

II Year/III Semester

CS23312- Object Oriented Programming

Subject Name / Code:OBJECT ORIENTED PROGRAMMING / CS23312

UNIT I PARADIGMS & BASIC CONSTRUCTS

9

Object oriented programming concepts – objects – classes – methods and messages – abstraction and encapsulation – inheritance – abstract classes – polymorphism - Objects and classes in Java – defining classes – methods - access specifiers – static members – constructors – finalize method.

UNIT II EXCEPTION HANDLING & STREAMS

9

Arrays – Strings - Packages – Java-Doc comments -- Inheritance – class hierarchy –polymorphism – dynamic binding – final keyword – abstract classes-Exception handling – exception hierarchy – throwing and catching exceptions-The Object class – Reflection – interfaces – object cloning – inner classes – proxies - I/O Streams

UNIT III GENERICS & MULTI THREADING

9

Motivation for generic programming – generic classes – generic methods – generic code and virtual machine – inheritance and generics – reflection and generics - Multi-threaded programming – interrupting threads – thread states – thread properties – thread synchronization – Executors – synchronizers.

UNIT IV JAVA NETWORKING & JDBC

9

Socket programming in Java-InetAddress and URL classes-TCP and UDP protocols in JavaServerSocket and Socket Classes-Multi-threaded servers-Handling multiple client connectionsIntroduction to RMI-Creating RMI servers and clients-RMI registry-RMI and object serializationOverview of JDBC-JDBC drivers-Connecting to databases-Executing SQL queries.

UNIT V GUI PROGRAMMING

9

Introduction to Swing – Model-View-Controller design pattern – layout management – Swing Components - Introduction to JavaFX - JavaFX components.

UNIT I

INTRODUCTION TO OOP AND JAVA

UNIT I PARADIGMS & BASIC CONSTRUCTS

9

Object oriented programming concepts – objects – classes – methods and messages – abstraction and encapsulation – inheritance – abstract classes – polymorphism - Objects and classes in Java – defining classes – methods - access specifiers – static members – constructors – finalize method.

1. OVERVIEW OF OBJECT-ORIENTED PROGRAMMING (OOP)

1. Define objects and classes in Java **(2) (Nov/Dec 2019) (Nov/Dec 2018)**
2. Explain the characteristics of OOPs **(6) (Nov/Dec 2019)**
3. Explain the features and characteristics of Java **(7) (Nov/Dec 2019)**
4. Define Encapsulation in Java(2) **(APR/MAY 2021)**
5. Discuss the three OOP principles in detail (7) **(APR/MAY 2019)**

Object-Oriented Programming (OOP) is a programming language model organized around objects rather than actions and data. An object-oriented program can be characterized as data controlling access to code. Concepts of OOPS

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

OBJECT

Object means a real world entity such as pen, chair, table etc. Any entity that has state and behavior is known as an object. Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing details of each other's data or code, the only necessary thing is that the type of message accepted and type of response returned by the objects.

An object has three characteristics:

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

CLASS

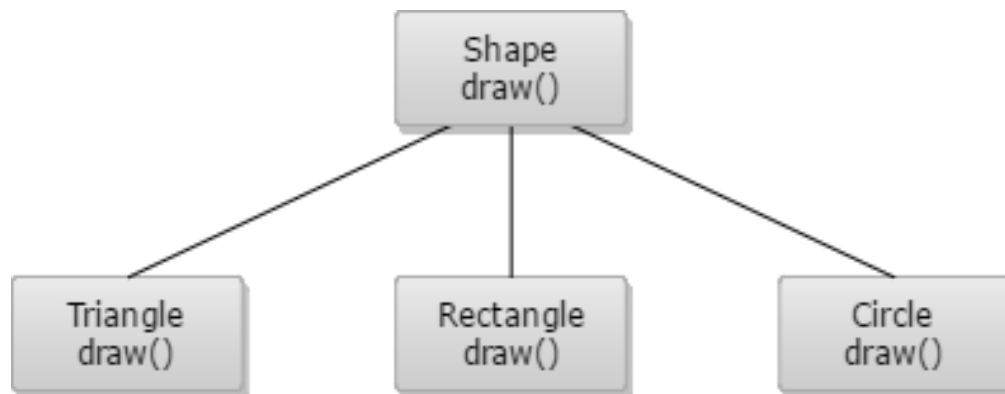
Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. A class consists of Data members and methods. The primary purpose of a class is to hold data/information. The member functions determine the behavior of the class, i.e. provide a definition for supporting various operations on data held in the form of an object. Class doesn't store any space.

INHERITANCE

Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behavior of one class to another, i.e., acquiring the properties and behavior of child class from the parent class. When one object acquires all the properties and behaviours of another object, it is known as inheritance. It provides code reusability and establishes relationships between different classes. A class which inherits the properties is known as Child Class (sub-class or derived class) whereas a class whose properties are inherited is known as Parent class (super-class or base class). Types of inheritance in java: single, multilevel and hierarchical inheritance. Multiple and hybrid inheritance is supported through interface only.

POLYMORPHISM

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.



Polymorphism is classified into two ways:

Method Overloading (Compile time Polymorphism)

Method Overloading is a feature that allows a class to have two or more methods having the same name but the arguments passed to the methods are different. Compile time polymorphism refers to a process in which a call to an overloaded method is resolved at compile time rather than at run time.

Method Overriding (Run time Polymorphism)

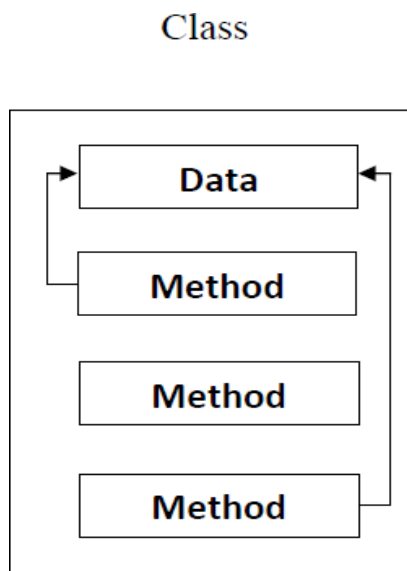
If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java. In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

ABSTRACTION

Abstraction is a process of hiding the implementation details and showing only functionality to the user. For example: phone call, we don't know the internal processing. In java, we use abstract class and interface to achieve abstraction.

ENCAPSULATION

Encapsulation in java is a process of wrapping code and data together into a single unit, for example capsule i.e. mixed of several medicines. A java class is the example of encapsulation



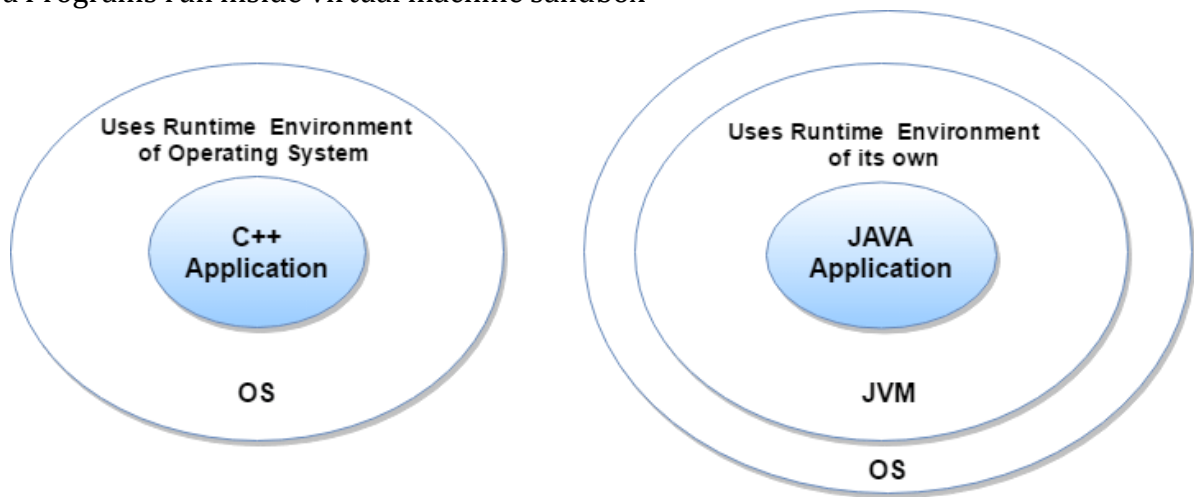
2. OVERVIEW OF JAVA

Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- No explicit pointer
- Java Programs run inside virtual machine sandbox



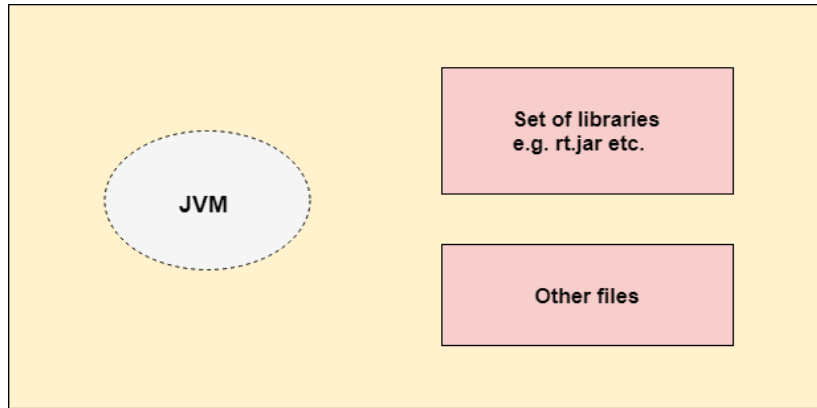
- **Class loader:** Class loader in Java is a part of the Java Runtime Environment(JRE) which is used to dynamically load Java classes into the Java Virtual Machine. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, Cryptography etc.

THE JAVA ENVIRONMENT

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing java applications. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime. Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JRE

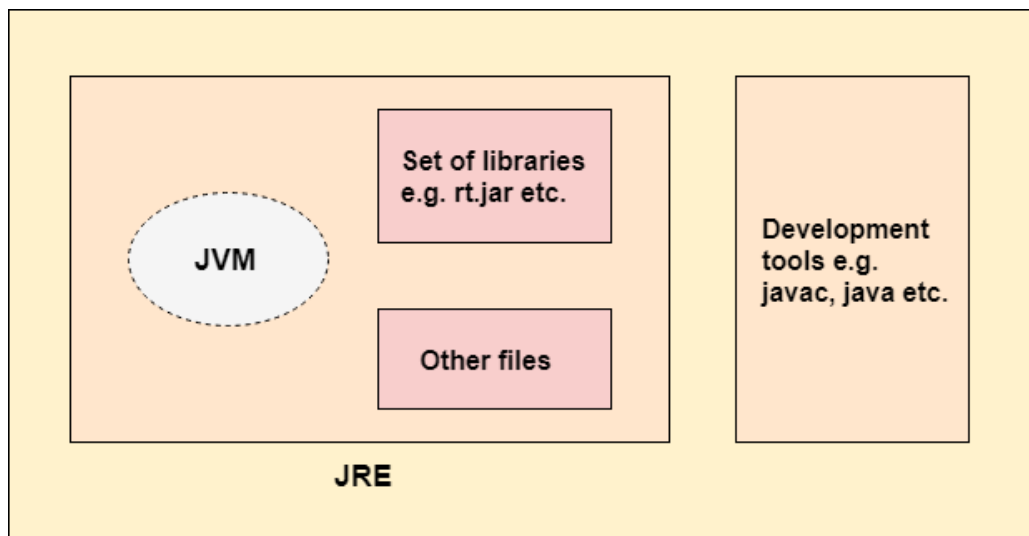
JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc. to complete the development of a Java Application.



JDK

JVM (Java Virtual Machine)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent). The JVM performs following operation:

- Loads code
- Verifies code
- Executes code

Provides runtime environment JVM provides definitions for the:

Memory area

Class file format

Register set

Garbage-collected heap

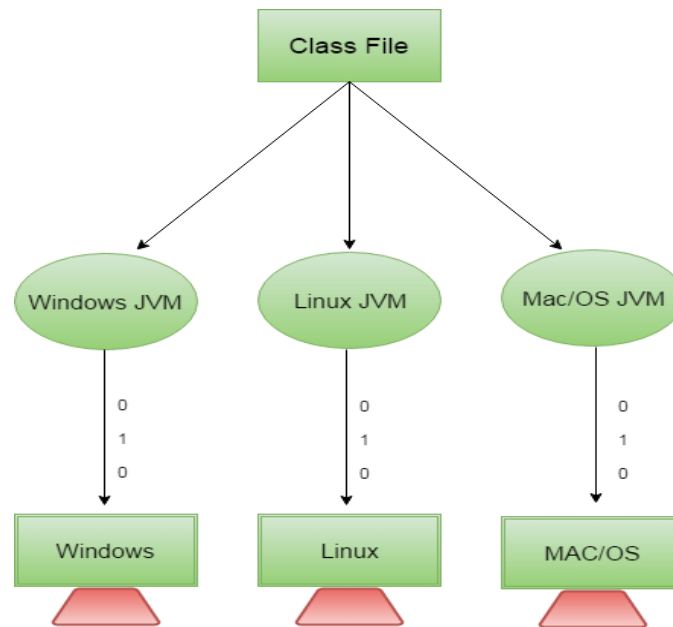
Fatal error reporting etc.

1. What is Java? Explain the internal architecture of JVM with neat sketch?(13) (NOV/DEC 2019)
2. How Java changed the internet?(9) (APR/MAY 2021)
3. If semicolons are needed at the end of each statement, why does the comment line not end with a semicolon ?(4) (APR/MAY 2021)
4. Outline the arithmetic operators in Java.(6) (NOV/DEC 2021)
5. Name the four integer types in Java and outline the bitwise operators that can be applied to the integer types.(7) (NOV/DEC 2021)
6. Outline the iteration statements in Java with syntax and example.(9) (NOV/DEC 2021)
7. What are the three categories of control statements used in Java ? Explain each category with example.(13) (APR/MAY 2021)
8. Write a java code using do-while loop that counts down to 1 from 10 printing exactly ten lines

The main objective of Java programming language creation was to make it portable, simple and secure programming language. Java is platform independent because it is different from other languages like C, C++ etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

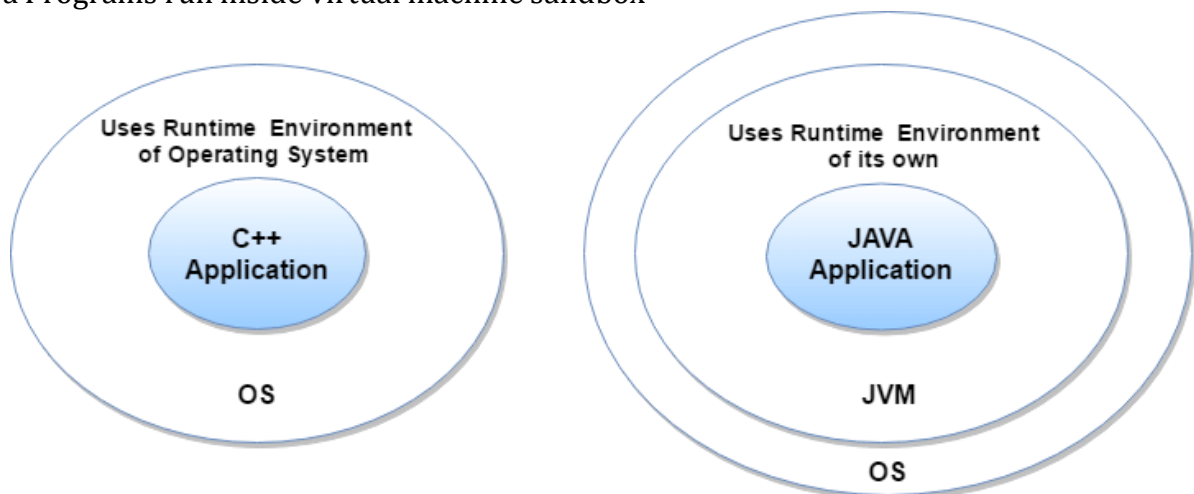


Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- No explicit pointer
- Java Programs run inside virtual machine sandbox



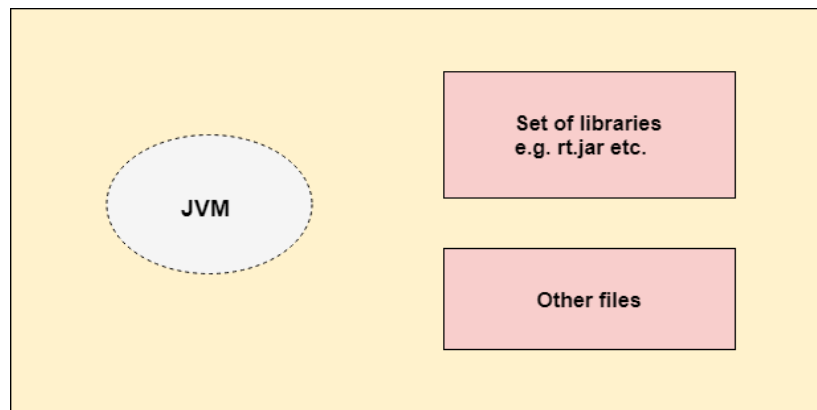
- **Class loader:** Class loader in Java is a part of the Java Runtime Environment(JRE) which is used to dynamically load Java classes into the Java Virtual Machine. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, Cryptography etc.

THE JAVA ENVIRONMENT

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing java applications. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime. Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JRE

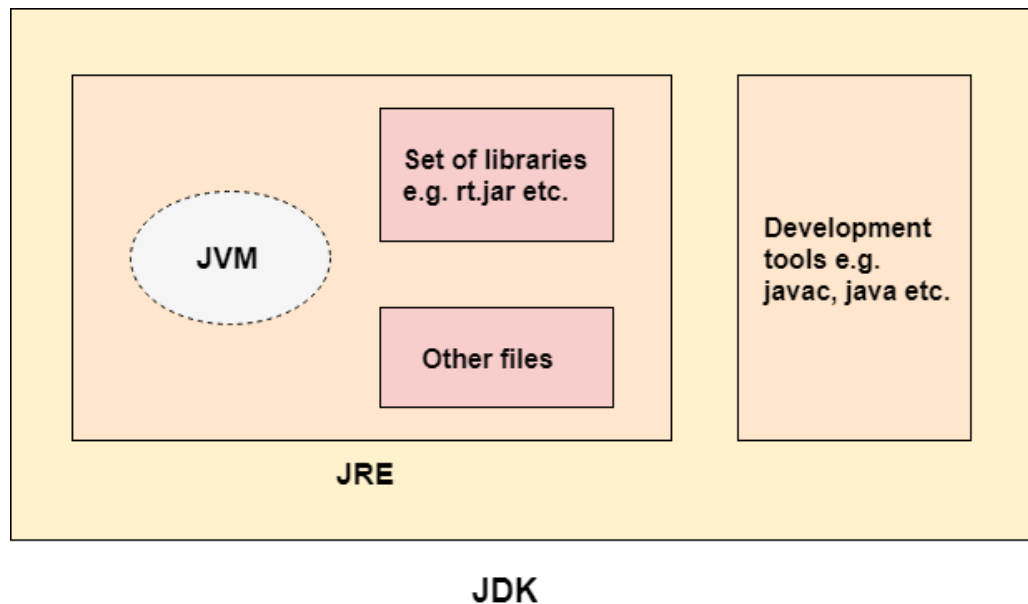
JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc. to complete the development of a Java Application.



JVM (Java Virtual Machine)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent). The JVM performs following operation:

- Loads code
- Verifies code
- Executes code

Provides runtime environment JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

3. PROGRAMMING STRUCTURES OF JAVA

1. Can a Java source file be saved using a name other than the class name?
Justify (2) **(APR/MAY 2019)**
2. Define Access Specifier(2) **(Nov/Dec 2018)**
3. What is Method? How method is defined? Give Example. **(Nov/Dec 2018)**
4. What is Java Doc?(2) **(Nov/Dec 2019)**
5. Explain in detail about Constructor with an example?(13)

A first Simple Java Program

```
class Simple
{
    public static void main(String args[])
    {
        System.out.println("Java World");
    }
}
```

To compile:

javac Simple.java

To execute:

java Simple

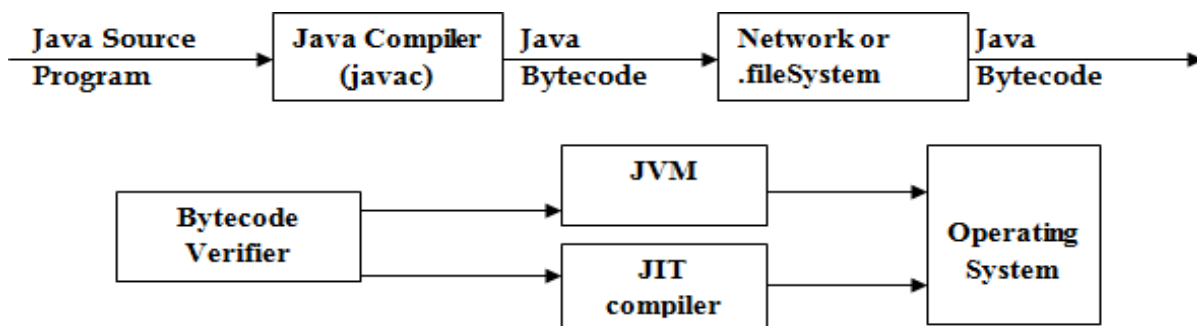
class keyword is used to declare a class in java.

public keyword is an access modifier which represents visibility, it means it is visible to all.

static is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory. **void** is the return type of the method, it means it doesn't return any value.

main represents the starting point of the program. **String[] args** is used for command line argument. **System.out.println()** is used print statement.

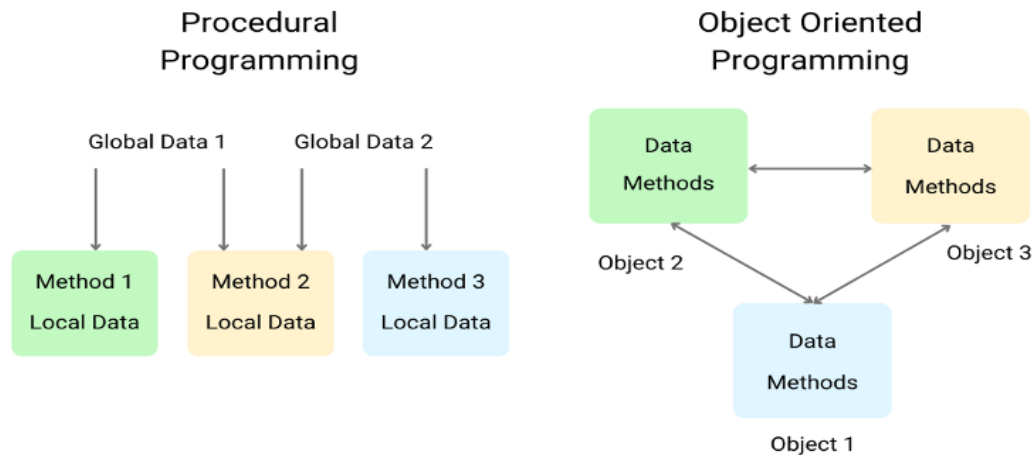
A program is written in JAVA, the javac compiles it. The result of the JAVA compiler is the .class file or the bytecode and not the machine native code (unlike C compiler). The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM. And finally program runs to give the desired output.



1.1 OBJECT ORIENTED PROGRAMMING PARADIGMS

The term programming paradigm is used to specify an overall approach to writing program code. The object-oriented programming paradigm (OOP) introduces a fundamentally different approach to program design. All computer programs consist of two elements: **code and data**. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around “what is happening” and others are written around “who is being affected.” These are the two paradigms that govern how a program is constructed.

The first way is called the **process- oriented model**. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success. To manage increasing complexity, the second approach, called **object- oriented programming**, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.



Object-oriented programming (OOP) is a programming paradigm based upon objects that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

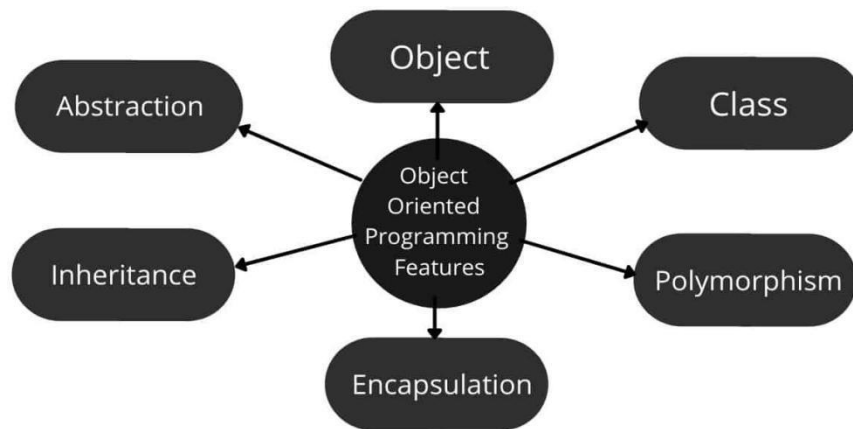
Object-Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system’s object model, which comprises of interacting objects.

Object-Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology-independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.

Procedure-Oriented Programming	Object-Oriented Programming
In POP, program is divided into small parts called functions	In OOP, program is divided into parts called objects .
In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world .
POP follows Top-Down approach .	OOP follows Bottom-Up approach .
POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
In POP, most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
POP does not have any proper way for hiding data so it is less secure .	OOP provides Data Hiding so provides more security .
In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Example of POP are: C, VB, FORTRAN, Pascal.	Example of OOP are: C++, JAVA, VB.NET, C#.NET.

The important features of Object Oriented programming are:

- **Inheritance**
- **Polymorphism**
- **Data Hiding**
- **Encapsulation**
- **Overloading**
- **Reusability**



It is important to know some new terminologies used in Object Oriented programming namely

- Objects
- Classes

Objects:

In other words, object is an instance of a class.

Classes:

These contain data and functions bundled together under a unit. In other words, class is a collection of similar objects. When we define a class, it just creates template or Skelton. So, no memory is created when class is created. Memory is occupied only by object.

Example:

```
Class classname
{
    Data
    Functions
};
main ( )
{
    classname objectname1,objectname2,...;
}
```

In other words classes acts as data types for objects.

Member functions:

The functions defined inside the class as above are called member functions. Here the concept of Data Hiding figures

Objects and Classes

Objects: Objects are instances of classes. They represent real-world entities with attributes and behaviors.

Classes: Classes are blueprints for creating objects. They define properties (attributes) and methods (behaviors).

Example

```
// Define a class
class Dog {
    // Attributes
    String name;
    int age;

    // Method
    void bark()
    {
        System.out.println(name + " is barking!");
    }
}
```

```
// Main class
public class Main
{
    public static void main(String[] args)
    {
        // Create an object
        Dog myDog = new Dog();
        myDog.name = "Buddy";
        myDog.age = 3;
        myDog.bark(); // Output: Buddy is barking!
    }
}
```

```
// Define a class
class Dog
{
    This line defines a new class named Dog. In Java, a class is a blueprint for creating objects. The keyword class is followed by the name of the class.
```

```
    // Attributes
    String name;
    int age;
```

These lines declare two attributes (also called fields or instance variables) for the Dog class. String name; declares a variable name of type String that will hold the dog's name. int age; declares a variable age of type int that will hold the dog's age.

```
    // Method
    void bark()
    {
        System.out.println(name + " is barking!");
    }
```

This is a method named bark in the Dog class. A method defines a behavior for the objects created from this class.

void indicates that this method does not return any value. Inside the method, System.out.println(name + " is barking!"); is a statement that prints the dog's name followed by the text " is barking!" to the console. The name variable is concatenated with the string " is barking!".

```
    }

// Main class
public class Main
{
    This line defines another class named Main. This class contains the main method, which is the entry point for the
```

program.

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

This line declares the main method, which is required for a Java program to run.
public means the method is accessible from anywhere.
static means the method belongs to the class, not a specific instance of the class.
void means the method does not return any value.

String[] args is an array of String objects that can store command-line arguments.

```
    // Create an object
    Dog myDog = new Dog();
```

This line creates an object of the Dog class.
Dog myDog; declares a variable myDog of type Dog.
new Dog(); creates a new instance of the Dog class and assigns it to myDog.

```
    myDog.name = "Buddy";
    myDog.age = 3;
```

These lines set the attributes of the myDog object.
myDog.name = "Buddy"; assigns the string "Buddy" to the name attribute of the myDog object.
myDog.age = 3; assigns the integer 3 to the age attribute of the myDog object.

```
    myDog.bark(); // Output: Buddy is barking!
```

This line calls the bark method on the myDog object.
myDog.bark(); invokes the bark method, which prints "Buddy is barking!" to the console because myDog.name is "Buddy".

```
    }
}
```

These closing braces } close the main method and the Main class, respectively.

Class Definition: class Dog and class Main
Attributes: String name and int age inside the Dog class
Method: void bark() inside the Dog class
Object Creation: Dog myDog = new Dog();
Setting Attributes: myDog.name = "Buddy"; and myDog.age = 3;
Method Invocation: myDog.bark();
This program defines a Dog class with attributes name and age, and a method bark(). The Main class contains the main method, which creates a Dog object, sets its attributes, and calls its bark method to display the dog's name followed by "is barking!" on the console.

Methods and Messages

Methods: Functions defined inside a class that describe the behaviors of the objects.

Messages: Calling methods on objects to perform actions.

Example

```

class Cat {
    // Attributes
    String name;
    int age;

    // Method
    void meow() {
        System.out.println(name + " says Meow!");
    }
}

public class Main {
    public static void main(String[] args) {
        Cat myCat = new Cat();
        myCat.name = "Whiskers";
        myCat.age = 2;
        myCat.meow(); // Output: Whiskers says Meow!
    }
}

```

Simple Algorithm

Define a class named Cat with attributes name and age.

Define a method named meow in the Cat class that prints a message including the cat's name.

Define a Main class containing the main method.

In the main method:

Create an instance of the Cat class.

Assign values to the name and age attributes of the cat instance.

Call the meow method on the cat instance to print the message.

Line-by-Line Explanation

```

class Cat

```

```

{

```

This line defines a new class named Cat. In Java, a class is a blueprint for creating objects. The keyword class is followed by the name of the class.

```

    // Attributes

```

```

    String name;

```

```

    int age;

```

These lines declare two attributes (also called fields or instance variables) for the Cat class.

String name; declares a variable name of type String that will hold the cat's name.

int age; declares a variable age of type int that will hold the cat's age.

```
// Method
void meow() {
    System.out.println(name + " says Meow!");
}
```

This is a method named meow in the Cat class. A method defines a behavior for the objects created from this class.

void indicates that this method does not return any value.

Inside the method, System.out.println(name + " says Meow!"); is a statement that prints the cat's name followed by the text " says Meow!" to the console. The name variable is concatenated with the string " says Meow!".

```
}
```

This closing brace } marks the end of the Cat class definition.

❖ public class Main {

This line defines another class named Main. This class contains the main method, which is the entry point for the program.

❖ public static void main(String[] args) {

This line declares the main method, which is required for a Java program to run.

public means the method is accessible from anywhere.

static means the method belongs to the class, not a specific instance of the class.

void means the method does not return any value.

String[] args is an array of String objects that can store command-line arguments.

❖ Cat myCat = new Cat();

This line creates an object of the Cat class.

❖ Cat myCat; declares a variable myCat of type Cat.

❖ new Cat(); creates a new instance of the Cat class and assigns it to myCat.

myCat.name = "Whiskers";
myCat.age = 2;

These lines set the attributes of the myCat object.

myCat.name = "Whiskers"; assigns the string "Whiskers" to the name attribute of the myCat object.

myCat.age = 2; assigns the integer 2 to the age attribute of the myCat object.

```
    myCat.meow(); // Output: Whiskers says Meow!
```

This line calls the meow method on the myCat object.

myCat.meow(); invokes the meow method, which prints "Whiskers says Meow!" to the console because myCat.name is "Whiskers".

```
    }
}
```

These closing braces } close the main method and the Main class, respectively.

Summary

Class Definition: class Cat and class Main

Attributes: String name and int age inside the Cat class

Method: void meow() inside the Cat class

Object Creation: `Cat myCat = new Cat();`

Setting Attributes: `myCat.name = "Whiskers";` and `myCat.age = 2;`

Method Invocation: `myCat.meow();`

This program defines a Cat class with attributes name and age, and a method meow(). The Main class contains the main method, which creates a Cat object, sets its attributes, and calls its meow method to display the cat's name followed by "says Meow!" on the console.

Data Hiding:

This concept is the main heart of an Object oriented programming. The data is hidden inside the class by declaring it as private inside the class. When data or functions are defined as private it can be accessed only by the class in which it is defined. When data or functions are defined as public then it can be accessed anywhere outside the class. Object Oriented programming gives importance to protecting data which in any system. This is done by declaring data as private and making it accessible only to the class in which it is defined. This concept is called data hiding. But one can keep member functions as public.

So above class structure becomes

Example:

```
Class classname
{
    private:
        datatype data;

    public:
        Member functions
};
main ( )
{
    classname objectname1,objectname2,...;
}
```

Encapsulation:

The technical term for combining data and functions together as a bundle is encapsulation.

Inheritance:

Inheritance as the name suggests is the concept of inheriting or deriving properties of an existing class to get new class or classes. In other words we may have common features or characteristics that may be needed by number of classes. So those features can be placed in a common tree class called base class and the other classes which have these characteristics can take the tree class and define only the new things that they have on their own in their classes. These classes are called derived class. The main advantage of using this concept of inheritance in Object oriented programming is it helps in reducing the code size since the common characteristic is placed separately called as base class and it is just referred in the derived class. This provides the users the important usage of terminology called as reusability.

Reusability:

This usage is achieved by the above explained terminology called as inheritance. Reusability is nothing but re-usage of structure without changing the existing one but adding new features or characteristics to it. It is very much needed for any programmers in different situations. Reusability gives the following advantages to user. It helps in reducing the code size since classes can be just derived from existing one and one need to add only the new features and it helps users to save their time.

For instance if there is a class defined to draw different graphical figures say there is a user who wants to draw graphical figure and also add the features of adding color to the graphical figure. In this scenario instead of defining a class to draw a graphical figure and coloring it what the user can do is make use of the existing class for drawing graphical figure by deriving the class and add new feature to the derived class namely add the feature of adding colors to the graphical figure.

Polymorphism and Overloading:

Poly refers many. So Polymorphism as the name suggests is a certain item appearing in different forms or ways. That is making a function or operator to act in different forms depending on the place they are present is called Polymorphism. Overloading is a kind of polymorphism. In other words say for instance we know that +, - operate on integer data type and is used to perform arithmetic additions and subtractions. But operator overloading is one in which we define new operations to these operators and make them operate on different data types in other words overloading the existing functionality with new one. This is a very important feature of object oriented programming methodology which extended the handling of data type and operations.

Thus the above given important features of object oriented programming among the numerous features it has give the following advantages to the programming world.

The advantages are namely,

- Data Protection or security of data is achieved by concept of data hiding

- Reduces program size and saves time by the concept of reusability which is achieved by the terminology of Inheritance
- Operators can be given new functions as per user which extends the usage.

Abstraction and Encapsulation

Abstraction: Hiding complex implementation details and showing only the essential features.

Encapsulation: Wrapping the data (attributes) and methods into a single unit or class. Access to the data is restricted by using access modifiers.

Example

```
class Person {
    // Private attributes
    private String name;
    private int age;

    // Public methods to access private attributes
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person();
        person.setName("Alice");
        person.setAge(30);
        System.out.println(person.getName() + " is " + person.getAge() + " years old."); // Output: Alice is 30 years old.
    }
}
```

Inheritance

Inheritance: Mechanism where one class (subclass) inherits the attributes and methods of another class (superclass).

Example

```
// Superclass
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}
```

```
// Subclass
class Dog extends Animal {
    void bark() {
        System.out.println("The dog barks.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat(); // Inherited method
        myDog.bark(); // Own method
    }
}
```

Abstract Classes

Abstract Classes: Classes that cannot be instantiated. They can have abstract methods (without implementation) and concrete methods (with implementation).

Example

```

abstract class Animal {
    abstract void sound(); // Abstract method

    void sleep() { // Concrete method
        System.out.println("This animal is sleeping.");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sound(); // Output: The dog barks.
        myDog.sleep(); // Output: This animal is sleeping.
    }
}

```

Polymorphism

Polymorphism: The ability to take many forms. It allows methods to do different things based on the object it is acting upon, even though they share the same name.

Example

```

class Animal {
    void sound() {
        System.out.println("Some sound.");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("The dog barks.");
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("The cat meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
    }
}

```

```

Animal myCat = new Cat();
myDog.sound(); // Output: The dog barks.
myCat.sound(); // Output: The cat meows.
}
}

```

DEFINING CLASSES IN JAVA

The class is at the core of Java .A class is a *template* for an object, and an object is an *instance* of a class. Aclass is declared by use of the **class** keyword

Syntax:

```

class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
...
type methodnameN(parameter-list) {
// body of method
}

```

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. The methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

A Simple Class

class called **Box** that defines three instance variables: **width**, **height**, and **depth**.

```

class Box
{
double width;
double height;

```

```
double depth;  
}
```

The new data type is called **Box**. This name is used to declare objects of type **Box**. The class declaration only creates a template. It does not create an actual object.

To create a Box object

```
Box mybox = new Box(); // create a Box object called mybox
```

mybox will be an instance of **Box**.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable.

Example1:

```
/* A program that uses the Box class. Call this file BoxDemo.java  
*/  
class Box  
{  
double width; double height; double depth;  
}  
// This class declares an object of type Box.  
class BoxDemo {  
public static void main(String args[])  
{Box mybox = new Box();  
double vol;  
// assign values to mybox's instance variables  
mybox.width = 10;  
mybox.height = 20;  
mybox.depth = 15;  
// compute volume of box  
vol = mybox.width * mybox.height * mybox.depth;  
System.out.println("Volume is " + vol);  
}  
}
```

Output:

Volume is 3000.0

Example2:

```
// This program declares two Box objects.  
class Box {  
double width; double height; double depth;  
}  
class BoxDemo2  
{
```

```

public static void main(String args[])
{Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's instance variables
*/
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}

```

Output:

Volume is 3000.0

Volume is 162.0

Declaring Objects

First, declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.

Second, you must acquire an actual, physical copy of the object and assign it to that variable. This is done using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

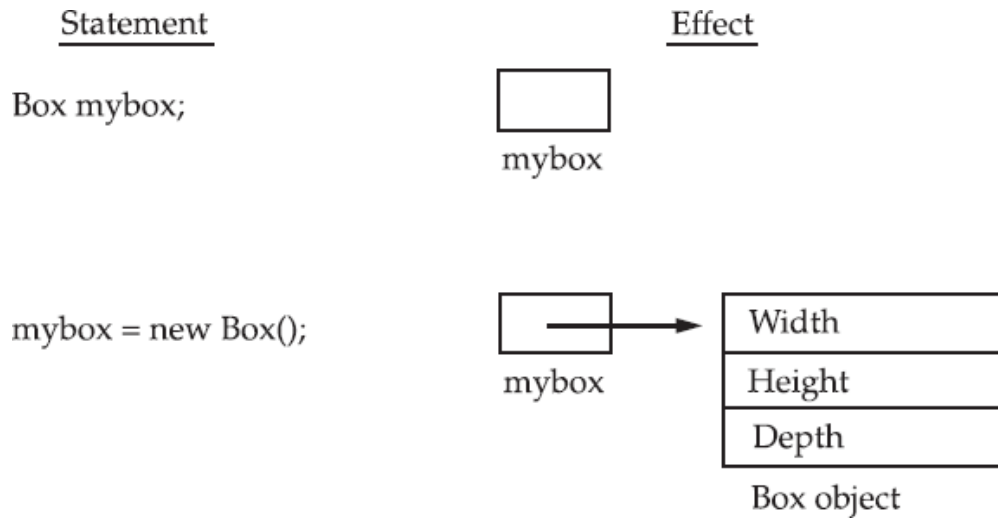
Syntax:

```

Box mybox = new Box();
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object

```

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, we can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds, in essence, the memory address of the actual **Box** object.



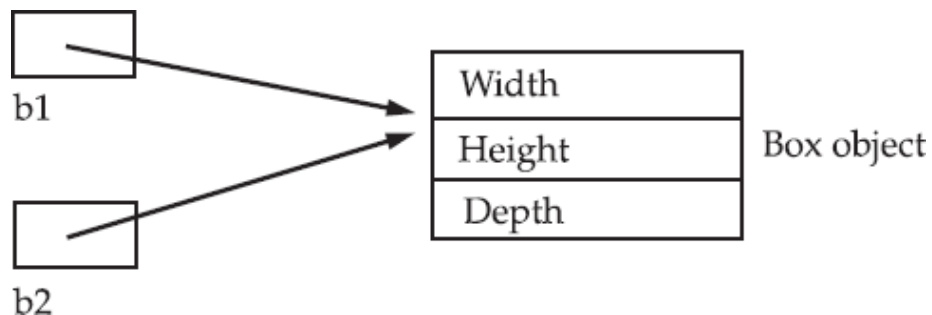
Assigning Object Reference

Variables Syntax:

Box b1 = new Box();

Box b2 = b1;

b2 is being assigned a reference to a copy of the object referred to by **b1**. **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.



CONSTRUCTORS

Constructors are special member functions whose task is to initialize the objects of its class. It is a special member function; it has the same as the class name.

Java constructors are invoked when their objects are created. It is named such because, it constructs the value, that is provides data for the object and are used to initialize objects.

Every class has a constructor when we don't explicitly declare a constructor for any java class the compiler creates a default constructor for that class which does not have any return type.

The constructor in Java cannot be abstract, static, final or synchronized and these modifiers are not allowed for the constructor.

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Default constructor (no-arg constructor)

A constructor having no parameter is known as default constructor and no-arg constructor.

Example:

```
/* Here, Box uses a constructor to initialize the dimensions of a box.
```

```
*/
class Box
{
    double width;double height;double depth;
    // This is the constructor for Box.
    Box()
    {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo6 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // get volume of first box vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

Output:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

new Box() is calling the **Box()** constructor. When the constructor for a class is not explicitly defined, then Java creates a default constructor for the class. The default constructor automatically initializes all instance variables to their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively.

Parameterized Constructors

A constructor which has a specific number of parameters is called parameterized constructor. Parameterized constructor is used to provide different values to the distinct objects.

Example:

```

/* Here, Box uses a parameterized
constructor to initialize the dimensions of a
box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d)
{width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box
objectsBox mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second
boxvol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}

```

Output:

Volume is 3000.0

Volume is 162.0

Box mybox1 = new Box(10, 20, 15);

The values 10, 20, and 15 are passed to the **Box()** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15 respectively.

Overloading

ConstructorsExample:

```

/* Here, Box defines three constructors to initialize the dimensions of a box various ways.
*/
class Box {
double width;
double height;

```

```

double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width  = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons
{
public static void main(String args[])
{
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second
boxvol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}

```

Output:

```

Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0

```

METHODS

Syntax:

```
type name(parameter-list) {  
    // body of method  
}
```

type specifies the type of data returned by the method. This can be any valid type, including classtypes that you create.

If the method does not return a value, its return type must be **void**.

The name of the method is specified by *name*.

The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

Syntax:

```
return value;
```

Example:

```
// This program includes a method inside the box  
class class Box {  
    double width;  
    double height;  
    double depth;  
    // display volume of a box  
    void volume() {  
        System.out.print("Volume is ");  
        System.out.println(width * height * depth);  
    }  
}  
class BoxDemo3 {  
    public static void main(String args[])  
    {Box mybox1 = new Box();  
     Box mybox2 = new Box();  
     // assign values to mybox1's instance variables  
     mybox1.width = 10;  
     mybox1.height = 20;  
     mybox1.depth = 15;  
     /* assign different values to  
     mybox2's instance variables */  
     mybox2.width = 3;  
     mybox2.height = 6;  
     mybox2.depth = 9;  
     // display volume of first box  
     mybox1.volume();  
     // display volume of second box  
     mybox2.volume();  
    }  
}
```

Output:

olume is 3000.0

Volume is 162.0

The first line here invokes the **volume()** method on **mybox1**. That is, it calls **volume()** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume()** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume()** displays the volume of the box defined by **mybox2**. Each time **volume()** is invoked, it displays the volume for the specified box.

Returning a

ValueExample:

// Now, volume() returns the volume of a box.

```
class Box {
    double width;
    double height;
    double depth;
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo4 {
    public static void main(String args[])
    {Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;
    /* assign different values to mybox2's instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);
    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
    }
}
```

Output:

Volume is 3000

Volume is 162

when **volume()** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume()**.

Syntax:

```
vol = mybox1.volume();
```

executes, the value of **mybox1.volume()** is 3,000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

Adding a Method That Takes Parameters

Example:

// This program uses a parameterized method.

```
class Box {  
    double    width;  
    double    height;  
    double depth;  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
    // sets dimensions of box  
    void setDim(double w, double h, double d)  
    {width = w;  
    height = h;  
    depth = d;  
    }  
}
```

```
class BoxDemo5 {  
    public static void main(String args[])  
    {Box mybox1 = new Box();  
    Box mybox2 = new Box();  
    double vol;  
    // initialize each box  
    mybox1.setDim(10, 20, 15);  
    mybox2.setDim(3, 6, 9);  
    // get volume of first box  
    vol = mybox1.volume();  
    System.out.println("Volume is " + vol);  
    // get volume of second  
    boxvol = mybox2.volume();  
    System.out.println("Volume is " + vol);  
    }  
}
```

Output:

```
Volume is 3000  
Volume is 162
```

This Keyword

this keyword is used to refer to the object that invoked it. this can be used inside any method to refer to the *current* object. That is, this is always a reference to the object on which the method was invoked. this() can be used to invoke current class constructor.

Syntax:

```
Box(double w, double h, double d) {  
    this.width = w;  
    this.height = h;  
    this.depth = d;  
}
```

Example:

```
class Student  
{  
    int id;  
    String name;  
    student(int id, String name)  
    {  
        this.id = id;  
        this.name = name;  
    }  
    void display()  
    {  
        System.out.println(id+" "+name);  
    }  
    public static void main(String args[])  
    {  
        Student stud1 = new Student(01,"Tarun");  
        Student stud2 = new Student(02,"Barun");  
        stud1.display();  
        stud2.display();  
    }  
}
```

Output:

```
01 Tarun  
02 Barun
```

Overloading Methods

When two or more methods within the same class that have the same name, but their parameter declarations are different. The methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism.

There are two ways to overload the method in java

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

Example:

```
// Demonstrate method overloading.  
class OverloadDemo {
```

```

void test() {
System.out.println("No parameters");
}

// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// Overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new
OverloadDemo();double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}

```

Output:

```

No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625

```

Method Overriding

When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Example:

```

// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;

```

```

j = b;
}
// display i and j
void show() {

System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c)
{super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[])
{B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}

```

Output:

k: 3

When **show()** is invoked on an object of type **B**, the version of **show()** defined within **B** is used. That is, the version of **show()** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show()** is invoked within the subclass' version. This allows all instance variables to be displayed.

```

class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
}

```

If you substitute this version of **A** into the previous program, you will see the following

Output:

i and j: 1 2
k: 3

Here, **super.show()** calls the superclass version of **show()**.

ACCESS SPECIFIERS

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class. There are 4 types of java access modifiers:

- ❖ private
- ❖ default
- ❖ protected
- ❖ public

1) Private Access Modifier

The private access modifier is accessible only within class. Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{
private int data=40;
private void msg()
{System.out.println("Hello java");}
}
public class Simple{
public static void main(String args[]){
A obj=new A();
System.out.println(obj.data);//Compile Time Error
obj.msg();//Compile Time Error
}
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
private A(){} //private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
public static void main(String args[]){
A obj=new A();//Compile Time Error
}
}
```

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
private A(){} //private constructor
void msg(){System.out.println("Hello java");}
}
```

```

public class Simple{
    public static void main(String args[]){
        A obj=new A();//Compile Time Error
    }
}

```

2) Default Access Modifier

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package.

Example:

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```

//save by A.java
package pack;
class A{
    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();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}

```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected Access Modifier

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example:

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```

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

```

```
//save by B.java
package mypack;
import pack.*;

class B extends A{
    public static void main(String args[]){
        B obj = new B();
        obj.msg();
    }
}
```

Output:

Hello

4) Public Access Modifier

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example:

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

Output:

Hello

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

Java access modifiers with method overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more

restrictive.

```
class A{
protected void msg(){System.out.println("Hello java");}
}
public class Simple extends A{
void msg(){System.out.println("Hello java");} //C.T.Error
public static void main(String args[]){
    Simple obj=new Simple();
    obj.msg();
}
}
```

The default modifier is more restrictive than protected. That is why there is compile time error.

STATIC MEMBERS

Static is a non-access modifier in Java which is applicable for the following:

- ❖ blocks
- ❖ variables
- ❖ methods
- ❖ nested classes

Static blocks

If you need to do computation in order to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded.

Example:

```
// Java program to demonstrate use of static blocks
class Test
{
    // static variable
    static int a = 10;
    static int b;

    // static
    blockstatic {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String[] args)
    {
        System.out.println("from main");
        System.out.println("Value of a : "+a);
        System.out.println("Value of b : "+b);
    }
}
```

Output:

```
Static block
initialized.from main
```

Value of a : 10

Value of b : 40

Static variables

When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables :-

- We can create static variables at class-level only.
- static block and static variables are executed in order they are present in a program.

Example:

// Demonstrate static variables, methods, and blocks.

```
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

Output:

```
Static block
initialized.x = 42
a = 3
b = 12
```

Static methods

When a method is declared with *static* keyword, it is known as static method. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. The most common example of a static method is *main()* method.

Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to **this** or **super** in any way.

Syntax:

```
classname.method( )
```

Example:

//Inside main(), the static method callme() and the static variable b are accessed through their class name

```
//StaticDemo.  
class StaticDemo {  
    static int a = 42; static int b = 99;  
    static void callme() {  
        System.out.println("a = " + a);  
    }  
}  
class StaticByName {  
    public static void main(String args[]) {  
        StaticDemo.callme();  
        System.out.println("b = " + StaticDemo.b);  
    }  
}
```

Output:

```
a = 42  
b = 99
```

Finalize method

Algorithm: Using the finalize Method

1. **Define a Class:** Create a class that will contain the finalize method.
2. **Create a Constructor:** Define a constructor for the class to initialize objects and optionally print a message when an object is created.
3. **Override the finalize Method:** Override the finalize method from the Object class. Inside this method, add the code to be executed just before the object is garbage collected.
4. **Create an Object:** In the main method, create an instance of the class.
5. **Make the Object Eligible for Garbage Collection:** Set the object reference to null to make it eligible for garbage collection.
6. **Request Garbage Collection:** Call `System.gc()` to suggest that the JVM performs garbage collection.
7. **Wait for Garbage Collection:** Optionally, add a small delay to give the garbage collector time to run.
8. **End the Program:** Print a message indicating the end of the program.

Step-by-Step Example

Let's use the Car class example again:

Step 1: Define a Class

```
java
Copy code
public class Car {
```

Step 2: Create a Constructor

```
java
Copy code
// Constructor
public Car() {
    System.out.println("Car is created");
}
```

Step 3: Override the finalize Method

```
java
Copy code
// The finalize method is called when the object is about to be garbage collected
@Override
protected void finalize() throws Throwable {
    try {
        System.out.println("Car is being destroyed");
    } finally {
        super.finalize(); // Ensure the superclass finalize method is called
    }
}
```

Step 4: Create an Object

```
java
Copy code
public static void main(String[] args) {
    Car myCar = new Car(); // Create a Car object
}
```

Step 5: Make the Object Eligible for Garbage Collection

```
java
Copy code
myCar = null; // Make the Car object eligible for garbage collection
```

Step 6: Request Garbage Collection

```
java
Copy code
// Requesting garbage collection
System.gc();
```

Step 7: Wait for Garbage Collection

```
java
Copy code
// Adding a small delay to ensure garbage collection happens
try {
    Thread.sleep(1000);
}
```

```
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

Step 8: End the Program

```
java  
Copy code  
    System.out.println("End of main method");  
}  
}
```

Full Example

```
java  
Copy code  
public class Car {  
    // Constructor  
    public Car() {  
        System.out.println("Car is created");  
    }  
  
    // The finalize method is called when the object is about to be garbage collected  
    @Override  
    protected void finalize() throws Throwable {  
        try {  
            System.out.println("Car is being destroyed");  
        } finally {  
            super.finalize(); // Ensure the superclass finalize method is called  
        }  
    }  
  
    public static void main(String[] args) {  
        Car myCar = new Car(); // Create a Car object  
        myCar = null; // Make the Car object eligible for garbage collection  
  
        // Requesting garbage collection  
        System.gc();  
  
        // Adding a small delay to ensure garbage collection happens  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
  
        System.out.println("End of main method");  
    }  
}
```

Summary

This step-by-step algorithm helps you understand how to use the finalize method in Java. By following these steps, you can see how the finalize method works in conjunction with garbage collection to perform cleanup tasks before an object is destroyed.

The finalize method in Java was traditionally used for cleanup activities before an object is garbage collected. However, it has several drawbacks, such as unpredictability in its execution time, performance issues, and potential risks of resource leaks if not handled properly. Because of these issues, its use is generally discouraged in favor of more reliable and predictable resource management techniques.

Applications of Using finalize

1. **Resource Cleanup:** finalize was used to release resources like file handles, sockets, and database connections that are not automatically managed by the garbage collector.
2. **Native Resources:** Cleanup of native resources allocated in JNI (Java Native Interface) code.
3. **Logging:** Logging when an object is garbage collected for debugging or monitoring purposes.

Why Avoid finalize

1. **Unpredictability:** You cannot predict when (or even if) the garbage collector will call finalize.
2. **Performance:** Objects with finalize methods take longer to be reclaimed by the garbage collector, leading to memory pressure.
3. **Complexity:** Using finalize can introduce complex bugs and unintended behavior, especially if exceptions occur within the finalize method.

Modern Alternatives

1. **Try-with-Resources:** For managing resources that implement the AutoCloseable interface, which ensures that resources are closed at the end of the statement.

```
java
Copy code
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    // Use the resource
} catch (IOException e) {
    e.printStackTrace();
}
```

2. **Explicit Cleanup Methods:** Define an explicit method to release resources and ensure it is called by the client code.

```
java
Copy code
public class Resource {
    public void close() {
        // Cleanup code
    }
}

public static void main(String[] args) {
    Resource resource = new Resource();
    try {
        // Use the resource
    } finally {
        resource.close();
    }
}
```

```
}  
}
```

3. **Phantom References and Cleaner API:** For more advanced use cases, Java provides the `java.lang.ref.Cleaner` API and phantom references for managing resource cleanup more effectively.

java

Copy code

```
public class Resource {  
    private static final Cleaner cleaner = Cleaner.create();  
    private final Cleaner.Cleanable cleanable;  
  
    public Resource() {  
        this.cleanable = cleaner.register(this, new ResourceCleanup());  
    }  
  
    private static class ResourceCleanup implements Runnable {  
        @Override  
        public void run() {  
            System.out.println("Resource is being cleaned up");  
            // Cleanup code  
        }  
    }  
}
```

Conclusion

While the `finalize` method was historically used for cleanup purposes, its drawbacks and the availability of better alternatives have led to it being deprecated and discouraged in modern Java development. Using `try-with-resources`, explicit cleanup methods, or the Cleaner API are recommended approaches for managing resource cleanup in a more reliable and predictable manner.

=====

=====

7. Write a Java program to swap two variables.

8. Write a Java program to print a face.

Expected Output

```
+*****+  
[| o o |]  
| ^ |  
| '-' |  
+-----+
```

9. Write a Java program to add two binary numbers.

Input Data:

Input first binary number: 10

Input second binary number: 11

Expected Output

Sum of two binary numbers: 101

10. Write a Java program to multiply two binary numbers.

Input Data:

Input the first binary number: 10

Input the second binary number: 11

Expected Output

Product of two binary numbers: 110

11. Write a Java program to convert a decimal number to binary number.

Input Data:

Input a Decimal Number : 5

Expected Output

Binary number is: 101

12. Write a Java program to convert a decimal number to hexadecimal number.

Input Data:

Input a decimal number: 15

Expected Output

Hexadecimal number is : F

13. Write a Java program to convert a decimal number to octal number.

Input Data:

Input a Decimal Number: 15

Expected Output

Octal number is: 17

14. Write a Java program to convert a binary number to decimal number.

Input Data:

Input a binary number: 100

Expected Output

Decimal Number: 4

15. Write a Java program to convert a binary number to hexadecimal number.

Input Data:

Input a Binary Number: 1101

Expected Output

HexaDecimal value: D

16. Write a Java program to convert a binary number to a Octal number.

Input Data:

Input a Binary Number: 111

Expected Output

Octal number: 7

17. Write a Java program to convert a octal number to a decimal number.

Input Data:

Input any octal number: 10

Expected Output

Equivalent decimal number: 8

18. Write a Java program to convert a octal number to a binary number.

Input Data:

Input any octal number: 7

Expected Output

Equivalent binary number: 111

19. Write a Java program to convert a octal number to a hexadecimal number.

Input Data:

Input a octal number : 100

Expected Output

Equivalent hexadecimal number: 40

20. Write a Java program to convert a hexadecimal to a decimal number.

Input Data:

Input a hexadecimal number: 25

Expected Output

Equivalent decimal number is: 37

21. Write a Java program to convert a hexadecimal to a binary number.

Input Data:

Enter Hexadecimal Number : 37

Expected Output

Equivalent Binary Number is: 110111

22. Write a Java program to convert a hexadecimal to a octal number.

Input Data:

Input a hexadecimal number: 40

Expected Output

Equivalent of octal number is: 100

23. Write a Java program to check whether Java is installed on your computer.

Expected Output

Java Version: 1.8.0_71

Java Runtime Version: 1.8.0_71-b15

Java Home: /opt/jdk/jdk1.8.0_71/jre

Java Vendor: Oracle Corporation

Java Vendor URL: <http://Java.oracle.com/>

Java Class Path: .

24. Write a Java program to compare two numbers.

Input Data:

Input first integer: 25

Input second integer: 39

Expected Output

25 != 39

25 < 39

25 <= 39

25. Write a Java program and compute the sum of the digits of an integer.

Input Data:

Input an integer: 25

Expected Output

The sum of the digits is: 7

26. Write a Java program to compute the area of a hexagon.

Area of a hexagon = $(6 * s^2) / (4 * \tan(\pi/6))$

where s is the length of a side

Input Data:

Input the length of a side of the hexagon: 6

Expected Output

The area of the hexagon is: 93.53074360871938

27. Write a Java program to compute the area of a polygon.

Area of a polygon = $(n * s^2) / (4 * \tan(\pi/n))$

where n is n-sided polygon and s is the length of a side

Input Data:

Input the number of sides on the polygon: 7

Input the length of one of the sides: 6

Expected Output

The area is: 130.82084798405722

28. Write a Java program to compute the distance between two points on the surface of earth.

Distance between the two points [(x1,y1) & (x2,y2)]

$d = \text{radius} * \arccos(\sin(x1) * \sin(x2) + \cos(x1) * \cos(x2) * \cos(y1 - y2))$

Radius of the earth r = 6371.01 Kilometers

Input Data:

Input the latitude of coordinate 1: 25

Input the longitude of coordinate 1: 35

Input the latitude of coordinate 2: 35.5

Input the longitude of coordinate 2: 25.5

Expected Output

The distance between those points is: 1480.0848451069087 km