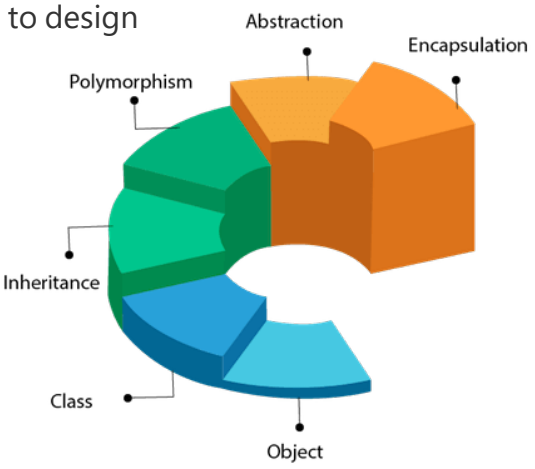
Java

Object Oriented Programming Notes

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

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.

Object

Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
Example:

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

   public void getInfo() {
        System.out.println("The name of this Student is " + this.name);
        System.out.println("The age of this Student is " + this.age);
   }
}

public class OOPS {
   public static void main(String args[]) {
        Student s1 = new Student();
        s1.name = "Aman";
        s1.age = 24;
        s1.getInfo();

         Student s2 = new Student();
```

Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as **subclass** (derived class, **child class**) and the class whose properties are inherited is known as **superclass** (base class, **parent class**).

extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

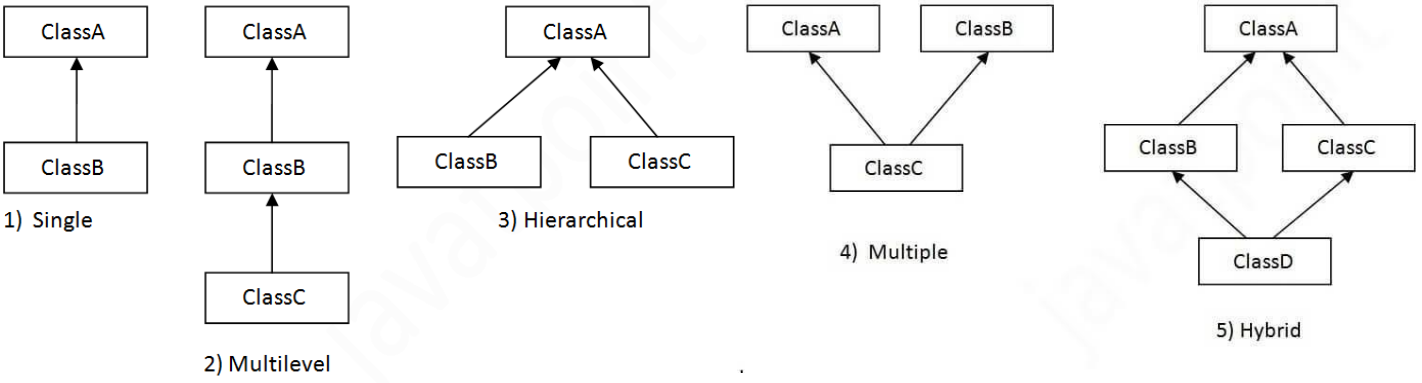
Syntax

```
class Super {
    .....
    .....
}
class Sub extends Super {
    .....
    .....
}
```

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.



Types of Inheritance explained:

1. Single inheritance: When one class inherits another class, it is known as single level inheritance

```
class Shape {
   public void area() {
        System.out.println("Displays Area of Shape");
   }
}
class Triangle extends Shape {
   public void area(int h, int b) {
        System.out.println((1/2)*b*h);
   }
}
```

2. Hierarchical inheritance: Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

```
class Shape {
   public void area() {
        System.out.println("Displays Area of Shape");
   }
}
class Triangle extends Shape {
   public void area(int h, int b) {
        System.out.println((1/2)*b*h);
   }
}
class Circle extends Shape {
   public void area(int r) {
        System.out.println((3.14)*r*r);
   }
}
```

3. Multilevel inheritance: Multilevel inheritance is a process of deriving a class from another derived class.

```
class Shape {
   public void area() {
        System.out.println("Displays Area of Shape");
   }
}
class Triangle extends Shape {
   public void area(int h, int b) {
        System.out.println((1/2)*b*h);
   }
}
class EquilateralTriangle extends Triangle {
   int side;
}
```

4. Hybrid inheritance: Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.

Polymorphism

Polymorphism is the ability to present the same interface for differing underlying forms (data types). With polymorphism, each of these classes will have different underlying data. Precisely, Poly means 'many' and morphism means 'forms'.

Types of Polymorphism IMP

1. Compile Time Polymorphism (Static)
2. Runtime Polymorphism (Dynamic)

Compile Time Polymorphism: The polymorphism which is implemented at the compile time is known as **compile-time** polymorphism. Example - **Method Overloading**

Method Overloading: Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality. Method overloading can be possible on the following basis:

1. The return type of the overloaded function.
2. The type of the parameters passed to the function.
3. The number of parameters passed to the function.

Code:

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

   public void displayInfo(String name) {
       System.out.println(name);
   }

   public void displayInfo(int age) {
       System.out.println(age);
   }

   public void displayInfo(String name, int age) {
       System.out.println(name);
       System.out.println(age);
   }
}
```

Runtime Polymorphism: Runtime polymorphism is also known as **dynamic polymorphism**. **Function Overriding** is an example of **runtime** polymorphism.

Function Overriding means when the child class contains the method which is already present in the parent class. Hence, **the child class overrides the method of the parent class**. In case of function overriding, parent and child classes both contain the same function with a different definition. The call to the function is determined at runtime is known as runtime polymorphism.

Code:

```
class Shape {
   public void area() {
       System.out.println("Displays Area of Shape");
   }
}
class Triangle extends Shape {
   public void area(int h, int b) {
       System.out.println((1/2)*b*h);
   }
}
class Circle extends Shape {
   public void area(int r) {
       System.out.println((3.14)*r*r);
   }
}
```

Abstraction

Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user.
In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

Abstraction is achieved in 2 ways:

- Abstract class
- Interfaces (Pure Abstraction)

Abstract Class

- 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.

Code:

```
abstract class Animal {
   abstract void walk();
   void breathe() {
       System.out.println("This animal breathes air");
   }
   Animal() {
       System.out.println("You are about to create an Animal.");
   }
}

class Horse extends Animal {
   Horse() {
       System.out.println("Wow, you have created a Horse!");
   }
   void walk() {
       System.out.println("Horse walks on 4 legs");
   }
}

class Chicken extends Animal {
   Chicken() {
       System.out.println("Wow, you have created a Chicken!");
   }
   void walk() {
       System.out.println("Chicken walks on 2 legs");
   }
}

public class OOPS {
   public static void main(String args[]) {
      Horse horse = new Horse();
      horse.walk();
      horse.breathe();
   }
}
```

Interfaces

- All the fields in interfaces are public, static and final by default.
- All methods are public & abstract by default.
- A class that implements an interface must implement all the methods declared in the interface.
- Interfaces support the functionality of multiple inheritance.

Code:

```
interface Animal {
   void walk();
}
class Horse implements Animal {
   public void walk() {
       System.out.println("Horse walks on 4 legs");
   }
}
class Chicken implements Animal {
   public void walk() {
       System.out.println("Chicken walks on 2 legs");
   }
}
public class OOPS {
   public static void main(String args[]) {
      Horse horse = new Horse();
      horse.walk();
   }
}
```

Encapsulation

Encapsulation is the process of combining data and functions into a single unit called class.
In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private or public - **getter** and **setter** methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible. (**Data hiding**: a language feature to restrict access to members of an object, reducing the negative effect due to dependencies. e.g., "protected", "private" feature in Java).

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Code:

```
/* File name: EncapTest.java */
public class EncapTest {
   private String name;
   private String idNum;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getIdNum() {
      return idNum;
   }

   public void setAge( int newAge) {
      age = newAge;
   }

   public void setName(String newName) {
      name = newName;
   }

   public void setIdNum( String newId) {
      idNum = newId;
   }
}
/* The public setXXX() and getXXX() methods are the access points of the instance variables of
the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any
class that wants to access the variables should access them through these getters and setters. */
/* The variables of the EncapTest class can be accessed using the following program */

/* File name: RunEncap.java */
public class RunEncap {

   public static void main(String args[]) {
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name: " + encap.getName() + " Age: " + encap.getAge());
   }
}
```

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program
- **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

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.

Some of the existing packages in Java are –
These packages consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:

1. **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
2. **java.io:** Contains classed for supporting input / output operations.
3. **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support; for Date / Time operations.
4. **java.applet:** Contains classes for creating Applets.
5. **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
6. **java.net:** Contain classes for supporting networking operations.

Here, we will have the detailed learning of creating and using **user-defined packages**.
Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

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.

Creating a Package

While creating a package, you should choose a name for the package and include a package statement along with that name at the **top of every source file** that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the **first line** in the source file. There can be only **one package statement** in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use **-d** option as shown below.

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

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

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.

- ✓ **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();
    }
}
```

- ✓ **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();
        }
}
```

- ✓ **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();
        }
}
```

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.

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

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.

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

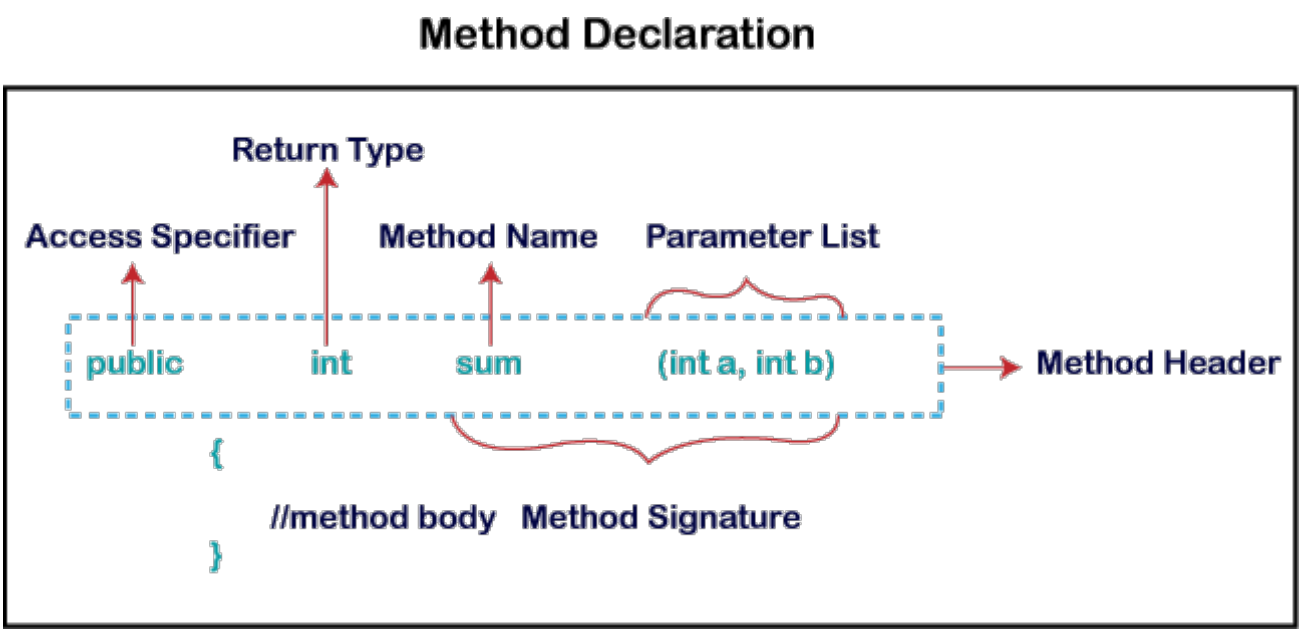
Method in Java

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**.

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 a Method
While defining a method, remember that the method name must be a **verb** and start witha **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.

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");
}
```

"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.

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.");
}
}
```

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;
}
}
```

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 knownas **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

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

Example of abstract method

Demo.java

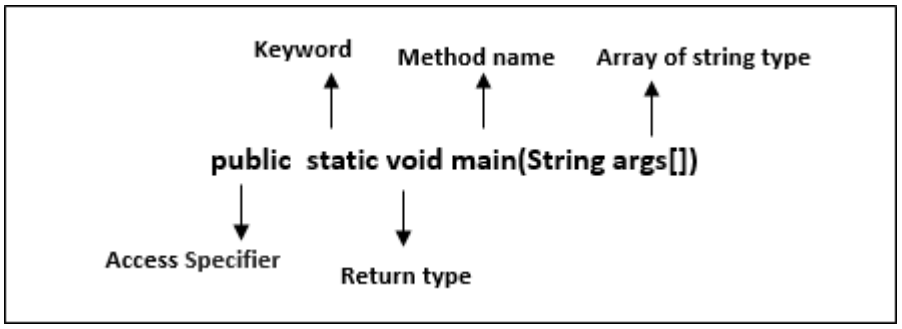
```
abstract class Demo //abstract class
{
//abstract method declaration
abstract void display();
}
public class MyClass extends Demo
{
//method impelmentation
void display()
{
System.out.println("Abstract method?");
}
public static void main(String args[])
{
//creating object of abstract class
Demo obj = new MyClass();
//invoking abstract method
obj.display();
}
}
```

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();**

Java main() method

The main() is the starting point for JVM to start execution of a Java program. Without the main() method, JVM will not execute the program. The syntax of the main() method is:



public: It is an access specifier. We should use a public keyword before the main() method so that JVM can identify the execution point of the program. If we use private, protected, and default before the main() method, it will not be visible to JVM.

static: You can make a method static by using the keyword static. We should call the main() method without creating an object. Static methods are the method which invokes without creating the objects, so we do not need any object to call the main() method.

void: In Java, every method has the return type. Void keyword acknowledges the compiler that main() method does not return any value.

main(): It is a default signature which is predefined in the JVM. It is called by JVM to execute a program line by line and end the execution after completion of this method. We can also overload the main() method.

String args[]: The main() method also accepts some data from the user. It accepts a group of strings, which is called a string array. It is used to hold the command line arguments in the form of string values.

```
main(String args[])
```

Here, agrs[] is the array name, and it is of String type. It means that it can store a group of string. Remember, this array can also store a group of numbers but in the form of string only. Values passed to the main() method is called arguments. These arguments are stored into args[] array, so the name args[] is generally used for it.

What happens if the main() method is written without String args[]?

The program will compile, but not run, because JVM will not recognize the main() method. Remember JVM always looks for the main() method with a string type array as a parameter.

Execution Process

First, JVM executes the static block, then it executes static methods, and then it creates the object needed by the program. Finally, it executes the instance methods. JVM executes a static block on the highest priority basis. It means JVM first goes to static block even before it looks for the main() method in the program.

Example

```
class  Demo
{
static                //static block
{
System.out.println("Static block");
}
public static void main(String args[])  //static method
{
System.out.println("Static method");
}
}
```

```
Static block
Static method
```

We observe that JVM first executes the static block, if it is present in the program. After that it searches for the main() method. If the main() method is not found, it gives error.

Example

A program that does not have the main() method gives an error at run time.

```
class DemoStaticBlock
{
Static                          //static block
{
System.out.println("Static block");
}
}
```

Output:

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

So the main() method should always be written as:

```
public static void main(String args[])
```

We can interchange public and static and write it as follows:

```
static public void main(String args[])
```

We can also use the different name for the String type array and write it as:

```
static public void main(String[] x)
```

Different ways of writing main() method are:

```
static public void main(String []x)
static public void main(String...args)
```

String...args: It allows the method to accept zero or multiple arguments. There should be exactly three dots between String and array; otherwise, it gives an error.

Example

A program that has no main() method, but compile and runs successfully.

```
abstract class DemoNoMain extends javafx.application.Application
{
static      //static block
{
System.out.println("Java");
System.exit(0);
}
}
```

Output:

```
Java
```