Arrays

Int a[5 ] =10,b=20,30,40;

Advantages

Arrays always start with zero ‘0’

* 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[];

Arrays always start with zero’0’

* int a[5]={10,20,30,40,50}

a[0-4]

a[0]=10

a[4]=50;

a[5]=null

Instantiation of an Array in Java

1. arrayRefVar=new datatype[size];

Ar[] = new int [];

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.

//Java Program to illustrate how to declare, instantiate, initialize

1. //and traverse the Java array.

class Testarray{

1. public static void main(String args[]){
2. int keerthana[]=new int[5];//declaration and instantiation
3. keerthana [0]=10;//initialization
4. keerthana [1]=20;
5. keerthana [2]=70;
6. keerthana [3]=40;
7. keerthana [4]=50;
8. //traversing array   length();
9. for(int i=0;i< keerthana .length;i++)//length is the property of array

System.out.println(keerthana [i]);

}}

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

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. }}

A= 2 4

1 3

Syntax to Declare Multidimensional Array in Java

1. dataType[][] arrayRefVar; (or)
2. dataType [][]arrayRefVar; (or)
3. dataType arrayRefVar[][]; (or)
4. dataType []arrayRefVar[];
5. int[][] arr=new int[3][3];//3 row and 3 column

Example to initialize Multidimensional Array in Java

1. arr[0 i][0 j]=1;

arr[0][1]=2;

arr[0][2]=3;

arr[1][0]=4;

arr[1][1]=5;

arr[1][2]=6;

arr[2][0]=7;

arr[2][1]=8;

arr[2][2]=9;

//Java Program to illustrate the use of multidimensional array

class Testarray3{

public static void main(String args[]){

//declaring and initializing 2D array

int arr[][]={{1,2},{2,4},{4,4}};

//printing 2D array

for(int i=0;i<3;i++){

 for(int j=0;j<2;j++){

   System.out.print(arr[i][j]+" ");

 }

 System.out.println();

}

}}

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. }}

## Hierarchy of Java Exception classes



Exception 🡨 🡪 Error

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, 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 which directly inherit 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 which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### 3) Error

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

Try{

Exception code

}

Catch(Exception code){

}

Catch(){

}

Finaly{

}

|  |  |
| --- | --- |
| Keyword | Description |
| Try | The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. |

## Java Exception Handling Example

Public class JavaExceptionExample{

  public static void main(String args[]){

   try{

      //code that may raise exception

      int data=100/0;

   }

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

   //rest code of the program

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

  }

}

Why use nested try block

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

: Syntax

....

try

{

    statement 1;

    statement 2;

----------------------------

    try

    {

       statement 1;

        statement 2;

    }

    catch(Exception e)

    {

}

catch(Exception e)

    {

}

---------------------------------------------------------------------

}

catch(Exception e)

{

}

....

Example program

class Excep6{

 public static void main(String args[]){

  try{

    try{

     System.out.println("going to divide");

     int b =39/0;

}

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

    try{

   int a[]=new int[5];

    a[5]=4;

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

    System.out.println("other statement);

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

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

 }

}

Case 1

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

class TestFinallyBlock{

  public static void main(String args[]){

  try{

   int data=25/5;

   System.out.println(data);

  }

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

  finally{System.out.println("finally block is always executed");}

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

  }

}

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

return\_type method\_name() throws exception\_class\_name{

//method code

}

import java.io.IOException;

class Testthrows1{

  void m()throws IOException,

{

    throw new IOException("device error");//checked exception

  }

  void n()throws IOException{

    m();

  }

  void p(){

   try{

    n();

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

  }

  public static void main(String args[]){

   Testthrows1 obj=new Testthrows1();

   obj.p();

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

  }

}

1. import java.io.\*;
2. class M{
3. void method()throws IOException{
4. throw new IOException("device error");
5. }
6. }
7. public class Testthrows2{
8. public static void main(String args[]){
9. try{
10. M m=new M();
11. m.method();
12. }catch(Exception e){System.out.println("exception handled");}
14. System.out.println("normal flow...");
15. }
16. }

import java.io.\*;

class M{

 void method()throws IOException{

  System.out.println("device operation performed");

 }

}

class Testthrows3{

   public static void main(String args[])throws IOException{//declare exception

     M m=new M();

     m.method();

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

  }

}

import java.io.\*;

class M{

 void method()throws IOException{

  throw new IOException("device error");

 }

}

class Testthrows4{

   public static void main(String args[])throws IOException{//declare exception

     M m=new M();

     m.method();

    System.out.println("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. |

## Java throw example

void m(){

throw new ArithmeticException("sorry");

}

## Java throws example

void m()throws ArithmeticException{

//method code

}

## Java throw and throws example

void m()throws ArithmeticException{

throw new ArithmeticException("sorry");

}

# Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Final | finally | finalize |
| 1) | Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| 2) | Final is a keyword. | Finally is a block. | Finalize is a method. |

## final example

class FinalExample{

public static void main(String[] args){

final int x=100;

x=200;//Compile Time Error

}}

## finally example

1. class FinallyExample{
2. public static void main(String[] args){
3. try{
4. int x=300;
5. }catch(Exception e){System.out.println(e);}
6. finally{
7. System.out.println("finally block is executed");
8. }
9. }}

## finalize example

class FinalizeExample{

public void finalize(){System.out.println("finalize called");}

public static void main(String[] args){

FinalizeExample f1=new FinalizeExample();

FinalizeExample f2=new FinalizeExample();

f1=null;

f2=null;

System.gc();

}}

# Wrapper classes in Java

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

## Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

* Change the value in Method: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* Serialization: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* Synchronization: Java synchronization works with objects in Multithreading.
* java.util package: The java.util package provides the utility classes to deal with objects.
* Collection Framework: Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the java.lang package are known as wrapper classes in Java. The list of eight wrapper classes are given below:

|  |  |
| --- | --- |
| Primitive Type | Wrapper class |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) |
| char | [Character](https://www.javatpoint.com/post/java-character) |
| byte | [Byte](https://www.javatpoint.com/java-byte) |
| short | [Short](https://www.javatpoint.com/java-short) |
| int | [Integer](https://www.javatpoint.com/java-integer) |
| long | [Long](https://www.javatpoint.com/java-long) |
| float | [Float](https://www.javatpoint.com/java-float) |
| double | [Double](https://www.javatpoint.com/java-double) |

## Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

//Java program to convert primitive into objects

//Autoboxing example of int to Integer

public class WrapperExample1{

public static void main(String args[]){

//Converting int into Integer

int a=20;

Integer i=Integer.valueOf(a);//converting int into Integer explicitly

Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);

}}

## Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

//Java program to convert object into primitives

//Unboxing example of Integer to int

public class WrapperExample2{

public static void main(String args[]){

//Converting Integer to int

Integer a=new Integer(3);

int i=a.intValue();//converting Integer to int explicitly

int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);

}}

## Wrapper classes Example

//Java Program to convert all primitives into its corresponding

//wrapper objects and vice-versa

public class WrapperExample3{

public static void main(String args[]){

byte b=10;

short s=20;

int i=30;

long l=40;

float f=50.0F;

double d=60.0D;

char c='a';

boolean b2=true;

//Autoboxing: Converting primitives into objects

Byte byteobj=b;

Short shortobj=s;

Integer intobj=i;

Long longobj=l;

Float floatobj=f;

Double doubleobj=d;

Character charobj=c;

Boolean boolobj=b2;

//Printing objects

System.out.println("---Printing object values---");

System.out.println("Byte object: "+byteobj);

System.out.println("Short object: "+shortobj);

System.out.println("Integer object: "+intobj);

System.out.println("Long object: "+longobj);

System.out.println("Float object: "+floatobj);

System.out.println("Double object: "+doubleobj);

System.out.println("Character object: "+charobj);

System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives

byte bytevalue=byteobj;

short shortvalue=shortobj;

int intvalue=intobj;

long longvalue=longobj;

float floatvalue=floatobj;

double doublevalue=doubleobj;

char charvalue=charobj;

boolean boolvalue=boolobj;

//Printing primitives

System.out.println("---Printing primitive values---");

System.out.println("byte value: "+bytevalue);

System.out.println("short value: "+shortvalue);

System.out.println("int value: "+intvalue);

System.out.println("long value: "+longvalue);

System.out.println("float value: "+floatvalue);

System.out.println("double value: "+doublevalue);

System.out.println("char value: "+charvalue);

System.out.println("boolean value: "+boolvalue);

}}

# Call by Value and Call by Reference in Java

|  |
| --- |
| There is only call by value in java, not call by reference. If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method. |

call by value in java

|  |
| --- |
| In case of call by value original value is not changed. Let's take a simple example: |

class Operation{

 int data=50;

 void change(int data){

 data=data+100;//changes will be in the local variable only

 }

 public static void main(String args[]){

   Operation op=new Operation();

   System.out.println("before change "+op.data);

   op.change(500);

   System.out.println("after change "+op.data);

 }

}

Another Example of call by value in java

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

class Operation2{

 int data=50;

 void change(Operation2 op){

 op.data=op.data+100;//changes will be in the instance variable

 }

 public static void main(String args[]){

   Operation2 op=new Operation2();

   System.out.println("before change "+op.data);

   op.change(op);//passing object

   System.out.println("after change "+op.data);

 }

}