Using clone() Method

The **clone()** method is the method of **Object** class. It creates a copy of an object and returns the same copy.

The [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates a new object when the clone() method is invoked. It copies all the content of the previously created object into new one object.

Note that it does not call any constructor. We must implement the **Cloneable** interface while using the clone() method.

The method throws **CloneNotSupportedException** exception if the object's class does not support the Cloneable interface.

The subclasses that override the clone() method can throw an exception if an instance cannot be cloned.

Note: The method creates a copy of the object not a new object.

**Syntax:**

1. **protected** Object clone() **throws** CloneNotSupportedException

We use the following statement to create a new object.

1. ClassName newobject = (ClassName) oldobject.clone();

**CreateObjectExample3.java**

1. **public** **class** CreateObjectExample3 **implements** Cloneable
2. {
3. @Override
4. **protected** Object clone() **throws** CloneNotSupportedException
5. {
6. //invokes the clone() method of the super class
7. **return** **super**.clone();
8. }
9. String str = "New Object Created";
10. **public** **static** **void** main(String[] args)
11. {
12. //creating an object of the class
13. CreateObjectExample3 obj1 = **new** CreateObjectExample3();
14. //try catch block to catch the exception thrown by the method
15. **try**
16. {
17. //creating a new object of the obj1 suing the clone() method
18. CreateObjectExample3 obj2 = (CreateObjectExample3) obj1.clone();
19. System.out.println(obj2.str);
20. }
21. **catch** (CloneNotSupportedException e)
22. {
23. e.printStackTrace();
24. }
25. }
26. }

**Output:**

New Object Created

Using newInstance() Method of Class class

The **newInstance()** method of the Class class is also used to create an object.

It calls the default constructor to create the object. It returns a newly created instance of the class represented by the object.

It internally uses the newInstance() method of the Constructor class.

**Syntax:**

1. **public** T newInstance() **throws** InstantiationException, IllegalAccessException

It throws the **IllegalAccessException,** **InstantiationException, ExceptionInInitializerError** exceptions.

We can create an object in the following ways:

1. ClassName object = ClassName.**class**.newInstance();

**Or**

1. ClassName object = (ClassName) Class.forName("fully qualified name of the class").newInstance();

In the above statement, **forName()** is a static method of Class class. It parses a parameter **className** of type String. It returns the object for the class with the fully qualified name. It loads the class but does not create any object. It throws **ClassNotFoundException** if the class cannot be loaded and **LinkageError** if the linkage fails.

To create the object, we use the **newInstance()** method of the Class class. It works only when we know the name of the class and the class has a public default constructor.

In the following program, we have creates a new object using the newInstance() method.

**CreateObjectExample4.java**

1. **public** **class** CreateObjectExample4
2. {
3. **void** show()
4. {
5. System.out.println("A new object created.");
6. }
7. **public** **static** **void** main(String[] args)
8. {
9. **try**
10. {
11. //creating an instance of Class class
12. Class cls = Class.forName("CreateObjectExample4");
13. //creates an instance of the class using the newInstance() method
14. CreateObjectExample4 obj = (CreateObjectExample4) cls.newInstance();
15. //invoking the show() method
16. obj.show();
17. }
18. **catch** (ClassNotFoundException e)
19. {
20. e.printStackTrace();
21. }
22. **catch** (InstantiationException e)
23. {
24. e.printStackTrace();
25. }
26. **catch** (IllegalAccessException e)
27. {
28. e.printStackTrace();
29. }
30. }
31. }

**Output:**

A new object created.

Using newInstance() Method of Constructor class

It is similar to the **newInstance()** method of the **Class** class. It is known as a reflective way to create objects.

The method is defined in the **Constructor** class which is the class of java.lang.reflect package.

We can also call the parameterized constructor and private constructor by using the **newInstance()** method.

It is widely preferred in comparison to [newInstance() method](https://www.javatpoint.com/new-instance()-method) of the Class class.

**Syntax:**

1. **public** T newInstance(Object... initargs) **throws** InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException

The method parses an array of Objects as an argument. The values of primitive types wrapped in a wrapper Object of the appropriate type. It returns a new object created by calling the constructor. It throws **IllegalAccessException, IllegalArgumentException, InstantiationException, InvocationTargetException, ExceptionInInitializerError Exceptions**.

We can create an object in the following way:

1. Constructor<Employee> constructor = Employee.**class**.getConstructor();
2. Employee emp3 = constructor.newInstance();

Let's create a program that creates an object using the newInstance() method.

**CreateObjectExample5.java**

1. **import** java.lang.reflect.\*;
2. **public** **class** CreateObjectExample5
3. {
4. **private** String str;
5. CreateObjectExample5()
6. {
7. }
8. **public** **void** setName(String str)
9. {
10. **this**.str = str;
11. }
12. **public** **static** **void** main(String[] args)
13. {
14. **try**
15. {
16. Constructor<CreateObjectExample5> constructor = CreateObjectExample5.**class**.getDeclaredConstructor();
17. CreateObjectExample5 r = constructor.newInstance();
18. r.setName("JavaTpoint");
19. System.out.println(r.str);
20. }
21. **catch** (Exception e)
22. {
23. e.printStackTrace();
24. }
25. }
26. }

**Output:**

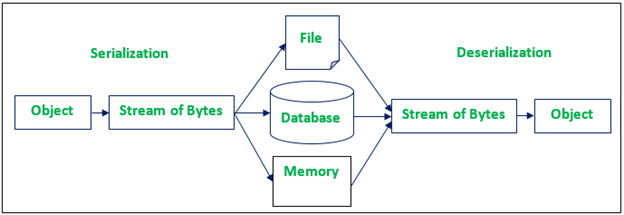
JavaTpoint

Using Deserialization

In Java, **serialization** is the process of converting an object into a sequence of byte-stream. The reverse process (byte-stream to object) of serialization is called **deserialization**.

The JVM creates a new object when we serialize or deserialize an object. It does not use constructor to create an object.

While using deserialization, the **Serializable** interface (marker interface) must be implemented in the class.



**Serialization:**

 The **writeObject()** method of the **ObjectOutputStream** class is used to serialize an object. It sends the object to the output stream.

**Syntax:**

1. **public** **final** **void** writeObject(object x) **throws** IOException

**Deserialization:** The method **readObject()** of **ObjectInputStream** class is used to deserialize an object. It references objects out of a stream.

**Syntax:**

1. **public** **final** Object readObject() **throws** IOException,ClassNotFoundException

Note: Make the filed static or transient if we do not want to include a field as a part of the object. It will not include in the serialization process.

Let's understand the serialization and deserialization through a program.

**Employee.java**

1. **import** java.io.Serializable;
2. **public** **class** Employee **implements** Serializable
3. {
4. **int** empid;
5. String empname;
6. **public** Empoyee(**int** empid, String empname)
7. {
8. **this**.empid = empid;
9. **this**.empname = empname;
10. }
11. }

We have created a class named **Employee** whose object is to be serialized and deserialized.

Serialization of Java Object:

In the following program, we have serialized an object of Employee class by using the **writeObject()** method of the ObjectOutputStream class. The state of the object is saved in the **employee.txt** file.

**SerializationExample.java**

1. **import** java.io.\*;
2. **class** SerializationExample
3. {
4. **public** **static** **void** main(String args[])
5. {
6. Try
7. {
8. //Creating the object
9. Employee emp = **new** Employee(198054,"Andrew");
10. //Creates a stream and writes the object
11. FileOutputStream fout=**new** FileOutputStream("employee.txt");
12. ObjectOutputStream out=**new** ObjectOutputStream(employeeout);
13. out.writeObject(emp);
14. out.flush();
15. //closes the output stream
16. out.close();
17. System.out.println("Successfully Created");
18. }
19. **catch**(Exception e)
20. {
21. System.out.println(e);
22. }
23. }
24. }

**Output:**

Successfully Created

Deserialization of Java Object:

In the following program, we going to deserialize an object that we have serialized in the above program.

**DeserializationExample.java**

1. **import** java.io.\*;
2. **class** DeserializationExample
3. {
4. **public** **static** **void** main(String args[])
5. {
6. **try**
7. {
8. //Creating a stream to read the object
9. ObjectInputStream in=**new** ObjectInputStream(**new** FileInputStream("employee.txt"));
10. Employee e=(Employee)in.readObject();
11. //prints the data of the serialized object
12. System.out.println(e.empid+" "+e.empname);
13. //closing the input stream
14. in.close();
15. }
16. **catch**(Exception e)
17. {
18. System.out.println(e);
19. }
20. }
21. }

**Output:**

198054 Andrew

# **Constructors in Java**

1. [Types of constructors](https://www.javatpoint.com/java-constructor" \l "constypes)
   1. [Default Constructor](https://www.javatpoint.com/java-constructor" \l "consdef)
   2. [Parameterized Constructor](https://www.javatpoint.com/java-constructor" \l "conspara)
2. [Constructor Overloading](https://www.javatpoint.com/java-constructor" \l "consoverloading)
3. [Does constructor return any value?](https://www.javatpoint.com/java-constructor" \l "consdoesreturn)
4. [Copying the values of one object into another](https://www.javatpoint.com/java-constructor" \l "conscopy)
5. [Does constructor perform other tasks instead of the initialization](https://www.javatpoint.com/java-constructor" \l "consothertask)

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

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

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](https://www.javatpoint.com/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



## Java Default Constructor

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

### Syntax of default constructor:

1. <class\_name>(){}

## Example of default constructor

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

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

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

1. //Let us see another example of default constructor
2. //which displays the default values
3. **class** Student3{
4. **int** id;
5. String name;
6. //method to display the value of id and name
7. **void** display(){System.out.println(id+" "+name);}
9. **public** **static** **void** main(String args[]){
10. //creating objects
11. Student3 s1=**new** Student3();
12. Student3 s2=**new** Student3();
13. //displaying values of the object
14. s1.display();
15. s2.display();
16. }
17. }

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.

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

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=Student4" \t "_blank)**

Output:

111 Karan

222 Aryan

## Constructor Overloading in Java

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

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

### Example of Constructor Overloading

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

Output:

111 Karan 0

222 Aryan 25

## Difference between constructor and method in Java

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

## Java Copy Constructor

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

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

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

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

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

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.

1. **class** Student7{
2. **int** id;
3. String name;
4. Student7(**int** i,String n){
5. id = i;
6. name = n;
7. }
8. Student7(){}
9. **void** display(){System.out.println(id+" "+name);}
11. **public** **static** **void** main(String args[]){
12. Student7 s1 = **new** Student7(111,"Karan");
13. Student7 s2 = **new** Student7();
14. s2.id=s1.id;
15. s2.name=s1.name;
16. s1.display();
17. s2.display();
18. }
19. }

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.

### **Java Polymorphism**

### Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

### Like we specified in the previous chapter; [Inheritance](https://www.w3schools.com/java/java_inheritance.asp) lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

### For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

### Example

class Animal {

public void animalSound() {

System.out.println("The animal makes a sound");

}}

class Pig extends Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}}

class Dog extends Animal {

public void animalSound() {

System.out.println("The dog says: bow wow");

}}

Remember from the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp) that we use the extends keyword to inherit from a class.

Now we can create Pig and Dog objects and call the animalSound() method on both of them:

### Example

class Animal {

public void animalSound() {

System.out.println("The animal makes a sound");

}}

class Pig extends Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}}

class Dog extends Animal {

public void animalSound() {

System.out.println("The dog says: bow wow");

}}

class MyMainClass {

public static void main(String[] args) {

Animal myAnimal = new Animal(); // Create a Animal object

Animal myPig = new Pig(); // Create a Pig object

Animal myDog = new Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}}

# Java Inheritance

## Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

* ****subclass**** (child) - the class that inherits from another class
* ****superclass**** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

### Example

class Vehicle {

protected String brand = "Ford"; // Vehicle attribute

public void honk() { // Vehicle method

System.out.println("Tuut, tuut!");

}}

class Car extends Vehicle {

private String modelName = "Mustang"; // Car attribute

public static void main(String[] args) {

// Create a myCar object

Car myCar = new Car();

// Call the honk() method (from the Vehicle class) on the myCar object

myCar.honk();

// Display the value of the brand attribute (from the Vehicle class) and the value of the modelName from the Car class

System.out.println(myCar.brand + " " + myCar.modelName);

}}

## Encapsulation

The meaning of ****Encapsulation****, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

* declare class variables/attributes as private
* provide public ****get**** and ****set**** methods to access and update the value of a private variable

## Get and Set

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public ****get**** and ****set**** methods.

The get method returns the variable value, and the set method sets the value.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

public class Person {

private String name; // private = restricted access

// Getter

public String getName() {

return name;

}

// Setter

public void setName(String newName) {

this.name = newName;

}}

public class MyClass {

public static void main(String[] args) {

Person myObj = new Person();

myObj.setName("John"); // Set the value of the name variable to "John"

System.out.println(myObj.getName());

}}

// Outputs "John"

# Java Abstraction

## Abstract Classes and Methods

Data ****abstraction**** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either ****abstract classes**** or **[interfaces](https://www.w3schools.com/java/java_interface.asp)** (which you will learn more about in the next chapter).

The abstract keyword is a non-access modifier, used for classes and methods:

* ****Abstract class:**** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* ****Abstract method:**** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

abstract class Animal {

public abstract void animalSound();

public void sleep() {

System.out.println("Zzz");

}}

From the example above, it is not possible to create an object of the Animal class:

Animal myObj = new Animal(); // will generate an error

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the [Polymorphism](https://www.w3schools.com/java/java_polymorphism.asp) chapter to an abstract class:

Remember from the [Inheritance chapter](https://www.w3schools.com/java/java_inheritance.asp) that we use the extends keyword to inherit from a class.

### Example

// Abstract classabstract class Animal {

// Abstract method (does not have a body)

public abstract void animalSound();

// Regular method

public void sleep() {

System.out.println("Zzz");

}}

// Subclass (inherit from Animal)class Pig extends Animal {

public void animalSound() {

// The body of animalSound() is provided here

System.out.println("The pig says: wee wee");

}}

class MyMainClass {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}}