System.out.println Statement

✅ **Definition**

- System.out.println() is a built-in Java statement used to print output to the console.
- It helps developers see results, debug programs, or display information to users.
- System is a predefined class.
- out is a static member of the System class and is an instance of PrintStream.
- println() is a method of the PrintStream class that prints the output followed by a new line.

✅ **Structure Breakdown**

System.out.println("Hello, Java!");

- **System** → A final class that provides access to system resources.
- **out** → A static member representing the standard output stream.
- **println()** → Method that prints data and moves to the next line.

✅ **Example**

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Welcome to Core Java!");
        System.out.println(123);         // Prints integer
        System.out.println(3.14);        // Prints decimal value
        System.out.println(true);        // Prints boolean value
    }
}
```

Output:

Welcome to Core Java!

123

3.14

true

✅ **Key Points**

✓ System.out.print() → prints without new line.

✓ System.out.println() → prints with a new line.

✓ Used in debugging and displaying outputs in console applications.

Package in Java

✅ Definition

- A package in Java is a group of related classes and interfaces.
- It helps organize code, avoid name conflicts, and control access.
- Packages are like folders/directories in a file system.

✅ Built-in Packages

Examples:

- java.lang → core classes like String, Math.
- java.util → utility classes like Scanner, ArrayList.
- java.io → input/output related classes.

✅ User-defined Packages

- We can create our own packages to organize classes.

✅ Creating a Package

Step 1: Create a folder structure representing the package.

Example: com.example.utilities

Step 2: Write a class inside the package.

```
package com.example.utilities;


public class MathHelper {
    public static int square(int number) {
        return number * number;
    }
}
```

Step 3: Import and use the package in another class.

```
import com.example.utilities.MathHelper;
```

```
public class TestPackage {
    public static void main(String[] args) {
        int result = MathHelper.square(5);
        System.out.println("Square of 5 is: " + result);
    }
}
```

Output:

```
Square of 5 is: 25
```

✅ **Why Use Packages?**

✓ **Code Organization:** Keeps related classes together.
✓ **Avoid Naming Conflicts:** Same class names can exist in different packages.
✓ **Access Protection:** Control visibility with access specifiers (public, private, protected).
✓ **Reusability:** Easy to reuse classes across multiple projects.

Subpackage in Java

✅ **Definition**

- A subpackage is a package nested inside another package.
- It helps further organize classes in a hierarchical way.

Example:

com.example.utilities.math

com.example.utilities.string

✅ **Usage Example**

Class in subpackage:

```
package com.example.utilities.math;


public class AdvancedMath {
    public static int cube(int number) {
        return number * number * number;
    }
}
```

Importing subpackage in main program:

```
import com.example.utilities.math.AdvancedMath;


public class TestSubpackage {
    public static void main(String[] args) {
        int result = AdvancedMath.cube(3);
        System.out.println("Cube of 3 is: " + result);
    }
}
```

Output:

Cube of 3 is: 27

✅ Key Notes on Packages & Subpackages

✓ Package names are usually in lowercase.
✓ Subpackages are created by adding further directories.
✓ Import statements are used to access classes from other packages.
✓ Classes in subpackages are fully qualified by their package name.

Problem:

```
public class MindBendingExample2 {
    public static void main(String[] args) {
        System.out.print("Path: C:\\Users\\Admin\n");
        System.out.println("Quote: \"Believe in yourself\"");

        System.out.print("Tab\tseparated\tvalues\n");
        System.out.println("Line1\nLine2");

        System.out.print("This is ");
        System.out.print("a test.\n");
        System.out.println("And this is another line.");
    }
}
```

Path: C:\Users\Admin

Quote: "Believe in yourself"

Tab separated values

Line1

Line2

This is a test.

And this is another line.

16/09/2025

Static Import

📖 Definition

The **static import** feature in Java allows members (fields and methods) which are declared static in another class to be used in the current class without specifying the class name.

This improves code readability by avoiding repetitive class names.

☑ Why use Static Import?

- Avoid writing the class name repeatedly.
- Makes code cleaner and concise when using constants or static methods.
- Used in cases like Math, Collections, Assertions, etc.

☑ Syntax

import static package_name.ClassName.static_member;

or

import static package_name.ClassName.*;

The asterisk (*) imports all static members of the class.

☑ Example 1 – Using Math class without static import

```
import java.lang.Math;


public class StaticImportExample {
    public static void main(String[] args) {
        double result = Math.sqrt(25);
        System.out.println("Square root of 25 is: " + result);
    }
}
```

☑ Example 2 – Using Math class with static import

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


public class StaticImportExample {
    public static void main(String[] args) {
        double result1 = sqrt(25);
        double result2 = pow(2, 3);
        System.out.println("Square root of 25 is: " + result1);
        System.out.println("2 raised to the power 3 is: " + result2);
    }
}
```

✅ Important Notes

✓ Static import should be used sparingly to maintain code clarity.
✓ It is best when constants or utility methods are heavily used.
✓ Excessive static import can reduce readability because it becomes unclear where a method or field comes from.

2. Introduction to Modules

✅ **One-line definitions:**

- Classpath: It is a parameter that tells the Java compiler and runtime where to find user-defined classes and packages during execution.
- Modules: A module is a group of related packages with a defined boundary that explicitly declares its dependencies and accessible packages.

Syntax:

Feature	Syntax Example
Classpath	**java -cp path_to_classes ClassName**
Module	**module module_name { requires ...; exports ...; }**

✅ **How modules are better than classpath (one line):**

Modules are better because they provide strong encapsulation and explicit dependency management, reducing conflicts and improving maintainability compared to the implicit and loose structure of classpath.

Classpath Example

Files:

1. HelloWorld.java

HelloWorld.java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello from classpath example!");
    }
}
```

✅ **How to Compile and Run**

Compile:

```
javac HelloWorld.java
```

Run using classpath:

```
java -cp . HelloWorld
```

Output:

```
Hello from classpath example!
```

Module Example

Directory Structure:

```
module-example/
└── src/
    └── com.example.app/
        ├── module-info.java
        └── com/
            └── example/
                └── app/
                    └── App.java
```

---

module-info.java

```
module com.example.app {
   exports com.example.app;
}
```

The module-info.java file defines the structure and relationships of a **module** in Java. It contains:

1. **Module declaration** – The name of the module.
2. **Requires clause** – Declares dependencies on other modules.
3. **Exports clause** – Specifies which packages can be accessed by other modules.

App.java

```
package com.example.app;


public class App {
   public static void main(String[] args) {
      System.out.println("Hello from module example!");
```

```
    }
}
```

✅ How to Compile and Run

Compile:

```
javac -d out --module-source-path src $(find src -name "*.java")
```

Run:

```
java --module-path out -m com.example.app/com.example.app.App
```

Output:

```
Hello from module example!
```

✅ Conclusion

- The **classpath example** shows how Java finds classes using the classpath.
- The **module example** shows how Java enforces module boundaries and explicitly declares exports.

📖 Definition

A **module** in Java is a group of packages designed to offer better modularization of code.

It was introduced in **Java 9** as part of the **Java Platform Module System (JPMS)** to:

- Improve encapsulation
- Avoid dependency conflicts
- Enhance performance and security

✅ Why Modules are Important

✓ Manage dependencies explicitly
✓ Define clear boundaries between parts of an application
✓ Reduce runtime footprint
✓ Avoid “classpath hell”
✓ Support scalable development

✅ Key Terms

- **module-info.java** – The configuration file defining a module's dependencies and exported packages.
- **requires** – Specifies dependencies on other modules.
- **exports** – Declares which packages can be accessed by other modules.

✅ Creating a Module

Step 1 – Create a directory structure

```
module-example/
└── src/
    └── com.example.app/
        ├── module-info.java
        └── com/
            └── example/
                └── app/
                    └── App.java
```

Step 2 – Define module-info.java

```
module com.example.app {
    requires java.base;    // Optional, as java.base is always included
    exports com.example.app;
}
```

Step 3 – App.java

```
package com.example.app;
public class App {
    public static void main(String[] args) {
        System.out.println("Welcome to Java Module System!");
    }
}
```

✅ Module Declaration Example

```
module com.example.utility {
    exports com.example.utility;
}
```

This declares a module com.example.utility and exports its package so other modules can use it.

✅ Using Multiple Modules

If com.example.app requires com.example.utility:

```
module com.example.app {
    requires com.example.utility;
}
```

This means com.example.app depends on the utility module.

✅ Compile and Run a Module

Compile:

```
javac -d out --module-source-path src $(find src -name "*.java")
```

Run:

```
java --module-path out -m com.example.app/com.example.app.App
```

✅ Benefits of Using Modules

✓ Strong encapsulation
✓ Reduced maintenance overhead
✓ Better dependency management
✓ Easier packaging and deployment
✓ More secure applications

✅ Module System vs Classpath

Feature	Classpath	Module System
Dependency	Implicit	Explicit
Encapsulation	Weak	Strong
Performance	Limited	Optimized
Security	Less control	Improved
Versioning	Complex	Supported

✅ Key Takeaways

- **Modules** help in organizing and structuring large applications by grouping packages, managing dependencies, and improving modularization.
- Modules offer better encapsulation, dependency management, and scalability compared to traditional classpath-based applications.

Introduction to Exceptions in Java

📖 What is an Exception?

An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

It is an unwanted or unexpected situation that arises when the program runs, such as:

- Division by zero
- File not found
- Array index out of bounds
- Null pointer access

✅ Why are Exceptions Important?

✓ They help in identifying and handling errors gracefully.
✓ Without handling exceptions, the program may terminate abruptly.
✓ With exception handling, programs can recover or exit cleanly without crashing.

✅ Exception vs Error

Exception	Error
Can be handled using try-catch blocks	Usually cannot be handled (like OutOfMemoryError)
Caused by external or logical issues	Caused by environment or JVM failures
Examples: IOException, ArithmeticException	Examples: StackOverflowError, VirtualMachineError

---

✅ Types of Exceptions

1. **Checked Exceptions**
    - Known at compile time
    - Must be handled using try-catch or declared using throws
    - Example: IOException, SQLException

2. **Unchecked Exceptions (Runtime Exceptions)**
    - Known at runtime
    - Not required to be declared or handled
    - Example: NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException
3. **Errors**
    - Severe problems mostly beyond control
    - Example: OutOfMemoryError

✅ **Exception Hierarchy**

java.lang.Throwable

├── java.lang.Error

└── java.lang.Exception

  ├── java.io.IOException (Checked)

  └── java.lang.RuntimeException (Unchecked)

    ├── NullPointerException

    ├── ArithmeticException

    └── ArrayIndexOutOfBoundsException

✅ **How Exceptions Work**

- When an exception occurs, Java creates an exception object.
- The runtime system looks for a matching exception handler (try-catch).
- If none is found, the program terminates.

✅ **Basic Exception Handling Syntax**

```
try {
   // code that may throw exception
} catch (ExceptionType e) {
```

```
    // code to handle exception
} finally {
    // optional block to execute code always
}
```

✅ **Example – ArithmeticException**

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

Output:

```
Cannot divide by zero!
Execution completed.
```

✅ **Key Points**

✓ Exception handling prevents abrupt program termination.
✓ try block surrounds code that may fail.
✓ catch handles specific exceptions.
✓ finally is optional and executes always.
✓ Checked exceptions must be handled or declared, whereas unchecked exceptions do not require explicit handling.

17/09/2025

Exception Handling in Java

✅ What is an Exception?

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- Example: dividing by zero, accessing an invalid index, file not found, etc.
- Java provides a way to handle exceptions to avoid program crashes.

✅ Types of Exceptions

1. **Checked Exception**
    - Handled at compile time.
    - Example: IOException, SQLException.
2. **Unchecked Exception**
    - Handled at runtime.
    - Example: ArithmeticException, NullPointerException.

✅ Keywords Used in Exception Handling

➤ try

- The block where you write the code that might throw an exception.
- Syntax:
- try {
- // risky code
- }

➤ catch

- The block where you handle the exception.
- Syntax:
- catch (ExceptionType e) {
- // code to handle exception
- }

➤ throw

- Used to explicitly throw an exception from a method or block.
- Syntax:
- throw new ExceptionType("message");

➤ throws

- Used in a method signature to declare that the method might throw exceptions.
- Syntax:
- public void methodName() throws ExceptionType {
- // code
- }

➤ finally

- A block that always executes whether an exception is thrown or not.
- Used to release resources like file handles, database connections.
- Syntax:
- finally {
- // cleanup code
- }

☑ Example 1: Using try and catch

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int a = 10, b = 0;
            int result = a / b; // throws ArithmeticException
            System.out.println(result);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero: " + e.getMessage());
        }
    }
}
```

✓ Output:
Cannot divide by zero: / by zero

☑ Example 2: Using throw

```
public class ThrowExample {
    static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");
        } else {
            System.out.println("You are eligible to vote.");
        }
    }
    public static void main(String[] args) {
        checkAge(16);
    }
}
```

✓ Output:
Exception in thread "main" java.lang.ArithmeticException: Not eligible to vote

☑ Example 3: Using throws

```
import java.io.*;


public class ThrowsExample {
    public static void readFile() throws IOException {
        FileReader file = new FileReader("nonexistent.txt");
        file.read();
    }


    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("File not found or error reading file: " + e.getMessage());
        }
    }
}
```

✓ Output:
File not found or error reading file: nonexistent.txt (No such file or directory)

☑ Example 4: Using finally

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            int data = 25 / 5;
            System.out.println("Result: " + data);
        } catch (ArithmeticException e) {
            System.out.println("Exception occurred");
        } finally {
            System.out.println("Finally block always executed");
        }
    }
}
```

✓ Output:

```
Inside try block
Result: 5
Finally block always executed
```

Built-in Exceptions in Java

Exception Type	Description
ArithmeticException	Division by zero
NullPointerException	Accessing methods or variables on a null object
ArrayIndexOutOfBoundsException	Invalid array index
ClassNotFoundException	Class not found at runtime
IOException	Input/output failures
FileNotFoundException	File not found
NumberFormatException	Invalid number format
IllegalArgumentException	Illegal argument passed to a method

Write a Java program to demonstrate how to avoid and handle exceptions during division by zero.

```
public class SimpleExceptionDemo {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;

        // Avoiding exception
        if (b != 0) {
            System.out.println("Result: " + (a / b));
        } else {
            System.out.println("Avoided exception: Cannot divide by zero");
        }

        // Detecting (catching) exception
        try {
            int result = a / b;  // risky line
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Exception Detected: " + e.getMessage());
        }
    }
}
```

Output:

Avoided exception: Cannot divide by zero
Exception Detected: / by zero
This short program:

- First **avoids** the error using an if check.
- Then **detects** the same error using try-catch.

Custom Exception

You can create your own exception by extending the Exception class.

Example:

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


public class CustomExceptionExample {
    static void validate(int age) throws MyException {
        if (age < 18) {
            throw new MyException("Age must be 18 or older.");
        } else {
            System.out.println("Age is valid.");
        }
    }

    public static void main(String[] args) {
        try {
            validate(15);
        } catch (MyException e) {
            System.out.println("Custom exception caught: " + e.getMessage());
        }
    }
}
```

✓ Output:
Custom exception caught: Age must be 18 or older.

✅ Key Points to Remember

✓ Exception handling avoids program crashes.
✓ try is where risky code is placed.
✓ catch handles the exception.
✓ throw explicitly throws an exception.
✓ throws declares possible exceptions in method signatures.
✓ finally always executes and is used for cleanup tasks.
✓ Java has many built-in exceptions to handle common errors.
✓ You can create custom exceptions to meet specific application requirements.

Throwable Class in Java

1. Introduction

- In Java, all **errors and exceptions** are objects.
- The root class for all such objects is **java.lang.Throwable**.
- It is the superclass of both:
    - **Exception** (for exceptional conditions a program might want to handle).
    - **Error** (for serious problems a program should not try to handle).

2. Throwable Class Hierarchy

Object

└── Throwable

├── Error

└── Exception

Error

- Represents serious issues related to the JVM (Java Virtual Machine).
- Examples: OutOfMemoryError, StackOverflowError.
- Programs should **not try to recover** from Errors.

Exception

- Represents abnormal conditions that can be **caught and handled**.
- Two types:
    - **Checked Exceptions**: Must be handled (e.g., IOException, SQLException).
    - **Unchecked Exceptions (Runtime Exceptions)**: Can occur during execution (e.g., NullPointerException, ArithmeticException).

3. Important Methods of Throwable Class

Method	Description
getMessage()	Returns a detailed message about the exception/error.
printStackTrace()	Prints the stack trace (useful for debugging).
getCause()	Returns the cause of the exception.
toString()	Returns a short description of the exception/error.
fillInStackTrace()	Records the stack trace of the Throwable object.

4. Example Program

```
public class ThrowableDemo {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;   // will cause ArithmeticException
        } catch (Throwable t) {   // catching Throwable (superclass)
            System.out.println("Caught Throwable!");
            System.out.println("Message: " + t.getMessage());
            System.out.println("Description: " + t.toString());
            System.out.println("Stack Trace:");
            t.printStackTrace();
        }
    }
}
```

Output:

```
Caught Throwable!
Message: / by zero
Description: java.lang.ArithmeticException: / by zero
```

Stack Trace:

java.lang.ArithmeticException: / by zero

at ThrowableDemo.main(ThrowableDemo.java:5)

5. Key Points

- Throwable is the root class for all **exceptions and errors**.
- Normally, we **catch Exception**, not Throwable (catching Throwable may also catch serious Errors).
- Useful methods: getMessage(), printStackTrace(), toString().
- Provides a unified way of handling abnormal situations in Java.