***@ Constructor***

# A breakdown of constructors concept:

Constructor is a block of code that initializes the newly created object. A constructor resembles an instance method in java but it’s not a method as it doesn’t have a return type. In short constructor and method are different(More on this at the end of this guide). People often refer constructor as special type of method in Java.

Constructor has same name as the class and looks like this in a java code.

1. public class MyClass{
2. //This is the constructor
3. MyClass(){
4. }
5. ..
6. }

Note that the constructor name matches with the class name and it doesn’t have a return type.

**How does a constructor work ?**

To understand the working of constructor, lets take an example. lets say we have a class MyClass.  
When we create the object of MyClass like this:

1. MyClass obj = new MyClass()

The **new keyword** here creates the object of class MyClass and invokes the constructor to initialize this newly created object.

You may get a little lost here as I have not shown you any initialization example, lets have a look at the code below:

**A simple constructor program in java**

Here we have created an object obj of class Hello and then we displayed the instance variable name of the object. As you can see that the output is Hello which is what we have passed to the name during initialization in constructor. This shows that when we created the object obj the constructor got invoked. In this example we have used **this keyword**, which refers to the current object, object obj in this example. We will cover this keyword in detail in the next tutorial.

1. public class Hello {
2. String name;
3. //Constructor
4. Hello(){
5. this.name = "Hello";
6. }
7. public static void main(String[] args) {
8. Hello obj = new Hello();
9. System.out.println(obj.name);
10. }
11. }

**Output:**

1. Hello

**Types of Constructors**

There are three types of constructors: Default, No-arg constructor and Parameterized.

**Default constructor**

If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code on your behalf. This constructor is known as default constructor. You would not find it in your source code(the java file) as it would be inserted into the code during compilation and exists in .class file. This process is shown in the diagram below:

If you implement any constructor then you no longer receive a default constructor from Java compiler.

**no-arg constructor:**

Constructor with no arguments is known as **no-arg constructor**. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.

Although you may see some people claim that that default and no-arg constructor is same but in fact they are not, even if you write **public Demo() { }** in your class Demo it cannot be called default constructor since you have written the code of it.

**Example: no-arg constructor**

1. class Demo
2. {
3. public Demo()
4. {
5. System.out.println("This is a no argument constructor");
6. }
7. public static void main(String args[]) {
8. new Demo();
9. }
10. }

Output:  
This is a no argument constructor

**Parameterized constructor**

Constructor with arguments(or you can say parameters) is known as Parameterized constructor.

**Example: parameterized constructor**

In this example we have a parameterized constructor with two parameters id and name. While creating the objects obj1 and obj2 I have passed two arguments so that this constructor gets invoked after creation of obj1 and obj2.

1. public class Employee {
3. int empId;
4. String empName;
6. //parameterized constructor with two parameters
7. Employee(int id, String name){
8. this.empId = id;
9. this.empName = name;
10. }
11. void info(){
12. System.out.println("Id: "+empId+" Name: "+empName);
13. }
15. public static void main(String args[]){
16. Employee obj1 = new Employee(10245,"Chaitanya");
17. Employee obj2 = new Employee(92232,"Negan");
18. obj1.info();
19. obj2.info();
20. }
21. }

**Output:**

1. Id: 10245 Name: Chaitanya
2. Id: 92232 Name: Negan

\*\* **Important points**

- Every class has a constructor whether it’s a normal class or a abstract class.

- Constructors are not methods and they don’t have any return type.

- Constructor name should match with class name .

- Constructor can use any access specifier, they can be declared as private also. Private constructors are possible in java but there scope is within the class only.

**- Like constructors method can also have name same as class name, but still they have return type, though which we can identify them that they are methods not constructors.**

- If you don’t implement any constructor within the class, compiler will do it for.

**- this() and super() should be the first statement in the constructor code.** If you don’t mention them, compiler does it for you accordingly.

- Constructor overloading is possible but overriding is not possible. Which means we can have overloaded constructor in our class but we can’t override a constructor.

- Constructors can not be inherited.

- If Super class doesn’t have a no-arg(default) constructor then compiler would not insert a default constructor in child class as it does in normal scenario.

- Interfaces do not have constructors.

- Abstract class can have constructor and it gets invoked when a class, which implements interface, is instantiated. (i.e. object creation of concrete class).

- A constructor can also invoke another constructor of the same class – By using this(). If you want to invoke a parameterized constructor then do it like this: **this(parameter list)**.



[Constructors](https://docs.oracle.com/javase/tutorial/reflect/member/ctor.html#:~:text=A%20constructor%20is%20used%20in,invoked%20or%20fields%20are%20accessed.)

//////////////////////////////////////////////////////////////////////////////////

***# Questions / Answers***

*Q) where does default constructor came from?*

> when you create a new class in Java, normally just one constructor exists with zero parameters, then you ask me: Why? Because behind the scenes ALL classes extends from a default base class of Java, because of that you don't need to make that one manually by yourself.

///////////////////////////////////////////

*Q) If we are able to initialize values using Constructors, what is the point of setters?****(Constructor vs Setter)***

Ans1> In OOP, sometimes when you retrieve the data from database and you store it in your object you can set a new value for some field without creating a new instance of your class, without using the constructor again, with the setter... So for new records constructors helps a lot in Java but when you need to update just one field it's easier to use just the setter...

Also setters can contain some validation or logic to treat the data that was passed to the method and, for concepts and principles of oop, it is a good practice to store the logic of your "Business rules" in just one method, in most of the cases, the setters.

Ans2> Some classes have non parametized constructors i.e.: they have constructors that when called, don't take in parameters but immediately have the fields initialized with values.  
So now trying to update a particular field of an object after creation is impossible. They best way would be to use a SETTER.

Also, it doesn't make sense to always create a new object in java whenever you want to update a field of an object .... this will help with memory management.

Plus both constructors and setter methods can have validations as to how the fields they relate to can be set.

Finally I would suggest that you use a constructor when you want to create an object (which obviously is the reason for it)... and then use setters as a way to update a field of an object .

////////////////////////////////////////////////////////

*Q) Static vs Non-static.*

> Static methods are the methods that are common throughout the class. You need not explicitly create an instance of the class to access that method. That method can be accessed from anywhere in the class.

Example: We have a method like this:

public static int calculateScore(String playerName, int score) {  
System.out.println("Player " + playerName + " scored " + score + " points");  
return score \* 1000;  
}

When I try to access this method from the main method, I need not create an instance of the class to access it. I can just type :

calculateScore("ABC", 500)

But, if it is not a static method, then I have to access it as follows:

(Syntax: Classname Objectref = new Classname;

Objectref.calculateScore("ABC", 500);

///////////////////////////////////////////////////////////

Q) If the field name(class variable name / instance variable) and parameter name(constructor parameter variable) are same, then we use ‘this’ keyword to indicate class variable(to differentiate them). So, question is why we are not using different names at both places and avoid ‘this’ keyword?

Ans >

One of the reasons to use same name is consistency. Another reason is refactoring, when you rename the field IDE will automatically rename getters and parameter names. For example prefixing all parameters with word "new" is not good idea.

Keyword this can also be used with chaining constructors when one constructor calls another constructor. With constructors you cant have different constructor names.

Lets say you have method

1. public void setFirstName(String firstName) {
2. this.firstName = firstName; // parameter has same name as field
3. }

Renaming field firstName to lets say just "name" IDE will automatically offer renaming of params and setter/getter methods

If you rename it manually you will have something like the following

1. public void setName(String firstName) {
2. this.name = firstName; // parameter has same name as field
3. }

That way it is not consistent, name of the method has different meaning than parameter.

There are 3 possible ways of doing it

1) Use the actual field name with an underscore:

1. public C(Type foo\_) {
2. foo = foo\_;
3. }
5. public void setFoo(Type foo\_) {
6. foo = foo\_;
7. }

2) Use the actual field name, just use "this" in setting:

1. public C(Type foo) {
2. this.foo = foo;
3. }
5. public void setFoo(Type foo) {
6. this.foo = foo;
7. }

3) Completely inconsistent things like:

1. public C(Type bar) {
2. this.foo = bar;
3. }
5. public void setFoo(Type bar) {
6. this.foo = bar;
7. }

There are some practices to use.

Do not use the underscore (\_) character in the field name -- underscores are not recommended for non-constant field names.

The use of the underscore character in an identifier is not recommended except for identifiers for constants.

The Variables page of The official Java Tutorials mentions the following about underscores:

If your variable stores a constant value, such as static final int NUM\_GEARS = 6 , the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.  
Since field names are not constants, according to what is written on that page, one should not use underscores in non-constant fields.

IDEs can automatically add Javadoc comments according to the name of the parameter of the method, so having the name of the field in the parameter list would be beneficial.

The following is an example of an automatically generated Javadoc:

1. /\*\*
2. \*
3. \* @param importance <-- Parameter name in Javadoc matches
4. \* the parameter name in the code.
5. \*/
6. public void setImportance(int importance) {
7. this.importance = importance;
8. }

Having the Javadoc reflect the name of the field has another benefit -- IDEs that have code completion can use the field name in the Javadoc in order to automatically fill out parameter names:

// Code completion gives the following:  
this.getImportance(importance);   
Giving meaning to the field name and parameter name will make it easier to understand what the parameter actually represents.

///////////////////////////////////////////////////////////////////

Q)