III SEMESTER

Object Oriented Programming

COURSE 2

P Veera Venkata Durga PraSad
DEPARTMENT OF COMPUTER SCIENCE (AWDC KKD)

Computer Science Minor:
III Semester Course 2:
Object Oriented Programming using Java

UNIT-I

OOPs Concepts and Java Programming: Introduction to Object-Oriented concepts, procedural and object-oriented programming paradigm

Java programming: An Overview of Java, Java Environment, Data types, Variables, constants, scope and life time of variables, operators, type conversion and casting, Accepting Input from the Keyboard, Reading Input with Java.util.Scanner Class, Displaying Output with System.out.printf(), Displaying Formatted Output with String.format(), Control Statements

UNIT-II

Arrays, Command Line Arguments, Strings-String Class Methods

Classes & Objects: Creating Classes, declaring objects, Methods, parameter passing, static fields and methods, Constructors, and 'this' keyword, overloading methods and access

Inheritance: Inheritance hierarchies, super and subclasses, member access rules, 'super' keyword, preventing inheritance: final classes and methods, the object class and its methods; **Polymorphism**: Dynamic binding, method overriding, abstract classes and methods;

UNIT-III

Interface: Interfaces VS Abstract classes, defining an interface, implement interfaces, accessing implementations through interface references, extending interface;

Packages: Defining, creating and accessing a package, understanding CLASSPATH, importing packages.

Exception Handling: Benefits of exception handling, the classification of exceptions, exception hierarchy, checked exceptions and unchecked exceptions, usage of try, catch, throw, throws and finally, rethrowing exceptions, exception specification, built in exceptions, creating own exception sub classes.

UNIT-IV

Multithreading: Differences between multiple processes and multiple threads, thread states, thread life cycle, creating threads, interrupting threads, thread priorities, synchronizing threads, inter thread communication.

Stream based I/O (java.io) – The Stream classes-Byte streams and Character streams, Reading console Input and Writing Console Output, File class, Reading and writing Files, The Console class, Serialization

UNIT-V

GUI Programming with Swing- Introduction, MVC architecture, components, containers. Understanding Layout Managers - Flow Layout, Border Layout, Grid Layout, Card Layout, Grid Bag Layout.

Event Handling- The Delegation event model- Events, Event sources, Event Listeners, Event classes, Handling mouse and keyboard events, Adapter classes, Inner classes, Anonymous Inner classes.

UNIT-1

❖ Introduction to OOPS / Features of OOPS:

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

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

Object:

➢ Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc.
➢ It can be physical or logical.An Object can be defined as an instance of a class.
➢ An object contains an address and takes up some space in memory.
Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class:

➢ *Collection of objects* is called class. It is a logical entity.
➢ A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance:

➢ *When one object acquires all the properties and behaviors of a parent object,* it is known as inheritance.
➢ When we write a class, we inherit properties from other classes.

It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism:

➢ If *one task is performed in different ways,* it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

➢ In Java, we use method overloading and method overriding to achieve polymorphism.

➢ Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction:

- *Hiding internal details and showing functionality* is known as abstraction.
- For example phone call, we don't know the internal processing.
- In Java, we use abstract class and interface to achieve abstraction.

Encapsulation:

- *Binding (or wrapping) code and data together into a single unit are known as encapsulation.*
- For example, a capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Association:

- Association represents the relationship between the objects. Here, one object can be associated with one object or many objects.
- There can be four types of association between the objects:
  - One to One
  - One to Many
  - Many to One, and
  - Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

Aggregation:

- Aggregation is a way to achieve Association.
- Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects.
- It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

Composition:

➢ The composition is also a way to achieve Association.

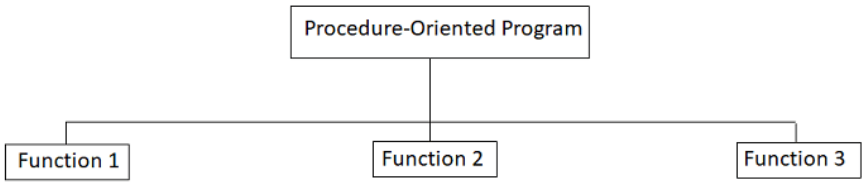
➢ The composition represents the relationship where one object contains other objects as a part of its state.

➢ There is a strong relationship between the containing object and the dependent object.

➢ It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

❖ Procedure Oriented Approach

➢ Procedure-Oriented Programming is the traditional way of programming, where an application problem is viewed as a sequence of steps (algorithms).
➢ As per the algorithm, the problem is broken down into many modules (functions) such as data entry, reporting, querying modules, etc. as shown in the figure.



Generalized Structure of a Procedure-Oriented Program:

➢ There are two types of data, which are associated with these modules- one is global and another is local data.
➢ Global data items are defined in the main program, whereas local data is define within associated functions.
➢ High-level languages like COBOL, Pascal, BASIC, Fortran, C, etc. are based on a procedure-oriented approach and hence are also called procedural languages.

❖ Differences between OOP and POP:

OOP

OOP, refers to Object Oriented Programming and its deals with objects and their properties. Major concepts of OOPs are –

- Class/objects
- Abstraction
- Encapsulation
- Polymorphism

- Inheritance

POP

POP, refers to Procedural Oriented Programming and its deals with programs and functions. Programs are divided into functions and data is global.

Following are the important differences between OOP and POP.

Sr. No.	Key	OOP	POP
1	Definition	OOP stands for Object Oriented Programing.	POP stands for Procedural Oriented Programming.
2	Approach	OOP follows bottom up approach.	POP follows top down approach.
3	Division	A program is divided to objects and their interactions.	A program is divided into funtions and they interacts.
4	Inheritance supported	Inheritance is supported.	Inheritance is not supported.
5	Access control	Access control is supported via access modifiers.	No access modifiers are supported.
6	Data Hiding	Encapsulation is used to hide data.	No data hiding present. Data is globally accessible.
7	Example	C++, Java	C, Pascal

❖ Applications of Object Oriented Programming

The modern software engineering landscape is changing at a blistering pace. AI integration, low-code applications, and new engineering efficiency tools have set the stage for a busy decade to come (DevPro Journal). These changes make programming even more important as a cornerstone strategy for building versatile, scalable, efficient applications.

1. Client-Server Systems

Object oriented client-server systems represent a powerful approach to designing and implementing robust IT infrastructures. In these systems, programming principles are applied to create a structured and modular architecture for building client and server components to

provide the IT infrastructure and create Object Oriented Client-Server Internet (OCSI) applications.

2. Object Oriented Databases

Object Oriented Databases, often referred to as Object Database Management Systems (ODBMS), represent a specialized category of database systems that store and manage objects directly instead of traditional relational data. These databases are designed to align seamlessly and offer unique advantages in certain application domains.

3. Real-Time System Design

Real-Time System Design is a critical discipline in computer science and engineering that focuses on creating systems capable of responding to external events or inputs within strict timing constraints. Object Oriented Programming techniques can play a significant role in simplifying the complexities associated with designing and implementing real-time systems.

4. Simulation and Modeling Systems

Simulation and Modeling Systems involve the creation of models that mimic real-world processes or systems for various purposes, such as scientific research, engineering, decision-making, and training. Object Oriented Programming offers an alternative and highly effective approach for simplifying the development and management of complex modeling systems.

5. Hypertext and Hypermedia

Hypertext and Hypermedia systems are interactive information systems that enable users to navigate through interconnected documents or multimedia content. Object Oriented Programming provides a robust foundation for creating frameworks and applications for hypertext and hypermedia.

6. Neural Networking and Parallel Programming

Neural Networking and Parallel Programming are two powerful fields of computer science with diverse applications, and Object Oriented Programming can significantly simplify the development and implementation of neural networks, especially in parallel computing environments.

7. Office Automation Systems

Office Automation Systems (OAS) are software applications that streamline and automate various administrative and communication tasks within organizations. These systems facilitate information sharing, improve efficiency, and enhance communication among employees. Object Oriented Programming is instrumental in the development of OAS, as it offers several advantages for creating both formal and informal electronic systems within organizations.

8. CIM/CAD/CAM Systems

Computer-Integrated Manufacturing (CIM), Computer-Aided Design (CAD), and Computer-Aided Manufacturing (CAM) systems are essential tools in manufacturing and design industries. Object Oriented Programming plays a vital role in these systems, simplifying complex tasks such

as blueprint and flowchart design, reducing development effort, and enhancing overall efficiency.

9. AI Expert Systems

AI Expert Systems are computer applications that use specialized knowledge and reasoning capabilities to solve complex problems or provide expertise in specific domains. Object Oriented Programming is a valuable approach in the development of AI Expert Systems, especially when dealing with intricate problems that go beyond human cognitive abilities.

10. E-Commerce Systems

E-Commerce Systems are complex platforms that enable online buying and selling of goods and services. Object Oriented Programming plays a pivotal role in transforming the development of E-Commerce systems by enhancing their scalability, modularity, and overall efficiency.

❖ Java programming

Java is a high-level and purely **object oriented programming language**. It is platform independent, robust, secure, and multithreaded programming language which makes it popular among other OOP languages. It is widely used for software, web, and mobile application development, along with this it is also used in big data analytics and server-side technology. Before moving towards features of Java, let us see how Java originated.

HISTORY:

- In 1990, Sun Microsystems Inc. started developing software for electronic devices. This project was called the Stealth Project (later known as Green Project).

- In 1991, Bill Joy, James Gosling, and Patrick Naughton started working on this project. Gosling decided to use C++ to develop this project, but the main problem he faced is that C++ is platform dependent language and could not work on different electronic device processors.

- As a solution to this problem, Gosling started developing a new language that can be worked over different platforms, and this gave birth to the most popular, platform-independent language known as Oak.

- Yes, you read that right, Oak, this was the first name of Java. But, later it was changed to Java due to copyright issues (some other companies already registered with this name).

- On 23 January 1996, Java's JDK 1.0 version was officially released by Sun

Microsystems. This time the latest release of Java is JDK 20.0 (March 2023).

- Now Java is being used in Web applications, Windows applications, enterprise applications, mobile applications, etc. Every new version of Java comes with some new features.

❖ Features of java:

1. Simple:

Java is a simple programming language and easy to understand because it does not contain complexities that exist in prior programming languages. In fact, simplicity was the design aim of Javasoft people, because it has to work on electronic devices where less memory/resources are available. Java contains the same syntax as C, and C++, so the programmers who are switching to Java will not face any problems in terms of syntax.

2. Object-Oriented:

Java is an Object Oriented Programming Language, which means in Java everything is written in terms of classes and objects. Now, what is an Object? The object is nothing but a real-world entity that can represent any person, place, or thing and can be distinguished from others. Every object near us has some state and behaviour associated with it.

For example, my mobile phone is a real-world entity and has states like colour, model, brand, camera quality, etc, and these properties are represented by variables. Also mobile is associated with actions like, calling, messaging, photography, etc and these actions are represented by methods in Java.

The main concepts of any Object-Oriented Programming language are given below:

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

3. Platform Independent:

The design objective of javasoft people is to develop a language that must work on any platform. Here platform means a type of operating system and hardware technology. Java allows programmers to write their program on any machine with any configuration and to execute it on any other machine having different configurations.

4. Portable:

The WORA (Write Once Run Anywhere) concept and platform-independent feature make Java portable. Now using the Java programming language, developers can yield the same result on any machine, by writing code only once. The reason behind this is JVM and bytecode. Suppose you wrote any code in Java, then that code is first converted to equivalent bytecode which is

only readable by JVM. We have different versions of JVM for different platforms.

5. Robust:

The Java Programming language is robust, which means it is capable of handling unexpected termination of a program. There are 2 reasons behind this, first, it has a most important and helpful feature called Exception Handling. If an exception occurs in java code then no harm will happen whereas, in other low-level languages, the program will crash.

Another reason why Java is strong lies in its memory management features. Unlike other low-level languages, Java provides a runtime Garbage collector offered by JVM, which collects all the unused variables

6. Secure:

In today's era, security is a major concern of every application. As of now, every device is connected to each other using the internet and this opens up the possibility of hacking. And our application built using java also needs some sort of security. So Java also provides security features to the programmers. Security problems like virus threats, tampering, eavesdropping, and impersonation can be handled or minimized using Java. Encryption and Decryption feature to secure your data from *eavesdropping* and *tampering* over the internet. An *Impersonation* is an act of pretending to be another person on the internet.

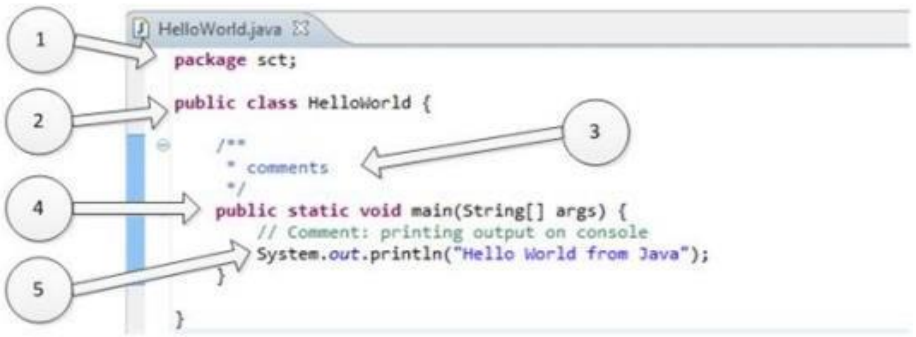
7. Interpreted:

In programming languages, you have learned that they use either the compiler or an interpreter, but Java programming language uses both a compiler and an interpreter. Java programs are compiled to generate bytecode files then JVM interprets the bytecode file during execution. Along with this JVM also uses a JIT compiler (it increases the speed of execution).

8. Multi-Threaded:

Thread is a lightweight and independent subprocess of a running program (i.e, process) that shares resources. And when multiple threads run simultaneously is called multithreading. In many applications, you have seen multiple tasks running simultaneously, for example, Google Docs where while typing text, the spell check and autocorrect tasks are running.

❖ JAVA PROGRAM STRUCTURE:



Let's use example of HelloWorld Java program to understand structure and features of class. This program is written on few lines, and its only task is to print "Hello World from Java" on the screen. Refer the following picture.

1. "packagesct":

It is package declaration statement. The package statement defines a name space in which classes are stored. Package is used to organize the classes based on functionality. If you omit the package statement, the class names are put into the default package, which has no name. Package statement cannot appear anywhere in program. It must be first line of your program or you can omit it.

2. "public class HelloWorld":

This line has various aspects of java programming.

a. **public:** This is access modifier keyword which tells compiler access to class. Various values of access modifiers can be public, protected,private or default (no value).

b. **class:** This keyword used to declare class. Name of class (HelloWorld) followed by this keyword.

3. **Comments section:** We can write comments in java in two ways.

a. **Line comments:** It start with two forward slashes (//) and continue to the end of the current line. Line comments do not require an ending symbol.

b. **Block comments** start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*/).Block comments can also extend across as many lines as needed.

4. "public static void main (String [ ]args)":

Its method (Function) named main with string array as argument.

a. **public :** Access Modifier

b. **static:** static is reserved keyword which means that a method is accessible and usable even though no objects of the class exist.

c. **void:** This keyword declares nothing would be returned from method. Method can return any primitive or object.

d. Method content inside curly braces. { }

5. System.out.println("Hello World from Java") :

a. **System:**It is name of Java utility class.

b. **out:**It is an object which belongs to System class.

c. **println:**It is utility method name which is used to send any String to console.

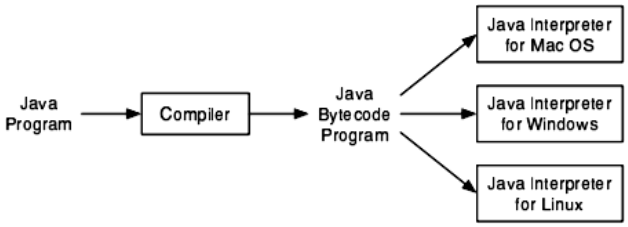
d. **"Hello World from Java":**It is String literal set as argument to println method.

❖ **Java virtual machine:**

The designers of Java chose to use a combination of compilation and interpretation. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the Java virtual machine. The machine language for the Java virtual machine is called Java bytecode. There is no reason why Java bytecode could not be used as the machine language of a real computer, rather than a virtual computer.

However, one of the main selling points of Java is that it can actually be used on any computer. All that the computer needs is an interpreter for Java bytecode.

Such an interpreter simulates the Java virtual machine in the same way that Virtual PC simulates a PC computer. Of course, a different Jave bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program. And the same Java bytecode program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.



What is JRE?

Java Run-time Environment (JRE) is the part of the Java Development Kit (JDK). It is a freely available software distribution which has Java Class Library, specific tools, and a stand-alone JVM. It is the most common environment available on devices to run java programs. The source Java code gets compiled and converted to Java bytecode. If you wish to run this bytecode on any platform, you require JRE. The JRE loads classes, verify access to memory, and retrieves the system resources. JRE acts as a layer on the top of the operating system.

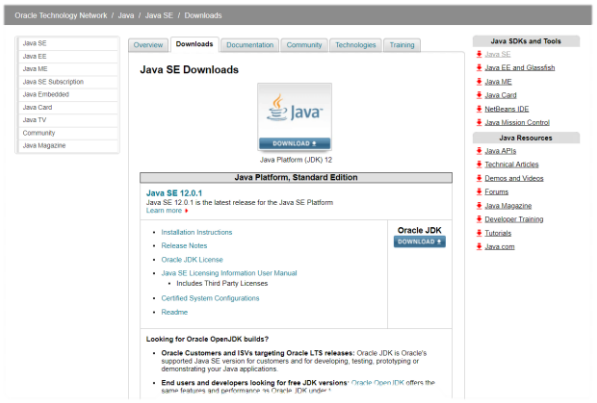
It also includes:

- Technologies which get used for deployment such as Java Web Start.
- Toolkits for user interface like Java 2D.
- Integration libraries like **Java Database Connectivity (JDBC)** and **Java Naming and Directory Interface (JNDI)**.
- Libraries such as Lang and util.

Set up Java JRE with PATH Environment Variables

To develop or run Java applications, you need to download and install the Java SE Development Kit.

Step 1.) Download the Java SE latest release from the official site of the oracle.

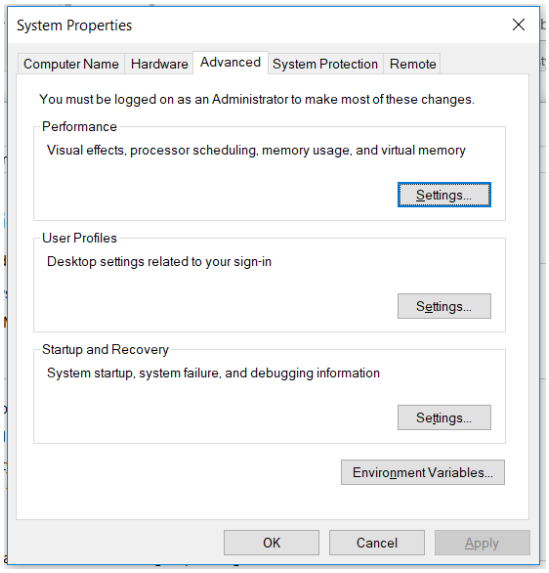


Step 2.) After downloading the file, you will have an executable file downloaded. Run that file and keep everything as default and keep clicking next and then install.

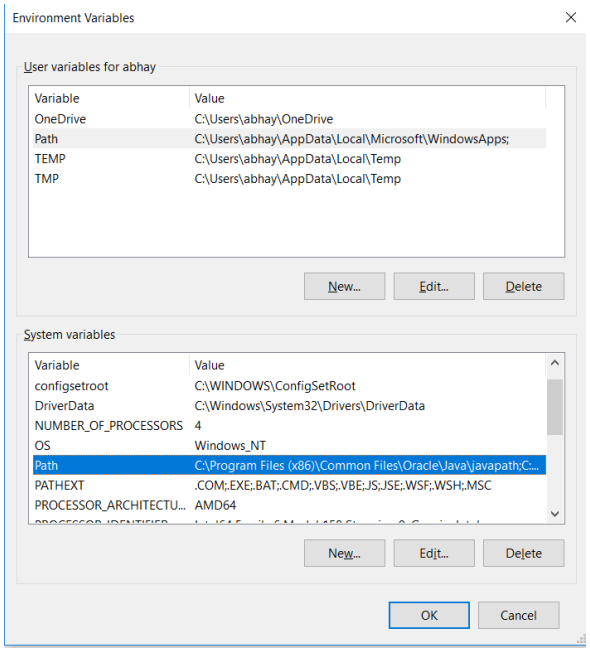
Step 3.) After completing the installation, your JDK and JRE would be downloaded in the program files folder.

Step 4.) After complete installation, you need to set up the environment variables.

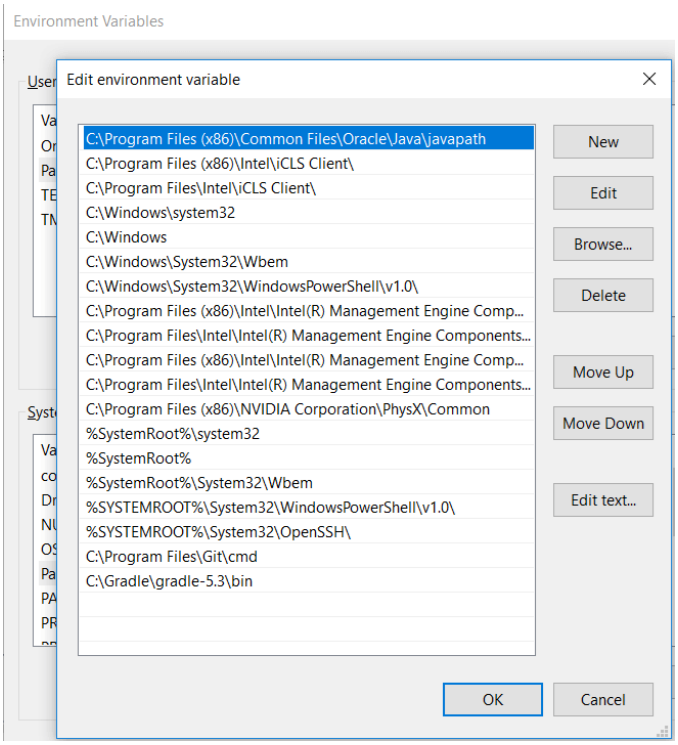
Step 5.) Go to control panel -> System and Security -> System -> Advanced System Settings. The following dialog box will appear.



Step 6.) Click on Environment Variables, go to system variables, and double click on Path.



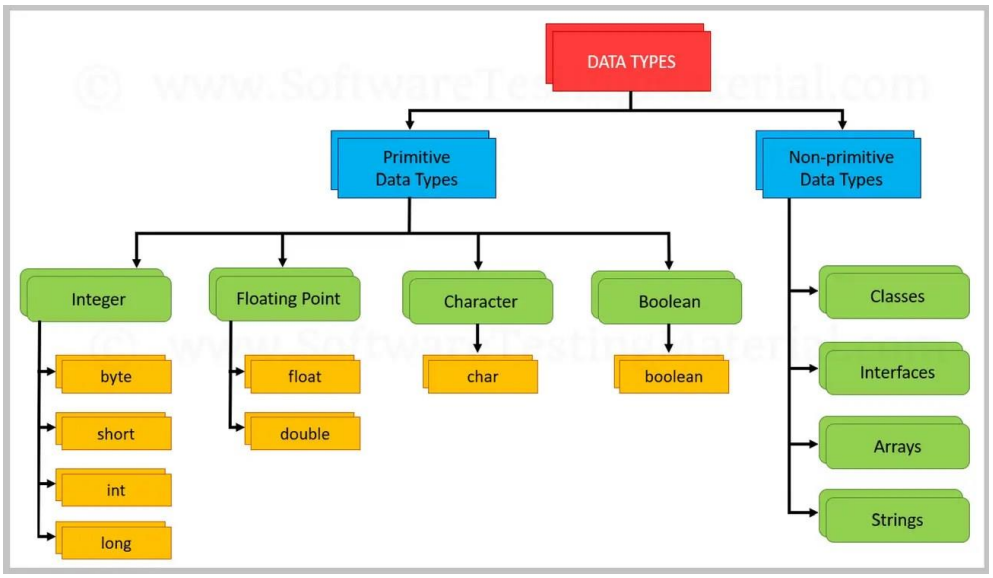
Step 7.) Now add the path of your bin file present in the JRE file to the Path variable.



The set up Java environment is complete.

❖ Datatypes:

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:



A. Primitive Data Types:

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

1. **Boolean:**

- The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.
- The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.
- Default value is false and default size is 1bit.

Example:

Boolean one = false

2. Byte:

- The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.
- The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

byte a = 10, byte b = -20

3. Short:

- The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.
- The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

short s = 10000, short v = -5000

4. **Int:**

- The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is -2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.
- The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

int a = 100000, int b = -200000

5. Long:

- The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0.
- The long data type is used when you need a range of values more than those provided by int.

Example:

long a = 100000L, long b = -200000L

6. Float:

- The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited.
- It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers.
- The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

```
float f1 = 234.5f
```

7. Double:

- The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited.
- The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
double d1 = 12.3
```

8. Char:

- The char data type is a single 16-bit Unicode character.
- Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example:

```
char letterA = 'A'
```

B. **Non-primitive data types:**

Unlike primitive data types, these are not predefined. These are user-defined data types created by programmers. These data types are used to store multiple values.

1.**Class:** A class in Java is a user defined data type i.e. it is created by the user. It acts a template to the data which consists of member variables and methods.

2.**interface:** An interface is similar to a class however the only difference is that its methods are abstract by default i.e. they do not have body. An interface has only the final variables and method declarations. It is also called a fully abstract class.

3.**Array:** An array is a data type which can store multiple homogenous variables i.e., variables of same type in a sequence. They are stored in an indexed manner starting with index 0. The variables can be either primitive or non-primitive data types.

4.**String:** A string represents a sequence of characters for example "Javanotes", Hello world", etc.

String is the class of Java.

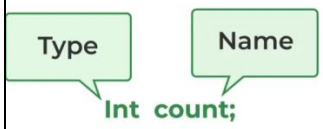
❖ Variables in Java

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.

Declare Variables in Java

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



From the image, it can be easily perceived that while declaring a variable, we need to take care of two things that are:

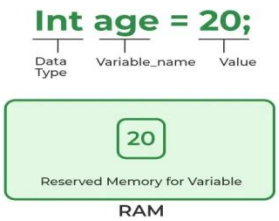
1. **datatype**: Type of data that can be stored in this variable.
2. **data_name:** Name was given to the variable.

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

Initialize Variables in Java

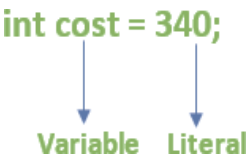
It can be perceived with the help of 3 components that are as follows:



- **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.

❖ **Literals or constant:**

literals are the constant values that appear directly in the program. It can be assigned directly to a variable. Java has various types of literals. The following figure represents a literal.



Types of literals:

There are the majorly following types of literals in Java:

1) **Integer Literal:**

Integer literals are sequences of digits. There are three types of integer literals:

Decimal integer: These are the set of numbers that consist of digits from 0 to 9. It may have a positive (**+**) or negative (-) Note that between numbers commas and non-digit characters are not permitted. For example, **5678,**

+657, -89, etc.

Octal Integer: It is a combination of number have digits from 0 to 7 with a leading 0. For example, **045, 027.**

Hexa-Decimal: The sequence of digits preceded by **0x** or **0X** is considered as hexadecimal integers. It may also include a character

from **a** to **f** or **A** to **F** that represents numbers from **10** to **15**, respectively. For example, **0xd, 0xf,**

Binary Integer: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary

literals in Java SE 7 and later). Prefix 0b represents the Binary system. For example, 0b11010.

2) Character Literal:

A character literal is expressed as a character or an escape sequence, enclosed in a **single** quote (**'**') mark. It is always a type of char. For example, **'a', '%', '\u000d',** etc.

3) String Literal:

String literal is a sequence of characters that is enclosed between **double** quotes ("") marks. It may be alphabet, numbers, special characters, blank space, etc. For example, "**Jack", "12345", "\n",** etc.

Boolean Literal:

Boolean literals are the value that is either true or false. It may also have values 0 and For example**, true, 0,** etc.

4) Floating Point Literal:

The vales that contain decimal are floating literals. In Java, float and double primitive types fall into floating-point literals. Keep in mind while dealing with floating-point literals.Floating-point literals for float type end with F or f. For example, 6f, 8.354F, etc. It is a 32-bit float literal.

Floating-point literals for double type end with D or d. It is optional to write D or d. For example, 6d, 8.354D, etc. It is a 64-bit double literal.

It can also be represented in the form of the exponent.

❖ Scope and Lifetime of a Variable in Java

The *scope* of a variable refers to the areas or the sections of the program in which the variable can be accessed, and the *lifetime* of a variable indicates how long the variable stays alive in the memory.
A joint statement defining the scope and lifetime of a variable is *"how and where the variable is defined."* Let me simplify it further. The comprehensive practice for the *scope* of a variable is that it is accessible only inside the block it is declared.

Types of Variables and its Scope

There are three types of variables.

1. Instance Variables
2. Class Variables
3. Local Variables

Now, let us dig into the scope and lifetime of each of the above mentioned type.

Instance Variables

A variable which is declared inside a class, but is declared outside any methods and blocks is known as *instance* variable.
Scope: Throughout the class except in the static methods.
Lifetime: Until the object of the class stays in the memory.

Class Variables

A variable which is declared inside a class, outside all the blocks and is declared as *static* is known as *class* variable.
Scope: Throughout the class.
Lifetime: Until the end of the program.

Local Variables

All variables which are not *instance* or *class* variables are known as *local* variables.
Scope: Within the block it is declared.
Lifetime: Until control leaves the block in which it is declared.
Now, let us look at an example code to paint a clear picture and understand the concept of scope and lifetime of variables better.

Example

```
public class scope_and_lifetime {

   int num1, num2;   //Instance Variables

   static int result;  //Class Variable

   int add(int a, int b){  //Local Variables

      num1 = a;

      num2 = b;

      return a+b;

   }

   public static void main(String args[]){

      scope_and_lifetime ob = new scope_and_lifetime();

      result = ob.add(10, 20);

      System.out.println("Sum = " + result);

            }

            }
```

❖ Java Operators

Operators in Java, a Java toolkit, are being used as a symbol that performs various operations according to the code. Some Operators of JAVA are "+","-","*","/" etc. The idea of using Operators has been taken from other languages so that it behaves expectedly.

Types of Operators in Java

There are various types of Operators in Java that are used for operating. These are,

1. Arithmetic operators in Java
2. Relational operators in Java
3. Logical operators in Java
4. Assignment operator in Java
5. Unary operator in Java
6. Bitwise operator in Java
7. Comparison operator in Java
8. Ternary operator in Java

Arithmetic Operators in Java

Arithmetic Operators in Java are particularly used for performing arithmetic operations on given data or variables. There are various types of operators in Java, such as

Operators	Operations
+	Addition
-	Subtraction
x	Multiplication
/	Division
%	Modulus

Assignment Operator in Java

Assignment Operators are mainly used to assign the values to the variable that is situated in Java programming. There are various assignment operators in Java, such as

Operators	Examples	Equivalent to
=	X = Y;	X = Y;
+=	X += Y;	X = X + Y;
-=	X -= Y;	X = X - Y;
*=	X *= Y;	X = X * Y;
/=	X /= Y;	X = X / Y;
%=	X %= Y;	X = X % Y;

Relational Operators in Java

Java relational operators are assigned to check the relationships between two particular operators. There are various relational operators in Java, such as

Operators	Description	Example
==	Is equal to	3 == 5 returns false
!=	Not equal to	3 != 5 returns true
>	Greater than	3 > 5 returns false
<	Less than	3 < 5 returns true
>=	Greater than or equal to	3 >= 5 returns false
<=	Less than or equal to	3 <= 5 returns true

Logical Operators in Java

Logical Operators in Java check whether the expression is true or false. It is generally used for making any decisions in Java programming. Not only that but Jump statements in Java are also used for checking whether the expression is true or false. It is generally used for making any decisions in Java programming.

Operators	Example	Meaning
&& [logical AND]	expression1 && expression2	(true) only if both of the expressions are true
|| [logical OR]	expression1 || expression2	(true) if one of the expressions in true
! [logical NOT]	!expression	(true) if the expression is false and vice-versa

Unary Operator in Java

Unary Operators in Java are used in only one operand. There are various types of Unary Operators in Java, such as

Operators	Description
+	Unary Plus
-	Unary Minus
++	Increment operator
--	Decrement Operator
!	Logical complement operator

Bitwise Operators in Java

Bitwise Operators in Java are used to assist the performance of the operations on individual bits. There are various types of Bitwise Operators in Java, such as. We will see the working of the Bitwise Operators in the Java Online Compiler.

Operators	Descriptions
~	Bitwise Complement
<<	Left shift
>>	Right shift

Operators	Descriptions
>>>	Unsigned Right shift
&	Bitwise AND
^	Bitwise exclusive OR

Comparison Operators in Java

To compare two values (or variables), comparison operators are used. This is crucial to programming since it facilitates decision-making and the search for solutions. A comparison's return value is either true or false. These are referred to as "Boolean values."

Operators	Operations
==	Equal to
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

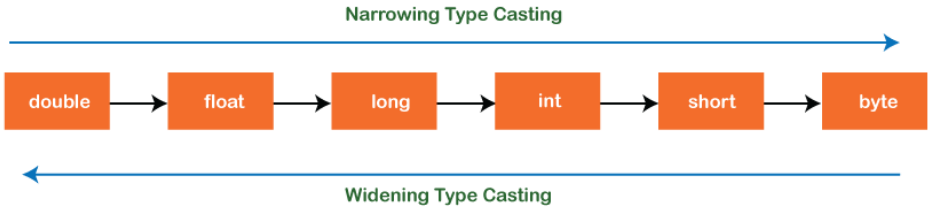
Ternary Operators in Java

The only conditional operator that accepts three operands is the ternary operator in Java. Java programmers frequently use it as a one-line alternative to the if-then-else expression. The ternary operator can be used in place of if-else statements, and it can even be used to create switch statements with nested ternary operators. The conditional operator uses less space and aids in writing if-else statements as quickly as possible even if it adheres to the same algorithm as an if-else statement

```
variable = Expression ? expression1 : expression2
```

❖ **Type Casting in Java**

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and **its types** with proper examples.



Type Casting in Java

Types of Type Casting

There are two types of type casting:

- Widening Type Casting
- Narrowing Type Casting

Widening Type Casting

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

1. **byte** -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
```

```
}
}
```

Output

```
Before conversion, the value is: 7

After conversion, the long value is: 7

After conversion, the float value is: 7.0
```

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

Narrowing Type Casting

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

1. **double** -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

Let's see an example of narrowing type casting.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

Output

```
Before conversion: 166.66

After conversion into long type: 166

After conversion into int type: 166
```

- **Java input**

Java brings various Streams with its I/O package that helps the user perform all the Java input-output operations. These streams support all types of objects, data types, characters, files, etc. to fully execute the I/O operations. Input in Java can be with certain methods mentioned below

Using Scanner class

From Java 1.5 Scanner class was introduced. This class accepts a File, InputStream, Path and, String objects, reads all the primitive data types and Strings (from the given source) token by token using regular expressions. By default, whitespace is considered as the delimiter (to break the data into tokens).

To read data from keyboard you need to use standard input as source (System.in). For each datatype a nextXXX() is provided namely, nextInt(), nextShort(), nextFloat(), nextLong(), nextBigDecimal(), nextBigInteger(), nextLong(), nextShort(), nextDouble(), nextByte(), nextFloat(), next().

Example

Following Java program reads data from user using the Scanner class.

```
import java.util.*;

class Example {

   public static void main(String[] args)

   {

      // Scanner definition

      Scanner scn = new Scanner(System.in);

      // input is a string ( one word )

      // read by next() function

      String str1 = scn.next();

      // print String

      System.out.println("Entered String str1 : " + str1);

      // input is a String ( complete Sentence )
```

```
        // read by nextLine()function
        String str2 = scn.nextLine();
        // print string
        System.out.println("Entered String str2 : " + str2);
        // input is an Integer
        // read by nextInt() function
        int x = scn.nextInt();
        // print integer
        System.out.println("Entered Integer : " + x);

        // input is a floatingValue
        // read by nextFloat() function
        float f = scn.nextFloat();

        // print floating value
        System.out.println("Entered FloatValue : " + f);
    }
}
```


Using BufferedReader

The BufferedReader class of Java is used to read stream of characters from the specified source (character-input stream). The constructor of this class accepts an InputStream object as a parameter, you can to pass an **InputStreamReader**.

Using Console class

This class is used to write/read data from the console (keyboard/screen) devices. It provides a **readLine()** method which reads a line from the key-board. You can get an object of the Console class using the **console()** method.

Note – If you try to execute this program in a non-interactive environment like IDE it doesn't work.

❖ Formatted Output in Java using printf()

Sometimes in programming, it is essential to print the output in a given specified format. Most users are familiar with the printf function in C. Let us discuss how we can Formatting Output with printf() in Java in this article.

Formatting Using Java Printf()

printf() uses format specifiers for formatting. There are certain data types are mentioned below:

- For Number Formatting
- Formatting Decimal Numbers
- For Boolean Formatting
- For String Formatting
- For Char Formatting
- For Date and Time Formatting

i). For Number Formatting

The number itself includes Integer, Long, etc. The formatting Specifier used is %d.

Below is the implementation of the above method:

```
// Java Program to demonstrate
// Use of printf to
// Formatting Integer
import java.io.*;

// Driver Class
class EXAMPLE {
    // main function
   public static void main (String[] args) {
      int a=10000;
```

```
        //System.out.printf("%.d%n",a);
        System.out.printf("%,d%n",a);
    }
}
```

Output

```
10,000
```

ii). For Decimal Number Formatting

Decimal Number Formatting can be done using print() and format specifier %f .

Below is the implementation of the above method:

```
// Java Programs to demonstrate
// Use of Printf() for decimal
// Number Formatting
import java.io.*;

// Driver Class
class EXAMPLE {
    // main function
    public static void main(String[] args)
    {
        // declaring double
        double a = 3.14159265359;

        // Printing Double Value with
        // different Formatting
        System.out.printf("%f\n", a);
        System.out.printf("%5.3f\n", a);
        System.out.printf("%5.2f\n", a);
    }
}
```

Output

```
3.141593

3.142

 3.14
```

iii). For Boolean Formatting

Boolean Formatting can be done using printf and ('%b' or '%B') depending upon the result needed.

Below is the implementation of the above method:

```
// Java Programs to demonstrate
// Use of Printf() for decimal
// Boolean Formatting
import java.io.*;

// Driver Function
class EXAMPLE {
    // main function
    public static void main(String[] args)
    {
        int a = 10;
        Boolean b = true, c = false;
        Integer d = null;

        // Fromatting Done using printf
        System.out.printf("%b\n", a);
        System.out.printf("%B\n", b);
        System.out.printf("%b\n", c);
        System.out.printf("%B\n", d);
    }
}
```

Output

```
true

TRUE

false

FALSE
```

iv). For Char Formatting

Char Formatting is easy to understand as it need printf() and Charracter format specifier used are '%c' and '%C'.

Below is the implementation of the above method:

```
// Java Program to Formatt
//
import java.io.*;

// Driver Class
```

```
class EXAMPLE {
    // main function
    public static void main(String[] args)
    {
        char c = 'g';

        // Formatting Done
        System.out.printf("%c\n", c);

        // Converting into Uppercase
        System.out.printf("%C\n", c);
    }
}
```

Output

```
g

G
```

v). For String Formatting

String Formatting requires the knowledge of Strings and format specifier used '%s' and '%S'.

Below is the implementation of the above method:

```
// Java Program to implement
// Printf() for String Formatting
import java.io.*;

// Driver Class
class EXAMPLE {
    // main function
    public static void main(String[] args)
    {
        String str = "hello java program";

        // Formatting from lowercase to
        // Uppercase
        System.out.printf("%s \n", str);
        System.out.printf("%S \n", str);

        str = "EXAMPLE";
        // Vice-versa not possible
        System.out.printf("%S \n", str);
        System.out.printf("%s \n", str);
    }
}
```

Output

```
hello java program

HELLO JAVA PROGRAM

HEL

HEL
```

❖ Java String format()

The **java string format()** method returns the formatted string by given locale, format and arguments.

If you don't specify the locale in String.format() method, it uses default locale by calling *Locale.getDefault()* method.

The format() method of java language is like *sprintf()* function in c language and *printf()* method of java language.

Internal implementation

```
public static String format(String format, Object... args) {
    return new Formatter().format(format, args).toString();
  }
```

Signature

There are two type of string format() method:

1. **public static** String format(String format, Object... args)
2. and,
3. **public static** String format(Locale locale, String format, Object... args)

Parameters

locale : specifies the locale to be applied on the format() method.

format : format of the string.

args : arguments for the format string. It may be zero or more.

Returns

formatted string

Throws

NullPointerException : if format is null.

IllegalFormatException : if format is illegal or incompatible.

Java String format() method example

```
public class FormatExample{
public static void main(String args[]){
String name="sonoo";
String sf1=String.format("name is %s",name);
String sf2=String.format("value is %f",32.33434);
String sf3=String.format("value is %32.12f",32.33434);//returns 12 char fractional part filling with
 0

System.out.println(sf1);
System.out.println(sf2);
System.out.println(sf3);
}}
```

```
name is sonoo
value is 32.334340
value is     32.334340000000
```

Java String Format Specifiers

Here, we are providing a table of format specifiers supported by the Java String.

Format Specifier	Data Type	Output
%a	floating point (except *BigDecimal*)	Returns Hex output of floating point number.
%b	Any type	"true" if non-null, "false" if null
%c	character	Unicode character
%d	integer (incl. byte, short, int, long, bigint)	Decimal Integer
%e	floating point	decimal number in scientific notation
%f	floating point	decimal number
%g	floating point	decimal number, possibly in scientific notation depending on the precision and value.
%h	any type	Hex String of value from hashCode() method.
%n	none	Platform-specific line separator.
%o	integer (incl. byte, short, int, long, bigint)	Octal number
%s	any type	String value
%t	Date/Time (incl. long, Calendar, Date and TemporalAccessor)	%t is the prefix for Date/Time conversions. More formatting flags are needed after this. See Date/Time conversion below.
%x	integer (incl. byte, short, int, long, bigint)	Hex string.

❖ **Control statements 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
3. Jump statements
   - break statement
   - continue statement

- **Decision Making statements:**

➢ As the name suggests, decision-making statements decide which statement to execute and when.

➢ Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided.

➢ There are two types of decision-making statements in Java, i.e., *If statement* and *switch statement*.

a) If Statement:

➢ The "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition.

➢ The condition of the If statement gives a Boolean value, either true or false.

➢ In Java, there are four types of if-statements given below.

i.Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax:

```
if(condition)
 {
 statement 1; //executes when condition is true
 }
```

```
class Biggrst {
public static void main(String[] args) {
int x = 10;
int y = 5;
if(x<y) {
System.out.println(x+ " is greater value");
```

```
}
}
}
```

Output:

```
10 is greater value
```

ii. **if-else statement:**

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition)
 {
statement 1; //executes when condition is true
 }
 else
{
statement 2; //executes when condition is false
 }
```

```
class EvenOrOdd {
public static void main(String[] args) {
int x = 10;
if(x%2==0) {
System.out.println(x +  "is even");
}  else
System.out.println(x + " is odd");
}
}
```

Output:

```
10 is even
```

iii. **if-else-if ladder:**

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax:

```
if(condition 1)
 {
statement 1; //executes when condition 1 is true
 }
else if(condition 2)
 {
 statement 2; //executes when condition 2 is true
 }
else
{
statement 2; //executes when all the conditions are false
}
```

```
import java.util.Scanner;
class Biggest{
public static void main(String []args){
int a,b,c;
Scanner s=new Scanner(System.in);
System.out.println("enter three numbers");
a=s.nextInt();
b=s.nextInt();
c=s.nextInt();
if((a>b)&&(a>c)){
System.out.print(a+"is biggest");
}
else if(b>c){
System.out.print(b+"is biggest");
}
else
System.out.print(c+"is biggest");
}
}
```

Output:

```
enter three numbers
```

```
30

40

20

20 is biggest
```

iv.**Nested if-statement:**

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax:

```
if(condition 1)
 {
 statement 1; //executes when condition 1 is true
if(condition 2)
{
statement 2; //executes when condition 2 is true
  }
 else
{
statement 2; //executes when condition 2 is false
 }
 }
```

b) Switch statement:

Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java.
- Cases cannot be duplicate.
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.

- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.

Syntax:

```
switch (expression)
{
case value1:
 statement1;
break;
.
.
.
case valueN:
statementN;
break;
default:
 default statement;
 }
```

```
class Test {

  public static void main(String args[]) {

    char grade = 'C';

    switch(grade) {
      case 'A' :
        System.out.println("Excellent!");
        break;
      case 'B' :
      case 'C' :
        System.out.println("Well done");
        break;
      case 'D' :
        System.out.println("You passed");
      case 'F' :
        System.out.println("Better try again");
        break;
      default :
        System.out.println("Invalid grade");
```

```
    }
    System.out.println("Your grade is " + grade);
  }
}
```

Output

```
Well done

Your grade is C
```

- **Loop statements:**

➢ In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true.

➢ However, loop statements are used to execute the set of instructions in a repeated order.

➢ The execution of the set of instructions depends upon a particular condition.

➢ In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

a) for loop:

➢ In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code.

➢ We use the for loop only when we exactly know the number of times, we want to execute the block of code.

Syntax:

```
for(initialization, condition, increment/decrement)
 {
 //block of statements
 }
```

```
public class ForExample {
public static void main(String[] args) {
   for(int i=1;i<=10;i++){
      System.out.print(i+"\t");
   }
}
}
```

Output:

```
1    2        3        4        5        6        7        8        9        10
```

b) while loop:

- The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop.
- Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.
- It is also known as the entry-controlled loop since the condition is checked at the start of the loop.
- If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

Syntax:

```
while(condition)
{
 //looping statements
 }
```

```
public class WhileExample {
public static void main(String[] args) {
int i=10;
   while(i>0){
      System.out.print(i+"\t");
i--;
   }
}
}
```

Output:

```
10   9        8        7        6        5        4        3        2        1
```

c) do-while loop:

- The do-while loop checks the condition at the end of the loop after executing the loop statements.
- When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.
- It is also known as the exit-controlled loop since the condition is not checked in advance.

Syntax:

```
do
 {
//statements
 }
while (condition);
```

```
public class DoWhileExample {
public static void main(String[] args) {
int i=1;
   {
      System.out.print(i+"\t");
i++;
   } while(i<10);
}
}
```

Output:

```
1    2        3        4        5        6        7        8        9        10
```

- **Jump statements:**

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

a) Break statement:

- The break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement.
- However, it breaks only the inner loop in the case of the nested loop.
- The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

Syntax:

```
break;
```

```
public class ForExample {
public static void main(String[] args) {
   for(int i=1;i<=10;i++){
    if(i==5)
```

```
    {
    break;
     }
System.out.print(i+"\t");
   }
}
}
```

Output:

```
1    2        3        4        5
```

b) Continue statement:

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

Syntax:

```
continue;
```

```
public class ForExample {
public static void main(String[] args) {
   for(int i=1;i<=10;i++){
    if((i>4)&& (i<7))
    {
    continue;
     }
System.out.print(i+"\t");
   }
}
}
```

Output:

```
1    2        3        4        7        8        9        10
```

- **Return statement:**

In Java programming, the return statement is used for returning a value when the execution of the block is completed.

Returning a value from a method:

- In Java, every method is declared with a return type such as int, float, double, string, etc.
- These return types required a return statement at the end of the method. A return keyword is used for returning the resulted value.
- The void return type doesn't require any return statement. If we try to return a value from a void method, the compiler shows an error.

Following are the important points must remember while returning a value:

- The return type of the method and type of data returned at the end of the method should be of the same type. For example, if a method is declared with the float return type, the value returned should be of float type only.

Syntax:

return return value;

UNIT-2

❖ Java Arrays (One Dimensional array)

An array is a collection of similar types of data. The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

1. Declare an array in Java:

In Java, here is how we can declare an array.

dataType[] arrayName;

dataType - it can be primitive data types like int, char, double, byte, etc.

arrayName - it is an identifier

For example,

double[] data;

Here, data is an array that can hold values of type double.

2. Memory allocation:

we have to allocate memory for the array in Java. Memory representation. When you use new to create an array, Java reserves space in memory for it (and initializes the values). This process is called memory allocation.

For example,

double[] data;

data = new double[10];

Here, the array can store 10 elements. We can also say that the size or length of the array is 10.

In Java, we can declare and allocate the memory of an array in one single statement. For example,

double[] data = new double[10];

3. Initialize Arrays in Java:

In Java, we can initialize arrays during declaration. For example,

int[] age = {12, 4, 5, 2, 5};

Here, we have created an array named age and initialized it with the values inside the curly brackets.

Note that we have not provided the size of the array. In this case, the Java compiler automatically specifies the size by counting the number of elements in the array

In the Java array, each memory location is associated with a number. The number is known as an array index.

Array indices always start from 0. That is, the first element of an array is at index 0.

If the size of an array is n, then the last element of the array will be at index n-1.

4. Access Elements of an Array:

We can use loops to access all the elements of the array at once.

Looping Through Array Elements:

In Java, we can also loop through each element of the array. For example,

Example: Using For Loop

```
class Main {
public static void main(String[] args) {
int[] age = {12, 4, 5};
System.out.println("Using for Loop:");
```

```
for(int i = 0; i < age.length; i++) {
System.out.println(age[i]);
}
}
}
```

Output

Using for Loop:
12
4
5

In the above example, we are using the for Loop in Java to iterate through each element of the array. Notice the expression inside the loop, age.length
Here, we are using the length property of the array to get the size of the array.

- **Types of Array In Java, there are two types of arrays:**

1. Single-Dimensional Array
2. Multi-Dimensional Array

Single-Dimensional Array:

- An array that has only one subscript or one dimension is known as a single-dimensional array. It is just a list of the same data type variables.
- One dimensional array can be of either one row and multiple columns or multiple rows and one column.
- The declaration and initialization of an single-dimensional array is same as array's initialization and declaration.

Example:
int marks[] = {56, 98, 77, 89, 99};

Multi-Dimensional Array:

A multi-dimensional array is just an array of arrays that represents multiple rows and columns.

In multi-dimensional arrays, we have two categories:
i. Two-Dimensional Arrays
ii. Three-Dimensional Arrays

i. Two-Dimensional Arrays:

An array involving two subscripts [] [] is known as a two-dimensional array. They are also known as the array of the array. Two-dimensional arrays are divided into rows and columns and are able to handle the data of the table.

Syntax:
DataType ArrayName[row_size][column_size];
Example:
int arr[5][5];

ii. Three-Dimensional Arrays:

- When we require to create two or more tables of the elements to declare the array elements, then in such a situation we use three-dimensional arrays.
- 3D array adds an extra dimension to the 2D array to increase the amount of space.

- Eventually, it is set of the 2D array. Each variable is identified with three indexes; the last two dimensions represent the number of rows and columns, and the first dimension is to select the block size.
- The first dimension depicts the number of tables or arrays that the 3D array contains.

Syntax:
DataType ArrayName[size1][size2][size3];
Example:
int a[5][5][5];

❖ Strings

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object. An array of characters works same as Java string.

For example:

```
char[] ch={'c','o','m','p','u','t','e','r'};
String s=new String(ch); is same as:
String s="computer";
```

Creating Strings:
There are two ways to create String object:
1. By string literal
2. By new keyword

1) By string literal:
Java String literal is created by using double quotes.

- String objects are stored in a special memory area known as the "string constant pool".
- Each time you create a string literal, the JVM checks the "string constant pool" first.
- If the string already exists in the pool, a reference to the pooled instance is returned.
- If the string doesn't exist in the pool, a new string instance is created and placed in the pool.
- The string literal concept is used to make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

Syntax:
<String_Type><string_variable> = "<sequence_of_string>";
Example:
String s="welcome";

2) By new keyword:
String s=new String("Welcome"); //creates two objects and one reference variable

- In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool.
- The variable s will refer to the object in a heap (non-pool).

Program:

```
public class StringExample {
 public static void main(String args[]) {
 String s1="java";
char ch[]={'s','t','r','i','n','g','s'};
```

```
String s2=new String(ch);
String s3=new String("example");
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
 }
 }
```

Output:
java
strings
example

❖ String class methods

- The **java.lang.String** class provides a lot of built-in methods that are used to manipulate **string in Java**.
- By the help of these methods, we can perform operations on String objects such as trimming, concatenating, converting, comparing, replacing strings etc.
- Java String is a powerful concept because everything is treated as a String if you submit any form in window based, web based or mobile application.

Some of the important methods are:

1) Java String toUpperCase() and toLowerCase() method:

The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.

Program:

```
public class Test {
public static void main(String args[]) {
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//Sachin
System.out.println(s);//Sachin(no change in original)
}
}
```

2) Java String trim() method:

The String class trim() method eliminates white spaces before and after the String.

Program:

```
public class Test2{
public static void main(String args[]){
String s=" Sachin ";
System.out.println(s);// Sachin
System.out.println(s.trim());//Sachin
}
}
```

Output:

Sachin Sachin

3) Java String startsWith() and endsWith() method:

The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.

Program:

```
public class Test3{
public static void main(String args[]){
String s="Sachin";
System.out.println(s.startsWith("Sa"));//true
System.out.println(s.endsWith("n"));//true
}
}
```

Output:

true true

4) Java String charAt() Method:

The String class charAt() method returns a character at specified index.

Program:

```
public class Test4{
public static void main(String args[]) {
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
```

```
}
}
```

Output:

S

h

5) Java String length() Method:

The String class length() method returns length of the specified String.

Program:

```
public class Test5{
public static void main(String args[]){
String s="Sachin";
System.out.println(s.length());//6
}
}
```

Output:

6

6) Java String valueOf() Method:

The String class valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into String.

Program:

```
public class Test6{
public static void main(String args[]){
int a=10;
String s=String.valueOf(a);
System.out.println(s+10);
}
}
```

Output:

1010

7) Java String replace() Method:

The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.

Program:

```
public class Test7
{
public static void main(String ar[])
{
String s1="Java is a programming language. Java is a platform. Java is an Island.";
String replaceString=s1.replace("Java","Oops");
//replaces all occurrences of "Java" to "Oops"
System.out.println(replaceString);
}
}
```

Output:

Oops is a programming language.Oops is a platform. Oops is an Island.

❖ Command line arguments

- Java command-line argument is an argument i.e. passed at the time of running the Java program.
- In the command line, the arguments passed from the console can be received in the java program and they can be used as input.
- The users can pass the arguments during the execution bypassing the command- line arguments inside the main() method.We need to pass the arguments as space-separated values.
- We can pass both strings and primitive data types(int, double, float, char, etc) as command-line arguments.
- These arguments convert into a string array and are provided to the main() function as a string array argument.
- When command-line arguments are supplied to JVM, JVM wraps these and supplies them to args[].
- It can be confirmed that they are wrapped up in an args array by checking the length of args using args.length.
- Internally, JVM wraps up these command-line arguments into the args[ ] array that we pass into the main() function.
- We can check these arguments using args.length method. JVM stores the

first command-line argument at args[0], the second at args[1], the third at args[2], and so on

```
public class CommandLine {

public static void main(String[] args) {

   System.out.println("No of arguments are "+args.length);

   System.out.println("arguments are ");

   for (int i = 0; i < args.length; i++) {

      System.out.println(args[i]);

         }

}

}
```

Output

```
C:\java programs\data science\second> javac CommandLine.java
C:\java programs\data science\second> java CommandLine hello java program
No of arguments are 3
arguments are
hello
java
program
```

❖ classes and objects

The **classes and objects** are the basic and important features of object-oriented programming system, Java supports the following fundamental OOPs concepts

Classes, Objects, Inheritance, Polymorphism, Encapsulation, Abstraction, Instance, Method, Message Passing

In this tutorial, we will learn about Java Classes and Objects, the creation of the classes and objects, accessing class methods, etc.

✓ Java Classes

A **class** is a blueprint from which individual objects are created (or, we can say a class is a data type of an object type). In Java, everything is related to classes and objects. Each class has its methods and attributes that can be accessed and manipulated through the objects.

For example, if you want to create a class for *students*. In that case, "*Student*" will be a class, and student records (like *student1*, *student2*, etc) will be objects.

We can also consider that class is a factory (user-defined blueprint) to produce objects.

Properties of Java Classes

- A class does not take any byte of memory.
- A class is just like a real-world entity, but it is not a real-world entity. It's a blueprint where we specify the functionalities.
- A class contains mainly two things: Methods and Data Members.
- A class can also be a nested class.
- Classes follow all of the rules of OOPs such as inheritance, encapsulation, abstraction, etc.

Types of Class Variables

A class can contain any of the following variable types.

- **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

Creating (Declaring) a Java Class

To create (declare) a class, you need to use *access modifiers* followed by **class** keyword and *class_name*.

Syntax to create a Java class

Use the below syntax to create (declare) class in Java:

```
access_modifier class class_name{
  data members;
  constructors;
  methods;
  ...;
}
```

Example of a Java Class

In this example, we are creating a class "**Dog**". Where, the class attributes are **breed**, **age**, and **color**. The class methods are **setBreed()**, **setAge()**, **setColor()**, and **printDetails()**.

```
// Creating a Java class
class Dog {
  // Declaring and initializing the attributes
  String breed;
  int age;
  String color;
  // methods to set breed, age, and color of the dog
  public void setBreed(String breed) {
    this.breed = breed;
  }
  public void setAge(int age) {
    this.age = age;
  }
  public void setColor(String color) {
    this.color = color;
  }

  // method to print all three values
  public void printDetails() {
    System.out.println("Dog detials:");
    System.out.println(this.breed);
    System.out.println(this.age);
    System.out.println(this.color);
  }
}
```

- ✓ Java Object

An **object** is a variable of the type **class**, it is a basic component of an object-oriented programming system. A class has the methods and data members (attributes), these methods and data members are accessed through an **object**. Thus, an object is an instance of a class.

If we consider the real world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, and color, and the behavior is - barking, wagging the tail, and running.

Creating (Declaring) a Java Object

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the **new** keyword is used to create new objects.

There are three steps when creating an object from a class –

- **Declaration** – A variable declaration with a variable name with an object type.
- **Instantiation** – The 'new' keyword is used to create the object.
- **Initialization** – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Syntax to Create a Java Object

Consider the below syntax to create an object of the class in Java:

```
Class_name object_name = new Class_name([parameters]);
```

Example to Create a Java Object

In this example, we are creating an object named **obj** of **Dog** class and accessing its methods.

```
// Creating a Java class
class Dog {
    // Declaring and initializing the attributes
    String breed;
    int age;
    String color;
    // methods to set breed, age, and color of the dog
    public void setBreed(String breed) {
      this.breed = breed;
    }
    public void setAge(int age) {
      this.age = age;
    }
    public void setColor(String color) {
      this.color = color;
    }
    // method to print all three values
    public void printDetails() {
      System.out.println("Dog detials:");
      System.out.println(this.breed);
      System.out.println(this.age);
      System.out.println(this.color);
    }
  }

  public class Main {
    public static void main(String[] args) {
      // Creating an object of the class Dog
      Dog obj = new Dog();

      // setting the attributes
      obj.setBreed("Golden Retriever");
      obj.setAge(2);
      obj.setColor("Golden");
```

```
        // Printing values
        obj.printDetails();
    }
}
```

Output

Dog detials:
Golden Retriever
2
Golden

❖ Java static keyword

The static keyword in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Understanding the problem without static variable

```
class Student{
    int rollno;
    String name;
    String college="ADITYA";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

> Java static property is shared to all objects.

Example

```
//Java Program to demonstrate the use of static variable
class Student{
  int rollno;//instance variable
```

```
  String name;
  static String college ="ADITYA DEGREE COLLEGE";//static variable
  //constructor
  Student(int r, String n){
  rollno = r;
  name = n;
  }
  //method to display the values
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Rama");
 Student s2 = new Student(222,"Manga");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

Output:

```
111 Rama ADITYA DEGREE COLLEGE
222 Manga ADITYA DEGREE COLLEGE
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "ADC";
```

```
    //static method to change the value of static variable
    static void change(){
    college = "AWDC";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
   //creating objects
   Student s1 = new Student(111,"Rama");
   Student s2 = new Student(222,"Manga");
   Student s3 = new Student(333,"Vasavi");
   //calling display method
   s1.display();
   s2.display();
   s3.display();
   }
}
```

Output: 111 Rama AWDC
222 Manga AWDC
333 Vasavi AWDC

❖ Java this Keyword

In Java, this keyword is used to refer to the current object inside a method or a constructor.

Use of this Keyword

In Java, it is not allowed to declare two or more variables having the same name inside a scope (class scope or method scope). However, instance variables and parameters may have the same name.

For example,

```
class MyClass {
    // instance variable
    int age;

    // parameter
    MyClass(int age){
        age = age;
    }
}
```

In the above program, the instance variable and the parameter have the same name: age.

Here, the Java compiler is confused due to name ambiguity.

In such a situation, we use this keyword. For example,

First, let's see an example without using this keyword:

```
class Main {
    int age;
    Main(int age){
        age = age;
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("obj.age = " + obj.age);
    }
}
```

Output:

```
obj.age = 0
```

In the above example, we have passed 8 as a value to the constructor. However, we are getting 0 as an output. This is because the Java compiler gets confused because of the ambiguity in names between instance the variable and the parameter.

Now, let's rewrite the above code using this keyword.

```
class Main {
    int age;
    Main(int age){
        this.age = age;
```

```
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("obj.age = " + obj.age);
    }
}
```

Output:

```
obj.age = 8
```

Now, we are getting the expected output. It is because when the constructor is called, `this` inside the constructor is replaced by the object *obj* that has called the constructor. Hence the *age* variable is assigned value *8*

❖ Parameters passing

Information can be passed to methods as a parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a String called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

```
public class Main {
  static void myMethod(String fname) {
    System.out.println(fname + " Refsnes");
  }
  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
```

output

```
Liam Refsnes
Jenny Refsnes
Anja Refsnes
```

❖ Java Constructors

Java constructors are special types of methods that are used to initialize an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

All classes have constructors, whether you define one or not because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your constructor, the default constructor is no longer used.

Rules for Creating Java Constructors

- You must follow the below-given rules while creating Java constructors:
- The name of the constructors must be the same as the class name.
- Java constructors do not have a return type. Even do not use void as a return type.
- There can be multiple constructors in the same class, this concept is known as constructor overloading.
- The access modifiers can be used with the constructors, use if you want to change the visibility/accessibility of constructors.
- Java provides a default constructor that is invoked during the time of object creation. If you create any type of constructor, the default constructor (provided by Java) is not invoked.

Creating a Java Constructor

To create a constructor in Java, simply write the constructor's name (that is the same as the class name) followed by the brackets and then write the constructor's body inside the curly braces ({}).

Syntax

Following is the syntax of a constructor −

```
class ClassName {
  ClassName() {
  }
}
```

Example to create a Java Constructor

The following example creates a simple constructor that will print "Hello world".

```
public class Main {
 // Creating a constructor
 Main() {
   System.out.println("Hello, World!");
 }

 public static void main(String[] args) {
   System.out.println("The main() method.");

   // Creating a class's object
   // that will invoke the constructor
```

```
    Main obj_x = new Main();
  }
}
```

Output:
The main() method.
Hello, World!

Types of Java Constructors

There are three different types of constructors in Java, we have listed them as follows:

1. Default Constructor
2. No-Args Constructor
3. Parameterized Constructor

1. Default Constructor

If you do not create any constructor in the class, Java provides a default constructor that initializes the object.

Example: Default Constructor (A Class Without Any Constructor)

In this example, there is no constructor defined by us. The default constructor is there to initialize the object.

```
public class Main {
  int num1;
  int num2;

  public static void main(String[] args) {
    // We didn't created any structure
    // a default constructor will invoke here
    Main obj_x = new Main();

    // Printing the values
    System.out.println("num1 : " + obj_x.num1);
    System.out.println("num2 : " + obj_x.num2);
  }
}
```

Output
num1 : 0
num2 : 0

2. No-Args (No Argument) Constructor

As the name specifies, the No-argument constructor does not accept any argument. By using the No-Args constructor you can initialize the class data members and perform various activities that you want on object creation.

Example: No-Args Constructor

This example creates no-args constructor.

```
public class Main {
  int num1;
  int num2;
```

```
  // Creating no-args constructor
  Main() {
    num1 = 16;
    num2 = 15;
  }

  public static void main(String[] args) {
    // no-args constructor will invoke
    Main obj_x = new Main();

    // Printing the values
    System.out.println("num1 : " + obj_x.num1);
    System.out.println("num2 : " + obj_x.num2);
  }
}
```

Output

num1 : 16

num2 : 15

3. Parameterized Constructor

A constructor with one or more arguments is called a parameterized constructor.

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example 1: Parameterized Constructor

This example creates a parameterized constructor.

```
public class Main {
  int num1;
  int num2;

  // Creating parameterized constructor
  Main(int a, int b) {
    num1 = a;
    num2 = b;
  }

  public static void main(String[] args) {
    // Creating two objects by passing the values
    // to initialize the attributes.
    // parameterized constructor will invoke
    Main obj_x = new Main(10, 20);
    Main obj_y = new Main(100, 200);

    // Printing the objects values
    System.out.println("obj_x");
    System.out.println("num1 : " + obj_x.num1);
    System.out.println("num2 : " + obj_x.num2);
```

```
    System.out.println("obj_y");
    System.out.println("num1 : " + obj_y.num1);
    System.out.println("num2 : " + obj_y.num2);
  }
}
```

Output

```
obj_x
num1 : 10
num2 : 20
obj_y
num1 : 100
num2 : 200
```

❖ 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.

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

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.

```
class Adder{
static int add(int a,int b){
return a+b;
}
static int add(int a,int b,int c){
return a+b+c;
}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
```

```
System.out.println(Adder.add(11,11,11));
}}
```

Output:

```
22
33
```

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.

```
class Adder{
static int add(int a, int b){
return a+b;
}
static double add(double a, double b){
return a+b;
}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

```
22
24.9
```

❖ Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviours 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.

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
}
```

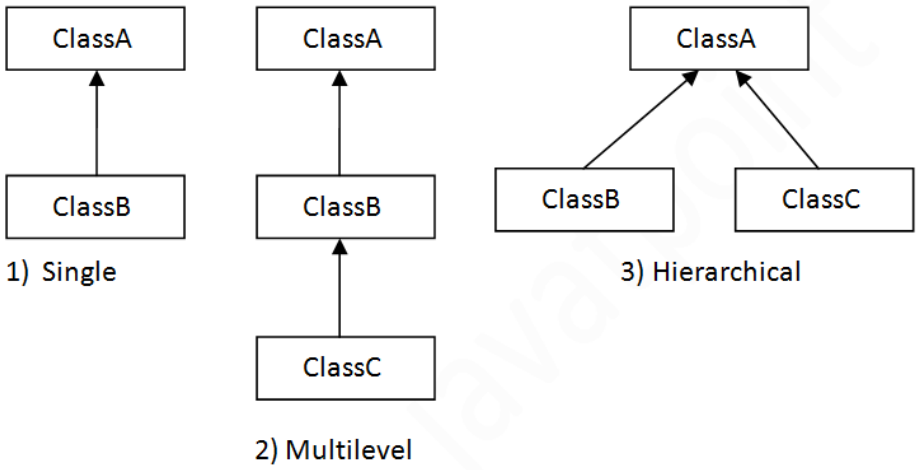
The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Types of inheritance in java

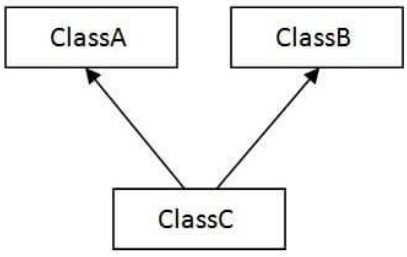
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

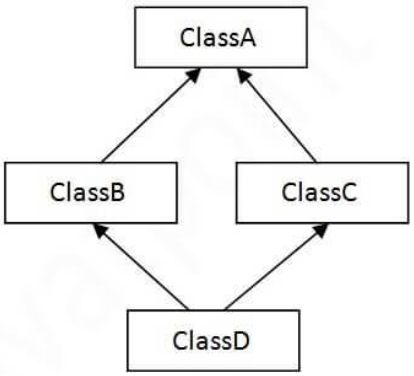


Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

.

Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:

```
barking...
eating...
```

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

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

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

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
```

```
c.eat();
//c.bark();
}}
```

Output:

```
meowing...
eating...
```

❖ Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

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

1) Private

The private access modifier is accessible only within the 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 a 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
   }
}
```

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

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

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.

It provides more accessibility than the default modifer.

Example of protected access modifier

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

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

Example of public access modifier

```
//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
```

❖ Super Keyword in Java

The **super keyword in Java** is a reference variable that is used to refer to parent class when we're working with objects. You need to know the basics of Inheritanceand Polymorphism to understand the Java super keyword.
The Keyword **"super"** came into the picture with the concept of Inheritance

1. Use of super with Variables

This scenario occurs when a derived class and base class have the same data members. In that case, there is a possibility of ambiguity r the JVM.
We can understand it more clearly using the following example:

Example

```
// super keyword in java example
  // Base class vehicle
class Vehicle {
    int maxSpeed = 120;
}
  // sub class Car extending vehicle
class Car extends Vehicle {
    int maxSpeed = 180;
```

```
    void display()
    {
        // print maxSpeed of base class (vehicle)
        System.out.println("Maximum Speed: "+ super.maxSpeed);
    }
}

// Driver Program
class Test {
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```

Output

```
Maximum Speed: 120
```

In the above example, both the base class and subclass have a member maxSpeed. We could access the maxSpeed of the base class in subclass using super keyword.

2. Use of super with Methods

This is used when we want to call the parent class method. So whenever a parent and child class have the same-named methods then to resolve ambiguity we use the super keyword. This code snippet helps to understand the said usage of the super keyword.

Example

```
// super keyword in java example
// superclass Person
class Person {
    void message()
    {
        System.out.println("This is person class\n");
    }
}
// Subclass Student
class Student extends Person {
    void message()
    {
        System.out.println("This is student class");
    }
    // Note that display() is
    // only in Student class
    void display()
    {
```

```
        // will invoke or call current
        // class message() method
        message();

        // will invoke or call parent
        // class message() method
        super.message();
    }
}
// Driver Program
class Test {
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

Output

```
This is student class

This is person class
```

In the above example, we have seen that if we only call method **message()** then, the current class message() is invoked but with the use of the super keyword, **message()** of the superclass could also be invoked.

❖ Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}
```

Output:Compile Time Error

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Output:Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output:Compile Time Error

❖ 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

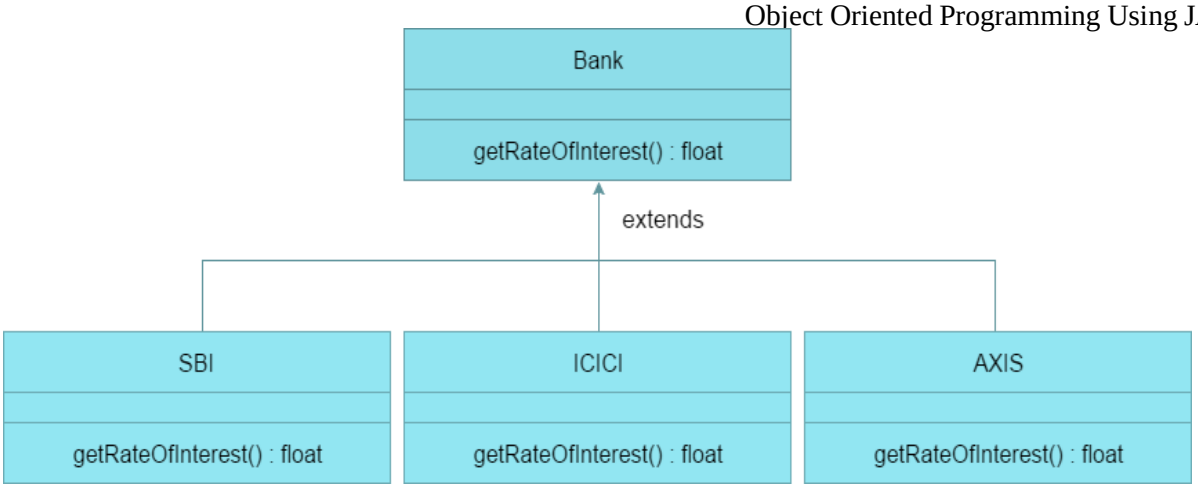
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

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).

Real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.




```
class Bank{
int getRateOfInterest(){return 0;}
}
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

Output:

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

❖ Dynamic binding

Dynamic binding in Java refers to the process of determining the specific implementation of a method at runtime, based on the actual type of the object being referred to, rather than the declared type of the reference variable. It is also known as late binding or runtime polymorphism.

In Java, dynamic binding is primarily associated with method overriding, where a subclass provides its own implementation of a method that is already defined in its superclass. The specific method implementation to be executed is determined dynamically based on the actual type of the object at runtime.
Here is an example of dynamic binding in java:

```java
class Shape {
    public void draw() {
        System.out.println("Drawing a shape");
    }
}

class Circle extends Shape {
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

class Rectangle extends Shape {
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape1 = new Circle();
        Shape shape2 = new Rectangle();

        shape1.draw(); // Output: Drawing a circle
        shape2.draw(); // Output: Drawing a rectangle
    }
}
```

In this example, we have a Shape class and two subclasses: Circle and Rectangle. Each subclass overrides the draw() method inherited from the Shape class.

In the main method, we create two shape objects: shape1 of type Circle and shape2 of type Rectangle. Although the reference type is Shape, the actual objects being referred to are Circle and Rectangle.

When we call the draw() method on shape1 and shape2, the JVM dynamically binds the appropriate version of the draw() method at runtime based on the actual type of the object.

Since shape1 refers to a Circle object, the draw() method in the Circle class is invoked, and the output is "Drawing a circle". Similarly, since shape2 refers to a Rectangle object, the draw() method in the Rectangle class is invoked, and the output is "Drawing a rectangle".

This dynamic binding allows us to write code that works with a generic reference type (Shape), but at runtime, the correct method implementation is determined based on the actual object type being referred to. This flexibility and polymorphic behaviour is one of the key advantages of dynamic binding in Java.

❖ Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.
Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

1) Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.

Example of abstract class

```
abstract class A{
}
```

2) Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an

abstract method.

Example of abstract method

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract class

```
abstract class Animal {
  abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}

class Dog extends Animal {
  public void makeSound() {
    System.out.println("Bow.. Bow..");
  }
}
class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.makeSound();
    d1.eat();
  }
}
```

Output:
Bow.. Bow..
I can eat.

UNIT-3

❖ Interfaces vs. Abstract Classes

Aspect	Interface	Abstract Class
Inheritance	Multiple interfaces can be implemented	Single class inheritance
Methods	No implementation (except defaults)	Can have both abstract and concrete
Fields	Only constants (static and final)	Can have instance variables
Constructors	No constructors	Can have constructors
Purpose	Define a contract for behavior	Base class for related subclasses
Use Case	Unrelated classes implementing similar behavior	Closely related classes sharing code
Access Modifiers	Methods are implicitly public	Can have private, protected, or public
Performance	Might be slower due to runtime indirection	Generally faster

Example Usage

Interface Example (Java)

```
public interface Flyable {
    void fly();
}
class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird is flying.");
    }
}

class Plane implements Flyable {
    @Override
    public void fly() {
        System.out.println("Plane is flying.");
    }
}
class FlyableExample{
    public static void main(String[] args) {
        Flyable bird = new Bird();
        Flyable plan = new Plane();
        bird.fly();
```

```
        plan.fly();
    }
}
```

Output:
Bird is flying.
Plane is flying.

Abstract Class Example (Java)

```
public abstract class Animal {
    abstract void makeSound(); // Abstract method

    public void sleep() { // Concrete method
        System.out.println("Sleeping...");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}
class AnimalExample{
    public static void main(String[] args) {
        Animal cat= new Cat();
        Animal dog = new Dog();
        cat.makeSound();
        cat.sleep();
        dog.makeSound();
        dog.sleep();
    }
}
```

Output:
Meow
Sleeping...

Bark
Sleeping...

In the interface example, Bird and Plane both implement the ability to fly(), but are unrelated. In the abstract class example, Dog and Cat are related because they share common behavior such as sleep(), but must implement their own version of makeSound().

❖ Interfaces

1. Defining an Interface

An **interface** is essentially a contract that specifies what methods a class must implement, without dictating how they should be implemented.

Syntax (Java Example):

```
public interface Vehicle {
    void start();  // abstract method
    void stop();
}
```

In this example, the Vehicle interface defines two methods, start() and stop(). Any class that implements Vehicle must provide concrete implementations for these methods.

- Interfaces only contain **method signatures** (and optionally, constants).
- Methods in an interface are implicitly public and abstract.
- Since Java 8, interfaces can also include **default methods** with a body (an implementation).

2. Implementing Interfaces

When a class implements an interface, it must provide implementations for all the methods declared in the interface.

Syntax (Java Example):

```
public class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car is starting.");
    }

    @Override
```

```
    public void stop() {
        System.out.println("Car is stopping.");
    }
}
```

In this example, the Car class implements the Vehicle interface and provides implementations for the start() and stop() methods.

- A class uses the implements keyword to implement an interface.
- All methods of the interface **must be implemented** unless the class is abstract.

3. Accessing Implementations Through Interface References

Once a class implements an interface, you can reference objects of that class via the interface. This is key for **polymorphism**, as the interface provides a common reference type.

Syntax (Java Example):

```
public class VehicleExample {
    public static void main(String[] args) {
    Vehicle myCar = new Car();
     myCar.start();  // Calls the start() method of Car
     myCar.stop();   // Calls the stop() method of Car

    }
}
```

```
Car is starting.
Car is stopping.
```

- The myCar variable is declared as a Vehicle (the interface), but it holds an instance of Car (a class that implements Vehicle).
- This enables you to swap out different implementations of Vehicle without changing the code that interacts with Vehicle.

Polymorphism Example:

```
Vehicle myBike = new Bike();
Vehicle myTruck = new Truck();

myBike.start();
myTruck.start();
```

Here, both Bike and Truck implement Vehicle. Using polymorphism, the same start() method is called on different objects, with different behaviors based on the class.

4. Extending Interfaces

An interface can extend another interface, allowing for more complex and hierarchical behavior definitions. When an interface extends another interface, it inherits all the methods from the parent interface.

Syntax (Java Example):

```
public interface ElectricVehicle extends Vehicle {
   void chargeBattery();
}
```

Here, ElectricVehicle extends the Vehicle interface, meaning any class that implements ElectricVehicle must implement both start(), stop(), and chargeBattery().

Implementing the Extended Interface:

```
public class Tesla implements ElectricVehicle {
   @Override
   public void start() {
      System.out.println("Tesla is starting.");
   }

   @Override
   public void stop() {
      System.out.println("Tesla is stopping.");
   }

   @Override
   public void chargeBattery() {
      System.out.println("Tesla is charging.");
   }
}
```

```
public class VehicleExample {
   public static void main(String[] args) {
   ElectricVehicle myCar = new Tesla();
    myCar.start();  // Calls the start() method of Car
    myCar.stop();   // Calls the stop() method of Car
    myCar.chargeBattery();

   }
}
```

Tesla is starting.
Tesla is stopping.
Tesla is charging.

Now, the Tesla class implements ElectricVehicle, so it must implement all the methods from both the Vehicle interface (start() and stop()) and the ElectricVehicle interface (chargeBattery()).

5. Interface Inheritance: Example

You can use interfaces and inheritance together for multiple levels of abstraction:

```
public interface WaterVehicle extends Vehicle {
   void anchor();
}

   class Boat implements WaterVehicle {
   @Override
   public void start() {
      System.out.println("Boat is starting.");
   }

   @Override
   public void stop() {
      System.out.println("Boat is stopping.");
   }

   @Override
   public void anchor() {
      System.out.println("Boat is anchoring.");
   }
}
```

```
public class VehicleExample {
   public static void main(String[] args) {
      WaterVehicle wv = new Boat();
      wv.start();
      wv.stop();
      wv.anchor();

   }
}
```

```
Boat is starting.
Boat is stopping.
Boat is anchoring.
```

Here, Boat implements WaterVehicle, which means it must implement methods from both Vehicle (start(), stop()) and WaterVehicle (anchor()).

Key Concepts:

1. **Defining an Interface**: Use `interface` keyword; specify abstract methods.
   - Example: `public interface Vehicle { void start(); void stop(); }`
2. **Implementing an Interface**: Use `implements` keyword in a class to define the body of methods from the interface.
   - Example: `public class Car implements Vehicle { @Override void start() { } @Override void stop() { } }`
3. **Accessing Implementations Through Interface References**: Use the interface as a reference type to call methods from an implemented class.
   - Example: `Vehicle myCar = new Car(); myCar.start();`
4. **Extending Interfaces**: An interface can extend another interface to inherit its methods.
   - Example: `public interface ElectricVehicle extends Vehicle { void chargeBattery(); }`

❖ Packages

packages are used to group related classes, interfaces, and sub-packages, helping organize code and avoid naming conflicts.

1. Defining a Package

A **package** in Java is a namespace that organizes classes and interfaces. Packages allow developers to manage large projects by grouping related classes into specific namespaces.

- To define a package in Java, you use the `package` keyword at the beginning of the Java file, followed by the package name.

Example (Java):

```
package com.example.animals;

public class Dog {
    public void bark() {
        System.out.println("Woof!");
    }
}
```

In this example:

- The `Dog` class is part of the `com.example.animals` package. This package name follows the **reverse domain name convention**.

2. Creating a Package

To create a package:

- **Step 1**: Define the package in the source code using the `package` keyword (as shown above).
- **Step 2**: Save the file in a directory structure that matches the package name. For example:
  - For the `com.example.animals` package, the source file should be saved in the directory structure `com/example/animals/`.

Example Directory Structure:

```
java/
  com/
    example/
      animals/
        Dog.java
```

Java's package system is tied to the filesystem, meaning that the directory structure should match the package structure.

3. Accessing a Package

Once a class is part of a package, it can be **accessed** by other classes or packages. To use a class from a package in another class, you need to **import** the package.

- **Default package**: If no package is declared, the class is placed in the **default package** (not recommended for large projects).

Example (Java):

```
package com.example.zoo;

import com.example.animals.Dog;  // Importing Dog class from com.example.animals package

public class Zoo {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.bark();  // Using the imported Dog class
    }
}
```

In this example:

- The Zoo class is in the com.example.zoo package.
- The Dog class from the com.example.animals package is imported using the import statement.

4. Understanding CLASSPATH

The **CLASSPATH** is an environment variable that tells the Java Virtual Machine (JVM) and Java compiler where to look for user-defined classes and packages during program execution.

- The CLASSPATH includes:
  - The directory structure where compiled .class files are located.
  - External libraries (JAR files) that might be needed during execution.

By default, the JVM searches the current directory (i.e., .) if no CLASSPATH is defined.

Setting CLASSPATH:

- You can set the CLASSPATH using the -cp or -classpath option when running the javac or java commands.
- Alternatively, you can set it as an environment variable in your operating system.

Example (Command Line):

```
D:\java> javac -d . Dog.java

D:\java> javac -d . Zoo.java

D:\java> java com.example.zoo.Zoo
```

Output:Woof!

In this example, the JVM is instructed to look in D:\java for the classes.

5. Importing Packages

To use classes from a different package, you need to import them using the import statement. This tells the compiler where to look for the class. There are two common ways to import packages:

a) Single-Class Import:

Imports only one class from a package.

```
import com.example.animals.Dog;  // Imports only the Dog class
```

b) Package Import (Wildcard):

Imports all the classes from a package. Be mindful that this does not include sub-packages.

```
import com.example.animals.*;  // Imports all classes from the com.example.animals package
```

Static Imports:

You can also import static members (methods, constants) of a class directly so that you don't need to qualify them with the class name.

```
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

public class Circle {
    public static void main(String[] args) {
        System.out.println(PI);       // No need to reference Math.PI
        System.out.println(sqrt(25)); // No need to reference Math.sqrt
    }
}
```

```
3.141592653589793
5.0
```

❖ Exception Handling in Java

Exception handling is a mechanism in Java that allows a program to deal with runtime errors, ensuring that the program can continue running or terminate gracefully rather than crashing unexpectedly. Here's a detailed overview:

1. Benefits of Exception Handling

- **Enhanced Robustness:** By handling exceptions, a program can deal with unexpected conditions without crashing. This improves the reliability of applications and ensures they continue to operate smoothly.
- **Separation of Concerns:** Exception handling allows you to separate error-handling code from the regular code. This makes the code cleaner and easier to understand and maintain.

- **Graceful Error Handling:** It enables the application to manage errors in a controlled manner, providing users with meaningful error messages and allowing for recovery or corrective actions.
- **Debugging and Logging:** Exceptions can be caught and logged, providing valuable information about errors that occur during runtime. This helps developers diagnose and fix issues more effectively.
- **Code Readability:** By using try, catch, finally, and custom exception classes, the code that deals with errors is distinct from the main logic, enhancing overall readability and maintainability.

2. Classification of Exceptions

Exceptions in Java are categorized into two main types:

Checked Exceptions

- **Definition:** Checked exceptions are those that the Java compiler forces you to handle or declare. They are checked at compile-time, and the programmer must either handle them using a try-catch block or declare them in the method signature using the throws keyword.
- **Examples:**
  - **IOException**: Thrown when an I/O operation fails or is interrupted.
  - **SQLException**: Thrown when a database access error occurs.
  - **ClassNotFoundException**: Thrown when an application tries to load a class through its name, but the class cannot be found.
- **Handling:** You must either catch these exceptions or declare them in the method signature. This requirement ensures that the code dealing with potential failures is written explicitly.

```
public void readFile(String fileName) throws IOException {
    FileReader file = new FileReader(fileName);
    // Read file content
    file.close();
}
```

Unchecked Exceptions

- **Definition:** Unchecked exceptions are those that are not checked at compile-time. They are subclasses of RuntimeException and typically represent programming errors or logical errors that can be avoided through better programming practices.
- **Examples:**
  - **NullPointerException**: Thrown when an application attempts to use null where an object is required.

- **ArrayIndexOutOfBoundsException**: Thrown when an array is accessed with an illegal index.
- **IllegalArgumentException**: Thrown when a method receives an argument that is inappropriate or illegal.

- **Handling:** These exceptions do not need to be declared in the method signature, and you are not required to handle them. They are usually handled through proper validation and error-checking practices within the code.

```
public void processArray(int[] array) {
    for (int i = 0; i <= array.length; i++) {
        // Potential ArrayIndexOutOfBoundsException
        System.out.println(array[i]);
    }
}
```

❖ Exception Hierarchy

In Java, exceptions are organized in a hierarchical structure rooted in the Throwable class. Understanding this hierarchy is key to effective exception handling and designing robust error management systems. Here's a detailed look at the exception hierarchy:

Exception Hierarchy Overview

Throwable

- The root class of the exception hierarchy. All exceptions and errors in Java are derived from this class.
  - **Exception**
  - **Error**

1. **Exception**
   - This class represents exceptional conditions that a program should catch. It is the base class for all checked exceptions and some runtime exceptions.
     - **IOException**
     - **SQLException**
     - **RuntimeException**

2. **Error**
   - Represents serious problems that applications should not typically catch. These are used by the JVM to indicate severe issues that usually

cannot be handled by the application itself (e.g., OutOfMemoryError, StackOverflowError).

Detailed Breakdown

1. **Exception**

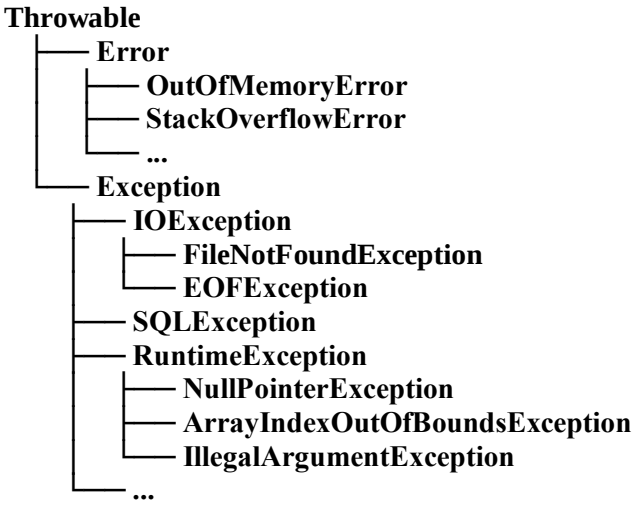
- **Description:** The superclass of all exceptions that can be handled. It is further divided into checked exceptions and unchecked exceptions.
- **Checked Exceptions:** These are exceptions that must be either caught or declared in the method signature using the `throws` keyword. They are typically related to external factors or resources that may fail.
    - **Examples:**
        - **IOException**: Represents I/O errors (e.g., file not found).
        - **SQLException**: Represents database access errors.
        - **ClassNotFoundException**: Thrown when a class cannot be located.
- **Unchecked Exceptions:** These are exceptions that do not need to be explicitly handled or declared. They are usually due to programming errors and are subclasses of `RuntimeException`.
    - **Examples:**
        - **NullPointerException**: Thrown when a `null` reference is accessed.
        - **ArrayIndexOutOfBoundsException**: Thrown when accessing an array with an illegal index.
        - **IllegalArgumentException**: Thrown when a method receives an illegal or inappropriate argument.

2. **Error**

- **Description:** Represents serious issues that are not intended to be caught by programs. These are generally caused by the JVM itself and are usually fatal.
- **Examples:**
    - **OutOfMemoryError**: Indicates that the JVM has run out of memory.
    - **StackOverflowError**: Thrown when a stack overflow occurs (e.g., excessive recursion).

Hierarchy Diagram

Here's a simplified diagram of the exception hierarchy:



❖ Java try-catch block and finally block

1. Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}
catch(Exception_class_Name ref){
}
```

Syntax of try-finally block

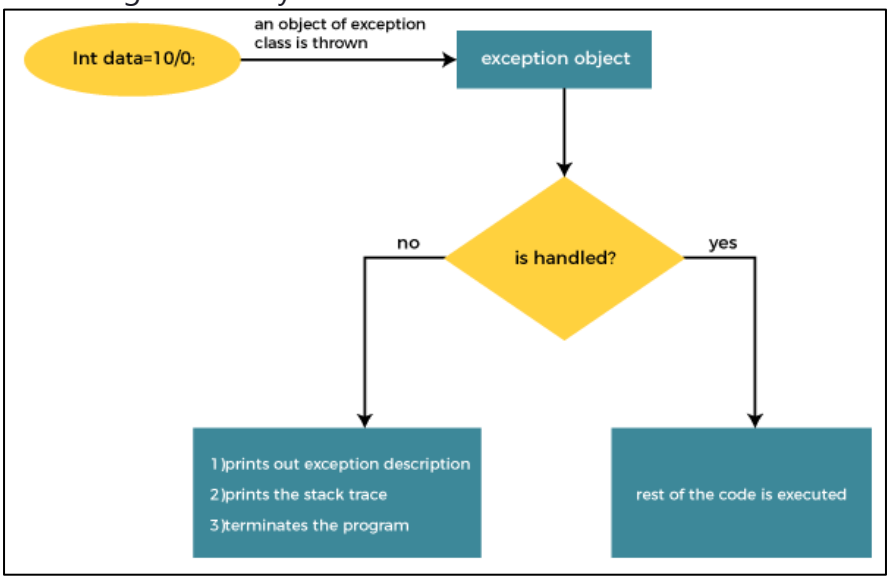
```
try{
//code that may throw an exception
}
finally{
}
```

2. Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks: Prints out exception description.

- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

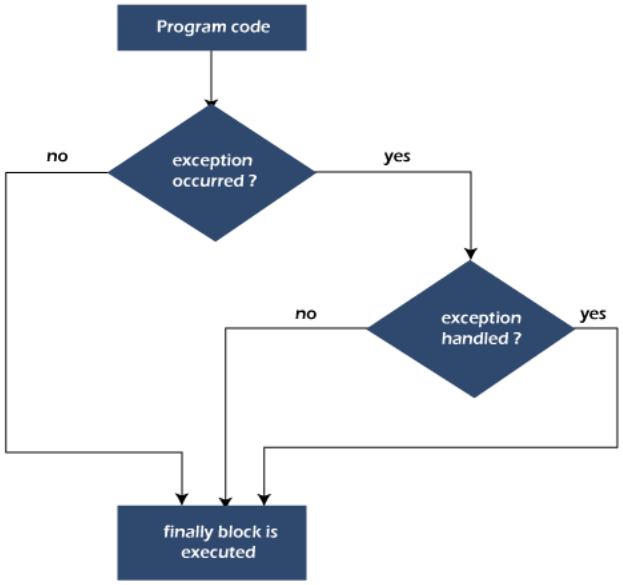
3. Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block



- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

```
import java.util.Scanner;

public class ExceptionHandle {

      public static void main(String args[]){
         Scanner s= new Scanner(System.in);
         System.out.println("enter numerator and denominator");
         int a = s.nextInt();
         int b = s.nextInt();
      try{

       int data=a/b;
       System.out.println(a+"/"+b+"="+data);
      }

       catch(ArithmeticException e){
     System.out.println(e);
     }

      finally {
```

```
    System.out.println("finally block is always executed");
    }

     }
    }
```

Output1:
enter numerator and denominator
10
2
10/2=5
finally block is always executed

Output2:
enter numerator and denominator
10
0
java.lang.ArithmeticException: / by zero
finally block is always executed.

❖ Rethrowing Exceptions

Rethrowing an exception refers to the practice of catching an exception in a catch block and throwing it again, allowing the exception to propagate further up the call stack. This is useful when you want to perform some action (e.g., logging) but still want the exception to be handled by another part of the program.

Example:

```
try {
   // Code that may throw an exception
} catch (Exception e) {
   // Perform some actions like logging
   throw e;  // Rethrowing the same exception
}
```

Rethrowing with a new exception:

- You can also wrap the original exception in a new one:

```
try {
   // Code that may throw an exception
} catch (IOException e) {
   throw new CustomException("Custom message", e);
// Wrapping the original exception
}
```

❖ Exception Specification (Checked and Unchecked Exceptions)

In Java, exception specification is handled through the throws keyword in the method signature. It indicates which exceptions a method may throw, forcing the caller to either handle or declare the exception.

- **Checked exceptions**: Must be caught or declared in the method signature.

```
 public void readFile() throws IOException {
   // Code that might throw IOException
}
```

- **Unchecked exceptions**: These are not required to be declared in the method signature. They are typically subclasses of RuntimeException.

❖ Built-in Exceptions

Java provides many built-in exceptions. These can be categorized as:

- **Checked Exceptions**:
  - IOException
  - SQLException
  - FileNotFoundException
  - ClassNotFoundException
- **Unchecked Exceptions (RuntimeException)**:
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - ArithmeticException
  - IllegalArgumentException

These exceptions represent common errors that can occur during program execution.

❖ Creating Custom Exception Subclasses

In Java, you can create your own exception classes to represent specific application-level errors. These custom exceptions can extend either the Exception class (for checked exceptions) or the RuntimeException class (for unchecked exceptions).

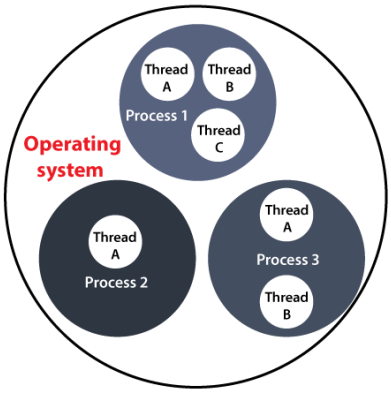
- **Creating a custom checked exception**:

```
public class MyCheckedException extends Exception {
    public MyCheckedException(String message) {
        super(message);
    }
}
```

UNIT-4

❖ Thread:

- Before introducing the **thread concept**, we were unable to run more than one task in parallel. It was a drawback, and to remove that drawback, **Thread Concept** was introduced.
- A **Thread** is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.
- In order to perform complicated tasks in the background, we used the **Threadconcept in Java**. All the tasks are executed without affecting the main program.
- In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.



- Another benefit of using **thread** is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads.
- All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as Multithreading.

- **Single Tasking:**
  Single-tasking in Java threads is when only one thread is executed at a time. This is achieved by using synchronization and locks to ensure that only one thread can access a particular resource at a time. Single-tasking is useful in situations where multiple threads accessing a resource could cause data corruption or other issues.
- **Multi Tasking:**
  Multitasking in Java threads is when multiple threads are executed simultaneously. This is useful in situations where an application needs to perform multiple tasks at the same time, such as in gaming, animation, or multimedia applications. Multitasking is achieved by creating multiple threads and running them concurrently.

❖ Multithreading vs. multiprocessing

Multithreading

Multithreading is a programming technique that assigns multiple code segments to a single process. These code segments, also referred to as threads, run concurrently and parallel to each other. These threads share the same memory space within a parent process. This saves system memory, increases computing speed and improves application performance.

For example, if you're working on your computer, you might have multiple browser tabs open while searching the internet. You might also be listening to music through a desktop application at the same time. The internet browser and music application represent two different processes, even though they are operating simultaneously. However, the multiple tabs you have open while browsing the internet represent threads of your internet browser, which is the parent process.

Multiprocessing

Multiprocessing refers to a system that has more than two central processing units (CPUs). Every additional CPU added to a system increases its speed, power and memory. This allows users to run multiple processes simultaneously. Each CPU may also function independently, and some CPUs may remain idle if they don't have anything to process. This can improve the reliability of a system because unused CPUs can act as a backup if technical issues arise.

There are two primary categories of multiprocessing systems:

- **Symmetric multiprocessing:** This multiprocessing system uses computer hardware and software that incorporates two or more identical processors connected by one memory space. These processors have complete access to all input and output devices and receive equal treatment.
- **Asymmetric multiprocessing:** In this multiprocessing system, different CPUs have access to separate input and output (I/O) devices. For example, one CPU might perform I/O operations, while another CPU might focus on maintaining the operating system.

Multithreading vs. multiprocessing

While multithreading and multiprocessing can both be used to increase the computing power of a system, there are some key differences between these

approaches. Here are some of the primary ways these methods differ from one another:

- Multiprocessing uses two or more CPUs to increase computing power, whereas multithreading uses a single process with multiple code segments to increase computing power.
- Multithreading focuses on generating computing threads from a single process, whereas multiprocessing increases computing power by adding CPUs.
- Multiprocessing is used to create a more reliable system, whereas multithreading is used to create threads that run parallel to each other.
- multithreading is quick to create and requires few resources, whereas multiprocessing requires a significant amount of time and specific resources to create.
- Multiprocessing executes many processes simultaneously, whereas multithreading executes many threads simultaneously.
- Multithreading uses a common address space for all the threads, whereas multiprocessing creates a separate address space for each process.

Benefits of multithreading

Here are some of the key benefits of multithreading:

- It requires less memory storage.
- Accessing memory is easier since threads share the same parent process.
- Switching between threads is fast and efficient.
- It's faster to generate new threads within an existing process than to create an entirely new process.
- All threads share one process memory pool and the same address space.
- Threads are more lightweight and have lower overhead.
- The cost of communication between threads is relatively low.
- Creating responsive user interfaces (UIs) is easy.

Drawbacks of multithreading

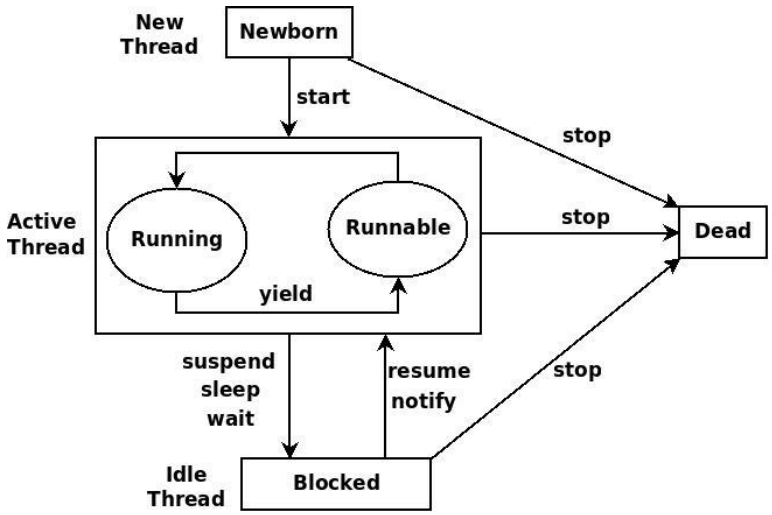
- Here are some potential drawbacks associated with multithreading:
- A multithreading system cannot be interrupted.
- The code can be more challenging to understand.
- The overhead associated with managing different threads may be too costly for basic tasks.
- Debugging and troubleshooting issues may become more challenging because the code can be complex.

❖ Lifecycle of Thread:

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked/Waiting
4. Timed Waiting
5. Terminated

The diagram shown below represents various states of a thread at any instant in time.



1. **New Thread:**
   When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.

2. **Runnable State:**
   A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

3. **Blocked/Waiting state:**

When a thread is temporarily inactive, then it's in one of the following states:

- Blocked
- Waiting

4. **Timed Waiting:**
   A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

5. **Terminated State:**
   A thread terminates because of either of the following reasons:

   - Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
   - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

Implementing the Thread States in Java

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread

```
public class MyThreadState extends Thread {
   public void run() {
      System.out.println(Thread.currentThread().getName() + " is running");
   }
   public static void main(String[] args) throws InterruptedException {
      Thread t1 = new MyThreadState();
      System.out.println("State: " + t1.getState());
      t1.start();
      System.out.println("State after start(): " + t1.getState());
      t1.join();
      System.out.println("State after join(): " + t1.getState());
   }
}
```

State: NEW
State after start(): RUNNABLE
Thread-0 is running
State after join(): TERMINATED

❖ Creating and running a Thread :

We can create Threads in java using two ways, namely :

1. Extending Thread Class
2. Implementing a Runnable interface

1. By Extending Thread Class :

We can run Threads in Java by using Thread Class, which provides constructors and methods for creating and performing operations on a Thread, which extends a Thread class that can implement Runnable Interface. We use the following constructors for creating the Thread:

- Thread
- Thread(Runnable r)
- Thread(String name)
- Thread(Runnable r, String name)

Example:

```
import java.io.*;
import java.util.*;
public class Demo extends Thread
 {
    // initiated run method for Thread
    public void run()
    {
        System.out.println("Thread Started Running...");
    }
   class DemoExample
{
    public static void main(String[] args)
    {
        Demo d1 = new Demo();

        // Invoking Thread using start() method
        d1.start();
    }
}
 }
```

Output:

Thread Started Running...

2.By Implementing a Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Example:

```
import java.io.*;
import java.util.*;
 class Demo2 implements Runnable
 {
    // method to start Thread
    public void run()
    {
        System.out.println("Thread is Running Successfully");
    }

       class DemoExample2
{
    public static void main(String[] args)
    {
        Demo2 d2= new Demo2();
        // initializing Thread Object
        Thread t1 = new Thread(d2);
        t1.start();
    }
}
 }
```

Output:

Thread is Running Successfully

❖ Interrupting Threads

Interrupting threads is a mechanism by which a thread can be signaled that it should stop or change its activity. This doesn't immediately terminate the thread; instead, it sets a flag or raises an exception that the thread can handle.

Java provides the interrupt() method, which raises an InterruptedException or sets the interrupt flag, depending on the thread's state. Here's how the interruption works:

1. **Calling interrupt()**:

- If the thread is in a blocking operation (e.g., sleep(), wait(), or join()), it throws an InterruptedException when interrupted.
- If the thread is not in a blocking operation, it doesn't immediately stop; instead, its interrupted status is set, and the thread continues running.

2. **Checking the interruption status**:
   - You can check whether a thread has been interrupted using the isInterrupted() method.
   - The interrupted() method checks and clears the interrupt flag.

Example in Java:

```
public class InterruptExample extends Thread {
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                System.out.println("Working on task " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread was interrupted.");
        }
    }
    public static void main(String[] args) {
        InterruptExample thread = new InterruptExample();
        thread.start();


        try {
            Thread.sleep(3000);
            thread.interrupt();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

Explanation:

- The run() method simulates a task by printing messages and sleeping for 1 second between iterations.
- If the thread is interrupted while sleeping, it catches an InterruptedException and exits.

❖ Priority of a Thread in Java

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. You can get and set the priority of a Thread. Thread class provides methods and constants for working with the priorities of a Thread.

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Built-in Property Constants of Thread Class

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

- **MIN_PRIORITY**: Specifies the minimum priority that a thread can have.
- **NORM_PRIORITY**: Specifies the default priority that a thread is assigned.
- **MAX_PRIORITY**: Specifies the maximum priority that a thread can have.

Thread Priority Setter and Getter Methods

- Thread.getPriority() Method: This method is used to get the priority of a thread.
- Thread.setPriority() Method: This method is used to set the priority of a thread, it accepts the priority value and updates an existing priority with the given priority.

```
import java.lang.*;
public class ThreadPriorityExample extends Thread{
    public void run()  {
System.out.println("Inside the run() method");
}
public static void main(String args[])  {
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
th1.setPriority(6);
```

```
th2.setPriority(3);
th3.setPriority(9);
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th3 is : " + th3.getPriority());
System.out.println("Currently Executing The Thread : " +
Thread.currentThread().getName());
System.out.println("Priority of the main thread is : " +
Thread.currentThread().getPriority());
Thread.currentThread().setPriority(10);
System.out.println("Priority of the main thread is : " +
Thread.currentThread().getPriority());
}
}
```

Output:

Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10

❖ Thread synchronization

Thread synchronization in Java is a way of programming several threads to carry out independent tasks easily. It is capable of controlling access to multiple threads to a particular shared resource.

The main reasons for using thread synchronization in Java are as follows:

- To prevent interference between threads.
- To prevent the problem of consistency.
-

Types of Synchronization

There are two types of thread synchronization in Java:

1. **Process synchronization:**

Process synchronization ensures that multiple processes in a system coordinate their actions to avoid conflicts and maintain consistency.

2. **Thread synchronization:**

Thread synchronization involves coordinating the execution of threads to ensure proper sharing of resources and prevent data inconsistency in a multithreaded environment.

Thread Synchronization

Thread synchronization refers to the concept where only one thread is executed at a time while other threads are in the waiting state. This process is called thread synchronization. It is used because it avoids interference of thread and the problem of inconsistency. There are two types of thread synchronization in Java:

- **Mutual exclusive-** It will keep the threads from interfering with each other while sharing any resources.
- **Inter-thread communication-** It is a mechanism in Java in which a thread running in the critical section is paused and another thread is allowed to enter or lock the same critical section that is executed.

❖ Thread Communication/Inter- Thread Communication:

- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of Object class:
  - wait()
  - notify()
  - notifyAll()

1) wait() method:

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method:

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

public final void notify()

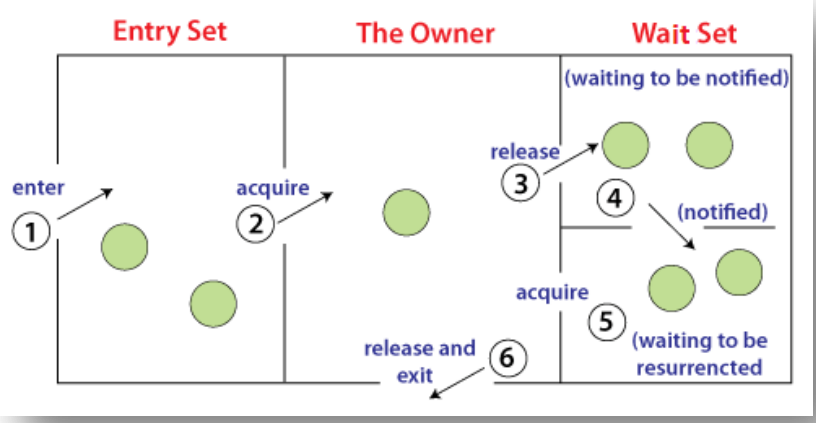
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

public final void notifyAll()

Understanding the process of inter-thread communication:



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.

4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

❖ The Stream Classes in Java

In Java, streams are used to perform input and output (I/O) operations. There are two major types of streams:

1. **Byte Streams** (for handling raw binary data)
2. **Character Streams** (for handling textual data)

1. Byte Streams

- **Description**: Byte streams are used to handle raw binary data like reading and writing binary files (images, audio, etc.). They operate on bytes (8-bit data), and all byte stream classes are derived from the abstract classes InputStream and OutputStream.

Common Byte Stream Classes:

- **InputStream**: The superclass for all byte input streams (e.g., FileInputStream, ByteArrayInputStream).
- **OutputStream**: The superclass for all byte output streams (e.g., FileOutputStream, ByteArrayOutputStream).

Writing to a File with FileOutputStream

```
import java.io.FileOutputStream;
import java.io.IOException;

public class WriteBytesExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.dat")) {
            byte[] data = {65, 66, 67};
            fos.write(data);
            System.out.println("Data written to file sucessfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
Output:
Data written to file sucessfully.
```

Reading from a File with FileInputStream

```
import java.io.FileInputStream;
import java.io.IOException;

public class ReadBytesExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("output.dat")) {
            int byteData;
            while ((byteData = fis.read()) != -1) {
                System.out.print((char) byteData);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
Output:
ABC
```

2. Character Streams

Character streams are used to handle text data (Unicode characters). They work with 16-bit Unicode characters and are derived from the abstract classes Reader and Writer.

In Java, a character stream is used to read and write data in characters (as opposed to bytes), which is particularly useful for handling text data. Character streams make it easier to work with text files, allowing for proper handling of character encoding.

Key Classes in Character Streams

1. **FileReader**: Used to read character files.

   FileReader reader = new FileReader("example.txt");

2. **FileWriter**: Used to write characters to a file.

   FileWriter writer = new FileWriter("example.txt");

3. **BufferedReader**: Buffers the input for efficiency and provides methods to read lines of text.

BufferedReader bufferedReader = new BufferedReader(new FileReader("example.txt"));

4. **BufferedWriter**: Buffers the output for efficiency and provides methods to write text to a file.

   BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter("example.txt"));

```
import java.io.*;

public class CharacterStreamExample {
    public static void main(String[] args) {
        String inputFile = "input.txt";
        String outputFile = "output.txt";

        try (BufferedReader reader = new BufferedReader(new FileReader(inputFile));
             BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {

            String line;
            while ((line = reader.readLine()) != null) {
                writer.write(line);

            }

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

❖ Reading Console Input and Writing Console Output

Java provides several ways to handle console input and output. The most commonly used methods are through the Scanner class for input and System.out for output. Additionally, the Console class is useful for handling sensitive data input such as passwords.

1. Reading Console Input

There are two popular ways to read input from the console in Java:

1.1 Using the Scanner Class

The Scanner class, introduced in Java 5, simplifies the process of reading primitive types and strings from the console.

Steps to use Scanner:

1. Import the Scanner class: import java.util.Scanner;
2. Create a Scanner object: Scanner sc = new Scanner(System.in);
3. Use methods like next(), nextLine(), nextInt(), nextDouble(), etc., to read input.

Example:

```
import java.util.Scanner;

public class ConsoleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.print("Enter your age: ");
        int age = scanner.nextInt();
        System.out.print("Enter your GPA: ");
        double gpa = scanner.nextDouble();
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("GPA: " + gpa);
        scanner.close();
    }
}
```

Enter your name: Ramu
Enter your age: 24
Enter your GPA: 9.7
Name: Ramu
Age: 24
GPA: 9.7

In the above example:

- nextLine() reads a line of text.
- nextInt() reads an integer.
- nextDouble() reads a floating-point number.

1.2 Using the BufferedReader Class

Before Scanner was introduced, the BufferedReader class was commonly used to read input. It's still useful if you want to read large input efficiently.

Steps to use BufferedReader:

1. Create a BufferedReader object wrapping InputStreamReader(System.in).
2. Use the readLine() method to read input.

Example:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ConsoleInputWithBufferedReader {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Enter your name: ");
        String name = reader.readLine();
        System.out.print("Enter your age: ");
        int age = Integer.parseInt(reader.readLine());
        System.out.println("Hello, " + name + ". You are " + age + " years old.");
    }
}
```

Here, BufferedReader reads input as a string and you can manually convert it to the desired data type (e.g., Integer.parseInt()).

2. Writing Console Output

Writing output to the console is straightforward in Java. You can use the System.out.print() and System.out.println() methods:

- **print()**: Outputs data without a newline at the end.
- **println()**: Outputs data followed by a newline.
- **printf()**: Formats output, similar to C's printf() function.

Example:

```
public class ConsoleOutputExample {
    public static void main(String[] args) {
        System.out.print("Hello ");
        System.out.print("World");
```

```
        System.out.println();
        System.out.println("Java programming is fun!");
        String language = "Java";
        int level = 5;
        System.out.printf("I am learning %s at level %d.%n", language, level);
    }
}
```

3. The Console Class for Input

The Console class, introduced in Java 6, provides a more specialized way to read from and write to the console. It's especially useful for reading sensitive information like passwords because it offers methods like readPassword(), which do not echo the characters typed.

Example using Console:

```
import java.io.Console;

public class ConsoleClassExample {
    public static void main(String[] args) {
        Console console = System.console();

        if (console != null) {
            String username = console.readLine("Enter your username: ");
            char[] password = console.readPassword("Enter your password: ");

            System.out.println("Username: " + username);
            System.out.println("Password: " + new String(password));
        } else {
            System.out.println("No console available.");
        }
    }
}
```

Output:
Enter your username: Ramu
Enter your password:
Username: Ramu
Password: ramachandra

This example uses the readLine() method for standard input and readPassword() for sensitive input, which hides the input as it is typed.

- **Reading Console Input**:
  - Use Scanner for simple and efficient input reading.
  - Use BufferedReader for faster reading of larger inputs.
  - Use the Console class for handling sensitive input like passwords.
- **Writing Console Output**:
  - Use System.out.print() and System.out.println() for output.
  - Use System.out.printf() for formatted output.

❖ File Class:

The File class in Java is part of the java.io package and represents a file or directory path in an abstract manner. It provides methods to create, delete, and check properties of files and directories, but it does not provide methods to read or write file content directly. Instead, it works in conjunction with input and output streams to manage file contents.

Key Features of the File Class:

- **File and Directory Creation**: You can create new files and directories.
- **File Information**: Retrieve information about files (size, permissions, last modified, etc.).
- **Delete Files/Directories**: Remove files or directories from the file system.
- **Path Manipulation**: Manipulate file paths and get various attributes.

Common Methods of the File Class:

- boolean createNewFile(): Creates a new, empty file if it does not already exist.
- boolean delete(): Deletes the file or directory.
- boolean exists(): Tests whether the file or directory exists.
- String getName(): Returns the name of the file or directory.
- long length(): Returns the length of the file in bytes.
- boolean isDirectory(): Tests whether the file is a directory.
- boolean isFile(): Tests whether the file is a regular file.
- String[] list(): Returns an array of strings naming the files and directories in the directory.

Example:

```
import java.io.*;
public class FileDemo{
```

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

            File file = new File("javaFile123.txt");
            if (file.createNewFile()){
                System.out.println("New File is created!");
            }
            else{
                System.out.println("File already exists.");
            }
        }
        catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Output:
New File is created!

❖ Reading and Writing Files

To read from and write to files in Java, you typically use classes from the `java.io` package, such as `FileReader`, `FileWriter`, `BufferedReader`, and `BufferedWriter`.

1. Reading Files

You can read files in Java using several methods. The most common way is to use `BufferedReader` in combination with `FileReader`.

Example of Reading a File with `BufferedReader`:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class FileReadingExample {
    public static void main(String[] args) {
        String filePath = "input.txt";
        try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
            String line;

            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

Output:
Hello java program

2. Writing to Files

To write to a file, you can use `BufferedWriter` along with `FileWriter`. This method allows you to write text to a file efficiently.

Example of Writing to a File with `BufferedWriter`:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class FileWritingExample {
    public static void main(String[] args) {
        String filePath = "output.txt";
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(filePath))) {
            writer.write("Hello, World!");
            writer.newLine();
            writer.write("Welcome to file writing in Java.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

❖ Serialization in Java

Serialization is the process of converting an object into a byte stream, allowing you to save the object's state to a file or send it over a network. Deserialization is the reverse process, where the byte stream is converted back into a copy of the original object. This mechanism is useful for persisting data and for communication in distributed systems.

Key Concepts of Serialization

1. **Serializable Interface**:

- To make a class serializable, it must implement the java.io.Serializable interface. This is a marker interface, which means it does not contain any methods.
- All non-transient instance variables of the class are serialized by default.

2. **transient Keyword**:
   - If you have certain fields in a class that you do not want to serialize, you can declare them as transient. These fields will not be included in the serialized representation of the object.
3. **ObjectOutputStream and ObjectInputStream**:
   - ObjectOutputStream is used to serialize objects, while ObjectInputStream is used to deserialize objects.

Example of Serialization and Deserialization

Here's a complete example demonstrating serialization and deserialization in Java:

Step 1: Create a Serializable Class

```
import java.io.Serializable;

class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;
    private transient String password;


    public Person(String name, int age, String password) {
        this.name = name;
        this.age = age;
        this.password = password;
    }


    public String getName() { return name; }
    public int getAge() { return age; }
    public String getPassword() { return password; }


    public void display() {
        System.out.println("Name: " + name + ", Age: " + age + ", Password: "
+ password);
    }
}
```

Step 2: Serialize the Object

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializationExample {
    public static void main(String[] args) {
        Person person = new Person("Ram", 30, "secret");


        try (FileOutputStream fileOut = new FileOutputStream("person.ser");
             ObjectOutputStream out = new ObjectOutputStream(fileOut)) {
            out.writeObject(person);
            System.out.println("Object has been serialized: " + person.getName());
        }
       catch (IOException e) {
              e.printStackTrace();
        }
    }
}
```

Output:

Object has been serialized: Ram

Step 3: Deserialize the Object

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeserializationExample {
    public static void main(String[] args) {
        Person person = null;

        try (FileInputStream fileIn = new FileInputStream("person.ser");
             ObjectInputStream in = new ObjectInputStream(fileIn)) {
            person = (Person) in.readObject();
            System.out.println("Object has been deserialized:");
            person.display();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

Object has been deserialized:

```
Name: Ram, Age: 30, Password: null
```

Explanation of the Example

1. **Serializable Class**: The Person class implements Serializable. The serialVersionUID is a unique identifier for the class, which helps in version control during deserialization. The password field is marked as transient, so it will not be serialized.
2. **Serialization**:
    - In the SerializationExample class, a Person object is created and serialized into a file named person.ser using ObjectOutputStream.
3. **Deserialization**:
    - In the DeserializationExample class, the object is deserialized from the file. The password field will be null since it was marked as transient.

UNIT-5

❖ The Origins of Swing

Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as heavyweight. The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently on different platforms.

The solution was Swing. Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Swing Is Built on the AWT

Before moving on, it is necessary to make one important point: although Swing eliminates a number of the limitations inherent in the AWT, Swing does not replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing.

Two Key Swing Features

As just explained, Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing. Each is examined here.

Swing Components Are Lightweight With very few exceptions, Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient

and more flexible. Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply "plugged in." Once this is done, all components are automatically rendered using that style.

Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look and feel that acts like a specific platform. For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.

❖ The MVC Connection

In general, a visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

No matter what architecture is used to implement a component, it must implicitly contain these three parts. Over the years, one component architecture has proven itself to be exceptionally effective: Model-View-Controller, or MVC for short.

The MVC architecture is successful because each piece of the design corresponds to an aspect of a component. In MVC terminology, the model corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The view determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The controller determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated. By separating a

component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate. For this reason, Swing's approach is called either the Model-Delegate architecture or the Separable Model architecture. Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.

Swing's pluggable look and feel is made possible by its Model-Delegate architecture. Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program. Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input.

To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate. Models are defined by interfaces. For example, the model for a button is defined by the ButtonModel interface. UI delegates are classes that inherit ComponentUI. For example, the UI delegate for a button is ButtonUI. Normally, your programs will not interact directly with the UI delegate.

❖ Components and Containers

A Swing GUI consists of two key items: components and containers. However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a component is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a containment hierarchy, at the top of which must be a top-level container.

Let's look a bit more closely at components and containers.

1. Components

In general, Swing components are derived from the JComponent class. (The only exceptions to this are the four top-level containers, described in the next section.) JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel. JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component.

All of Swing's components are represented by classes defined within the package javax.swing. The following table shows the class names for Swing components (including those used as containers).

JApplet ,JButton ,JCheckBox, JCheckBoxMenuItem, JColorChooser, JComboBox, JComponent, JDesktopPane, JDialog, JEditorPane, JFileChooser, JFormattedTextField, JFrame, JInternalFrame, JLabel, JLayer, JLayeredPane, JList, JMenu, JMenuBar, JMenuItem, JOptionPane, JPanel, JPasswordField, JPopupMenu, JProgressBar, JRadioButton, JRadioButtonMenuItem, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JSpinner, JSplitPane, JTabbedPane.

Notice that all component classes begin with the letter J. For example, the class for a label is JLabel; the class for a push button is JButton; and the class for a scroll bar is JScrollBar.

2. Containers

Swing defines two types of containers. The first are top-level containers: JFrame, JApplet, JWindow, and JDialog. These containers do not inherit JComponent. They do, however, inherit the AWT classes Component and Container. Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The

one most commonly used for applications is JFrame. The one used for applets is JApplet.

The second type of containers supported by Swing are lightweight containers. Lightweight containers do inherit JComponent. An example of a lightweight container is JPanel, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus, you can use lightweight containers such as JPanel to create subgroups of related controls that are contained within an outer container.

❖ Layout Manager in Java

In Java, graphical user interfaces (GUIs) play a vital role in creating interactive applications. To design a visually appealing and organized interface, the choice of layout manager becomes crucial. Layout managers define how components are arranged within a container, such as a JFrame or JPanel. Java provides several layout managers to suit various design needs. In this section, we will delve into the details of the different types of layout managers available in Java, along with code examples and explanations.

1. FlowLayout

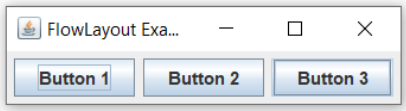
FlowLayout is a simple layout manager that arranges components in a row, left to right, wrapping to the next line as needed. It is ideal for scenarios where components need to maintain their natural sizes and maintain a flow-like structure.

FlowLayoutExample.java

```
import javax.swing.*;
import java.awt.*;
public class FlowLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FlowLayout Example");
        frame.setLayout(new FlowLayout());
        frame.add(new JButton("Button 1"));
        frame.add(new JButton("Button 2"));
        frame.add(new JButton("Button 3"));
        frame.pack();
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    }
}
```

Output:



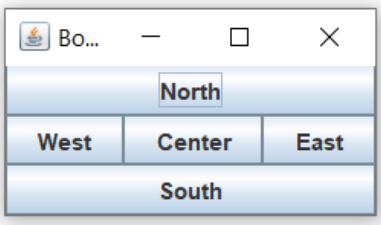
2. BorderLayout

BorderLayout divides the container into five regions: NORTH, SOUTH, EAST, WEST, and CENTER. Components can be added to these regions, and they will occupy the available space accordingly. This layout manager is suitable for creating interfaces with distinct sections, such as a title bar, content area, and status bar.

BorderLayoutExample.java

```
import javax.swing.*;
import java.awt.*;
public class BorderLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("BorderLayout Example");
        frame.setLayout(new BorderLayout());
        frame.add(new JButton("North"), BorderLayout.NORTH);
        frame.add(new JButton("South"), BorderLayout.SOUTH);
        frame.add(new JButton("East"), BorderLayout.EAST);
        frame.add(new JButton("West"), BorderLayout.WEST);
        frame.add(new JButton("Center"), BorderLayout.CENTER);
        frame.pack();
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Output:



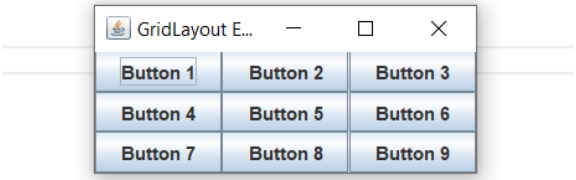
3. GridLayout

GridLayout arranges components in a grid with a specified number of rows and columns. Each cell in the grid can hold a component. This layout manager is ideal for creating a uniform grid of components, such as a calculator or a game board.

GridLayoutExample.java

```
import javax.swing.*;
import java.awt.*;
public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridLayout Example");
        frame.setLayout(new GridLayout(3, 3));
        for (int i = 1; i <= 9; i++) {
            frame.add(new JButton("Button " + i));
        }
        frame.pack();
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Output:



4. CardLayout

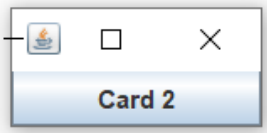
CardLayout allows components to be stacked on top of each other, like a deck of cards. Only one component is visible at a time, and you can switch between components using methods like next() and previous(). This layout is useful for creating wizards or multi-step processes.

CardLayoutExample.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class CardLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("CardLayout Example");
```

```
        CardLayout cardLayout = new CardLayout();
        JPanel cardPanel = new JPanel(cardLayout);
        JButton button1 = new JButton("Card 1");
        JButton button2 = new JButton("Card 2");
        JButton button3 = new JButton("Card 3");
        cardPanel.add(button1, "Card 1");
        cardPanel.add(button2, "Card 2");
        cardPanel.add(button3, "Card 3");
        frame.add(cardPanel);
        frame.pack();
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        button1.addActionListener(e -> cardLayout.show(cardPanel, "Card
2"));
        button2.addActionListener(e -> cardLayout.show(cardPanel, "Card
3"));
        button3.addActionListener(e -> cardLayout.show(cardPanel, "Card
1"));
    }
}
```

Output:



5. GroupLayout

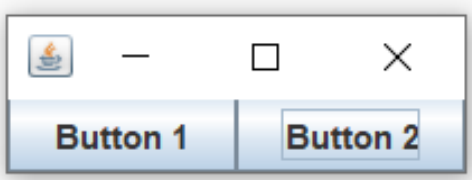
GroupLayout is a versatile and complex layout manager that provides precise control over the positioning and sizing of components. It arranges components in a hierarchical manner using groups. GroupLayout is commonly used in GUI builders like the one in NetBeans IDE.

GroupLayoutExample.java

```
import javax.swing.*;
public class GroupLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GroupLayout Example");
        JPanel panel = new JPanel();
        GroupLayout layout = new GroupLayout(panel);
        panel.setLayout(layout);
```

```
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        layout.setHorizontalGroup(layout.createSequentialGroup()
            .addComponent(button1)
            .addComponent(button2));
        layout.setVerticalGroup(layout.createParallelGroup()
            .addComponent(button1)
            .addComponent(button2));
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Output:



6. GridBagLayout

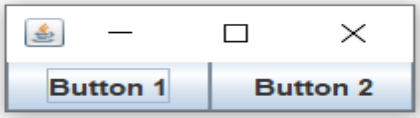
GridBagLayout is a powerful layout manager that allows you to create complex layouts by specifying constraints for each component. It arranges components in a grid, but unlike GridLayout, it allows components to span multiple rows and columns and have varying sizes.

GridBagLayoutExample.java

```
import javax.swing.*;
import java.awt.*;
public class GridBagLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridBagLayout Example");
        JPanel panel = new JPanel(new GridBagLayout());
        GridBagConstraints constraints = new GridBagConstraints();
        constraints.fill = GridBagConstraints.HORIZONTAL;
        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        constraints.gridx = 0;
        constraints.gridy = 0;
```

```
        panel.add(button1, constraints);
        constraints.gridx = 1;
        panel.add(button2, constraints);
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Output:



❖ Event handling

Event handling in Swing is crucial for creating interactive applications. Java uses the **Delegation Event Model**, which separates the source of events from the objects that handle those events. Let's break down the key components of this model.

The Delegation Event Model

1. **Events**: An event is an occurrence that happens in the application (e.g., a button click, mouse movement, key press).
2. **Event Sources**: These are the components that generate events. For instance, a button or a text field can be an event source.
3. **Event Listeners**: These are interfaces that define methods to handle events. An object must implement these interfaces to respond to specific events.
4. **Event Classes**: Java provides predefined classes for different types of events (e.g., ActionEvent, MouseEvent, KeyEvent).

Handling Events

To handle an event, you typically follow these steps:

1. **Register an Event Listener**: Attach a listener to the event source.

2. **Implement Listener Methods**: Define what happens when the event occurs by implementing the necessary methods in the listener interface.

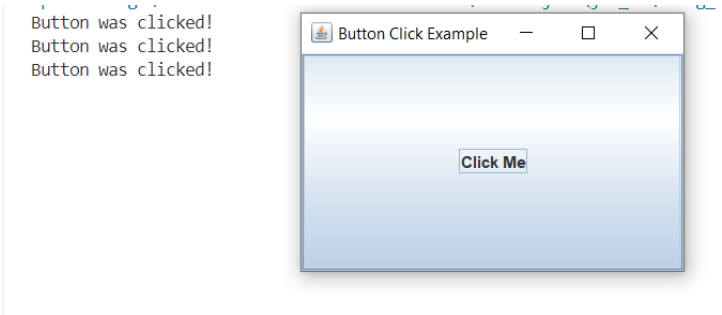
Example: Handling Button Clicks

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonClickExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Click Example");
        JButton button = new JButton("Click Me");

        // Adding action listener to the button
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button was clicked!");
            }
        });

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(button);
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}
```



Handling Mouse and Keyboard Events

1. **Mouse Events**: Handled using MouseListener and MouseMotionListener.
   - **MouseListener**: Handles mouse clicks, entry, exit, etc.
   - **MouseMotionListener**: Handles mouse movement.

```
button.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse clicked!");
    }
});
```

2. **Keyboard Events**: Handled using KeyListener.
    - It detects key presses, releases, and typing.

```
JTextField textField = new JTextField();
textField.addKeyListener(new KeyAdapter() {
    @Override
    public void keyPressed(KeyEvent e) {
        System.out.println("Key pressed: " + e.getKeyChar());
    }
});
```

Adapter Classes

Adapter classes are abstract classes that implement the listener interfaces. They allow you to override only the methods you're interested in, reducing boilerplate code.

For example, MouseAdapter provides default implementations for all methods in MouseListener, so you can override just the ones you need.

Inner Classes

Inner classes can be used to define event listeners within the same class. This is useful for accessing the outer class's members.

```
public class MyFrame extends JFrame {

    public MyFrame() {

        JButton button = new JButton("Click Me");

        button.addActionListener(new ActionListener() {

            @Override

            public void actionPerformed(ActionEvent e) {

                System.out.println("Button clicked in inner class!");

            }

        });

        add(button);

    }

}
```

Anonymous Inner Classes

Anonymous inner classes allow you to create a class on-the-fly, ideal for short, single-use implementations. This reduces the need for creating a separate class file.

```
button.addActionListener(new ActionListener() {

    @Override

    public void actionPerformed(ActionEvent e) {

        System.out.println("Button clicked using anonymous inner class!");

    }

});
```