

Java OOPs Concepts

In this page, we will learn about the basics of OOPs. Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

Smalltalk is considered the first truly object-oriented programming language.



The popular object-oriented languages are **Java**, **C#**, **PHP**, **Python**, **C++**, etc.

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc.

OOPs (Object-Oriented Programming System)

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

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

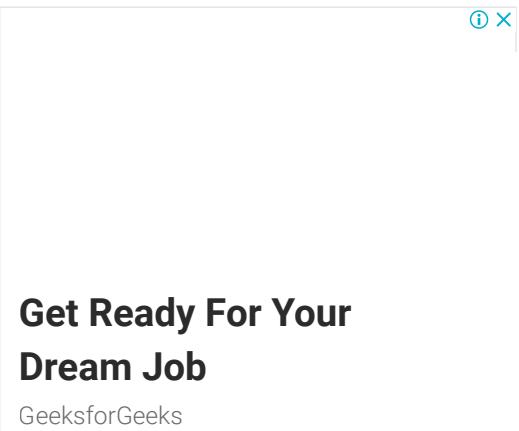
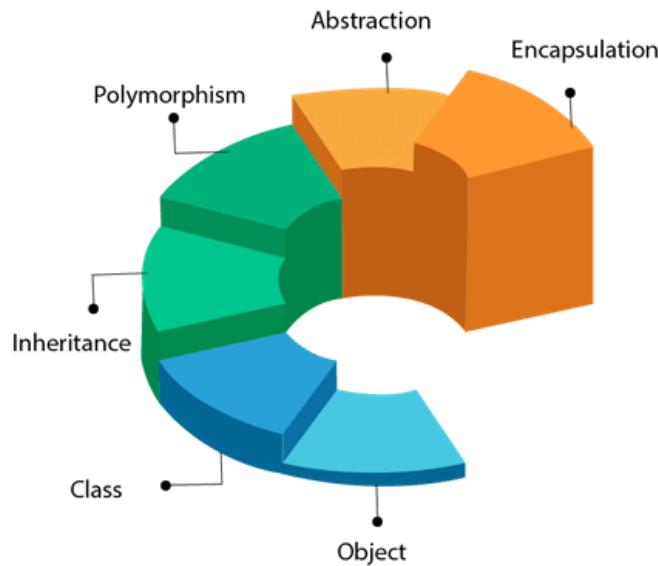
Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- **Coupling**
- **Cohesion**
- **Association**
- **Aggregation**



- Composition

OOPs (Object-Oriented Programming System)



Object



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

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

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

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.



Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.



Polymorphism

If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.



Capsule

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

Cohesion



Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The `java.io` package is a highly cohesive package because it has I/O related classes and interface. However, the `java.util` package is a weakly cohesive package because it has unrelated classes and interfaces.

Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

Advantage of OOPs over Procedure-oriented programming language

- 1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.
- 2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

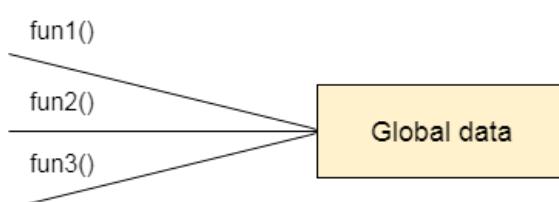


Figure: Data Representation in Procedure-Oriented Programming

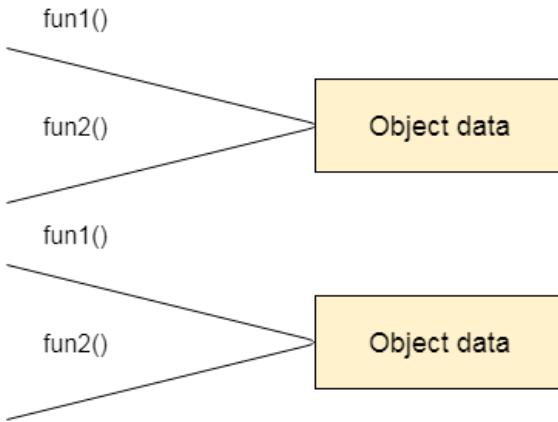


Figure: Data Representation in Object-Oriented Programming

3) OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

What is the difference between an object-oriented programming language and object-based programming language?

Object-based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object-based programming languages.

Do You Know?

- o Can we overload the main method?
- o A Java Constructor returns a value but, what?
- o Can we create a program without main method?
- o What are the six ways to use this keyword?
- o Why is multiple inheritance not supported in Java?
- o Why use aggregation?
- o Can we override the static method?
- o What is the covariant return type?
- o What are the three usages of Java super keyword?
- o Why use instance initializer block?
- o What is the usage of a blank final variable?
- o What is a marker or tagged interface?
- o What is runtime polymorphism or dynamic method dispatch?
- o What is the difference between static and dynamic binding?
- o How downcasting is possible in Java?
- o What is the purpose of a private constructor?
- o What is object cloning?

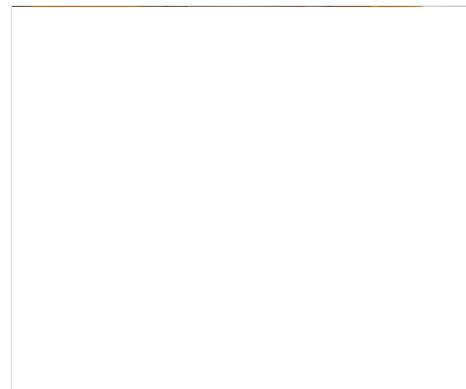


Java Naming conventions

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.



Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

- The name must not contain any white spaces.
- The name should not start with special characters like & (ampersand), \$ (dollar), _ (underscore).

Let's see some other rules that should be followed by identifiers.

Class

- It should start with the uppercase letter.
- It should be a noun such as Color, Button, System, Thread, etc.
- Use appropriate words, instead of acronyms.
- **Example: -**

```
public class Employee
{
    //code snippet
}
```

Interface

- It should start with the uppercase letter.
- It should be an adjective such as Runnable, Remote, ActionListener.
- Use appropriate words, instead of acronyms.
- **Example:** -

```
interface Printable
{
//code snippet
}
```

Method

- It should start with lowercase letter.
- It should be a verb such as main(), print(), println().
- If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
- **Example:-**

```
class Employee
{
//method
void draw()
{
//code snippet
}
}
```

Variable

- It should start with a lowercase letter such as id, name.
- It should not start with the special characters like & (ampersand), \$ (dollar), _ (underscore).
- If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
- Avoid using one-character variables such as x, y, z.
- **Example :-**

```
class Employee
{
//variable
int id;
//code snippet
}
```

Package

- It should be a lowercase letter such as java, lang.
- If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
- **Example :-**

```
package com.javatpoint; //package
class Employee
{
//code snippet
}
```

Constant

- It should be in uppercase letters such as RED, YELLOW.
- If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
- It may contain digits but not as the first letter.
- **Example :-**

```
class Employee
{
//constant
static final int MIN_AGE = 18;
//code snippet
}
```

CamelCase in java naming conventions

Java follows camel-case syntax for naming the class, interface, method, and variable.

If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

← Prev

Next →

 [For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Objects and Classes in Java

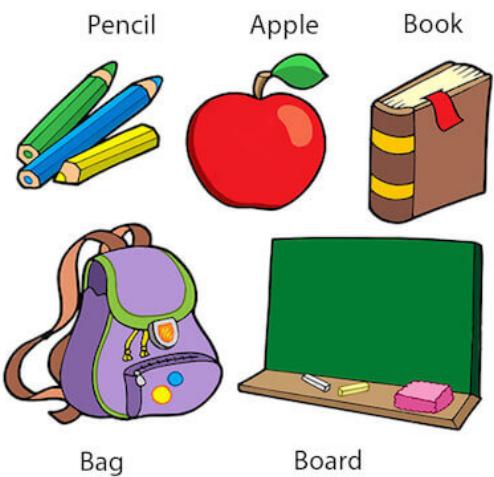
In this page, we will learn about Java objects and classes. In object-oriented programming technique, we design a program using objects and classes.

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

What is an object in Java

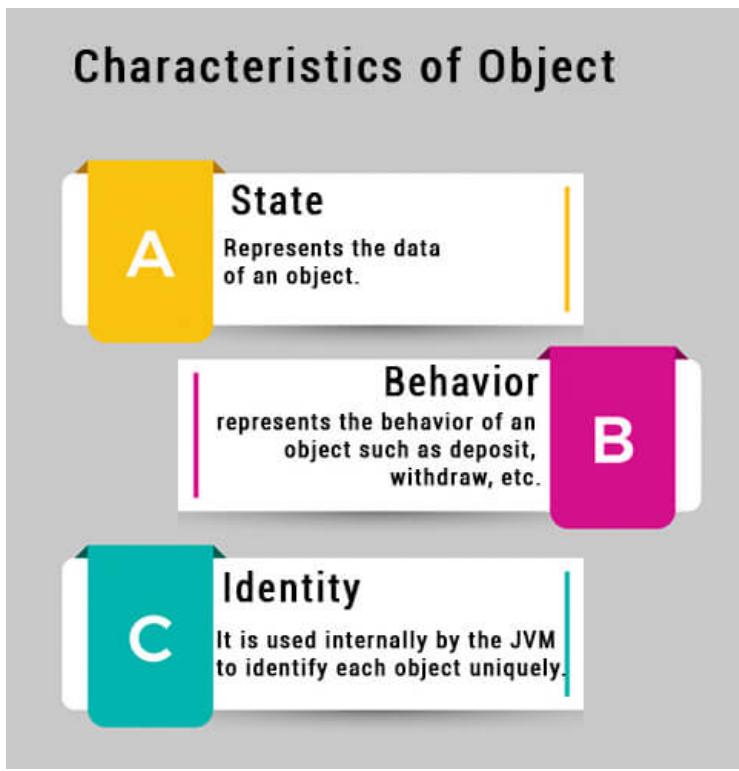
An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

Objects: Real World Examples



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.



For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

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*.

ⓘ X

Online Coding Competitions

GeeksforGeeks

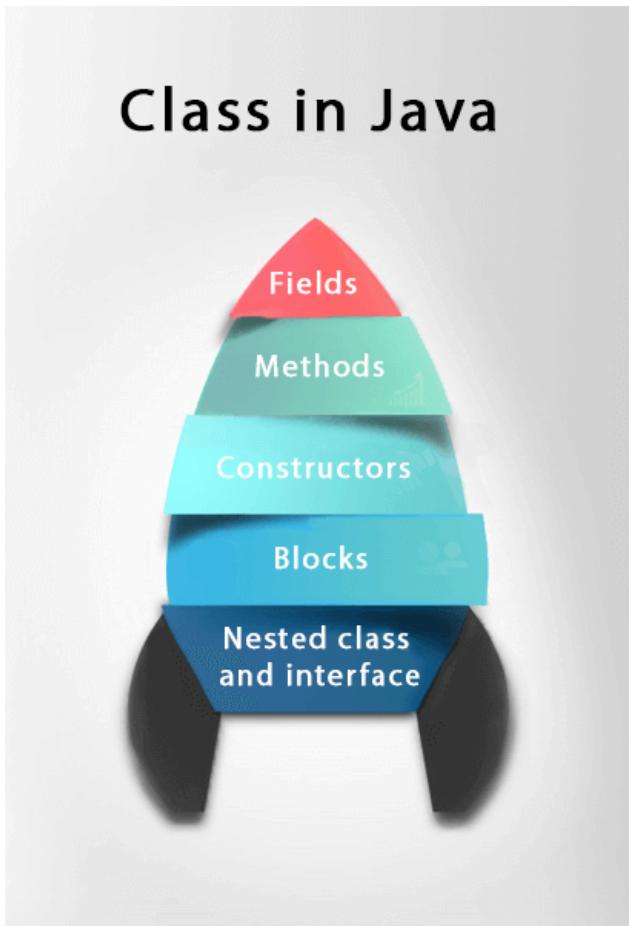
What is a class in Java

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**



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 area.

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.

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.

class Student{
    //defining fields
    int id;//field or data member or instance variable
    String name;
    //creating main method inside the Student class
    public static void main(String args[]){
        //Creating an object or instance
        Student s1=new Student();//creating an object of Student
        //Printing values of the object
        System.out.println(s1.id);//accessing member through reference variable
        System.out.println(s1.name);
    }
}
```

[Test it Now](#)

Output:

```
0
null
```

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.

File: TestStudent1.java

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.

class Student{
```

```

int id;
String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
public static void main(String args[]){
Student s1=new Student();
System.out.println(s1.id);
System.out.println(s1.name);
}
}

```

[Test it Now](#)

Output:

```

0
null

```

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.

File: TestStudent2.java

```

class Student{
int id;
String name;
}
class TestStudent2{
public static void main(String args[]){
Student s1=new Student();
s1.id=101;
s1.name="Sonoo";
System.out.println(s1.id+" "+s1.name);//printing members with a white space
}
}

```

[Test it Now](#)

Output:

```
101 Sonoo
```

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

File: TestStudent3.java

```
class Student{
    int id;
    String name;
}

class TestStudent3{
    public static void main(String args[]){
        //Creating objects
        Student s1=new Student();
        Student s2=new Student();

        //Initializing objects
        s1.id=101;
        s1.name="Sonoo";
        s2.id=102;
        s2.name="Amit";

        //Printing data
        System.out.println(s1.id+" "+s1.name);
        System.out.println(s2.id+" "+s2.name);
    }
}
```

[Test it Now](#)

Output:

```
101 Sonoo
102 Amit
```

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.

File: TestStudent4.java

```
class Student{
    int rollno;
    String name;
    void insertRecord(int r, String n){
        rollno=r;
        name=n;
    }
}
```

```

void displayInformation(){System.out.println(rollno+" "+name);}

}

class TestStudent4{

public static void main(String args[]){

Student s1=new Student();
Student s2=new Student();
s1.insertRecord(111,"Karan");
s2.insertRecord(222,"Aryan");
s1.displayInformation();
s2.displayInformation();

}
}

```

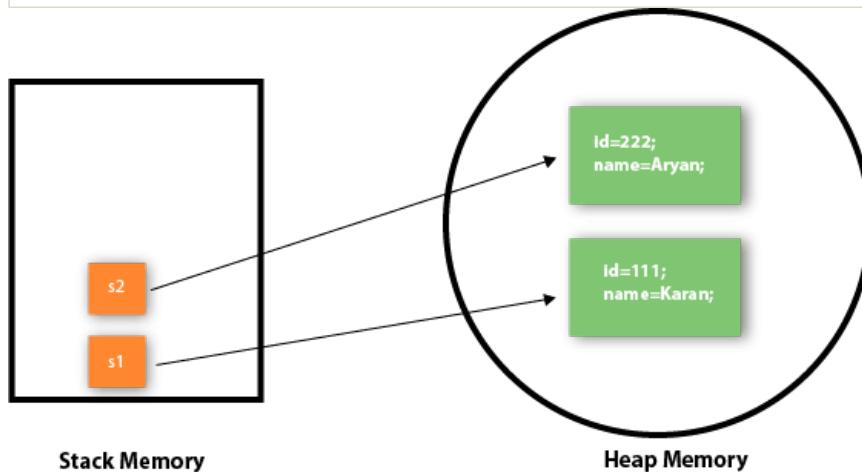
Test it Now

Output:

```

111 Karan
222 Aryan

```



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.

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```

class Employee{

    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
    }
}

```

```

        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}

public class TestEmployee {
    public static void main(String[] args) {
        Employee e1=new Employee();
        Employee e2=new Employee();
        Employee e3=new Employee();
        e1.insert(101,"ajeet",45000);
        e2.insert(102,"irfan",25000);
        e3.insert(103,"nakul",55000);
        e1.display();
        e2.display();
        e3.display();
    }
}

```

[Test it Now](#)

Output:

```

101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0

```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```

class Rectangle{
    int length;
    int width;
    void insert(int l, int w){
        length=l;
        width=w;
    }
    void calculateArea(){System.out.println(length*width);}
}

class TestRectangle1{
    public static void main(String args[]){
        Rectangle r1=new Rectangle();
        Rectangle r2=new Rectangle();
        r1.insert(11,5);
        r2.insert(3,15);
        r1.calculateArea();
    }
}

```

```
r2.calculateArea();
}
}
```

[Test it Now](#)

Output:

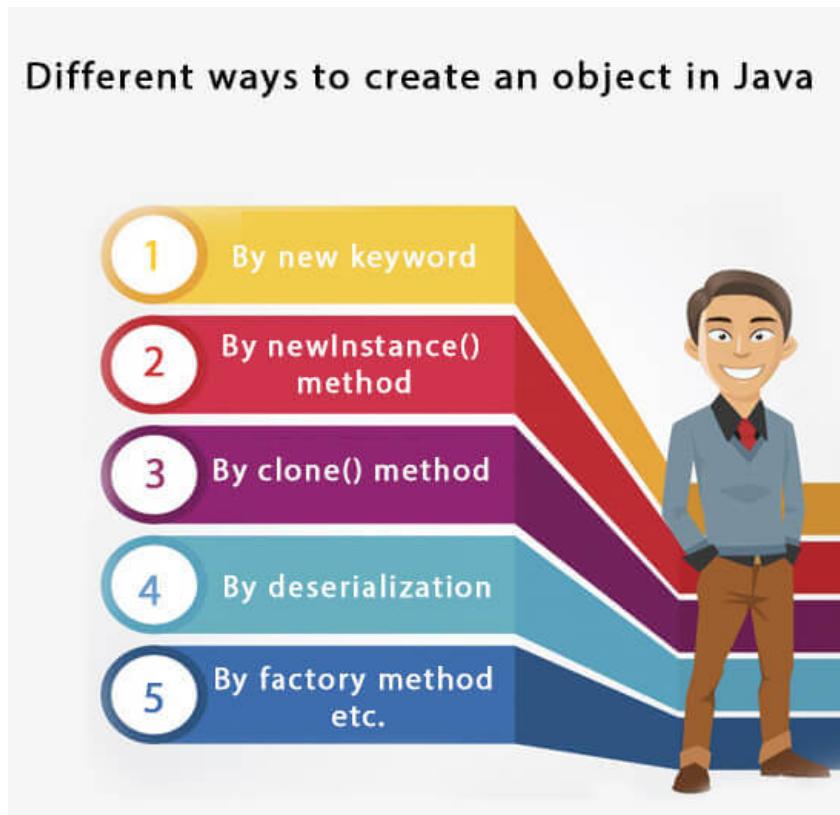
```
55
45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.



Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
new Calculation(); //anonymous object
```

Calling method through a reference:

```
Calculation c=new Calculation();
c.fact(5);
```

Calling method through an anonymous object

```
new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```
class Calculation{
    void fact(int n){
        int fact=1;
        for(int i=1;i<=n;i++){
            fact=fact*i;
        }
        System.out.println("factorial is "+fact);
    }
    public static void main(String args[]){
        new Calculation().fact(5);//calling method with anonymous object
    }
}
```

Output:

```
Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```
int a=10, b=20;
```

Initialization of reference variables:

```
Rectangle r1=new Rectangle(), r2=new Rectangle(); //creating two objects
```

Let's see the example:

```
//Java Program to illustrate the use of Rectangle class which
//has length and width data members
class Rectangle{
    int length;
    int width;
    void insert(int l,int w){
        length=l;
        width=w;
    }
}
```

```

}

void calculateArea(){System.out.println(length*width);}

}

class TestRectangle2{

public static void main(String args[]){

    Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects

    r1.insert(11,5);
    r2.insert(3,15);
    r1.calculateArea();
    r2.calculateArea();

}
}

```

Test it Now

Output:

```

55
45

```

Real World Example: Account

File: TestAccount.java

```

//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() methods

class Account{

    int acc_no;
    String name;
    float amount;
    //Method to initialize object
    void insert(int a,String n,float amt){
        acc_no=a;
        name=n;
        amount=amt;
    }
    //deposit method
    void deposit(float amt){
        amount=amount+amt;
        System.out.println(amt+" deposited");
    }
    //withdraw method
    void withdraw(float amt){
        if(amount<amt){
            System.out.println("Insufficient Balance");
        }else{
            amount=amount-amt;
            System.out.println(amt+" withdrawn");
        }
    }
}

```

```
}

}

//method to check the balance of the account
void checkBalance(){System.out.println("Balance is: "+amount);}

//method to display the values of an object
void display(){System.out.println(acc_no+" "+name+" "+amount);}

}

//Creating a test class to deposit and withdraw amount
class TestAccount{

public static void main(String[] args){

Account a1=new Account();
a1.insert(832345,"Ankit",1000);
a1.display();
a1.checkBalance();
a1.deposit(40000);
a1.checkBalance();
a1.withdraw(15000);
a1.checkBalance();
}}
```

[Test it Now](#)

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

[← Prev](#)

[Next →](#)

 [For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Method in Java

In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**. In this section, we will learn **what is a method in Java**, **types of methods**, **method declaration**, and **how to call a method in Java**.

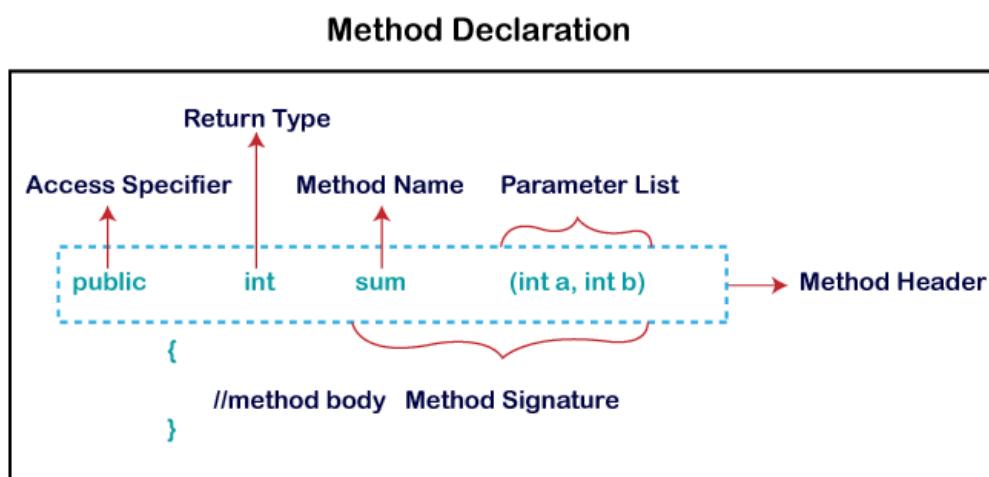
What is a method in Java?

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

The most important method in Java is the **main()** method. If you want to read more about the main() method, go through the link <https://www.javatpoint.com/java-main-method>.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**, as we have shown in the following figure.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction()**. A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length()**, **equals()**, **compareTo()**, **sqrt()**, etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Let's see an example of the predefined method.

Demo.java

```
public class Demo
{
    public static void main(String[] args)
    {
        // using the max() method of Math class
        System.out.print("The maximum number is: " + Math.max(9,7));
    }
}
```

Output:

```
The maximum number is: 9
```

In the above example, we have used three predefined methods **main()**, **print()**, and **max()**. We have used these methods directly without declaration because they are predefined. The **print()** method is a method of **PrintStream** class that prints the result on the console. The **max()** method is a method of the **Math** class that returns the greater of two numbers.

We can also see the method signature of any predefined method by using the link <https://docs.oracle.com/>. When we go through the link and see the **max()** method signature, we find the following:

max
<pre>public static int max(int a, int b)</pre> <p>Returns the greater of two int values. same value.</p> <p>Parameters: a - an argument. b - another argument.</p> <p>Returns: the larger of a and b.</p>

In the above method signature, we see that the method signature has access specifier **public**, non-access modifier **static**, return type **int**, method name **max()**, parameter list (**int a, int b**). In the above example, instead of defining the method, we have just invoked the method. This is the advantage of a predefined method. It makes programming less complicated.

Similarly, we can also see the method signature of the **print()** method.

User-defined Method

The method written by the user or programmer is known as a **user-defined** method. These methods are modified according to the requirement.

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
//user defined method
public static void findEvenOdd(int num)
{
    //method body
```

```

if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}

```

We have defined the above method named `findevenodd()`. It has a parameter `num` of type `int`. The method does not return any value that's why we have used `void`. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number **is even**, else prints the number **is odd**.

How to Call or Invoke a User-defined Method

Once we have defined a method, it should be called. The calling of a method in a program is simple. When we call or invoke a user-defined method, the program control transfer to the called method.

```

import java.util.Scanner;
public class EvenOdd
{
    public static void main (String args[])
    {
        //creating Scanner class object
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        //reading value from the user
        int num=scan.nextInt();
        //method calling
        findEvenOdd(num);
    }
}

```

In the above code snippet, as soon as the compiler reaches at line `findEvenOdd(num)`, the control transfer to the method and gives the output accordingly.

Let's combine both snippets of codes in a single program and execute it.

EvenOdd.java

```

import java.util.Scanner;
public class EvenOdd
{
    public static void main (String args[])
    {
        //creating Scanner class object
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        //reading value from user
        int num=scan.nextInt();
        //method calling
        findEvenOdd(num);
    }
    //user defined method
    public static void findEvenOdd(int num)

```

```
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

Output 1:

```
Enter the number: 12
12 is even
```

Output 2:

```
Enter the number: 99
99 is odd
```

Let's see another program that return a value to the calling method.

In the following program, we have defined a method named **add()** that sum up the two numbers. It has two parameters **n1** and **n2** of integer type. The values of **n1** and **n2** correspond to the value of **a** and **b**, respectively. Therefore, the method adds the value of **a** and **b** and store it in the variable **s** and returns the sum.

Addition.java

```
public class Addition
{
public static void main(String[] args)
{
int a = 19;
int b = 5;
//method calling
int c = add(a, b); //a and b are actual parameters
System.out.println("The sum of a and b is= " + c);
}
//user defined method
public static int add(int n1, int n2) //n1 and n2 are formal parameters
{
int s;
s=n1+n2;
return s; //returning the sum
}
}
```

Output:

```
The sum of a and b is= 24
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword **static** before the method name.

The main advantage of a static method is that we can call it without creating an object. It can access static data members and also change the value of it. It is used to create an instance method. It is invoked by using the class name. The best example of a static method is the **main()** method.

Example of static method

Display.java

```
public class Display
{
    public static void main(String[] args)
    {
        show();
    }

    static void show()
    {
        System.out.println("It is an example of static method.");
    }
}
```

Output:

It is an example of a static method.

Instance Method

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

InstanceMethodExample.java

```
public class InstanceMethodExample
{
    public static void main(String [] args)
    {
        //Creating an object of the class
        InstanceMethodExample obj = new InstanceMethodExample();
        //invoking instance method
        System.out.println("The sum is: "+obj.add(12, 13));
    }

    int s;
    //user-defined method because we have not used static keyword
    public int add(int a, int b)
    {
        s = a+b;
    }
}
```

```
//returning the sum
return s;
}
}
```

Output:

The sum is: 25

There are two types of instance method:

- **Accessor Method**
- **Mutator Method**

Accessor Method: The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word **get**. It is also known as **getters**. It returns the value of the private field. It is used to get the value of the private field.

Example

```
public int getId()
{
    return Id;
}
```

Mutator Method: The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word **set**. It is also known as **setters** or **modifiers**. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Example

```
public void setRoll(int roll)
{
    this.roll = roll;
}
```

Example of accessor and mutator method

Student.java

```
public class Student
{
    private int roll;
    private String name;
    public int getRoll() //accessor method
    {
        return roll;
    }
    public void setRoll(int roll) //mutator method
    {
        this.roll = roll;
    }
}
```

```

public String getName()
{
    return name;
}

public void setName(String name)
{
    this.name = name;
}

public void display()
{
    System.out.println("Roll no.: "+roll);
    System.out.println("Student name: "+name);
}

```

Abstract Method

The method that does not have method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the **abstract class**. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword **abstract**.

Syntax

```
abstract void method_name();
```

Example of abstract method

Demo.java

```

abstract class Demo //abstract class
{
    //abstract method declaration
    abstract void display();
}

public class MyClass extends Demo
{
    //method implementation
    void display()
    {
        System.out.println("Abstract method?");
    }

    public static void main(String args[])
    {
        //creating object of abstract class
        Demo obj = new MyClass();
        //invoking abstract method
        obj.display();
    }
}
```

Output:

```
Abstract method...
```

Factory method

It is a method that returns an object to the class to which it belongs. All static methods are factory methods. For example,

```
NumberFormat obj = NumberFormat.getNumberInstance();
```

[← Prev](#)
[Next →](#)

[Youtube](#) For Videos Join Our Youtube Channel: [Join Now](#)

Help Others, Please Share



Learn Latest Tutorials

SoapUI tutorial SoapUI	RPA tutorial RPA	manual testing tutorial Manual T.	cucumber tutorial Cucumber
Appium tutorial Appium	postgresql tutorial PostgreSQL	Apache Solr Tutorial Solr	MongoDB tutorial MongoDB
Gimp Tutorial Gimp	Verilog Tutorial Verilog	Teradata Tutorial Teradata	PhoneGap Tutorial PhoneGap

Preparation

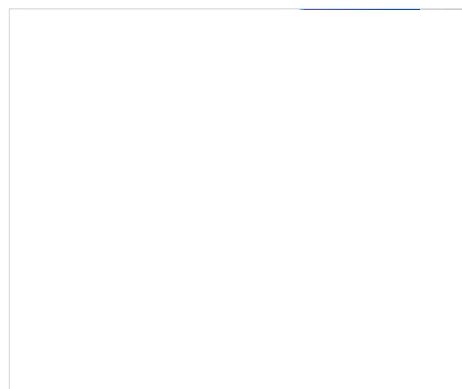
Aptitude	Logical Reasoning	Verbal Ability	Interview Questions
--------------------------	-----------------------------------	--------------------------------	-------------------------------------

Constructors in Java

In **Java**, a constructor is a block of codes similar to the method. It is called when an instance of the **class** 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.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

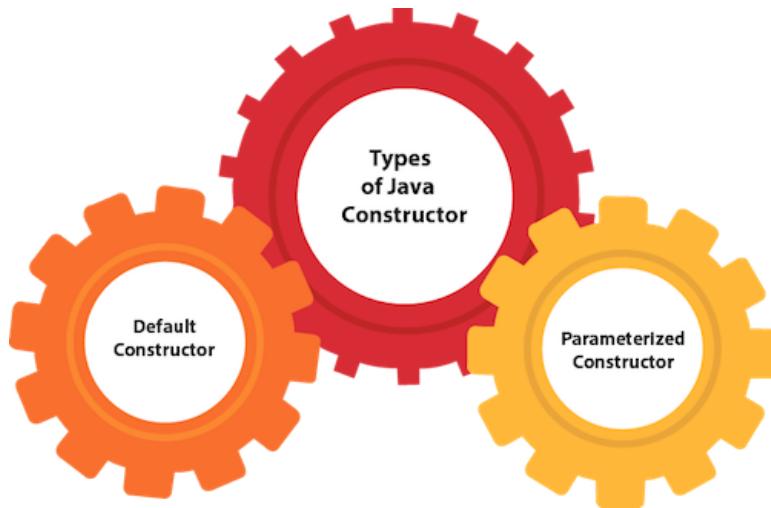
1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use **access modifiers** while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

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



ⓘ X

Get Ready For Your Dream Jo

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.

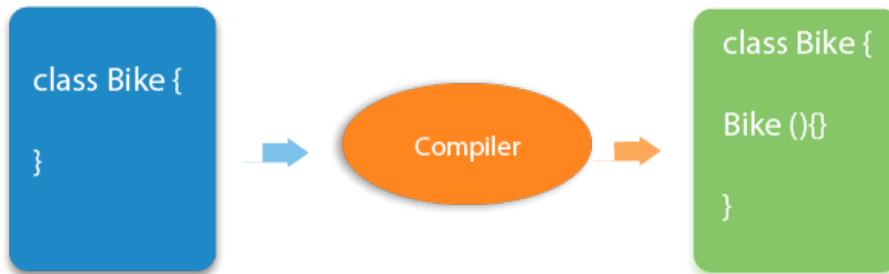
```
//Java Program to create and call a default constructor
class Bike1{
    //creating a default constructor
    Bike1(){System.out.println("Bike is created");}
    //main method
    public static void main(String args[]){
        //calling a default constructor
        Bike1 b=new Bike1();
    }
}
```

[Test it Now](#)

Output:

```
Bike is created
```

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



Q) What is the 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

```

//Let us see another example of default constructor
//which displays the default values
class Student3{
    int id;
    String name;
    //method to display the value of id and name
    void display(){System.out.println(id+ " "+name);}

    public static void main(String args[]){
        //creating objects
        Student3 s1=new Student3();
        Student3 s2=new Student3();
        //displaying values of the object
        s1.display();
        s2.display();
    }
}
  
```

[Test it Now](#)

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.

Why use the 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.

```
//Java Program to demonstrate the use of the parameterized constructor.

class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i, String n){
        id = i;
        name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        //creating objects and passing values
        Student4 s1 = new Student4(111, "Karan");
        Student4 s2 = new Student4(222, "Aryan");
        //calling method to display the values of object
        s1.display();
        s2.display();
    }
}
```

Test it Now

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** 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

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
        id = i;
        name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
        id = i;
        name = n;
        age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}
}

public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
}
```

[Test it Now](#)

Output:

```
111 Karan 0
222 Aryan 25
```

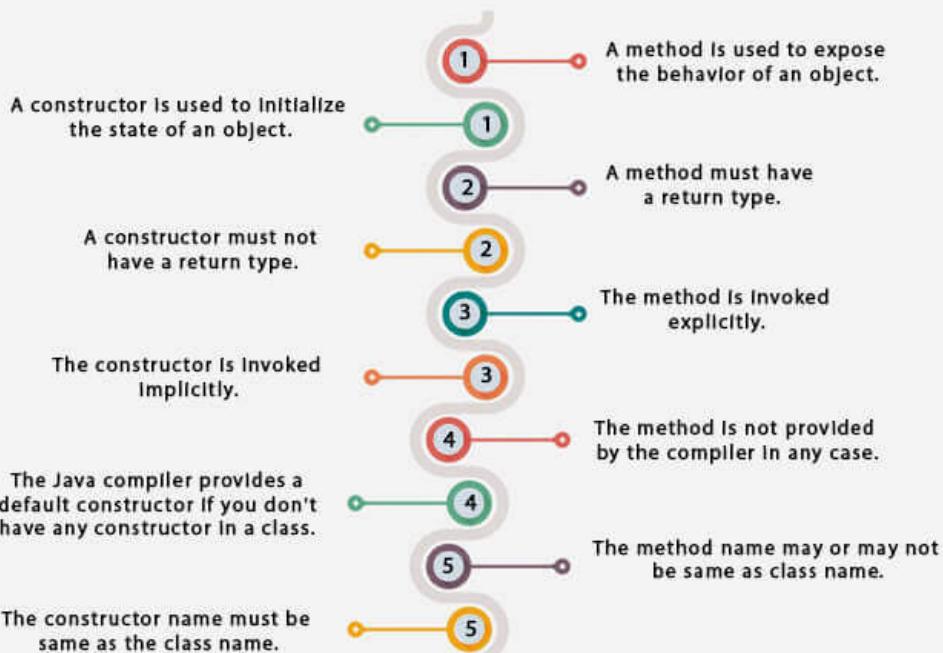
Difference between constructor and method in Java

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.

Difference between constructor and method in Java



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:

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

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

```
//Java program to initialize the values from one object to another object.

class Student6{
    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i, String n){
        id = i;
        name = n;
    }
}
```

```

}

//constructor to initialize another object
Student6(Student6 s){
    id = s.id;
    name = s.name;
}
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
}
}

```

Test it Now

Output:

```

111 Karan
111 Karan

```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```

class Student7{
    int id;
    String name;
    Student7(int i,String n){
        id = i;
        name = n;
    }
    Student7(){}
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        Student7 s1 = new Student7(111,"Karan");
        Student7 s2 = new Student7();
        s2.id=s1.id;
        s2.name=s1.name;
        s1.display();
        s2.display();
    }
}

```

Test it Now

Output:

```
111 Karan  
111 Karan
```

Q) Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Is there Constructor class in Java?

Yes.

What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the `java.lang.reflect` package.

← Prev

Next →

 For Videos Join Our YouTube Channel: [Join Now](#)

Help Others, Please Share



Learn Latest Tutorials



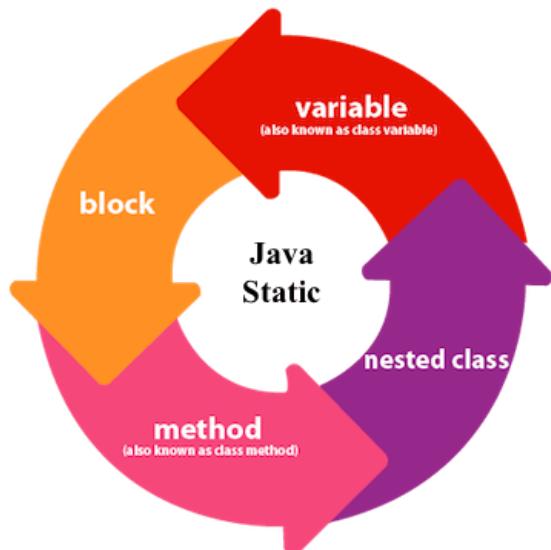


Java static keyword

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with **variables**, methods, blocks and **nested classes**. The static keyword belongs to the class than an instance of the class.

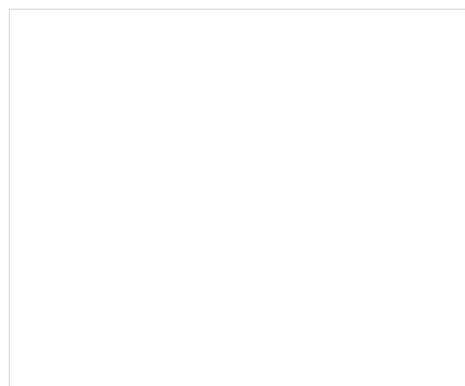
The static can be:

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.



- o 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.
- o 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

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

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**. 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

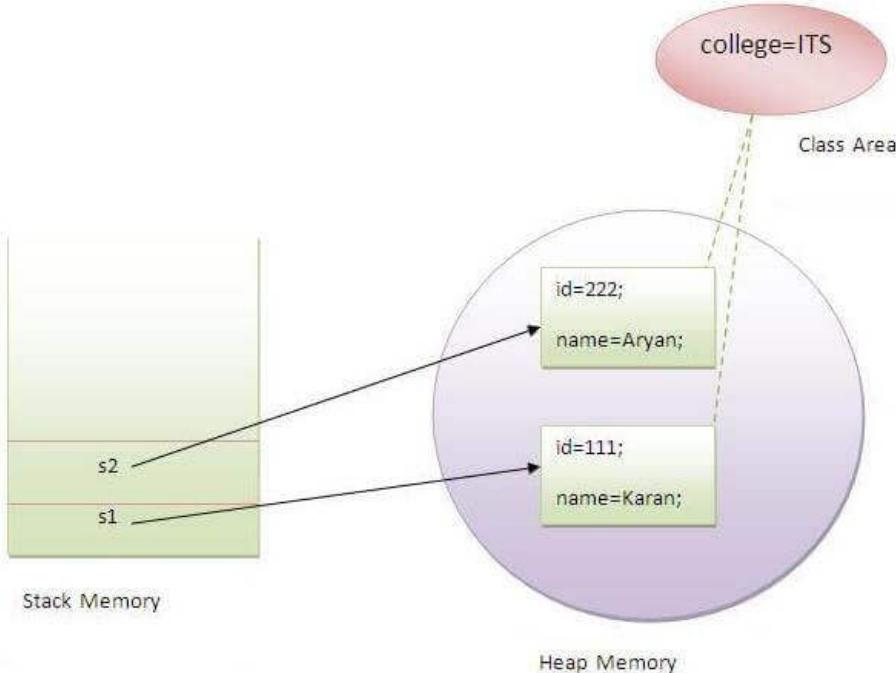
```
//Java Program to demonstrate the use of static variable
class Student{
    int rollno;//instance variable
    String name;
    static String college ="ITS";//static variable
    //constructor
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display the values
    void display (){System.out.println(rollno+" "+name+" "+college);}
}

//Test class to show the values of objects
public class TestStaticVariable1{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        //we can change the college of all objects by the single line of code
        //Student.college="BBDIT";
        s1.display();
        s2.display();
    }
}
```

[Test it Now](#)

Output:

```
111 Karan ITS
222 Aryan ITS
```



Get Ready For Your Dream Job

GeeksforGeeks

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.

```
//Java Program to demonstrate the use of an instance variable
//which get memory each time when we create an object of the class.

class Counter{
    int count=0;//will get memory each time when the instance is created

    Counter(){
        count++;//incrementing value
        System.out.println(count);
    }

    public static void main(String args[]){
        //Creating objects
    }
}
```

```
Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
}
```

[Test it Now](#)

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.

```
//Java Program to illustrate the use of static variable which
//is shared with all objects.

class Counter2{
    static int count=0;//will get memory only once and retain its value

    Counter2(){
        count++;//incrementing the value of static variable
        System.out.println(count);
    }

    public static void main(String args[]){
        //creating objects
        Counter2 c1=new Counter2();
        Counter2 c2=new Counter2();
        Counter2 c3=new Counter2();
    }
}
```

[Test it Now](#)

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

```
//Java Program to demonstrate the use of a static method.

class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
        college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}

//Test class to create and display the values of object
public class TestStaticMethod{
    public static void main(String args[]){
        Student.change(); //calling change method
        //creating objects
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        Student s3 = new Student(333,"Sonoo");
        //calling display method
        s1.display();
        s2.display();
        s3.display();
    }
}
```

Test it Now

```
Output:111 Karan BBDIT
      222 Aryan BBDIT
      333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

//Java Program to get the cube of a given number using the static method

```
class Calculate{
    static int cube(int x){
        return x*x*x;
    }

    public static void main(String args[]){
        int result=Calculate.cube(5);
        System.out.println(result);
    }
}
```

[Test it Now](#)

Output:125

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A{
    int a=40;//non static

    public static void main(String args[]){
        System.out.println(a);
    }
}
```

[Test it Now](#)

Output:Compile Time Error

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
class A2{
    static{System.out.println("static block is invoked");}
}
```

```
public static void main(String args[]){
    System.out.println("Hello main");
}
```

[Test it Now](#)

Output:static block is invoked
Hello main

Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the **main** method.

```
class A3{
    static{
        System.out.println("static block is invoked");
        System.exit(0);
    }
}
```

[Test it Now](#)

Output:

static block is invoked

Since JDK 1.7 and above, output would be:

```
Error: Main method not found in class A3, please define the main method as:
 public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

[← Prev](#)

[Next →](#)

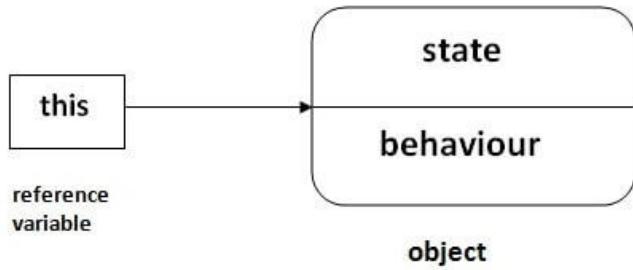
 [For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



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.

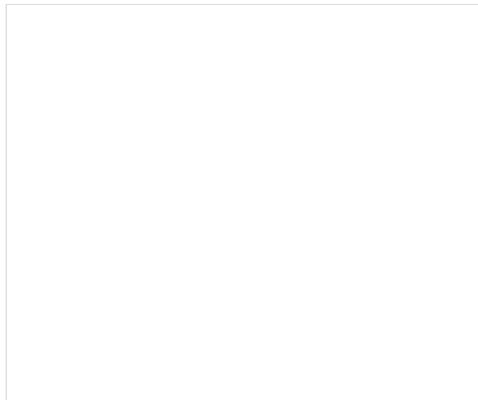


Usage of java this keyword

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.

Suggestion: If you are beginner to java, lookup only three usage of this keyword.



Usage of java this keyword

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

- 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.



X

Test in Under 15 Minutes

TestProject

Sic

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:

```
class Student{
    int rollno;
    String name;
```

```

float fee;
Student(int rollno, String name, float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111, "ankit", 5000f);
Student s2=new Student(112, "sumit", 6000f);
s1.display();
s2.display();
}}

```

Test it Now

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

```

class Student{
int rollno;
String name;
float fee;
Student(int rollno, String name, float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

```

```

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111, "ankit", 5000f);
Student s2=new Student(112, "sumit", 6000f);
s1.display();
s2.display();
}}

```

}}

Test it Now

Output:

```
111 ankit 5000
112 sumit 6000
```

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

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int r, String n, float f){
        rollno=r;
        name=n;
        fee=f;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis3{
    public static void main(String args[]){
        Student s1=new Student(111, "ankit", 5000f);
        Student s2=new Student(112, "sumit", 6000f);
        s1.display();
        s2.display();
    }
}
```

Test it Now

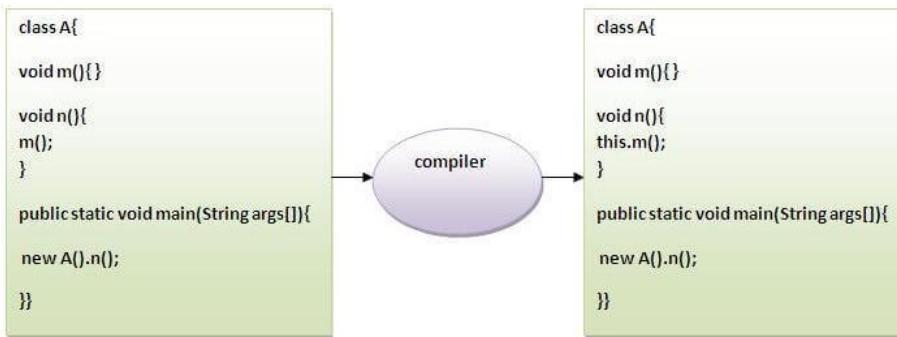
Output:

```
111 ankit 5000
112 sumit 6000
```

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

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



```

class A{
    void m(){System.out.println("hello m");}
    void n(){
        System.out.println("hello n");
        //m();//same as this.m()
        this.m();
    }
}
class TestThis4{
    public static void main(String args[]){
        A a=new A();
        a.n();
    }
}

```

Test it Now

Output:

```

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:

```

class A{
    A(){System.out.println("hello a");}
    A(int x){
        this();
        System.out.println(x);
    }
}

```

```
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

[Test it Now](#)

Output:

```
hello a
10
```

Calling parameterized constructor from default constructor:

```
class A{
A(){}
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

[Test it Now](#)

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.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
```

```

this.course=course;
}

Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}

void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}

}

class TestThis7{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}
}

```

Test it Now

Output:

```

111 ankit java null
112 sumit java 6000

```

Rule: Call to this() must be the first statement in constructor.

```

class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this.fee=fee;
this(rollno,name,course);//C.T.Error
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}

class TestThis8{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
}
}

```

```
s2.display();
}
```

[Test it Now](#)

Compile Time Error: Call to this must be first statement in constructor

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:

```
class S2{
    void m(S2 obj){
        System.out.println("method is invoked");
    }
    void p(){
        m(this);
    }
    public static void main(String args[]){
        S2 s1 = new S2();
        s1.p();
    }
}
```

[Test it Now](#)

Output:

method is invoked

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

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:

```
class B{
    A4 obj;
    B(A4 obj){
        this.obj=obj;
    }
    void display(){
        System.out.println(obj.data);//using data member of A4 class
    }
}
```

```

}

class A4{
    int data=10;
    A4(){
        B b=new B(this);
        b.display();
    }
    public static void main(String args[]){
        A4 a=new A4();
    }
}

```

[Test it Now](#)

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

```

return_type method_name(){
    return this;
}

```

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

```

class A{
    A getA(){
        return this;
    }
    void msg(){System.out.println("Hello java");}
}
class Test1{
    public static void main(String args[]){
        new A().getA().msg();
    }
}

```

[Test it Now](#)

Output:

```
Hello java
```

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

```
class A5{  
    void m(){  
        System.out.println(this); //prints same reference ID  
    }  
    public static void main(String args[]){  
        A5 obj=new A5();  
        System.out.println(obj); //prints the reference ID  
        obj.m();  
    }  
}
```

[Test it Now](#)

Output:

```
A5@22b3ea59  
A5@22b3ea59
```

[← Prev](#)

[Next →](#)

 [For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



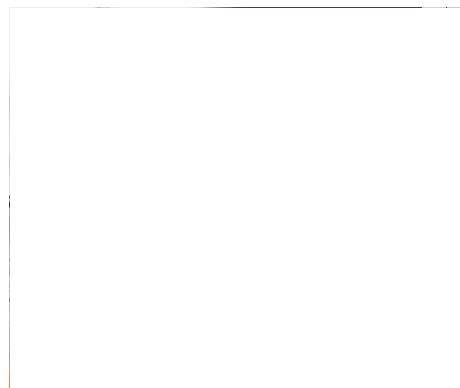
Learn Latest Tutorials

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of **OOPS** (Object Oriented programming system).

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

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.



Why use inheritance in java

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

Terms used in Inheritance

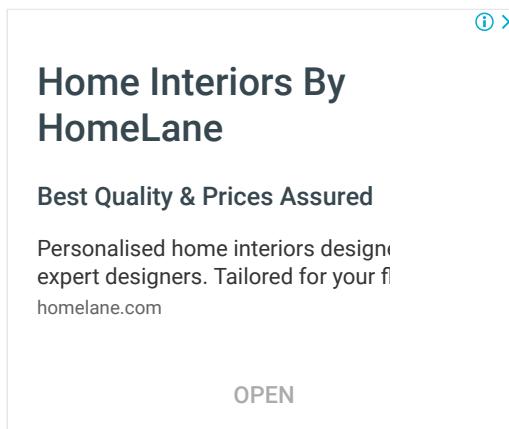
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

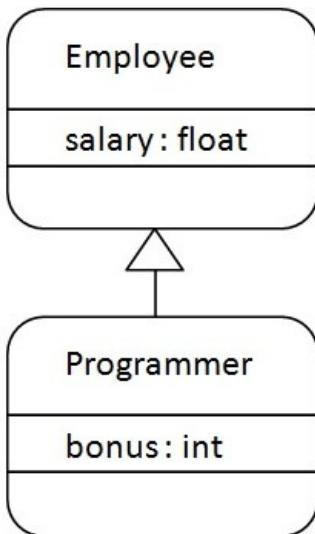
```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

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

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



Java Inheritance Example



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

```

class Employee{
    float salary=40000;
}

class Programmer extends Employee{
    int bonus=10000;

    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}
  
```

[Test it Now](#)

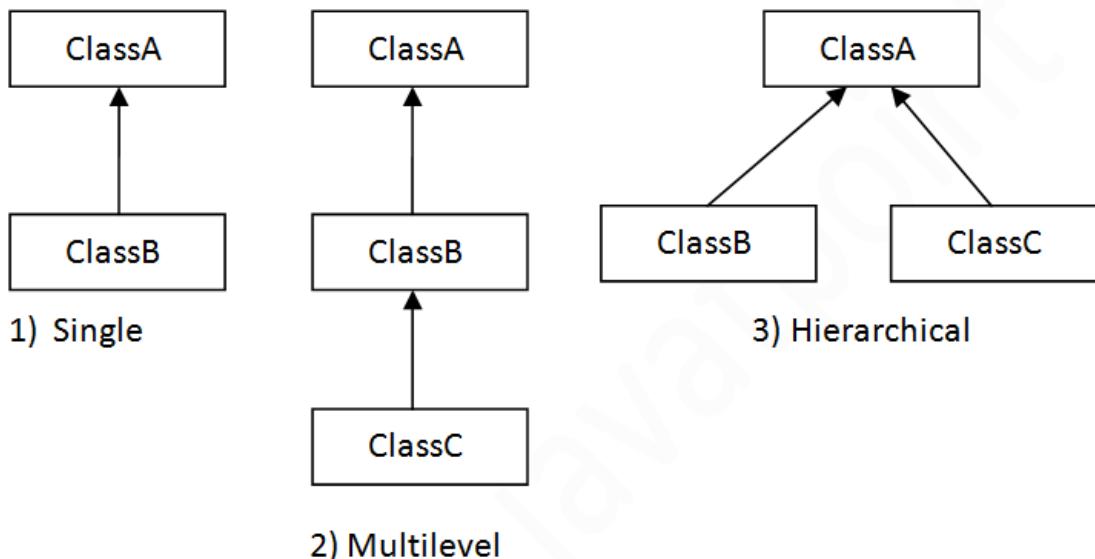
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

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

Types of inheritance in java

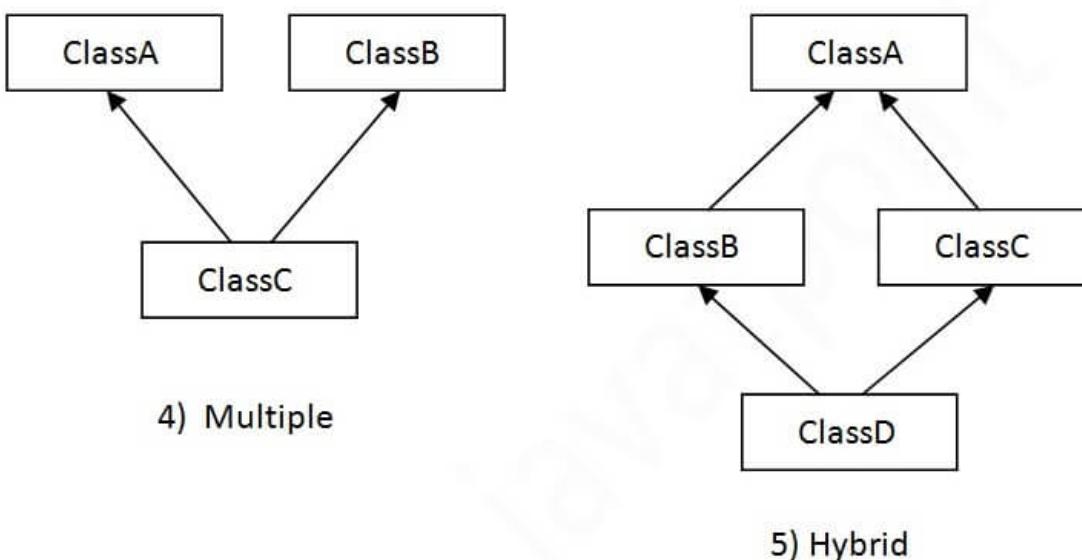
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

```
class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}

class TestInheritance{
    public static void main(String args[]){
        Dog d=new Dog();
        d.bark();
        d.eat();
    }
}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

```
class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}

class BabyDog extends Dog{
    void weep(){System.out.println("weeping...");}
}

class TestInheritance2{
    public static void main(String args[]){
        BabyDog d=new BabyDog();
        d.weep();
        d.bark();
        d.eat();
    }
}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}

class Cat extends Animal{
    void meow(){System.out.println("meowing...");}
}

class TestInheritance3{
    public static void main(String args[]){
        Cat c=new Cat();
        c.meow();
        c.eat();
        //c.bark(); //C.T.Error
    }
}
```

Output:

```
meowing...
eating...
```

Q) Why multiple inheritance is not supported in java?

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

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

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

```
class A{
void msg(){System.out.println("Hello");}
}

class B{
void msg(){System.out.println("Welcome");}
}

class C extends A,B{//suppose if it were

public static void main(String args[]){
    C obj=new C();
    obj.msg();//Now which msg() method would be invoked?
}
}
```

[Test it Now](#)

[Compile Time Error](#)

[← Prev](#)

[Next →](#)

 [For Videos Join Our YouTube Channel: Join Now](#)

Help Others, Please Share



Learn Latest Tutorials

 SoapUI tutorial

SoapUI

 RPA tutorial

RPA

 manual testing tutorial

Manual T.

 cucumber tutorial

Cucumber

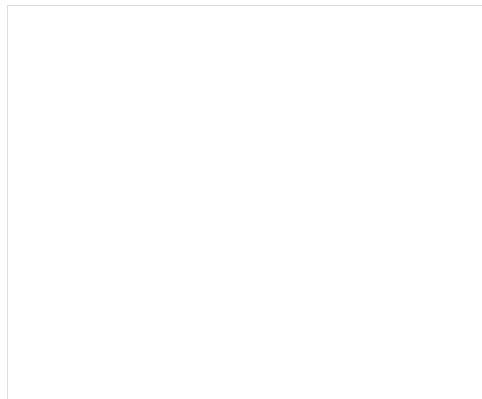
Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

```
class Employee{
    int id;
    String name;
    Address address;//Address is a class
    ...
}
```

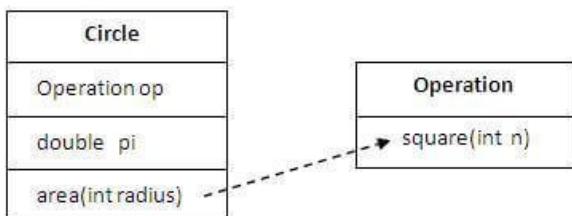
In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.



Why use Aggregation?

- For Code Reusability.

Simple Example of Aggregation



In this example, we have created the reference of Operation class in the Circle class.

```
class Operation{
    int square(int n){
        return n*n;
    }
}

class Circle{
    Operation op;//aggregation
    double pi=3.14;

    double area(int radius){
        op=new Operation();
        int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
        return pi*rsquare;
    }

    public static void main(String args[]){
        Circle c=new Circle();
        double result=c.area(5);
        System.out.println(result);
    }
}
```

Test it Now

Output:78.5

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Understanding meaningful example of Aggregation

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

Address.java

```
public class Address {
    String city,state,country;

    public Address(String city, String state, String country) {
        this.city = city;
        this.state = state;
        this.country = country;
    }

}
```

Emp.java

```
public class Emp {
    int id;
    String name;
    Address address;

    public Emp(int id, String name,Address address) {
        this.id = id;
        this.name = name;
        this.address=address;
    }

    void display(){
        System.out.println(id+" "+name);
        System.out.println(address.city+" "+address.state+" "+address.country);
    }

    public static void main(String[] args) {
        Address address1=new Address("gzb","UP","india");
        Address address2=new Address("gno","UP","india");

        Emp e=new Emp(111,"varun",address1);
        Emp e2=new Emp(112,"arun",address2);

        e.display();
        e2.display();

    }
}
```

Test it Now

```
Output:111 varun
      gzb UP india
```

112 arun
gno UP india

[download this example](#)

← Prev

Next →

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Learn Latest Tutorials

 SoapUI tutorial SoapUI	 RPA tutorial RPA	 manual testing tutorial Manual T.	 cucumber tutorial Cucumber
 Appium tutorial Appium	 postgresql tutorial PostgreSQL	 Apache Solr Tutorial Solr	 MongoDB tutorial MongoDB
 Gimp Tutorial Gimp	 Verilog Tutorial Verilog	 Teradata Tutorial Teradata	 PhoneGap Tutorial PhoneGap

Preparation

 Aptitude	 Logical Reasoning	 Verbal Ability	 Interview Questions
Aptitude	Reasoning	Verbal A.	Interview

Method Overloading in Java

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

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

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

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

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

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



In Java, Method Overloading is not possible by changing the return type of the method only.

[Learn more](#)

1) Method Overloading: changing no. of arguments

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

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

```
class Adder{
    static int add(int a,int b){return a+b;}
    static int add(int a,int b,int c){return a+b+c;}
}

class TestOverloading1{
    public static void main(String[] args){
        System.out.println(Adder.add(11,11));
        System.out.println(Adder.add(11,11,11));
    }
}
```

[Test it Now](#)

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

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

```
class Adder{
    static int add(int a, int b){return a+b;}
    static double add(double a, double b){return a+b;}
}

class TestOverloading2{
    public static void main(String[] args){
        System.out.println(Adder.add(11,11));
        System.out.println(Adder.add(12.3,12.6));
    }
}
```

[Test it Now](#)

Output:

```
22
24.9
```

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

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

```
class Adder{
    static int add(int a,int b){return a+b;}
    static double add(int a,int b){return a+b;}
}

class TestOverloading3{
    public static void main(String[] args){
        System.out.println(Adder.add(11,11));//ambiguity
    }
}
```

Test it Now

Output:

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

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

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

Can we overload java main() method?

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

```
class TestOverloading4{
    public static void main(String[] args){System.out.println("main with String[]");}
    public static void main(String args){System.out.println("main with String");}
    public static void main(){System.out.println("main without args");}
}
```

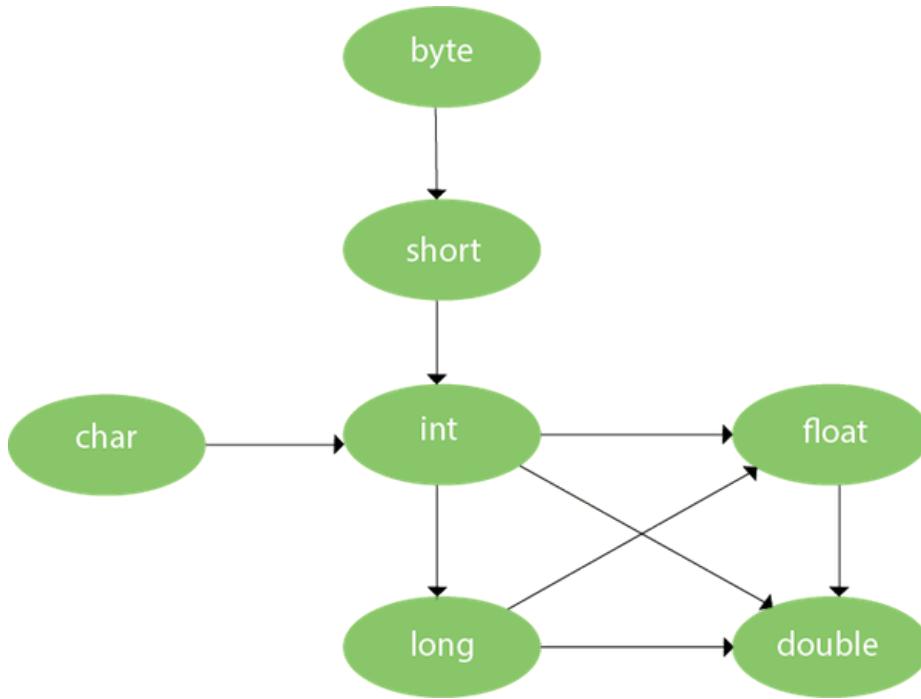
Test it Now

Output:

```
main with String[]
```

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```

class OverloadingCalculation1{
    void sum(int a,long b){System.out.println(a+b);}
    void sum(int a,int b,int c){System.out.println(a+b+c);}

    public static void main(String args[]){
        OverloadingCalculation1 obj=new OverloadingCalculation1();
        obj.sum(20,20); //now second int literal will be promoted to long
        obj.sum(20,20,20);
    }
}
  
```

[Test it Now](#)

Output:
40
60

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```

class OverloadingCalculation2{
    void sum(int a,int b){System.out.println("int arg method invoked");}
    void sum(long a,long b){System.out.println("long arg method invoked");}

    public static void main(String args[]){
        OverloadingCalculation2 obj=new OverloadingCalculation2();
    }
}
  
```

```

    obj.sum(20,20);//now int arg sum() method gets invoked
}
}

```

Test it Now

Output:int arg method invoked

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```

class OverloadingCalculation3{
    void sum(int a,long b){System.out.println("a method invoked");}
    void sum(long a,int b){System.out.println("b method invoked");}

    public static void main(String args[]){
        OverloadingCalculation3 obj=new OverloadingCalculation3();
        obj.sum(20,20);//now ambiguity
    }
}

```

Test it Now

Output:Compile Time Error

One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

← Prev

Next →

 [For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

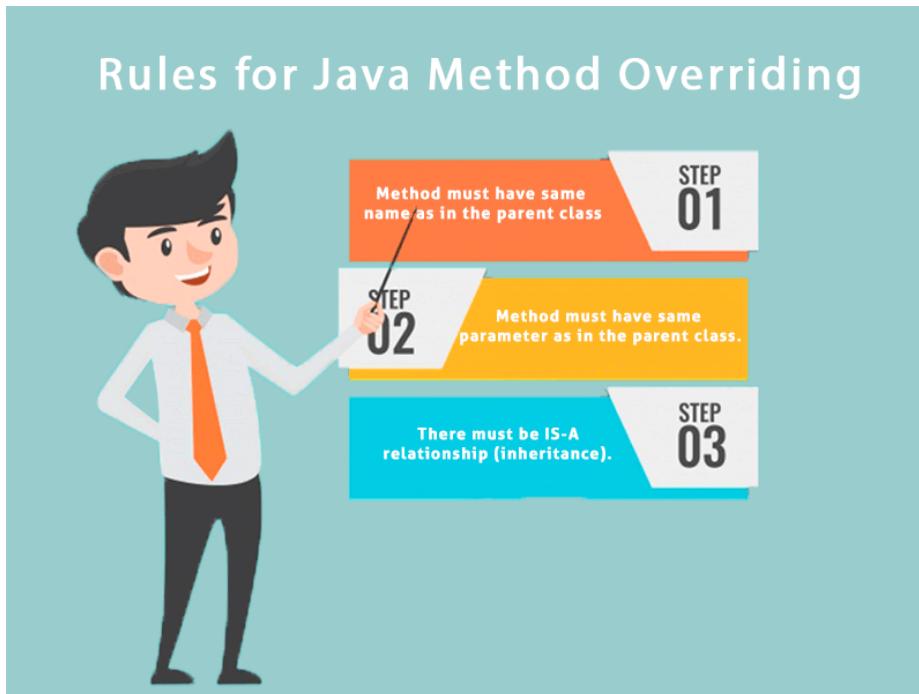
In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).



Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
//Java Program to demonstrate why we need method overriding  
//Here, we are calling the method of parent class with child  
//class object.  
//Creating a parent class
```

```

class Vehicle{
    void run(){System.out.println("Vehicle is running");}
}

//Creating a child class
class Bike extends Vehicle{
    public static void main(String args[]){
        //creating an instance of child class
        Bike obj = new Bike();
        //calling the method with child class instance
        obj.run();
    }
}

```

Test it Now

Output:

```
Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```

//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
    //defining a method
    void run(){System.out.println("Vehicle is running");}
}

//Creating a child class
class Bike2 extends Vehicle{
    //defining the same method as in the parent class
    void run(){System.out.println("Bike is running safely");}
}

public static void main(String args[]){
    Bike2 obj = new Bike2(); //creating object
    obj.run(); //calling method
}

```

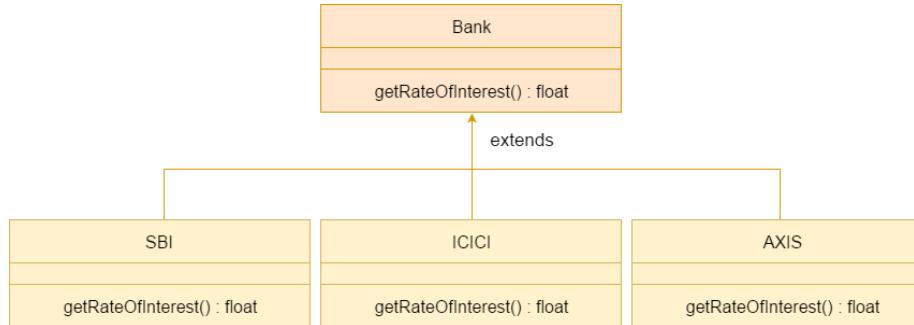
Test it Now

Output:

Bike is running safely

A real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.



Java method overriding is mostly used in Runtime Polymorphism which we will learn in next pages.

```

//Java Program to demonstrate the real scenario of Java Method Overriding
//where three classes are overriding the method of a parent class.

//Creating a parent class.
class Bank{
    int getRateOfInterest(){return 0;}
}

//Creating child classes.
class SBI extends Bank{
    int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
    int getRateOfInterest(){return 7;}
}

class AXIS extends Bank{
    int getRateOfInterest(){return 9;}
}

//Test class to create objects and call the methods
class Test2{
    public static void main(String args[]){
        SBI s=new SBI();
        ICICI i=new ICICI();
        AXIS a=new AXIS();
        System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
    }
}
  
```

```
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

[Test it Now](#)

Output:

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

Difference between method Overloading and Method Overriding in java

[Click me for the difference between method overloading and overriding](#)

More topics on Method Overriding (Not For Beginners)

[Method Overriding with Access Modifier](#)

Let's see the concept of method overriding with access modifier.

[Exception Handling with Method Overriding](#)

Let's see the concept of method overriding with exception handling.

← Prev

Next →

 [For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

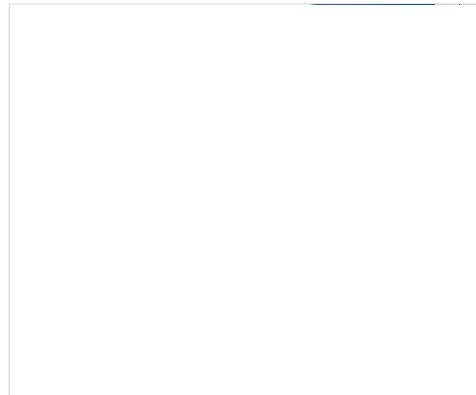
Simple example of Covariant Return Type

```
class A{  
A get(){return this;}  
}  
  
class B1 extends A{  
B1 get(){return this;}  
void message(){System.out.println("welcome to covariant return type");}  
  
public static void main(String args[]){  
new B1().get().message();  
}  
}
```

Test it Now

Output:welcome to covariant return type

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.



How is Covariant return types implemented?

Java doesn't allow the return type based overloading but JVM always allows return type based overloading. JVM uses full signature of a method for lookup/resolution. Full signature means it includes return type in addition to argument types. i.e., a class can have two or more methods differing only by return type. javac uses this fact to implement covariant return types.

<< Prev

Next>>

[Youtube](#) For Videos Join Our Youtube Channel: [Join Now](#)

Help Others, Please Share



(i) X

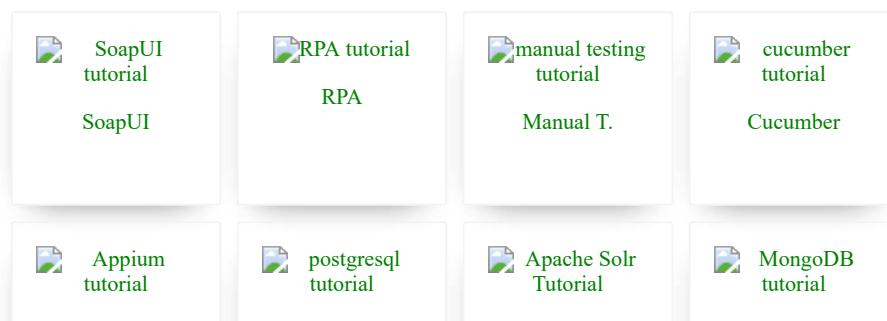
Test in Under 15 Minutes

TestProject makes open source testing efficient. No fr
headaches required.

TestProject

S

Learn Latest Tutorials



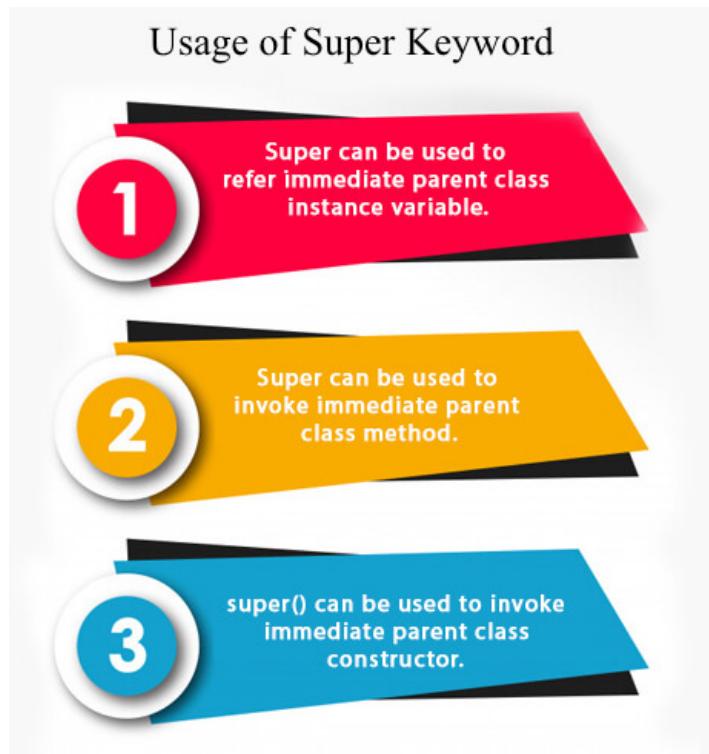
Super Keyword in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.



- 1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```

class Animal{
    String color="white";
}
class Dog extends Animal{
    String color="black";
    void printColor(){
        System.out.println(color);//prints color of Dog class
        System.out.println(super.color);//prints color of Animal class
    }
}
class TestSuper1{
    public static void main(String args[]){
        Dog d=new Dog();
        d.printColor();
    }
}

```

[Test it Now](#)

Output:

```

black
white

```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

Competitive Programming Live

Course Mentored & Guided By Programming Experts
Who Are Highly Ranked At Competitive Sites
GeeksforGeeks

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```

class Animal{
    void eat(){System.out.println("eating...");}
}

```

```

}

class Dog extends Animal{
    void eat(){System.out.println("eating bread...");}
    void bark(){System.out.println("barking...");}
    void work(){
        super.eat();
        bark();
    }
}

class TestSuper2{
    public static void main(String args[]){
        Dog d=new Dog();
        d.work();
    }
}

```

Test it Now**Output:**

```

eating...
barking...

```

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```

class Animal{
    Animal(){System.out.println("animal is created");}
}

class Dog extends Animal{
    Dog(){
        super();
        System.out.println("dog is created");
    }
}

class TestSuper3{
    public static void main(String args[]){
        Dog d=new Dog();
    }
}

```

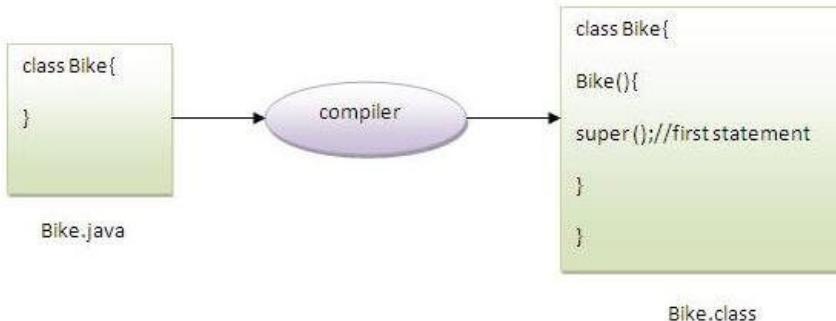
Test it Now**Output:**

```

animal is created
dog is created

```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

```
class Animal{  
Animal(){System.out.println("animal is created");}  
}  
  
class Dog extends Animal{  
Dog(){  
System.out.println("dog is created");  
}  
}  
  
class TestSuper4{  
public static void main(String args[]){  
Dog d=new Dog();  
}}}
```

Test it Now

Output:

animal is created
dog is created

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
class Person{  
    int id;  
    String name;  
    Person(int id, String name){  
        this.id=id;  
        this.name=name;  
    }  
}
```

```

}

class Emp extends Person{
    float salary;
    Emp(int id, String name, float salary){
        super(id, name); //reusing parent constructor
        this.salary=salary;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}

class TestSuper5{
    public static void main(String[] args){
        Emp e1=new Emp(1, "ankit", 45000f);
        e1.display();
    }
}

```

[Test it Now](#)

Output:

1 ankit 45000

[← Prev](#)[Next →](#)

[Youtube For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Learn Latest Tutorials



Instance initializer block

Instance Initializer block is used to initialize the instance data member. It runs each time when object of the class is created.

The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Ques) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

```
class Bike{  
    int speed=100;  
}
```

Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

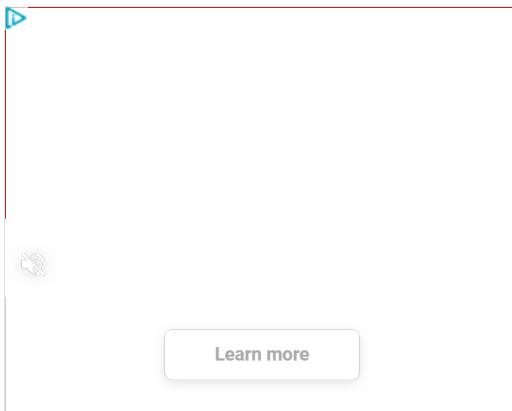
Example of instance initializer block

Let's see the simple example of instance initializer block that performs initialization.

```
class Bike7{  
    int speed;  
  
    Bike7(){System.out.println("speed is "+speed);}  
  
    {speed=100;}  
  
    public static void main(String args[]){  
        Bike7 b1=new Bike7();  
        Bike7 b2=new Bike7();  
    }  
}
```

Test it Now

```
Output:speed is 100  
      speed is 100
```



There are three places in java where you can perform operations:

1. method
2. constructor
3. block

What is invoked first, instance initializer block or constructor?

```
class Bike8{  
    int speed;  
  
    Bike8(){System.out.println("constructor is invoked");}  
  
    {System.out.println("instance initializer block invoked");}  
  
    public static void main(String args[]){  
        Bike8 b1=new Bike8();  
        Bike8 b2=new Bike8();  
    }  
}
```

Test it Now

```
Output:instance initializer block invoked  
       constructor is invoked  
       instance initializer block invoked  
       constructor is invoked
```

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance initializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below:

Note: The java compiler copies the code of instance initializer block in every constructor.

```
class B{
B(){
System.out.println("constructor");}
}
{System.out.println("instance initializer block");}
}
```



Compiler

```
class B{
B(){
super();
{System.out.println("instance initializer block");}
System.out.println("constructor");}
}
}
```

Rules for instance initializer block :

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

Program of instance initializer block that is invoked after super()

```
class A{
A(){
System.out.println("parent class constructor invoked");
}
}

class B2 extends A{
B2(){
super();
System.out.println("child class constructor invoked");
}

{System.out.println("instance initializer block is invoked");}
}
```

```
public static void main(String args[]){
B2 b=new B2();
}
}
```

Test it Now

```
Output:parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked
```

Another example of instance block

```
class A{
A(){}
System.out.println("parent class constructor invoked");
}

class B3 extends A{
B3(){}
super();
System.out.println("child class constructor invoked");
}

B3(int a){
super();
System.out.println("child class constructor invoked "+a);
}

{System.out.println("instance initializer block is invoked");}

public static void main(String args[]){
B3 b1=new B3();
B3 b2=new B3(10);
}
}
```

Test it Now

```
parent class constructor invoked
instance initializer block is invoked
child class constructor invoked
parent class constructor invoked
instance initializer block is invoked
child class constructor invoked 10
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Java Final Keyword

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

javatpoint.com

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{  
    final int speedlimit=90;//final variable  
    void run(){  
        speedlimit=400;  
    }  
    public static void main(String args[]){  
        Bike9 obj=new Bike9();  
    }  
}
```

```
obj.run();
}
} //end of class
```

[Test it Now](#)

Output:Compile Time Error



[Learn more](#)

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
    final void run(){System.out.println("running");}
}

class Honda extends Bike{
    void run(){System.out.println("running safely with 100kmph");}
}

public static void main(String args[]){
    Honda honda= new Honda();
    honda.run();
}
```

[Test it Now](#)

Output:Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{ }

class Honda1 extends Bike{
    void run(){System.out.println("running safely with 100kmph");}
}

public static void main(String args[]){
    Honda1 honda= new Honda1();
    honda.run();
}
}
```

[Test it Now](#)

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{
    final void run(){System.out.println("running...");}
}

class Honda2 extends Bike{
    public static void main(String args[]){
        new Honda2().run();
    }
}
```

[Test it Now](#)

Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
class Student{
    int id;
    String name;
    final String PAN_CARD_NUMBER;
    ...
}
```

}

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
class Bike10{
    final int speedlimit;//blank final variable

    Bike10(){
        speedlimit=70;
        System.out.println(speedlimit);
    }

    public static void main(String args[]){
        new Bike10();
    }
}
```

Test it Now

Output: 70

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
class A{
    static final int data;//static blank final variable
    static{ data=50;}
    public static void main(String args[]){
        System.out.println(A.data);
    }
}
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
class Bike11{
    int cube(final int n){
        n=n+2;//can't be changed as n is final
        n*n*n;
    }
    public static void main(String args[]){
        Bike11 b=new Bike11();
        b.cube(5);
    }
}
```

{
}[Test it Now](#)

Output: Compile Time Error

Q) Can we declare a constructor final?

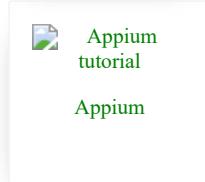
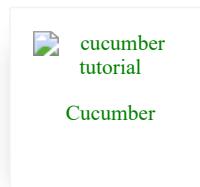
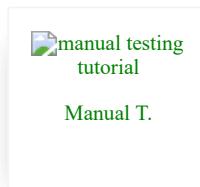
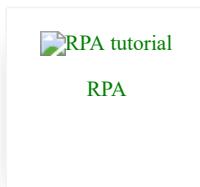
No, because constructor is never inherited.

[← Prev](#)[Next →](#) [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Learn Latest Tutorials



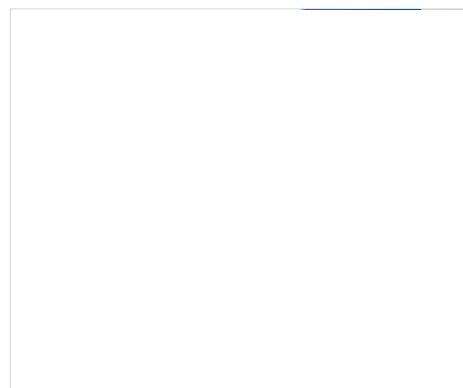
Preparation

Polymorphism in Java

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.



Runtime Polymorphism in Java

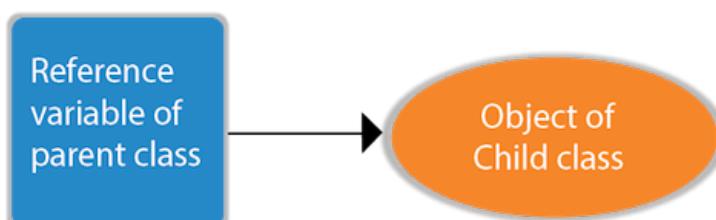
Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



```
class A{}  
class B extends A{}
```

```
A a=new B(); //upcasting
```

For upcasting, we can use the reference variable of class type or an interface type. For Example:

```
interface I{ }
class A{ }
class B extends A implements I{ }
```

Here, the relationship of B class would be:

```
B IS-A A
B IS-A I
B IS-A Object
```

Since Object is the root class of all classes in Java, so we can write B IS-A Object.



Test in Under 15 Minutes

TestProject

Sic

Example of Java Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{
    void run(){System.out.println("running");}
}

class Splendor extends Bike{
    void run(){System.out.println("running safely with 60km");}
}

public static void main(String args[]){
    Bike b = new Splendor(); //upcasting
    b.run();
}
```

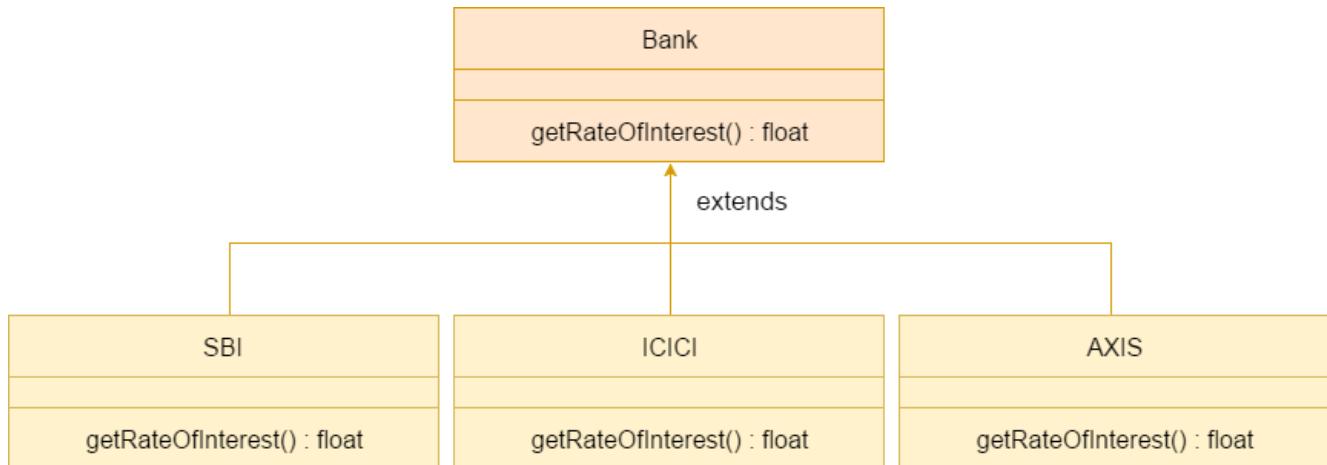
[Test it Now](#)

Output:

```
running safely with 60km.
```

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



Note: This example is also given in method overriding but there was no upcasting.

```

class Bank{
    float getRateOfInterest(){return 0;}
}

class SBI extends Bank{
    float getRateOfInterest(){return 8.4f;}
}

class ICICI extends Bank{
    float getRateOfInterest(){return 7.3f;}
}

class AXIS extends Bank{
    float getRateOfInterest(){return 9.7f;}
}

class TestPolymorphism{
    public static void main(String args[]){
        Bank b;
        b=new SBI();
        System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
        b=new ICICI();
        System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
        b=new AXIS();
        System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
    }
}
  
```

[Test it Now](#)

Output:

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Java Runtime Polymorphism Example: Shape

```
class Shape{
    void draw(){System.out.println("drawing...");}
}

class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle...");}
}

class Circle extends Shape{
    void draw(){System.out.println("drawing circle...");}
}

class Triangle extends Shape{
    void draw(){System.out.println("drawing triangle...");}
}

class TestPolymorphism2{
    public static void main(String args[]){
        Shape s;
        s=new Rectangle();
        s.draw();
        s=new Circle();
        s.draw();
        s=new Triangle();
        s.draw();
    }
}
```

[Test it Now](#)

Output:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

Java Runtime Polymorphism Example: Animal

```
class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
    void eat(){System.out.println("eating bread...");}
}
```

```

}

class Cat extends Animal{
void eat(){System.out.println("eating rat...");}
}

class Lion extends Animal{
void eat(){System.out.println("eating meat...");}
}

class TestPolymorphism3{
public static void main(String[] args){
Animal a;
a=new Dog();
a.eat();
a=new Cat();
a.eat();
a=new Lion();
a.eat();
}}

```

Test it Now

Output:

```

eating bread...
eating rat...
eating meat...

```

Java Runtime Polymorphism with Data Member

A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.

Rule: Runtime polymorphism can't be achieved by data members.

```

class Bike{
int speedlimit=90;
}

class Honda3 extends Bike{
int speedlimit=150;

public static void main(String args[]){
Bike obj=new Honda3();
System.out.println(obj.speedlimit);//90
}

```

Test it Now

Output:

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
class Animal{
    void eat(){System.out.println("eating");}
}

class Dog extends Animal{
    void eat(){System.out.println("eating fruits");}
}

class BabyDog extends Dog{
    void eat(){System.out.println("drinking milk");}
}

public static void main(String args[]){
    Animal a1,a2,a3;
    a1=new Animal();
    a2=new Dog();
    a3=new BabyDog();
    a1.eat();
    a2.eat();
    a3.eat();
}
```

[Test it Now](#)

Output:

```
eating
eating fruits
drinking Milk
```

Try for Output

```
class Animal{
    void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
    void eat(){System.out.println("dog is eating...");}
}

class BabyDog1 extends Dog{
    public static void main(String args[]){
        Animal a=new BabyDog1();
        a.eat();
    }
}
```

[Test it Now](#)

Output:

```
Dog is eating
```

Since, BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked.

← Prev

Next →

 [Youtube For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Learn Latest Tutorials

 SoapUI tutorial SoapUI	 RPA tutorial RPA	 manual testing tutorial Manual T.	 cucumber tutorial Cucumber
 Appium tutorial Appium	 postgresql tutorial PostgreSQL	 Apache Solr Tutorial Solr	 MongoDB tutorial MongoDB
 Gimp Tutorial Gimp	 Verilog Tutorial Verilog	 Teradata Tutorial Teradata	 PhoneGap Tutorial PhoneGap

Preparation

 Aptitude	 Logical Reasoning	 Verbal Ability	 Interview Questions
Reasoning		Verbal A.	Interview

Static Binding and Dynamic Binding

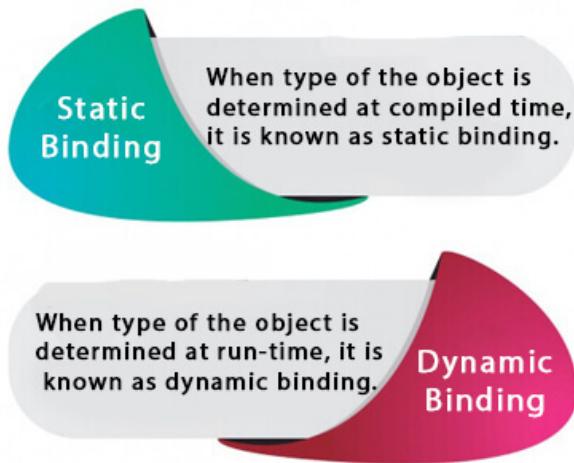
Connecting a method call to the method body is known as binding.



There are two types of binding

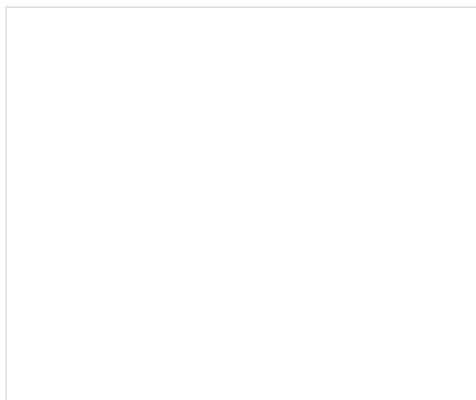
1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

Static vs Dynamic Binding



Understanding Type

Let's understand the type of instance.



1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

```
int data=30;
```

Here data variable is a type of int.

2) References have a type

```
class Dog{
    public static void main(String args[]){
        Dog d1;//Here d1 is a type of Dog
    }
}
```

3) Objects have a type

An object is an instance of particular java class, but it is also an instance of its superclass.

```
class Animal{}

class Dog extends Animal{
    public static void main(String args[]){
        Dog d1=new Dog();
    }
}
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

```
class Dog{
    private void eat(){System.out.println("dog is eating...");}

    public static void main(String args[]){
        Dog d1=new Dog();
        d1.eat();
    }
}
```

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
class Animal{
    void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
    void eat(){System.out.println("dog is eating...");}
}

public static void main(String args[]){
    Animal a=new Dog();
    a.eat();
}
```

[Test it Now](#)

Output:dog is eating...

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So compiler doesn't know its type, only its base type.

[<<Prev](#)

[Next>>](#)

[For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Test in Under 15 Minutes

TestProject makes open source testing efficient. No fridges required.

TestProject

S

Learn Latest Tutorials

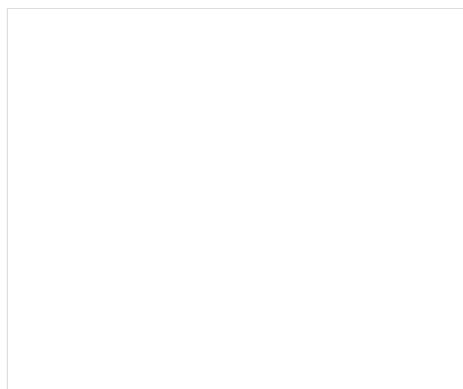
Java instanceof

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Simple example of java instanceof

Let's see the simple example of instance operator where it tests the current class.



```
class Simple1{  
    public static void main(String args[]){  
        Simple1 s=new Simple1();  
        System.out.println(s instanceof Simple1); //true  
    }  
}
```

[Test it Now](#)

Output:true

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.



Get Ready For Your Dream Job

GeeksforGeeks

Another example of java instanceof operator

```
class Animal{}  
class Dog1 extends Animal{//Dog inherits Animal  
  
public static void main(String args[]){  
Dog1 d=new Dog1();  
System.out.println(d instanceof Animal);//true  
}  
}
```

Test it Now

Output:true

instanceof in java with a variable that have null value

If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

```
class Dog2{  
public static void main(String args[]){  
Dog2 d=null;  
System.out.println(d instanceof Dog2);//false  
}  
}
```

Test it Now

Output:false

Downcasting with java instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

```
Dog d=new Animal();//Compilation error
```

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

```
Dog d=(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime
```

Possibility of downcasting with instanceof

Let's see the example, where downcasting is possible by instanceof operator.

```
class Animal { }

class Dog3 extends Animal {
    static void method(Animal a) {
        if(a instanceof Dog3){
            Dog3 d=(Dog3)a;//downcasting
            System.out.println("ok downcasting performed");
        }
    }

    public static void main (String [] args) {
        Animal a=new Dog3();
        Dog3.method(a);
    }
}
```

[Test it Now](#)

Output:ok downcasting performed

Downcasting without the use of java instanceof

Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

```
class Animal { }

class Dog4 extends Animal {
    static void method(Animal a) {
        Dog4 d=(Dog4)a;//downcasting
        System.out.println("ok downcasting performed");
    }

    public static void main (String [] args) {
        Animal a=new Dog4();
        Dog4.method(a);
    }
}
```

```

    }
}

```

Test it Now

Output:ok downcasting performed

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

```

Animal a=new Animal();
Dog.method(a);
//Now ClassCastException but not in case of instanceof operator

```

Understanding Real use of instanceof in java

Let's see the real use of instanceof keyword by the example given below.

```

interface Printable{}
class A implements Printable{
public void a(){System.out.println("a method");}
}

class B implements Printable{
public void b(){System.out.println("b method");}
}

class Call{
void invoke(Printable p){//upcasting
if(p instanceof A){
A a=(A)p;//Downcasting
a.a();
}

if(p instanceof B){
B b=(B)p;//Downcasting
b.b();
}

}
}

}//end of Call class

class Test4{
public static void main(String args[]){
Printable p=new B();
Call c=new Call();
c.invoke(p);
}
}

```

Test it Now

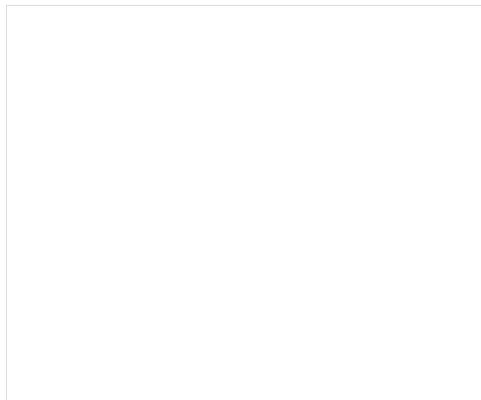
Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in [Java](#). It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.



Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the [object](#) does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

[Learn more](#)

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have **constructors** and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Rules for Java Abstract class



Example of abstract class

```
abstract class A{}
```

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

```
abstract void printStatus(); //no method body and abstract
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
    abstract void run();
}

class Honda4 extends Bike{
    void run(){System.out.println("running safely");}
    public static void main(String args[]){
        Bike obj = new Honda4();
        obj.run();
    }
}
```

Test it Now

```
running safely
```

Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
abstract class Shape{
    abstract void draw();
}

//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
    void draw(){System.out.println("drawing rectangle");}
}
```

```

}

class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}

//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1(); //In a real scenario, object is provided through method, e.g., getShape() method
s.draw();
}
}

```

Test it Now

drawing circle

Another example of Abstract class in java

File: TestBank.java

```

abstract class Bank{
abstract int getRateOfInterest();

}

class SBI extends Bank{
int getRateOfInterest(){return 7;}

}

class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}

```

Test it Now

Rate of Interest is: 7 %
Rate of Interest is: 8 %

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

File: TestAbstraction2.java

```
//Example of an abstract class that has abstract and non-abstract methods
abstract class Bike{
    Bike(){System.out.println("bike is created");}
    abstract void run();
    void changeGear(){System.out.println("gear changed");}
}

//Creating a Child class which inherits Abstract class
class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}

//Creating a Test class which calls abstract and non-abstract methods
class TestAbstraction2{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

Test it Now

```
bike is created
running safely..
gear changed
```

Rule: If there is an abstract method in a class, that class must be abstract.

```
class Bike12{
    abstract void run();
}
```

Test it Now

```
compile time error
```

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the [interface](#). In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
interface A{
    void a();
    void b();
    void c();
    void d();
}

abstract class B implements A{
    public void c(){System.out.println("I am c");}
}

class M extends B{
    public void a(){System.out.println("I am a");}
    public void b(){System.out.println("I am b");}
    public void d(){System.out.println("I am d");}
}

class Test5{
    public static void main(String args[]){
        A a=new M();
        a.a();
        a.b();
        a.c();
        a.d();
    }
}
```

Test it Now

```
Output:I am a
      I am b
      I am c
      I am d
```

← Prev

Next →

 [For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share

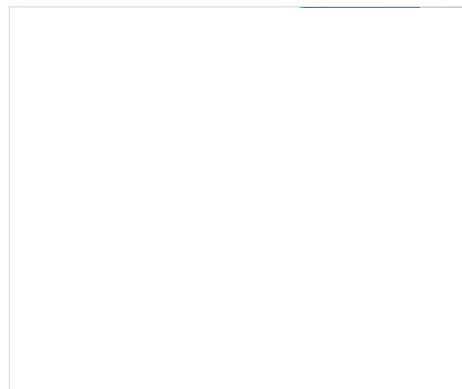


Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a *mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple **inheritance in Java**.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.



Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.



How to declare an interface?

An interface is declared by using the `interface` keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{  
  
    // declare constant fields  
    // declare methods that abstract  
    // by default.  
}
```

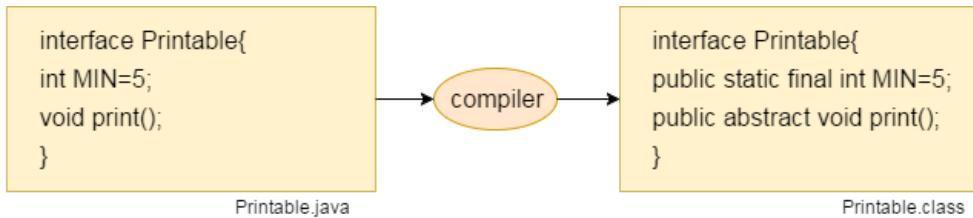
Java 8 Interface Improvement

Since [Java 8](#), interface can have default and static methods which is discussed later.

Internal addition by the compiler

The Java compiler adds `public` and `abstract` keywords before the interface method. Moreover, it adds `public`, `static` and `final` keywords before data members.

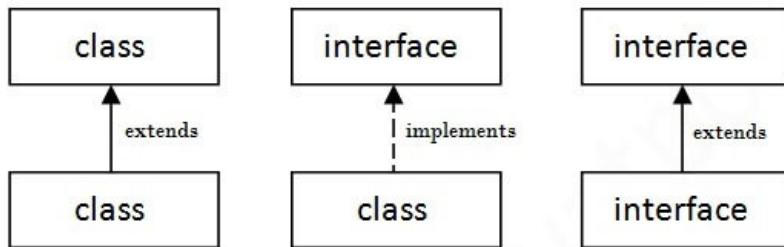
In other words, Interface fields are `public`, `static` and `final` by default, and the methods are `public` and `abstract`.



This Is An Ac For Coders

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

In this example, the `Printable` interface has only one method, and its implementation is provided in the `A6` class.

```
interface printable{
void print();
}

class A6 implements printable{
public void print(){System.out.println("Hello");}
}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
```

[Test it Now](#)

Output:

```
Hello
```

Java Interface Example: Drawable

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

File: TestInterface1.java

```
//Interface declaration: by first user
interface Drawable{
    void draw();
}

//Implementation: by second user
class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}

class Circle implements Drawable{
    public void draw(){System.out.println("drawing circle");}
}

//Using interface: by third user
class TestInterface1{
    public static void main(String args[]){
        Drawable d=new Circle(); //In real scenario, object is provided by method e.g. getDrawable()
        d.draw();
    }
}
```

[Test it Now](#)

Output:

```
drawing circle
```

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```
interface Bank{
    float rateOfInterest();
}

class SBI implements Bank{
    public float rateOfInterest(){return 9.15f;}
}

class PNB implements Bank{
    public float rateOfInterest(){return 9.7f;}
}
```

```

    }
}

class TestInterface2{
    public static void main(String[] args){
        Bank b=new SBI();
        System.out.println("ROI: "+b.rateOfInterest());
    }
}

```

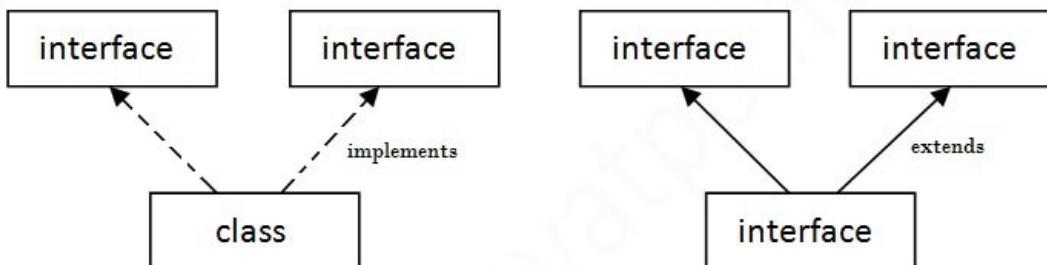
Test it Now

Output:

ROI: 9.15

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```

interface Printable{
    void print();
}

interface Showable{
    void show();
}

class A7 implements Printable,Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}
}

public static void main(String args[]){
    A7 obj = new A7();
    obj.print();
    obj.show();
}

```

Test it Now

Output:Hello
Welcome

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of **class** because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```
interface Printable{
void print();
}

interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
}
}
```

[Test it Now](#)

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}

interface Showable extends Printable{
void show();
}

class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
}
```

}

[Test it Now](#)

Output:

```
Hello
Welcome
```

Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

File: TestInterfaceDefault.java

```
interface Drawable{
    void draw();
    default void msg(){System.out.println("default method");}
}

class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceDefault{
    public static void main(String args[]){
        Drawable d=new Rectangle();
        d.draw();
        d.msg();
    }
}
```

[Test it Now](#)

Output:

```
drawing rectangle
default method
```

Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```
interface Drawable{
    void draw();
    static int cube(int x){return x*x*x;}
}

class Rectangle implements Drawable{
    public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
    public static void main(String args[]){
}
```

```
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

[Test it Now](#)

Output:

```
drawing rectangle
27
```

Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, `Serializable`, `Cloneable`, `Remote`, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?
public interface Serializable{}
```

Nested Interface in Java

Note: An interface can have another interface which is known as a nested interface. We will learn it in detail in the [nested classes](#) chapter. For example:

```
interface printable{
    void print();
    interface MessagePrintable{
        void msg();
    }
}
```

[More about Nested Interface](#)

[« Prev](#)

[Next »](#)

 For Videos Join Our YouTube Channel: [Join Now](#)

Help Others, Please Share



Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance .	Interface supports multiple inheritance .
3) Abstract class can have final, non-final, static and non-static variables .	Interface has only static and final variables .
4) Abstract class can provide the implementation of interface .	Interface can't provide the implementation of abstract class .
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
9) Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
//Creating interface that has 4 methods
interface A{
    void a();//bydefault, public and abstract
    void b();
    void c();
    void d();
}

//Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{
    public void c(){System.out.println("I am C");}
}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods
class M extends B{
    public void a(){System.out.println("I am a");}
    public void b(){System.out.println("I am b");}
    public void d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of A interface
class Test5{
    public static void main(String args[]){
        A a=new M();
        a.a();
        a.b();
        a.c();
        a.d();
    }
}
```

[Test it Now](#)

Output:

```
I am a
I am b
I am c
I am d
```

← Prev

Next →

For Videos Join Our YouTube Channel: [Join Now](#)

Help Others, Please Share



The advertisement features a black background with the Coding Ninjas logo at the top left. In the center, it says "We're going to give you an offer you can't refuse". Below that is a yellow button with the text "Start your free trial". To the right is a portrait of Marlon Brando as Don Corleone, holding a cigar. There are some code snippets floating around him.

Learn Latest Tutorials

SoapUI tutorial SoapUI	RPA tutorial RPA	manual testing tutorial Manual T.	cucumber tutorial Cucumber
Appium tutorial Appium	postgresql tutorial PostgreSQL	Apache Solr Tutorial Solr	MongoDB tutorial MongoDB
Gimp Tutorial Gimp	Verilog Tutorial Verilog	Teradata Tutorial Teradata	PhoneGap Tutorial PhoneGap

Preparation

Aptitude	Logical Reasoning	Verbal Ability	Interview Questions
Aptitude	Reasoning	Verbal A.	Interview



Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

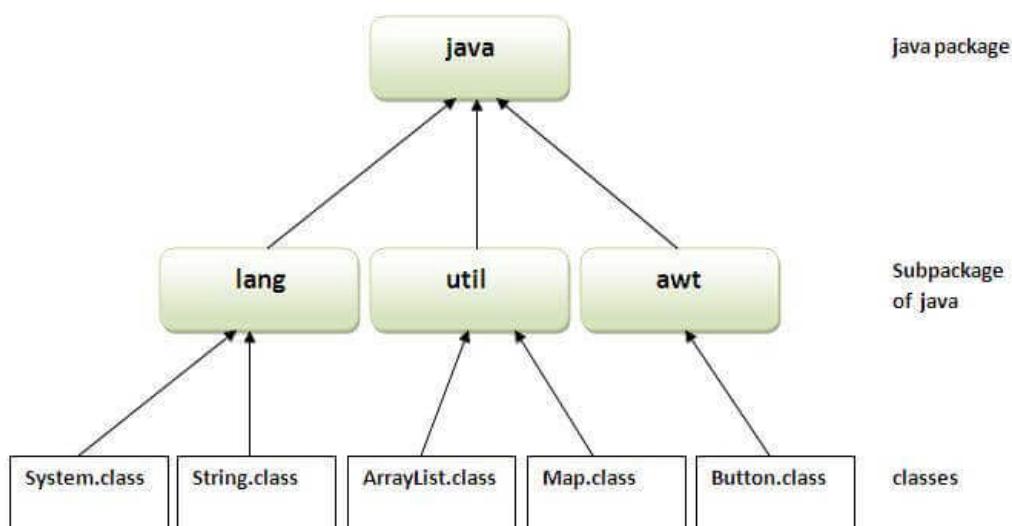
Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For example

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.



Let Gatik Help You Take Your First Step
Towards New Beginnings And Endless Possibilities

Gatik Junior College

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
```

```
import pack.A;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

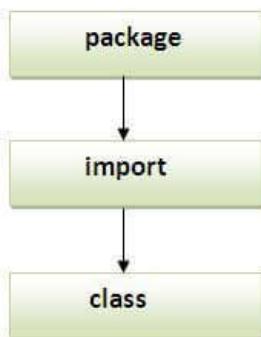
```
//save by B.java
package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A();//using fully qualified name
        obj.msg();
    }
}
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has defined a package named `java` that contains many classes like `System`, `String`, `Reader`, `Writer`, `Socket` etc. These classes represent a particular group e.g. `Reader` and `Writer` classes are for Input/Output operation, `Socket` and `ServerSocket` classes are for networking etc and so on. So, Sun has subcategorized the `java` package into subpackages such as `lang`, `net`, `io` etc. and put the Input/Output related classes in `io` package, `Server` and `ServerSocket` classes in `net` packages and so on.

The standard of defining package is `domain.company.package` e.g. `com.javatpoint.bean` or `org.sssit.dao`.

Example of Subpackage

```

package com.javatpoint.core;
class Simple{
    public static void main(String args[]){
        System.out.println("Hello subpackage");
    }
}
  
```

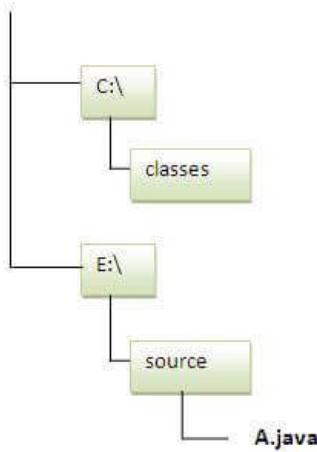
To Compile: `javac -d . Simple.java`

To Run: `java com.javatpoint.core.Simple`

Output:Hello subpackage

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of `A.java` source file in `classes` folder of `c:` drive. For example:



```
//save as Simple.java
package mypack;
public class Simple{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

e:\sources> set classpath=c:\classes;..;

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

Output:Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- o Temporary
 - o By setting the classpath in the command prompt
 - o By -classpath switch
- o Permanent

- By setting the classpath in the environment variables
- By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

//save as C.java otherwise Compile Time Error

```
class A{}  
class B{}  
public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

//save as A.java

```
package javatpoint;  
public class A{}
```

//save as B.java

```
package javatpoint;  
public class B{}
```

What is static import feature of Java5?

Click [Static Import](#) feature of Java5.

What about package class?

Click for [Package class](#)

← Prev

Next →

 [For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package

Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y


[Learn more](#)

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
    private int data=40;
    private void msg(){System.out.println("Hello java");}
}

public class Simple{
    public static void main(String args[]){
        A obj=new A();
        System.out.println(obj.data); //Compile Time Error
        obj.msg(); //Compile Time Error
    }
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
    private A(){} //private constructor
    void msg(){System.out.println("Hello java");}
}
```

```

}

public class Simple{
    public static void main(String args[]){
        A obj=new A(); //Compile Time Error
    }
}

```

Note: A class cannot be private or protected except nested class.

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```

//save by A.java
package pack;
class A{
    void msg(){System.out.println("Hello");}
}

```

```

//save by B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A(); //Compile Time Error
        obj.msg(); //Compile Time Error
    }
}

```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifier.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B extends A{
public static void main(String args[]){
B obj = new B();
obj.msg();
}
}
```

Output:Hello

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
```

```
package mypack;
import pack.*;
```

```
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
class A{
    protected void msg(){System.out.println("Hello java");}
}

public class Simple extends A{
    void msg(){System.out.println("Hello java");}//C.T.Error
    public static void main(String args[]){
        Simple obj=new Simple();
        obj.msg();
    }
}
```

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

← Prev

Next →

 For Videos Join Our YouTube Channel: [Join Now](#)

Help Others, Please Share



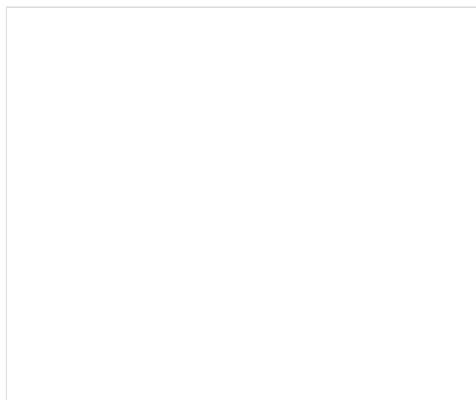
Encapsulation in Java

Encapsulation in Java is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.

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



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



Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.



Low Cost Programming Courses

Simple Example of Encapsulation in Java

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

File: Student.java

```
//A Java class which is a fully encapsulated class.  
//It has a private data member and getter and setter methods.  
  
package com.javatpoint;  
  
public class Student{  
    //private data member  
    private String name;  
    //getter method for name  
    public String getName(){  
        return name;  
    }  
    //setter method for name  
    public void setName(String name){  
        this.name=name  
    }  
}
```

File: Test.java

```
//A Java class to test the encapsulated class.  
  
package com.javatpoint;  
  
class Test{  
    public static void main(String[] args){  
        //creating instance of the encapsulated class  
        Student s=new Student();  
        //setting value in the name member  
        s.setName("vijay");  
        //getting value of the name member  
        System.out.println(s.getName());  
    }  
}
```

```
Compile By: javac -d . Test.java
Run By: java com.javatpoint.Test
```

Output:

```
vijay
```

Read-Only class

```
//A Java class which has only getter methods.

public class Student{
//private data member
private String college="AKG";
//getter method for college
public String getCollege(){
return college;
}
}
```

Now, you can't change the value of the college data member which is "AKG".

```
s.setCollege("KITE");//will render compile time error
```

Write-Only class

```
//A Java class which has only setter methods.

public class Student{
//private data member
private String college;
//getter method for college
public void setCollege(String college){
this.college=college;
}
}
```

Now, you can't get the value of the college, you can only change the value of college data member.

```
System.out.println(s.getCollege());//Compile Time Error, because there is no such method
System.out.println(s.college);//Compile Time Error, because the college data member is private.
//So, it can't be accessed from outside the class
```

Another Example of Encapsulation in Java

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

File: Account.java

```
//A Account class which is a fully encapsulated class.
//It has a private data member and getter and setter methods.

class Account {
    //private data members

    private long acc_no;
    private String name,email;
    private float amount;

    //public getter and setter methods

    public long getAcc_no() {
        return acc_no;
    }

    public void setAcc_no(long acc_no) {
        this.acc_no = acc_no;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public float getAmount() {
        return amount;
    }

    public void setAmount(float amount) {
        this.amount = amount;
    }
}
```

File: TestAccount.java

```
//A Java class to test the encapsulated class Account.

public class TestEncapsulation {
    public static void main(String[] args) {
        //creating instance of Account class
        Account acc=new Account();
        //setting values through setter methods
        acc.setAcc_no(7560504000L);
        acc.setName("Sonoo Jaiswal");
        acc.setEmail("sonoojaiswal@javatpoint.com");
    }
}
```

```
acc.setAmount(500000f);
//getting values through getter methods
System.out.println(acc.getAcc_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());
}
}
```

[Test it Now](#)

Output:

```
7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0
```

← Prev

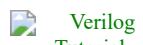
Next →

 [For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Learn Latest Tutorials

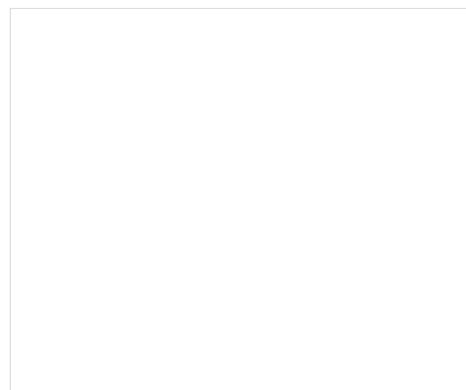
 SoapUI tutorial SoapUI	 RPA tutorial RPA	 manual testing tutorial Manual T.	 cucumber tutorial Cucumber
 Appium tutorial Appium	 postgresql tutorial PostgreSQL	 Apache Solr Tutorial Solr	 MongoDB tutorial MongoDB
 Gimp Tutorial Gimp	 Verilog Tutorial Verilog	 Teradata Tutorial Teradata	 PhoneGap Tutorial PhoneGap

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.

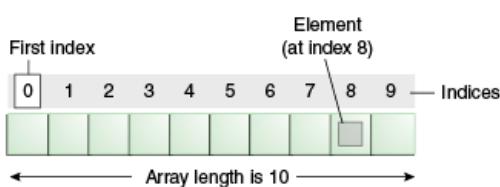
Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.



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.

In Java, array is an object of a dynamically generated class. Java array inherits the `Object` class, and implements the `Serializable` as well as `Cloneable` interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



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.



Get Ready For Your Dream Job

GeeksforGeeks

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

```
dataType[] arr; (or)  
dataType []arr; (or)  
dataType arr[];
```

Instantiation of an Array in Java

```
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.

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

Test it Now

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

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

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

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

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

[Test it Now](#)

Output:

```
33
3
4
5
```

For-each Loop for Java Array

We can also print the Java array using **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.

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

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.

```
//Java Program to demonstrate the way of passing an array
//to method.

class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
if(min>arr[i])
min=arr[i];

System.out.println(min);
}

public static void main(String args[]){
int a[]={33,3,4,5}; //declaring and initializing an array
min(a); //passing array to method
}}
```

[Test it Now](#)

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.

```
//Java Program to demonstrate the way of passing an anonymous array
//to method.

public class TestAnonymousArray{
    //creating a method which receives an array as a parameter
    static void printArray(int arr[]){
        for(int i=0;i<arr.length;i++)
            System.out.println(arr[i]);
    }

    public static void main(String args[]){
        printArray(new int[]{10,22,44,66}); //passing anonymous array to method
    }
}
```

[Test it Now](#)

Output:

```
10
22
44
66
```

Returning Array from the Method

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

```
//Java Program to return an array from the method
class TestReturnArray{
    //creating method which returns an array
    static int[] get(){
        return new int[]{10,30,50,90,60};
    }

    public static void main(String args[]){
        //calling method which returns an array
        int arr[]=get();
        //printing the values of an array
        for(int i=0;i<arr.length;i++)
            System.out.println(arr[i]);
    }
}
```

[Test it Now](#)

Output:

```
10
30
50
90
60
```

ArrayIndexOutOfBoundsException

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

```
//Java Program to demonstrate the case of
//ArrayIndexOutOfBoundsException in a Java Array.

public class TestArrayException{
    public static void main(String args[]){
        int arr[]={50,60,70,80};
        for(int i=0;i<=arr.length;i++){
            System.out.println(arr[i]);
        }
    }
}
```

[Test it Now](#)

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

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

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

Example to initialize Multidimensional Array in Java

```
arr[0][0]=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;
```

Example of Multidimensional Java Array

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

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

[Test it Now](#)

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.

```
//Java Program to illustrate the jagged array
class TestJaggedArray{
public static void main(String[] args){
//declaring a 2D array with odd columns
int arr[][] = new int[3][];
arr[0] = new int[3];
arr[1] = new int[4];
arr[2] = new int[2];
//initializing a jagged array
int count = 0;
for (int i=0; i<arr.length; i++)
    for(int j=0; j<arr[i].length; j++)
        arr[i][j] = count++;

//printing the data of a jagged array
for (int i=0; i<arr.length; i++){
```

```

for (int j=0; j<arr[i].length; j++){
    System.out.print(arr[i][j]+" ");
}
System.out.println();//new line
}
}
}

```

[Test it Now](#)

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.

```

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

}}

```

[Test it Now](#)

Output:

```
I
```

Copying a Java Array

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

Syntax of `arraycopy` method

```

public static void arraycopy(
Object src, int srcPos, Object dest, int destPos, int length
)

```

Example of Copying an Array in Java

```
//Java Program to copy a source array into a destination array in Java
class TestArrayCopyDemo {
    public static void main(String[] args) {
        //declaring a source array
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
            'l', 'n', 'a', 't', 'e', 'd' };
        //declaring a destination array
        char[] copyTo = new char[7];
        //copying array using System.arraycopy() method
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        //printing the destination array
        System.out.println(String.valueOf(copyTo));
    }
}
```

[Test it Now](#)

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.

```
//Java Program to clone the array
class Testarray1{
    public static void main(String args[]){
        int arr[]={33,3,4,5};
        System.out.println("Printing original array:");
        for(int i:arr)
            System.out.println(i);

        System.out.println("Printing clone of the array:");
        int carr[]=arr.clone();
        for(int i:carr)
            System.out.println(i);

        System.out.println("Are both equal?");
        System.out.println(arr==carr);

    }}
```

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.

```
//Java Program to demonstrate the addition of two matrices in Java
class Testarray5{
    public static void main(String args[]){
        //creating two matrices
        int a[][]={{1,3,4},{3,4,5}};
        int b[][]={{1,3,4},{3,4,5}};

        //creating another matrix to store the sum of two matrices
        int c[][]=new int[2][3];

        //adding and printing addition of 2 matrices
        for(int i=0;i<2;i++){
            for(int j=0;j<3;j++){
                c[i][j]=a[i][j]+b[i][j];
                System.out.print(c[i][j]+" ");
            }
            System.out.println(); //new line
        }
    }
}
```

[Test it Now](#)

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.

$$\text{Matrix 1} \left\{ \begin{array}{ccc} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{array} \right\} \quad \text{Matrix 2} \left\{ \begin{array}{ccc} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{array} \right\}$$

$$\begin{aligned} \text{Matrix 1} * \text{Matrix 2} &= \left\{ \begin{array}{ccc} 1*1+1*2+1*3 & 1*1+1*2+1*3 & 1*1+1*2+1*3 \\ 2*1+2*2+2*3 & 2*1+2*2+2*3 & 2*1+2*2+2*3 \\ 3*1+3*2+3*3 & 3*1+3*2+3*3 & 3*1+3*2+3*3 \end{array} \right\} \\ \text{Matrix 1} * \text{Matrix 2} &= \left\{ \begin{array}{ccc} 6 & 6 & 6 \\ 12 & 12 & 12 \\ 18 & 18 & 18 \end{array} \right\} \end{aligned}$$

JavaTpoint

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

```
//Java Program to multiply two matrices
public class MatrixMultiplicationExample{
    public static void main(String args[]){
        //creating two matrices
        int a[][]={{1,1,1},{2,2,2},{3,3,3}};
        int b[][]={{1,1,1},{2,2,2},{3,3,3}};

        //creating another matrix to store the multiplication of two matrices
        int c[][]=new int[3][3]; //3 rows and 3 columns

        //multiplying and printing multiplication of 2 matrices
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                c[i][j]=0;
                for(int k=0;k<3;k++){
                    {
                        c[i][j]+=a[i][k]*b[k][j];
                    }
                }
                System.out.print(c[i][j]+" ");
            }
            System.out.println(); //new line
        }
    }
}
```

Test it Now

Output:

```
6 6 6
12 12 12
18 18 18
```

Related Topics

- 1) Java Program to copy all elements of one array into another array
- 2) Java Program to find the frequency of each element in the array
- 3) Java Program to left rotate the elements of an array
- 4) Java Program to print the duplicate elements of an array
- 5) Java Program to print the elements of an array
- 6) Java Program to print the elements of an array in reverse order
- 7) Java Program to print the elements of an array present on even position
- 8) Java Program to print the elements of an array present on odd position
- 9) Java Program to print the largest element in an array
- 10) Java Program to print the smallest element in an array
- 11) Java Program to print the number of elements present in an array
- 12) Java Program to print the sum of all the items of the array
- 13) Java Program to right rotate the elements of an array
- 14) Java Program to sort the elements of an array in ascending order
- 15) Java Program to sort the elements of an array in descending order
- 16) Find 3rd Largest Number in an Array
- 17) Find 2nd Largest Number in an Array
- 18) Find Largest Number in an Array
- 19) Find 2nd Smallest Number in an Array
- 20) Find Smallest Number in an Array
- 21) Remove Duplicate Element in an Array
- 22) Add Two Matrices
- 23) Multiply Two Matrices
- 24) Print Odd and Even Number from an Array
- 25) Transpose matrix
- 26) Java Program to subtract the two matrices
- 27) Java Program to determine whether a given matrix is an identity matrix
- 28) Java Program to determine whether a given matrix is a sparse matrix
- 29) Java Program to determine whether two matrices are equal
- 30) Java Program to display the lower triangular matrix
- 31) Java Program to display the upper triangular matrix
- 32) Java Program to find the frequency of odd & even numbers in the given matrix
- 33) Java Program to find the product of two matrices
- 34) Java Program to find the sum of each row and each column of a matrix
- 35) Java Program to find the transpose of a given matrix

Object class in Java

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

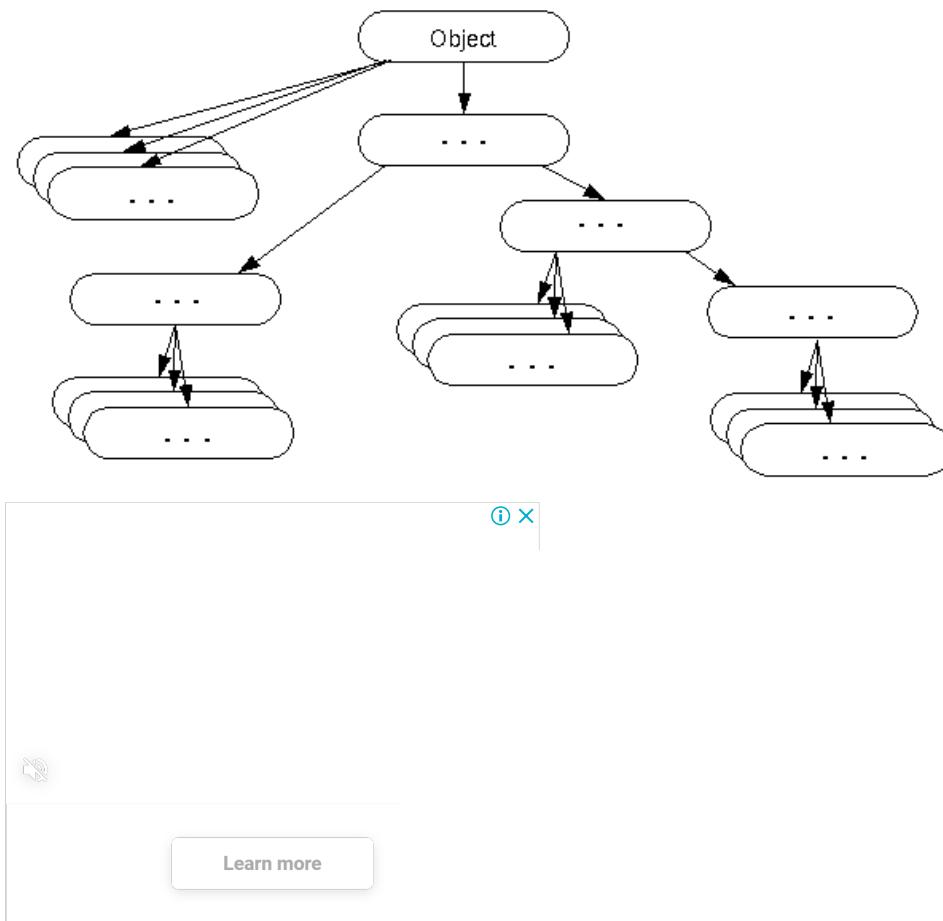
The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, known as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee, Student etc, we can use Object class reference to refer that object. For example:



```
Object obj=getObject(); //we don't know what object will be returned from this method
```

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.



Methods of Object class

The Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout) throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).

public final void wait(long timeout,int nanos) throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait() throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize() throws Throwable	is invoked by the garbage collector before object is being garbage collected.

We will have the detailed learning of these methods in next chapters.

← Prev

Next →

 [For Videos Join Our YouTube Channel: Join Now](#)

Help Others, Please Share



(i) X

Get Started with C Language

You Will Learn A Lot From Our Dedicated Experts.

GeeksforGeeks

S

Learn Latest Tutorials

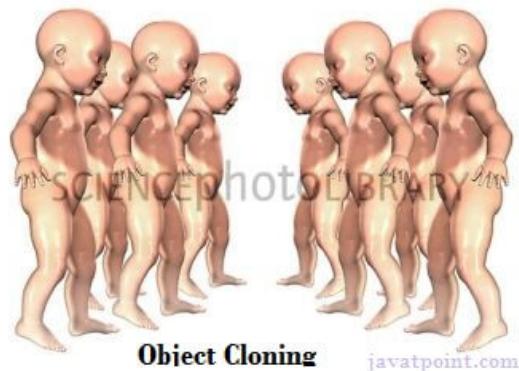
 SoapUI tutorial SoapUI	 RPA tutorial RPA	 manual testing tutorial Manual T.	 cucumber tutorial Cucumber
 Appium tutorial Appium	 PostgreSQL tutorial PostgreSQL	 Apache Solr Tutorial Solr	 MongoDB tutorial MongoDB
 Gimp Tutorial Gimp	 Verilog Tutorial Verilog	 Teradata Tutorial Teradata	 PhoneGap Tutorial PhoneGap

Object Cloning in Java

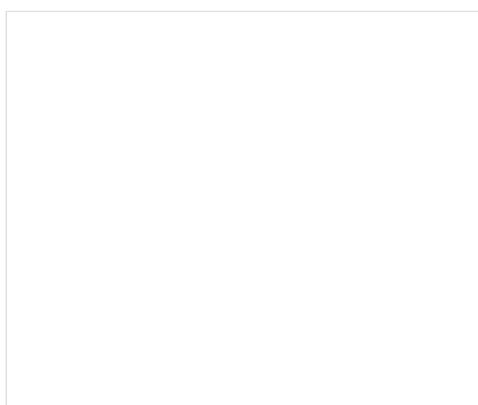
The **object cloning** is a way to create exact copy of an object.

The `clone()` method of `Object` class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement `Cloneable` interface, `clone()` method generates **CloneNotSupportedException**.



The **clone() method** is defined in the `Object` class. Syntax of the `clone()` method is as follows:



```
protected Object clone() throws CloneNotSupportedException
```

Why use `clone()` method ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the `new` keyword, it will take a lot of processing time to be performed that is why we use object cloning.

Advantage of Object cloning

Although `Object.clone()` has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using `clone()` method:

- You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long `clone()` method.
- It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a

parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.

- Clone() is the fastest way to copy array.

Disadvantage of Object cloning

Following is a list of some disadvantages of clone() method:

- To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
- We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
- Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
- Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
- If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
- Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

The box contains a small icon of a hand pointing right, a close button (X) in the top right corner, and a 'Learn more' button at the bottom center.

Example of clone() method (Object cloning)

Let's see the simple example of object cloning

```
class Student18 implements Cloneable{  
    int rollno;  
    String name;  
  
    Student18(int rollno,String name){  
        this.rollno=rollno;  
        this.name=name;  
    }  
  
    public Object clone()throws CloneNotSupportedException{
```

```

return super.clone();
}

public static void main(String args[]){
try{
Student18 s1=new Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}

}
}

```

[Test it Now](#)

Output:101 amit
101 amit

[download the example of object cloning](#)

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

[← Prev](#)

[Next →](#)

[For Videos Join Our YouTube Channel: Join Now](#)

Help Others, Please Share



We know you'll love our courses as much as we do!



[START YOUR FREE TRIAL](#)

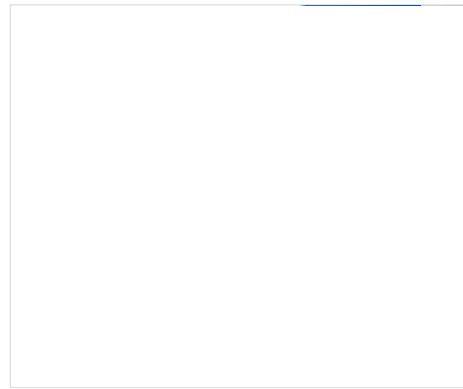


Java Math class

Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

Unlike some of the StrictMath class numeric methods, all implementations of the equivalent function of Math class can't define to return the bit-for-bit same results. This relaxation permits implementation with better-performance where strict reproducibility is not required.

If the size is int or long and the results overflow the range of value, the methods addExact(), subtractExact(), multiplyExact(), and toIntExact() throw an ArithmeticException.



For other arithmetic operations like increment, decrement, divide, absolute value, and negation overflow occur only with a specific minimum or maximum value. It should be checked against the maximum and minimum value as appropriate.

Example 1

```
public class JavaMathExample1
{
    public static void main(String[] args)
    {
        double x = 28;
        double y = 4;

        // return the maximum of two numbers
        System.out.println("Maximum number of x and y is: " + Math.max(x, y));

        // return the square root of y
        System.out.println("Square root of y is: " + Math.sqrt(y));

        //returns 28 power of 4 i.e. 28*28*28*28
        System.out.println("Power of x and y is: " + Math.pow(x, y));
    }
}
```

```
// return the logarithm of given value
System.out.println("Logarithm of x is: " + Math.log(x));
System.out.println("Logarithm of y is: " + Math.log(y));

// return the logarithm of given value when base is 10
System.out.println("log10 of x is: " + Math.log10(x));
System.out.println("log10 of y is: " + Math.log10(y));

// return the log of x + 1
System.out.println("log1p of x is: " + Math.log1p(x));

// return a power of 2
System.out.println("exp of a is: " + Math.exp(x));

// return (a power of 2)-1
System.out.println("expm1 of a is: " + Math.expm1(x));
}
```

[Test it Now](#)

Output:

```
Maximum number of x and y is: 28.0
Square root of y is: 2.0
Power of x and y is: 614656.0
Logarithm of x is: 3.332204510175204
Logarithm of y is: 1.3862943611198906
log10 of x is: 1.4471580313422192
log10 of y is: 0.6020599913279624
log1p of x is: 3.367295829986474
exp of a is: 1.446257064291475E12
expm1 of a is: 1.446257064290475E12
```

Example 2

```
public class JavaMathExample2
{
    public static void main(String[] args)
    {
        double a = 30;

        // converting values to radian
        double b = Math.toRadians(a);
```

```
// return the trigonometric sine of a
System.out.println("Sine value of a is: " +Math.sin(a));

// return the trigonometric cosine value of a
System.out.println("Cosine value of a is: " +Math.cos(a));

// return the trigonometric tangent value of a
System.out.println("Tangent value of a is: " +Math.tan(a));

// return the trigonometric arc sine of a
System.out.println("Sine value of a is: " +Math.asin(a));

// return the trigonometric arc cosine value of a
System.out.println("Cosine value of a is: " +Math.acos(a));

// return the trigonometric arc tangent value of a
System.out.println("Tangent value of a is: " +Math.atan(a));

// return the hyperbolic sine of a
System.out.println("Sine value of a is: " +Math.sinh(a));

// return the hyperbolic cosine value of a
System.out.println("Cosine value of a is: " +Math.cosh(a));

// return the hyperbolic tangent value of a
System.out.println("Tangent value of a is: " +Math.tanh(a));
}
```

[Test it Now](#)

Output:

```
Sine value of a is: -0.9880316240928618
Cosine value of a is: 0.15425144988758405
Tangent value of a is: -6.405331196646276
Sine value of a is: NaN
Cosine value of a is: NaN
Tangent value of a is: 1.5374753309166493
Sine value of a is: 5.343237290762231E12
Cosine value of a is: 5.343237290762231E12
Tangent value of a is: 1.0
```

Java Math Methods

The **java.lang.Math** class contains various methods for performing basic numeric operations such as the logarithm, cube root, and trigonometric functions etc. The various java math methods are as follows:

Basic Math methods

Method	Description
<code>Math.abs()</code>	It will return the Absolute value of the given value.
<code>Math.max()</code>	It returns the Largest of two values.
<code>Math.min()</code>	It is used to return the Smallest of two values.
<code>Math.round()</code>	It is used to round off the decimal numbers to the nearest value.
<code>Math.sqrt()</code>	It is used to return the square root of a number.
<code>Math.cbrt()</code>	It is used to return the cube root of a number.
<code>Math.pow()</code>	It returns the value of first argument raised to the power to second argument.
<code>Math.signum()</code>	It is used to find the sign of a given value.
<code>Math.ceil()</code>	It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer.
<code>Math.copySign()</code>	It is used to find the Absolute value of first argument along with sign specified in second argument.
<code>Math.nextAfter()</code>	It is used to return the floating-point number adjacent to the first argument in the direction of the second argument.
<code>Math.nextUp()</code>	It returns the floating-point value adjacent to d in the direction of positive infinity.
<code>Math.nextDown()</code>	It returns the floating-point value adjacent to d in the direction of negative infinity.
<code>Math.floor()</code>	It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value.
<code>Math.floorDiv()</code>	It is used to find the largest integer value that is less than or equal to the algebraic quotient.
<code>Math.random()</code>	It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
<code>Math.rint()</code>	It returns the double value that is closest to the given argument and equal to mathematical integer.
<code>Math.hypot()</code>	It returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.
<code>Math.ulp()</code>	It returns the size of an ulp of the argument.
<code>Math.getExponent()</code>	It is used to return the unbiased exponent used in the representation of a value.
<code>Math.IEEEremainder()</code>	It is used to calculate the remainder operation on two arguments as prescribed by the IEEE 754 standard and returns value.
<code>Math.addExact()</code>	It is used to return the sum of its arguments, throwing an exception if the result overflows an int or long.
<code>Math.subtractExact()</code>	It returns the difference of the arguments, throwing an exception if the result overflows an int.
<code>Math.multiplyExact()</code>	It is used to return the product of the arguments, throwing an exception if the result overflows an int or long.
<code>Math.incrementExact()</code>	It returns the argument incremented by one, throwing an exception if the result overflows an int.

<code>Math.decrementExact()</code>	It is used to return the argument decremented by one, throwing an exception if the result overflows an int or long.
<code>Math.negateExact()</code>	It is used to return the negation of the argument, throwing an exception if the result overflows an int or long.
<code>Math.toIntExact()</code>	It returns the value of the long argument, throwing an exception if the value overflows an int.

Logarithmic Math Methods

Method	Description
<code>Math.log()</code>	It returns the natural logarithm of a double value.
<code>Math.log10()</code>	It is used to return the base 10 logarithm of a double value.
<code>Math.log1p()</code>	It returns the natural logarithm of the sum of the argument and 1.
<code>Math.exp()</code>	It returns E raised to the power of a double value, where E is Euler's number and it is approximately equal to 2.71828.
<code>Math.expm1()</code>	It is used to calculate the power of E and subtract one from it.

Trigonometric Math Methods

Method	Description
<code>Math.sin()</code>	It is used to return the trigonometric Sine value of a Given double value.
<code>Math.cos()</code>	It is used to return the trigonometric Cosine value of a Given double value.
<code>Math.tan()</code>	It is used to return the trigonometric Tangent value of a Given double value.
<code>Math.asin()</code>	It is used to return the trigonometric Arc Sine value of a Given double value
<code>Math.acos()</code>	It is used to return the trigonometric Arc Cosine value of a Given double value.
<code>Math.atan()</code>	It is used to return the trigonometric Arc Tangent value of a Given double value.

Hyperbolic Math Methods

Method	Description
<code>Math.sinh()</code>	It is used to return the trigonometric Hyperbolic Cosine value of a Given double value.
<code>Math.cosh()</code>	It is used to return the trigonometric Hyperbolic Sine value of a Given double value.
<code>Math.tanh()</code>	It is used to return the trigonometric Hyperbolic Tangent value of a Given double value.

Angular Math Methods

Method	Description
<code>Math.toDegrees</code>	It is used to convert the specified Radians angle to equivalent angle measured in Degrees.

Math.toRadians

It is used to convert the specified Degrees angle to equivalent angle measured in Radians.

← Prev

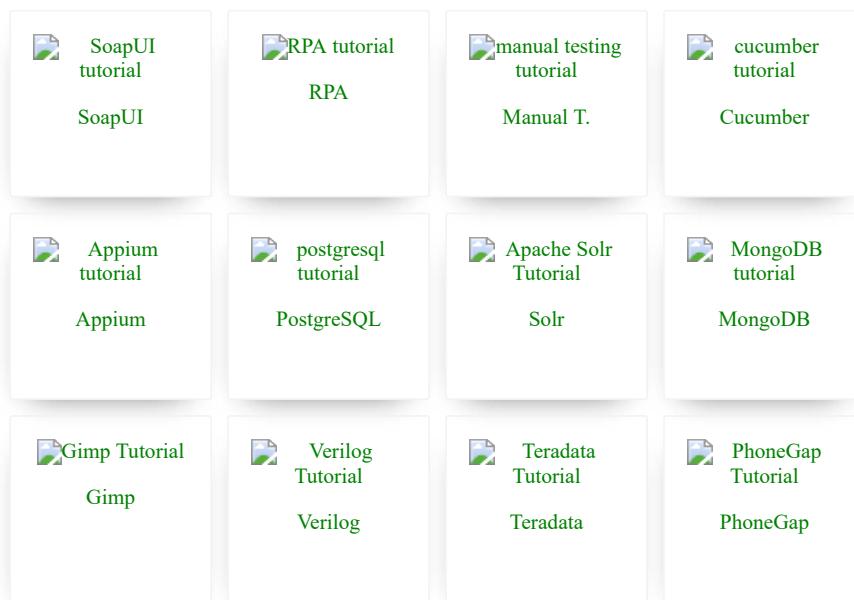
Next →

 Youtube For Videos Join Our Youtube Channel: [Join Now](#)

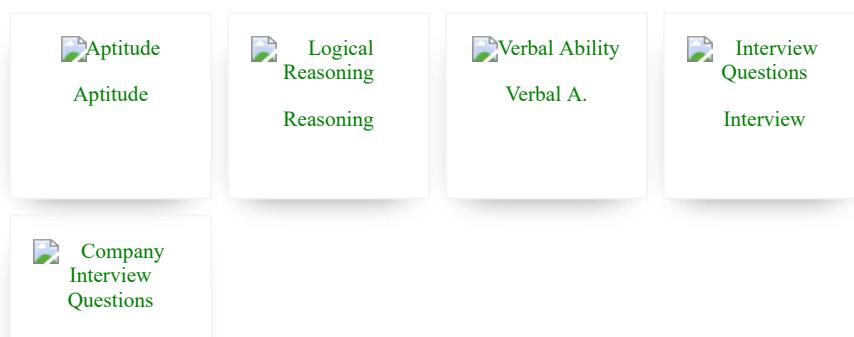
Help Others, Please Share



Learn Latest Tutorials



Preparation



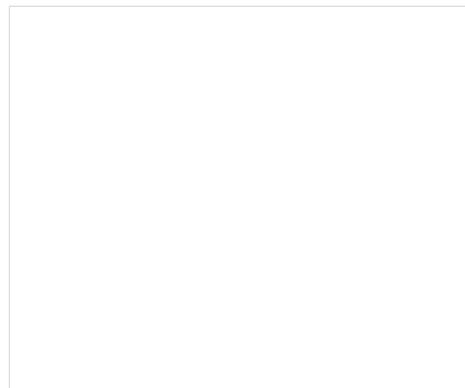
Wrapper classes in Java

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.

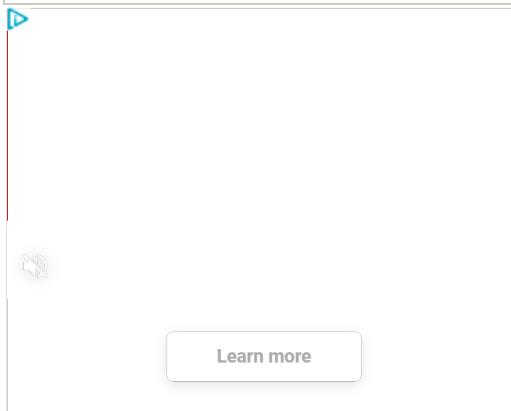


- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform 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
char	Character
byte	Byte
short	Short
int	Integer

long	Long
float	Float
double	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);
    }
}
```

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

```
//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);
    }
}
```

Output:

```
3 3 3
```

Java 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);
}}
```

Output:

```
--Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

Custom Wrapper class in Java

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

```
//Creating the custom wrapper class
class Javatpoint{
    private int i;
    Javatpoint(){}
    Javatpoint(int i){
        this.i=i;
    }
    public int getValue(){
        return i;
    }
    public void setValue(int i){
        this.i=i;
    }
    @Override
    public String toString() {
        return Integer.toString(i);
    }
}

//Testing the custom wrapper class
public class TestJavatpoint{
    public static void main(String[] args){
        Javatpoint j=new Javatpoint(10);
        System.out.println(j);
    }
}
```

Output:

```
10
```

← Prev

Next →

 For Videos Join Our YouTube Channel: [Join Now](#)

Help Others, Please Share



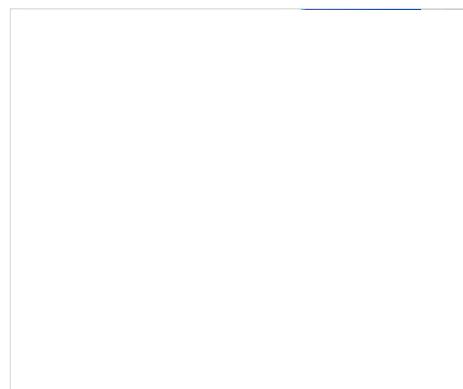


Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

It makes the code compact but complex to understand.

Syntax:



```
returntype methodname(){  
    //code to be executed  
    methodname(); //calling same method  
}
```

Java Recursion Example 1: Infinite times

```
public class RecursionExample1 {  
    static void p(){  
        System.out.println("hello");  
        p();  
    }  
  
    public static void main(String[] args) {  
        p();  
    }  
}
```

Output:

```
hello  
hello  
...  
java.lang.StackOverflowError
```

Data Structures & Algorithms

Programming Pathshala

Java Recursion Example 2: Finite times

```
public class RecursionExample2 {  
    static int count=0;  
    static void p(){  
        count++;  
        if(count<=5){  
            System.out.println("hello "+count);  
            p();  
        }  
    }  
    public static void main(String[] args) {  
        p();  
    }  
}
```

Output:

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

Java Recursion Example 3: Factorial Number

```
public class RecursionExample3 {  
    static int factorial(int n){  
        if (n == 1)  
            return 1;  
        else  
            return(n * factorial(n-1));  
    }  
}
```

```
public static void main(String[] args) {
    System.out.println("Factorial of 5 is: "+factorial(5));
}
}
```

Output:

```
Factorial of 5 is: 120
```

Working of above program:

```
factorial(5)
  factorial(4)
    factorial(3)
      factorial(2)
        factorial(1)
          return 1
        return 2*1 = 2
      return 3*2 = 6
    return 4*6 = 24
  return 5*24 = 120
```

Java Recursion Example 4: Fibonacci Series

```
public class RecursionExample4 {
    static int n1=0,n2=1,n3=0;
    static void printFibo(int count){
        if(count>0){
            n3 = n1 + n2;
            n1 = n2;
            n2 = n3;
            System.out.print(" "+n3);
            printFibo(count-1);
        }
    }

    public static void main(String[] args) {
        int count=15;
        System.out.print(n1+" "+n2);//printing 0 and 1
        printFibo(count-2);//n-2 because 2 numbers are already printed
    }
}
```

Output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

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.

Example of 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);  
  
    }  
}
```

[download this example](#)

```
Output:before change 50  
        after change 50
```

[Learn more](#)

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);  
    }  
}
```

[download this example](#)

```
Output:before change 50  
                  after change 150
```

[← Prev](#)[Next →](#) For Videos Join Our YouTube Channel: [Join Now](#)

Help Others, Please Share

Java Strictfp Keyword

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

Legal code for strictfp keyword

The strictfp keyword can be applied on methods, classes and interfaces.

```
strictfp class A{}//strictfp applied on class  
  
strictfp interface M{}//strictfp applied on interface  
  
class A{  
    strictfp void m(){}//strictfp applied on method  
}
```

**C++ Foundation @
Just 699**

GeeksforGeeks

Illegal code for strictfp keyword

The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.



```
class B{  
    strictfp abstract void m();//Illegal combination of modifiers  
}
```

```
class B{  
    strictfp int data=10;//modifier strictfp not allowed here  
}
```

```
class B{  
    strictfp B(){}//modifier strictfp not allowed here  
}
```

← Prev

Next →

For Videos Join Our YouTube Channel: [Join Now](#)

Help Others, Please Share



Basics Of DS and Alg

Your DS And Algo
Fundamentals Basics
Should Be Strong
GeeksforGeeks

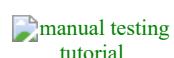
Learn Latest Tutorials



SoapUI
tutorial



RPA tutorial



manual testing
tutorial



cucumber
tutorial

Creating API Document | javadoc tool

We can create document api in java by the help of **javadoc** tool. In the java file, we must use the documentation comment `/**... */` to post information for the class, method, constructor, fields etc.

Let's see the simple class that contains documentation comment.

```
package com.abc;
/** This class is a user-defined class that contains one methods cube.*/
public class M{

    /** The cube method prints cube of the given number */
    public static void cube(int n){System.out.println(n*n*n);}
}
```

To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write:

```
javadoc M.java
```

to generate the document api. Now, there will be created a lot of html files. Open the index.html file to get the information about the classes.

[Learn more](#)[← Prev](#)[Next →](#) For Videos Join Our Youtube Channel: [Join Now](#)

Help Others, Please Share



Test in Under 15 Minutes

TestProject makes open source testing efficient. No framework headaches required.

TestProject

S

Learn Latest Tutorials

SoapUI tutorial SoapUI	RPA tutorial RPA	manual testing tutorial Manual T.	cucumber tutorial Cucumber
Appium tutorial Appium	postgresql tutorial PostgreSQL	Apache Solr Tutorial Solr	MongoDB tutorial MongoDB
Gimp Tutorial Gimp	Verilog Tutorial Verilog	Teradata Tutorial Teradata	PhoneGap Tutorial PhoneGap

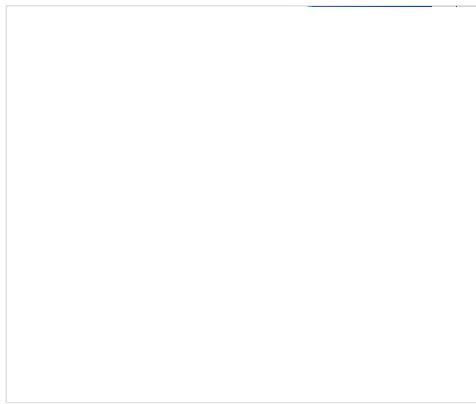
Preparation

Java 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.

```
class CommandLineExample{  
    public static void main(String args[]){  
        System.out.println("Your first argument is: "+args[0]);  
    }  
}
```

```
compile by > javac CommandLineExample.java  
run by > java CommandLineExample sonoo
```

```
Output: Your first argument is: sonoo
```



Low Cost Programming Courses

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.

```
class A{  
    public static void main(String args[]){  
  
        for(int i=0;i<args.length;i++)  
            System.out.println(args[i]);  
  
    }  
}
```

```
compile by > javac A.java  
run by > java A sonoo jaiswal 1 3 abc
```

```
Output: sonoo  
        jaiswal  
        1  
        3  
        abc
```

[← Prev](#)[Next →](#)

[For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



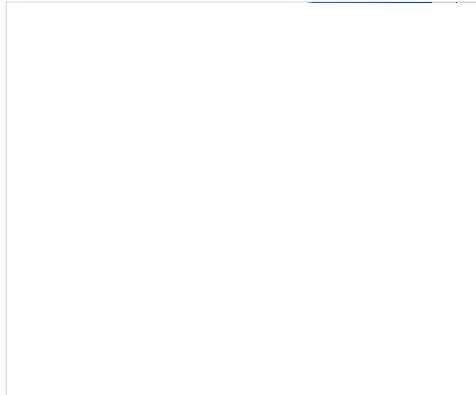
Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

No.	Object	Class
1)	Object is an instance of a class.	Class is a blueprint or template from which objects are created.
2)	Object is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a group of similar objects .
3)	Object is a physical entity.	Class is a logical entity.
4)	Object is created through new keyword mainly e.g. Student s1=new Student();	Class is declared using class keyword e.g. class Student{}
5)	Object is created many times as per requirement.	Class is declared once .
6)	Object allocates memory when it is created .	Class doesn't allocated memory when it is created .
7)	There are many ways to create object in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only one way to define class in java using class keyword.

Let's see some real life example of class and object in java to understand the difference well:

Class: Human **Object:** Man, Woman



Class: Fruit **Object:** Apple, Banana, Mango, Guava wtc.

Class: Mobile phone **Object:** iPhone, Samsung, Moto

Class: Food **Object:** Pizza, Burger, Samosa

ⓘ ×

Test in Under 15 Minutes

TestProject

Sig

← Prev

Next →

[Youtube](#) For Videos Join Our Youtube Channel: [Join Now](#)

Help Others, Please Share



ⓘ ×

Test in Under 15 Minutes

TestProject makes open source testing efficient. No fridges required.

TestProject

S

Learn Latest Tutorials

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used <i>to increase the readability</i> of the program.	Method overriding is used <i>to provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

(i) X

Test in Under 15 Minutes

TestProject

Sig

Java Method Overloading example

```
class OverloadingExample{
    static int add(int a,int b){return a+b;}
    static int add(int a,int b,int c){return a+b+c;}
}
```

Java Method Overriding example

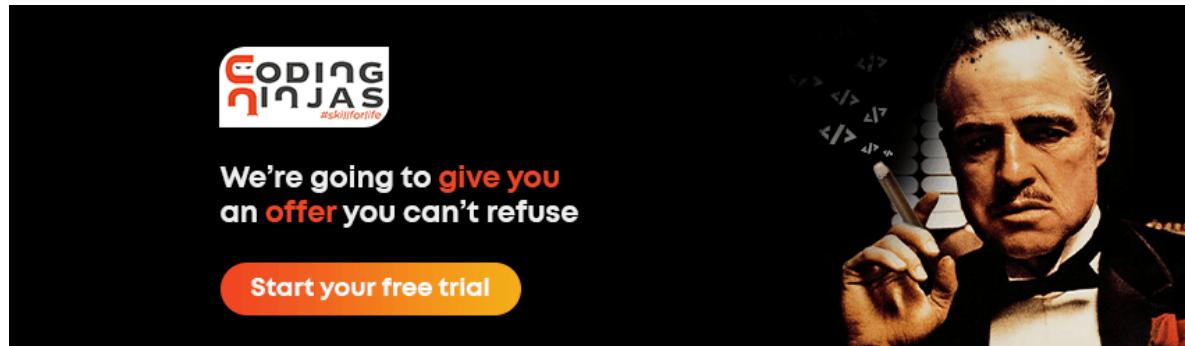
```
class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
    void eat(){System.out.println("eating bread...");}
}
```

[← Prev](#)[Next →](#)

[Youtube](#) For Videos Join Our Youtube Channel: [Join Now](#)

Help Others, Please Share



Learn Latest Tutorials

SoapUI tutorial SoapUI	RPA tutorial RPA	manual testing tutorial Manual T.	cucumber tutorial Cucumber
Appium tutorial Appium	postgresql tutorial PostgreSQL	Apache Solr Tutorial Solr	MongoDB tutorial MongoDB
Gimp Tutorial Gimp	Verilog Tutorial Verilog	Teradata Tutorial Teradata	PhoneGap Tutorial PhoneGap

Preparation

Aptitude	Logical Reasoning	Verbal Ability	Interview Questions
Aptitude	Reasoning	Verbal A.	Interview

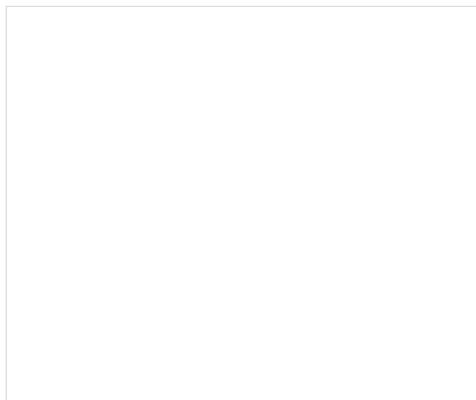
Difference between JDK, JRE, and JVM

We must understand the differences between JDK, JRE, and JVM before proceeding further to [Java](#). See the brief overview of JVM here.

If you want to get the detailed knowledge of Java Virtual Machine, move to the next page. Firstly, let's see the differences between the JDK, JRE, and JVM.

JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.



JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each [OS](#) is different from each other. However, Java is platform independent. There are three notions of the JVM: *specification*, *implementation*, and *instance*.

The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

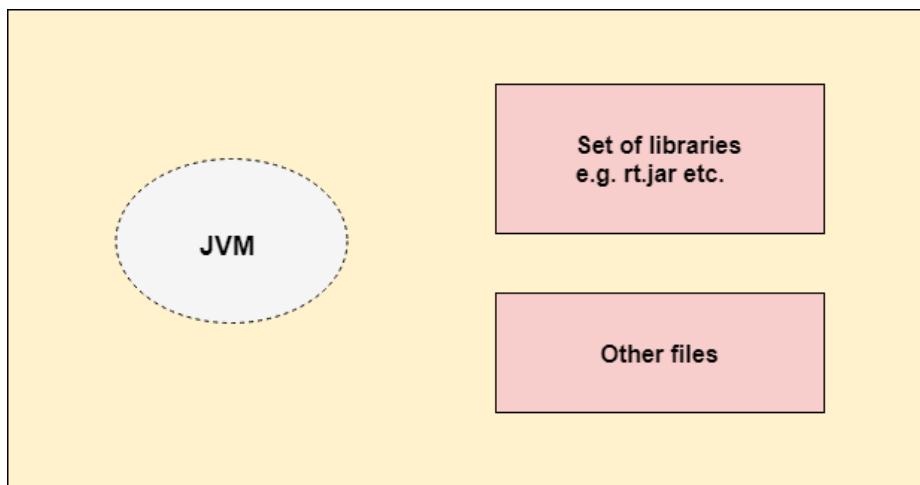
[More Details.](#)



JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



JDK

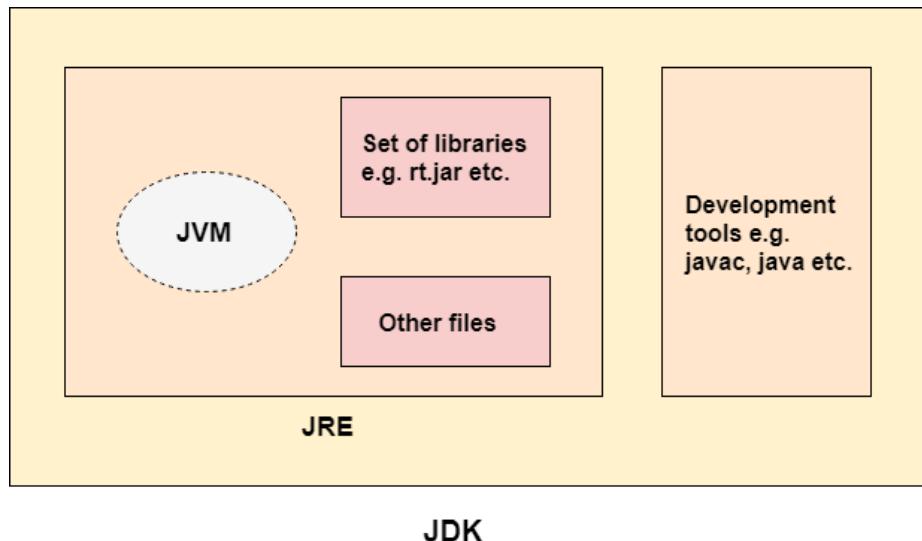
JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and **applets**. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform



The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



[More Details.](#)

Reference Video

0:00 / 5:29

[Video Player Area]

← Prev

Next →

[For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



OutSystems, Software Development Platform Leader, Raises \$150 Million Investment at \$9.5 Billion Valuation

[Learn](#)

Learn Latest Tutorials

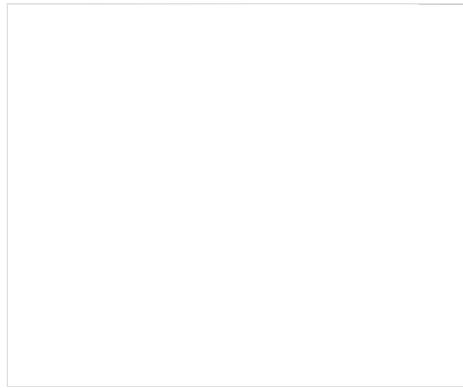


Java Regex

The **Java Regex** or Regular Expression is an API to *define a pattern for searching or manipulating strings*.

It is widely used to define the constraint on strings such as password and email validation. After learning Java regex tutorial, you will be able to test your regular expressions by the Java Regex Tester Tool.

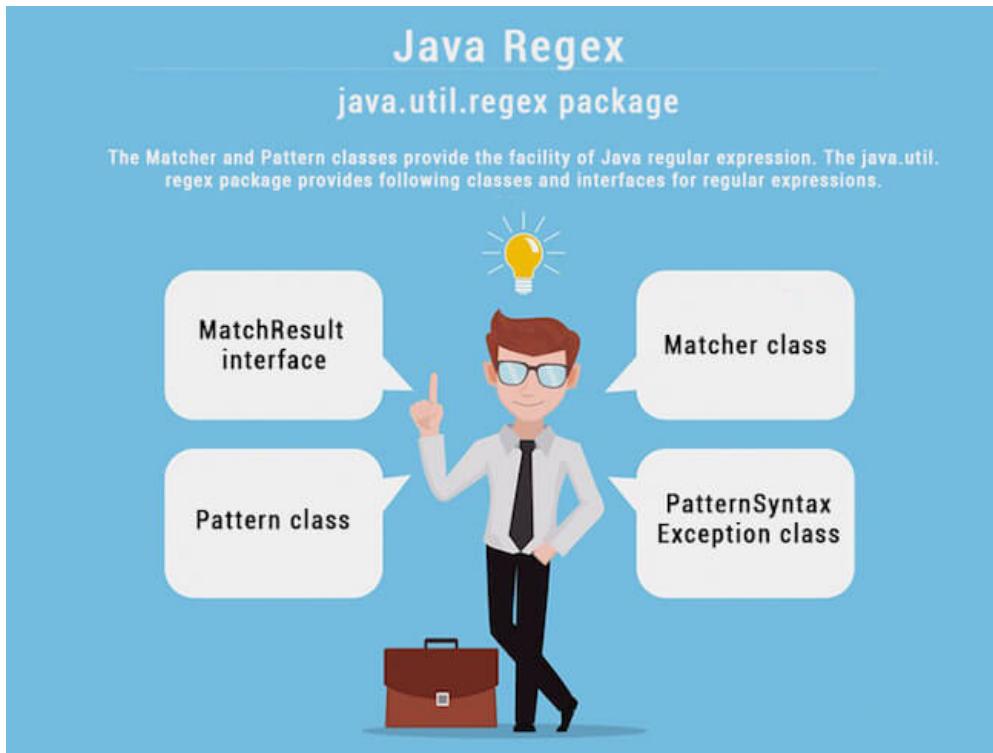
Java Regex API provides 1 interface and 3 classes in **java.util.regex** package.



java.util.regex package

The Matcher and Pattern classes provide the facility of Java regular expression. The **java.util.regex** package provides following classes and interfaces for regular expressions.

1. MatchResult interface
2. Matcher class
3. Pattern class
4. PatternSyntaxException class



Matcher class

It implements the **MatchResult** interface. It is a *regex engine* which is used to perform match operations on a character sequence.

No.	Method	Description
1	boolean matches()	test whether the regular expression matches the pattern.
2	boolean find()	finds the next expression that matches the pattern.
3	boolean find(int start)	finds the next expression that matches the pattern from the given start number.
4	String group()	returns the matched subsequence.
5	int start()	returns the starting index of the matched subsequence.
6	int end()	returns the ending index of the matched subsequence.
7	int groupCount()	returns the total number of the matched subsequence.

Pattern class

It is the *compiled version of a regular expression*. It is used to define a pattern for the regex engine.

No.	Method	Description
1	static Pattern compile(String regex)	compiles the given regex and returns the instance of the Pattern.
2	Matcher matcher(CharSequence input)	creates a matcher that matches the given input with the pattern.
3	static boolean matches(String regex, CharSequence input)	It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.
4	String[] split(CharSequence input)	splits the given input string around matches of given pattern.
5	String pattern()	returns the regex pattern.

Example of Java Regular Expressions

There are three ways to write the regex example in Java.

```

import java.util.regex.*;
public class RegexExample1{
    public static void main(String args[]){
        //1st way
        Pattern p = Pattern.compile(".s");//. represents single character
        Matcher m = p.matcher("as");
        boolean b = m.matches();

        //2nd way
        boolean b2=Pattern.compile(".s").matcher("as").matches();

        //3rd way
        boolean b3 = Pattern.matches(".s", "as");

        System.out.println(b+" "+b2+" "+b3);
    }
}

```

Test it Now

Output

```
true true true
```

Regular Expression . Example

The . (dot) represents a single character.

```
import java.util.regex.*;
class RegexExample2{
public static void main(String args[]){
System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)
System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)
System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)
System.out.println(Pattern.matches(".s", "amms"));//false (has more than 2 char)
System.out.println(Pattern.matches(..s, "mas"));//true (3rd char is s)
}}
```

[Test it Now](#)

Regex Character classes

No.	Character Class	Description
1	[abc]	a, b, or c (simple class)
2	[^abc]	Any character except a, b, or c (negation)
3	[a-zA-Z]	a through z or A through Z, inclusive (range)
4	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
5	[a-z&&[def]]	d, e, or f (intersection)
6	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
7	[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

Regular Expression Character classes Example

```
import java.util.regex.*;
class RegexExample3{
public static void main(String args[]){
System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)
System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)
System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a comes more than once)
}}
```

[Test it Now](#)

Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

Regex	Description
X?	X occurs once or not at all

X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only
X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

Regular Expression Character classes and Quantifiers Example

```

import java.util.regex.*;
class RegexExample4{
public static void main(String args[]){
System.out.println("? quantifier ....");
System.out.println(Pattern.matches("[amn]?", "a")); //true (a or m or n comes one time)
System.out.println(Pattern.matches("[amn]?", "aaa")); //false (a comes more than one time)
System.out.println(Pattern.matches("[amn]?", "aammmnnn")); //false (a m and n comes more than one time)
System.out.println(Pattern.matches("[amn]?", "aazzta")); //false (a comes more than one time)
System.out.println(Pattern.matches("[amn]?", "am")); //false (a or m or n must come one time)

System.out.println("+ quantifier ....");
System.out.println(Pattern.matches("[amn]+", "a")); //true (a or m or n once or more times)
System.out.println(Pattern.matches("[amn]+", "aaa")); //true (a comes more than one time)
System.out.println(Pattern.matches("[amn]+", "aammmnnn")); //true (a or m or n comes more than once)
System.out.println(Pattern.matches("[amn]+", "aazzta")); //false (z and t are not matching pattern)

System.out.println("* quantifier ....");
System.out.println(Pattern.matches("[amn]*", "ammmna")); //true (a or m or n may come zero or more times)
}

```

[Test it Now](#)

Regex Metacharacters

The regular expression metacharacters work as shortcodes.

Regex	Description
.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]

\W	Any non-word character, short for [^\w]
\b	A word boundary
\B	A non word boundary

Regular Expression Metacharacters Example

```

import java.util.regex.*;
class RegexExample5{
public static void main(String args[]){
System.out.println("metacharacters d....");\\d means digit

System.out.println(Pattern.matches("\\d", "abc")); //false (non-digit)
System.out.println(Pattern.matches("\\d", "1")); //true (digit and comes once)
System.out.println(Pattern.matches("\\d", "4443")); //false (digit but comes more than once)
System.out.println(Pattern.matches("\\d", "323abc")); //false (digit and char)

System.out.println("metacharacters D....");\\D means non-digit

System.out.println(Pattern.matches("\\D", "abc")); //false (non-digit but comes more than once)
System.out.println(Pattern.matches("\\D", "1")); //false (digit)
System.out.println(Pattern.matches("\\D", "4443")); //false (digit)
System.out.println(Pattern.matches("\\D", "323abc")); //false (digit and char)
System.out.println(Pattern.matches("\\D", "m")); //true (non-digit and comes once)

System.out.println("metacharacters D with quantifier...");
System.out.println(Pattern.matches("\\D*", "mak")); //true (non-digit and may come 0 or more times)

}}

```

Test it Now

Regular Expression Question 1

```

/*Create a regular expression that accepts alphanumeric characters only.
Its length must be six characters long only.*/

import java.util.regex.*;
class RegexExample6{
public static void main(String args[]){
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32")); //true
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32")); //false (more than 6 char)
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2")); //true
System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2")); //false ($ is not matched)
}}

```

Test it Now

Regular Expression Question 2

```

/*Create a regular expression that accepts 10 digit numeric characters
starting with 7, 8 or 9 only.*/

import java.util.regex.*;
class RegexExample7{
public static void main(String args[]){
System.out.println("by character classes and quantifiers ...");
System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949")); //true
System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949")); //true

System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490")); //false (11 characters)
System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949")); //false (starts from 6)
System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949")); //true

System.out.println("by metacharacters ...");
System.out.println(Pattern.matches("[789]{1}\\\d{9}", "8853038949")); //true
System.out.println(Pattern.matches("[789]{1}\\\d{9}", "3853038949")); //false (starts from 3)

}}

```

[Test it Now](#)

Java Regex Finder Example

```

import java.util.regex.Pattern;
import java.util.Scanner;
import java.util.regex.Matcher;
public class RegexExample8{
public static void main(String[] args){
Scanner sc=new Scanner(System.in);
while (true) {
    System.out.println("Enter regex pattern:");
    Pattern pattern = Pattern.compile(sc.nextLine());
    System.out.println("Enter text:");
    Matcher matcher = pattern.matcher(sc.nextLine());
    boolean found = false;
    while (matcher.find()) {
        System.out.println("I found the text "+matcher.group()+" starting at index "+
        matcher.start()+" and ending at index "+matcher.end());
        found = true;
    }
    if(!found){
        System.out.println("No match found.");
    }
}

```

```

    }
}

```

Output:

```

Enter regex pattern: java
Enter text: this is java, do you know java
I found the text java starting at index 8 and ending at index 12
I found the text java starting at index 26 and ending at index 30

```

← Prev

Next →

[Youtube For Videos Join Our Youtube Channel: Join Now](#)

Help Others, Please Share



Learn Latest Tutorials

SoapUI tutorial SoapUI	RPA tutorial RPA	manual testing tutorial Manual T.	cucumber tutorial Cucumber
Appium tutorial Appium	postgresql tutorial PostgreSQL	Apache Solr Tutorial Solr	MongoDB tutorial MongoDB
Gimp Tutorial Gimp	Verilog Tutorial Verilog	Teradata Tutorial Teradata	PhoneGap Tutorial PhoneGap

Preparation

