UNIT-1

Java is a class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is intended to let application developers write once, and run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java was first released in 1995 and is widely used for developing applications for desktop, web, and mobile devices. Java is known for its simplicity, robustness, and security features, making it a popular choice for enterprise-level applications.

JAVA was developed by James Gosling at **Sun Microsystems** Inc in the year **1995** and later acquired by Oracle Corporation. It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs. Java is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible. A general-purpose programming language made for developers to *write once run anywhere* that is compiled Java code can run on all platforms that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine. The syntax of Java is similar to c/c++.

History: Java's history is very interesting. It is a programming language created in 1991. James Gosling, Mike Sheridan, and Patrick Naughton, a team of Sun engineers known as the **Green team** initiated the Java language in 1991. **Sun Microsystems** released its first public implementation in 1996 as **Java 1.0**. It provides no-cost -run-times on popular platforms. Java1.0 compiler was re-written in Java by Arthur Van Hoff to strictly comply with its specifications. With the arrival of Java 2, new versions had multiple configurations built for different types of platforms.

In 1997, Sun Microsystems approached the ISO standards body and later formalized Java, but it soon withdrew from the process. At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System.

On November 13, 2006, Sun released much of its Java virtual machine as free, open-source software. On May 8, 2007, Sun finished the process, making all of its JVM's core code available under open-source distribution terms.

The principles for creating java were simple, robust, secured, high-performance, portable, multi-threaded, interpreted, dynamic, etc. In 1995 Java was developed by **James Gosling**, who is known as the Father of Java. Currently, Java is used in mobile devices, internet programming, games, e-business, etc.

Implementation of a Java application program involves a following step. They include:

1. Creating the program
2. Compiling the program
3. Running the program

Remember that, before we begin creating the program, the Java Development Kit (JDK) must be properly installed on our system and also path will be set.

• Compiling the program

To compile the program, we must run the Java compiler (javac), with the name of the source file on "command prompt" like as follows

If everything is OK, the "javac" compiler creates a file called "Test.class" containing byte code of the program.

• Running the program

We need to use the Java Interpreter to run a program.

Java programming language is named JAVA. Why?

After the name OAK, the team decided to give it a new name to it and the suggested words were Silk, Jolt, revolutionary, DNA, dynamic, etc. These all names were easy to spell and fun to say, but they all wanted the name to reflect the essence of technology. In accordance with James Gosling, Java the among the top names along with Silk, and since java was a unique name so most of them preferred it.

Java is the name of an **island** in Indonesia where the first coffee(named java coffee) was produced. And this name was chosen by James Gosling while having coffee near his office. Note that Java is just a name, not an acronym.

Java Terminology

Before learning Java, one must be familiar with these common terms of Java.

1. Java Virtual Machine(JVM): This is generally referred to as JVM. There are three execution phases of a program. They are written, compile and run the program.

- Writing a program is done by a java programmer like you and me.
- The compilation is done by the **JAVAC** compiler which is a primary Java compiler included in the Java development kit (JDK). It takes the Java program as input and generates bytecode as output.
- In the Running phase of a program, **JVM** executes the bytecode generated by the compiler.

Now, we understood that the function of Java Virtual Machine is to execute the bytecode produced by the compiler. Every Operating System has a different JVM but the output they produce after the execution of bytecode is the same across all the operating systems. This is why Java is known as a **platform-independent language.**

2. Bytecode in the **Development Process:** As discussed, the Javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. It is saved as **.class** file by the compiler. To view the bytecode, a disassembler like javap can be used.

3. Java Development Kit(JDK): While we were using the term JDK when we learn about bytecode and JVM. So, as the name suggests, it is a complete Java development kit that includes everything including compiler, Java Runtime Environment (JRE), java debuggers, java docs, etc. For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.

4. Java Runtime Environment (JRE): JDK includes JRE. JRE installation on our computers allows the java program to run, however, we cannot compile it. JRE includes a browser, JVM, applet support, and plugins. For running the java program, a computer needs JRE.

5. Garbage Collector: In Java, programmers can't delete the objects. To delete or recollect that memory JVM has a program called Garbage Collector. Garbage Collectors can recollect the objects that are not referenced. So Java makes the life of a programmer easy by handling memory management. However, programmers should be careful about their code whether they are using objects that have been used for a long time. Because Garbage cannot recover the memory of objects being referenced.

6. ClassPath: The classpath is the file path where the java runtime and Java compiler look for **.class** files to load. By default, JDK provides many libraries. If you want to include external libraries they should be added to the classpath.

Primary/Main Features of Java

1. Platform Independent: Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of the bytecode. That is why we call java a platform-independent language.

2. Object-Oriented Programming Language: Organizing the program in the terms of a collection of objects is a way of object-oriented programming, each of which represents an instance of the class.
The four main concepts of Object-Oriented programming are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

3. Simple: Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, and Explicit memory allocation.
4. Robust: Java language is robust which means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.
5. Secure: In java, we don't have pointers, so we cannot access out-of-bound arrays i.e it shows **ArrayIndexOutOfBound Exception** if we try to do so. That's why several security flaws like stack corruption or buffer overflow are impossible to exploit in Java. Also, java programs run in an environment that is independent of the os(operating system) environment which makes java programs more secure.
6. Distributed: We can create distributed applications using the java programming language. Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.
7. Multithreading: Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU.
8. Portable: As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.
9. High Performance: Java architecture is defined in such a way that it reduces overhead during the runtime and at some times java uses Just In Time (JIT) compiler where the compiler compiles code on-demand basics where it only compiles those methods that are called making applications to execute faster.
10. Dynamic flexibility: Java being completely object-oriented gives us the flexibility to add classes, new methods to existing classes, and even create new classes through sub-classes. Java even supports functions written in other languages such as C, C++ which are referred to as native methods.
11. Sandbox Execution: Java programs run in a separate space that allows user to execute their applications without affecting the underlying system with help of a bytecode verifier. Bytecode verifier also provides additional security as its role is to check the code for any violation of access.
12. Write Once Run Anywhere: As discussed above java application generates a '.class' file that corresponds to our applications(program) but contains code in binary format. It provides ease t architecture-neutral ease as bytecode is not dependent on any machine architecture. It is the primary reason java is used in the enterprising IT industry globally worldwide.

13. Power of compilation and interpretation: Most languages are designed with the purpose of either they are compiled language or they are interpreted language. But java integrates arising enormous power as Java compiler compiles the source code to bytecode and JVM executes this bytecode to machine OS-dependent executable code.

class : class keyword is used to declare classes in Java
public : It is an access specifier. Public means this function is visible to all.
static : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The main() method here is called by JVM, without creating any object for class.
void : It is the return type, meaning this function will not return anything.
main : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.
String[] args : This is used to signify that the user may opt to enter parameters to the Java Program at command line. We can use both String[] args or String args[]. Java compiler would accept both forms.

System.out.println : This is used to print anything on the console like “printf” in C language.

JAVA SOURCE FILE STRUCTURE:

Java source file structure describes that the Java source code file must follow a schema or structure. In this article, we will see some of the important guidelines that a Java program must follow.

A Java program has the following structure:

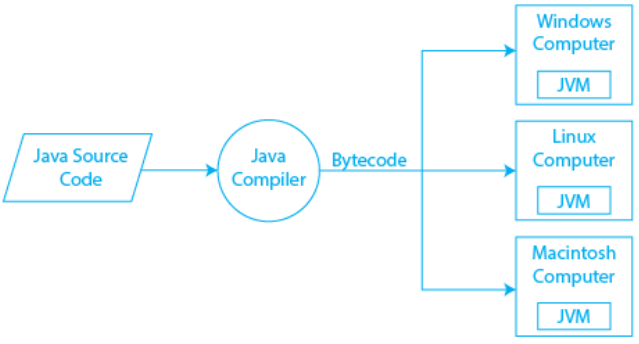
1. **package statements:** A package in Java is a mechanism to encapsulate a group of classes, sub-packages, and interfaces.
2. **import statements:** The import statement is used to import a package, class, or interface.
3. **class definition:** A class is a user-defined blueprint or prototype from which objects are created, and it is a passive entity.

Compilation

First, the source ‘.java’ file is passed through the compiler, which then encodes the source code into a machine-independent encoding, known as Bytecode. The content of each class contained in the source file is stored in a separate ‘.class’ file. While converting the source code into the bytecode, the compiler follows the following steps:

What is the Compilation Process in Java?

The source code of a Java code is compiled into an intermediate binary code known as the Bytecode during the Java compilation process. The machine cannot directly execute this Bytecode. A virtual machine known as the Java Virtual Machine, or JVM, understands it. JVM includes a Java interpreter that converts Bytecode to target computer machine code. JVM is platform-specific, which means that each platform has its own JVM. However, once the proper JVM is installed on the machine, any Java Bytecode code can be run. This is shown in the diagram below:



Java application development (implementation) and can be broken down into the following phases:

Compilation:
A special application called a compiler, executes our Java program on what is known as a virtual Java machine (JVM).

The compiler transforms source code into so-called JVM bytecode, or machine code read by JVM. In addition, the compiler should check the code for lexical and semantic issues and optimise it.

Java Classes
A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:
   - Data member
   - Method
   - Constructor
   - Nested Class
   - Interface

Class Declaration in Java

access_modifier **class** <*class_name*>
{
  data member;
  method;
  constructor;
  nested class;
  interface;
}

Example of Java Class

```
// Java Program for class example

 class Student {

   // data member (also instance variable)

   int id;

   // data member (also instance variable)

   String name;

    public static void main(String args[])

   {       // creating an object of Student
```

```
        Student s1 = new Student();

        System.out.println(s1.id);

        System.out.println(s1.name);

    }

}
```

What are 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 the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

How Java Constructors are Different From Java Methods?

- Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Note: *Default constructor provides the default values to the object like 0, null, etc. depending on the type.*

2. Parameterized Constructor in Java

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

JAVA METHOD

The **method in Java** or Methods of Java is a collection of statements that perform some specific task and return the result to the caller. A Java method can perform some specific task without returning anything. Java Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class that is different from languages like C, C++, and Python.
1. A method is like a function i.e. used to expose the behavior of an object.
2. It is a set of codes that perform a particular task.

Syntax of Method

<access_modifier> <return_type> <method_name>(list_of_parameters)

```
{
   //body
}
```

Advantage of Method

- Code Reusability
- Code Optimization

Note: *Methods are time savers and help us to reuse the code without retyping the code.*

Method Declaration

In general, method declarations have 6 components:

1. Modifier: It defines the **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 types of access specifiers.

- **public:** It is accessible in all classes in your application.
- **protected:** It is accessible within the class in which it is defined and in its subclass/es
- **private:** It is accessible only within the class in which it is defined.
- **default:** It is declared/defined without using any modifier. It is accessible within the same class and package within which its class is defined.

Note: *It is* ***Optional*** *in syntax.*

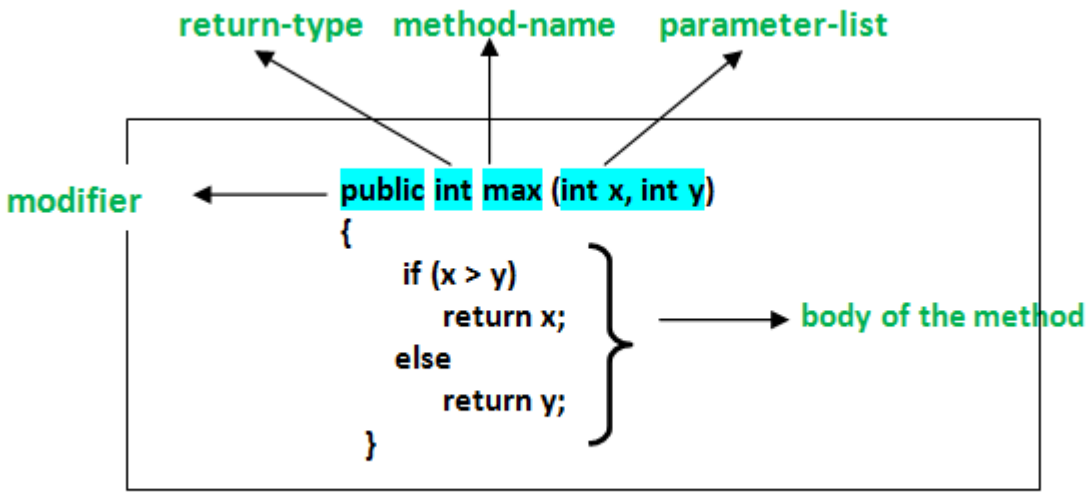
2. The return type: The data type of the value returned by the method or void if does not return a value. It is **Mandatory** in syntax.

3. Method Name: the rules for field names apply to method names as well, but the convention is a little different. It is **Mandatory** in syntax.

4. Parameter list: Comma-separated list of the input parameters is defined, preceded by their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses (). It is **Optional** in syntax.

5. Exception list: The exceptions you expect by the method can throw, you can specify these exception(s). It is **Optional** in syntax.

6. Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations. It is **Optional** in syntax.



Types of Methods in Java

There are two types of methods in Java:

1. 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.

2. 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.

ACCESS MODIFIER

1. Default Access Modifier

When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible **only within the same package**

2. Private Access Modifier

The private access modifier is specified using the keyword **private**. The methods or data members declared as private are accessible only **within the class** in which they are declared.

- Any other **class of** the **same package will not be able to access** these members.
- Top-level classes or interfaces can not be declared as private because
    - private means “only visible within the enclosing class”.
    - protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

3. Protected Access Modifier

The protected access modifier is specified using the keyword **protected**.

The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

2. Public Access modifier

The public access modifier is specified using the keyword **public**.

- The public access modifier has the **widest scope** among all other access modifiers.
- Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

The **static keyword** in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class. The users can apply static keywords with variables, methods, blocks, and nested classes. The static keyword belongs to the class than an instance of the class. The static keyword is used for a constant variable or a method that is the same for every instance of a class.

The *static* keyword is a non-access modifier in Java that is applicable for the following:

1. Blocks
2. Variables
3. Methods
4. Classes

Characteristics of static keyword:

- **Shared memory allocation**: Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.
- **Accessible without object instantiation:** Static members can be accessed without the need to create an instance of the class. This makes them useful for providing utility functions and constants that can be used across the entire program.
- **Associated with class, not objects:** Static members are associated with the class, not with individual objects. This means that changes to a static member are reflected in all instances of the class, and that you can access static members using the class name rather than an object reference.
- **Cannot access non-static members:** Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class.
- **Can be overloaded, but not overridden**: Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. For example, in the below java program, we are accessing static method *m1()* without creating any object of the *Test* class.

final keyword is used in different contexts. First of all, *final* is a non-access modifier applicable only to a variable, a method, or a class. The following are different contexts where final is used.

Characteristics of final keyword in java:

In Java, the final keyword is used to indicate that a variable, method, or class cannot be modified or extended. Here are some of its characteristics:

- **Final variables:** When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.

- **Final methods**: When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that are part of a class's public API and should not be modified by subclasses.
- **Final classes:** When a class is declared as final, it cannot be extended by a subclass. This is useful for classes that are intended to be used as is and should not be modified or extended.
- **Initialization:** Final variables must be initialized either at the time of declaration or in the constructor of the class. This ensures that the value of the variable is set and cannot be changed.
- **Performance:** The use of final can sometimes improve performance, as the compiler can optimize the code more effectively when it knows that a variable or method cannot be changed.
- **Security:** final can help improve security by preventing malicious code from modifying sensitive data or behavior.

Overall, the final keyword is a useful tool for improving code quality and ensuring that certain aspects of a program cannot be modified or extended. By declaring variables, methods, and classes as final, developers can write more secure, robust, and maintainable code.

Java Comments

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Single-line Comments

Single-line comments start with two forward slashes (//).

Data Type	Size	Description
Byte	1 byte	Stores whole numbers from -128 to 127
Short	2 bytes	Stores whole numbers from -32,768 to 32,767
Int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
Long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,
Float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digi
Double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
Boolean	1 bit	Stores true or false values
Char	2 bytes	Stores a single character/letter or ASCII values

Any text between // and the end of the line is ignored by Java (will not be executed).

This example uses a single-line comment before a line of code:

```
// This is a comment
System.out.println("Hello World");
```

Java Multi-line Comments

Multi-line comments start with /* and ends with */.

Any text between /* and */ will be ignored by Java.

This example uses a multi-line comment (a comment block) to explain the code:

```
/* The code below will print the words Hello World
to the screen, and it is amazing */
System.out.println("Hello World");
```

Data types are divided into two groups:

- Primitive data types - includes byte, short, int, long, float, double, boolean and char
- Non-primitive data types - such as String, Arrays and Classes (you will learn more about these in a later chapter

Variables in Java

Java variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- Variables in Java are only a name given to a memory location. All the operations done on the variable affect that memory location.
- In Java, all variables must be declared before use.

How to Declare Variables in Java?

We can declare variables in Java as pictorially depicted below as a visual aid.

1. **datatype**: Type of data that can be stored in this variable.
2. **data_name:** Name was given to the variable.

In this way, a name can only be given to a memory location. It can be assigned values in two ways:

- Variable Initialization
- Assigning value by taking input

How to Initialize Variables in Java?

It can be perceived with the help of 3 components that are as follows:

- **datatype**: Type of data that can be stored in this variable.
- **variable_name**: Name given to the variable.
- **value**: It is the initial value stored in the variable.

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators

ARITHMETIC OPERATOR

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

JAVA ASSIGNMENT OPERATORS

Assignment operators are used to assign values to variables.

In the example below, we use the **assignment** operator (=) to assign the value **10** to a variable called **x**:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

```
int x = 10;
```

JAVA COMPARISON OPERATORS

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

JAVA LOGICAL OPERATORS

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

JAVA CONTROL STATEMENTS | CONTROL FLOW IN JAVA

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
    - if statements
    - switch statement
2. Loop statements
    - do while loop

- while loop
- for loop
- for-each loop

3. Jump statements
    - break statement
    - continue statement

UNDERSTANDING CLASSES AND OBJECTS IN JAVA

The term *Object-Oriented* explains the concept of organizing the software as a combination of different types of objects that incorporates both data and behavior. Hence, Object-oriented programming(OOPs) is a programming model, that simplifies software development and maintenance by providing some rules. Programs are organised around objects rather than action and logic. It increases the flexibility and maintainability of the program. Understanding the working of the program becomes easier, as OOPs brings data and its behavior(methods) into a single(objects) location.

Classes: A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Classes are required in OOPs because:

- It provides template for creating objects, which can bind code into data.
- It has definitions of methods and data.
- It supports inheritance property of Object Oriented Programming and hence can maintain class hierarchy.
- It helps in maintaining the access specifications of member variables.

Objects: It is the basic unit of Object Oriented Programming and it represents the real life entities. Real-life entities share two characteristics: they all have attributes and behavior.

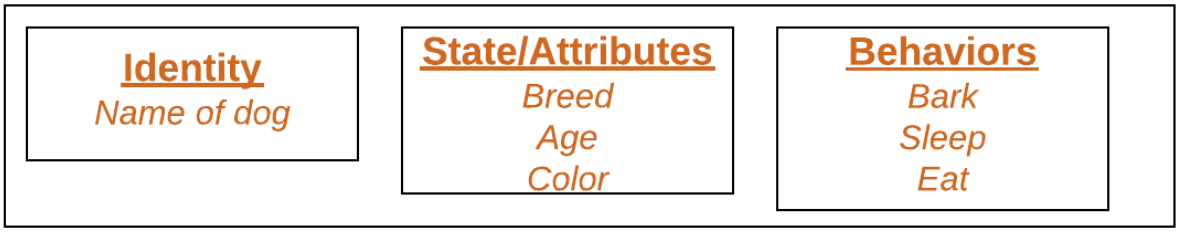
An object consists of:

- **State:** It is represented by *attributes* of an object. It also shows properties of an object.
- **Behavior:** It is represented by *methods* of an object. It shows response of an object with other objects.
- **Identity:** It gives a unique name to an object. It also grants permission to one object to interact with other objects.

Objects are required in OOPs because they can be created to call a non-static function which are not present inside the Main Method but present inside the Class and also provide the name to the space which is being used to store the data.

Example : For addition of two numbers, it is required to store the two numbers separately from each other, so that they can be picked and the desired operations can be performed on them. Hence creating two different objects to store the two numbers will be an ideal solution for this scenario.

Example to demonstrate the use of Objects and classes in OOPs

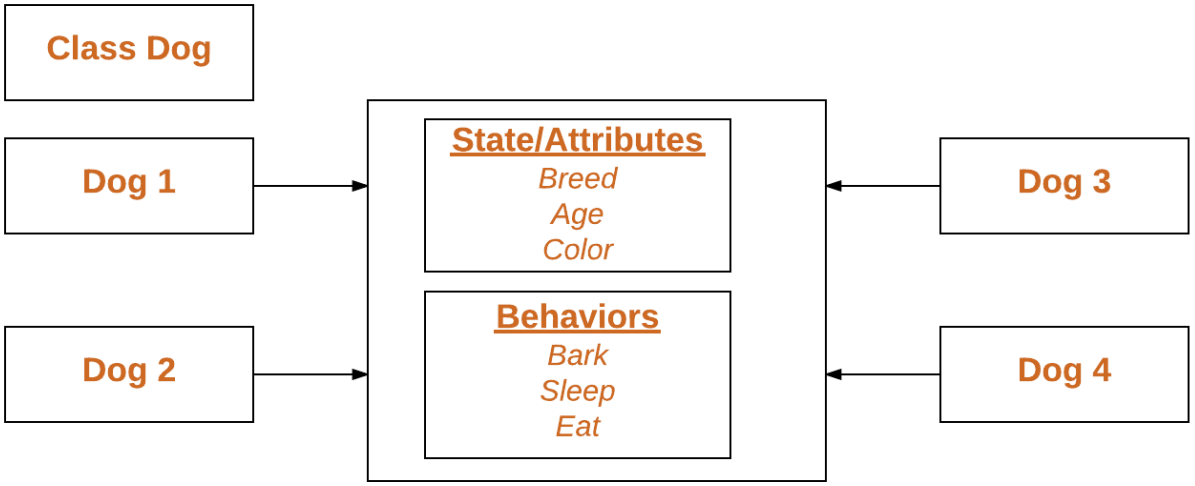


OBJECTS

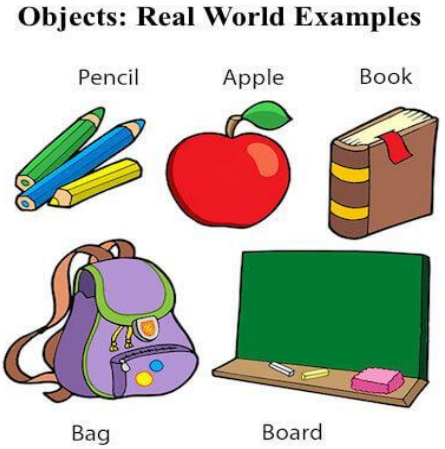
Objects relate to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.



What is an object in Java



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

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

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

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

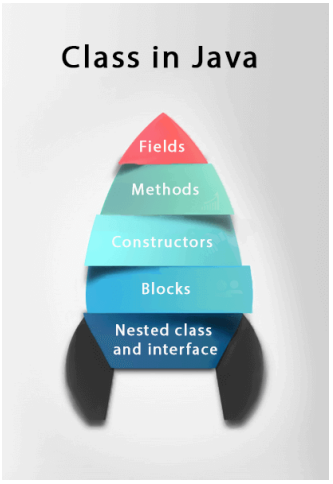
Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- 

  Nested class and interface

Syntax to declare a class:

```
class <class_name>{
    field;
    method;
}
```

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

NEW KEYWORD IN JAVA

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

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

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1 OBJECT AND CLASS EXAMPLE: INITIALIZATION THROUGH REFERENCE

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent2{
6.  public static void main(String args[]){
7.   Student s1=new Student();
8.   s1.id=101;
9.   s1.name="Sonoo";
10.  System.out.println(s1.id+" "+s1.name);//printing members with a white space
11. }
12.}
```

2) Object and Class Example: Initialization through method

```
1. class Student{
2.  int rollno;
3.  String name;
4.  void insertRecord(int r, String n){
```

```
5.    rollno=r;
6.    name=n;
7.   }
8.   void displayInformation(){System.out.println(rollno+" "+name);}
9.  }
10. class TestStudent4{
11.  public static void main(String args[]){
12.   Student s1=new Student();
13.   Student s2=new Student();
14.   s1.insertRecord(111,"Karan");
15.   s2.insertRecord(222,"Aryan");
16.   s1.displayInformation();
17.   s2.displayInformation();
18. }
19.}
```

3) Object and Class Example: Initialization through a constructor

```
public class Dog {
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;
       Dog(String name, String breed,int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }
public static void main(String... args)
{
Dog d=new Dog("puppy","reg",10,"red");
System.out.println(d.name+" "+ d.breed+" " + d.color);   } }
```

Inheritance in Java

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

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

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

Why use inheritance in java

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

Terms used in Inheritance

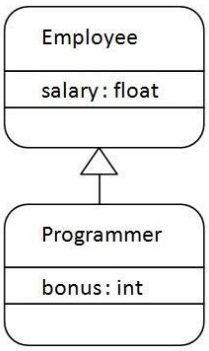
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent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

The syntax of Java Inheritance

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

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

Java Inheritance Example



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

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
```

```
10.}
11.}
```

Multilevel Inheritance Example

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

File: TestInheritance2.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

Hierarchical Inheritance Example

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

File: TestInheritance3.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
8. void meow(){System.out.println("meowing...");}
9. }
10. class TestInheritance3{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}
```

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

Method Overloading in Java

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

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

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

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

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

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

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

---

1) Method Overloading: changing no. of arguments

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

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

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
5. class TestOverloading1{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}
```

2) Method Overloading: changing data type of arguments

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

```
1. class Adder{
2. static int add(int a, int b){return a+b;}
3. static double add(double a, double b){return a+b;}
4. }
5. class TestOverloading2{
```

```
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}
```

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

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

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static double add(int a,int b){return a+b;}
4. }
5. class TestOverloading3{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));//ambiguity
8. }
9. }
```

NOTE:-Compile Time Error: method add(int,int) is already defined in class Adder
POINT:-System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Can we overload java main() method?

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

```
1. class TestOverloading4{
2. public static void main(String[] args){System.out.println("main with String[]");}
3. public static void main(String args){System.out.println("main with String");}
4. public static void main(){System.out.println("main without args");}
5. }
```

Method Overriding in Java

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

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

Usage of Java Method Overriding

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

Rules for Java Method Overriding

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

Understanding the problem without method overriding

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

```java
//Java Program to demonstrate why we need method overriding
//Here, we are calling the method of parent class with child
//class object.
//Creating a parent class
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Example of method overriding

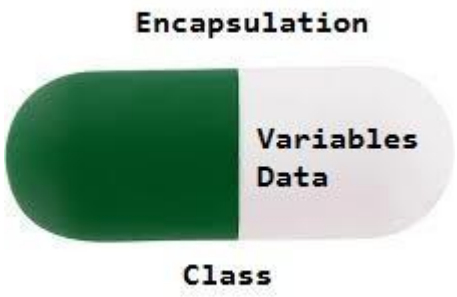
In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

```java
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

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

Encapsulation in Java is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Java Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.



Implementation of Java Encapsulation

Below is the example with Java Encapsulation:

```
// Java Encapsulation


// Person Class

class Person {

    // Encapsulating the name and age

    // only approachable and used using

    // methods defined

    private String name;

    private int age;


    public String getName() { return name; }


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


    public int getAge() { return age; }


    public void setAge(int age) { this.age = age; }
```

```
}

 // Driver Class

public class Main {

    // main function

    public static void main(String[] args)

    {

        // person object created

        Person person = new Person();

        person.setName("John");

        person.setAge(30);



        // Using methods to get the values from the

        // variables

        System.out.println("Name: " + person.getName());

        System.out.println("Age: " + person.getAge());

    }

}
```

- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
- It is more defined with the setter and getter method.

Advantages of Encapsulation

- **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know 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 read-only or write-only depending on our requirements. If we wish to make the variables 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 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 is easy to change with new requirements.

- **Testing code is easy:** Encapsulated code is easy to test for unit testing.
- **Freedom to programmer in implementing the details of the system:** This is one of the major advantage of encapsulation that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

Disadvantages of Encapsulation in Java

- Can lead to increased complexity, especi
- ally if not used properly.
- Can make it more difficult to understand how the system works.
- May limit the flexibility of the implementation.

Polymorphism

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism are:

1. Compile-time polymorphism (Method overloading)
2. Run-time polymorphism (Method Overriding)

ABSTRACT

In Java, abstract is a non-access modifier in java applicable for classes, and methods but **not** variables. It is used to achieve abstraction which is one of the pillars of Object Oriented Programming (OOP). Following are different contexts where *abstract* can be used in Java.

Characteristics of Java abstract Keyword

In Java, the abstract keyword is used to define abstract classes and methods. Here are some of its key characteristics:

- **Abstract classes cannot be instantiated**: An abstract class is a class that cannot be instantiated directly. Instead, it is meant to be extended by other classes, which can provide concrete implementations of its abstract methods.
- **Abstract methods do not have a body**: An abstract method is a method that does not have an implementation. It is declared using the abstract keyword and ends with a semicolon instead of a method body. Subclasses of an abstract class must provide a concrete implementation of all abstract methods defined in the parent class.
- **Abstract classes can have both abstract and concrete methods:** Abstract classes can contain both abstract and concrete methods. Concrete methods are implemented in the abstract class itself and can be used by both the abstract class and its subclasses.
- **Abstract classes can have constructors**: Abstract classes can have constructors, which are used to initialize instance variables and perform other initialization tasks. However, because abstract classes cannot be instantiated directly, their constructors are typically called constructors in concrete subclasses.
- **Abstract classes can contain instance variables**: Abstract classes can contain instance variables, which can be used by both the abstract class and its subclasses. Subclasses can access these variables directly, just like any other instance variables.
- **Abstract classes can implement interfaces:** Abstract classes can implement interfaces, which define a set of methods that must be implemented by any class that implements the interface. In this case, the abstract class must provide concrete implementations of all methods defined in the interface.

Overall, the abstract keyword is a powerful tool for defining abstract classes and methods in Java. By declaring a class or method as abstract, developers can provide a structure for subclassing and ensure that certain methods are implemented in a consistent way across all subclasses.

Abstract Methods in Java

Sometimes, we require just method declaration in super-classes. This can be achieved by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the super-class. Thus, a subclass must override them to provide a method definition. To declare an abstract method, use this general form:

Example of Abstract class that has an abstract method

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

```
1. abstract class Bike{
2.   abstract void run();
3. }
4. class Honda4 extends Bike{
5. void run(){System.out.println("running safely");}
6. public static void main(String args[]){
7.  Honda4 obj = new Honda4();
8.  obj.run();
9. }
10. }
```

Interface in Java

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

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

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

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

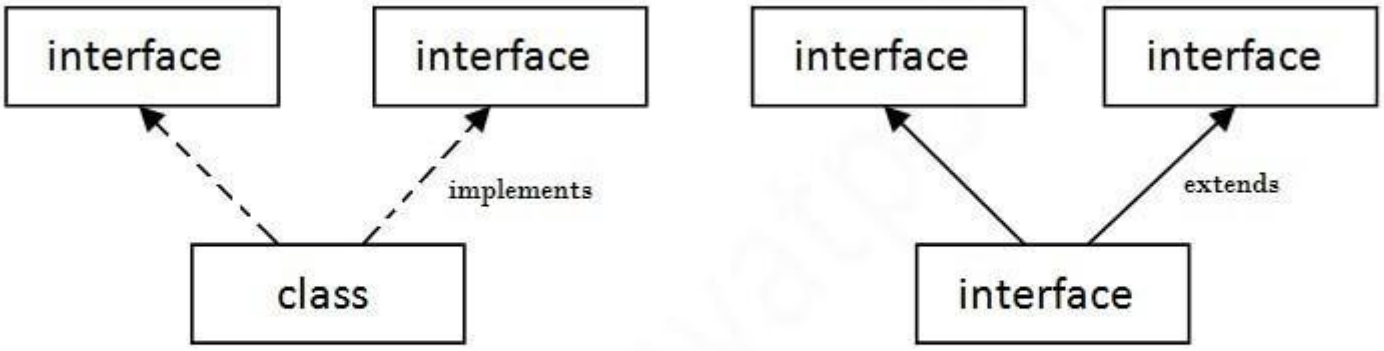
Java Interface Example

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

```
1. interface printable{
2. void print();
3. }
4. class A6 implements printable{
5. public void print(){System.out.println("Hello");}
6.
7. public static void main(String args[]){
8. A6 obj = new A6();
9. obj.print();
10.  }
11. }
```

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

```
1.  interface Printable{
2.  void print();
3.  }
4.  interface Showable{
5.  void show();
6.  }
7.  class A7 implements Printable,Showable{
8.  public void print(){System.out.println("Hello");}
9.  public void show(){System.out.println("Welcome");}
10.
11. public static void main(String args[]){
12. A7 obj = new A7();
13. obj.print();
14. obj.show();
15.  }
16. }
```

Difference between abstract class and interface

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

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

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.

4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

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

Example of abstract class and interface in Java

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

```
1.                          //Creating interface that has 4 methods
2.  interface A{
3.  void a();//bydefault, public and abstract
4.  void b();
5.  void c();
6.  void d();
7.  }
8.
9.  //Creating abstract class that provides the implementation of one method of A interface
10. abstract class B implements A{
11. public void c(){System.out.println("I am C");}
12. }
13.
14. //Creating subclass of abstract class, now we need to provide the implementation of rest of the metho
    ds
15. class M extends B{
16. public void a(){System.out.println("I am a");}
17. public void b(){System.out.println("I am b");}
18. public void d(){System.out.println("I am d");}
19. }
20.
21. //Creating a test class that calls the methods of A interface
22. class Test5{
23. public static void main(String args[]){
24. A a=new M();
```

```
25. a.a();
26. a.b();
27. a.c();
28. a.d();
29. }}
```

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.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Simple example of java package

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

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.    System.out.println("Welcome to package");
6.   }
7. }
```

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

```
Output:Welcome to package
```

How to access package from another package?

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

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

*1) Using packagename.**

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

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

Example of package that import the packagename.*

```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.    public void msg(){System.out.println("Hello");}
5.  }
```

```
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.
5.  class B{
6.    public static void main(String args[]){
7.      A obj = new A();
8.      obj.msg();
9.    }
10. }
```

```
Output:Hello
```

---

2) Using packagename.classname

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

Example of package by import package.classname

```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.    public void msg(){System.out.println("Hello");}
5.  }
```

```
1.  //save by B.java
2.  package mypack;
3.  import pack.A;
4.
5.  class B{
6.    public static void main(String args[]){
7.      A obj = new A();
8.      obj.msg();
9.    }
10. }
```

```
Output:Hello
```

---

3) Using fully qualified name

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

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

Example of package by import fully qualified name

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

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

```
Output:Hello
```

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

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

IMPORT AND STATIC IMPORT NAMING CONVENTION FOR PACKAGE

With the help of static import, we can access the static members of a class directly without class name or any object. For Example: we always use sqrt() method of Math class by using Math class i.e. **Math.sqrt()**, but by using static import we can access sqrt() method directly. According to SUN microSystem, it will improve the code readability and enhance coding. But according to the programming experts, it will lead to confusion and not good for programming. If there is no specific requirement then we should **not** go for static import.
Advantage of static import:
If user wants to access any static member of class then less coding is required.

Disadvantage of static import:
Static import makes the program unreadable and unmaintainable if you are reusing this feature.

- JAVA

```java
// Java Program to illustrate

// calling of predefined methods

// without static import

class Geeks {

    public static void main(String[] args)

    {
```

```
        System.out.println(Math.sqrt(4));

        System.out.println(Math.pow(2, 2));

        System.out.println(Math.abs(6.3));

    }

}
```

Output:

```
2.0

4.0

6.3
```

- JAVA

```
// Java Program to illustrate

// calling of predefined methods

// with static import

import static java.lang.Math.*;

class Test2 {

    public static void main(String[] args)

    {

        System.out.println(sqrt(4));

        System.out.println(pow(2, 2));

        System.out.println(abs(6.3));

    }

}
```

Output:

```
2.0

4.0

6.3
```

- JAVA

```
// Java to illustrate calling of static member of

// System class without Class name

import static java.lang.Math.*;

import static java.lang.System.*;

class Geeks {

    public static void main(String[] args)

    {

        // We are calling static member of System class

        // directly without System class name

        out.println(sqrt(4));

        out.println(pow(2, 2));

        out.println(abs(6.3));

    }

}
```

Output:

```
2.0

4.0

6.3
```

NOTE : System is a class present in java.lang package and out is a static variable present in System class. By the help of static import we are calling it without class name.

CLASSPATH Setting for Package

CLASSPATH: CLASSPATH is an environment variable which is used by Application Class Loader to locate and load the .class files. The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform. Include all the directories which contain .class files and JAR files when setting the CLASSPATH.

You need to set the CLASSPATH if:

- You need to load a class that is not present in the current directory or any sub-directories.
- You need to load a class that is not in a location specified by the extensions mechanism.

The CLASSPATH depends on what you are setting the CLASSPATH. The CLASSPATH has a directory name or file name at the end. The following points describe what should be the end of the CLASSPATH.

- If a JAR or zip, the file contains class files, the CLASSPATH end with the name of the zip or JAR file.
- If class files placed in an unnamed package, the CLASSPATH ends with the directory that contains the class files.
- If class files placed in a named package, the CLASSPATH ends with the directory that contains the root package in the full package name, that is the first package in the full package name.

The default value of CLASSPATH is a dot (.). It means the only current directory searched. The default value of CLASSPATH overrides when you set the CLASSPATH variable or using the -classpath command (for short -cp). Put a dot (.) in the new setting if you want to include the current directory in the search path.

If CLASSPATH finds a class file which is present in the current directory, then it will load the class and use it, irrespective of the same name class presents in another directory which is also included in the CLASSPATH.

If you want to set multiple classpaths, then you need to separate each CLASSPATH by a semicolon (;).

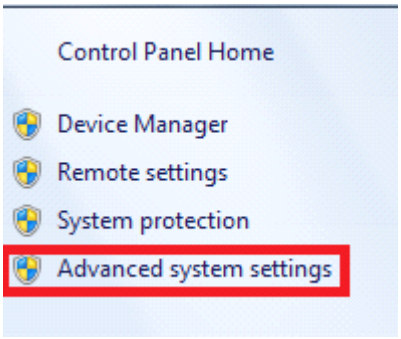
The third-party applications (MySQL and Oracle) that use the JVM can modify the CLASSPATH environment variable to include the libraries they use. The classes can be stored in directories or archives files. The classes of the Java platform are stored in rt.jar.

There are two ways to ways to set CLASSPATH: through Command Prompt or by setting Environment Variable.

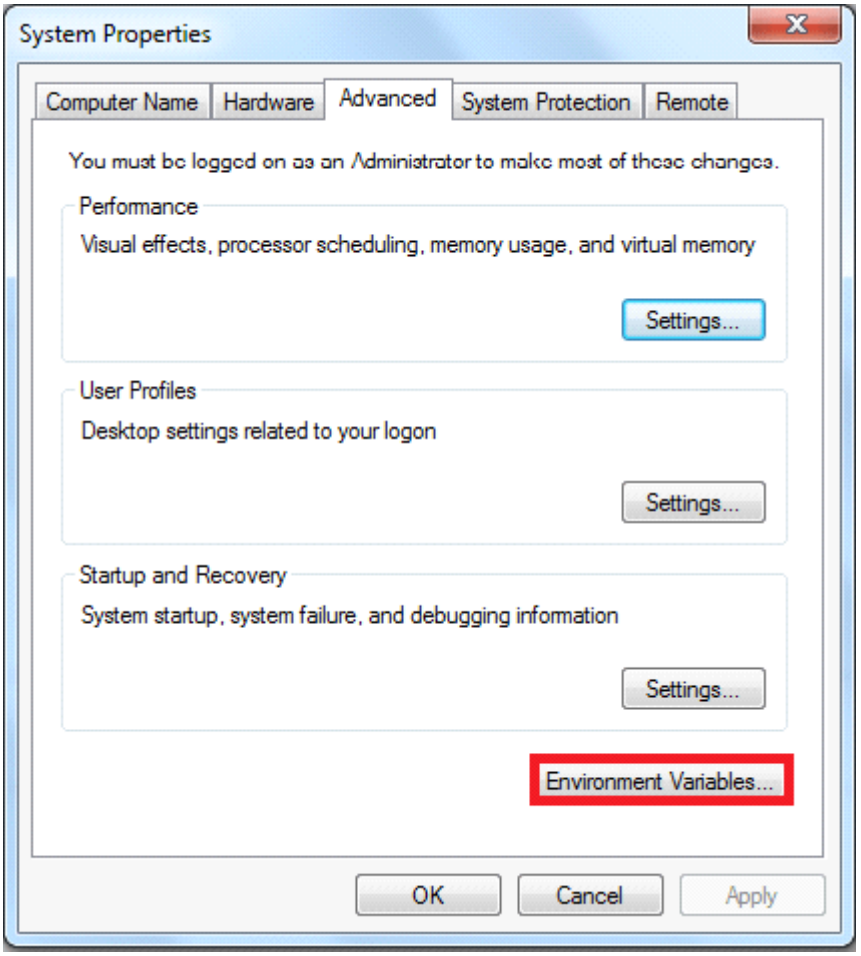
Let's see how to set CLASSPATH of MySQL database:

Step 1: Click on the Windows button and choose Control Panel. Select System.

Step 2: Click on **Advanced System Settings**.



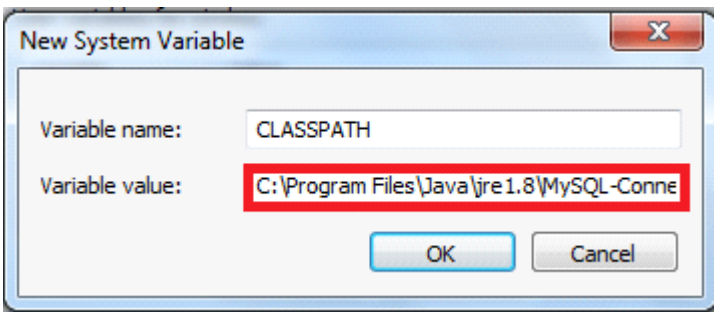
Step 3: A dialog box will open. Click on Environment Variables.



Step 4: If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end. Paste the Path of MySQL-Connector Java.jar file.

If the CLASSPATH doesn't exist in System Variables, then click on the New button and type Variable name as CLASSPATH and Variable value as *C:\Program Files\Java\jre1.8\MySQL-Connector Java.jar;.;*

Remember: Put ;.; at the end of the CLASSPATH.



JAR FILE

A JAR (Java Archive) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform.
In simple words, a JAR file is a file that contains a compressed version of .class files, audio files, image files, or directories. We can imagine a .jar file as a zipped file(.zip) that is created by using WinZip software. Even, WinZip software can be used to extract the contents of a .jar . So you can use them for tasks such as lossless data compression, archiving, decompression, and archive unpacking.
Let us see how to create a .jar file and related commands which help us to work with .jar files

1.1 Create a JAR file

In order to create a .jar file, we can use ***jar cf command*** in the following ways as

Jar -cf jarfilename.jar classfilename.class
Ex- jar –cf A.jar Dream.class

1. 2 View a JAR file

Jar tf A.jar

Note: *When we create .jar files, it automatically receives the default manifest file. There can be only one manifest file in an archive, and it always has the pathname.*

1.3 Extracting a JAR file

Jar xf A.jar

1.5 Running a JAR file

In order to run an application packaged as a JAR file (requires the Main-class manifest header), the following command can be used as listed:
Syntax:

```
C:\>java -jar A.jar
```