CORE JAVA FUNDAMENTALS

prepared By K.Gowthaman MCA - 9894083890

Java virtual Machine

What Is the JVM (Java Virtual Machine)?
The **Java Virtual Machine (JVM)** is a **software-based engine** that runs Java bytecode.
It provides a **runtime environment** that allows Java programs to be **platform-independent**.

JVM = A virtual computer inside your computer that runs Java programs.

Java Execution Flow:

1. **You write source code** → Program.java
2. **Compiler (javac)** compiles it into **bytecode** → Program.class
3. **JVM** executes that bytecode on your machine.

Program.java → (javac) → Program.class → (JVM) → Output

JVM – Executes Byte Code
JRE = JVM + Libraries
JDK = JRE + Development Tools

Java data types – primitive data type

Type	Size	Default Value	Example	Description
byte	1 byte (8 bits)	0	byte a = 10;	Stores small integers (-128 to 127). Useful for saving memory.
short	2 bytes	0	short s = 1000;	Stores medium-range integers (-32,768 to 32,767).
int	4 bytes	0	int i = 100000;	Most commonly used integer type.
long	8 bytes	0L	long l = 10000000000L;	Stores large integers. Needs suffix **L**.
float	4 bytes	0.0f	float f = 10.5f;	Single-precision decimal numbers. Needs suffix **f**.
double	8 bytes	0.0d	double d = 99.99;	Double-precision decimal numbers (default for decimals).
char	2 bytes	'\u0000'	char c = 'A';	Stores a single Unicode character.
boolean	1 bit (logical)	false	boolean flag = true;	Stores true or false.

Java data types (non primitive data types)

Type	Example	Description
String	String name = "Gowthaman";	Sequence of characters.
Array	int[] arr = {1, 2, 3};	Collection of similar elements.
Class	class Car { }	User-defined type with methods and variables.
Interface	interface Vehicle { }	Abstract type defining methods a class must implement.
Enum	enum Level { LOW, MEDIUM, HIGH }	Special type representing a group of constants.

Variables

What Is a Variable?

variable acts like a container that holds data of a specific **data type**.

Syntax:

dataType variableName = value;

Example:

int age = 25; String name = "Gowthaman";

Types of Variables in Java

Type	Declared Inside	Scope	Lifetime	Example
Local Variable	A method, constructor, or block	Only inside that method/block	Until the method finishes	int count = 10; inside a method
Instance Variable	Inside a class but outside any method	For each object of the class	As long as the object exists	String name;
Static Variable (Class Variable)	Inside a class with static keyword	Shared among all objects	Until the program ends	static int count;

Keywords

- **What Are Keywords?**
- Keywords tell the Java compiler how to interpret and execute your code.
- There are **around 67 keywords** in Java (including newer ones from Java 9+).

Data Type Keywords

Keyword	Description
byte	8-bit integer type
short	16-bit integer type
int	32-bit integer type
long	64-bit integer type
float	32-bit floating-point
double	64-bit floating-point
char	16-bit Unicode character
boolean	true or false

Keywords

Access Modifiers

Keyword	Description
public	Accessible from anywhere
private	Accessible only within the class
protected	Accessible within package and subclasses
default *(no keyword)*	Accessible only within the same package

Control Flow Keywords

Keyword	Description
if, else	Conditional statements
switch, case, default	Multi-way branching
while, do, for	Loops
break	Exits a loop or switch
continue	Skips current iteration in a loop
return	Exits from a method and returns a value

Class, Object, and Inheritance Keywords

Keyword	Description
class	Declares a class
interface	Declares an interface
extends	Used to inherit a class
implements	Used to implement an interface
new	Creates a new object
this	Refers to the current object
super	Refers to the parent class object
abstract	Defines an abstract class or method
final	Used to make constant, prevent inheritance, or override
static	Belongs to the class, not to an object

Exception Handling Keywords

Keyword	Description
try	Starts a block of code to test for errors
catch	Catches exceptions
finally	Always executes after try-catch
throw	Throws an exception explicitly
throws	Declares exceptions in method signature

Package and Import Keywords

Keyword	Description
package	Defines a package
import	Imports other Java packages or classes

Others

Keyword	Description
void	Specifies no return value
enum	Defines a set of named constants
synchronized	Ensures thread-safe code
volatile	Tells the compiler a variable can change unexpectedly
transient	Excludes a field from serialization
assert	For debugging purposes
instanceof	Tests if an object is an instance of a class
native	Specifies a method written in another language (like C)

Operators

In **Java**, **operators** are special symbols used to **perform operations** on variables and values — like arithmetic, comparison, or logic.

Types of Operators in Java

Category	Description	Examples
1. Arithmetic Operators	Perform basic math operations	+, -, *, /, %
2. Unary Operators	Work on a single operand	++, --, +, -, !
3. Assignment Operators	Assign values to variables	=, +=, -=, *=, /=, %=
4. Relational Operators	Compare two values	==, !=, >, <, >=, <=
5. Logical Operators	Combine multiple conditions	&&, || ,!
6. Bitwise Operators	Perform bit-level operations	&, | ^ << >>
7. Ternary Operator	Shorthand for if-else	?:
8. Type Comparison Operator	Check object type	instanceof

```
String name = "Gowthaman";
System.out.println(name instanceof String); // true
```

expressions

In **Java**, an **expression** is a **combination of variables, constants, operators, and method calls** that produces a **single value** when evaluated.

Types of Expressions in Java

Type	Example	Description
1. Arithmetic Expression	a + b - c * d	Performs mathematical operations
2. Relational Expression	a > b	Compares two values and returns true or false
3. Logical Expression	(a > b) && (b < c)	Combines two or more conditions
4. Assignment Expression	x = 10	Assigns a value to a variable
5. Conditional (Ternary) Expression	(a > b) ? a : b	Returns one of two values based on a condition
6. Bitwise Expression	a & b	Operates on bits
7. Object Creation Expression	new Student()	Creates a new object
8. Method Call Expression	sum(a, b)	Calls a method and returns its value
9. String Expression	"Hello " + name	Concatenates strings

Control Statements

In **Java**, **control statements** are used to **control the flow of execution** in a program — deciding **which statements to execute**, **how many times**, and **under what conditions**.

Types of Control Statements in Java
Java control statements are grouped into three main categories:

Category	Description	Example Keywords
1. Decision-making statements	Execute code blocks based on conditions	if, if-else, nested if, switch
2. Looping statements (Iteration)	Repeat a block of code multiple times	for, while, do-while, for-each
3. Jump statements	Transfer control to another part of code	break, continue, return

```
if (age >= 18) {
   System.out.println("You are an adult.");
}
```

```
if (age >= 18) {
   System.out.println("Eligible to vote.");
} else {
   System.out.println("Not eligible to vote.");
}
```

```
if (marks >= 90) {
   System.out.println("Grade A");
} else if (marks >= 75) {
   System.out.println("Grade B");
} else {
   System.out.println("Grade C");
}
```

```
if (num > 0) {
   if (num % 2 == 0) {
      System.out.println("Positive Even");
   }
}
```

```
int day = 3;
switch (day) {
   case 1: System.out.println("Monday"); break;
   case 2: System.out.println("Tuesday"); break;
   case 3: System.out.println("Wednesday"); break;
   default: System.out.println("Invalid day");
}
```

From Java 14 onward,
you can also use the **enhanced switch expression**:

```
String result = switch(day)
 {
 case 1 -> "Monday";
 case 2 -> "Tuesday";
 case 3 -> "Wednesday";
 default -> "Invalid day";
 };
```

```java
for (int i = 1; i <= 5; i++) {
   System.out.println("Count: " + i);
}

int i = 1;
while (i <= 5) {
   System.out.println(i);
   i++;
}

int i = 1;
do {
   System.out.println(i);
   i++;
} while (i <= 5);


int[] numbers = {10, 20, 30};
for (int num : numbers) {
   System.out.println(num);
}
```

```java
for (int i = 1; i <= 5; i++) {
   if (i == 3) break;
   System.out.println(i);
}

for (int i = 1; i <= 5; i++) {
   if (i == 3) continue;
   System.out.println(i);
}


public int sum(int a, int b) {
   return a + b;
}
```

What Is a Class in Java?

A **class** is a **blueprint** or **template** for creating objects.
It defines **properties (variables)** and **behaviors (methods)** that the objects will have.

```
class Car {
    // Data members (fields or attributes)
    String color;
    String model;
    int year;

    // Method (behavior)
    void displayInfo() {
        System.out.println("Model: " + model + ", Color: " + color + ", Year: " + year);
    }
}
```

What Is an Object in Java?

An **object** is an **instance** of a class.
When a class is defined, no memory is allocated until we create an **object** using the new keyword.

```
public class Main {
  public static void main(String[] args) {
    // Creating objects of the Car class
    Car car1 = new Car();
    Car car2 = new Car();

    // Assigning values
    car1.color = "Red";
    car1.model = "Tesla";
    car1.year = 2024;

    car2.color = "Blue";
    car2.model = "BMW";
    car2.year = 2023;

    // Calling method
    car1.displayInfo();
    car2.displayInfo();
  }
}
```

Class vs Object — Difference Table

Feature	Class	Object
Definition	Blueprint for creating objects	Instance of a class
Memory	No memory allocation	Occupies memory
Keyword	Declared using class	Created using new
Example	class Car {}	Car myCar = new Car();
Count	Defined once	Many objects can be created

Constructor in Java

A **constructor** is a special method used to **initialize objects**.
It has the **same name as the class** and **no return type**.

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

   // Constructor
   Student(String n, int a) {
      name = n;
      age = a;
   }

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

```
public class Main {
   public static void main(String[] args) {
      Student s1 = new Student("Gowthaman", 20);
      Student s2 = new Student("Anita", 19);
      s1.display();
      s2.display();
   }
}
```

In **Java**, a **constructor** is a **special method** that is **used to initialize objects**.
It is called **automatically** when an object is created — **no need to call it manually**.

What Is a Constructor?

✅ **Definition:**

A **constructor** is a block of code that:

- Has the **same name as the class**.
- **Does not have a return type** (not even void).
- Is **automatically invoked** when an object is created.

```
class ClassName {
   // Constructor
   ClassName() {
      // Initialization code
   }
}
```

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

   // Constructor
   Student(String n, int a) {
      name = n;
      age = a;
   }

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

public class Main {
   public static void main(String[] args) {
      Student s1 = new Student("Gowthaman", 20);
      Student s2 = new Student("Anita", 19);
      s1.display();
      s2.display();
   }
}
```

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

    Student(String n, int a) {
        name = n;
        age = a;
    }

    // Copy constructor
    Student(Student s) {
        name = s.name;
        age = s.age;
    }

    void display() {
        System.out.println(name + " - " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Gowthaman", 20);
        Student s2 = new Student(s1); // Copy s1 data to s2
        s2.display();
    }
}
```

```
class Box {
    int width, height;

    Box() {
        width = height = 10;
    }

    Box(int w, int h) {
        width = w;
        height = h;
    }

    void display() {
        System.out.println("Width: " + width + ", Height: " + height);
    }
}

public class Main {
    public static void main(String[] args) {
        Box b1 = new Box();       // Calls default constructor
        Box b2 = new Box(5, 15);   // Calls parameterized constructor
        b1.display();
        b2.display();
    }
}
```

In **Java**, **access control** (or **access modifiers**) defines **how accessible** classes, methods, variables, and constructors are **to other classes**.

What Are Access Modifiers?

Access modifiers are **keywords** that set the **visibility level** of classes and their members (fields, methods, etc.).

There are **4 types** of access levels in Java:

Access Level	Keyword	Accessible Within Class	Within Package	Subclass (outside package)	Other Packages
Public	public	✓	✓	✓	✓
Protected	protected	✓	✓	✓	✕
Default (Package-private)	(no keyword)	✓	✓	✕	✕
Private	private	✓	✕	✕	✕

Method Overloading is a feature in Java that allows a class to have **multiple methods with the same name** but **different parameters** (number, type, or order).

What Is Method Overloading?

When **two or more methods** in the **same class** share the **same name** but have **different parameter lists**, it is called **method overloading**.

```
class MathUtil {
   // Method 1
   int add(int a, int b) {
      return a + b;
   }

   // Method 2 (different number of parameters)
   int add(int a, int b, int c) {
      return a + b + c;
   }

   // Method 3 (different data type)
   double add(double a, double b) {
      return a + b;
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      MathUtil obj = new MathUtil();

      System.out.println(obj.add(10, 20));       // Calls Method 1
      System.out.println(obj.add(10, 20, 30));   // Calls Method 2
      System.out.println(obj.add(5.5, 6.5));     // Calls Method 3
   }
}
```

In Java, the keyword static is used to define class-level members — that is, members that belong to the class itself, not to any particular object.

What Are Static Members?
When a member (variable, method, block, or nested class) is declared with the static keyword, it belongs to the class rather than to instances of the class.
☞ That means:

- All objects share the same static member.
- You don't need to create an object to access it.

Types of Static Members

Type	Description
Static Variables (Fields)	Shared by all objects of a class
Static Methods	Can be called without creating an object
Static Blocks	Used to initialize static data
Static Nested Classes	A class defined inside another class using static

What Is an Array in Java?

An **array** is a container object that holds a fixed number of values of **a single type**.

Declaring Arrays

There are two main ways to declare arrays:

```
// Method 1: Declare then allocate

int[] arr;

arr = new int[3];

// Method 2: Declare and allocate together

 int[] arr = new int[3];
```

```
int[] arr = new int[3];
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
```

```
int[] arr = {10, 20, 30};
```

```
System.out.println(arr.length); // Outputs 3
```

Multidimensional Arrays

Java supports arrays of arrays (like matrices):

```
int[][] matrix = { {1, 2, 3}, {4, 5, 6} };
System.out.println(matrix[1][2]); // Outputs 6
```

```
import java.util.Arrays;

int[] nums = {3, 1, 4, 1, 5};

Arrays.sort(nums);
System.out.println(Arrays.toString(nums)); // [1, 1, 3, 4, 5]

int index = Arrays.binarySearch(nums, 4); // Search for element 4
System.out.println(index); // 3
```

Common String Methods

Method	Description	Example	Output
length()	Returns number of characters	"Java".length()	4
charAt(i)	Returns character at index	"Java".charAt(2)	'v'
substring(a, b)	Extracts substring from index a to b-1	"Hello".substring(1,4)	"ell"
toUpperCase()	Converts to uppercase	"java".toUpperCase()	"JAVA"
toLowerCase()	Converts to lowercase	"JAVA".toLowerCase()	"java"
trim()	Removes leading and trailing spaces	" Hello ".trim()	"Hello"
equals()	Compares content	"Java".equals("java")	false
equalsIgnoreCase()	Ignores case	"Java".equalsIgnoreCase("java")	true
contains()	Checks if substring exists	"Programming".contains("gram")	true
replace(a,b)	Replaces characters	"Java".replace('a','o')	"Jovo"

String Comparison

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = new String("Hello");

System.out.println(s1 == s2);      // true (same reference in pool)
System.out.println(s1 == s3);      // false (different object)
System.out.println(s1.equals(s3));  // true (same content)
```

StringBuilder and StringBuffer

Since Strings are immutable, use these classes for **mutable strings** (modifiable):

```
StringBuilder sb = new StringBuilder("Hello"); s
b.append(" World");
System.out.println(sb); // Hello World
```

Class	Thread-Safe	Performance
String	Immutable	Medium
StringBuilder	Not Thread-Safe	Fastest
StringBuffer	Thread-Safe	Slightly Slower

What Is Inheritance?

Inheritance is the process by which one class **acquires the properties and behaviors (fields and methods)** of another class.
It allows **code reuse** and establishes a **parent–child relationship** between classes.

```
class Parent {
    // parent class members
}

class Child extends Parent {
    // child class members
}
```

Why Use Inheritance?

- Promotes **code reusability**
- Supports **method overriding**
- Provides **hierarchical classification**
- Simplifies **maintenance and scalability**

Single Inheritance

A single class inherits from one superclass.

```
class A {
   void displayA() { System.out.println("Class A"); }
}
class B extends A {
   void displayB() { System.out.println("Class B"); }
}
```

Multilevel Inheritance

A class inherits from another derived class (a chain).

```
class A {
   void msgA() { System.out.println("Class A"); }
}
class B extends A {
   void msgB() { System.out.println("Class B"); }
}
class C extends B {
   void msgC() { System.out.println("Class C"); }
}
```

Hierarchical Inheritance

Multiple classes inherit from the same parent.

```
class Animal {
   void eat() { System.out.println("Eating..."); }
}
class Dog extends Animal {
   void bark() { System.out.println("Barking..."); }
}
class Cat extends Animal {
   void meow() { System.out.println("Meowing..."); }
}
```

Multiple Inheritance (Not Supported with Classes)

Java **does not support multiple inheritance** using classes (to avoid ambiguity known as the *Diamond Problem*).

Hybrid Inheritance

Combination of two or more types (like multilevel + hierarchical).

Java supports this **only through interfaces**, not with classes.

super Keyword in Inheritance

Used to refer to the **immediate parent class**.

```
class Animal {
   Animal() {
      System.out.println("Animal constructor");
   }
   void eat() { System.out.println("Eating..."); }
}

class Dog extends Animal {
   Dog() {
      super(); // Calls parent constructor
      System.out.println("Dog constructor");
   }
   void eat() {
      super.eat(); // Calls parent method
      System.out.println("Eating bread...");
   }
}
```

constructors in inheritance in Java.
This is a very important concept that helps you understand **how objects are created** in a class hierarchy.

Do subclasses inherit constructors?
☞ **No**, constructors are **not inherited** in Java.
However, when a subclass object is created, **the constructor of the parent class is automatically called first**, before the child class’s constructor.
This ensures that the parent’s part of the object is initialized properly.

```
class Parent {
   Parent() {
      System.out.println("Parent constructor called");
   }
}

class Child extends Parent {
   Child() {
      System.out.println("Child constructor called");
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      Child c = new Child();
   }
}
```

Output:

```
Parent constructor called
Child constructor called
```

What Is a String in Java?

String in Java is an **object** that represents a sequence of characters.
It is part of the java.lang package.
Example:

```
String name = "Gowthaman";
```

Creating Strings

You can create strings in two ways:

1.Using string literals (recommended)

```
String s1 = "Hello";
```

☞ **Stored in the String pool (a special memory area in Java).**

2.Using the new keyword

```
String s2 = new String("Hello");
```

☞ **Always creates a new object in heap m**emory, even if the same string exists in the pool.

String Immutability

Strings are **immutable** — once created, they cannot be changed.

```
String s = "Java";
s.concat(" Programming");  // Creates a new String, doesn't modify s
System.out.println(s);     // Output: Java
```

Method Overriding in Java

Method Overriding is one of the key features of **Object-Oriented Programming (OOP)** in Java.
It allows a **subclass (child class)** to **provide its own version** of a method that is **already defined in its superclass (parent class)**.

What Is Method Overriding?

When a **child class defines a method** that has the **same name, return type, and parameters** as a **method in its parent class**,
the child's version **overrides** (replaces) the parent's version.

Feature	Overriding	Overloading
Definition	Redefining a method in a child class	Defining multiple methods with same name but different parameters
Inheritance	Required	Not required
Parameters	Must be same	Must be different
Return Type	Same or subclass	Can differ
Runtime/Compile Time	Runtime (Dynamic Binding)	Compile Time (Static Binding)

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    // Overriding the sound() method
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog(); // Parent reference, child object
        a.sound();  // Calls Dog's sound() method
    }
}
```

interfaces in Java, one of the key OOP features for achieving **abstraction** and **multiple inheritance**.

What Is an Interface in Java?
An **interface** in Java is like a **contract** — it defines a set of **methods** that a class must implement.
It contains **abstract methods** (without implementation) and **constants**.
Think of it as a **blueprint** for a class.

```
interface InterfaceName {
   // Constant (public, static, final by default)
   int VALUE = 10;

   // Abstract methods (public and abstract by default)
   void method1();
   void method2();
}
```

```
class MyClass implements InterfaceName {
   public void method1() {
      System.out.println("Method 1 implemented");
   }

   public void method2() {
      System.out.println("Method 2 implemented");
   }
}
```

Why Use Interfaces?

- To achieve **abstraction** (hide implementation details)
- To achieve **multiple inheritance**
- To define **common behavior** for unrelated classes
- To establish **contracts** that multiple classes can follow

Interface Inheritance

Interfaces can **inherit from other interfaces** using the extends keyword.

```
interface A {
   void methodA();
}

interface B extends A {
   void methodB();
}

class C implements B {
   public void methodA() { System.out.println("Method A"); }
   public void methodB() { System.out.println("Method B"); }
}
```

Functional Interfaces (Java 8+)

A **functional interface** has **exactly one abstract method**.

It can be used with **lambda expressions**.

```
@FunctionalInterface
interface Greeting {
   void sayHello(String name);
}

public class FunctionalInterfaceExample {
   public static void main(String[] args) {
      Greeting g = (name) -> System.out.println("Hello, " + name);
      g.sayHello("Gowthaman");
   }
}
```

Dynamic Method Dispatch (also known as **Runtime Polymorphism**).

What Is Dynamic Method Dispatch?

Dynamic Method Dispatch is the mechanism by which a **call to an overridden method** is resolved **at runtime**, not at compile time.

In simple words.

The **method that gets executed depends on the type of object** (not the reference variable).

Key Idea

- A **reference variable** of a **parent class** can refer to an **object of a child class**.
- When an **overridden method** is called through the parent reference, **Java decides at runtime** which version (parent or child) to execute.

Why Use Dynamic Method Dispatch?

- Supports **runtime polymorphism** (real flexibility)
- Enables **generic programming**
- Allows code to call overridden methods **without knowing the exact object type**

What Is a Package in Java?
A **package** in Java is a **namespace** that groups related **classes, interfaces, and sub-packages** together.
Think of it like a **folder in a file system** — it helps organize your classes and avoid name conflicts.

Types of Packages
There are **two main types**:

Type	Description	Example
Built-in packages	Provided by Java	java.lang, java.util, java.io, java.net
User-defined packages	Created by the programmer	package myproject;

What Is an Exception in Java?

An **exception** is an **unwanted or unexpected event** that occurs during the execution of a program, disrupting its normal flow.

Example:

- Dividing by zero
- Accessing an invalid array index
- Reading a missing file

```
Object
             |
         Throwable
        /       \
     Exception      Error
      /    \
 Checked   Unchecked (RuntimeException)
```

Type	Description	Example
Checked Exceptions	Checked at compile-time	IOException, SQLException, FileNotFoundException
Unchecked Exceptions	Occur at runtime	ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException
Errors	Serious issues, not recoverable	OutOfMemoryError, StackOverflowError

```
public class WithoutException {
    public static void main(String[] args) {
        int a = 10, b = 0;
        int result = a / b; // ❌ Causes ArithmeticException
        System.out.println("Result: " + result);
    }
}
```

```
public class WithException {
    public static void main(String[] args) {
        int a = 10, b = 0;
        try {
            int result = a / b;  // risky code
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        }
        System.out.println("Program continues...");
    }
}
```

```
try {
    int a = 10 / 0;
    int arr[] = new int[2];
    arr[3] = 5;
} catch (ArithmeticException e) {
    System.out.println("Arithmetic error: " + e);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array error: " + e);
} catch (Exception e) {
    System.out.println("General error: " + e);
}
```

Keyword	Purpose
try	Defines block of code where exception may occur
catch	Used to handle the exception
finally	Executes whether exception occurs or not
throw	Used to manually throw an exception
throws	Declares exceptions that a method can throw

```
public class ThrowExample {
    static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");
        } else {
            System.out.println("Eligible to vote");
        }
    }

    public static void main(String[] args) {
        checkAge(15);
        System.out.println("End of program");
    }
}
```

Real Time Example – Exception Handling

```
import java.util.Scanner;

public class ATM {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter withdrawal amount: ");
        int amount = sc.nextInt();

        try {
            withdraw(amount);
        } catch (Exception e) {
            System.out.println("Transaction failed: " +
e.getMessage());
        } finally {
            System.out.println("Thank you for banking with
us!");
        }
    }

static void withdraw(int amount) throws Exception {
        int balance = 5000;
        if (amount > balance)
            throw new Exception("Insufficient balance!");
        else
            System.out.println("Withdrawal successful.
Remaining balance: " + (balance - amount));
    }
}
```

Thank You