 707985 bytes
BIODATA.pdf Can Write: true Is Hidden: false Length: 69681 bytes
BIODATA.png Can Write: true Is Hidden: false Length: 282125 bytes
workspace Can Write: true Is Hidden: false Length: 1972 bytes
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file –

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows –

```
File f = new File("C:/java/hello");
```

```
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links –

- ByteArrayInputStream
- DataInputStream