WIPRO JAVA CONCEPS

23 October 2025 08:42

Java Concepts (Up to Multithreading)

◇ Basics

- JVM, JRE, JDK
- Data types: primitive vs reference
- Variables, constants (final)
- Operators: arithmetic, logical, relational

◇ Control Flow

- Conditionals: if, else, switch
- Loops: for, while, do-while
- Loop control: break, continue

◇ OOP Principles

- Class & Object
- Encapsulation (private, getters/setters)
- Inheritance (extends)
- Polymorphism: overloading & overriding
- Abstraction: abstract classes & interfaces
- Access modifiers: public, private, protected, default

◇ Core Concepts

- Constructors: default, parameterized
- this and super
- Static vs instance
- Packages & imports
- Exception handling: try-catch-finally, throw, throws

◇ Collections Framework

- List: ArrayList, LinkedList
- Set: HashSet, TreeSet
- Map: HashMap, TreeMap
- Queue: PriorityQueue, Deque
- Iterators: Iterator, ListIterator

◇ Java I/O

- Streams: InputStream, OutputStream, Reader, Writer
- File handling: File, Scanner, BufferedReader

◇ Multithreading

- Thread creation: Thread class, Runnable interface
- Thread lifecycle
- Synchronization: synchronized, locks
- Inter-thread communication: wait(), notify()
- Thread safety: volatile, atomic classes

SQL Concepts (Up to Stored Procedures)

◇ Basics

- Database, table, row, column
- Data types: INT, VARCHAR, DATE, etc.
- DDL: CREATE, ALTER, DROP
- DML: INSERT, UPDATE, DELETE
- DQL: SELECT, WHERE, ORDER BY, GROUP BY

◇ Constraints

- PRIMARY KEY, FOREIGN KEY
- UNIQUE, NOT NULL, DEFAULT, CHECK

◇ Joins

- INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN
- ON vs USING

◇ Aggregations

- COUNT, SUM, AVG, MIN, MAX
- GROUP BY, HAVING

◇ Subqueries & Views

- Nested SELECT
- CREATE VIEW, DROP VIEW

◇ Transactions

- BEGIN, COMMIT, ROLLBACK
- ACID properties

◇ Stored Procedures

- Definition: reusable SQL logic stored in DB
- Syntax:
  sql
  CREATE PROCEDURE GetEmployee(IN empId INT)
  BEGIN

```
    SELECT * FROM Employees WHERE id = empId;
END;
```

- Execution: CALL GetEmployee(101);

Basic DSA Concepts: Stack & Heap

◇ Stack (Data Structure)

- LIFO: Last In First Out
- Operations: push, pop, peek, isEmpty
- Use cases: recursion, undo, expression evaluation
- Java: Stack class or Deque as stack

◇ Heap (Memory Model)

- **Stack memory**: method calls, local variables
- **Heap memory**: objects, class instances
- Managed by Garbage Collector

◇ Heap (Data Structure)

- Binary heap: Min-Heap & Max-Heap
- Use cases: priority queues, scheduling
- Java: PriorityQueue

JAVA :

Differences Between JDK, JRE and JVM

Understanding the difference between JDK, JRE, and JVM plays a vital role in understanding how Java works and how each component contributes to the development and execution of Java applications.

JDK, JRE and JVM

- JDK stands for Java Development Kit. It is a set of development tools and libraries used to create Java programs. It works together with the JVM and JRE to run and build Java applications.
- JRE stands for Java Runtime Environment, and it provides an environment to run Java programs on the system. The environment includes Standard Libraries and JVM.
- JVM stands for Java Virtual Machine. It's responsible for executing the Java program.

Note: *Java bytecode can run on any machine with a JVM, but JVM implementations are platform-dependent for each operating system.*

JDK (Java Development Kit)

The JDK is a software development kit used to build Java applications. It contains the JRE and a set of development tools.

- Includes compiler (javac), debugger, and utilities like jar and javadoc.
- Provides the JRE, so it also allows running Java programs.
- Required by developers to write, compile, and debug code.

Components of JDK:

- JRE (JVM + libraries)
- Development tools (compiler, jar, javadoc, debugger)

Note:

- JDK is only for development (it is not needed for running Java programs)
- JDK is platform-dependent (different version for windows, Linux, macOS)

Working of JDK:

- **Source Code (.java):** Developer writes a Java program.
- **Compilation:** The JDK's compiler (javac) converts the code into bytecode stored in .class files.
- **Execution:** The JVM executes the bytecode, translating it into native instructions.

JRE (Java Runtime Environment)

The JRE provides an environment to run Java programs but does not include development tools. It is intended for end-users who only need to execute applications.

- Contains the JVM and standard class libraries.
- Provides all runtime requirements for Java applications.
- Does not support compilation or debugging.

Note:

- JRE is only for running applications, not for developing them.
- It is platform-dependent (different builds for different OS).

Working of JRE:

1. **Class Loading:** Loads compiled .class files into memory.
2. **Bytecode Verification:** Ensures security and validity of bytecode.

3. **Execution:** Uses the JVM (interpreter + JIT compiler) to execute instructions and make system calls.

JVM (Java Virtual Machine)

The JVM is the core execution engine of Java. It is responsible for converting bytecode into machine-specific instructions.

- Part of both JDK and JRE.
- Performs memory management and garbage collection.
- Provides portability by executing the same bytecode on different platforms.

Note:

- JVM implementations are platform-dependent.
- Bytecode is platform-independent and can run on any JVM.
- Modern JVMs rely heavily on Just-In-Time (JIT) compilation for performance.

Working of JVM:

JVM working

4. **Loading:** Class loader loads bytecode into memory.
5. **Linking:** Performs verification, preparation, and resolution.
6. **Initialization:** Executes class constructors and static initializers.
7. **Execution:** Interprets or compiles bytecode into native code.

JDK vs JRE vs JVM

Aspect	JDK	JRE	JVM
Purpose	Used to develop Java applications	Used to run Java applications	Executes Java bytecode
Platform Dependency	Platform-dependent (OS specific)	Platform-dependent (OS specific)	JVM is OS-specific, but bytecode is platform-independent
Includes	JRE + Development tools (javac, debugger, etc.)	JVM + Libraries (e.g., rt.jar)	ClassLoader, JIT Compiler, Garbage Collector
Use Case	Writing and compiling Java code	Running a Java application on a system	Convert bytecode into native machine code

1. Primitive Data Types

- **Definition**: Built-in basic types in Java that store **actual values** directly in memory.
- **Examples**:
  byte, short, int, long, float, double, char, boolean
- **Memory Storage**: Stores the value itself.
- **Default Values** (for instance variables):
  - int → 0
  - boolean → false
  - char → '\u0000' (null character)
  - double → 0.0
- **Immutable**: Once assigned, the value itself cannot be changed without reassigning.

Example:

Java
Copy code

```
public class PrimitiveExample {
   public static void main(String[] args) {
      int a = 10;      // stores value 10 directly
      int b = a;       // copies the value of a into b
      b = 20;          // changes b only, a remains unchanged
      System.out.println("a = " + a); // 10
      System.out.println("b = " + b); // 20
   }
}
```

2. Reference Data Types

- **Definition**: Variables that store the **memory address (reference)** of an object, not the object itself.
- **Examples**:
  All **objects**, **arrays**, **Strings**, **custom classes**, **interfaces**.
- **Memory Storage**: Stores a pointer (reference) to the location where the object is stored in the heap.
- **Default Value**: null (if not initialized).
- **Mutable or Immutable**: Depends on the object type (e.g., String is immutable, ArrayList is mutable).

Example:

Java

```
Copy code
public class ReferenceExample {
   public static void main(String[] args) {
      int[] arr1 = {1, 2, 3}; // arr1 stores reference to array in heap
      int[] arr2 = arr1;     // arr2 points to the same array
      arr2[0] = 99;          // modifies the same array
      System.out.println(arr1[0]); // 99 (change reflected in arr1)
   }
}
```

3. Key Differences Table

Feature	Primitive Type	Reference Type
Stores	Actual value	Memory address (reference)
Memory Location	Stack	Stack (reference) + Heap (object)
Default Value	Type-specific (e.g., 0, false)	null
Examples	int, double, boolean	String, arrays, objects
Mutability	Immutable values	Depends on object type
Size	Fixed (depends on type)	Varies (depends on object)

Java Variables and Constants :

In Java, variables are containers used to store data in memory. Variables define how data is stored, accessed, and manipulated.

A variable in Java has three components,

- **Data Type:** Defines the kind of data stored (e.g., int, String, float).
- **Variable Name:** A unique identifier following Java naming rules.
- **Value:** The actual data assigned to the variable.

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

      // Declaring and initializing variables

      // Integer variable
      int age = 25;

      // String variable
      String name = "GeeksforGeeks";

      // Double variable
      double salary = 50000.50;

      // Displaying the values of variables
      System.out.println("Age: " + age);
      System.out.println("Name: " + name);
      System.out.println("Salary: " + salary);
   }
}
```

Output

```
Age: 25
Name: GeeksforGeeks
Salary: 50000.5
```

How to Declare Java Variables?

The image below demonstrates how we can declare a variable in Java:

Variable Declaration

From the image, it can be easily perceived that while declaring a variable, we need to take care of two things that are data type of the variable and name.

How to Initialize Java Variables?

It can be perceived with the help of 3 components explained above:

Variable Initialization

Example: Here, we are initalizing variables of different types like float, int and char.

```
class Geeks{
   public static void main(String[] args) {
      // Declaring and initializing variables

      // Initializing float variable
      float si = 5.5f;

      // Initializing integer variables
      int t = 10;
      int s = 20;
```

```
        // Initializing character variable
        char var = 'h';

        // Displaying the values of the variables
        System.out.println("Simple Interest: " + si);
        System.out.println("Speed: " + s);
        System.out.println("Time: " + t);
        System.out.println("Character: " + var);
    }
}
```

Output

```
Simple Interest: 5.5
Speed: 20
Time: 10
Character: h
```

Rules to Name Java Variables

- Start with a Letter, $, or _ – Variable names must begin with a letter (a–z, A–Z), dollar sign $, or underscore _.
- **No Keywords:** Reserved Java keywords (e.g., int, class, if) cannot be used as variable names.
- **Case Sensitive:** age and Age are treated as different variables.
- **Use Letters, Digits, $, or _ :** After the first character, you can use letters, digits (0–9), $, or _.
- **Meaningful Names:** Choose descriptive names that reflect the purpose of the variable (e.g., studentName instead of s).
- **No Spaces:** Variable names cannot contain spaces.
- **Follow Naming Conventions:** Typically, use camelCase for variable names in Java (e.g., totalMarks).

1. Declaring Constants

Java

```
public class ConstantsExample {
   // Constant declaration
   public static final double PI = 3.141592653589793;
   public static final int MAX_USERS = 100;
public static void main(String[] args) {
      System.out.println("Value of PI: " + PI);
      System.out.println("Max Users Allowed: " + MAX_USERS);
   }
}
```

Explanation:

- final → Makes the variable immutable (cannot be reassigned).
- static → Belongs to the class, not an instance (shared across all objects).
- Naming convention → All uppercase letters with underscores.

2. Types of Constants

1. **Primitive Constants**
   Example: public static final int DAYS_IN_WEEK = 7;
2. **String Constants**
   Example: public static final String APP_NAME = "MyApp";
3. **Enum Constants** (for fixed sets of values)

   Java

   ```
   public enum Day {
      MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
   }
   ```
4. **Class Constants** (grouped in a constants class)

   Java

   ```
   public class AppConstants {
      public static final String VERSION = "1.0.0";
      public static final int TIMEOUT_SECONDS = 30;
   }
   ```

3. Best Practices

☑ Use public static final for global constants.
☑ Group related constants in a dedicated class or interface.
☑ Use **Enums** instead of multiple related String or int constants.
☑ Follow **UPPERCASE_WITH_UNDERSCORES** naming convention.

4. Example with Enum

Java

```
public enum Status {
   SUCCESS, ERROR, PENDING
}
public class TestEnum {
   public static void main(String[] args) {
      Status s = Status.SUCCESS;
      System.out.println("Status: " + s);
```

```
  }
}
```

OPERATORS :

1. Arithmetic Operators

Used for basic mathematical operations.

Operator	Description	Example (a=10, b=5)	Result
+	Addition	a + b	15
-	Subtraction	a - b	5
*	Multiplication	a * b	50
/	Division	a / b	2
%	Modulus (remainder)	a % b	0

2. Assignment Operators

Used to assign values to variables.

Operator	Example	Equivalent To
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 2	x = x - 2
*=	x *= 4	x = x * 4
/=	x /= 2	x = x / 2
%=	x %= 3	x = x % 3

3. Relational (Comparison) Operators

Used to compare two values.

Operator	Description	Example (a=10, b=5)	Result
==	Equal to	a == b	false
!=	Not equal to	a != b	true
>	Greater than	a > b	true
<	Less than	a < b	false
>=	Greater or equal	a >= b	true
<=	Less or equal	a <= b	false

4. Logical Operators

Used to combine conditional statements.

Operator	Description	Example (x=true, y=false)	Result
&&	Logical AND	x && y	false
`		`	Logical OR
!	Logical NOT	!x	false

5. Unary Operators

Operate on a single operand.

Operator	Description	Example (a=5)	Result
+	Positive	+a	5
-	Negation	-a	-5
++	Increment	++a (pre)	6
--	Decrement	--a (pre)	4
!	Logical NOT	!(a>0)	false

6. Bitwise Operators

Work at the binary level.

Operator	Description	Example (a=5, b=3)	Result
&	AND	a & b → 0101 & 0011	1
`	`	OR	`a
^	XOR	a ^ b → 0101 ^ 0011	6
~	NOT	~a	-6
<<	Left shift	a << 1	10
>>	Right shift	a >> 1	2
>>>	Unsigned right shift	a >>> 1	2

7. Ternary Operator

A shorthand for if-else.

Java
Copy code
```
int a = 10, b = 20;
int max = (a > b) ? a : b; // max = 20
```
8. instanceof Operator

Checks if an object is an instance of a specific class.

Java
Copy code
```
String str = "Hello";
boolean result = str instanceof String; // true
```
Example Program

Java
Copy code
```
public class OperatorsDemo {
   public static void main(String[] args) {
      int a = 10, b = 5;
// Arithmetic
      System.out.println("a + b = " + (a + b));
      System.out.println("a - b = " + (a - b));
// Relational
      System.out.println("a > b: " + (a > b));
// Logical
      boolean x = true, y = false;
      System.out.println("x && y: " + (x && y));
// Ternary
      int max = (a > b) ? a : b;
      System.out.println("Max: " + max);
   }
}
```
CONTROL FLOW STATEMENTS :

Control flow statements in Programming - GeeksforGeeks
Refer geeks for geeks for in detail explanation

OOPS :

Java OOP(Object Oriented Programming) Concepts - GeeksforGeeks
Java OOP(Object Oriented Programming) Concepts

Last Updated : 24 Sep, 2025
Before Object-Oriented Programming (OOPs), most programs used a procedural approach, where the focus was on writing step-by-step functions. This made it harder to manage and reuse code in large applications.

To overcome these limitations, Object-Oriented Programming was introduced. Java is built around OOPs, which helps in organizing code using classes and objects.

Key Features of OOP in Java:

- Structures code into logical units (classes and objects)
- Keeps related data and methods together (encapsulation)
- Makes code modular, reusable and scalable
- Prevents unauthorized access to data
- Follows the DRY (Don't Repeat Yourself) principle

OOP's concept

1. Class

A Class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. In general, class declarations can include these components in order:

- **Modifiers**: A class can be public or have default access (Refer to this for details).
- **Class name:** The class name should begin with the initial letter capitalized by convention.
- **Body:** The class body is surrounded by braces, { }.

2. Object

An Object is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

- **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity**: It is a unique name given to an object that enables it to interact with other objects.
- **Method**: A method is a collection of statements that perform some specific task and return the result to the caller.

Example:

```
public class Employee {
    // Instance variables (non-static)
    private String name;
    private float salary;

    // Constructor
    public Employee(String name, float salary) {
        this.name = name;
        this.salary = salary;
    }

    // getters method
    public String getName() { return name; }
    public float getSalary() { return salary; }

    // setters method
    public void setName(String name) { this.name = name; }
    public void setSalary(float salary) { this.salary = salary; }

    // Instance method
    public void displayDetails() {
        System.out.println("Employee: " + name);
        System.out.println("Salary: " + salary);
    }

    public static void main(String[] args) {
        Employee emp = new Employee("Geek", 10000.0f);
        emp.displayDetails();
    }
}
```

Output

```
Employee: Geek
Salary: 10000.0
```

Note: For more information, please refer to the article - **Classes and Object**.

3. Abstraction

Abstraction in Java is the process of hiding the implementation details and only showing the essential details or features to the user. It allows to focus on what an object does rather than how it does it. The unnecessary details are not displayed to the user.

Note: In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Example:

```
abstract class Vehicle {
    // Abstract methods (what it can do)
    abstract void accelerate();
    abstract void brake();

    // Concrete method (common to all vehicles)
    void startEngine() {
        System.out.println("Engine started!");
    }
}

// Concrete implementation (hidden details)
class Car extends Vehicle {
    @Override
    void accelerate() {
        System.out.println("Car: Pressing gas pedal...");
        // Hidden complex logic: fuel injection, gear shifting, etc.
    }

    @Override
    void brake() {
        System.out.println("Car: Applying brakes...");
```

```
        // Hidden logic: hydraulic pressure, brake pads, etc.
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.startEngine();
        myCar.accelerate();
        myCar.brake();
    }
}
```

Note: To learn more about the Abstraction refer to the **Abstraction in Java** article

4. Encapsulation

Encapsulation is defined as the process of wrapping data and the methods into a single unit, typically a class. It is the mechanism that binds together the code and the data. It manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.
- In encapsulation, the data in a class is hidden from other classes, which is similar to what **data-hiding** does. So, the terms "encapsulation" and "data-hiding" are used interchangeably.
- Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

Encapsulation
Example:

```
class Employee {
    // Private fields (encapsulated data)
    private int id;
    private String name;

    // Setter methods
    public void setId(int id) {
        this.id = id;
    }

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

    // Getter methods
    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee();

        // Using setters
        emp.setId(101);
        emp.setName("Geek");

        // Using getters
        System.out.println("Employee ID: " + emp.getId());
        System.out.println("Employee Name: " + emp.getName());
    }
}
```

Output

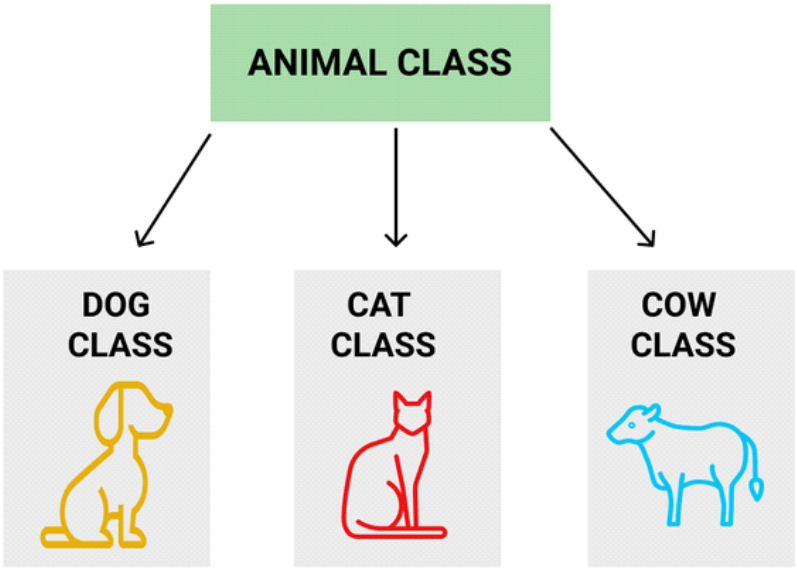
```
Employee ID: 101
Employee Name: Geek
```

Note: To learn more about topic refer to **Encapsulation in Java** article.

5. Inheritance

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using **extends** keyword. Inheritance is also known as "**is-a**" relationship.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



Inheritance
Let us discuss some frequently used important terminologies:

- **Superclass:** The class whose features are inherited is known as superclass (also known as base or parent class).
- **Subclass:** The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example:

```
// Superclass (Parent)
class Animal {
    void eat() {
        System.out.println("Animal is eating...");
    }

    void sleep() {
        System.out.println("Animal is sleeping...");
    }
}

// Subclass (Child) - Inherits from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking!");
    }
}

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

        // Inherited methods (from Animal)
        myDog.eat();
        myDog.sleep();

        // Child class method
        myDog.bark();
    }
}
```

Output

```
Animal is eating...
Animal is sleeping...
Dog is barking!
```

Note: To learn more about topic refer to **Inheritance in Java** article.

6. Polymorphism

The word **polymorphism** means having **many forms**, and it comes from the Greek words **poly (many)** and **morph (forms)**, this means one entity can take many forms. In Java, polymorphism allows the same method or object to behave differently

based on the context, specially on the project's actual runtime class.

Polymorphism in Java

Types of Polymorphism

Polymorphism in Java is mainly of 2 types as mentioned below:

8. Method Overloading
9. Method Overriding

Method Overloading and Method Overriding

1. Method Overloading: Also, known as **compile-time polymorphism**, is the concept of Polymorphism where more than one method share the same name with different signature(Parameters) in a class. The return type of these methods can or cannot be same.

2. Method Overriding: Also, known as run-time polymorphism, is the concept of Polymorphism where method in the child class has the same name, return-type and parameters as in parent class. The child class provides the implementation in the method already written.

Below is the implementation of both the concepts:

```
// Parent Class
class Parent {
    // Overloaded method (compile-time polymorphism)
    public void func() {
        System.out.println("Parent.func()");
    }

    // Overloaded method (same name, different parameter)
    public void func(int a) {
        System.out.println("Parent.func(int): " + a);
    }
}

// Child Class
class Child extends Parent {
    // Overrides Parent.func(int) (runtime polymorphism)
    @Override
    public void func(int a) {
        System.out.println("Child.func(int): " + a);
    }
}

public class Main {
    public static void main(String[] args) {
        Parent parent = new Parent();
        Child child = new Child();
        // Dynamic dispatch
        Parent polymorphicObj = new Child();

        // Method Overloading (compile-time)
        parent.func();
        parent.func(10);

        // Method Overriding (runtime)
        child.func(20);

        // Polymorphism in action
        polymorphicObj.func(30);
    }
}
```

Output

```
Parent.func()
Parent.func(int): 10
Child.func(int): 20
Child.func(int): 30
```

Advantage of OOPs over Procedure-Oriented Programming Language

Object-oriented programming (OOP) offers several key advantages over procedural programming:

- By using objects and classes, you can create reusable components, leading to less duplication and more efficient development.
- It provides a clear and logical structure, making the code easier to understand, maintain, and debug.
- OOP supports the DRY (Don't Repeat Yourself) principle.This principle encourages minimizing code repetition, leading to cleaner, more maintainable code. Common functionalities are placed in a single location and reused, reducing redundancy.
- By reusing existing code and creating modular components, OOP allows for quicker and more efficient application development

Disadvantages of OOPs

- OOP has concepts like classes, objects, inheritance etc. For beginners, this can be confusing and takes time to learn.
- If we write a small program, using OOP can feel too heavy. We might have to write more code than needed just to follow the OOP structure.
- The code is divided into different classes and layers, so in this, finding and fixing bugs can sometimes take more time.
- OOP creates a lot of objects, so it can use more memory compared to simple programs written in a procedural way.

ACCESS MODIFIERS :

Access Modifiers in Java - GeeksforGeeks

Access Modifiers in Java

Last Updated : 10 Oct, 2025

In Java, **access modifiers** are essential tools that define how the members of a class, like **variables, methods**, and even the **class** itself, can be accessed from other parts of our program.

There are 4 types of access modifiers available in Java:

Private Access Modifier

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

```
class Person {

    // private variable
    private String name;

    public void setName(String name)  {

        this.name = name; // accessible within class
    }

    public String getName() { return name; }
}

public class Geeks {
    public static void main(String[] args)
    {

        Person p = new Person();
        p.setName("Alice");

        // System.out.println(p.name); // Error: 'name'
        // has private access
        System.out.println(p.getName());
    }
}
```

Output

```
Alice
```

Explanation: Direct access to name is not allowed outside Person, enforcing encapsulation.

Default Access Modifier

When no access modifier is specified for a class, method, or data member, it is said to have the default access modifier by default. This means only classes within the same package can access it.

```
class Car {
    String model; // default access
}

public class Main {

    public static void main(String[] args){

        Car c = new Car();
        c.model = "Tesla"; // accessible within the same package
        System.out.println(c.model);
    }
}
```

Output

```
Tesla
```

Explanation: Members with default access cannot be accessed from classes in a different package.

Geeks.java: Default class within the same package

```java
// default access modifier
package p1;
// Class Geek is having
// Default access modifier
class Geek
{
    void display()
    {
        System.out.println("Hello World!");
    }
}
```

GeeksNew.java: Default class from a different package (for contrast)

```java
// package with default modifier
package p2;
import p1.*;    // importing package p1
// This class is having
// default access modifier
class GeekNew {
    public static void main(String args[]) {

        // Accessing class Geek from package p1
        Geek o = new Geek();
        o.display();
    }
}
```

Explanation: In this example, the program will show the compile-time error when we try to access a default modifier class from a different package.

Protected Access Modifier

The protected access modifier is specified using the keyword protected. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

```java
class Vehicle {
    protected int speed; // protected member
}

class Bike extends Vehicle {
    void setSpeed(int s)
    {
        speed = s; // accessible in subclass
    }

    int getSpeed()
    {
        return speed; // accessible in subclass
    }
}

public class Main {
    public static void main(String[] args){

        Bike b = new Bike();
        b.setSpeed(100);
        System.out.println("Access via subclass method: "
                           + b.getSpeed());

        Vehicle v = new Vehicle();
        System.out.println(v.speed);
    }
}
```

Output

```
Access via subclass method: 100
0
```

Explanation: speed is accessible via subclass methods and other classes in the same package, but direct access from a different package (non-subclass) would fail.

Public Access Modifier

The public access modifier is specified using the keyword public. Public members are accessible from everywhere in the program. There is no restriction on the scope of public data members.

```java
class MathUtils {

    public static int add(int a, int b) {
        return a + b;
    }
}

public class Main {

    public static void main(String[] args) {

        System.out.println(MathUtils.add(5, 10)); // accessible anywhere
    }
```

```
}
```
Output
```
15
```
Explanation: add() is globally accessible due to the public modifier.

Top-level classes or interfaces can not be declared as private because, private means "only visible within the enclosing class".

Comparison Table of Access Modifiers in Java

Access-Modifier

When to Use Each Access Modifier in Real-World Projects

- **Private:** The idea should be use as restrictive access as possible, so private should be used as much as possible.
- **Default** (Package-Private): Often used in package-scoped utilities or helper classes.
- **Protected:** Commonly used in inheritance-based designs like framework extensions.
- **Public**: This is used for API endpoints, service classes, or utility methods shared across different parts of an application.

CONSTRUCTORS :

A constructor is a special method that initializes an object and is automatically called when a class instance is created using new. It is used to set default or user-defined values for the object's attributes

- A constructor has the same name as the class.
- It does not have a return type, not even void.
- It can accept parameters to initialize object properties.

Types of Constructors in Java

There are Four types of constructors in Java

Constructor

1. Default Constructor

A default constructor has no parameters. It's used to assign default values to an object. If no constructor is explicitly defined, Java provides a default constructor.

```
import java.io.*;

class Geeks{

    // Default Constructor
    Geeks(){

        System.out.println("Default constructor");

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

        Geeks hello = new Geeks();
    }
}
```
Output
```
Default constructor
```
Note: *It is not necessary to write a constructor for a class because the Java compiler automatically creates a default constructor (a constructor with no arguments) if your class doesn't have any.*

2. Parameterized Constructor

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

```
import java.io.*;

class Geeks{

    // data members of the class
    String name;
    int id;

    Geeks(String name, int id){

        this.name = name;
        this.id = id;
    }
}
```

```
class GFG{

    public static void main(String[] args){

        // This would invoke the parameterized constructor
        Geeks geek1 = new Geeks("Sweta", 68);
        System.out.println("GeekName: " + geek1.name
                           + " and GeekId: " + geek1.id);
    }
}
```

Output

```
GeekName: Sweta and GeekId: 68
```

3. Copy Constructor in Java

Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

```
import java.io.*;

class Geeks{

    // data members of the class
    String name;
    int id;

    // Parameterized Constructor
    Geeks(String name, int id)
    {
        this.name = name;
        this.id = id;
    }

    // Copy Constructor
    Geeks(Geeks obj2)
    {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}

class GFG {
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor
        System.out.println("First Object");
        Geeks geek1 = new Geeks("Sweta", 68);
        System.out.println("GeekName: " + geek1.name
                           + " and GeekId: " + geek1.id);

        System.out.println();

        // This would invoke the copy constructor
        Geeks geek2 = new Geeks(geek1);
        System.out.println(
            "Copy Constructor used Second Object");
        System.out.println("GeekName: " + geek2.name
                           + " and GeekId: " + geek2.id);
    }
}
```

Output

```
First Object
GeekName: Sweta and GeekId: 68
Copy Constructor used Second Object
GeekName: Sweta and GeekId: 68
```

Note: *Java does not provide a built-in copy constructor like C++. We can create our own by writing a constructor that takes an object of the same class as a parameter and copies its fields.*

4. Private Constructor

A private constructor cannot be accessed from outside the class. It is commonly used in:

- **Singleton Pattern:** To ensure only one instance of a class is created.
- **Utility/Helper Classes:** To prevent instantiation of a class containing only static methods.

```
class GFG {
```

```
    // Private constructor
    private GFG(){

        System.out.println("Private constructor called");
    }

    // Static method
    public static void displayMessage(){

        System.out.println("Hello from GFG class!");
    }
}

class Main{

    public static void main(String[] args){

        // GFG u = new GFG(); // Error: constructor is
        // private
       GFG.displayMessage();
    }
}
```

Output

```
Hello from GFG class!
```

Constructor Overloading

This is a key concept in OOP related to constructors is constructor overloading. This allows us to create multiple constructors in the same class with different parameter lists.

```
import java.io.*;

class Geeks{

    // constructor with one argument
    Geeks(String name){

        System.out.println("Constructor with one "
                           + "argument - String: " + name);
    }

    // constructor with two arguments
    Geeks(String name, int age){

        System.out.println(
            "Constructor with two arguments: "
            + " String and Integer: " + name + " " + age);
    }

    // Constructor with one argument but with different
    // type than previous
    Geeks(long id)
    {
        System.out.println(
            "Constructor with one argument: "
            + "Long: " + id);
    }
}

class GFG {
    public static void main(String[] args){

        // Creating the objects of the class named 'Geek'
        // by passing different arguments
        Geeks geek2 = new Geeks("Sweta");

        // Invoke the constructor with two arguments
        Geeks geek3 = new Geeks("Amiya", 28);

        // Invoke the constructor with one argument of
        // type 'Long'.
        Geeks geek4 = new Geeks(325614567);
    }
}
```

Output

```
Constructor with one argument - String: Sweta
Constructor with two arguments:  String and Integer: Amiya 28
Constructor with one argument: Long: 325614567
```

THIS AND SUPER :

super and this keywords in Java

Last Updated : 10 Jun, 2024

In java, **super** keyword is used to access methods of the **parent class** while **this** is used to access methods of the **current class**.

this keyword is a reserved keyword in java i.e, we can't use it as an identifier. It is used to refer current class's instance as well as static members. It can be used in various contexts as given below:

- to refer instance variable of current class
- to invoke or initiate current class constructor
- can be passed as an argument in the method call
- can be passed as argument in the constructor call
- can be used to return the current class instance

Example

```
// Program to illustrate this keyword
// is used to refer current class
class RR {
    // instance variable
    int a = 10;

    // static variable
    static int b = 20;

    void GFG()
    {
        // referring current class(i.e, class RR)
        // instance variable(i.e, a)
        this.a = 100;

        System.out.println(a);

        // referring current class(i.e, class RR)
        // static variable(i.e, b)
        this.b = 600;

        System.out.println(b);
    }

    public static void main(String[] args)
    {
        // Uncomment this and see here you get
        // Compile Time Error since cannot use
        // 'this' in static context.
        // this.a = 700;
        new RR().GFG();
    }
}
```

Output

```
100
600
```

super keyword

10. **super** is a reserved keyword in java i.e, we can't use it as an identifier.
11. **super** is used to refer **super-class's instance as well as static members**.
12. **super** is also used to invoke **super-class's method or constructor**.
13. **super** keyword in java programming language refers to the superclass of the class where the super keyword is currently being used.
14. The most common use of **super** keyword is that it eliminates the confusion between the superclasses and subclasses that have methods with same name.

super can be used in various contexts as given below:

- it can be used to refer immediate parent class instance variable
- it can be used to refer immediate parent class method
- it can be used to refer immediate parent class constructor.

Example

```
// Program to illustrate super keyword
// refers super-class instance

class Parent {
    // instance variable
    int a = 10;
```

```
    // static variable
    static int b = 20;
}

class Base extends Parent {
    void rr()
    {
        // referring parent class(i.e, class Parent)
        // instance variable(i.e, a)
        System.out.println(super.a);

        // referring parent class(i.e, class Parent)
        // static variable(i.e, b)
        System.out.println(super.b);
    }

    public static void main(String[] args)
    {
        // Uncomment this and see here you get
        // Compile Time Error since cannot use 'super'
        // in static context.
        // super.a = 700;
        new Base().rr();
    }
}
```

Output

```
10
20
```

Similarities in this and super

- We can use *this* as well as *super* **anywhere except static area**. Example of this is already shown above where we use this as well as super inside public static void main(String[]args) hence we get Compile Time Error since cannot use them inside static area.
- We can use *this* as well as *super* **any number of times in a program**.
- Both are **non-static** keyword.

Example

```
// Java Program to illustrate using this
// many number of times

class RRR {
    // instance variable
    int a = 10;

    // static variable
    static int b = 20;

    void GFG()
    {
        // referring current class(i.e, class RR)
        // instance variable(i.e, a)
        this.a = 100;

        System.out.println(a);

        // referring current class(i.e, class RR)
        // static variable(i.e, b)
        this.b = 600;

        System.out.println(b);

        // referring current class(i.e, class RR)
        // instance variable(i.e, a) again
        this.a = 9000;

        System.out.println(a);
    }

    public static void main(String[] args)
    {
        new RRR().GFG();
    }
}
```

Output

```
100
600
9000
```

See above we have used **this** 3 times. So **this** can be used any number of times.

```
// Java Program to illustrate using super
// many number of times

class Parent {
    // instance variable
    int a = 36;

    // static variable
    static float x = 12.2f;
}

class Base extends Parent {
    void GFG()
    {
        // referring super class(i.e, class Parent)
        // instance variable(i.e, a)
        super.a = 1;
        System.out.println(a);

        // referring super class(i.e, class Parent)
        // static variable(i.e, x)
        super.x = 60.3f;

        System.out.println(x);
    }
    public static void main(String[] args)
    {
        new Base().GFG();
    }
}
```

Output

```
1
60.3
```

See above we have used **super** 2 times. So **super** can be used any number of times.

Note: *We can use 'this' as well as 'super' any number of times but main thing is that we* ***cannot*** *use them inside static context.*

Let us consider a **tricky** example of this keyword:

```
// Java program to illustrate
// the usage of this keyword

class RR {
    int first = 22;
    int second = 33;

    void garcia(int a, int b)
    {
        a = this.first;
        b = this.second;
        System.out.println(first);
        System.out.println(second);
        System.out.println(a);
        System.out.println(b);
    }

    void louis(int m, int n)
    {
        this.first = m;
        this.second = n;
        System.out.println(first);
        System.out.println(second);
        System.out.println(m);
        System.out.println(n);
    }

    public static void main(String[] args)
    {
        new RR().garcia(100, 200);
        new RR().louis(1, 2);
    }
}
```

Output

```
22
33
22
33
1
2
1
2
```

Understanding the Output:

```
//it is of S.O.P(first) of garcia method
22
//it is of S.O.P(second) of garcia method
33
//it is of S.O.P(a) of garcia method
22
//it is of S.O.P(b) of garcia method
33
//it is of S.O.P(first) of louis method
1
//it is of S.O.P(second) of louis method
2
//it is of S.O.P(m) of louis method
1
//it is of S.O.P(n) of louis method
2
```

Flow of program : First, it start from main and then we have **new RR().garcia(100, 200)** then flow goes to garcia method of RR class and then in that we have:

```
a = this.first
b = this.second
```

Here, what is happening is that the value of instance variable(i.e, first) and the value of static variable(i.e, second) are assigned to the garcia method's local variables a and b respectively. Hence value of a and b comes out to be 22 and 33 respectively. Next we have **new RR().louis(1, 2)** hence here flow goes to the louis method of RR class and in that we have:

```
this.first = m
this.second = n
```

Here, above what happens is that the value of the louis method's local variables i.e, m and n are assigned to the instance as well as static variables i.e, first and second respectively.

Hence, the value of first and second variables are same which we have passed into the louis method i.e, 1 and 2 respectively.

Can we use both this() and super() in the same constructor?

An interesting thing to note here is that according to Java guidelines, this() or super() must be the first statement in the constructor block for it to be executed. So we cannot have them together in the same constructor as both need to be the first statement in the block for proper execution, which is not possible. Trying to do so would raise a compiler error.

```
class Vehicle {
    Vehicle() { System.out.println("Vehicle is created."); }
}

class Bike extends Vehicle {
    Bike() { System.out.println("Bike is created."); }

    Bike(String brand)
    {
        super(); // it calls Vehicle(), the parent class
                 // constructor of class Bike
        this();
        System.out.println("Bike brand is " + brand);
    }
}

public class GFG {
    public static void main(String args[])
    {
        Bike bike = new Bike("Honda");
    }
}
```

On running the above program, we get an error on line number 12 saying that "call to this must be first statement in constructor". This is because we tried using both super() and this() together. While super() was the first line of the constructor, this() statement violated the condition that it should be the first line, hence raising an error.

Compile Errors:

```
prog.java:12: error: call to this must be first statement in constructor
        this();
```

```
            ^
1 error
```

STATIC AND NON STATIC

Difference between static and non-static variables in Java

Last Updated : 01 Jul, 2025

In Java, variables are mainly categorized into two main types based on their working and memory allocation, and these two variables are static variables and non-static variables. The main difference between them is listed below:

- **Static variables:** These are variables that are shared among all the instances of a class.
- **Non-static variables:** These are variables that belong to each individual instance of a class.

Java Static Variables

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

Note: All instances of the class share the same static variable.

Example:

```
// Java program to demonstrate execution
// of static blocks and variables

class Geeks {
    // static variable
    static int a = m1();

    // static block
    static { System.out.println("Inside static block"); }

    // static method
    static int m1()
    {
        System.out.println("from m1");
        return 20;
    }

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

Output

```
from m1
Inside static block
Value of a : 20
from main
```

Java Non-static variables

Non-static variables are variables that belongs to a specified object of a class, it is also known as instance variable. These variables are declared outside of a method, constructor or block. Each object created from the class gets its own separate copy of these variables.

Note: Non-static variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.

Example:

```
// Java program to demonstrates
// the working of non-static variables

public class Geeks
{
    // declaration of non-static variables.
    public String name;
    String division;
    private int age;

    // Constructor that initialize non-static variable.
    public Geeks(String sname)
    {
        name = sname;
    }

    //Method to initialize non-static variable.
    public void setDiv(String sdiv)
    {
        division = sdiv;
```

```
    }

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

    // Method to display the values
    public void printstud()
    {
        System.out.println("Student Name: " + name );
        System.out.println("Student Division: " + division);
        System.out.println("Student Age: " + age);
    }

    public static void main(String args[])
    {
        Geeks g = new Geeks("Monica");
        g.setAge(14);
        g.setDiv("B");
        g.printstud();
    }
}
```

Output

```
Student Name: Monica
Student Division: B
Student Age: 14
```

Static variables Vs Non-static variables

The main differences between static and non static variables are listed below:

Static variable	Non static variable
Static variables can be accessed using class name	Non static variables can be accessed using instance of a class
Static variables can be accessed by static and non static methods	Non static variables cannot be accessed inside a static method.
Static variables reduce the amount of memory used by a program.	Non static variables do not reduce the amount of memory used by a program
In Static variable Memory is allocated only once, at the time of class loading.	In non Static variable Memory is allocated each time an instance of the class is created.
Static variables Can be accessed from any part of the program.	Non Static variables Can be accessed only within the class or its instance.
Static variables Exists for the entire lifetime of the program.	Non Static variables Exists for the lifetime of the object.
Static variables Default value is assigned automatically.	Non Static variables Default value is not assigned automatically.
Static variables are shared among all instances of a class.	Non static variables are specific to that instance of a class.
Static variable is like a global variable and is available to all methods.	Non static variable is like a local variable and they can be accessed through only instance of a class.

Java Exception Handling

In Java, exception handling is a mechanism to handle runtime errors, allowing the normal flow of a program to continue. Exceptions are events that occur during program execution that disrupt the normal flow of instructions.

Basic try-catch Example

- The try block contains code that might throw an exception,
- The catch block handles the exception if it occurs.

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

        int n = 10;
        int m = 0;

        try {
            int ans = n / m;
            System.out.println("Answer: " + ans);
        } catch (ArithmeticException e){
            System.out.println("Error: Division by 0!");
        }
    }
}
```

Output

```
Error: Division by 0!
```

Finally Block

The finally block always executed whether an exception is thrown or not. The finally is used for closing resources like db connections, open files and network connections, It is used after a try-catch block to execute code that must run.

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

        int[] numbers = { 1, 2, 3 };
        try {
            // This will throw ArrayIndexOutOfBoundsException
            System.out.println(numbers[5]);

        }
        catch (ArrayIndexOutOfBoundsException e){

            System.out.println("Exception caught: " + e);
        }
        finally{
            System.out.println("This block always executes.");
        }
        System.out.println("Program continues...");
    }
}
```

Output

```
Exception caught: java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
This block always executes.
Program continues...
```

throw and throws Keywords

1. throw: Used to explicitly throw a single exception. We use throw when something goes wrong (or “shouldn’t happen”) and we want to stop normal flow and hand control to exception handling.

```
class Demo {
    static void checkAge(int age) {

        if (age < 18) {
            throw new ArithmeticException("Age must be 18 or above");
        }
    }
    public static void main(String[] args) {

        checkAge(15);
    }
}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: Age must be 18 or above

at Demo.checkAge(Demo.java:5)

at Demo.main(Demo.java:11)

2. throws: Declares exceptions that a method might throw, informing the caller to handle them. It is mainly used with checked exceptions (explained below). If a method calls another method that throws a checked exception, and it doesn’t catch it, it must declare that exception in its throws clause

```
import java.io.*;

class Demo {
    static void readFile(String fileName) throws IOException {

        FileReader file = new FileReader(fileName);
    }

    public static void main(String[] args){

        try {
            readFile("test.txt");
        } catch (IOException e){

            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

Output

```
File not found: test.txt (No such file or directory)
```

Internal Working of try-catch Block:

- JVM executes code inside the try block.
- If an exception occurs, remaining try code is skipped and JVM searches for a matching catch block.
- If found, the catch block executes.
- Control then moves to the finally block (if present).
- If no matching catch is found, the exception is handled by JVM's default handler.
- The finally block always executes, whether an exception occurs or not.
  Note: *When an exception occurs and is not handled, the program terminates abruptly and the code after it, will never execute.*

Java Exception Hierarchy

In Java, all exceptions and errors are subclasses of the Throwable class. It has two main branches

15. Exception.
16. Error

The below figure demonstrates the exception hierarchy in Java:

Types of Java Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their it's exceptions.

Exception

1. Built-in Exception

Build-in Exception are pre-defined exception classes provided by Java to handle common errors during program execution. There are two type of built-in exception in java.

- **Checked Exception:** These exceptions are checked at compile time, forcing the programmer to handle them explicitly.
- **Unchecked Exception:** These exceptions are checked at runtime and do not require explicit handling at compile time.
  To know more about Checked and Unchecked Exception -> *Checked and Unchecked Exception*

2. User-Defined Exception

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called "user-defined Exceptions".

Methods to Print the Exception Information

- **printStackTrace()**: Prints the full stack trace of the exception, including the name, message and location of the error.
- **toString()**: Prints exception information in the format of the Name of the exception.
- **getMessage()** : Prints the description of the exception

Nested try-catch

In Java, you can place one try-catch block inside another to handle exceptions at multiple levels.

```
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            System.out.println("Outer try block");
            try {
                int a = 10 / 0; // This causes ArithmeticException
            } catch (ArithmeticException e) {
                System.out.println("Inner catch: " + e);
            }
            String str = null;
            System.out.println(str.length()); // This causes NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Outer catch: " + e);
        }
    }
}
```

Output

```
Outer try block
Inner catch: java.lang.ArithmeticException: / by zero
Outer catch: java.lang.NullPointerException: Cannot invoke "String.length()" because "<local1>" is null
```

Handling Multiple Exception

We can handle multiple type of exceptions in Java by using multiple catch blocks, each catching a different type of exception.

```
try {

    // Code that may throw an exception
```

```
} catch (ArithmeticException e) {

    // Code to handle the exception

} catch(ArrayIndexOutOfBoundsException e){

    // Code to handle the anothert exception

}catch(NumberFormatException e){

    // Code to handle the anothert exception
}
```

How Does JVM Handle an Exception?

When an Exception occurs, the JVM creates an exception object containing the error name, description and program state. Creating the exception object and handling it in the run-time system is called throwing an exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of methods is called call stack. Now the following procedure will happen:

- The run-time system searches the call stack for an exception handler
- It starts searching from the method where the exception occurred and proceeds backward through the call stack.
- If a handler is found, the exception is passed to it.
- If no handler is found, the default exception handler terminates the program and prints the stack trace.

Exception in thread "abc" Name of Exception : Description

// Call Stack

Look at the below diagram to understand the flow of the call stack:

Illustration:

```
class Geeks{
    public static void main(String args[])
    {
        // Taking an empty string
        String s = null;

        // Getting length of a string
        System.out.println(s.length());
    }
}
```

Try it on GfG Practice

Output:

```
PS C:\Devanshu> javac Geeks.java
PS C:\Devanshu> java Geeks
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
 because "<local1>" is null
        at Geeks.main(Geeks.java:9)
PS C:\Devanshu>
```

output

Difference Between Exception and Error

Feature	Exception	Error
Definition	An event that occurs during program execution, disrupting normal flow, which can be handled using try-catch.	A serious problem that occurs in the JVM, generally cannot be handled by the application.
Package	java.lang.Exception	java.lang.Error
Recoverable	Yes, can be caught and handled.	No, usually not recoverable.
Examples	IOException, SQLException, ArithmeticException	OutOfMemoryError, StackOverflowError

JAVA COLLECTIONS :

Absolutely, Attada! Here's a **detailed, mentor-friendly breakdown of the Java Collections Framework**, complete with **clear examples**, **use cases**, and **best practices** — perfect for onboarding or teaching others.

What Is the Java Collections Framework?

The **Java Collections Framework (JCF)** is a set of **interfaces**, **classes**, and **algorithms** that help manage groups of objects efficiently. It replaces the need for custom data structures and provides built-in tools for sorting, searching, and iterating.

Core Interfaces and Their Purpose

Interface	Description	Example Classes
Collection	Root interface for all collections	—
List	Ordered collection, allows duplicates	ArrayList, LinkedList
Set	Unordered, no duplicates	HashSet, TreeSet
Queue	FIFO structure	PriorityQueue, Deque
Map	Key-value pairs	HashMap, TreeMap

Detailed Examples

1. List Interface

☑ **Use Case: Ordered data with duplicates**

```
List<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Apple"); // duplicates allowed
System.out.println(fruits); // [Apple, Banana, Apple]
```

- **ArrayList**: Fast access, slow insert/delete
- **LinkedList**: Fast insert/delete, slower access

2. Set Interface

☑ **Use Case: Unique elements only**

```
Set<String> countries = new HashSet<>();
countries.add("India");
countries.add("USA");
countries.add("India"); // ignored
System.out.println(countries); // [India, USA]
```

- **HashSet**: Unordered, fast lookup
- **TreeSet**: Sorted order

3. Map Interface

☑ **Use Case: Key-value storage**

```
Map<Integer, String> employeeMap = new HashMap<>();
employeeMap.put(101, "Alice");
employeeMap.put(102, "Bob");
System.out.println(employeeMap.get(101)); // Alice
```

- **HashMap**: Unordered, fast access
- **TreeMap**: Sorted by keys

4. Queue Interface

☑ **Use Case: FIFO structure**

```
Queue<String> tasks = new LinkedList<>();
tasks.add("Task1");
tasks.add("Task2");
System.out.println(tasks.poll()); // Task1
```

- **PriorityQueue**: Elements sorted by priority
- **Deque**: Double-ended queue (can act as stack)

Utility Class: Collections

```
List<Integer> numbers = Arrays.asList(3, 1, 4, 2);
Collections.sort(numbers);
System.out.println(numbers); // [1, 2, 3, 4]
```

- sort(), reverse(), shuffle(), min(), max()

Thread Safety

- **Not thread-safe**: ArrayList, HashMap
- **Thread-safe alternatives**:
  - Collections.synchronizedList(new ArrayList<>())
  - ConcurrentHashMap

Best Practices

- Use **List** when order matters.
- Use **Set** to eliminate duplicates.
- Use **Map** for fast key-based lookup.
- Prefer **interface types** (List, Set, Map) for flexibility.
- Use **generics** to enforce type safety: List<String>, Map<Integer, String>

And refer geeks for geeks and w3 schools

MULTITHREADING :

Multithreading in Java is a feature that enables a program to run multiple threads simultaneously, allowing tasks to execute in parallel and utilize the CPU more efficiently. A thread is a lightweight, independent unit of execution inside a program (process).

- A process can have multiple threads.
- Each thread runs independently but shares the same memory.
  Example: Imagine a restaurant kitchen. Multiple chefs (threads) are preparing different dishes at the same time. This speeds up service and utilizes all available resources (CPU).

Multithreading

Different Ways to Create Threads

Threads can be created by using two mechanisms:

1. Extending the Thread class

We create a class that extends Thread and override its run() method to define the task. Then, we make an object of this class and call start(), which automatically calls run() and begins the thread's execution.

Example: Restaurant Kitchen (Extending Thread)

```
class CookingTask extends Thread {
    private String task;

    CookingTask(String task) {
        this.task = task;
    }

    public void run() {
        System.out.println(task + " is being prepared by " +
            Thread.currentThread().getName());
    }
}

public class Restaurant {
    public static void main(String[] args) {
        Thread t1 = new CookingTask("Pasta");
        Thread t2 = new CookingTask("Salad");
        Thread t3 = new CookingTask("Dessert");
        Thread t4 = new CookingTask("Rice");

        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

Output

```
Salad is being prepared by Thread-1
Rice is being prepared by Thread-3
Dessert is being prepared by Thread-2
Pasta is being prepared by Thread-0
```

Explanation:

- We created multiple threads (t1–t4) using the CookingTask class.
- Each thread represents a dish being prepared.
- Calling start() runs them concurrently.

2. Implementing the Runnable Interface

We create a new class which implements java.lang.Runnable interface and define the run() method there. Then we instantiate a Thread object and call start() method on this object.

Example: Restaurant Kitchen (Runnable Interface)

```
class CookingJob implements Runnable {
    private String task;

    CookingJob(String task) {
        this.task = task;
    }

    public void run() {
        System.out.println(task + " is being prepared by " +
            Thread.currentThread().getName());
    }
}

public class RestaurantRunnable {
    public static void main(String[] args) {
        Thread t1 = new Thread(new CookingJob("Soup"));
        Thread t2 = new Thread(new CookingJob("Pizza"));
        Thread t3 = new Thread(new CookingJob("Burger"));
```

```
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output

```
Burger is being prepared by Thread-2
Pizza is being prepared by Thread-1
Soup is being prepared by Thread-0
```

Explanation:

- CookingJob implements Runnable and overrides run().
- We pass a Runnable object to the Thread constructor.
- Calling start() runs them in parallel.

When to Use Which?

- **Use extends Thread:** if your class does not extend any other class.
- **Use implements Runnable:** if your class already extends another class (preferred because Java doesn't support multiple inheritance).

Advantages of Multithreading in Java

17. **Improved Performance:** Multiple tasks can run simultaneously, reducing execution time.
18. **Efficient CPU Utilization:** Threads keep the CPU busy by running tasks in parallel.
19. **Responsiveness:** Applications (like GUIs) remain responsive while performing background tasks.
20. **Resource Sharing:** Threads within the same process share memory and resources, avoiding duplication.
21. **Better User Experience:** Smooth execution of tasks like file downloads, animations, and real-time updates.

JAVA IO STREAMS :

Input/Output in Java with Examples

Last Updated : 04 Sep, 2025

Java I/O (Input/Output) is a collection of classes and streams in the java.io package that handle reading data from sources (like files, keyboard, or network) and writing data to destinations (like files, console or sockets). It provides both byte and character streams to support all types of data.

Flow from Source to Destination

Default/Standard Streams in Java

Before exploring various input and output streams, let's look at 3 most commonly used default streams that Java has provided:

Default Stream

- **System.in:** This is the standard input stream that is used to read characters from the keyboard or any other standard input device.
- **System.out:** This is the standard output stream that is used to produce the result of a program on an output device like the computer screen.
- **System.err:** This is the standard error stream that is used to display error messages separately from normal output.

Functions used with Default Streams:

1. System.in

System.in is an InputStream, so it provides methods from the InputStream class. Commonly used ones are:

- **int read():** reads one byte of data.
- **int read(byte[] b):** reads bytes into an array.
- **int read(byte[] b, int off, int len):** reads bytes into part of an array.
- **void close():** closes the input stream.
- **int available():** returns the number of bytes that can be read without blocking.

Usually, System.in is wrapped with classes like Scanner or BufferedReader for easier input handling.

Example:

```
import java.io.IOException;

public class SystemInExample {
    public static void main(String[] args) throws IOException {
        System.out.println("Enter a character:");

        // Reads a single byte from System.in
        int data = System.in.read();

        // Print the character and its ASCII value
        System.out.println("You entered: " + (char) data);
        System.out.println("ASCII Value: " + data);
    }
```

}

Output:

```
Enter a character:
a
You entered: a
ASCII Value: 97
```

Output

2. System.out

print(): This Java method displays text on the console, keeping the cursor at the end of the printed text so the next output continues from the same line.

Syntax:

System.out.print(parameter);

Example:

```
import java.io.*;

class Geeks {
    public static void main(String[] args)
    {
        // using print() all are printed in the same line
        System.out.print("GfG! ");
        System.out.print("GfG! ");
        System.out.print("GfG! ");
    }
}
```

Output

```
GfG! GfG! GfG!
```

println(): This method in Java is also used to display a text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.

Syntax:

System.out.println(parameter);

Example:

```
import java.io.*;

class Geeks {
    public static void main(String[] args)
    {
        // using println() all are printed in the different line
        System.out.println("GfG! ");
        System.out.println("GfG! ");
        System.out.println("GfG! ");
    }
}
```

Output

```
GfG!
GfG!
GfG!
```

printf(): The printf() method in Java is used for formatted output and can take multiple arguments, making it more flexible than print() or println().

Example:

```
class Geeks {

    public static void main(String[] args) {
        int x = 100;

        // Printing a simple integer
        System.out.printf("Printing simple integer: x = %d%n", x);

        // Printing a floating-point number with precision
        System.out.printf("Formatted with precision: PI = %.2f%n", Math.PI);

        float n = 5.2f;

        // Formatting a float to 4 decimal places
        System.out.printf("Formatted to specific width: n = %.4f%n", n);

        n = 2324435.3f;
```

```
        // Right-aligning and formatting a float to 20-character width
        System.out.printf("Formatted to right margin: n = %20.4f%n", n);
    }
}
```

Output

```
Printing simple integer: x = 100
Formatted with precision: PI = 3.14
Formatted to specific width: n = 5.2000
Formatted to right margin: n =         2324435.2500
```

3. System.err

It is used to display the error messages. It works similarly to System.out with print(), println(), and printf() methods.

Example: Java Program demonstrating System.err

```
public class Geeks {

    public static void main(String[] args) {

        // Using print()
        System.err.print("This is an error message using print().\n");
        // Using println()
        System.err.println("This is another error message using println().");
        //Using printf()
        System.err.printf("Error code: %d, Message: %s%n", 404, "Not Found");
    }
}
```

Output:

```
This is an error message using print().
This is another error message using println().
Error code: 404, Message: Not Found
```

Output

Types of Streams

Depending on the type of operations, streams can be divided into two primary classes:

1. Input Stream: These streams are used to read data that must be taken as an input from a source array or file or any peripheral device. For eg., FileInputStream, BufferedInputStream, ByteArrayInputStream etc.

InputStream

2. Output Stream: These streams are used to write data as outputs into an array or file or any output peripheral device. For eg., FileOutputStream, BufferedOutputStream, ByteArrayOutputStream etc.

Output Stream

To know more about types of Streams, you can refer to: Java.io.InputStream Class & Java.io.OutputStream class.

Types of Streams Depending on the Types of File

Depending on the types of file, Streams can be divided into two primary classes which can be further divided into other classes as can be seen through the diagram below followed by the explanations.

Classification

1. ByteStream:

Byte streams in Java are used to perform input and output of 8-bit bytes. They are suitable for handling raw binary data such as images, audio, and video, using classes like InputStream and OutputStream.

Here is the list of various ByteStream Classes:

Stream class	Description
BufferedInputStream	Used to read data more efficiently with buffering.
DataInputStream	Provides methods to read Java primitive data types.
FileInputStream	This is used to read from a file.
InputStream	This is an abstract class that describes stream input.
PrintStream	This contains the most used print() and println() method
BufferedOutputStream	This is used for Buffered Output Stream.
DataOutputStream	This contains method for writing java standard data types.
FileOutputStream	This is used to write to a file.
OutputStream	This is an abstract class that describes stream output.

Example: Java Program illustrating the Byte Stream to copy contents of one file to another file.

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

```
    public static void main(
        String[] args) throws IOException
    {
        FileInputStream sourceStream = null;
        FileOutputStream targetStream = null;
        try {
            sourceStream
                = new FileInputStream("sorcefile.txt");
            targetStream
                = new FileOutputStream("targetfile.txt");
            // Reading source file and writing content to target file byte by byte
            int temp;
            while ((
                    temp = sourceStream.read())
                != -1)
                targetStream.write((byte)temp);
        }
        finally {
            if (sourceStream != null)
                sourceStream.close();
            if (targetStream != null)
                targetStream.close();
        }
    }
}
```

Output:

Shows contents of file test.txt

2. CharacterStream:

Character streams in Java are used to perform input and output of 16-bit Unicode characters. They are best suited for handling text data, using classes like Reader and Writer which automatically handle character encoding and decoding.

Here is the list of various CharacterStream Classes:

Stream class	Description
BufferedReader	It is used to handle buffered input stream.
FileReader	This is an input stream that reads from file.
InputStreamReader	This input stream is used to translate byte to character.
OutputStreamWriter	Converts character stream to byte stream.
Reader	This is an abstract class that define character stream input.
PrintWriter	This contains the most used print() and println() method
Writer	This is an abstract class that define character stream output.
BufferedWriter	This is used to handle buffered output stream.
FileWriter	This is used to output stream that writes to file.

Example:

```
import java.io.*;
public class Geeks
{
    public static void main(String[] args) throws IOException
    {
        FileReader sourceStream = null;

            try {
            sourceStream = new FileReader("test.txt");
            // Reading sourcefile and writing content to target file character by character.
            int temp;

            while (( temp = sourceStream.read())!= -1 )
                System.out.println((char)temp);
        }
        finally {

            // Closing stream as no longer in use
            if (sourceStream != null)
                sourceStream.close();
        }
    }
}
```

WIPRO SQL CONCEPTS

23 October 2025 09:13

Absolutely, Attada! Let's break down each of the SQL concepts from your image into **clear explanations** with **realistic examples** so you can understand how they work in practice.

1. Database, Table, Row, Column

◇ Database

A **database** is a structured collection of data. Think of it as a digital filing cabinet.

```
CREATE DATABASE CompanyDB;
```

◇ Table

A **table** is like a spreadsheet inside the database — it stores data in rows and columns.

```
CREATE TABLE Employees (
  ID INT,
  Name VARCHAR(100),
  Department VARCHAR(50)
);
```

◇ Row

A **row** is a single record in a table.

```
INSERT INTO Employees VALUES (1, 'Alice', 'HR');
```

◇ Column

A **column** defines the type of data stored (like Name, ID, etc.).

2. Data Types

SQL supports various **data types** to define the kind of data each column can hold.

Data Type	Description	Example
INT	Integer numbers	101, -5, 2000
VARCHAR	Variable-length text	'John', 'HR'
DATE	Date values	'2023-10-23'
DECIMAL	Fixed-point numbers	99.99
BOOLEAN	True/False	TRUE, FALSE

```
CREATE TABLE Products (
  ProductID INT,
  Name VARCHAR(100),
  Price DECIMAL(10,2),
  LaunchDate DATE
);
```

3. DDL (Data Definition Language)

Used to **define or modify** database structures.

◇ CREATE

Creates a new table or database.

```
CREATE TABLE Students (
  RollNo INT,
  Name VARCHAR(50)
);
```

◇ ALTER

Modifies an existing table.

```
ALTER TABLE Students ADD Age INT;
```

◇ DROP

Deletes a table or database.

```
DROP TABLE Students;
```

4. DML (Data Manipulation Language)

Used to **modify data** inside tables.

◇ INSERT

Adds new records.
INSERT INTO Students VALUES (1, 'Rahul', 20);

◇ UPDATE

Modifies existing records.
UPDATE Students SET Age = 21 WHERE RollNo = 1;

◇ DELETE

Removes records.
DELETE FROM Students WHERE RollNo = 1;

5. DQL (Data Query Language)

Used to **retrieve data** from tables.

◇ SELECT

Fetches data.
SELECT * FROM Students;

◇ WHERE

Filters results.
SELECT * FROM Students WHERE Age > 18;

◇ ORDER BY

Sorts results.
SELECT * FROM Students ORDER BY Name ASC;

◇ GROUP BY

Groups rows with the same values.
SELECT Department, COUNT(*) FROM Employees GROUP BY Department;

SQL Constraints

Constraints enforce **rules on data** in tables to maintain **data integrity**.

1. PRIMARY KEY

- Uniquely identifies each row.
- Cannot be NULL or duplicated.

```
CREATE TABLE Employees (
  EmpID INT PRIMARY KEY,
  Name VARCHAR(100),
  Department VARCHAR(50)
);
```

🧠 Example:

```
INSERT INTO Employees VALUES (101, 'Alice', 'HR'); -- ☑
INSERT INTO Employees VALUES (101, 'Bob', 'IT');   -- ❌ Duplicate EmpID
```

2. FOREIGN KEY

- Links one table to another.
- Enforces referential integrity.

```
CREATE TABLE Departments (
  DeptID INT PRIMARY KEY,
  DeptName VARCHAR(50)
);
CREATE TABLE Employees (
  EmpID INT PRIMARY KEY,
  Name VARCHAR(100),
  DeptID INT,
  FOREIGN KEY (DeptID) REFERENCES Departments(DeptID)
);
```

🧠 Example:

```
INSERT INTO Departments VALUES (1, 'HR');
INSERT INTO Employees VALUES (101, 'Alice', 1); -- ☑
INSERT INTO Employees VALUES (102, 'Bob', 99);  -- ❌ DeptID 99 doesn't exist
```

3. UNIQUE

- Ensures all values in a column are different.

```
CREATE TABLE Users (
  UserID INT PRIMARY KEY,
  Email VARCHAR(100) UNIQUE
```

);
🤔 Example:
INSERT INTO Users VALUES (1, 'alice@example.com'); -- ☑
INSERT INTO Users VALUES (2, 'alice@example.com'); -- ❌ Duplicate email

4. NOT NULL

- Prevents NULL values in a column.

```
CREATE TABLE Products (
  ProductID INT PRIMARY KEY,
  Name VARCHAR(100) NOT NULL
);
```

🤔 Example:
INSERT INTO Products VALUES (1, NULL); -- ❌ Name cannot be NULL

5. DEFAULT

- Sets a default value if none is provided.

```
CREATE TABLE Orders (
  OrderID INT PRIMARY KEY,
  Status VARCHAR(20) DEFAULT 'Pending'
);
```

🤔 Example:
INSERT INTO Orders (OrderID) VALUES (1001); -- Status = 'Pending'

6. CHECK

- Restricts values based on a condition.

```
CREATE TABLE Students (
  RollNo INT PRIMARY KEY,
  Age INT CHECK (Age >= 18)
);
```

🤔 Example:
INSERT INTO Students VALUES (1, 17); -- ❌ Age < 18

SQL Joins

Joins combine rows from **two or more tables** based on a related column.

Sample Tables

Employees

EmpID	Name	DeptID
101	Alice	1
102	Bob	2
103	Carol	NULL

Departments

DeptID	DeptName
1	HR
2	IT
3	Finance

1. INNER JOIN

Returns rows with matching values in both tables.

```
SELECT e.Name, d.DeptName
FROM Employees e
INNER JOIN Departments d ON e.DeptID = d.DeptID;
```

🤔 Result:

Name	DeptName
Alice	HR
Bob	IT

2. LEFT JOIN

Returns all rows from the left table, and matched rows from the right.
SELECT e.Name, d.DeptName
FROM Employees e
LEFT JOIN Departments d ON e.DeptID = d.DeptID;
🤔 Result:

Name	DeptName
Alice	HR
Bob	IT
Carol	NULL

3. RIGHT JOIN
Returns all rows from the right table, and matched rows from the left.
SELECT e.Name, d.DeptName
FROM Employees e
RIGHT JOIN Departments d ON e.DeptID = d.DeptID;
🤔 Result:

Name	DeptName
Alice	HR
Bob	IT
NULL	Finance

4. FULL JOIN
Returns all rows when there is a match in either table.
SELECT e.Name, d.DeptName
FROM Employees e
FULL JOIN Departments d ON e.DeptID = d.DeptID;
🤔 Result:

Name	DeptName
Alice	HR
Bob	IT
Carol	NULL
NULL	Finance

5. ON vs USING
- **ON**: Explicit column match, works with different column names.
- **USING**: Simpler syntax, only works when column names are the same.

```
-- Using ON
SELECT e.Name, d.DeptName
FROM Employees e
JOIN Departments d ON e.DeptID = d.DeptID;
-- Using USING
SELECT e.Name, d.DeptName
FROM Employees e
JOIN Departments d USING (DeptID);
```

Absolutely, Attada! Let's break down the SQL concepts from your image into **clear, detailed explanations** with **realistic examples** using sample tables. These are essential for mastering data analysis and reporting.

1. Aggregations
Aggregation functions summarize data across multiple rows.

◇ COUNT
Returns the number of rows.
SELECT COUNT(*) FROM Employees;
-- Output: Total number of employees

◇ SUM
Adds up numeric values.
SELECT SUM(Salary) FROM Employees;
-- Output: Total salary paid to all employees

◇ AVG

Calculates the average of numeric values.
SELECT AVG(Salary) FROM Employees;
-- Output: Average salary

◇ MIN / MAX

Finds the smallest or largest value.
SELECT MIN(Salary), MAX(Salary) FROM Employees;
-- Output: Lowest and highest salary

◇ GROUP BY

Groups rows that have the same value in a column.
Sample Table: Employees

EmpID	Name	Department	Salary
101	Alice	HR	50000
102	Bob	IT	60000
103	Carol	HR	55000
104	Dave	IT	62000

SELECT Department, AVG(Salary)
FROM Employees
GROUP BY Department;
🧠 Output:

Department	AVG(Salary)
HR	52500
IT	61000

◇ HAVING

Filters groups after aggregation (like WHERE for groups).
SELECT Department, COUNT(*) AS TotalEmployees
FROM Employees
GROUP BY Department
HAVING COUNT(*) > 1;
🧠 Output:

Department	TotalEmployees
HR	2
IT	2

2. Subqueries & Views

◇ Subqueries (Nested SELECT)

A query inside another query.
SELECT Name, Salary
FROM Employees
WHERE Salary > (
  SELECT AVG(Salary) FROM Employees
);
🧠 Output: Employees earning above average salary.

◇ CREATE VIEW

Creates a virtual table based on a query.
CREATE VIEW HighEarners AS
SELECT Name, Salary
FROM Employees
WHERE Salary > 60000;
You can now query it like a table:
SELECT * FROM HighEarners;

◇ DROP VIEW

Deletes a view.
DROP VIEW HighEarners;

ACID Properties in DBMS

Transactions are fundamental operations that allow us to modify and retrieve data. However, to ensure the integrity of a database, it is important that these transactions are executed in a way that maintains consistency, correctness, and reliability even in case of failures / errors. This is where the ACID properties come into play.

ACID stands for Atomicity, Consistency, Isolation, and Durability.

Properties Of ACID:
There are Four Properties of ACID

1. Atomicity

Atomicity means a transaction is all-or-nothing either all its operations succeed, or none are applied. If any part fails, the entire transaction is rolled back to keep the database consistent.

- **Commit**: If the transaction is successful, the changes are permanently applied.
- **Abort/Rollback**: If the transaction fails, any changes made during the transaction are discarded.
  Example: Consider the following transaction **T** consisting of **T1** and **T2** : Transfer of $100 from account **X** to account **Y** .

Atomicity
If the transaction fails after completion of T1 but before completion of T2, the database would be left in an inconsistent state. With Atomicity, if any part of the transaction fails, the entire process is rolled back to its original state, and no partial changes are made.

2. Consistency

Consistency in transactions means that the database must remain in a valid state before and after a transaction.

- A valid state follows all defined rules, constraints, and relationships (like primary keys, foreign keys, etc.).
- If a transaction violates any of these rules, it is rolled back to prevent corrupt or invalid data.
- If a transaction deducts money from one account but doesn't add it to another (in a transfer), it violates consistency.
  Example: Suppose the sum of all balances in a bank system should always be constant. Before a transfer, the total balance is $700. After the transaction, the total balance should remain $700. If the transaction fails in the middle (like updating one account but not the other), the system should maintain its consistency by rolling back the transaction.

Total before T occurs = 500 + 200 = 700 .

Total after T occurs = 400 + 300 = 700 .

Consistency

3. Isolation

Isolation ensures that transactions run independently without affecting each other. Changes made by one transaction are not visible to others until they are committed.

It ensures that the result of concurrent transactions is the same as if they were run one after another, preventing issues like:

- **Dirty reads:** reading uncommitted data
- **Non-repeatable reads:** data changes between two reads
- **Phantom reads:** new rows appear during a transaction
  Example: Consider two transactions T and T''.
- **X = 500**, **Y = 500**

Isolation

Explanation:

1. Transaction T:

- T wants to transfer $50 from X to Y.
- T reads **Y** (value: 500), deducts $50 from X (new X = 450), and adds $50 to Y (new Y = 550).

2. Transaction T'':

- T**''** starts and reads X (500) and Y (500).
- It calculates the sum: 500 + 500 = 1000.
- Meanwhile, values of X and Y change to 450 and 550 respectively.
- So, the correct sum should be 450 + 550 = 1000.
- Isolation ensures that T'' does not read outdated values while another transaction (T) is still in progress.
- Transactions should be independent, and T'' should access the final values only after T commits.
- This avoids inconsistent results, like the incorrect sum calculated by T''.

4. Durability:

Durability ensures that once a transaction is committed, its changes are permanently saved, even if the system fails. The data is stored in non-volatile memory, so the database can recover to its last committed state without losing data.

Example: After successfully transferring money from Account A to Account B, the changes are stored on disk. Even if there is a crash immediately after the commit, the transfer details will still be intact when the system recovers, ensuring durability.

How ACID Properties Impact DBMS Design and Operation

The ACID properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

1. Data Integrity and Consistency

ACID properties safeguard the data integrity of a DBMS by ensuring that transactions either complete successfully or leave no trace if interrupted. They prevent partial updates from corrupting the data and ensure that the database transitions only between valid states.

2. Concurrency Control

ACID properties provide a solid framework for managing concurrent transactions. Isolation ensures that transactions do not interfere with each other, preventing data anomalies such as lost updates, temporary inconsistency, and uncommitted data.

3. Recovery and Fault Tolerance

Durability ensures that even if a system crashes, the database can recover to a consistent state. Thanks to the Atomicity and Durability properties, if a transaction fails midway, the database remains in a consistent state.

Property	Responsibility for maintaining properties
Atomicity	Transaction Manager
Consistency	Application programmer
Isolation	Concurrency Control Manager
Durability	Recovery

Critical Use Cases for ACID in Databases

In modern applications, ensuring the reliability and consistency of data is crucial. ACID properties are fundamental in sectors like:

- **Banking**: Transactions involving money transfers, deposits, or withdrawals must maintain strict consistency and durability to prevent errors and fraud.
- **E-commerce**: Ensuring that inventory counts, orders, and customer details are handled correctly and

consistently, even during high traffic, requires ACID compliance.
- **Healthcare**: Patient records, test results, and prescriptions must adhere to strict consistency, integrity, and security standards.

What is a Stored Procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.
So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.
You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedure Syntax

```
CREATE PROCEDURE procedure_name

AS

sql_statement

GO;
```

Execute a Stored Procedure

```
EXEC procedure_name;
```

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Stored Procedure Example

The following SQL statement creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers

AS

SELECT * FROM Customers

GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers;
```


Stored Procedure With One Parameter

The following SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)

AS

SELECT * FROM Customers WHERE City = @City

GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers @City = 'London';
```

Stored Procedure With Multiple Parameters

Setting up multiple parameters is very easy. Just list each parameter and the data type separated by a comma as shown below.
The following SQL statement creates a stored procedure that selects Customers from a particular City with a particular PostalCode from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode nvarchar(10)

AS

SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode

GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';
```

23 October 2025 09:21

Java Stack vs Heap Memory Allocation

Last Updated : 08 Oct, 2025

In Java, memory allocation is primarily divided into two categories, i.e., Stack and Heap memory. Both are used for different purposes, and they have different characteristics.

- **Stack Memory:** Stores primitive local variables, method call information, and references to objects during program execution.
- **Heap Memory:** Stores actual objects and dynamic data allocated at runtime. Objects created with new are placed here, and this memory is managed by the Garbage Collector.

```
class Employee {
    int id;        // Primitive stored inside the object in Heap
    String name;   // Reference points to String object in Heap
    double salary; // Primitive stored inside the object in Heap

    public Employee(int id, String name, double salary) {
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public void display() {
        System.out.println("Employee ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Salary: " + salary);
    }
}

public class Main {
    // Reference variable 'emp' stored in Stack
    public static void display(Employee emp) {
        emp.display(); // Accesses Employee object in Heap
    }

    public static void main(String[] args) {
        // 'emp1' and 'emp2' references in Stack, objects in Heap
        Employee emp1 = new Employee(101, "Maddy", 50000.0);
        Employee emp2 = new Employee(102, "Maddy", 60000.0);

        display(emp1);
        display(emp2);
    }
}
```

Output

```
Employee ID: 101
Name: Maddy
Salary: 50000.0
Employee ID: 102
Name: Maddy
```

```
Salary: 60000.0
```
Memory representation for the above example:

Memory representation

- **emp1 and emp2:** references stored on stack.
- **new Employee(...):** objects stored in heap.
- **"Maddy"**; stored once in String Pool (heap).
- **Both emp1.name and emp2.name**: point to the same string object.
- **display(emp1):** creates a stack frame, parameter e points to emp1.
- **(emp1.name == emp2.name):** true since both refer to the same pooled string.

Stack Memory Allocation

Stack memory is used for method calls, local variables, and references. Memory is automatically allocated when a method starts and cleared when it ends. Data exists only during the method's execution. If the stack runs out of space, a StackOverflowError occurs.

```
class Geeks {
    public static int add(int a, int b) {
        int res = a + b;  // local variable in stack
        return res;
    }

    public static void main(String[] args) {
        int a = 10;   // stored in stack
        int b = 20;   // stored in stack
        int sum = add(a, b);
        System.out.println("The sum is: " + sum);
    }
}
```

Output

```
The sum is: 30
```

Heap Memory Allocation

Heap memory is used for objects and instance variables created using the new keyword. The size depends on the class structure. The Garbage Collector manages this area by removing unused objects.

Heap Memory Regions:

- **Young Generation:** Stores newly created objects.
- **Survivor Space:** Holds objects that survive garbage collection from Eden space.
- **Old Generation:** Stores long-lived objects.
- **Metaspace (earlier Permanent Generation):** Stores class metadata.
- **Code Cache:** Stores compiled/optimized code.

Example: Object in heap

```
Scanner sc = new Scanner(System.in);
```

Here, the Scanner object is in the heap, while the reference sc is in the stack.

Note: *Garbage collection in the heap ensures automatic memory management.*

Difference Between Stack and Heap Memory in Java

Stack	Heap
Memory is allocated in a contiguous block.	Memory is allocated in any random order.
Allocation and deallocation are automatic (handled by compiler).	Allocation and deallocation are manual (handled programmer).
It is less costly	It is more costly
Its implementation is easy	Its implementation is hard
Access time is faster	Access time is slower
Limited memory size may lead to shortage issues.	Can suffer from memory fragmentation due to dynamic allocation.
Stack provides excellent memory locality	Heap is adequate but not as efficient
Thread safe, data stored can only be accessed by the owner	Not thread safe, data stored is visible to all threads
Stack is fixed in size	Heap allows resizing as needed
Stack uses a linear data structure	Heap uses a hierarchical data structure
Static memory allocation is preferred in an array.	Heap memory allocation is preferred in the linked list.
Smaller than heap memory.	Larger than stack memory.