What is Java Features

What is Java? Features of Java

✅ What is Java?

Java is a high-level, object-oriented, class-based programming language developed by **Sun Microsystems** (now owned by Oracle Corporation) and released in **1995**. It is designed to have **as few implementation dependencies as possible**, which makes it platform-independent.

Java is widely used for:

- Web applications
- Desktop GUI applications
- Mobile applications (especially Android)
- Enterprise applications
- Embedded systems

✅ Features of Java

Java has several key features that make it powerful and popular:

Feature	Description
1. Simple	Java is easy to learn, especially for those with C/C++ background. It removes complex features like pointers, operator overloading, etc.
2. Object-Oriented	Everything in Java is treated as an object, which makes it easier to manage and modify code.
3. Platform Independent	Java code is compiled into **bytecode** which can run on any device with a **Java Virtual Machine (JVM)**.
4. Secure	Java has built-in security features like bytecode verification, sandboxing, and runtime security APIs.
5. Robust	Java emphasizes strong memory management, exception handling, and type checking, reducing the chances of crashes.
6. Multithreaded	Java supports multithreading, allowing multiple tasks to run concurrently for better performance.
7. Architecture-Neutral	Java bytecode is not tied to any processor architecture, making it highly portable.
8. High Performance	Java performance is enhanced with **Just-In-Time (JIT)** compiler, though it may not be as fast as native C/C++.
9. Distributed	Java has built-in networking libraries for developing distributed applications (like RMI, sockets).
10. Dynamic	Java can dynamically load classes at runtime, making it extensible and flexible.

Would you like a quick comparison of Java vs. other languages (like Python, C++) or an explanation of the JVM, JRE, and JDK next?

JVM, JRE, JDK

Sure! Here's a clear and concise explanation of **JVM, JRE, and JDK**, which are core components of the Java ecosystem:

✅ 1. JVM (Java Virtual Machine)

Aspect	Details
Definition	JVM is the engine that runs Java bytecode.
Purpose	It converts **compiled bytecode** into **machine code** specific to the platform.
Platform Independence	JVM allows Java to be “**Write Once, Run Anywhere**.”
Tasks	Loads, verifies, executes code, and provides memory management and garbage collection.
Note	Each OS (Windows, Linux, Mac) has its own version of JVM.

✅ 2. JRE (Java Runtime Environment)

Aspect	Details
Definition	JRE is a software package that contains everything needed to **run** Java applications.
Includes	JVM + Java class libraries + other runtime components.
Purpose	To provide an environment to **run** Java programs (but not to develop them).
Does NOT Include	Development tools like **`javac`** (Java compiler).

✅ 3. JDK (Java Development Kit)

Aspect	Details
Definition	JDK is a complete software development kit used to **develop, compile, debug, and run** Java applications.
Includes	JRE + development tools (compiler **`javac`**, debugger **`jdb`**, archiver **`jar`**, etc.)
Purpose	To develop Java applications and applets.
Required for Developers?	✅ Yes, mandatory for Java development.

✅ Summary Table

Component	Contains	Used For
JVM	Part of JRE	Running Java bytecode

Component	Contains	Used For
JRE	JVM + Libraries	Running Java programs
JDK	JRE + Development tools	Developing Java programs

Would you like a diagram or visual representation of JDK → JRE → JVM hierarchy or a sample example of compiling and running a Java program?

Data Types and Variables

Here's a clear and structured explanation of **Data Types** and **Variables** in Java:

✅ Variables in Java

A **variable** is a container that holds data which can be changed during the execution of a program.

🔹 Syntax:

```java
dataType variableName = value;
```

🔹 Example:

```java
int age = 25;
String name = "Koti";
```

✅ Data Types in Java

Java is a **statically typed language**, which means all variables must be declared with a data type.

Java data types are classified into two main categories:

1️⃣ Primitive Data Types (8 types)

Type	Size	Description	Example
`byte`	1 byte	Whole numbers from -128 to 127	`byte b = 100;`
`short`	2 bytes	Whole numbers from -32K to 32K	`short s = 5000;`
`int`	4 bytes	Whole numbers (default integer)	`int i = 100000;`
`long`	8 bytes	Large whole numbers	`long l = 99999L;`
`float`	4 bytes	Decimal numbers (single-precision)	`float f = 10.5f;`

Type	Size	Description	Example
`double`	8 bytes	Decimal numbers (double-precision)	`double d = 99.99;`
`char`	2 bytes	Single character (Unicode)	`char c = 'A';`
`boolean`	1 bit	True/False values	`boolean b = true;`

2️⃣ Non-Primitive (Reference/Object) Data Types

These refer to objects, and they include:

Type	Description	Example
`String`	Sequence of characters	`String name = "Java";`
`Array`	Collection of elements	`int[] numbers = {1, 2};`
`Class`	User-defined data type	`class MyClass {...}`
`Interface`	Abstract data type (like a contract)	`interface Drawable {...}`

✅ Variable Types in Java

Type	Scope	Description
Local	Method/block	Declared inside methods.
Instance	Class level (non-static)	Belongs to objects.
Static/Class	Class level (static)	Shared across all instances.

Would you like a Java program to demonstrate different data types and variables?

yes

Here's a simple Java program that demonstrates how to declare and use **different data types** and **variables**:

✅ Java Program: Demonstrating Data Types and Variables

```java
public class DataTypesExample {

    // Instance variable
    int instanceVar = 50;

    // Static variable
    static String courseName = "Java Programming";

    public static void main(String[] args) {

        // Local variables
        int age = 25;
        float height = 5.9f;
        double salary = 75000.50;
        char grade = 'A';
        boolean isPass = true;
        long phoneNumber = 9876543210L;
        byte b = 120;
        short s = 30000;

        // Non-primitive type
        String name = "Koti";

        // Print all variables
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height);
        System.out.println("Salary: ₹" + salary);
        System.out.println("Grade: " + grade);
        System.out.println("Passed: " + isPass);
        System.out.println("Phone Number: " + phoneNumber);
        System.out.println("Byte value: " + b);
        System.out.println("Short value: " + s);

        // Accessing static variable
        System.out.println("Course: " + courseName);

        // Accessing instance variable through object
        DataTypesExample obj = new DataTypesExample();
        System.out.println("Instance variable: " + obj.instanceVar);
    }
}
```

✅ Output:

```yaml
Name: Koti
Age: 25
Height: 5.9
Salary: ₹75000.5
Grade: A
Passed: true
Phone Number: 9876543210
Byte value: 120
Short value: 30000
Course: Java Programming
Instance variable: 50
```

Would you like an explanation of each part of the code or to try it in an online Java compiler?

Type Casting

✅ What is Type Casting in Java?

Type Casting is the process of converting a variable from one data type to another.
In Java, type casting is classified into two types:

1️⃣ Widening Casting (Implicit)

Also called **upcasting**, this happens **automatically** when converting a smaller type to a larger type.

🔹 Example:

```
java

int a = 10;
double b = a;  // automatic conversion: int → double
System.out.println(b);  // Output: 10.0
```

🔹 Allowed widening conversions:

From	To
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

2️⃣ Narrowing Casting (Explicit)

Also called **downcasting**, this must be done **manually** and may result in **data loss**.

🔹 Syntax:

```
java

dataType variable = (dataType) value;
```

🔹 Example:

```
java

double a = 10.75;
int b = (int) a;  // manual conversion: double → int
System.out.println(b);  // Output: 10
```

✅ Java Program: Type Casting Example

```
java

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

        // Widening (Implicit)
        int x = 100;
        long y = x;
        float z = y;
        System.out.println("Widening Casting:");
        System.out.println("int to long: " + y);
        System.out.println("long to float: " + z);

        // Narrowing (Explicit)
        double d = 99.99;
        int i = (int) d;
        byte b = (byte) i;
        System.out.println("\nNarrowing Casting:");
        System.out.println("double to int: " + i);
        System.out.println("int to byte: " + b);
    }
}
```

✅ Output:

```vbnet
Widening Casting:
int to long: 100
long to float: 100.0

Narrowing Casting:
double to int: 99
int to byte: 99
```

Would you like to see how type casting behaves with `char`, `boolean`, or in real-world use cases (like user input, file handling, etc.)?

yes

Great! Let's explore **type casting** with:

1. ✅ `char` to numeric and vice versa
2. ✅ Real-world use case: **user input**
3. ⚠️ Why **`boolean` cannot** be cast

✅ 1. Type Casting with `char` and numbers

```java
public class CharCasting {
    public static void main(String[] args) {
        char ch = 'A';

        // char to int (implicit)
        int ascii = ch;
        System.out.println("ASCII of 'A': " + ascii);  // 65

        // int to char (explicit)
        int num = 66;
        char letter = (char) num;
        System.out.println("Character for 66: " + letter);  // B
    }
}
```

🔸 **Explanation**: `char` stores a Unicode character, which can be converted to/from `int` as it's backed by numbers.

❌ `boolean` Cannot Be Cast

```java
boolean b = true;
// int i = (int) b;  // ❌ Compilation error
```

🔸 **Reason**: Java treats `boolean` as a completely separate type. It cannot be cast to/from numeric types directly.

✅ 2. Real-World Use Case: Type Casting User Input

When reading user input from the console, it often comes in as `String` and must be **parsed** (cast) to numeric types.

```java
import java.util.Scanner;

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

        System.out.print("Enter your age: ");
        String ageInput = scanner.nextLine();  // read as String

        int age = Integer.parseInt(ageInput);  // cast to int
        System.out.println("You will be " + (age + 1) + " next year.");

        scanner.close();
    }
}
```

🔸 **Explanation**:

- `nextLine()` reads input as `String`.
- `Integer.parseInt()` is used to convert that `String` to `int`.

✅ Summary

From	To	Type	Notes
`char`	`int`	Implicit	Gets Unicode/ASCII value
`int`	`char`	Explicit	Converts number to character
`String`	`int/double`	Explicit	Via parsing (`parseInt`, etc.)
`boolean`	❌	❌	Cannot be cast

Would you like a small **mini-project** combining user input, type casting, and output formatting (like a BMI calculator or age checker)?

Operators (Arithmetic, Relational, Logical, Bitwise, Assignment, Ternary)

Sure! Here's a complete and beginner-friendly guide to **Operators in Java**, including all important categories:

✅ What are Operators?

Operators in Java are **symbols** that perform operations on **variables and values** (operands).

✅ 1. Arithmetic Operators

Used to perform basic math operations.

Operator	Description	Example	Result
`+`	Addition	`10 + 5`	15
`-`	Subtraction	`10 - 5`	5
`*`	Multiplication	`10 * 5`	50
`/`	Division	`10 / 5`	2
`%`	Modulus (remainder)	`10 % 3`	1

✅ 2. Relational / Comparison Operators

Used to compare two values; result is **true or false**.

Operator	Description	Example	Result
`==`	Equal to	`5 == 5`	`true`
`!=`	Not equal to	`5 != 3`	`true`
`>`	Greater than	`5 > 3`	`true`
`<`	Less than	`5 < 3`	`false`
`>=`	Greater than or equal	`5 >= 5`	`true`
`<=`	Less than or equal	`3 <= 5`	`true`

✅ 3. Logical Operators

Used for **boolean logic** (AND, OR, NOT).

Operator	Description	Example	Result
`&&`	Logical AND	`true && false`	`false`

Operator	Description	Example	Result
`		`	Logical OR
`!`	Logical NOT (reverse)	`!true`	`false`

✅ 4. Bitwise Operators

Used to perform bit-level operations.

Operator	Description	Example	Result (binary)
`&`	Bitwise AND	`5 & 3`	`0001` (1)
`	`	Bitwise OR	`5
`^`	Bitwise XOR	`5 ^ 3`	`0110` (6)
`~`	Bitwise NOT	`~5`	`-6` (in 2's comp)
`<<`	Left shift	`5 << 1`	`1010` (10)
`>>`	Right shift	`5 >> 1`	`0010` (2)

✅ 5. Assignment Operators

Used to assign values to variables.

Operator	Description	Example	Meaning
`=`	Assign	`x = 5`	x gets 5
`+=`	Add and assign	`x += 3`	x = x + 3
`-=`	Subtract and assign	`x -= 2`	x = x - 2
`*=`	Multiply and assign	`x *= 2`	x = x * 2
`/=`	Divide and assign	`x /= 2`	x = x / 2
`%=`	Modulus and assign	`x %= 2`	x = x % 2

✅ 6. Ternary Operator (?:)

A short-hand **if-else** statement.

🔹 Syntax:

```
java

variable = (condition) ? value_if_true : value_if_false;
```

🔹 Example:

```
java

int a = 10, b = 20;
int max = (a > b) ? a : b;
```

```
System.out.println("Maximum: " + max);
```

✅ Example Program to Demonstrate All Operators

```java
public class OperatorsDemo {
    public static void main(String[] args) {
        int a = 10, b = 3;

        // Arithmetic
        System.out.println("Addition: " + (a + b));
        System.out.println("Modulus: " + (a % b));

        // Relational
        System.out.println("Is a > b? " + (a > b));

        // Logical
        boolean x = true, y = false;
        System.out.println("x && y: " + (x && y));

        // Bitwise
        System.out.println("a & b: " + (a & b));

        // Assignment
        a += 5;
        System.out.println("a after += 5: " + a);

        // Ternary
        int max = (a > b) ? a : b;
        System.out.println("Max: " + max);
    }
}
```

Would you like to practice with operator-based interview questions or mini problems like a calculator app using switch + operators?

Input from user (Scanner class)

Here's a complete guide to **taking input from the user using the `Scanner` class** in Java:

✅ What is the `Scanner` Class?

The **`Scanner`** class (in **`java.util`** package) is used to **read input from the user** — from the **keyboard**, files, or other sources.

✅ How to Use the Scanner Class

🔹 Step 1: Import the Scanner class

```java
import java.util.Scanner;
```

🔹 Step 2: Create Scanner object

```java
Scanner scanner = new Scanner(System.in);
```

🔹 Step 3: Use methods like `nextInt()`, `nextLine()`, `nextDouble()`, etc.

✅ Example: Taking Different Types of Input

```java
import java.util.Scanner;

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

        // Taking input from user
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();  // reads a whole line

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();  // reads an integer

        System.out.print("Enter your height (in cm): ");
        double height = scanner.nextDouble();  // reads a decimal number

        System.out.print("Are you a student (true/false)? ");
        boolean isStudent = scanner.nextBoolean();  // reads boolean

        // Output
        System.out.println("\n--- User Info ---");
        System.out.println("Name     : " + name);
        System.out.println("Age      : " + age);
        System.out.println("Height   : " + height + " cm");
        System.out.println("Student? : " + isStudent);

        // Close scanner to prevent memory leaks
        scanner.close();
    }
}
```

✅ Common `Scanner` Methods

Method	Input Type
`nextLine()`	Whole line (String)
`next()`	Single word (String)
`nextInt()`	Integer
`nextDouble()`	Decimal number

Method	Input Type
`nextBoolean()`	Boolean (**`true`**/**`false`**)
`nextFloat()`	Float
`nextLong()`	Long integer
`nextShort()`	Short integer
`nextByte()`	Byte

⚠️ Tip:

After using **`nextInt()`** or **`nextDouble()`**, use **`scanner.nextLine()`** to **clear the newline character** if you're reading **`nextLine()`** afterward.

Would you like a mini project like a **simple login system** or **BMI calculator** using Scanner input?