**JAVA LEARNING**

** Core Java (Basics first)**

* **Object-Oriented Programming (OOP)**
* **Data Structures (Arrays, Lists, HashMaps)**
* **Exception Handling, Multithreading**
* **File Handling, Streams**
* **JDBC (Database Connectivity)**

** Spring Boot (Backend Framework)**

* **Setting up a Spring Boot project**
* **REST APIs (Controllers, Services, Repositories)**
* **Database with JPA & Hibernate**
* **Security (JWT, OAuth, Role-based authentication)**

** Frontend (If needed)**

* **HTML, CSS, JavaScript**
* **React.js or Angular (pick one)**

** Database**

* **SQL (PostgreSQL, MySQL, or MongoDB for NoSQL)**

** DevOps & Deployment *(Later, not now!)***

* **Docker, Kubernetes**
* **CI/CD with Jenkins/GitHub Actions**

**1. Core Java (Back-end)**

* **Java Basics: Data types, operators, control flow (if-else, loops)**
* **Object-Oriented Programming (OOP): Classes, Objects, Inheritance, Polymorphism, Encapsulation, Abstraction**
* **Java Data structures**
* **Exception Handling: Try-catch blocks, custom exceptions**
* **Collections Framework: List, Set, Map, Queue, and their implementations**
* **Generics: Types and wildcards**
* **Streams and Lambdas: Stream API, functional programming with lambdas**
* **Concurrency: Threads, Runnable interface, synchronization, Executor service**
* **File I/O: Reading and writing files, serialization**
* **JVM Internals: Memory management, garbage collection, JVM tuning**
* **Design Patterns: Singleton, Factory, Observer, MVC, DAO, etc.**

**2. Java Web Development (Back-end)**

* **Servlets: Creating web applications using Servlets**
* **JSP (Java Server Pages): For rendering dynamic content in web applications**
* **Spring Framework: The most important framework for full-stack developers** 
  + **Spring Boot: Creating REST APIs, microservices, and application configurations**
  + **Spring MVC: Model-View-Controller architecture for web applications**
  + **Spring Security: Authentication, Authorization, OAuth2, JWT**
  + **Spring Data JPA: Working with databases using ORM (Hibernate)**
  + **Spring Cloud: Microservices architecture, service discovery, and resilience patterns**
* **RESTful Web Services: Building REST APIs, HTTP methods, status codes, and JSON**
* **JDBC: Database connections, transactions, and queries**

**3. Front-End Technologies**

* **HTML5: Structure of web pages, semantic tags, forms**
* **CSS3: Styling web pages, Flexbox, Grid, Responsive Design, Media Queries**
* **JavaScript: Programming logic for the web, DOM manipulation, Event handling** 
  + **ES6 Features: Arrow functions, async/await, destructuring, template literals, modules**
  + **JavaScript Frameworks:** 
    - **React.js: Building dynamic user interfaces, components, state management**
    - **Angular: Building full-fledged web applications, two-way data binding, RxJS**
    - **Vue.js: Lightweight and flexible framework for building UIs**
* **TypeScript: Superset of JavaScript with type annotations**
* **AJAX: Asynchronous JavaScript and XML for making non-blocking requests**
* **HTML5 APIs: LocalStorage, SessionStorage, Geolocation, WebSockets**

**4. Databases (Back-end)**

* **SQL: Writing queries, joins, subqueries, aggregations, and CRUD operations**
* **Relational Databases: MySQL, PostgreSQL, Oracle, SQL Server**
* **NoSQL Databases: MongoDB, Cassandra, CouchDB**
* **ORM (Object Relational Mapping): Hibernate, Spring Data JPA, MyBatis**

**5. Version Control and CI/CD**

* **Git: Version control, branching, merging, rebasing, GitHub, GitLab, Bitbucket**
* **CI/CD Pipelines: Jenkins, GitLab CI, CircleCI, Travis CI, Azure DevOps**
* **Docker: Containerization for Java applications**
* **Kubernetes: Orchestrating containers in production**

**6. Testing (Back-end)**

* **JUnit: Unit testing for Java applications**
* **Mockito: Mocking dependencies for unit tests**
* **Integration Testing: Test integration between components, database, etc.**
* **Selenium: Automated browser testing for front-end functionality**

**7. Build Tools**

* **Maven: Dependency management, build automation for Java projects**
* **Gradle: Alternative to Maven, build automation**
* **Docker: Creating containerized Java applications for easier deployment**

**8. Cloud Services & Deployment**

* **AWS: EC2, S3, RDS, Lambda, API Gateway**
* **Azure: Azure App Service, Azure Functions, Azure SQL Database**
* **Google Cloud: GCE, App Engine, Cloud SQL, Firebase**
* **Containerization & Orchestration: Docker, Kubernetes**

**9. Additional Tools and Frameworks**

* **Thymeleaf: Template engine for server-side rendering in Spring MVC**
* **Swagger/OpenAPI: API documentation and testing**
* **RabbitMQ/Kafka: Messaging queues for handling asynchronous communication**
* **GraphQL: Query language for APIs**

**10. Soft Skills**

* **Agile/Scrum: Understanding Agile development methodology**
* **Team Collaboration: Working in a team using tools like Jira, Trello**
* **Problem-Solving: Writing efficient algorithms, debugging, and troubleshooting**
* **Communication: Interacting with clients, explaining technical details to non-technical stakeholders**

**YV:**

**Alright, let's build a comprehensive roadmap for becoming a full-stack developer with Java and**

**React. This roadmap emphasizes practical skills and a structured learning path.**

**Phase 1: Java Fundamentals (3-6 Months)**

● Core Java:

○ Variables, data types, operators, control flow (if/else, loops).

○ Object-oriented programming (OOP): classes, objects, inheritance, polymorphism,

encapsulation, abstraction.

○ Collections framework: Lists, Sets, Maps.

○ Exception handling.

○ File I/O.

○ Multithreading and concurrency (basic concepts).

**● Java Development Tools:**

○ Install and configure JDK (Java Development Kit).

○ Use an IDE (Integrated Development Environment) like IntelliJ IDEA, Eclipse, or VS

Code with Java extensions.

○ Learn Maven or Gradle for build automation and dependency management.

○ Understand Git for version control (GitHub or GitLab).

**● Java Standard Libraries:**

○ Understand the core Java APIs.

○ Learn about common utility classes.

**● Database Basics (SQL):**

○ Understand relational databases (e.g., MySQL, PostgreSQL).

○ Learn SQL: SELECT, INSERT, UPDATE, DELETE, JOINs, basic database design.

○ Setup a local database.

**Phase 2: Backend Development with Java (4-8 Months)**

● Spring Framework:

○ Spring Core: Dependency injection, IoC (Inversion of Control).

○ Spring MVC: Building RESTful APIs.

○ Spring Boot: Rapid application development, auto-configuration.

○ Spring Data JPA: Object-Relational Mapping (ORM) and database interaction.

○ Spring Security: Authentication and authorization.

**● RESTful API Design:**

○ Understand REST principles (HTTP methods, status codes, resource

representation).

○ Design and implement RESTful APIs using Spring Boot.

○ Learn about API documentation (Swagger/OpenAPI).

**● Testing:**

○ Unit testing with JUnit and Mockito.

○ Integration testing with Spring Test.

○ API testing with Postman or curl.

**● Database Integration:**

**○** Connect your Java application to a database.

○ Perform CRUD operations using Spring Data JPA or JDBC.

○ Understand database transactions.

**● Deployment:**

○ Learn to deploy your backend application to a cloud platform (e.g., AWS, Azure,

Google Cloud) or a local server.

○ Understand basic Linux commands.

**● Logging:**

**○** Implement logging into your java application using Log4j, or SLF4j.

**Phase 3: Frontend Development with React (4-8 Months)**

**● JavaScript Fundamentals:**

**○ ES6+ features (arrow functions, destructuring, promises, async/await).**

**○ DOM manipulation.**

**○ Asynchronous JavaScript.**

**●** React Basics:

○ Components, JSX, props, state.

○ Component lifecycle.

**○ Handling events.**

**○ React Hooks (useState, useEffect, useContext, etc.).**

**○ React Router: Client-side routing.**

**● React Development Tools:**

**○ Node.js and npm (or yarn).**

**○ Create React App (or Vite).**

**○ VS Code with React extensions.**

**○ Browser developer tools.**

**● State Management:**

**○ Context API.**

**○ Redux or Zustand (for larger applications).**

**● API Integration:**

**○ Fetch API or Axios for making HTTP requests to your backend.**

**○ Handling asynchronous data.**

**● UI Libraries:**

**○ Material UI, Ant Design, or Tailwind CSS for styling.**

**● Testing:**

**○ Jest and React Testing Library for unit and component testing.**

**● Deployment:**

**○ Deploy your React application to platforms like Netlify, Vercel, or AWS S3.**

**Phase 4: Full-Stack Integration and Deployment (2-4 Months)**

**● Connecting Frontend and Backend:**

○ Integrate your React frontend with your Spring Boot backend.

○ Handle data flow between the two layers.

○ Address CORS (Cross-Origin Resource Sharing) issues.

**● Full-Stack Application Development:**

○ Build a complete full-stack application (e.g., a to-do list, a blog, an e-commerce

platform).

○ Focus on user experience and application performance.

● Containerization (Docker):

○ Containerize your backend and frontend applications using Docker.

○ Learn Docker Compose for managing multi-container applications.

● Cloud Deployment (Advanced):

○ Deploy your full-stack application to a cloud platform using Docker containers and

orchestration tools (e.g., Kubernetes).

○ Learn about cloud services (e.g., AWS ECS, Azure Kubernetes Service, Google

Kubernetes Engine).

● CI/CD (Continuous Integration/Continuous Deployment):

○ Set up CI/CD pipelines using tools like Jenkins, GitHub Actions, or GitLab CI/CD.

○ Automate the build, test, and deployment processes.

Key Considerations:

● Practice Regularly: Consistent practice is crucial for mastering these technologies.

● Build Projects: Work on personal projects to solidify your understanding.

● Learn from Documentation and Tutorials: Refer to official documentation and reputable

online resources.

● Join Communities: Participate in online forums and communities (e.g., Stack Overflow,

Reddit, Discord) to get help and learn from others.

● Stay Updated: The web development landscape is constantly evolving, so stay

up-to-date with the latest trends and technologies.

● Focus on Fundamentals: Ensure a solid understanding of core concepts before moving

on to advanced topics.

● Break down large projects: Start with smaller, more manageable projects and gradually

increase the complexity.

This roadmap provides a structured path, but feel free to adjust it based on your learning style

and goals. Good luck on your full-stack development journey!

1. So every application in java starts with a class name and class name must be similar to file name ; (In Java, every application begins with a class name, and that class must match the filename.)
2. Every line of code that runs in Java must be inside a class.
3. public class Main {

public static void main(String[] args) {

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

}

}

**public static void main(String[] args) /main():** Any code inside the main() method will be executed.

**System.out.println():**

Inside the main() method, we can use the println() method to print a line of text to the screen:

The curly braces {} marks the beginning and the end of a block of code.

System is a built-in Java class that contains useful members, such as out, which is short for "output". The println() method, short for "print line", is used to print a value to the screen (or a file).

println() adds a new line next to the written line

example:

public class Main {

public static void main(String[] args) {

System.out.println("Hello World!");

System.out.print("I am learning Java.");

System.out.print("It is awesome!");

}

}

Output:

Hello World!

I am learning Java.It is awesome!

THE println adds a newline at the end of the output;

print will not insert a new line at the end of output.

**Double quotes:**

unlike text, we don't put numbers inside double quotes if we want to calculate them

example :

System.out.println(3 + 3); // Outputs: 6 (because it's arithmetic)

System.out.println("3" + "3"); // Outputs: 33 (String concatenation)

**Comments:**

Single line comments: // this is a single line comment

Multi line comment:

/\* I am lahari

I am leanrign java

This is a multiline comment \*/

**Variables:**

there are different **types** of variables, for example:

* String - stores text, such as "Hello". String values are surrounded by double quotes
* int - stores integers (whole numbers), without decimals, such as 123 or -123
* float - stores floating point numbers, with decimals, such as 19.99 or -19.99
* char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
* boolean - stores values with two states: true or false

CODE:

public class Main{

public static void main(String[] args){

int num=5;

int n=1;

String name="lahari";

char a='c';

System.out.println(num+" "+n+" "+name+" "+a);

System.out.printf("%d %d %s %c %n",num,n,name,a);

}

}

OUTPUT:

C:\Users\lahar>java Main

5 1 lahari c

5 1 lahari c

if you assign a new value to an existing variable, it will overwrite the previous value:

Example

Change the value of myNum from 15 to 20:

int myNum = 15;

myNum = 20; // myNum is now 20

System.out.println(myNum);

**Final Variables:**

If you don't want others (or yourself) to overwrite existing values, use the final keyword (this will declare the variable as "final" or "constant", which means unchangeable and read-only):

Example

final int myNum = 15;

myNum = 20; // will generate an error: cannot assign a value to a final variable

**A demonstration of how to declare variables of other types:**

Example

int myNum = 5; // Integer(Whole number)

float myFloatNum = 5.99f;

char myLetter = 'D';

boolean myBool = true;

String myText = "Hello";

|  |  |
| --- | --- |
| byte | Stores whole numbers from -128 to 127 |
| short | Stores whole numbers from -32,768 to 32,767 |
| int | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| double | Stores fractional numbers. Sufficient for storing 15 to 16 decimal digits |
| boolean | Stores true or false values |
| char | Stores a single character/letter or ASCII values |

**int (32-bit signed integer) Range:**

**-2,147,483,648 to 2,147,483,647**

* The range of a signed 32-bit integer (int) is:−231 to 231−1-2^{31} \text{ to } 2^{31} - 1−231 to 231−1
  + **Minimum:** −231=−2,147,483,648-2^{31} = -2,147,483,648−231=−2,147,483,648
  + **Maximum:** 231−1=2,147,483,6472^{31} - 1 = 2,147,483,647231−1=2,147,483,647

**long (64-bit signed integer) Range:**

**-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807**

* The range of a signed 64-bit integer (long) is:−263 to 263−1-2^{63} \text{ to } 2^{63} - 1−263 to 263−1
  + **Minimum:** −263=−9,223,372,036,854,775,808-2^{63} = -9,223,372,036,854,775,808−263=−9,223,372,036,854,775,808
  + **Maximum:** 263−1=9,223,372,036,854,775,8072^{63} - 1 = 9,223,372,036,854,775,807263−1=9,223,372,036,854,775,807

Scientific Numbers

A floating-point number can also be a scientific number with an "e" to indicate the power of 10:

Example

float f1 = 35e3f;

double d1 = 12E4d;

System.out.println(f1);

System.out.println(d1);

Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main differences between **primitive** and **non-primitive** data types are:

* Primitive types in Java are predefined and built into the language, while non-primitive types are created by the programmer (except for String).
* Non-primitive types can be used to call methods to perform certain operations, whereas primitive types cannot.
* Primitive types start with a lowercase letter (like int), while non-primitive types typically starts with an uppercase letter (like String).
* Primitive types always hold a value, whereas non-primitive types can be null.

Examples of non-primitive types are [Strings](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp), [Classes](https://www.w3schools.com/java/java_classes.asp) etc. You will learn more about these in a later chapter.

Widening Casting

Widening casting is done automatically when passing a smaller size type to a larger size type:

Example[Get your own Java Server](https://www.w3schools.com/java/java_server.asp)

public class Main {

public static void main(String[] args) {

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9

System.out.println(myDouble); // Outputs 9.0

}

}

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_casting_wide)

Narrowing Casting

Narrowing casting must be done manually by placing the type in parentheses () in front of the value:

Example

public class Main {

public static void main(String[] args) {

double myDouble = 9.78d;

int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble); // Outputs 9.78

System.out.println(myInt); // Outputs 9

}

}

Try

**IMPORTANT TOPICS**

**Method overloading:**

With method overloading , multiple methods can have the same name with different parameters: Multiple methods can have the same name as long as the number and/or type of parameters are different.

<https://www.w3schools.com/java/java_methods_overloading.asp>

**Scope:**

public class Main {

public static void main(String[] args) {

// Code here CANNOT use x

{ // This is a block

// Code here CANNOT use x

int x = 100;

// Code here CAN use x

System.out.println(x);

} // The block ends here

int x = 10;

System.out.println(x);

}

}

Output:

100

10

**Recursion:**

Function calling itself, It is commonly used for problems that can be broken down into smaller, similar subproblems.

A recursive function must have:

1. **Base Case** – A condition that stops the recursion.
2. **Recursive Case** – The function calls itself with a modified argument, gradually approaching the base case.

**Example (Factorial using Recursion in Python):**

python

CopyEdit

def factorial(n):

if n == 0: # Base Case

return 1

else:

return n \* factorial(n - 1) # Recursive Case

print(factorial(5)) # Output: 120

Recursion is useful for tasks like tree traversal, searching, sorting (e.g., QuickSort, MergeSort), and solving mathematical problems like Fibonacci sequences. However, excessive recursion can lead to high memory usage and stack overflow errors.

**OOPS (Object Oriented Programming)**

* object-oriented programming is about creating objects that contain both data and methods.
* OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
* a class is a template for objects, and an object is an instance of a class.

**CLASEES IN JAVA**

Everything in Java is associated with classes and objects, along with its attributes and methods.

public class Main{

int x=5;

public static void main(String[] args){

Main obj1=new Main();

System.out.println(obj1.x);

}

}

**Syntax for creating an object:**

**Class\_name object\_name=new class\_name();**

Even though x is out of scope, it can access x because it is an object of Main so it can access the attributes and functions inside the Main

‘x’ is an instance variable or non static variable of Main class

**MULTIPLE CLASSES IN JAVA**

**So**

**1. What happens if we don't compile Main.java and only type javac Second.java? Will it give an error?**

**Scenario:**

If you type:

sh

CopyEdit

javac Second.java

without compiling Main.java first, **it will not give an error** as long as Main.java is in the same directory.

**Why?**

* When you compile Second.java, the compiler **automatically looks for Main.class** because Second is trying to use Main.
* If Main.class is already there (from a previous compilation), there’s no issue.
* But **if Main.java has never been compiled**, then Main.class doesn’t exist, and the compiler **will compile Main.java automatically** while compiling Second.java.

**Error Case:**

If Main.java contains errors and cannot be compiled, then javac Second.java will fail because it depends on Main.class.

**2. What is the difference between javac filename.java and java filename?**

| **Command** | **Purpose** |
| --- | --- |
| javac filename.java | **Compiles** the Java file into bytecode (.class file) |
| java filename | **Executes** the compiled bytecode |

**Explanation:**

* javac Main.java → Converts Main.java into Main.class (bytecode)
* java Main → Runs the compiled Main.class file (if it contains a main method)

**Example:**

sh

CopyEdit

javac Second.java # Compiles Second.java to Second.class

java Second # Runs Second.class

🔹 **Important:** Do **not** include .java when running with java. Just use the class name (java Second), **not** java Second.java.

**3. When we execute Second (java Second), will it call Main first?**

Yes! Here's the **flow of execution** when you run java Second:

1. **main() in Second starts execution.**
2. Inside main(), this line runs:

java

CopyEdit

Main myObj = new Main();

* + This **creates an object of Main**.
  + When we create an object, the **constructor of Main is called** (even if we don’t explicitly define one, Java provides a default constructor).

1. Now, myObj.x is accessed:

java

CopyEdit

System.out.println(myObj.x);

* + Since x is an **instance variable**, it belongs to the object myObj.
  + The JVM **fetches x from the Main class** and prints its value (5).

So, **execution order is:**

* Second.main() starts
* Main object is created
* x is accessed from Main
* The value 5 is printed

**1. What happens if we don't compile Main.java and only type javac Second.java? Will it give an error?**

**Scenario:**

If you type:

sh

CopyEdit

javac Second.java

without compiling Main.java first, **it will not give an error** as long as Main.java is in the same directory.

**Why?**

* When you compile Second.java, the compiler **automatically looks for Main.class** because Second is trying to use Main.
* If Main.class is already there (from a previous compilation), there’s no issue.
* But **if Main.java has never been compiled**, then Main.class doesn’t exist, and the compiler **will compile Main.java automatically** while compiling Second.java.

**Error Case:**

If Main.java contains errors and cannot be compiled, then javac Second.java will fail because it depends on Main.class.

**2. What is the difference between javac filename.java and java filename?**

| **Command** | **Purpose** |
| --- | --- |
| javac filename.java | **Compiles** the Java file into bytecode (.class file) |
| java filename | **Executes** the compiled bytecode |

**Explanation:**

* javac Main.java → Converts Main.java into Main.class (bytecode)
* java Main → Runs the compiled Main.class file (if it contains a main method)

**Example:**

sh

CopyEdit

javac Second.java # Compiles Second.java to Second.class

java Second # Runs Second.class

🔹 **Important:** Do **not** include .java when running with java. Just use the class name (java Second), **not** java Second.java.

**3. When we execute Second (java Second), will it call Main first?**

Yes! Here's the **flow of execution** when you run java Second:

1. **main() in Second starts execution.**
2. Inside main(), this line runs:

java

CopyEdit

Main myObj = new Main();

* + This **creates an object of Main**.
  + When we create an object, the **constructor of Main is called** (even if we don’t explicitly define one, Java provides a default constructor).

1. Now, myObj.x is accessed:

java

CopyEdit

System.out.println(myObj.x);

* + Since x is an **instance variable**, it belongs to the object myObj.
  + The JVM **fetches x from the Main class** and prints its value (5).

So, **execution order is:**

* Second.main() starts
* Main object is created
* x is accessed from Main
* The value 5 is printed

**We cannot access a non static variable(instance variable of class) without an object .**

Each object (obj1 and obj2) **has its own copy of the instance variable x**. The method display() prints the value of x specific to each object.

**Step-by-Step Execution**

java

CopyEdit

public class Main {

int x = 5; // Instance variable

public static void main(String[] args) {

Main obj1 = new Main(); // Object 1

obj1.x = 10; // Change x for obj1

obj1.display(); // ✅ Prints 10

Main obj2 = new Main(); // Object 2 (NEW OBJECT)

obj2.x = 20; // Change x for obj2

obj2.display(); // ✅ Prints 20

}

public void display() {

System.out.println(x); // Prints the x of the object calling it

}

}

**Understanding Why Output is 10 20**

✅ **Object 1 (obj1) has x = 10**  
✅ **Object 2 (obj2) has x = 20**

Even though display() is the same function, it prints **different values** because:

* obj1.display(); prints obj1.x, which is 10.
* obj2.display(); prints obj2.x, which is 20.

Each **object maintains its own copy** of x, and changes to one object do **not** affect another object.

**Key Concept**

🔹 **Instance variables (x) belong to objects, not the class itself.**  
🔹 **Each object has its own copy of x.**  
🔹 **The display() function prints x for the specific object that calls it.**

**How to Prove Objects Are Independent?**

Try printing x **before modifying it**:

java

CopyEdit

public class Main {

int x = 5;

public static void main(String[] args) {

Main obj1 = new Main();

System.out.println("Before change: " + obj1.x); // ✅ Prints 5

obj1.x = 10;

obj1.display(); // ✅ Prints 10

Main obj2 = new Main();

System.out.println("Before change: " + obj2.x); // ✅ Prints 5 (new object)

obj2.x = 20;

obj2.display(); // ✅ Prints 20

}

public void display() {

System.out.println(x);

}

}

**Expected Output**

yaml

CopyEdit

Before change: 5

10

Before change: 5

20

🔹 This proves that obj1 and obj2 **start with x = 5 independently**  
🔹 Then, **changing x in obj1 does not affect obj2**

**Final Answer**

✅ **Yes! The method display() prints the value of x specific to the object calling it.**  
✅ **Since obj1 and obj2 are different objects, they have separate values for x.**  
✅ **Methods do not store values—they just print the current state of an object's variables.**

If you don't want the ability to override existing values, declare the attribute as final:

Example

public class Main {

**final** int x = 10;

public static void main(String[] args) {

Main myObj = new Main();

myObj.x = 25; // will generate an error: cannot assign a value to a **final** variable

System.out.println(myObj.x);

}

}

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_attributes_final)

**final keyword is used for some values like pi=3.14…… which cannot be changed in future**

1. **If a class (Class1) has a non-static attribute (attribute1) and a non-static method (method1), then:**
   * **They cannot be accessed directly inside main()** because main() is static.
   * **To access them, you must create an object of Class1 and use it.**

**Example:**

java

CopyEdit

public class Class1 {

int attribute1 = 10; // Non-static attribute

void method1() { // Non-static method

System.out.println("Inside method1: " + attribute1);

}

public static void main(String[] args) {

Class1 obj = new Class1(); // ✅ Create an object

System.out.println(obj.attribute1); // ✅ Access non-static attribute

obj.method1(); // ✅ Call non-static method

}

}

**Output:**

scss

CopyEdit

10

Inside method1: 10

1. **If you use the static keyword for attribute1 or method1, then:**
   * They **belong to the class itself**, not to an object.
   * They **can be accessed directly inside main() without an object**.

**Example:**

java

CopyEdit

public class Class1 {

static int attribute1 = 10; // Static attribute

static void method1() { // Static method

System.out.println("Inside static method1: " + attribute1);

}

public static void main(String[] args) {

System.out.println(attribute1); // ✅ Direct access, no object needed

method1(); // ✅ Direct access, no object needed

}

}

**Output:**

scss

CopyEdit

10

Inside static method1: 10

**🔹 Final Summary:**

✔ **Non-static members** (attributes/methods) **need an object** to be accessed inside main().  
✔ **Static members** (attributes/methods) **belong to the class** and can be accessed **directly** inside main().  
✔ **Inside a static method (like main()), you cannot use non-static members directly—only through an object**.

**Static variables can be accessed anywhere in the class** (inside static & non-static methods).  
✔ **They can be accessed from another class using ClassName.variableName**.  
✔ **They should be accessed using the class name (best practice).**  
✔ **Cannot use this with static variables.**

You're getting the concepts perfectly! Keep going, girl! 🚀🔥

**CONSTRUCTORS:**

**A constructor in Java is a special method that is used to initialize objects**

**Note that**

**the constructor name must match the class name, and it cannot have a return type (like void)**

**Constructor is called when the object is created.**

Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

Example

public class Main {

int x;

public Main(int y) {

x = y;

}

public static void main(String[] args) {

Main myObj = new Main(5);

System.out.println(myObj.x);

}

}

// Outputs 5

/\*public class Main {

int Year;

String Name;

public Main(int year, String name) {

this.Year = year;

this.Name = name;

}

public static void main(String[] args) {

Main myCar = new Main(1969, "Mustang");

System.out.println(myCar.Year + " " + myCar.Name);

}

}

// Outputs 1969 Mustang

\*/

public class Main {

int Year;

String Name;

public Main(int year, String name) {

int Year = year;

int Name = name;

}

public static void main(String[] args) {

Main myCar = new Main(1969, "Mustang");

System.out.println(myCar.modelYear + " " + myCar.modelName);

}

}

//outputs 0 NULL

**Summarizing Your Thought Process:**

1. **Instance variables (Year, Name)** exist **inside the class** but **outside any method or constructor**.
2. When you **declared local variables with the same names** inside the constructor:

java

CopyEdit

int Year = year;

String Name = name;

* + These became **local variables** inside the constructor.
  + They **shadowed** the instance variables (Year and Name).
  + The **instance variables remained unchanged**.

1. When you **printed myCar.Year and myCar.Name**,
   * You were actually printing the **unchanged instance variables**.
   * Since instance variables for int **default to 0** and String **defaults to null**, you got:

yaml

CopyEdit

Output: 0 null

**How this Solves the Problem:**

To **correctly modify the instance variables**, use this:

java

CopyEdit

this.Year = year; // Assigns the parameter 'year' to the instance variable 'Year'

this.Name = name; // Assigns the parameter 'name' to the instance variable 'Name'

Now, when the constructor **executes**, it **actually modifies the instance variables**, so when you use myCar.Year and myCar.Name, they **hold the correct values**.

**Final Thought:**

Yes! **After the instance variables get changed inside the constructor**, when you use the **object to access them**, you are now seeing the **updated values** instead of the default ones.

In Java, this is a **reference variable** that refers to the **current object**. It is mainly used when there is a **name conflict** between instance variables and local variables or to pass the current object as a parameter.

**this variable in java**

public class Main {

int year;

String name;

public Main(int year, String name) {

this.year = year;

this.name = name;

}

public void display() {

System.out.println(year + " " + name);

}

public static void main(String[] args) {

Main myCar = new Main(1969, "Mustang");

myCar.display();

}

}

Output:

C:\Users\lahar>java Main

1969 Mustang

Without this :

public class Main {

int year;

String name;

public Main(int year, String name) {

year = year;

name = name;

}

public void display() {

System.out.println(year + " " + name);

}

public static void main(String[] args) {

Main myCar = new Main(1969, "Mustang");

myCar.display();

}

}

Output:

0 null

**Can java have multiple classes in one file then if we write code how do we save it:**

ood question! 😃 The short answer is **No, a Java file can have multiple classes**, but with some rules. Let me explain.

**Can a Java File Have Multiple Classes?**

Yes, a single .java file **can** contain multiple classes, but **only one class can be public**.

**1️⃣ A Java File with Multiple Classes (Only One public Class)**

java

CopyEdit

// File: Main.java

public class Main { // ✅ Public class matches the filename

public static void main(String[] args) {

Helper h = new Helper();

h.sayHello();

}

}

class Helper { // ✅ Another class in the same file (not public)

void sayHello() {

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

}

}

**Output:**

javascript

CopyEdit

Hello from Helper class!

✅ **Rules:**  
✔ You **must** save the file with the name of the public class (e.g., Main.java).  
✔ Other classes in the file **cannot be public**.

**2️⃣ A Java File with Multiple Non-Public Classes**

If there’s **no public class**, you can name the file **anything** and still compile it.

java

CopyEdit

// File: Anything.java (Can have any name)

class FirstClass {

void display() {

System.out.println("Hello from FirstClass");

}

}

class SecondClass {

void show() {

System.out.println("Hello from SecondClass");

}

}

class ThirdClass {

public static void main(String[] args) {

FirstClass f = new FirstClass();

SecondClass s = new SecondClass();

f.display();

s.show();

}

}

**Output:**

csharp

CopyEdit

Hello from FirstClass

Hello from SecondClass

✅ **Rules:**  
✔ If there’s **no public class**, the file **can have any name**.  
✔ You can have **multiple non-public classes** in one file.

**3️⃣ What If You Have More Than One public Class?**

If you try this:

java

CopyEdit

// File: Main.java

public class Main {

public static void main(String[] args) {

System.out.println("Hello from Main");

}

}

public class Helper { // ❌ ERROR! You cannot have two public classes in one file

void sayHello() {

System.out.println("Hello from Helper");

}

}

**You get a compilation error:**

kotlin

CopyEdit

error: class Helper is public, should be declared in a file named Helper.java

🚨 **Fix:**  
Move Helper to a separate file Helper.java:

java

CopyEdit

// File: Helper.java

public class Helper {

void sayHello() {

System.out.println("Hello from Helper");

}

}

**Conclusion**

✔ A .java file **can have multiple classes**.  
✔ **Only one class can be public**, and the filename **must match** the public class name.  
✔ If there’s **no public class**, the file can have **any name**.  
✔ If you need **multiple public classes**, they **must be in separate files**.

**JAVA MODIFIERS:**

**public keywrd:**

The public keyword is an access modifier, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

* Access Modifiers - controls the access level
* Non-Access Modifiers - do not control access level, but provides other functionality

IMPORTANT: <https://www.w3schools.com/java/java_modifiers.asp>

You want a **real-world example** of **Encapsulation** that makes sense beyond just code. Let’s take an **ATM Machine** as an example.

**🔹 Real-World Example: ATM Machine**

**How does it relate to Encapsulation?**

Think about how you interact with an **ATM machine** when you withdraw money.

**Step 1: You can't access bank data directly**

* Imagine the ATM is connected to a **bank database**.
* Can you directly change your bank balance? **No!**
* You can only access your balance **through the ATM interface.**

**Step 2: ATM provides controlled access**

* You enter a **PIN** (authentication).
* You choose an option like **Check Balance** or **Withdraw Money**.
* The system **internally updates your balance**, but you **never see** how it works inside.

**Step 3: Encapsulation in Action**

* Your **bank balance** is like a **private variable** → It is **hidden from direct access**.
* The **ATM provides methods** like checkBalance() and withdrawMoney(amount), which act as **getters and setters**.
* You cannot directly modify your balance; you must go through these **public methods**.

**🔹 Java Code Example: ATM System**

Now, let's translate this into Java:

// BankAccount class (Encapsulation)

public class BankAccount {

private double balance; // Private variable (hidden data)

// Constructor to set initial balance

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

// Getter method to check balance

public double getBalance() {

return balance;

}

// Setter method to withdraw money (controlled access)

public void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount; // Deduct the amount

System.out.println("Withdrawal successful! New Balance: " + balance);

} else {

System.out.println("Insufficient balance or invalid amount.");

}

}

}

// ATM Machine (Using Encapsulation)

class ATM {

public static void main(String[] args) {

BankAccount myAccount = new BankAccount(5000); // Creating an account with $5000

System.out.println("Current Balance: " + myAccount.getBalance()); // Checking balance

myAccount.withdraw(1000); // Withdrawing $1000

myAccount.withdraw(5000); // Trying to withdraw more than balance (Invalid)

}

}

**🔹 Output of the Code**

Current Balance: 5000.0

Withdrawal successful! New Balance: 4000.0

Insufficient balance or invalid amount.

**🔹 How Does This Relate to Encapsulation?**

1️⃣ **Data Hiding:**

* The balance variable is private, so **nobody can access it directly** (like how you can't change your bank balance directly).

2️⃣ **Controlled Access with Methods:**

* You can **only check balance (getBalance())** or **withdraw money (withdraw())**, just like how you interact with an ATM.
* If you try to withdraw more than your balance, it **won't allow it** (just like a real ATM).

3️⃣ **Prevents Unauthorized Access:**

* If someone tries to access balance directly, they **can't** because it's private.
* They must go through the **ATM’s controlled functions** (like how you need a PIN in real life).

**🔹 Why is Encapsulation Important?**

✔ **Protects sensitive data** (like your bank balance).  
✔ **Provides controlled access** to modify values.  
✔ **Prevents accidental modification** of data.  
✔ **Increases security and reliability** in applications.

**💡 Quick Question:**

If balance were **public**, what could go wrong? 🤔

myAccount.balance = 999999; // Modify balance directly (BAD)

This is why **Encapsulation protects data from direct modification!** 🚀

**💡 Encapsulation in Simple Terms**

1️⃣ **Make variables private** so they **cannot be accessed or modified directly** from outside the class.  
2️⃣ **Provide public methods (functions)** inside the class that **allow controlled access** to those private variables.  
3️⃣ **Changes to the private variable** can only happen **through these public methods**, which ensure that the updates follow certain rules.

**Java Packages & API**

A package in Java is used to group related classes. Packages are divided into two categories:

* Built-in Packages (packages from the Java API)
* User-defined Packages (create your own packages)
* The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
* To use a class or a package from the library, you need to use the import keyword:
* import *package*.*name*.*Class*; // Import a single class
* import *package*.*name*.\*; // Import the whole package

**Step 1: What is the Scanner Class?**

The Scanner class in Java is **used to read user input** from different sources like:

* The **keyboard (console input)**
* **Files**
* **Strings**
* **Streams (like network input, etc.)**

**Step 2: Creating a Scanner Object**

Here’s the line of code that creates a Scanner object:

Scanner myObj = new Scanner(System.in);

Let’s break this down **piece by piece**.

**(1) Scanner myObj**

* This declares a **variable** named myObj of type Scanner.
* It will store the **Scanner object** once it's created.

**(2) new Scanner(System.in)**

* This **creates a new Scanner object**.
* The Scanner constructor **requires an input source** (where it should read data from).
* System.in is passed as an **argument** to tell Scanner **where to read input from**.

**Step 3: What is System.in?**

**System.in is an Input Stream**

* System.in is a **predefined input stream** in Java.
* It represents **keyboard input** from the console.
* In simple words, it tells the Scanner to **take input from the keyboard**.

**Why do we pass System.in?**

* The Scanner class needs a **source** to read from.
* System.in means: **"Read from the keyboard (console)."**
* Without this, Scanner wouldn’t know **where to get input**.

**Step 4: How Does Scanner Work?**

Let’s look at the full example:

java

CopyEdit

import java.util.Scanner; // Importing Scanner class

class MyClass {

public static void main(String[] args) {

Scanner myObj = new Scanner(System.in); // Step 1: Creating Scanner object

System.out.println("Enter username"); // Step 2: Asking user for input

String userName = myObj.nextLine(); // Step 3: Taking input from keyboard

System.out.println("Username is: " + userName); // Step 4: Printing input

myObj.close(); // Step 5: Closing Scanner (important to prevent resource leak)

}

}

**Step-by-Step Explanation**

1️⃣ **Scanner myObj = new Scanner(System.in);**

* Creates a Scanner object that reads **keyboard input**.

2️⃣ **System.out.println("Enter username");**

* Prints a message asking the user to enter something.

3️⃣ **String userName = myObj.nextLine();**

* **Waits for user input.**
* The user types something and presses Enter.
* The entered text is stored in userName.

4️⃣ **System.out.println("Username is: " + userName);**

* Displays the text the user entered.

5️⃣ **myObj.close();**

* Closes the Scanner object to free up system resources (Good Practice ✅).

what happens if we **don’t pass anything** to Scanner:

java

CopyEdit

Scanner myScanner = new Scanner(); // ❌ No arguments passed

🔴 **This will give an ERROR!**

**⛔ Compilation Error**

When you try to compile this, you'll get an error like:

vbnet

CopyEdit

Error: Scanner() in java.util.Scanner cannot be applied to ()

**Why Does This Happen?**

* The Scanner class **requires a source** to read input from.
* There is **no constructor in Scanner** that allows **empty arguments**.
* Scanner **must know where to read data from**, but in this case, **it doesn’t know what to do!**

**✅ Fix: Pass an Input Source**

To fix this, we **must** pass an input source, such as: 1️⃣ **Keyboard (Console Input) →** new Scanner(System.in)  
2️⃣ **A String →** new Scanner("Hello World")  
3️⃣ **A File →** new Scanner(new File("data.txt"))

**🔹 Example of a Correct Usage**

**1️⃣ Correct: Using System.in**

java

CopyEdit

Scanner myScanner = new Scanner(System.in); // ✅ Works fine

**2️⃣ Correct: Using a String**

java

CopyEdit

Scanner myScanner = new Scanner("Hello, World!"); // ✅ Works fine

System.out.println(myScanner.nextLine()); // Output: Hello, World!

myScanner.close();

**3️⃣ Correct: Using a File**

java

CopyEdit

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class FileExample {

public static void main(String[] args) {

try {

Scanner fileScanner = new Scanner(new File("data.txt")); // ✅ Works fine

System.out.println(fileScanner.nextLine());

fileScanner.close();

} catch (FileNotFoundException e) {

System.out.println("File not found!");

}

}

}

**🔹 Key Takeaways**

✔ **Scanner() without arguments does NOT exist in Java!**  
✔ We **must** provide a source (keyboard, string, or file).  
✔ **If we don’t pass anything, we get a compilation error** because Scanner wouldn’t know where to get input from.

So now you know **why we pass System.in!** 🎯🔥

**Understanding Java Packages Step by Step**

**📌 Step 1: Creating a Java File Inside a Package**

**What Happens in the Code?**

java

CopyEdit

package pack1; // Declaring that this class belongs to the package "pack1"

public class PackageClass {

public static void main(String[] args) {

System.out.println("This is my package!");

}

}

**Why Do We Use package pack1;?**

* Java organizes code using **packages** (just like folders on your computer).
* The **package** keyword tells Java **where this class belongs**.
* pack1 is the **package name**, meaning **this file must be inside a folder named pack1**.

**📌 Step 2: Saving the File Correctly**

* You saved the file as PackageClass.java in \*\*C:\Users\lahar`.
* But **it should be inside a folder named pack1**, like this:

mathematica

CopyEdit

C:\Users\lahar

├── 📁 pack1

│ ├── PackageClass.java ✅ (Source File)

* **This matches the package name (pack1).**

**📌 Step 3: Compiling the Java File**

**Command:**

bash

CopyEdit

javac -d . pack1\PackageClass.java

**What Happens Here?**

1. **javac** → Java compiler
2. **-d .** →
   * -d tells Java to **store the compiled file** in the right package folder.
   * . means **"use the current directory"** (C:\Users\lahar).
3. **pack1\PackageClass.java** → The Java file to be compiled.

**After Compilation, What Happens?**

A **new folder structure** is created:

mathematica

CopyEdit

C:\Users\lahar

├── 📁 pack1

│ ├── PackageClass.class ✅ (Compiled File)

│ ├── PackageClass.java ✅ (Source File)

* **PackageClass.class** is created in pack1\, meaning Java now recognizes it as part of the package.

**📌 Step 4: Running the Java Package**

**Command:**

bash

CopyEdit

java pack1.PackageClass

**What Happens Here?**

1. **java** → Runs the Java program.
2. **pack1.PackageClass** → Java looks inside the pack1 package for PackageClass.class.
3. It executes the main method, and you see the output:

kotlin

CopyEdit

This is my package!

**🔴 What If We Hadn’t Used a Package?**

If we removed the package pack1; line and just saved PackageClass.java in C:\Users\lahar\, we could **compile and run it normally** like this:

bash

CopyEdit

javac PackageClass.java

java PackageClass

✔ **But without a package, Java wouldn’t organize the file inside a folder**  
✔ **Packages help keep big projects structured!**

**🔹 Final Summary (What We Learned)**

✅ **package pack1;** makes Java put the class inside a **folder named pack1**.  
✅ **javac -d . pack1\PackageClass.java** creates a **compiled file inside the package folder**.  
✅ **java pack1.PackageClass** runs the program correctly.

**INHERITANCE**

class Parent {

protected String brand = "Ford";

public void showBrand() {

System.out.println("Brand: " + brand);

}

}

class Child extends Parent {

private String modelName = "M"; // Private to Child class

public void showDetails() {

System.out.println("Model: " + modelName + ", Brand: " + brand);

}

}

public class Test {

public static void main(String[] args) {

Parent p = new Parent();

// p.modelName; // ❌ ERROR: Parent can't access Child's private field

Child c = new Child();

c.showDetails(); // ✅ Works because showDetails() is inside Child

System.out.println(c.modelName); // ✅ Works because c is an instance of Child

}

}

**Key Takeaways**

1. **A parent class cannot access child class members**, even if they are public or protected.
2. **A child class inherits from the parent, but not the other way around.**
3. To access modelName, you must use an instance of Child, not Parent.

**Java Access Modifiers Overview**

There are **four** types of access modifiers in Java:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Modifier** | **Same Class** | **Same Package (Subclass & Non-Subclass)** | **Subclass (Different Package)** | **Other Classes** |
| private | ✅ Yes | ❌ No | ❌ No | ❌ No |
| (default) (No modifier) | ✅ Yes | ✅ Yes | ❌ No | ❌ No |
| protected | ✅ Yes | ✅ Yes | ✅ Yes | ❌ No |
| public | ✅ Yes | ✅ Yes | ✅ Yes | ✅ Yes |

Polymorphism is one of the core principles of **Object-Oriented Programming (OOP)**. It allows one interface to be used for different types of actions, enabling **flexibility and reusability** in code.

**Definition of Polymorphism**

The word "Polymorphism" comes from the Greek words **"poly" (many) and "morph" (forms)**, meaning **"many forms"**. In Java, polymorphism allows objects of different classes to be treated as objects of a common superclass.

**Types of Polymorphism in Java**

Java supports **two types of polymorphism**:

1. **Compile-time Polymorphism (Method Overloading)**
2. **Runtime Polymorphism (Method Overriding)**

**1. Compile-time Polymorphism (Method Overloading)**

This happens when **multiple methods in the same class** share the same name but have **different parameters** (method signature). The method to be executed is determined at **compile-time**.

**Example: Method Overloading**

java

CopyEdit

class MathOperations {

// Method with two integer parameters

int add(int a, int b) {

return a + b;

}

// Method with three integer parameters

int add(int a, int b, int c) {

return a + b + c;

}

// Method with double parameters (Different data type)

double add(double a, double b) {

return a + b;

}

}

public class Test {

public static void main(String[] args) {

MathOperations obj = new MathOperations();

System.out.println("Sum of 2 numbers: " + obj.add(5, 10)); // Calls add(int, int)

System.out.println("Sum of 3 numbers: " + obj.add(5, 10, 15)); // Calls add(int, int, int)

System.out.println("Sum of doubles: " + obj.add(5.5, 10.5)); // Calls add(double, double)

}

}

**Key Points:**

✔ Methods have the **same name** but **different parameter lists**.  
✔ The method call is resolved **at compile time**.  
✔ It allows **improving code readability** by using the same method name for similar operations.

**2. Runtime Polymorphism (Method Overriding)**

This occurs when **a subclass provides a specific implementation of a method that is already defined in its parent class**. The method call is resolved **at runtime** based on the object's actual type.

**Example: Method Overriding**

java

CopyEdit

class Animal {

void makeSound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

@Override

void makeSound() {

System.out.println("Cat meows");

}

}

public class Test {

public static void main(String[] args) {

Animal myAnimal = new Dog(); // Upcasting

myAnimal.makeSound(); // Output: Dog barks

myAnimal = new Cat(); // Upcasting

myAnimal.makeSound(); // Output: Cat meows

}

}

**Key Points:**

✔ The **subclass overrides** the method of the parent class.  
✔ Method resolution happens **at runtime** based on the actual object type.  
✔ **Upcasting** (Animal myAnimal = new Dog();) allows treating a subclass as its parent type, enabling **dynamic method dispatch**.

**Method Overloading vs. Method Overriding**

| **Feature** | **Method Overloading (Compile-time)** | **Method Overriding (Runtime)** |
| --- | --- | --- |
| **Definition** | Multiple methods in the same class with the same name but different parameters | Subclass provides a different implementation of a method defined in the parent class |
| **Method Resolution** | Compile-time (Static Binding) | Runtime (Dynamic Binding) |
| **Return Type** | Can be the same or different | Must be the same or a covariant type (subtype of return type) |
| **Parameters** | Must be different in number or type | Must be the same as in the parent class |
| **Access Modifier** | Can be different | Cannot be more restrictive than the parent method |
| **Usage** | Improves code readability | Enables runtime behavior changes |

**3. Polymorphism Using Interfaces**

Polymorphism also applies to **interfaces** because different classes can implement the same interface, but each class can have its own implementation.

**Example: Interface-Based Polymorphism**

java

CopyEdit

interface Shape {

void draw();

}

class Circle implements Shape {

public void draw() {

System.out.println("Drawing a Circle");

}

}

class Rectangle implements Shape {

public void draw() {

System.out.println("Drawing a Rectangle");

}

}

public class Test {

public static void main(String[] args) {

Shape s1 = new Circle(); // Upcasting

s1.draw(); // Output: Drawing a Circle

Shape s2 = new Rectangle(); // Upcasting

s2.draw(); // Output: Drawing a Rectangle

}

}

**Key Points:**

✔ Interfaces allow different classes to **implement the same method differently**.  
✔ The actual method execution is **determined at runtime**.  
✔ Helps achieve **loose coupling** and **scalability** in large applications.

**4. Real-World Example of Polymorphism**

Imagine an **online payment system** where multiple payment methods exist (Credit Card, PayPal, UPI).

java

CopyEdit

abstract class Payment {

abstract void pay();

}

class CreditCard extends Payment {

void pay() {

System.out.println("Payment done using Credit Card");

}

}

class PayPal extends Payment {

void pay() {

System.out.println("Payment done using PayPal");

}

}

public class Test {

public static void main(String[] args) {

Payment payment1 = new CreditCard(); // Upcasting

payment1.pay(); // Output: Payment done using Credit Card

Payment payment2 = new PayPal(); // Upcasting

payment2.pay(); // Output: Payment done using PayPal

}

}

**Key Points:**

✔ Different payment methods **share a common interface (Payment)**.  
✔ Each payment method **overrides the pay() method** differently.  
✔ **Polymorphism allows dynamic method selection** at runtime.

**5. Why Use Polymorphism?**

✅ **Code Reusability** – The same method name can be used for different purposes.  
✅ **Extensibility** – New subclasses can be added without modifying existing code.  
✅ **Maintainability** – Helps in writing flexible and easily maintainable code.  
✅ **Loose Coupling** – Interfaces allow different implementations without tight dependency.

**Final Summary**

| **Concept** | **Type** | **Key Idea** |
| --- | --- | --- |
| **Method Overloading** | Compile-time Polymorphism | Same method name, different parameters (in the same class) |
| **Method Overriding** | Runtime Polymorphism | Subclass provides a specific implementation of a parent class method |
| **Interface Polymorphism** | Runtime Polymorphism | Different classes implement the same interface with different behaviors |

1. **Method Overloading (Compile-time polymorphism)** – Defining multiple methods with the same name but different parameters.
2. **Method Overriding (Runtime polymorphism)** – Redefining a method in a subclass that is already defined in the parent class.

Here’s an implementation of an add method using both types of polymorphism:

**1. Method Overloading Example (Compile-time Polymorphism)**

java

CopyEdit

class Addition {

// Overloaded methods with different parameter lists

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

String add(String a, String b) {

return a + b; // Concatenation

}

}

public class OverloadingExample {

public static void main(String[] args) {

Addition addition = new Addition();

System.out.println("Integer addition: " + addition.add(5, 10)); // Calls int version

System.out.println("Double addition: " + addition.add(5.5, 2.2)); // Calls double version

System.out.println("String addition: " + addition.add("Hello, ", "Lahari!")); // Calls String version

}

}

**2. Method Overriding Example (Runtime Polymorphism)**

java

CopyEdit

class Parent {

int add(int a, int b) {

return a + b;

}

}

class Child extends Parent {

@Override

int add(int a, int b) {

System.out.println("Overridden method in Child class");

return (a + b) \* 2; // Modified behavior

}

}

public class OverridingExample {

public static void main(String[] args) {

Parent obj1 = new Parent();

Parent obj2 = new Child(); // Runtime polymorphism

System.out.println("Parent class add method: " + obj1.add(5, 10));

System.out.println("Child class overridden add method: " + obj2.add(5, 10));

}

}

**Explanation:**

* **Method Overloading**: The add method is overloaded in the Addition class, allowing us to add integers, doubles, and even concatenate strings.
* **Method Overriding**: The Child class overrides the add method of the Parent class. At runtime, the overridden method in Child is executed instead of the parent class method when using a Parent reference.

**Abstract Class in Java – Detailed Explanation with Examples**

An **abstract class** in Java is a special type of class that **cannot be instantiated** directly. It is used as a **blueprint for other classes**.

**1. Declaring an Abstract Class**

An **abstract class** is declared using the abstract keyword.

**Example: Abstract Class Declaration**

java

CopyEdit

abstract class Animal {

// Abstract method (no implementation)

abstract void makeSound();

}

✔ Here, Animal is an **abstract class**.  
✔ The method makeSound() is an **abstract method** because it has no body.  
✔ Any class that **extends Animal must provide its own implementation of makeSound()**.

**2. Abstract Class Cannot Be Instantiated**

You **cannot create an object** of an **abstract class** directly.

**Example: Attempting to Instantiate an Abstract Class**

java

CopyEdit

abstract class Animal {

abstract void makeSound();

}

public class Test {

public static void main(String[] args) {

Animal a = new Animal(); // ❌ ERROR: Cannot instantiate abstract class

}

}

❌ **This will give a compilation error** because abstract classes **must be extended** before being used.

✔ Instead, you must **create a subclass** that provides an implementation of the abstract method.

**3. Abstract Class Can Have Both Abstract and Concrete Methods**

An **abstract class can contain:**

* **Abstract methods** (without implementation)
* **Concrete methods** (with implementation)

**Example: Abstract and Concrete Methods**

java

CopyEdit

abstract class Animal {

abstract void makeSound(); // Abstract method

void sleep() { // Concrete method

System.out.println("Sleeping...");

}

}

class Dog extends Animal {

void makeSound() { // Implementing the abstract method

System.out.println("Dog barks");

}

}

public class Test {

public static void main(String[] args) {

Dog dog = new Dog();

dog.makeSound(); // Output: Dog barks

dog.sleep(); // Output: Sleeping...

}

}

**Explanation:**

✔ The Animal class has an **abstract method** (makeSound()) and a **concrete method** (sleep()).  
✔ The Dog class **must implement** makeSound().  
✔ The Dog class **inherits** the sleep() method from Animal.

**4. Abstract Class Allows Partial Implementation**

* Abstract classes allow you to **partially implement** methods.
* Subclasses **must implement the remaining abstract methods**.

**Example: Partial Implementation**

java

CopyEdit

abstract class Vehicle {

abstract void start(); // Abstract method

void stop() { // Concrete method

System.out.println("Vehicle stopped.");

}

}

class Car extends Vehicle {

void start() { // Implementing the abstract method

System.out.println("Car starts with a key.");

}

}

public class Test {

public static void main(String[] args) {

Car car = new Car();

car.start(); // Output: Car starts with a key.

car.stop(); // Output: Vehicle stopped.

}

}

✔ The Vehicle class has an **abstract method start()**, which is **implemented in Car**.  
✔ The stop() method is **inherited as it is**.

**5. Calling Parent Constructor using super():**

* **When to Use super()**: The super keyword is used to call the parent class constructor. It is required when the parent class constructor takes parameters.

**Example:**

java

CopyEdit

class Parent {

Parent(String name) {

System.out.println("Parent constructor called with name: " + name);

}

}

class Child extends Parent {

Child(String name) {

super(name); // Calls the Parent constructor with the argument 'name'

System.out.println("Child constructor called");

}

}

public class Test {

public static void main(String[] args) {

Child child = new Child("John");

}

}

**Conclusion**: super(name) is necessary when the parent class has a parameterized constructor. It passes the name to the parent constructor to initialize it before executing the child constructor.

**6. Empty Constructor and Default Constructor Behavior:**

* **When No Constructor is Defined**: If a parent class has no constructor or an empty constructor, the child class does not need to explicitly call super(). Java will automatically call the default constructor of the parent class.

**Example:**

java

CopyEdit

class Parent {

Parent() {

System.out.println("Parent constructor called");

}

}

class Child extends Parent {

Child() {

// No need to use super() explicitly

}

}

public class Test {

public static void main(String[] args) {

Child child = new Child(); // Output: Parent constructor called

}

}

**Conclusion**: If the parent class has a default constructor, the child class does not need to explicitly call super(), and the parent class's default constructor is called automatically.

**7. Summary of Constructor Flow:**

* When a **child object is created**, the **child class constructor is called first**.
* **Parent constructor** is invoked before the child constructor completes, either implicitly (if there is a default constructor) or explicitly (if the parent constructor requires arguments).

**So here we need to use super keyword if the parent class have arguments else we don’t have to**

**Step-by-Step Workflow:**

1. **Child Object Creation:**
   * **When an object of the child class is created (new Child()), the child class constructor is invoked first.**
2. **Implicit or Explicit Call to Parent Constructor:**
   * **If the child class does not explicitly call a constructor of the parent class using super(), the default constructor of the parent class is called automatically by Java.**
   * **If the child class does explicitly call the parent constructor using super(arg), the specified parent constructor (which matches the parameters) is called first.**
3. **Parent Constructor Execution:**
   * **The parent class constructor runs first.** 
     + **If the parent constructor has parameters, they are passed from the child constructor using super(arg).**
     + **If the parent class constructor has no parameters (default constructor), it is invoked automatically.**
4. **Initialization in Parent Constructor:**
   * **The parent class constructor initializes any fields or performs any setup defined within it. After this, the control returns to the child class constructor.**
5. **Child Constructor Execution:**
   * **After the parent constructor finishes, the child class constructor continues its execution.**
   * **The child class constructor can initialize additional fields or execute additional logic as needed.**
6. **Method Execution:**
   * **After the constructor execution is completed, any method calls (like draw()) in the child class can be executed.**

**Example Breakdown:**

**1. No Parameters in Parent Constructor:**

java

CopyEdit

class Parent {

Parent() {

System.out.println("Parent constructor called");

}

}

class Child extends Parent {

Child() {

// No explicit call to super(), Java calls Parent() automatically

System.out.println("Child constructor called");

}

}

public class Test {

public static void main(String[] args) {

Child child = new Child(); // Output:

// Parent constructor called

// Child constructor called

}

}

* **Child Object Creation** → Calls Child() constructor.
* **Parent Constructor** → Implicitly called (since no super() is used in Child).
* **Child Constructor** → Executes after the Parent constructor finishes.

**2. Parent Constructor with Parameters:**

java

CopyEdit

class Parent {

Parent(String name) {

System.out.println("Parent constructor called with name: " + name);

}

}

class Child extends Parent {

Child(String name) {

super(name); // Explicitly calling the parent constructor with a parameter

System.out.println("Child constructor called");

}

}

public class Test {

public static void main(String[] args) {

Child child = new Child("John"); // Output:

// Parent constructor called with name: John

// Child constructor called

}

}

* **Child Object Creation** → Calls Child(String name) constructor.
* **Child Constructor** → First, super(name) is invoked to call the **Parent constructor** with a parameter.
* **Parent Constructor** → Initializes the name field and prints its output.
* **Child Constructor** → Executes after the Parent constructor finishes, printing its own output.

**Key Points to Remember:**

* The **child constructor** is always called first.
* If the **parent class** has a constructor that takes arguments, the **child class must explicitly call the parent constructor** using super().
* If the **parent class** has a **default constructor** (no arguments), it is **automatically called**.
* **Parent constructor** execution completes before **child constructor** execution.

This is the **workflow** of how constructors are executed when creating a child object in Java. The **parent constructor** is always called before the **child constructor** to ensure that the parent’s fields and initialization logic are handled first.

**THE ABOVE NOTES IS COMPLETELY MY UNDERSTANDING**

**6. Abstract Class with Static Methods**

An **abstract class can have static methods** because static methods **belong to the class, not instances**.

**Example: Static Method in Abstract Class**

java

CopyEdit

abstract class Utility {

static void printMessage() {

System.out.println("Hello from static method in abstract class!");

}

}

public class Test {

public static void main(String[] args) {

Utility.printMessage(); // ✅ Allowed

}

}

✔ **Static methods can be called using the class name** because they **don’t depend on an object**.

**Abstract Class Extending Another Abstract Class**

An **abstract class can inherit from another abstract class**.

**Example: Abstract Class Inheritance**

abstract class Animal {

abstract void sound();

void sleep() { // Concrete method

System.out.println("Sleeping...");

}

}

abstract class Mammal extends Animal { // Extending abstract class

abstract void walk(); // Additional abstract method

}

class Dog extends Mammal {

void sound() {

System.out.println("Dog barks");

}

void walk() {

System.out.println("Dog walks on four legs");

}

}

public class Test {

public static void main(String[] args) {

Dog d = new Dog();

d.sound(); // Output: Dog barks

d.walk(); // Output: Dog walks on four legs

d.sleep(); // Output: Sleeping...

}

}

✔ Mammal extends Animal, so it **inherits the sleep() method**.  
✔ Dog extends Mammal and **implements both abstract methods**.

**9. When to Use Abstract Classes?**

✅ **When you want to enforce a structure for subclasses**  
✅ **When code reuse is needed** (common methods for multiple subclasses)  
✅ **When you need partial implementation**  
✅ **When you are designing a framework or API**

**Final Summary**

| **Feature** | **Description** |
| --- | --- |
| **Abstract Class** | A class that cannot be instantiated but can be inherited |
| **Abstract Methods** | Methods with no body, must be implemented by subclasses |
| **Concrete Methods** | Methods with implementation, inherited by subclasses |
| **Can Have Constructor?** | ✅ Yes |
| **Can Have Static Methods?** | ✅ Yes |
| **Can Implement an Interface?** | ✅ Yes |
| **Can Extend Another Abstract Class?** | ✅ Yes |
| **Can Be Instantiated?** | ❌ No |

**Conclusion**

Abstract classes are **powerful tools** in Java that provide **structure, code reuse, and flexibility** for object-oriented programming. They **enforce a contract** on subclasses while allowing some default implementations.

Yes, you are correct! In Java, **a class can only extend one abstract class**. This is because Java does not support **multiple inheritance** for classes. This means that **a child class can extend only one parent class**, whether that parent class is abstract or concrete.

**Example: One Child Class Extending One Abstract Class**

java

CopyEdit

abstract class Animal {

abstract void sound();

}

abstract class Mammal extends Animal {

abstract void walk();

}

class Dog extends Mammal {

void sound() {

System.out.println("Dog barks");

}

void walk() {

System.out.println("Dog walks on four legs");

}

}

public class Test {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound(); // Output: Dog barks

dog.walk(); // Output: Dog walks on four legs

}

}

✔ In this case, Dog **extends** Mammal, and Mammal extends Animal.  
✔ Dog is a **subclass of only one class**: Mammal.

**What If You Try to Extend Multiple Classes?**

If you try to extend more than one class, whether abstract or not, **you'll get a compilation error**.

**Example: Compilation Error for Multiple Class Inheritance**

java

CopyEdit

abstract class Animal {

abstract void sound();

}

abstract class Mammal {

abstract void walk();

}

class Dog extends Animal, Mammal { // ❌ Compilation error: Cannot extend multiple classes

void sound() {

System.out.println("Dog barks");

}

void walk() {

System.out.println("Dog walks on four legs");

}

}

public class Test {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound();

dog.walk();

}

}

❌ **This will result in a compilation error** because Java does not allow **multiple inheritance** for classes.

**Why Does Java Limit Inheritance to One Class?**

Java avoids **multiple inheritance** for the following reasons:

1. **Ambiguity**: If a class inherits from multiple classes with the same method name, it would create ambiguity about which method to call.
2. **Complexity**: Multiple inheritance would increase the complexity of the language's design, leading to more difficult maintenance and code understanding.

**What About Interfaces?**

While a class can only extend **one abstract class**, Java **does support multiple inheritance** through **interfaces**. A class can **implement multiple interfaces**. This is Java’s way of allowing classes to inherit behavior from multiple sources while avoiding the issues that come with multiple class inheritance.

**Example: Multiple Interface Implementation**

java

CopyEdit

interface Animal {

void sound();

}

interface Mammal {

void walk();

}

class Dog implements Animal, Mammal {

public void sound() {

System.out.println("Dog barks");

}

public void walk() {

System.out.println("Dog walks on four legs");

}

}

public class Test {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound(); // Output: Dog barks

dog.walk(); // Output: Dog walks on four legs

}

}

✔ **Dog implements both Animal and Mammal interfaces**, allowing multiple inheritance of behavior.

**Summary**

* **A class can only extend one abstract class** because Java does not support multiple inheritance of classes.
* **A class can implement multiple interfaces**, allowing for multiple inheritance of behavior through interfaces.

## **Java Inner Classes**

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

### **Example**[**Get your own Java Server**](https://www.w3schools.com/java/java_server.asp)

class OuterClass {

int x = 10;

class InnerClass {

int y = 5;

}

}

public class Main {

public static void main(String[] args) {

OuterClass myOuter = new OuterClass();

**OuterClass.InnerClass myInner = myOuter.new InnerClass();**

System.out.println(myInner.y + myOuter.x);

}

}

// Outputs 15 (5 + 10)

[Try it Yourself »](https://www.w3schools.com/java/tryjava.asp?filename=demo_class_inner)

Static Inner Class

An inner class can also be static, which means that you can access it without creating an object of the outer class:

Example

class OuterClass {

int x = 10;

static class InnerClass {

int y = 5;

}

}

public class Main {

public static void main(String[] args) {

OuterClass.InnerClass myInner = new OuterClass.InnerClass();

System.out.println(myInner.y);

}

}

// Outputs 5

**Key Point:**

While an inner class can access the **outer class's instance members** (variables and methods), it **cannot directly call** those members without an instance of the **outer class**.

**Example:**

java

CopyEdit

class OuterClass {

int outerVar = 10; // Instance variable of OuterClass

// Method of OuterClass

public int outerMethod() {

return outerVar;

}

// InnerClass - non-static inner class

class InnerClass {

int innerVar = 20; // Instance variable of InnerClass

// Method of InnerClass

public int innerMethod() {

return innerVar;

}

// Inner class trying to directly access outer class's members

public void accessOuter() {

// Can access outer class's instance variable and method

System.out.println("Accessing outerVar directly: " + outerVar); // ✅ Works: inner class has direct access

System.out.println("Accessing outerMethod directly: " + outerMethod()); // ✅ Works: inner class can call outer method

}

}

}

public class Test {

public static void main(String[] args) {

OuterClass outer = new OuterClass(); // Create instance of OuterClass

OuterClass.InnerClass inner = outer.new InnerClass(); // Create instance of InnerClass

// Access methods of the inner class

inner.innerMethod(); // Accesses inner class's own method

// Access outer class members via inner class method

inner.accessOuter(); // ✅ Works: inner class can access outer class's members

}

}

**Explanation:**

1. **Accessing Outer Class Members from Inner Class**:
   * Inside the method accessOuter() of the inner class, we can directly access outerVar (an instance variable of OuterClass) and outerMethod() (a method of OuterClass).
   * This works because an **inner class** has a reference to the outer class's instance, allowing it to access its members.
2. **The Key Point — Direct Access via Inner Class**:
   * Notice, **you can access outer class members directly inside the inner class** (as shown in accessOuter()).
   * However, if you try to **access outer class members** from outside the inner class **using the inner class instance**, you will encounter an issue. The **inner class** does not **directly** expose the **outer class's members**. You still need the outer class's instance to access its members.

**To Illustrate the Issue:**

Let’s try to call the outer class's method using the inner class instance directly from the main method:

java

CopyEdit

public class Test {

public static void main(String[] args) {

OuterClass outer = new OuterClass(); // Create instance of OuterClass

OuterClass.InnerClass inner = outer.new InnerClass(); // Create instance of InnerClass

// Attempting to directly access outer class members through the inner class instance

// System.out.println(inner.outerVar); // ❌ ERROR: Cannot access outerVar directly via inner class instance

// System.out.println(inner.outerMethod()); // ❌ ERROR: Cannot access outerMethod directly via inner class instance

}

}

**Why Does This Fail?**

* **inner.outerVar and inner.outerMethod()** do not work because **outerVar and outerMethod() are members of the OuterClass**, not InnerClass.
* The **inner class has access to the outer class's members**, but you **cannot access the outer class's members directly** via the **inner class instance**. Instead, you need to use the **outer class's instance** to call its members, as the inner class is bound to an instance of the outer class.

**Correct Way to Access the Outer Class's Members:**

You would need to call the outer class's methods or access its variables via an **instance of the outer class**, as shown in the earlier example.

**Output for the Working Example:**

java

CopyEdit

Accessing outerVar directly: 10

Accessing outerMethod directly: 10

**Conclusion:**

While the **inner class can access** the **outer class's members**, it cannot directly **access them through the inner class instance** from the outside. The **inner class** is bound to the **outer class instance**, and you need that instance to access the outer class's members properly.

Let me know if you need further clarification!

**You said:**

so you mean in class itself we can use but not with the object

**ChatGPT said:**

Yes, exactly!

**In an inner class:**

* **Within the inner class itself**, you can access the **outer class's members (instance variables and methods)** directly, because the inner class has a reference to the outer class instance implicitly.
* However, **when accessing through an object (instance) of the inner class**, you **cannot directly access** the **outer class's members** unless you have an **explicit reference to the outer class**.

**To clarify:**

1. **Within the inner class**:
   * You can directly access the outer class's instance variables and methods because the inner class has access to the outer class instance by default.
2. **From the outside** (i.e., in the main method or another class):
   * You cannot access the outer class's members directly through the **inner class instance** (because the inner class does not directly expose outer class members).

**Example:**

java

CopyEdit

class OuterClass {

int outerVar = 10; // Instance variable of OuterClass

// Method of OuterClass

public int outerMethod() {

return outerVar;

}

// InnerClass - non-static inner class

class InnerClass {

int innerVar = 20; // Instance variable of InnerClass

// Method of InnerClass

public int innerMethod() {

return innerVar;

}

// Accessing outer class members directly within inner class

public void accessOuter() {

System.out.println("Accessing outerVar directly: " + outerVar); // ✅ Works: inner class has direct access

System.out.println("Accessing outerMethod directly: " + outerMethod()); // ✅ Works: inner class can call outer method

}

}

}

public class Test {

public static void main(String[] args) {

OuterClass outer = new OuterClass(); // Create instance of OuterClass

OuterClass.InnerClass inner = outer.new InnerClass(); // Create instance of InnerClass

// Accessing outer class members via the inner class's method

inner.accessOuter(); // ✅ Works: inner class can access outer class's members

}

}

**Breakdown:**

* **Inside InnerClass**, you can **access outerVar and outerMethod() directly** because InnerClass has an implicit reference to the **outer class instance**.
* **In the main method**, if you try to access **outerVar or outerMethod()** via inner (the instance of InnerClass), it won't work because the inner class does not directly expose the outer class's members outside the inner class. You need the outer class instance to access its members.

**Conclusion:**

* **Inside the inner class**, you can **use the outer class's members** directly.
* **Outside the inner class**, you need a **reference to the outer class** to access its members (not just the inner class's instance).

Let me know if you'd like more examples or further clarification!

**You said:**

ok give me allthe examples i made wring and right above with exampkes ans conclusion at end

**ChatGPT said:**

Certainly! Let me summarize all the examples you've worked on, including both the **correct and incorrect** versions, and provide a **detailed explanation** and **conclusion** at the end.

**Example 1: Accessing outer class members from the inner class directly**

**Correct Example:**

java

CopyEdit

class OuterClass {

int outerVar = 10; // Instance variable of OuterClass

// Method of OuterClass

public int outerMethod() {

return outerVar;

}

// InnerClass - non-static inner class

class InnerClass {

int innerVar = 20; // Instance variable of InnerClass

// Method of InnerClass

public int innerMethod() {

return innerVar;

}

// Accessing outer class members directly within inner class

public void accessOuter() {

System.out.println("Accessing outerVar directly: " + outerVar); // ✅ Works: inner class has direct access

System.out.println("Accessing outerMethod directly: " + outerMethod()); // ✅ Works: inner class can call outer method

}

}

}

public class Test {

public static void main(String[] args) {

OuterClass outer = new OuterClass(); // Create instance of OuterClass

OuterClass.InnerClass inner = outer.new InnerClass(); // Create instance of InnerClass

// Accessing outer class members via the inner class's method

inner.accessOuter(); // ✅ Works: inner class can access outer class's members

}

}

**Explanation:**

* The inner class can directly access the outer class's **instance variables** and **methods** because it has an implicit reference to the **outer class** instance.
* In the accessOuter() method inside the inner class, both the instance variable outerVar and the method outerMethod() are directly accessible and can be called.

**Example 2: Attempting to access outer class members directly through the inner class instance**

**Incorrect Example (Results in an error):**

java

CopyEdit

public class Test {

public static void main(String[] args) {

OuterClass outer = new OuterClass(); // Create instance of OuterClass

OuterClass.InnerClass inner = outer.new InnerClass(); // Create instance of InnerClass

// Attempting to directly access outer class members through the inner class instance

System.out.println(inner.outerVar); // ❌ ERROR: Cannot access outerVar directly via inner class instance

System.out.println(inner.outerMethod()); // ❌ ERROR: Cannot access outerMethod directly via inner class instance

}

}

**Explanation:**

* You cannot access the **outer class's members** directly through the **inner class's instance** because the **inner class doesn't expose outer class members directly**. Even though the inner class can access them internally, it doesn't mean the inner class instance can be used to access the outer class's members from outside the class.

**Example 3: Accessing outer class members through the inner class method (Correct)**

**Correct Example:**

java

CopyEdit

class OuterClass {

int outerVar = 10; // Instance variable of OuterClass

// Method of OuterClass

public int outerMethod() {

return outerVar;

}

// InnerClass - non-static inner class

class InnerClass {

int innerVar = 20; // Instance variable of InnerClass

// Method of InnerClass

public int innerMethod() {

return innerVar;

}

// Inner class method accessing outer class's members

public void accessOuter() {

System.out.println("Accessing outerVar directly: " + outerVar); // ✅ Works: inner class has direct access

System.out.println("Accessing outerMethod directly: " + outerMethod()); // ✅ Works: inner class can call outer method

}

}

}

public class Test {

public static void main(String[] args) {

OuterClass outer = new OuterClass(); // Create instance of OuterClass

OuterClass.InnerClass inner = outer.new InnerClass(); // Create instance of InnerClass

// Accessing outer class members via the inner class's method

inner.accessOuter(); // ✅ Works: inner class can access outer class's members

}

}

**Explanation:**

* The accessOuter() method inside the inner class is able to access the **outer class's members** because the inner class can directly reference the outer class instance.
* You can **indirectly access** the outer class's members by calling a method in the inner class that internally accesses them.

**Conclusion:**

1. **Inside the inner class**:
   * The **inner class** has direct access to the **outer class's instance variables** and **methods**, because it is inherently linked to the **outer class's instance**.
2. **From outside the inner class**:
   * You **cannot directly access** the **outer class's members** via the **inner class instance**. The inner class does not expose the outer class's members to the outside world.
   * To access the **outer class's members**, you either need an **instance of the outer class** or need to call methods from the **inner class** that interact with the outer class.
3. **Correct approach**:
   * The inner class can access outer class members directly **inside the inner class itself**.
   * If you want to **access the outer class's members** from outside, you need to use **methods of the inner class** that make use of the outer class's members.

To access an inner class, you can: Create an object of the outer class, and then create an object of the inner class Create a class, and specify the inner keyword Create a class, and specify the friend keyword is it not possible to create only inner class object

**ChatGPT said:**

In Java, you **cannot create an instance of an inner class directly** without first creating an instance of the outer class, unless the inner class is **static**. Let's break down the different scenarios:

**1. Non-static Inner Class (Instance Inner Class)**

For **non-static inner classes**, you cannot directly create an instance of the inner class on its own. You must first create an instance of the **outer class** and then use that to create an instance of the inner class.

**Example:**

java

CopyEdit

class OuterClass {

class InnerClass {

void display() {

System.out.println("This is an inner class.");

}

}

}

public class Test {

public static void main(String[] args) {

OuterClass outer = new OuterClass(); // Creating outer class instance

OuterClass.InnerClass inner = outer.new InnerClass(); // Creating inner class instance using outer class instance

inner.display(); // Accessing method of inner class

}

}

**Explanation:**

* You **cannot directly create an instance of InnerClass** using new InnerClass() because it is a **non-static inner class**.
* You must first create an instance of the outer class (outer) and then use it to create the inner class object (outer.new InnerClass()).

**2. Static Inner Class**

For a **static inner class**, you can create an instance of the inner class **without needing an instance of the outer class**. This is because static inner classes are associated with the class itself, not with an instance of the outer class.

**Example:**

java

CopyEdit

class OuterClass {

static class InnerClass {

void display() {

System.out.println("This is a static inner class.");

}

}

}

public class Test {

public static void main(String[] args) {

OuterClass.InnerClass inner = new OuterClass.InnerClass(); // No need for outer class instance

inner.display(); // Accessing method of inner class

}

}

**Explanation:**

* You can create an instance of the **static inner class** directly using new OuterClass.InnerClass() without needing to create an instance of the outer class.

**3. Incorrect Statements in Your Question:**

* **"Create a class, and specify the inner keyword"**: In Java, there's no inner keyword. Inner classes are just regular classes defined inside another class.
* **"Create a class, and specify the friend keyword"**: Java does not have a friend keyword (like C++). All inner classes have access to the members of the outer class depending on the access modifiers used.

**Conclusion:**

* **Non-static inner class**: You **cannot create** an instance of the inner class directly. You must first create an instance of the **outer class** and then create the inner class object.
* **Static inner class**: You **can create** an instance of the inner class directly without needing an outer class instance.

### Summary of Differences:

| **Inner Class Type** | **Creation Syntax** | **Explanation** |
| --- | --- | --- |
| **Non-static Inner Class** | **OuterClass.InnerClass inner = outer.new InnerClass();** | **Requires an instance of the outer class** to create an instance of the inner class. |
| **Static Inner Class** | **OuterClass.InnerClass inner = new OuterClass.InnerClass();** | **Does not require an instance of the outer class** because the inner class is static and associated with the class itself, not an instance. |

**JAVA ABSTRACTION**

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or [**interfaces**](https://www.w3schools.com/java/java_interface.asp) (which you will learn more about in the next chapter).

**JAVA INTERFACES**

[**https://www.w3schools.com/java/java\_interface.asp**](https://www.w3schools.com/java/java_interface.asp)

**1. Interface with Method Definitions:**

java

CopyEdit

// Interface

interface Animal {

public void animalSound(); // Interface method (no body)

public void sleep(); // Interface method (no body)

}

// Pig "implements" the Animal interface

class Pig implements Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

public void sleep() {

System.out.println("The pig sleeps");

}

}

class Test {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

myPig.animalSound();

myPig.sleep();

}

}

**Conclusion:**

* **Interface methods** do not have a body; they are only declared.
* Any class that implements the interface must **override all the methods** of the interface.
* The interface methods are **implicitly abstract** and **public**.

**2. Interface Implementation by Multiple Classes:**

java

CopyEdit

interface Animal {

public void animalSound(); // Interface method (no body)

public void sleep(); // Interface method (no body)

}

// Pig "implements" the Animal interface

class Pig implements Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

public void sleep() {

System.out.println("The pig sleeps");

}

}

// Goat "implements" the Animal interface

class Goat implements Animal {

public void animalSound() {

System.out.println("The goat says: baa baa");

}

public void sleep() {

System.out.println("The goat sleeps");

}

}

class Test {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

Goat myGoat = new Goat(); // Create a Goat object

myPig.animalSound();

myPig.sleep();

myGoat.animalSound();

myGoat.sleep();

}

}

**Conclusion:**

* **Multiple classes** can implement the same interface.
* Each class must provide its own implementation of the interface methods.
* **Interface** provides a contract, and the implementing classes must follow this contract.

**3. Interface with Missing Method Implementation:**

java

CopyEdit

interface Animal {

public void animalSound(); // Interface method (no body)

public void sleep(); // Interface method (no body)

}

// Pig "implements" the Animal interface

class Pig implements Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

}

// Goat "implements" the Animal interface

class Goat implements Animal {

public void sleep() {

System.out.println("The goat sleeps");

}

}

class Test {

public static void main(String[] args) {

Pig myPig = new Pig(); // Create a Pig object

Goat myGoat = new Goat(); // Create a Goat object

myPig.animalSound();

// myPig.sleep(); // This would throw an error because Pig doesn't implement sleep()

myGoat.sleep();

// myGoat.animalSound(); // This would throw an error because Goat doesn't implement animalSound()

}

}

**Conclusion:**

* **All methods in the interface must be implemented** by the class. If a class implements an interface and does not provide implementations for all its methods, the code will not compile.
* If any method is missing, the class will produce a **compile-time error**.

**4. Interface with Multiple Implementations in One Class:**

java

CopyEdit

interface Animal {

void animalSound();

}

interface Sleepable {

void sleep();

}

class Pig implements Animal, Sleepable {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

public void sleep() {

System.out.println("The pig sleeps");

}

}

class Test {

public static void main(String[] args) {

Pig myPig = new Pig();

myPig.animalSound();

myPig.sleep();

}

}

**Conclusion:**

* A **class can implement multiple interfaces**. In this case, the Pig class implements both Animal and Sleepable interfaces.
* This allows a class to have behavior from multiple sources (interfaces).

**5. Interface with Default Methods:**

java

CopyEdit

interface Animal {

default void sleep() {

System.out.println("Animal sleeps");

}

}

class Pig implements Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

}

class Test {

public static void main(String[] args) {

Pig myPig = new Pig();

myPig.animalSound();

myPig.sleep(); // Using default method from the Animal interface

}

}

**Conclusion:**

* **Default methods** in interfaces provide a method implementation directly in the interface.
* **Classes** that implement the interface can use the default method, or override it if needed.

**Final Conclusion on Interfaces:**

* **Interfaces** define a contract (methods), and classes that implement the interface must provide concrete implementations for those methods.
* Interface methods are **implicitly abstract and public**.
* **Interfaces cannot contain method implementations** unless the method is declared default.
* **Multiple interfaces** can be implemented by a single class, allowing it to inherit behavior from multiple sources.
* **If a class implements an interface and does not provide implementations for all of the methods declared in the interface, a compile-time error occurs.**

**All the child interfaces should execute all the methods of the parent interfaces else it will give error**

**VERY IMPORTANT:**

Notes on Interfaces:

* Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interface methods are by default abstract and public
* Interface attributes are by default public, static and final
* An interface cannot contain a constructor (as it cannot be used to create objects)

**Example without explicit modifiers:**

java

CopyEdit

interface Animal {

String species = "Animal"; // Implicitly public, static, and final

void animalSound();

}

class Pig implements Animal {

public void animalSound() {

System.out.println("The pig says: wee wee");

}

}

public class Test {

public static void main(String[] args) {

System.out.println(Animal.species); // No error: can access the static field

// Attempt to modify the final field

// Animal.species = "Pig"; // ERROR: Cannot assign a value to final variable 'species'

}

}

**Explanation:**

1. The species field in the Animal interface is **implicitly**:
   * public: It can be accessed from outside the interface.
   * static: It belongs to the Animal interface, not an instance of it.
   * final: The value of the field cannot be changed.
2. In the Test class:
   * You can access Animal.species because it is **public** and **static**.
   * If you try to modify species (e.g., Animal.species = "Pig";), you will get a **compilation error** because the field is implicitly **final** and cannot be reassigned.

**Conclusion:**

* Even if you don't explicitly declare public static final, Java still treats interface fields as public static final by default.
* You **cannot change the value** of a field declared in an interface because it is **final**.
* You can **access the field** using the interface name, as it is **static**.

**Default Methods in Interfaces**

* **Default methods** are a special kind of method introduced in Java 8 that **provide a method body**. These methods are not abstract.
* **Default methods can have an implementation** in the interface itself, and they are still **public** by default.

**Access Modifiers in Interface Methods**

1. **Abstract methods** in interfaces are implicitly public and abstract. You cannot change their access modifier to anything other than public.
2. **Default methods** are also public by default. Even though they have a body, they are still **accessible** by any class that implements the interface. Unlike abstract methods, which must be implemented by the class, default methods have an implementation that can be inherited directly.

**Example: Access Modifiers and Default Methods**

java

CopyEdit

interface MyInterface {

// Abstract method (implicitly public and abstract)

void abstractMethod();

// Default method (implicitly public)

default void defaultMethod() {

System.out.println("This is a default method.");

}

}

class MyClass implements MyInterface {

// Implementing the abstract method

@Override

public void abstractMethod() {

System.out.println("Implemented abstract method.");

}

// Optional: Override the default method

@Override

public void defaultMethod() {

System.out.println("Overridden default method.");

}

}

public class Test {

public static void main(String[] args) {

MyClass obj = new MyClass();

obj.abstractMethod(); // Calls the implemented abstract method

obj.defaultMethod(); // Calls the overridden default method

}

}

**Key Points:**

1. **Abstract methods** in interfaces are implicitly **public** and **abstract**.
2. **Default methods** are **public** by default (you don't need to specify public explicitly).
3. **Default methods** can be inherited by classes, and they can be overridden if needed.
4. **Public** methods in an interface are accessible to any class that implements the interface, so you can access **default methods** from any class that implements the interface.

**Conclusion:**

* Interface methods are **public** by default, and **abstract** unless explicitly marked as default (in the case of default methods).
* **Default methods** in an interface are also **public** and can be accessed by any implementing class.
* The confusion might arise because the default keyword in Java refers to the method's implementation, but the method is still **public** and can be accessed like any public method from an implementing class.

**You said:**

can an interface implemet ab==notgher interfcae

**ChatGPT said:**

Yes, an interface **can** extend another interface in Java. This is similar to how classes can inherit from other classes. When an interface extends another interface, it inherits all the abstract methods of the parent interface. The child interface can add more methods if needed, but it doesn't need to implement the methods from the parent interface; it's up to the classes that implement the child interface to implement all methods.

Here’s how it works:

**Key Points:**

* An interface can **extend** another interface using the extends keyword.
* The child interface will inherit all abstract methods from the parent interface.
* A class implementing the child interface needs to provide implementations for all abstract methods of both the parent and the child interface.

**Example:**

java

CopyEdit

// Parent interface

interface Animal {

void animalSound(); // Abstract method

void sleep(); // Abstract method

}

// Child interface extending Animal

interface Mammal extends Animal {

void feedBaby(); // New method specific to Mammal

}

// Class implementing the child interface

class Dog implements Mammal {

// Implementing all methods from both Animal and Mammal

public void animalSound() {

System.out.println("Woof");

}

public void sleep() {

System.out.println("The dog sleeps");

}

public void feedBaby() {

System.out.println("The dog feeds its baby");

}

}

public class Test {

public static void main(String[] args) {

Dog myDog = new Dog();

myDog.animalSound(); // "Woof"

myDog.sleep(); // "The dog sleeps"

myDog.feedBaby(); // "The dog feeds its baby"

}

}

**In this example:**

* The Animal interface has two abstract methods: animalSound() and sleep().
* The Mammal interface extends Animal and adds an additional method: feedBaby().
* The Dog class implements Mammal and must provide implementations for all methods in both the Animal and Mammal interfaces.

**Conclusion:**

* Yes, an interface can extend another interface.
* The child interface inherits the abstract methods of the parent interface.
* Any class implementing the child interface must provide implementations for all inherited methods from both the parent and the child interfaces.

**You said:**

so we cannot create an object for interface?

**ChatGPT said:**

Correct! In Java, **you cannot create an object of an interface**. Interfaces are abstract by nature, meaning they don't have a complete implementation of their methods. Instead, they define a contract that other classes or interfaces can implement.

**Why can't we create an object of an interface?**

* An **interface** only declares method signatures (abstract methods) but does not provide concrete method implementations (unless there are default methods).
* An **interface** cannot be instantiated directly because it does not have the logic to perform any actions or behaviors by itself.
* The role of an interface is to be implemented by a class, which will then provide the concrete implementations for the methods declared in the interface.

**INTERFACES DONOT HAVE CONSTRUCTORS BECASE THEY CANT BE INTIATED YES**

* **Class → extends → Class**
* **Class → implements → Interface**
* **Interface → extends → Interface**

**Interface cannot extend a class; interface cannot implement a class.**

**Scanner Class in Java – Detailed Explanation**

The Scanner class in Java is a part of the java.util package and is used to take input from the user or from different sources like files, strings, or input streams.

**1. What is Scanner Class?**

The Scanner class provides methods to read input of various data types, including integers, floating-point numbers, strings, and more. It simplifies the process of accepting user input compared to older methods like BufferedReader.

**2. Importing Scanner Class**

Before using the Scanner class, you need to import it:

java

CopyEdit

import java.util.Scanner;

**3. Creating a Scanner Object**

To use Scanner, you need to create an instance of it:

java

CopyEdit

Scanner scanner = new Scanner(System.in);

* System.in is used to take input from the keyboard.
* The scanner object can now be used to read different types of input.

**4. Reading Different Types of Input**

The Scanner class provides multiple methods to read different types of data.

**4.1. Reading an Integer**

java

CopyEdit

Scanner scanner = new Scanner(System.in);

System.out.print("Enter an integer: ");

int num = scanner.nextInt();

System.out.println("You entered: " + num);

* nextInt() reads an integer from the user.

**4.2. Reading a Float**

java

CopyEdit

System.out.print("Enter a floating number: ");

float num = scanner.nextFloat();

System.out.println("You entered: " + num);

* nextFloat() reads a floating-point number.

**4.3. Reading a Double**

java

CopyEdit

System.out.print("Enter a double number: ");

double num = scanner.nextDouble();

System.out.println("You entered: " + num);

* nextDouble() reads a double-precision number.

**4.4. Reading a String (Single Word)**

java

CopyEdit

System.out.print("Enter a word: ");

String word = scanner.next();

System.out.println("You entered: " + word);

* next() reads a single word (stops at space).

**4.5. Reading a Full Line**

java

CopyEdit

System.out.print("Enter a sentence: ");

scanner.nextLine(); // To consume the leftover newline

String line = scanner.nextLine();

System.out.println("You entered: " + line);

* nextLine() reads the entire line, including spaces.

**4.6. Reading a Character**

Java's Scanner does not have a direct method to read a character, but we can achieve this using:

java

CopyEdit

System.out.print("Enter a character: ");

char ch = scanner.next().charAt(0);

System.out.println("You entered: " + ch);

* next().charAt(0) extracts the first character of the input.

**5. Handling Different Delimiters**

By default, Scanner uses whitespace as a delimiter. We can change it using useDelimiter().

Example:

java

CopyEdit

scanner.useDelimiter(",");

This makes Scanner read input values separated by a comma instead of whitespace.

**6. Checking for Input Type (Validation)**

Before reading input, you can check if the input is of a specific type.

Example:

java

CopyEdit

System.out.print("Enter an integer: ");

if (scanner.hasNextInt()) {

int num = scanner.nextInt();

System.out.println("Valid integer: " + num);

} else {

System.out.println("Invalid input!");

}

* hasNextInt() checks if the next input is an integer.

**7. Closing the Scanner**

After taking input, it's good practice to close the Scanner to free system resources.

java

CopyEdit

scanner.close();

However, **do not close** Scanner if you are using System.in and still need more input, as it will close the input stream permanently.

**8. Example Program Using Scanner**

Here’s a complete Java program that takes different types of user input using Scanner:

java

CopyEdit

import java.util.Scanner;

public class ScannerExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your name: ");

String name = scanner.nextLine();

System.out.print("Enter your age: ");

int age = scanner.nextInt();

System.out.print("Enter your salary: ");

double salary = scanner.nextDouble();

System.out.print("Enter your favorite letter: ");

char letter = scanner.next().charAt(0);

System.out.println("\n--- User Details ---");

System.out.println("Name: " + name);

System.out.println("Age: " + age);

System.out.println("Salary: " + salary);

System.out.println("Favorite Letter: " + letter);

scanner.close();

}

}

**Output:**

yaml

CopyEdit

Enter your name: Lahari

Enter your age: 24

Enter your salary: 50000.75

Enter your favorite letter: L

--- User Details ---

Name: Lahari

Age: 24

Salary: 50000.75

Favorite Letter: L

**9. Common Mistakes When Using Scanner**

1. **Skipping nextLine() after nextInt() or nextDouble()**
   * When using nextInt() or nextDouble(), it does not consume the newline character, which can cause issues when reading nextLine().
   * Fix:

java

CopyEdit

scanner.nextLine(); // Consume leftover newline

1. **Closing Scanner too early**
   * If you close Scanner, you cannot use System.in again in the program.

**10. Summary**

| **Method** | **Description** |
| --- | --- |
| nextInt() | Reads an integer |
| nextFloat() | Reads a floating-point number |
| nextDouble() | Reads a double-precision number |
| nextBoolean() | Reads a boolean (true/false) |
| next() | Reads a single word (stops at space) |
| nextLine() | Reads a full line (including spaces) |
| next().charAt(0) | Reads a single character |
| hasNextInt() | Checks if the next token is an integer |
| useDelimiter() | Changes the input delimiter |

**Conclusion**

The Scanner class is a powerful and easy way to handle user input in Java. It supports multiple data types, handles delimiters, and provides validation methods. However, you need to be mindful of input issues like newline characters and closing the Scanner object.

Would you like a practice exercise on Scanner? 😊

**You said:**

9. Common Mistakes When Using Scanner Skipping nextLine() after nextInt() or nextDouble() When using nextInt() or nextDouble(), it does not consume the newline character, which can cause issues when reading nextLine(). Fix: java Copy Edit scanner.nextLine(); // Consume leftover newline Closing Scanner too early If you close Scanner, you cannot use System.in again in the program. can you please explain this in detail

**ChatGPT said:**

**Common Mistakes When Using Scanner in Java**

When using Scanner in Java, two frequent mistakes can cause unexpected behavior:

1. **Skipping nextLine() after nextInt() or nextDouble()**
2. **Closing the Scanner too early**

Let’s dive deeper into these mistakes and how to fix them.

**1. Skipping nextLine() After nextInt() or nextDouble()**

**Why does this happen?**

* nextInt(), nextDouble(), nextFloat(), etc., only read the numeric value and **leave the newline character (\n) in the input buffer**.
* When nextLine() is called immediately after, it **reads the leftover newline (\n) instead of waiting for new user input**.
* This causes the program to **skip** the nextLine() input.

**Example: Unexpected Behavior**

java

CopyEdit

import java.util.Scanner;

public class ScannerIssue {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your age: ");

int age = scanner.nextInt(); // Reads integer but leaves newline `\n` in buffer

System.out.print("Enter your name: ");

String name = scanner.nextLine(); // Reads leftover `\n`, not actual user input

System.out.println("Your age: " + age);

System.out.println("Your name: " + name);

scanner.close();

}

}

**Expected Input:**

yaml

CopyEdit

Enter your age: 24

Enter your name: Lahari

**Actual Output:**

yaml

CopyEdit

Enter your age: 24

Enter your name: Your age: 24

Your name: (empty)

**Issue:**

* nextInt() reads 24, but **does not consume the newline** (\n) after it.
* nextLine() reads the **leftover newline**, so name is empty.

**Fix: Consume the Newline**

To fix this, **add an extra scanner.nextLine() after nextInt()** to consume the leftover newline before reading the actual string input.

java

CopyEdit

import java.util.Scanner;

public class ScannerFix {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your age: ");

int age = scanner.nextInt();

scanner.nextLine(); // Consume leftover newline

System.out.print("Enter your name: ");

String name = scanner.nextLine(); // Now it properly reads user input

System.out.println("Your age: " + age);

System.out.println("Your name: " + name);

scanner.close();

}

}

**Corrected Output:**

yaml

CopyEdit

Enter your age: 24

Enter your name: Lahari

Your age: 24

Your name: Lahari

**✔ Fix Summary:**

* Always **consume the newline** (\n) **before calling nextLine()** after nextInt(), nextDouble(), etc.
* Use:

java

CopyEdit

scanner.nextLine(); // Consume newline

**2. Closing Scanner Too Early**

**Why is this a problem?**

* Once a Scanner object is **closed**, it **cannot be reopened** to read input from System.in again in the same program.
* If you try using scanner.nextInt(), scanner.nextLine(), etc., **after closing Scanner**, the program will throw a runtime error:

pgsql

CopyEdit

java.util.NoSuchElementException: No line found

**Example: Closing Scanner Too Early**

java

CopyEdit

import java.util.Scanner;

public class ScannerCloseIssue {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your name: ");

String name = scanner.nextLine();

scanner.close(); // Scanner is closed here

System.out.print("Enter your age: ");

int age = scanner.nextInt(); // ❌ Error! Scanner is already closed

System.out.println("Your name: " + name);

System.out.println("Your age: " + age);

}

}

**Error Output:**

pgsql

CopyEdit

Enter your name: Lahari

Exception in thread "main" java.util.NoSuchElementException: No line found

at java.base/java.util.Scanner.nextInt(Scanner.java:2264)

**Why does this happen?**

* The Scanner is **closed after reading the name**, so when nextInt() is called, it cannot read further input.

**Fix: Close the Scanner at the End of the Program**

To avoid this, **only close the Scanner after all user input is taken**.

java

CopyEdit

import java.util.Scanner;

public class ScannerFixClose {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your name: ");

String name = scanner.nextLine();

System.out.print("Enter your age: ");

int age = scanner.nextInt(); // No issue now

System.out.println("Your name: " + name);

System.out.println("Your age: " + age);

scanner.close(); // Close at the very end

}

}

**Correct Output:**

yaml

CopyEdit

Enter your name: Lahari

Enter your age: 24

Your name: Lahari

Your age: 24

**✔ Fix Summary:**

* **Close Scanner only at the end** of the program.
* If Scanner is closed **too early**, further input operations will fail.

**Final Best Practices for Using Scanner**

| **Problem** | **Cause** | **Fix** |
| --- | --- | --- |
| nextLine() gets skipped after nextInt() | nextInt() leaves \n in buffer | Add scanner.nextLine(); before calling nextLine() |
| Scanner closed too early | No further input can be read | Close Scanner only at the end |

**ADVANCED TOPICS IN JAVA**

**ArrayList:**

ArrayList is found in util package in java.

https://www.w3schools.com/java/java\_arraylist.asp

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> obj=new ArrayList<String>();

obj.add("one");

obj.add("two");

obj.add("three");

obj.add("four");

obj.add(0,"zero");

obj.add(0,"zeero"); // it just adds zeero in first place but donot replace already existing pne

System.out.println(obj.get(0));

obj.set(0,"variable"); //in first place but replace already existing one

System.out.println(obj.get(0));

System.out.println(obj);

System.out.println(obj.size());

for(int i=0;i<obj.size();i++)

{

System.out.println(obj.get(i));

}

for(String i: obj)

{

System.out.println(i);

}

obj.sort();

System.out.println(obj);

obj.clear();

System.out.println(obj);

//looping

//for each for(string i: arrayname){}

//

}

}

**Reference methods:** [**https://www.w3schools.com/java/java\_ref\_arraylist.asp**](https://www.w3schools.com/java/java_ref_arraylist.asp)

**Array copy methods:**

* 1. **Manual copy by for loop**
  2. **Copying using the System.arraycopy() method which helps in copying one array to another array**
  3. **Arrays.copyOf() (Creates a new array with a new size)**
  4. **Using clone() (Creates a duplicate of the array)**

## **What is Copying an Array?**

In Java, arrays have a **fixed size**, so if you need to modify an array (like adding or removing elements), you often have to **copy elements into a new array**.

There are **4 main ways** to copy an array in Java:

1. **Using a loop** (Manual Copy)
2. **Using System.arraycopy()** (Efficient and built-in)
3. **Using Arrays.copyOf()** (Creates a new array with a new size)
4. **Using clone()** (Creates a duplicate of the array)

## **1️⃣ Copying an Array Using a Loop (Manual Copy)**

The simplest way to copy an array is using a loop.

java

CopyEdit

int[] original = {1, 2, 3, 4};

int[] copy = new int[original.length];

for (int i = 0; i < original.length; i++) {

copy[i] = original[i]; // Copying elements one by one

}

System.out.println(Arrays.toString(copy)); // Output: [1, 2, 3, 4]

✔ **Simple but slower for large arrays**

## **2️⃣ Using** System.arraycopy() **(Efficient Method)**

This method is faster than a loop because it copies data **at the memory level**.

java

CopyEdit

int[] original = {1, 2, 3, 4};

int[] copy = new int[original.length];

System.arraycopy(original, 0, copy, 0, original.length);

System.out.println(Arrays.toString(copy)); // Output: [1, 2, 3, 4]

### **📌 Understanding** System.arraycopy()

java

CopyEdit

System.arraycopy(sourceArray, srcPos, destinationArray, destPos, length);

| **Parameter** | **Meaning** |
| --- | --- |
| sourceArray | The original array you want to copy from |
| srcPos | The **starting index** in the source array |
| destinationArray | The array you want to copy into |
| destPos | The **starting index** in the destination array |
| length | Number of elements to copy |

### **📌 Example: Copying a Part of an Array**

java

CopyEdit

int[] numbers = {10, 20, 30, 40, 50};

int[] partialCopy = new int[3];

System.arraycopy(numbers, 1, partialCopy, 0, 3);

System.out.println(Arrays.toString(partialCopy)); // Output: [20, 30, 40]

✔ **Fast and efficient for large arrays**

## **3️⃣ Using** Arrays.copyOf()

This method creates a **new array with the specified length** and copies elements.

java

CopyEdit

int[] original = {1, 2, 3, 4};

int[] copy = Arrays.copyOf(original, original.length);

System.out.println(Arrays.toString(copy)); // Output: [1, 2, 3, 4]

✔ **Useful when resizing an array**  
🔹 **Example: Expanding an Array**

java

CopyEdit

int[] expanded = Arrays.copyOf(original, 6);

System.out.println(Arrays.toString(expanded)); // Output: [1, 2, 3, 4, 0, 0]

* If the **new size is larger**, Java fills extra spaces with **default values (0 for int, null for objects).**

## **4️⃣ Using** clone() **(Duplicates the Array)**

This method creates an **exact duplicate** of an array.

java

CopyEdit

int[] original = {1, 2, 3, 4};

int[] clone = original.clone();

System.out.println(Arrays.toString(clone)); // Output: [1, 2, 3, 4]

✔ **Simple and easy**  
❌ **Does not work well with multi-dimensional arrays (deep copy needed)**

## **🔹 When to Use Each Method?**

| **Method** | **Best For** | **Performance** |
| --- | --- | --- |
| **Loop (for)** | Small arrays, learning purpose | **Slowest** |
| **System.arraycopy()** | Large arrays, performance-critical tasks | **Fast** |
| **Arrays.copyOf()** | Resizing arrays, making copies easily | **Medium** |
| **clone()** | Making a quick duplicate | **Fast** (but limited for deep copying) |
| In Java, you cannot directly print an array using System.out.println(arr1), as it will print the memory reference instead of the elements. Here are several ways to properly print an array:  **1. Using Arrays.toString()**  This is the easiest and most common method:  java  CopyEdit  import java.util.Arrays;  public class Main {  public static void main(String[] args) {  int[] arr1 = {1, 2, 3, 4, 5};  System.out.println(Arrays.toString(arr1));  }  }  **Output:**  csharp  CopyEdit  [1, 2, 3, 4, 5]  **2. Using a for loop**  java  CopyEdit  public class Main {  public static void main(String[] args) {  int[] arr1 = {1, 2, 3, 4, 5};  for (int i = 0; i < arr1.length; i++) {  System.out.print(arr1[i] + " ");  }  }  }  **Output:**  CopyEdit  1 2 3 4 5  **3. Using an Enhanced for Loop (foreach)**  java  CopyEdit  public class Main {  public static void main(String[] args) {  int[] arr1 = {1, 2, 3, 4, 5};  for (int num : arr1) {  System.out.print(num + " ");  }  }  }  **Output:**  CopyEdit  1 2 3 4 5  **4. Using Streams (Java 8+)**  java  CopyEdit  import java.util.Arrays;  public class Main {  public static void main(String[] args) {  int[] arr1 = {1, 2, 3, 4, 5};  Arrays.stream(arr1).forEach(num -> System.out.print(num + " "));  }  }  **Output:**  CopyEdit  1 2 3 4 5 |  |  |

import java.util.\*;

public class Main {

public static void main(String[] args) {

int arr1[]={10,20,30,40,50};

Scanner sc=new Scanner(System.in);

System.out.println("enter the element you wanna update");

int element=sc.nextInt();

System.out.println("enter the place you wanna update");

int place=sc.nextInt();

System.out.println(Arrays.toString(arr1)+" "+element+" "+place);

int[] newarr= new int[arr1.length+1]; //created a new array with one extra size

//copying ONLY the first two elements of array into arr1 as follows

//using copy array because i know how to use for loop

System.arraycopy(arr1,0,newarr,0,place);

newarr[2]=25;

System.arraycopy(arr1,place,newarr, place+1,arr1.length - place);

System.out.println("Updated Array: " + Arrays.toString(newarr));

}

**}**

**Exception Handling in Java:**

**Exception handling is a mechanism to handle runtime errors, allowing a program to continue execution even after an exception occurs. In Java, exception handling is accomplished using a set of keywords: try, catch, throw, throws, and finally.**

**Exceptions are runtime errors that occur during the execution of a program.**

### **Types of Exceptions:**

| **Exception Type** | **Description** | **Example** |
| --- | --- | --- |
| **Checked Exception** | Checked at compile time | File not found, network issues |
| **Unchecked Exception** | Occurs at runtime (program crashes if not handled) | Division by zero, accessing invalid index |

## **1️⃣ try-catch Block**

The try block contains code that might throw an exception. The catch block catches and handles the exception.

### **Example: Handling Division by Zero**

java

CopyEdit

public class TryCatchExample {

public static void main(String[] args) {

try {

int result = 10 / 0; // ❌ This will cause ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Error: Cannot divide by zero!");

}

System.out.println("Program continues...");

}

}

### **🔍 Explanation**

* The try block attempts to divide 10 by 0, which causes an ArithmeticException.
* The catch block catches the exception and prints:  
  **"Error: Cannot divide by zero!"**
* The program does not crash and continues execution.

### **💡 Output**

vbnet

CopyEdit

Error: Cannot divide by zero!

Program continues...

## **Where is IllegalArgumentException Used in Real Life?**

## Think of an **ATM withdrawal system**:

## If you **try to withdraw negative money**, it's **invalid**.

## The system should **throw an exception**.

## **2️⃣ throw Keyword**

The throw keyword is used to **manually throw an exception** in Java.

### **Example: Preventing Negative Balance Withdrawals**

java

CopyEdit

public class ThrowExample {

public static void withdraw(double amount) {

if (amount < 0) {

throw new IllegalArgumentException("Withdrawal amount cannot be negative: " + amount);

}

System.out.println("Successfully withdrawn: $" + amount);

}

public static void main(String[] args) {

try {

withdraw(-500); // ❌ Throws exception

} catch (IllegalArgumentException e) {

System.out.println("Caught exception: " + e.getMessage());

}

withdraw(1000); // ✅ Valid transaction

}

}

### **🔍 Explanation**

* If amount < 0, the method **throws an IllegalArgumentException**.
* The catch block catches and handles the error.
* The second call withdraw(1000) runs successfully.

### **💡 Output**

yaml

CopyEdit

Caught exception: Withdrawal amount cannot be negative: -500

Successfully withdrawn: $1000

**What is throws in Java?**

The throws keyword **declares** that a method **might cause an exception**.

**✅ Example: Declaring an Exception**

java

CopyEdit

import java.io.\*;

public class Example {

public static void readFile() throws IOException {

throw new IOException("File not found!"); // Simulating file error

}

public static void main(String[] args) {

try {

readFile(); // ❌ This might throw an exception

} catch (IOException e) {

System.out.println("Caught Exception: " + e.getMessage());

}

}

}

💡 **Output**

pgsql

CopyEdit

Caught Exception: File not found!

☝ throws IOException means this method **may cause an error**, and we must **handle** it.