

# Java main method explanation

public static void main  
why we use in java

In Java, we use

```
java

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

because this is the fixed entry point from where the JVM starts executing a Java program.

Let's break it word by word, very clearly 📌

## 1 public – Why public?

```
java

public
```

- The JVM is outside your class
- To start the program, JVM must be able to access the `main` method
- If it's not `public`, JVM cannot call it

✗ If `main` is not public → program will compile, but won't run

📌 So:

`public` ensures JVM can access the method

## 2 static – Why static?

```
java

static
```

- JVM does not create an object of your class to start execution

- `static` methods can be called without creating an object

If `main` were not static, JVM would need to do this:

```
java

ClassName obj = new ClassName();
obj.main();
```

But JVM never does that

📌 So:

`static` allows JVM to call `main` directly using the class name

## 3 void – Why void?

```
java

void
```

- `main` is just a starting point
- JVM does not expect any return value
- Returning anything from `main` is meaningless

📌 So:

`void` means `main` returns nothing to JVM

## 4 main – Why this name?

```
java

main
```

- JVM looks specifically for a method named `main`
- This name is fixed by Java language design

✗ If you rename it ( `start`, `run`, etc.) → JVM won't find it

📌 So:

`main` is the standard method name JVM searches for

## 5 String[] args – Why this parameter?

```
java
String[] args
```

- Used to pass **command-line arguments**
- JVM supplies arguments as **strings**
- They are stored in an **array**

Example:

```
bash
java MyClass hello 10
```

```
java
args[0] = "hello"
args[1] = "10"
```

👉 So:

It allows external input while starting the program

## ✓ Complete Signature (Standard Form)

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

This exact signature tells JVM:

- ✓ It can **access** the method
- ✓ It can **call it without object creation**
- ✓ It doesn't need a **return value**
- ✓ This is the **program starting point**

## 📄 JVM Execution Flow (Very Important)

1. You run the program
2. JVM loads the class
3. JVM searches for

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

4. JVM starts execution **from inside this method**

## ! What happens if something is wrong?

Mistake	Result
<code>main</code> not public	Runtime error
<code>main</code> not static	Runtime error
Wrong method name	Runtime error
Wrong parameters	Runtime error

## 💬 One-Line Summary

`public static void main` is used because JVM needs a publicly accessible, static method with no return value to start executing a Java program.

If you want, next I can explain:

- How JVM finds `main` internally
- Multiple `main` methods
- Overloading `main`
- What happens before `main` (class loading, static blocks)

Just tell me 👉

how many types we can take input in java and give me detailed explanation in this so that i donot have any doubt in this

In Java, there are several ways to take input, and each exists for a specific use-case. I'll explain ALL input types, from basic to professional level, with when to use what, how it works internally, pros/cons, and examples — so you have zero doubts.

## ◆ How many ways can we take input in Java?

👉 Practically, Java provides 7 major input mechanisms

We can group them into 3 levels:

Level	Category
● Beginner	Console / Keyboard input
● Intermediate	File & Stream input
● Advanced / Enterprise	Network, UI, Web input

## ● 1 Command Line Arguments ( `main(String[] args)` )

### ✓ What it is

Input passed while starting the program.

```
bash
```

```
java MyClass 10 Piyush
```

```
java
```

```
public static void main(String[] args) {
    System.out.println(args[0]); // 10
}
```

```
System.out.println(args[1]); // Piyush
}
```

### 🔧 How it works internally

- JVM reads command line input
- Stores it in a `String[]`
- Passes it to `main()`

### ✦ Characteristics

- ✓ Input is always `String`
- ✗ Not interactive
- ✗ Limited flexibility

### 💡 Use case

- Configuration values
- Startup parameters

## ● 2 `Scanner` class (Most commonly used)

```
java
```

```
Scanner sc = new Scanner(System.in);
int x = sc.nextInt();
String name = sc.next();
```

### 🔧 Internal Working

- Uses Regular Expressions
- Reads input from `System.in`
- Converts text → primitive types

### ✦ Supported Inputs

Method	Type
<code>nextInt()</code>	int
<code>nextDouble()</code>	double

Method	Type
<code>next()</code>	word
<code>nextLine()</code>	full line

### ⚠ Common Pitfall

```
java
nextInt() → leaves newline in buffer
nextLine() → reads empty line
```

### 💡 Use case

- Beginner programs
- Competitive coding practice

## 🟢 3 `BufferedReader` (Fast & professional)

```
java
BufferedReader br =
    new BufferedReader(new InputStreamReader(System.in));

String name = br.readLine();
int x = Integer.parseInt(br.readLine());
```

### 🔧 Internal Working

- Reads input **line by line**
- Uses **buffering** → faster than Scanner
- You manually convert data types

### 🌟 Pros & Cons

- ✓ Very fast
- ✓ Less memory overhead
- ✗ Manual parsing
- ✗ Slightly more code

### 💡 Use case

- High-performance input
- Competitive programming
- Backend services

## 🟡 4 `Console` class ( `System.console()` )

```
java
Console c = System.console();
String user = c.readLine();
char[] pass = c.readPassword();
```

### 🔧 Special Feature

- Password input **hidden**
- No echo on screen

### ✗ Limitations

- Doesn't work in IDEs
- Only works in real terminal

### 💡 Use case

- Secure input (passwords)

## 🟡 5 `File` Input ( `FileReader` , `BufferedReader` )

```
java
BufferedReader br =
    new BufferedReader(new FileReader("data.txt"));

String line = br.readLine();
```

### 🔧 Internal Working

- Reads data from files using OS I/O
- Uses streams & buffers

💬 Use case

- Reading logs
- Config files
- Batch processing

🟡 6 Byte Stream Input ( `InputStream` )

```
java
InputStream in = System.in;
int data = in.read();
```

🔧 How it works

- Reads raw bytes
- ASCII based input

📌 Used internally by

- Scanner
- `BufferedReader`

💬 Use case

- Low-level I/O
- Binary data

🟡 7 GUI / Web Input (Advanced)

GUI (Swing / JavaFX)

```
java
String input = textField.getText();
```

Web (Spring Boot)

```
java
```

```
@RequestParam String name
@RequestBody User user
```

💬 Use case

- Enterprise applications
- Web services
- APIs

📊 Complete Comparison Table

Input Type	Speed	Easy	Interactive	Professional
Command Line Args	★ ★ ★	★ ★ ★	✗	★
Scanner	★ ★	★ ★ ★ ★	✓	★ ★
BufferedReader	★ ★ ★ ★	★ ★	✓	★ ★ ★ ★
Console	★ ★	★ ★	✓	★ ★ ★
File Input	★ ★ ★ ★	★ ★	✗	★ ★ ★ ★
InputStream	★ ★ ★ ★	★	✗	★ ★ ★
GUI/Web	★ ★ ★ ★	★ ★ ★	✓	★ ★ ★ ★ ★

💬 Which one should YOU use?

Since you're aiming for Backend / Spring Boot:

- ✓ Learn `BufferedReader` deeply
- ✓ Understand `InputStream`
- ✓ Use Web input annotations ( `@RequestParam` , `@RequestBody` )

Scanner is good for learning, not production.

## 🔥 Interview One-Line Answer

Java supports multiple input mechanisms such as command-line arguments, Scanner, BufferedReader, Console, file input, byte streams, and GUI/Web inputs, each designed for different performance, security, and application needs.

If you want next:

- Difference between `Scanner` vs `BufferedReader` internally
- Why Scanner is slow
- Input buffering internals
- How Spring Boot handles input internally
- JVM input pipeline diagram

Just tell me 🙌

what is `System.out.println` and `System.in`

In Java, `System.out.println` and `System.in` are the **most fundamental I/O mechanisms**, and understanding them deeply is **very important** (especially for backend & JVM-level clarity).

I'll explain this **from JVM → OS → Java API level**, step by step, so you'll have **zero doubt**.

## ◆ What is `System` in Java?

java

`System`

- `System` is a **final class**
- Present in `java.lang` package (auto-imported)
- Acts as a **bridge between Java program and the Operating System**

java

```
public final class System {
    public static InputStream in;
```

```
public static PrintStream out;
public static PrintStream err;
}
```

👉 `System` gives access to **standard I/O streams**.

## ◆ What is `System.out.println`?

### 1 `System.out`

java

`System.out`

- `out` is a **static variable**
- Type: `PrintStream`
- Connected to **standard output stream**
- By default → **Console / Terminal**

java

```
PrintStream out = System.out;
```

### 2 `println()`

java

```
System.out.println("Hello");
```

- `println()` is a method of `PrintStream`
- Prints data + **adds a newline**
- Flushes output buffer automatically

### Overloaded versions

java

```
println(int)
println(double)
```

```
println(String)
println(Object)
```

### 🔧 Internal Working of `System.out.println`

```
arduino

Java Code
↓
PrintStream.println()
↓
OutputStream.write()
↓
OS Standard Output
↓
Console / Terminal
```

- ✓ Uses **buffered output**
- ✓ Thread-safe
- ✓ Platform-independent

### 💬 Why not `System.print` directly?

Method	Behavior
<code>print()</code>	Prints without newline
<code>println()</code>	Prints + newline
<code>printf()</code>	Formatted output

### ◆ What is `System.in`?

```
java

System.in
```

- `in` is a **static variable**
- Type: `InputStream`
- Connected to **standard input**
- Default source → **Keyboard**

```
java

InputStream in = System.in;
```

### 🔧 Internal Working of `System.in`

```
mathematica

Keyboard
↓
OS Input Buffer
↓
System.in (InputStream)
↓
Java Program
```

- Reads **raw bytes**
- ASCII / Unicode based
- Low-level input

### ⚠️ Why `System.in` is rarely used directly?

```
java

int ch = System.in.read();
```

- ✗ Reads only **one byte**
- ✗ Returns ASCII value
- ✗ Hard to handle strings and numbers

👉 That's why we wrap it with:

```
java
```

```
Scanner sc = new Scanner(System.in);
BufferedReader br = new BufferedReader(
    new InputStreamReader(System.in));
```

◆ Relationship Between Them

Component	Direction	Type
System.in	Input	InputStream
System.out	Output	PrintStream
System.err	Error Output	PrintStream

🔥 JVM-Level View (Very Important)

When JVM starts:

- It connects System.in → OS stdin
- It connects System.out → OS stdout
- It connects System.err → OS stderr

These are pre-initialized static streams.

🔄 Can we change them?

Redirect output

```
java
System.setOut(new PrintStream("output.txt"));
```

Redirect input

```
java
System.setIn(new FileInputStream("input.txt"));
```

- ✓ Common in testing
- ✓ Used in logging

💬 One-Line Summary (Interview Perfect)

System.in is a standard input stream used to read data from the keyboard, while System.out.println prints data to the console using the standard output stream, both provided by the JVM as OS-level I/O bridges.

🔧 Why this matters for Backend Devs

- Logging systems replace System.out
- Input streams power HTTP requests
- Streams are core to Spring Boot & Microservices

If you want next:

- Why logging is preferred over System.out
- Difference between System.out and Logger
- How Spring Boot replaces console output
- Deep dive into Java Streams vs I/O Streams

Just tell me 🙌

why we use buffer reader instead of scanner class in java

give me the datilsed explanation of this os that i can able to understand it well and donot have any doubt in this

This is an excellent and very important question, Piyush 🙌

The difference between BufferedReader and Scanner is not just syntax — it is about performance, internal working, memory usage, and professional coding standards.

I'll explain this from inside → outside, so you'll have crystal-clear understanding with zero doubts.



## ◆ Why do we use `BufferedReader` instead of `Scanner` in Java?

### 👉 Short answer (interview-ready):

We prefer `BufferedReader` over `Scanner` when performance matters because `BufferedReader` reads input in large chunks (buffered I/O) and does not use regex parsing, making it significantly faster and more memory-efficient than `Scanner`.

Now let's break this down deeply.

### 1 How `Scanner` works internally (WHY it is slow)

```
java
Scanner sc = new Scanner(System.in);
int x = sc.nextInt();
```

#### 🔧 Internal Process

1. Reads input from `System.in` (byte stream)
2. Converts bytes → characters
3. Uses **Regular Expressions (Regex)** to match patterns
4. Validates input format
5. Converts string → int

```
pgsql
Input → Stream → Regex Matching → Parsing → Conversion
```

#### ✗ Problems with Scanner

- Regex processing is CPU expensive
- Creates many temporary objects
- High memory overhead
- Slower for large input

✦ Scanner is designed for **convenience**, not performance.

### 2 How `BufferedReader` works internally (WHY it is fast)

```
java
BufferedReader br =
    new BufferedReader(new InputStreamReader(System.in));

int x = Integer.parseInt(br.readLine());
```

#### 🔧 Internal Process

1. Reads **large blocks of data** into a buffer
2. No regex involved
3. Returns raw strings
4. You manually convert data

```
pgsql
Input → Buffer (8KB+) → Line → Conversion
```

#### ✅ Advantages

- ✓ Much faster
- ✓ Low memory overhead
- ✓ No regex cost
- ✓ Efficient for large input

### 3 Buffering – THE REAL DIFFERENCE 🔥

#### ● Scanner (No real buffering)

- Reads **small chunks**
- Frequent OS calls

#### ● `BufferedReader` (Buffered I/O)

- Reads **8KB+ at once**
- Fewer system calls
- Faster execution

OS calls are expensive — fewer calls = better performance

4 Performance Comparison (Real-world)

Operation	Scanner	BufferedReader
Large input	✗ Slow	✓ Very fast
Competitive programming	✗ Avoid	✓ Preferred
Backend services	✗ Avoid	✓ Preferred
Small programs	✓ OK	⚠ Overkill

5 Memory Usage Difference

Feature	Scanner	BufferedReader
Regex engine	✗ Heavy	✓ None
Object creation	✗ High	✓ Minimal
GC pressure	✗ High	✓ Low

6 Input Control & Validation

Scanner

```
java
sc.nextInt(); // auto validation
```

- ✓ Safer for beginners
- ✗ Less control

BufferedReader

```
java
```

```
Integer.parseInt(br.readLine());
```

- ✓ Full control
- ✓ Better error handling
- ✓ Cleaner business logic

7 Exception Handling

Scanner

- Throws unchecked exceptions
- Harder to predict failures

BufferedReader

- Uses checked IOException
- Encourages proper error handling
- ✓ Professional coding standard

8 Industry & Interview Perspective

? Interview Question:

Why is Scanner not used in production?

- ✓ Slower due to regex
- ✓ High memory usage
- ✓ Poor performance for large input

✓ What industry uses:

- BufferedReader
- InputStream
- NIO
- HTTP request streams

9 When SHOULD you use Scanner?

- ✓ Learning Java
- ✓ Small console programs
- ✓ Quick testing
- ✗ Not for performance-critical code

## 10 When SHOULD you use BufferedReader?

- ✓ Competitive coding
- ✓ Backend systems
- ✓ Reading files
- ✓ High-volume input
- ✓ Professional projects

## 🔥 One Diagram Summary (Mental Model)

mathematica

Scanner:

Input → Regex → Parse → Convert → Value

BufferedReader:

Input → Buffer → String → Manual Convert

## 💬 One-Line Ultimate Summary (Interview Perfect)

BufferedReader is preferred over Scanner because it uses buffered I/O without regex parsing, making it faster, memory-efficient, and more suitable for large-scale and production-level applications.

## 🎯 Since you're a Backend Developer Aspirant

You **must** be comfortable with:

- `BufferedReader`
- `InputStream`

- Stream pipelines
- Exception handling

Scanner is **only a learning tool**, not a professional one.

If you want next, I can explain:

- How buffering works at OS level
- Scanner vs BufferedReader with benchmark code
- Why Spring Boot never uses Scanner
- How HTTP input uses streams internally

Just tell me 🙌

explain deeper about scanner class and bufferreader class.

Perfect 🙌

Now let's go **DEEPER** than normal explanations — not just *what* `Scanner` and `BufferedReader` are, but **how they work internally**, how JVM + OS interact, memory behavior, buffering, regex, object creation, and why industry prefers one over the other.

By the end, you'll have zero confusion and professional-level clarity.

## ◆ Big Picture: Where Scanner & BufferedReader Fit

markdown

Keyboard

↓

OS Input Buffer

↓

System.in (InputStream - byte stream)

↓

| `Scanner`      `BufferedReader` |

↓  
Java Program

Both read from `System.in`, but **how** they process data is completely different.

## Scanner Class – DEEP INTERNAL VIEW

### 1 What is Scanner?

```
java
java.util.Scanner
```

- High-level input utility
- Token-based input reader
- Designed for **ease of use**, not performance

### 2 Scanner Architecture (Very Important)

```
pgsql
InputStream (System.in)
  ↓
ReadableByteChannel
  ↓
Scanner Buffer
  ↓
Regex Pattern Matching
  ↓
Token Extraction
  ↓
Type Conversion
```

#### 🔥 Core Engine = Regex

Scanner internally uses:

```
java
```

```
java.util.regex.Pattern
java.util.regex.Matcher
```

Every call like:

```
java
nextInt()
```

means:

1. Match regex for integer
2. Validate format
3. Convert String → int

👉 **Regex = expensive CPU operation**

### 3 Scanner Buffering (Truth)

- Scanner **does have a small buffer**
- But it is **token-oriented**, not block-oriented
- Leads to **frequent OS reads**

✗ Not optimized for large continuous input

### 4 Scanner Object Creation Cost

Scanner creates:

- Pattern objects
- Matcher objects
- Temporary String objects
- Wrapper conversions

🌟 Result:

- High GC pressure
- More memory usage
- Slower execution

## 5 Scanner Error Handling

```
java
```

```
InputMismatchException
NoSuchElementException
IllegalStateException
```

- ✗ Runtime exceptions
- ✗ Harder to control in production

## 6 Scanner Pros & Cons (Deep)

### ✓ Pros

- ✓ Very easy to use
- ✓ Auto parsing
- ✓ Beginner friendly

### ✗ Cons

- ✗ Slow
- ✗ Regex overhead
- ✗ Memory heavy
- ✗ Not scalable

## ● BufferedReader Class – DEEP INTERNAL VIEW

### 1 What is BufferedReader?

```
java
```

```
java.io.BufferedReader
```

- Low-level, high-performance reader
- Line-based input reader
- Designed for **speed & efficiency**

## 2 BufferedReader Architecture

```
SCSS
```

```
InputStream (System.in)
  ↓
InputStreamReader (byte → char)
  ↓
BufferedReader (8KB+ char buffer)
  ↓
readLine()
```

### 🔥 Core Engine = Buffering

- Reads large blocks (default 8KB)
- Minimizes OS calls

## 3 Buffering Mechanism (KEY CONCEPT)

### Without Buffer

```
pgsql
```

```
Read → OS call → Read → OS call → Read
```

### With Buffer

```
pgsql
```

```
Read 8KB once → Serve from memory
```

- ✓ Fewer system calls
- ✓ Faster execution

## 4 BufferedReader Memory Behavior

- Single reusable buffer
- Minimal object creation
- Low GC pressure

✔ Ideal for large inputs

5 Manual Parsing (Control Advantage)

```
java

int x = Integer.parseInt(br.readLine());
```

- ✔ Full control
- ✔ Clear business logic
- ✔ Custom validation

6 BufferedReader Error Handling

```
java

IOException (checked exception)
```

- ✔ Forces proper handling
- ✔ Production-safe
- ✔ Predictable failures

7 BufferedReader Pros & Cons (Deep)

✔ Pros

- ✔ Extremely fast
- ✔ Memory efficient
- ✔ Scales well
- ✔ Production ready

✖ Cons

- ✖ Slightly more code
- ✖ Manual parsing required
- ✖ Not beginner friendly

🔥 Scanner vs BufferedReader – Internals Comparison

Feature	Scanner	BufferedReader
Parsing	Regex-based	Manual
Buffering	Token-based	Block-based
OS calls	Many	Few
Speed	Slow	Fast
Memory	High	Low
Exceptions	Runtime	Checked
Scalability	Poor	Excellent

💡 Why Competitive Programmers NEVER use Scanner?

Because:

- Input size is huge
- Regex kills performance
- Time Limit Exceeded (TLE)

👉 They use:

```
java

BufferedReader + StringTokenizer
```

💡 Why Backend Systems NEVER use Scanner?

- HTTP input is stream-based
- Performance is critical
- Predictability matters
- Memory usage must be low

Spring Boot internally uses:

- `InputStream`
  - `BufferedReader`
  - `NIO`
- 

## Mental Model (Remember this forever)

ini

`Scanner` = Convenience

`BufferedReader` = Performance

---

## Ultimate Interview Answer (Perfect)

Scanner is a high-level, regex-based input reader optimized for ease of use, whereas `BufferedReader` is a low-level, buffered input reader optimized for speed, memory efficiency, and large-scale applications.

---

## Since you are preparing for Backend & Interviews

You MUST:

- ✓ Know Scanner syntax
  - ✓ Master `BufferedReader` internals
  - ✓ Understand stream & buffer concepts
- 

If you want next:

- Scanner vs `BufferedReader` benchmark demo
- How `StringTokenizer` fits in
- `NIO` vs `BufferedReader`
- How Spring Boot reads request body internally

Just tell me 👍