**UNIT-II**

* Classes and objects
* Class declaration
* Creating objects
* Methods
* Constructors and Constructor Overloading
* Importance of Static Keyword and Examples
* this Keyword
* Arrays
* Command Line Arguments
* Nested Classes
* Garbage Collector

**Classes and objects:**

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

* **State:** represents the data (value) of an object.
* **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

* An object is a real-world entity.
* An object is a runtime entity.
* The object is an entity which has state and behavior.
* The object is an instance of a class.

A **class** is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A **class** in Java can contain:

* Fields
* Methods
* Constructors
* Blocks
* Nested class and interface

**Class declaration:**

Syntax to declare a class:

class <class\_name>

{

field;

method;

}

**Instance variable in Java**

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

**Method in Java**

In Java, a method is like a function which is used to expose the behavior of an object.

**Advantage of Method**

* Code Reusability
* Code Optimization

**new keyword in Java**

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory

**Object and Class Example: main within the class**

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

1. **class** Student
2. {
3. **int** id;
4. String name;
5. **public** **static** **void** main(String args[])
6. {
7. Student s1=**new** Student();
8. System.out.println(s1.id);
9. System.out.println(s1.name);
10. }
11. }

**Object and Class Example: main outside the class**

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

1. **class** Student
2. {
3. **int** id;
4. String name;
5. }
6. **class** TestStudent1
7. {
8. **public** **static** **void** main(String args[])
9. {
10. Student s1=**new** Student();
11. System.out.println(s1.id);
12. System.out.println(s1.name);
13. }
14. }

## 3 Ways to initialize object

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

## 1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

1. **class** Student
2. {
3. **int** id;
4. String name;
5. }
6. **class** TestStudent2
7. {
8. **public** **static** **void** main(String args[])
9. {
10. Student s1=**new** Student();
11. s1.id=101;
12. s1.name="abc";
13. System.out.println(s1.id+" "+s1.name);
14. }
15. }

We can also create multiple objects and store information in it through reference variable.

1. class Student
2. {
3. int id;
4. String name;
5. }
6. class TestStudent3
7. {
8. public static void main(String args[])
9. {
10. //Creating objects
11. Student s1=new Student();
12. Student s2=new Student();
13. //Initializing objects
14. s1.id=101;
15. s1.name="abc";
16. s2.id=102;
17. s2.name="xyz";
18. //Printing data
19. System.out.println(s1.id+" "+s1.name);
20. System.out.println(s2.id+" "+s2.name);
21. }
22. }

### 2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

1. class Student
2. {
3. int rollno;
4. String name;
5. void insertRecord(int r, String n)
6. {
7. rollno=r;
8. name=n;
9. }
10. void displayInformation()
11. {
12. System.out.println(rollno+" "+name);
13. }
14. }
15. class TestStudent4
16. {
17. public static void main(String args[])
18. {
19. Student s1=new Student();
20. Student s2=new Student();
21. s1.insertRecord(111,"Karan");
22. s2.insertRecord(222,"Aryan");
23. s1.displayInformation();
24. s2.displayInformation();
25. }
26. }



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

**Constructors:**In [Java](https://www.javatpoint.com/java-tutorial), a constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created. At the time of calling constructor, memory for the object is allocated in the memory.

* It is a special type of method which is used to initialize the object.
* Every time an object is created using the new() keyword, at least one constructor is called.
* It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

**Rules for creating Java constructor**

There are two rules defined for the constructor.

* Constructor name must be the same as its class name
* A Constructor must have no explicit return type
* A Java constructor cannot be abstract, static, final, and synchronized

**Types of Java constructors**

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

**Java Default Constructor**

A constructor is called "Default Constructor" when it doesn't have any parameter.

### Syntax of default constructor:

<class\_name>(){}

**Example of default constructor**

|  |
| --- |
| In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation. |

1. //Java Program to create and call a default constructor
2. **class** Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
5. //main method
6. **public** **static** **void** main(String args[]){
7. //calling a default constructor
8. Bike1 b=**new** Bike1();
9. }
10. }

#### Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



### Purpose of a default constructor

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

### Example of default constructor that displays the default values

1. **class** Student3
2. {
3. **int** id;
4. String name;
5. **void** display()
6. {
7. System.out.println(id+" "+name);
8. }
9. **public** **static** **void** main(String args[])
10. {
11. Student3 s1=**new** Student3();
12. Student3 s2=**new** Student3();
13. s1.display();
14. s2.display();
15. }
16. }

Output:

0 null

0 null

**Explanation:**In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

### Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor. The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

### Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

1. //Java Program to demonstrate the use of the parameterized constructor.
2. **class** Student4{
3. **int** id;
4. String name;
5. //creating a parameterized constructor
6. Student4(**int** i,String n){
7. id = i;
8. name = n;
9. }
10. //method to display the values
11. **void** display(){System.out.println(id+" "+name);}
13. **public** **static** **void** main(String args[]){
14. //creating objects and passing values
15. Student4 s1 = **new** Student4(111,"Karan");
16. Student4 s2 = **new** Student4(222,"Aryan");
17. //calling method to display the values of object
18. s1.display();
19. s2.display();
20. }
21. }

Output:

111 Karan

222 Aryan

### Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor [overloading in Java](https://www.javatpoint.com/method-overloading-in-java) is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

### Example of Constructor Overloading

1. //Java program to overload constructors
2. **class** Student5{
3. **int** id;
4. String name;
5. **int** age;
6. //creating two arg constructor
7. Student5(**int** i,String n){
8. id = i;
9. name = n;
10. }
11. //creating three arg constructor
12. Student5(**int** i,String n,**int** a){
13. id = i;
14. name = n;
15. age=a;
16. }
17. **void** display(){System.out.println(id+" "+name+" "+age);}
19. **public** **static** **void** main(String args[]){
20. Student5 s1 = **new** Student5(111,"Karan");
21. Student5 s2 = **new** Student5(222,"Aryan",25);
22. s1.display();
23. s2.display();
24. }
25. }

Output:

111 Karan 0

222 Aryan 25

## Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |

## Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

* By constructor
* By assigning the values of one object into another
* By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

1. //Java program to initialize the values from one object to another object.
2. **class** Student6{
3. **int** id;
4. String name;
5. //constructor to initialize integer and string
6. Student6(**int** i,String n){
7. id = i;
8. name = n;
9. }
10. //constructor to initialize another object
11. Student6(Student6 s){
12. id = s.id;
13. name =s.name;
14. }
15. **void** display(){System.out.println(id+" "+name);}
17. **public** **static** **void** main(String args[]){
18. Student6 s1 = **new** Student6(111,"Karan");
19. Student6 s2 = **new** Student6(s1);
20. s1.display();
21. s2.display();
22. }
23. }

**Importance of Static Keyword and Examples**

The **static keyword** in [Java](https://www.javatpoint.com/java-tutorial) is used for memory management mainly. We can apply static keyword with [variables](https://www.javatpoint.com/java-variables), methods, blocks and [nested classes](https://www.javatpoint.com/java-inner-class). The static keyword belongs to the class than an instance of the class.

The static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

## 1) Java static variable

If you declare any variable as static, it is known as a static variable.

* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.

### Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

#### Understanding the problem without static variable

1. **class** Student{
2. **int** rollno;
3. String name;
4. String college="ITS";
5. }

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all [objects](https://www.javatpoint.com/object-and-class-in-java). If we make it static, this field will get the memory only once.

#### Java static property is shared to all objects.

### Example of static variable

1. //Java Program to demonstrate the use of static variable
2. **class** Student{
3. **int** rollno;//instance variable
4. String name;
5. **static** String college ="ITS";//static variable
6. //constructor
7. Student(**int** r, String n){
8. rollno = r;
9. name = n;
10. }
11. //method to display the values
12. **void** display (){System.out.println(rollno+" "+name+" "+college);}
13. }
14. //Test class to show the values of objects
15. **public** **class** TestStaticVariable1{
16. **public** **static** **void** main(String args[]){
17. Student s1 = **new** Student(111,"Karan");
18. Student s2 = **new** Student(222,"Aryan");
19. //we can change the college of all objects by the single line of code
20. //Student.college="BBDIT";
21. s1.display();
22. s2.display();
23. }
24. }

Output:

111 Karan ITS

222 Aryan ITS



### Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

1. //Java Program to demonstrate the use of an instance variable
2. //which get memory each time when we create an object of the class.
3. **class** Counter{
4. **int** count=0;//will get memory each time when the instance is created
6. Counter(){
7. count++;//incrementing value
8. System.out.println(count);
9. }
11. **public** **static** **void** main(String args[]){
12. //Creating objects
13. Counter c1=**new** Counter();
14. Counter c2=**new** Counter();
15. Counter c3=**new** Counter();
16. }
17. }

Output:

1

1

1

### Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

1. //Java Program to illustrate the use of static variable which
2. //is shared with all objects.
3. **class** Counter2{
4. **static** **int** count=0;//will get memory only once and retain its value
6. Counter2(){
7. count++;//incrementing the value of static variable
8. System.out.println(count);
9. }
11. **public** **static** **void** main(String args[]){
12. //creating objects
13. Counter2 c1=**new** Counter2();
14. Counter2 c2=**new** Counter2();
15. Counter2 c3=**new** Counter2();
16. }
17. }

Output:

1

2

3

## 2) Java static method

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.

### Example of static method

1. //Java Program to demonstrate the use of a static method.
2. **class** Student{
3. **int** rollno;
4. String name;
5. **static** String college = "ITS";
6. //static method to change the value of static variable
7. **static** **void** change(){
8. college = "BBDIT";
9. }
10. //constructor to initialize the variable
11. Student(**int** r, String n){
12. rollno = r;
13. name = n;
14. }
15. //method to display values
16. **void** display(){System.out.println(rollno+" "+name+" "+college);}
17. }
18. //Test class to create and display the values of object
19. **public** **class** TestStaticMethod{
20. **public** **static** **void** main(String args[]){
21. Student.change();//calling change method
22. //creating objects
23. Student s1 = **new** Student(111,"Karan");
24. Student s2 = **new** Student(222,"Aryan");
25. Student s3 = **new** Student(333,"Sonoo");
26. //calling display method
27. s1.display();
28. s2.display();
29. s3.display();
30. }
31. }

Output:111 Karan BBDIT

222 Aryan BBDIT

333 Sonoo BBDIT

# this keyword in Java

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



## Usage of Java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

**1) this: to refer current class instance variable**

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

#### Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. **class** TestThis1{
13. **public** **static** **void** main(String args[]){
14. Student s1=**new** Student(111,"ankit",5000f);
15. Student s2=**new** Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}

**Output:**

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

#### Solution of the above problem by this keyword

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis2{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

**Output:**

111 ankit 5000.0

112 sumit 6000.0

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

#### Program where this keyword is not required

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** r,String n,**float** f){
6. rollno=r;
7. name=n;
8. fee=f;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis3{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

**Output:**

111 ankit 5000.0

112 sumit 6000.0

### 2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example



1. **class** A{
2. **void** m(){System.out.println("hello m");}
3. **void** n(){
4. System.out.println("hello n");
5. //m();//same as this.m()
6. **this**.m();
7. }
8. }
9. **class** TestThis4{
10. **public** **static** **void** main(String args[]){
11. A a=**new** A();
12. a.n();
13. }}

hello n

hello m

**3) this() : to invoke current class constructor**

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor:**

1. **class** A{
2. A(){System.out.println("hello a");}
3. A(**int** x){
4. **this**();
5. System.out.println(x);
6. }
7. }
8. **class** TestThis5{
9. **public** **static** **void** main(String args[]){
10. A a=**new** A(10);
11. }}

**Output:**

hello a

10

**Calling parameterized constructor from default constructor:**

1. **class** A{
2. A(){
3. **this**(5);
4. System.out.println("hello a");
5. }
6. A(**int** x){
7. System.out.println(x);
8. }
9. }
10. **class** TestThis6{
11. **public** **static** **void** main(String args[]){
12. A a=**new** A();
13. }}

Output:

5

hello a

### Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**(rollno,name,course);//reusing constructor
12. **this**.fee=fee;
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis7{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

**Output:**

111 ankit java 0.0

112 sumit java 6000.0

### 4) this: to pass as an argument in the method

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

1. **class** S2{
2. **void** m(S2 obj){
3. System.out.println("method is invoked");
4. }
5. **void** p(){
6. m(**this**);
7. }
8. **public** **static** **void** main(String args[]){
9. S2 s1 = **new** S2();
10. s1.p();
11. }
12. }

**Output:**

method is invoked

### 5) this: to pass as argument in the constructor call

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

1. **class** B{
2. A4 obj;
3. B(A4 obj){
4. **this**.obj=obj;
5. }
6. **void** display(){
7. System.out.println(obj.data);//using data member of A4 class
8. }
9. }
11. **class** A4{
12. **int** data=10;
13. A4(){
14. B b=**new** B(**this**);
15. b.display();
16. }
17. **public** **static** **void** main(String args[]){
18. A4 a=**new** A4();
19. }
20. }

**Output:10**

### 6) this keyword can be used to return current class instance

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

### Syntax of this that can be returned as a statement

1. return\_type method\_name(){
2. **return** **this**;
3. }

### Example of this keyword that you return as a statement from the method

1. **class** A{
2. A getA(){
3. **return** **this**;
4. }
5. **void** msg(){System.out.println("Hello java");}
6. }
7. **class** Test1{
8. **public** **static** **void** main(String args[]){
9. **new** A().getA().msg();
10. }
11. }

**Output:** Hello java

**Arrays**

Normally, an array is a collection of similar type of elements which has contiguous memory location.

**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. 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 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

### 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[];

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**

## For-each Loop for Java Array

We can also print the Java array using [**for-each loop**](https://www.javatpoint.com/for-each-loop). The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

**for**(data\_type variable:array){

//body of the loop

}

Let us see the example of print the elements of Java array using the for-each loop.

1. //Java Program to print the array elements using for-each loop
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. //printing array using for-each loop
6. **for**(**int** i:arr)
7. System.out.println(i);
8. }}

Output:

33

3

4

5

## Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get the minimum number of an array using a method.

1. //Java Program to demonstrate the way of passing an array
2. //to method.
3. **class** Testarray2{
4. //creating a method which receives an array as a parameter
5. **static** **void** min(**int** arr[]){
6. **int** min=arr[0];
7. **for**(**int** i=1;i<arr.length;i++)
8. **if**(min>arr[i])
9. min=arr[i];
11. System.out.println(min);
12. }
14. **public** **static** **void** main(String args[]){
15. **int** a[]={33,3,4,5};//declaring and initializing an array
16. min(a);//passing array to method
17. }}

Output: 3

## Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

1. //Java Program to demonstrate the way of passing an anonymous array
2. //to method.
3. **public** **class** TestAnonymousArray{
4. //creating a method which receives an array as a parameter
5. **static** **void** printArray(**int** arr[]){
6. **for**(**int** i=0;i<arr.length;i++)
7. System.out.println(arr[i]);
8. }
10. **public** **static** **void** main(String args[]){
11. printArray(**new** **int**[]{10,22,44,66});//passing anonymous array to method
12. }}

**Output:**

**10**

**22**

**44**

**66**

## Returning Array from the Method

We can also return an array from the method in Java.

1. //Java Program to return an array from the method
2. **class** TestReturnArray{
3. //creating method which returns an array
4. **static** **int**[] get(){
5. **return** **new** **int**[]{10,30,50,90,60};
6. }
8. **public** **static** **void** main(String args[]){
9. //calling method which returns an array
10. **int** arr[]=get();
11. //printing the values of an array
12. **for**(**int** i=0;i<arr.length;i++)
13. System.out.println(arr[i]);
14. }}

Output:

10

30

50

90

60

## ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

1. //Java Program to demonstrate the case of
2. //ArrayIndexOutOfBoundsException in a Java Array.
3. **public** **class** TestArrayException{
4. **public** **static** **void** main(String args[]){
5. **int** arr[]={50,60,70,80};
6. **for**(**int** i=0;i<=arr.length;i++){
7. System.out.println(arr[i]);
8. }
9. }}

Output:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at TestArrayException.main(TestArrayException.java:5)

50

60

70

80

## 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

## Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

1. //Java Program to illustrate the jagged array
2. **class** TestJaggedArray{
3. **public** **static** **void** main(String[] args){
4. //declaring a 2D array with odd columns
5. **int** arr[][] = **new** **int**[3][];
6. arr[0] = **new** **int**[3];
7. arr[1] = **new** **int**[4];
8. arr[2] = **new** **int**[2];
9. //initializing a jagged array
10. **int** count = 0;
11. **for** (**int** i=0; i<arr.length; i++)
12. **for**(**int** j=0; j<arr[i].length; j++)
13. arr[i][j] = count++;
15. //printing the data of a jagged array
16. **for** (**int** i=0; i<arr.length; i++){
17. **for** (**int** j=0; j<arr[i].length; j++){
18. System.out.print(arr[i][j]+" ");
19. }
20. System.out.println();//new line
21. }
22. }
23. }

Output:

0 1 2

3 4 5 6

7 8

## What is the class name of Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by getClass().getName() method on the object.

1. //Java Program to get the class name of array in Java
2. **class** Testarray4{
3. **public** **static** **void** main(String args[]){
4. //declaration and initialization of array
5. **int** arr[]={4,4,5};
6. //getting the class name of Java array
7. Class c=arr.getClass();
8. String name=c.getName();
9. //printing the class name of Java array
10. System.out.println(name);
12. }}

Output:

I

## Copying a Java Array

We can copy an array to another by the arraycopy() method of System class.

**Syntax of arraycopy method**

1. **public** **static** **void** arraycopy(
2. Object src, **int** srcPos,Object dest, **int** destPos, **int** length
3. )

### Example of Copying an Array in Java

1. //Java Program to copy a source array into a destination array in Java
2. **class** TestArrayCopyDemo {
3. **public** **static** **void** main(String[] args) {
4. //declaring a source array
5. **char**[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
6. 'i', 'n', 'a', 't', 'e', 'd' };
7. //declaring a destination array
8. **char**[] copyTo = **new** **char**[7];
9. //copying array using System.arraycopy() method
10. System.arraycopy(copyFrom, 2, copyTo, 0, 7);
11. //printing the destination array
12. System.out.println(String.valueOf(copyTo));
13. }
14. }

Output:

caffein

## Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

1. //Java Program to clone the array
2. **class** Testarray1{
3. **public** **static** **void** main(String args[]){
4. **int** arr[]={33,3,4,5};
5. System.out.println("Printing original array:");
6. **for**(**int** i:arr)
7. System.out.println(i);
9. System.out.println("Printing clone of the array:");
10. **int** carr[]=arr.clone();
11. **for**(**int** i:carr)
12. System.out.println(i);
14. System.out.println("Are both equal?");
15. System.out.println(arr==carr);
17. }}

Output:

Printing original array:

33

3

4

5

Printing clone of the array:

33

3

4

5

Are both equal?

false

## Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

1. //Java Program to demonstrate the addition of two matrices in Java
2. **class** Testarray5{
3. **public** **static** **void** main(String args[]){
4. //creating two matrices
5. **int** a[][]={{1,3,4},{3,4,5}};
6. **int** b[][]={{1,3,4},{3,4,5}};
8. //creating another matrix to store the sum of two matrices
9. **int** c[][]=**new** **int**[2][3];
11. //adding and printing addition of 2 matrices
12. **for**(**int** i=0;i<2;i++){
13. **for**(**int** j=0;j<3;j++){
14. c[i][j]=a[i][j]+b[i][j];
15. System.out.print(c[i][j]+" ");
16. }
17. System.out.println();//new line
18. }
20. }}

Output:

2 6 8

6 8 10

## Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.



Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

1. //Java Program to multiply two matrices
2. **public** **class** MatrixMultiplicationExample{
3. **public** **static** **void** main(String args[]){
4. //creating two matrices
5. **int** a[][]={{1,1,1},{2,2,2},{3,3,3}};
6. **int** b[][]={{1,1,1},{2,2,2},{3,3,3}};
8. //creating another matrix to store the multiplication of two matrices
9. **int** c[][]=**new** **int**[3][3];  //3 rows and 3 columns
11. //multiplying and printing multiplication of 2 matrices
12. **for**(**int** i=0;i<3;i++){
13. **for**(**int** j=0;j<3;j++){
14. c[i][j]=0;
15. **for**(**int** k=0;k<3;k++)
16. {
17. c[i][j]+=a[i][k]\*b[k][j];
18. }//end of k loop
19. System.out.print(c[i][j]+" ");  //printing matrix element
20. }//end of j loop
21. System.out.println();//new line
22. }
23. }}

Output:

6 6 6

12 12 12

18 18 18

**Command Line Arguments:**

* The java command-line argument is an argument i.e. passed at the time of running the java program.
* The arguments passed from the console can be received in the java program and it can be used as an input.
* So, it provides a convenient way to check the behavior of the program for the different values. You can pass N (1,2,3 and so on) numbers of arguments from the command prompt.

**Simple example of command-line argument in java**

|  |
| --- |
| In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt. |

1. **class** CommandLineExample{
2. **public** **static** **void** main(String args[]){
3. System.out.println("Your first argument is: "+args[0]);
4. }
5. }

compile by > javac CommandLineExample.java

run by > java CommandLineExample svec

Output: Your first argument is: svec

Example of command-line argument that prints all the values

|  |
| --- |
| In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop. |

1. **class** A{
2. **public** **static** **void** main(String args[]){
4. **for**(**int** i=0;i<args.length;i++)
5. System.out.println(args[i]);
7. }
8. }

compile by > javac A.java

run by > java cai aim 1 3 abc

Output: cai

aim

1

3

Abc

**Wrapper classes in Java**

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

|  |  |  |
| --- | --- | --- |
| **Primitive Type** | **Wrapper class** | **Metods** |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) | parseBoolean() |
| char | [Character](https://www.javatpoint.com/post/java-character) | parseChar() |
| byte | [Byte](https://www.javatpoint.com/java-byte) | parseByte() |
| short | [Short](https://www.javatpoint.com/java-short) | parseShort() |
| int | [Integer](https://www.javatpoint.com/java-integer) | parseInt() |
| long | [Long](https://www.javatpoint.com/java-long) | parseLong() |
| float | [Float](https://www.javatpoint.com/java-float) | parseFloat() |
| double | [Double](https://www.javatpoint.com/java-double) | parseDouble() |

## 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**

1. //Java program to convert primitive into objects
2. //Autoboxing example of int to Integer
3. **public** **class** WrapperExample1{
4. **public** **static** **void** main(String args[]){
5. //Converting int into Integer
6. **int** a=20;
7. Integer i=Integer.valueOf(a);//converting int into Integer explicitly
8. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
10. System.out.println(a+" "+i+" "+j);
11. }}

Output:

20 20 20

## 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**

1. //Java program to convert object into primitives
2. //Unboxing example of Integer to int
3. **public** **class** WrapperExample2{
4. **public** **static** **void** main(String args[]){
5. //Converting Integer to int
6. Integer a=**new** Integer(3);
7. **int** i=a.intValue();//converting Integer to int explicitly
8. **int** j=a;//unboxing, now compiler will write a.intValue() internally
10. System.out.println(a+" "+i+" "+j);
11. }}

Output:

3 3 3

**Nested Classes:**

**Java inner class** or nested class is a class that is declared inside the class or interface.We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.Additionally, it can access all the members of the outer class, including private data members and methods.

#### Syntax of Inner class

1. **class** Java\_Outer\_class{
2. //code
3. **class** Java\_Inner\_class{
4. //code
5. }
6. }

### Advantage of Java inner classes

There are three advantages of inner classes in Java.

* Nested classes represent a particular type of relationship that is it can access all the members (data members and methods) of the outer class, including private.
* Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
* Code Optimization: It requires less code to write.

### Types of Nested classes:

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

* Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
* Static nested class

|  |  |
| --- | --- |
| **Type** | **Description** |
| [Member Inner Class](https://www.javatpoint.com/member-inner-class) | A class created within class and outside method. |
| [Anonymous Inner Class](https://www.javatpoint.com/anonymous-inner-class) | A class created for implementing an interface or extending class. The java compiler decides its name. |
| [Local Inner Class](https://www.javatpoint.com/local-inner-class) | A class was created within the method. |
| [Static Nested Class](https://www.javatpoint.com/static-nested-class) | A static class was created within the class. |
| [Nested Interface](https://www.javatpoint.com/nested-interface) | An interface created within class or interface. |

## Java Member Inner Class Example

1. **class** TestMemberOuter1{
2. **private** **int** data=30;
3. **class** Inner{
4. **void** msg(){System.out.println("data is "+data);}
5. }
6. **public** **static** **void** main(String args[]){
7. TestMemberOuter1 obj=**new** TestMemberOuter1();
8. TestMemberOuter1.Inner in=obj.**new** Inner();
9. in.msg();
10. }
11. }

**Java local inner class example**

1. **public** **class** localInner1{
2. **private** **int** data=30;//instance variable
3. **void** display(){
4. **class** Local{
5. **void** msg(){System.out.println(data);}
6. }
7. Local l=**new** Local();
8. l.msg();
9. }
10. **public** **static** **void** main(String args[]){
11. localInner1 obj=**new** localInner1();
12. obj.display();
13. }
14. }

## Java static nested class example with instance method

**TestOuter1.java**

1. **class** TestOuter1{
2. **static** **int** data=30;
3. **static** **class** Inner{
4. **void** msg(){System.out.println("data is "+data);}
5. }
6. **public** **static** **void** main(String args[]){
7. TestOuter1.Inner obj=**new** TestOuter1.Inner();
8. obj.msg();
9. }
10. }

**Garbage Collector**

In java, garbage means unreferenced objects.Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### Advantage of Garbage Collection

* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

**How can an object be unreferenced?**

There are many ways:

* By nulling the reference
* By assigning a reference to another
* By anonymous object etc.

1) By nulling a reference:

1. Employee e=**new** Employee();
2. e=**null**;

2) By assigning a reference to another:

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

3) By anonymous object:

1. **new** Employee();

## finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

protected void finalize(){}

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

public static void gc(){}

**Simple Example of garbage collection in java**

1. **public** **class** TestGarbage1{
2. **public** **void** finalize(){System.out.println("object is garbage collected");}
3. **public** **static** **void** main(String args[]){
4. TestGarbage1 s1=**new** TestGarbage1();
5. TestGarbage1 s2=**new** TestGarbage1();
6. s1=**null**;
7. s2=**null**;
8. System.gc();
9. }
10. }