Chapter 6

Object-Oriented Programming

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Tip: The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

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:

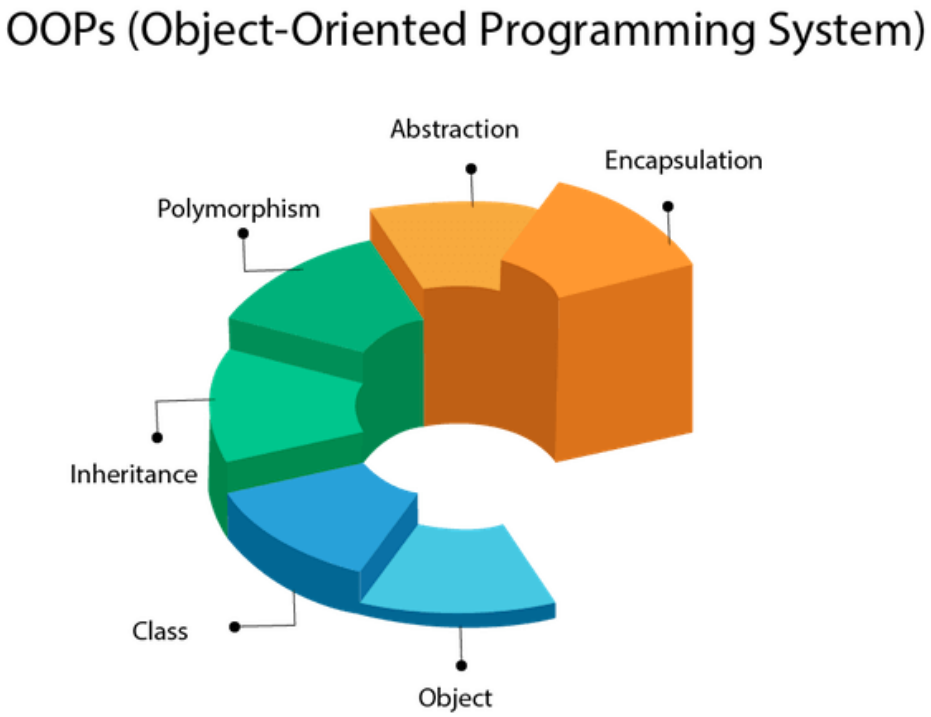
- Object
- **Class**
- **Inheritance**
- **Polymorphism**
- **Abstraction**
- **Encapsulation**

4 Pillars of OOPs

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

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

- **Composition**



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.

OR

The process of representing one Form in multiple forms is known as **Polymorphism**. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.

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.

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.

Advantage of OOPs over Procedure-oriented programming language

1) OOPs make 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.

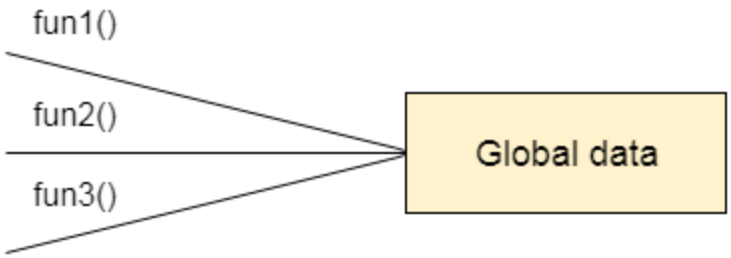


Figure: Data Representation in Procedure-Oriented Programming

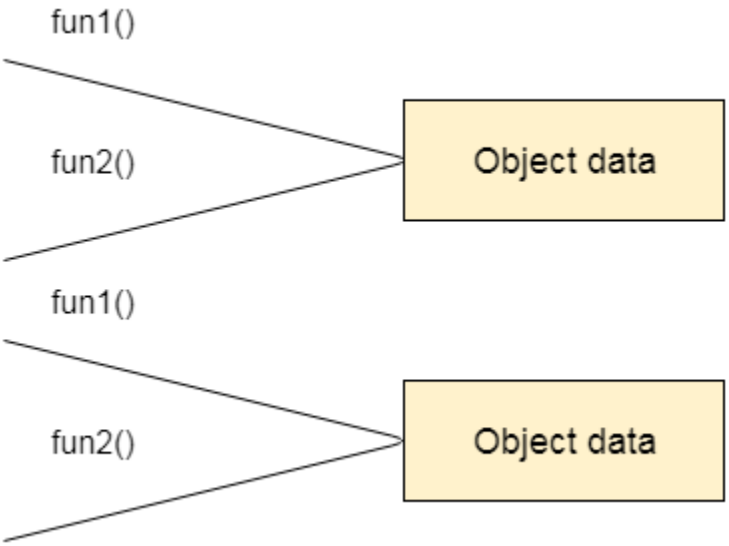


Figure: Data Representation in Object-Oriented Programming

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.

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.

Why Using naming Conventions

Different Java programmers can have different styles and approaches to write program. By using standard Java naming conventions they make their code easier to read for themselves and

for other programmers. Readability of Java code is important because it means less time is spent trying to figure out what the code does, and leaving more time to fix or modify it.

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.

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.

```
package student;  // creating package
import java.lang;  // import package
```

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.
- First letter of every word of class name should exists in upper case.

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

```
class StudentDetails
 {
  .....
  .....
 }
```

Interface

- It should start with the uppercase letter.

- It should be an adjective such as Runnable, WIndowListener, ActionListener.
- Use appropriate words, instead of acronyms.
- First letter of every word of interface name should exists in upper case.

Example: -

```
interface FacultyDetail
 {
  .....
  .....
 }
```

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
}
```

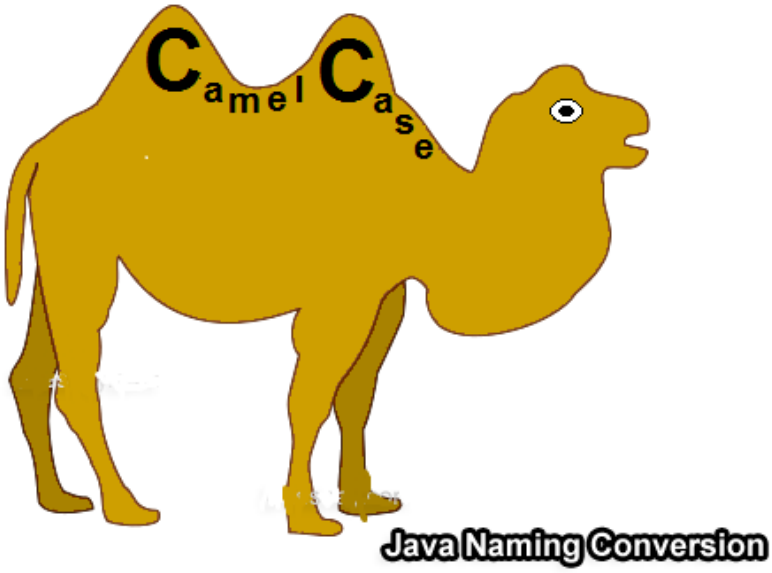
Constant

- 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 camelcase syntax for naming the class, interface, method and variable. According to CamelCase if name is combined with two words, second word will start with uppercase letter always. General Example studentName, customerAccount. In term of java programming e.g. actionPerformed(), firstName, ActionEvent, ActionListener etc.

Object and class in Java

Object is the physical as well as logical entity where as class is the only logical entity.

Class: Class is a blue print which is containing only list of variables and method and no memory is allocated for them. A class is a group of objects that has common properties.

A class in java contains:

- Data Member
- Method
- Constructor
- Block
- Class and Interface

Object: Object is a instance of class, object has state and behaviors.

An Object in java has three characteristics:

- State
- Behavior
- Identity

State: Represents data (value) of an object.

Behavior: Represents the behavior (functionality) of an object such as deposit, withdraw etc.

Identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But,it is used internally by the JVM to identify each object uniquely.

Class is also can be used to achieve user defined data types.

Real life example of object and class

In real world many examples of object and class like dog, cat, and cow are belong to animal's class. Each object has state and behaviors. For example a dog has state:- color, name, height, age as well as behaviors:- barking, eating, and sleeping.

Animal class

Vehicle class

Car, bike, truck these all are belongs to vehicle class. These Objects have also different different states and behaviors. For Example car has state - color, name, model, speed, Mileage. as we; as behaviors - distance travel

Vechicle class

Difference between Class and Object in Java

	Class	Object
1	Class is a container which collection of variables and methods.	object is a instance of class

2	No memory is allocated at the time of declaration	Sufficient memory space will be allocated for all the variables of class at the time of declaration.
3	One class definition should exist only once in the program.	For one class multiple objects can be created.

Syntax to declare a Class

```
class Class_Name
 {
   data member;
   method;
 }
```

Simple Example of Object and Class

In this example, we have created a Employee class that have two data members eid and ename. We are creating the object of the Employee class by new keyword and printing the objects value.

Example

```
class Employee
{
 int eid;  // data member (or instance variable)
 String ename;  // data member (or instance variable)
 eid=101;
 ename="Hitesh";
 public static void main(String args[])
 {
  Employee e=new Employee(); // Creating an object of class Employee
  System.out.println("Employee ID: "+e.eid);
  System.out.println("Name: "+e.ename);
 }
}
```

Output

```
Employee ID: 101

Name: Hitesh
```

Note: A new keyword is used to allocate memory at runtime, new keyword is used for create an object of class, later we discuss all the way for create an object of class.

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

Java Default Constructor

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

<class_name>(){}

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();
}
}
```

> *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();
}
}
```

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();
   }
}
```

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 Student
{
   int id;
   String name;
   int age;
   //creating two arg constructor
   Student (int i,String n){
   id = i;
   name = n;
   }
   //creating three arg constructor
   Student(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[]){
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan",25);
```

```
    s1.display();
    s2.display();
   }
}
```

Output:

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

Difference between constructor and method in Java

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

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

Java Copy Constructor

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

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

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

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

```
//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();
       }
    }
```

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();
  }
}
```

Output:

111 Karan
111 Karan

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?

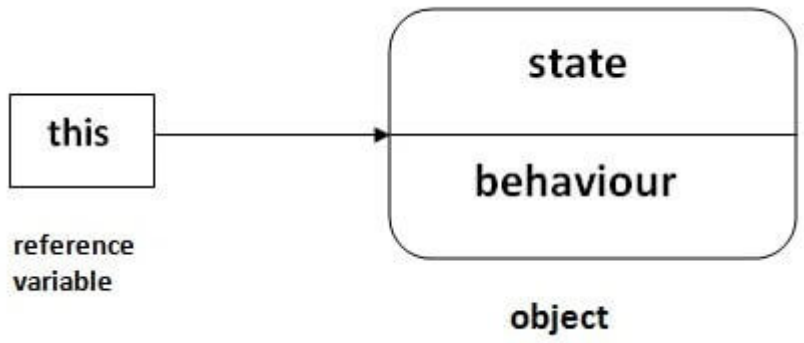
Yes.

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.

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.

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();
}}
```

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();
}}
```

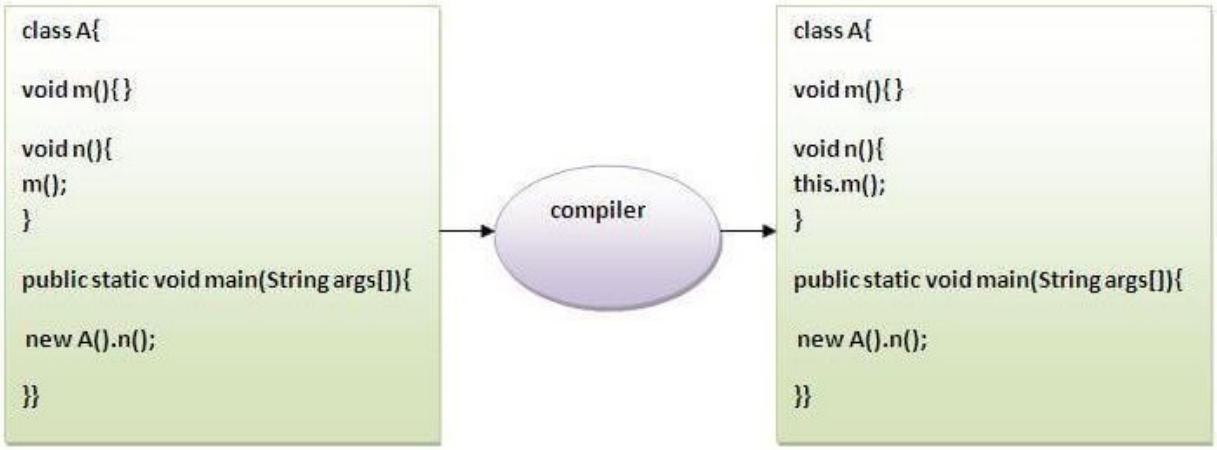
Output:

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

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword.

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();
}}
```

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);
}}
```

Method in Java

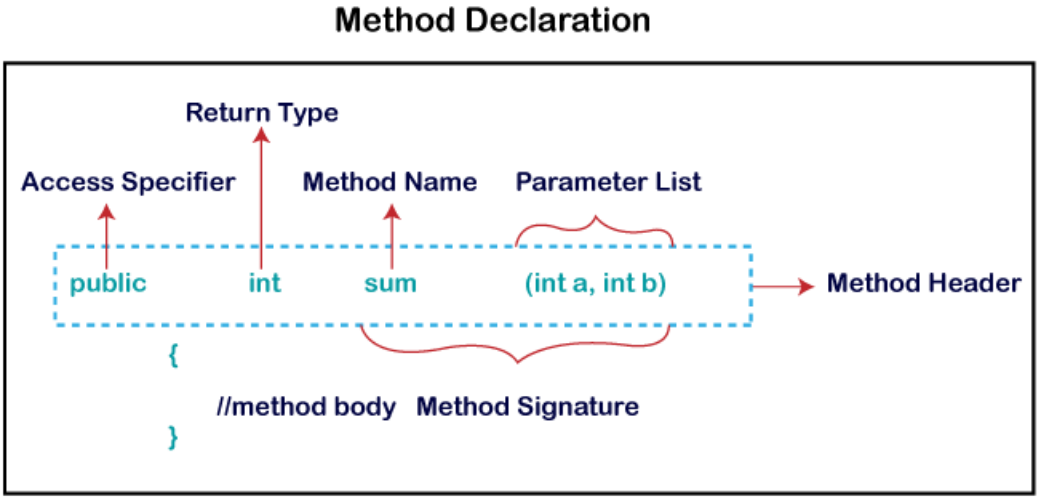
In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**. In this section, we will learn **what is a method in Java, types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

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 Convention of 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.

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 has 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

1. `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 impelmentation
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();**

Variable Argument (Varargs):

The varrags allows the method to accept zero or multiple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

Advantage of Varargs:

We don't have to provide overloaded methods so less code.

Syntax of varargs:

The varargs uses ellipsis[…] i.e. three dots after the data type. Syntax is as follows:

```
<return_type>  <method_name>(<data_type>...< variableName>)
{
}
```

Simple Example of Varargs in java:

```
class VarargsExample1
{
   static void display(String... values)
   {
     System.out.println("display method invoked ");
   }

  public static void main(String args[])
  {
   display();//zero argument
   display("my","name","is","varargs");//four arguments
  }
}
```

Output:display method invoked
display method invoked

Another Program of Varargs in java:

```
class VarargsExample
{
    static void display(String... values)
   {
     System.out.println("display method invoked ");
     for(String  s : values)
     {
       System.out.println(s);
     }
   }
  public static void main(String args[])
  {
    display();//zero argument
    display("hello");//one argument
    display("my","name","is","varargs");//four arguments
  }
}
```

Output:display method invoked
display method invoked
hello
display method invoked
my
name
is
varargs

Rules for varargs:

While using the varargs, you must follow some rules otherwise program code won't compile. The rules are as follows:

- There can be only one variable argument in the method.
- Variable argument (varargs) must be the last argument.

Examples of varargs that fails to compile:

```
void method(String... a, int... b) //Compile time error
{
}

void method(int... a, String b)//Compile time error
{
}
```

Example of Varargs that is the last argument in the method:

```
class VarargsExample3
{

 static void display(int num, String... values)
 {
  System.out.println("number is "+num);
  for(String s:values)
  {
    System.out.println(s);
  }
 }

 public static void main(String args[])
 {
    display(500,"hello");//one argument
    display(1000,"my","name","is","varargs");//four arguments
  }
}
```

Output:number is 500
hello
number is 1000
my
name
is
varargs

A varargs method can also be overloaded by a non-varargs method. For example, **vaTest(int x)** is a valid overload of **vaTest( )** in the foregoing program. This version is invoked only when one **int** argument is present. When two or more **int** arguments are passed, the varargs version **vaTest(int…v)** is used.

Varargs and Ambiguity

Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method. For example, consider the following program:

```
// Varargs, overloading, and ambiguity.

class VarArgs
{
  static void vaTest(int ... v)
  {
    System.out.print("vaTest(int ...): " + "Number of args: " + v.length +
     " Contents: ");

    for(int x : v)
        System.out.print(x + " ");

    System.out.println();
  }
  static void vaTest(boolean ... v)
  {
    System.out.print("vaTest(boolean ...) " + "Number of args: " +
                        v.length + " Contents: ");
    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }
  public static void main(String args[])
  {
   vaTest(1, 2, 3); // OK
   vaTest(true, false, false); // OK
   vaTest(); // Error: Ambiguous! Compile
  }
}
```

In this program, the overloading of **vaTest( )** is perfectly correct. However, this program will not compile because of the following call:

```
vaTest(); // Error: Ambiguous!
```