Syllabus

This syllabus provides a comprehensive breakdown of core and advanced Java concepts, along with resources for further exploration.

**I. Introduction to Java (1-2 weeks)**

* **History and Features of Java:**
  + Overview of Java's creation and evolution
  + Key features like platform independence, object-oriented nature, security
* **Setting up the Development Environment (IDE):**
  + Choosing an IDE (Integrated Development Environment) like Eclipse, IntelliJ IDEA
  + Installing and configuring the IDE
  + Understanding project creation, navigation, and debugging tools
* **Understanding Java Program Structure (1 week):**
  + Basic program structure (class, methods, main method)
  + Source files (.java) and compilation process
  + Running Java programs (command line vs IDE execution)

**Resources:**

* TutorialsPoint - Java Tutorial [invalid URL removed]
* Oracle Java Documentation - Getting Started [invalid URL removed]
* Online IDEs - JDoodle [invalid URL removed]

**II. Java Fundamentals (2-3 weeks)**

**A. Basic Syntax (1 week):**

* **Variables:**
  + Data types (primitive - int, double, char, etc.; reference - String, Object)
  + Declaring and initializing variables
* **Operators:**
  + Arithmetic operators (+, -, \*, /)
  + Logical operators (&&, ||, !)
  + Assignment operators (=, +=, -=)
  + Other operators (increment/decrement, relational, etc.)
* **Expressions and Statements:**
  + Combining operators and variables to form expressions
  + Using statements like if-else, for loops, to control program flow

**B. Control Flow (1 week):**

* **Conditional Statements:**
  + if-else statements for decision making
  + switch statements for multi-way branching
* **Looping Statements:**
  + for loops for repetitive execution with counter
  + while loops for indefinite execution based on condition
  + do-while loops for guaranteed execution at least once

**C. Functions (Methods) (1 week):**

* **Defining and Calling Methods:**
  + Creating reusable blocks of code
  + Passing arguments (parameters) to methods
* **Parameter Passing:**
  + By value (primitive data types) vs by reference (objects)
* **Method Overloading:**
  + Having multiple methods with the same name but different parameter lists
* **Recursion:**
  + A function calling itself

**Resources:**

* W3Schools - [Java Operators](https://www.w3schools.com/java/java_operators.asp)
* Javapoint - Java Control Flow Statements [invalid URL removed]
* GeeksforGeeks - [Java Methods](https://www.geeksforgeeks.org/methods-in-java/)

**III. Object-Oriented Programming (OOP) Concepts (2-3 weeks)**

**A. Classes and Objects (1 week):**

* **Defining Classes:**
  + Blueprints for creating objects
  + Specifying attributes (variables) and methods (functions)
* **Creating Objects:**
  + Instantiating classes to create objects
  + Accessing object members (attributes and methods)

**B. Inheritance (1 week):**

* **Subclasses and Superclasses:**
  + Creating hierarchical relationships between classes
  + Code reusability through inheritance
* **Polymorphism:**
  + Overriding methods in subclasses for different behavior
  + Method overriding vs overloading

**C. Abstraction (0.5-1 week):**

* **Abstract Classes and Interfaces:**
  + Defining abstract classes with unimplemented methods
  + Interfaces for specifying what a class must do (without implementation)

**D. Packages (0.5-1 week):**

* **Organizing Classes:**
  + Grouping related classes into packages for better structure and reusability

**Resources:**

* Oracle Java Documentation - Object-Oriented Programming [invalid URL removed]
* Baeldung - [Java Inheritance](https://www.baeldung.com/java-inheritance)
* Programiz - Java Abstraction [invalid URL removed]

**IV. Advanced Java Topics (3-4 weeks)**

**A. Exception Handling (1 week):**

* **Try-Catch Blocks:**
  + Handling errors and exceptions that occur during program execution

--------------------------------------------- ----------------------------------

Introduction to programming

Programming, at its core, is the process of giving a computer a set of instructions that it can understand and execute. These instructions, called code, are written in a specific language that the computer can interpret. Imagine it as giving a detailed recipe to a kitchen appliance that can follow every step precisely.

**Languages of Instruction**

* Just like humans have different languages, computers have their own languages for programming. These languages are designed with specific syntax (rules for structuring the code) and semantics (meaning attached to the code).
* Popular programming languages include Python, Java, C++, JavaScript, and many more. Each language has its strengths and weaknesses, making it suitable for different tasks.

Giving Instructions (Coding)

* Programming involves writing code that tells the computer what to do step-by-step. This code can be as simple as performing calculations or as complex as building entire applications.
* The code typically involves:
  + **Variables**: Storing data like numbers, text, or even collections of data.
  + **Operators**: Performing operations on data (addition, comparison, etc.).
  + **Control Flow Statements**: Deciding which instructions to execute based on conditions (if-else statements, loops).
  + **Functions/Methods**: Reusable blocks of code that perform specific tasks.

Translating to Machine Code

* While programmers write code in a human-readable language, computers can only understand machine code, a series of 0s and 1s.
* A translator called a compiler or interpreter converts the programmer's code into machine code that the computer can execute.

Problem-Solving and Logic

* Programming is a powerful tool for solving problems. It requires breaking down complex tasks into smaller, logical steps that the computer can understand.
* Programmers need to think critically, analyze problems, and design algorithms (step-by-step solutions) that can be translated into code.

Beyond the Basics

* As you progress in programming, you'll encounter more advanced concepts like object-oriented programming (OOP), data structures, algorithms, and software development methodologies.
* OOP helps create reusable and modular code, while data structures organize information efficiently, and algorithms provide efficient solutions to problems. Software development methodologies establish best practices for building large software systems.

Examples of Programming in Action

* The websites you visit, the apps you use on your phone, the games you play - all rely on programming to function.
* Programmers create software for various purposes, including:
  + Developing web applications
  + Building mobile apps
  + Creating video games
  + Automating tasks
  + Data analysis and machine learning

**Programming Paradigm**

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfill each and every demand.

Types of programming paradigm

1. Imperative Programming Paradigm
2. Declarative Programming Paradigm
3. Object-Oriented programming paradigm
4. Functional programming Paradigm
5. Logic Programming Paradigm
6. Reactive Programming Paradigm

note: Here we explained most popular programming paradigm only.

Imperative Programming Paradigm

Imagine you're giving instructions to a friend on how to bake a cake. Imperative programming works similarly. It's all about giving the computer a set of explicit instructions, step-by-step, to achieve a specific outcome.

**Key Concepts**

* **Focus on State** Programs manipulate data (variables) and change their state (values) throughout execution.
* **Control Flow Statements** if-else, for, while statements are used to control the flow of execution based on conditions and loops.
* **Procedures/Functions** Reusable blocks of code that perform specific tasks, often taking input and returning output.

**Python Example**

def bake\_cake(ingredients, temperature, time):

# Mix ingredients (changing state of ingredients variable)

mixed\_batter = mix(ingredients)

# Put batter in oven (changing state of batter location)

oven.bake(mixed\_batter, temperature, time)

# Take cake out (changing state of cake location)

return cake

cake = bake\_cake(["flour", "sugar", "eggs"], 350, 30)

**Strengths**

* Simple and intuitive for beginners, easy to reason about program flow.
* Efficient for low-level tasks and hardware manipulation.

**Weaknesses**

* Code can become verbose and difficult to maintain for complex problems.
* Error handling can be cumbersome.

Declarative Programming Paradigm

Instead of giving step-by-step instructions, declarative programming focuses on what you want the program to achieve. It's like describing the cake you desire, and the computer figures out the recipe and baking process.

**Key Concepts**

* **Focus on What** Programs specify the desired outcome or goal without detailing the "how."
* **Declarative Statements** High-level statements define the properties or relationships between data.
* **Less Control Flow** Focuses on the result rather than the specific execution steps.

SQL Example

SELECT \* FROM cakes WHERE temperature > 300 AND time > 25;

This query declares the desired outcome (selecting cakes) based on specific conditions (temperature and time) without specifying how to find them in the database.

**Strengths**

* Code can be concise and easier to maintain for complex problems.
* Focus on desired outcome improves readability.

**Weaknesses**

* May be less intuitive for beginners, requires a different way of thinking about problems.
* Limited control over specific execution steps.

Object-Oriented Programming (OOP)

OOP takes inspiration from the real world. It organizes code around objects, which are self-contained entities that encapsulate data (attributes) and related behaviors (methods). Imagine a cake object that has attributes like ingredients, temperature, and methods like bake() and frost().

**Key Concepts**

* **Objects:** Combine data and functionality, representing real-world entities.
* **Classes:** Blueprints for creating objects, defining attributes and methods.
* **Inheritance:** Allows creating new classes (subclasses) that inherit properties and behaviors from existing classes (superclasses).
* **Encapsulation:** Protects data integrity by restricting direct access to attributes.

**Example (Java - OOP):**

Java

class Cake {

private String ingredients; // Encapsulated attribute

private int temperature;

public void bake(int time) {

// Baking logic using ingredients and temperature

}

}

class ChocolateCake extends Cake { // Inheritance

public void frost() {

// Frosting logic specific to chocolate cake

}

}

**Strengths:**

* Promotes modularity, reusability, and code maintainability.
* Makes code more organized and easier to understand for complex systems.

**Weaknesses:**

* Can be more complex to learn and design compared to imperative programming.
* Might be overkill for simple tasks.

Functional Programming:

Here, programs are built around the concept of pure functions. These functions take input and always return the same output for that given input, without causing side effects (changes to global state). It's like following a precise recipe every time to ensure consistent cake results.

**Key Concepts:**

* **Pure Functions:** No side effects, always return the same output for a given input.
* **Immutability:** Data is treated as immutable (unchanging) after creation, leading to new data structures for modifications.
* **Recursion:** Functions can call themselves, breaking down problems into smaller subproblems until a base case is reached.

**Example (Haskell - Functional):**

mixIngredients :: [String] -> [String]

mixIngredients ingredients = concat [map toUpper ingredient | ingredient <- ingredients]

bakeCake :: Int -> Int -> [String] -> [String]

bakeCake temperature time ingredients = mixIngredients ingredients ++ ["Baked for", show time, "minutes at", show temperature, "degrees"]

This code defines pure functions for mixing ingredients and baking a cake, avoiding side effects and emphasizing immutability.

**Strengths:**

* Encourages code that is predictable, easier to test and reason about.
* Well-suited for parallel processing and concurrency.

**Weaknesses:**

* Can be less intuitive for beginners due to its emphasis on immutability and recursion.
* May not be the most efficient choice for all types of problems.

Choosing the Right Paradigm:

The best paradigm for a project depends on the specific problem you're trying to solve. Here are some general guidelines:

* **Imperative:** Well-suited for low-level tasks, hardware interaction, and tasks requiring precise control flow.
* **Declarative:** Ideal for complex data manipulation, configuration management, and querying databases.
* **OOP:** Effective for modeling real-world entities, building large-scale applications, and promoting code reusability.
* **Functional:** Excellent for data analysis, parallel processing, and creating predictable and reliable code.
* **Other Paradigms:** Apply to specific problem domains and can be used in conjunction with other paradigms.

**Java Introduction**

Java is a programming language and computing platform first released by Sun Microsystems in 1995. It has evolved from humble beginnings to power a large share of today’s digital world, by providing the reliable platform upon which many services and applications are built. New, innovative products and digital services designed for the future continue to rely on Java, as well.

While most modern Java applications combine the Java runtime and application together, there are still many applications and even some websites that will not function unless you have a desktop Java installed. Java.com, this website, is intended for consumers who may still require Java for their desktop applications – specifically applications targeting Java 8. Developers as well as users that would like to learn Java programming should visit the [dev.java](https://dev.java/) website instead and business users should visit [oracle.com/java](https://www.oracle.com/java/) for more information.

**Inventor Of Java**

A person with a beard and glasses

Description automatically generatedJames Gosling, a Canadian computer scientist, invented Java. In 1991, while working at Sun Microsystems, he started the Java project. The project's initial goal was to create a platform-independent language for interactive television, but the digital cable television industry at the time was not yet ready for it.

Gosling and his team created a new virtual machine and a new programming language, which they called Java. Java was designed to be portable, secure, and easy to use. It quickly became popular for developing web applications, and it is now one of the most widely used programming languages in the world.

Gosling has received numerous awards for his work on Java, including the ACM Software Systems Award and the IEEE John von Neumann Medal. He is a member of the National Academy of Engineering and the Royal Society of Canada.

**1. The Birth of Java (1991-1995):**

* **Green Team Origins:** Created by James Gosling, Bill Joy, Mike Sheridan, and Guy Steele at Sun Microsystems (later acquired by Oracle).
* Project "**Oak**": The initial name, inspired by an oak tree outside Gosling's office.
* Focus on Embedded Systems: Designed for consumer electronics like set-top boxes, targeting real-time performance and a small memory footprint.

**2. The Rise of the Web (1995-2000):**

* **A Web-Ready Language**: Netscape's interest in developing interactive web content led to Java's adaptation for the web.
* **Applets and Security**: Applets, small Java programs embedded in web pages, provided dynamic content. The Java Virtual Machine (JVM) ensured platform independence and security by sandboxing applets.
* **Java 1.0 Released (1996)**: The official public release marked the beginning of widespread adoption.

**3. Enterprise Boom and Beyond (2000-2010):**

* **J2EE (Java 2 Platform, Enterprise Edition**): A suite of APIs and specifications for building large-scale enterprise applications.
* **Open Source Movement**: The release of the Java Development Kit (JDK) as open-source software in 2006 fostered community development and innovation.
* **Mobile Java (J2ME):** An attempt to establish Java in the mobile market, facing competition from native development tools.

**4. The Modern Java Landscape (2010-Present):**

* **Rise of Android (2008):** Java became the primary programming language for Android app development, significantly boosting its popularity in mobile computing.
* **Java 7 and 8 (2011-2014):** Introduced significant improvements like closures, lambda expressions, and the Fork/Join framework for parallel processing.
* **Focus on Cloud and Microservices**: Modern Java frameworks like Spring Boot and technologies like Docker containers cater to cloud-based development and microservices architectures.
* **Java 11 and Beyond (2018-Present):** Modularization, improved garbage collection, and features like long support releases (LTS) enhance developer experience and long-term application stability.

**Why use JAVA**

* Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.).
* It is one of the most popular programming languages in the world.
* It has a large demand in the current job market.
* It is easy to learn and simple to use.
* It is open-source and free.
* It is secure, fast and powerful.
* It has huge community support (tens of millions of developers).
* Java is an object-oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs.
* As Java is close to [C++](https://www.w3schools.com/cpp/default.asp) and [C#](https://www.w3schools.com/cs/default.asp), it makes it easy for programmers to switch to Java or vice versa.

**Features of Java**

Java's success stems from its well-designed features that promote code quality, efficiency, and maintainability. Here's a breakdown of key Java features:

**1. Platform Independence ("Write Once, Run Anywhere" - WORA):**

* Java code is compiled into bytecode, an intermediate representation that can run on any platform with a Java Virtual Machine (JVM).
* The JVM interprets the bytecode and translates it into machine code specific to the underlying system.
* This feature allows developers to write code once and deploy it across various operating systems (Windows, Linux, macOS, etc.) without modifications.

**2. Object-Oriented Programming (OOP):**

* Java is an object-oriented language, meaning programs are built around objects.
* An object encapsulates data (attributes) and related behaviors (methods).
* OOP principles like inheritance, polymorphism, and encapsulation promote code reusability, modularity, and maintainability.

**3. Garbage Collection:**

* Java includes automatic garbage collection that manages memory allocation and deallocation.
* This feature reduces the risk of memory leaks and simplifies memory management for developers.

**4. Strong Typing:**

* Java is a statically typed language, meaning variables must be declared with a specific data type (e.g., int, double, String).
* This helps catch potential errors during compilation, improving code reliability.

**5. Exception Handling:**

* Java provides mechanisms for handling errors and exceptions that may occur during program execution.
* Using try-catch blocks, developers can gracefully handle exceptions and prevent program crashes.

**6. Secure:**

* Java's design prioritizes security by incorporating features like:
* Absence of explicit pointers (reducing vulnerabilities like buffer overflows)
* Secure runtime environment (JVM sandboxing)
* Strong typing (reducing type conversion errors)
* These features make Java a popular choice for developing secure applications.

**7. Rich Set of Libraries and APIs (Application Programming Interfaces):**

* Java provides a vast library of pre-written code for various functionalities.
* The Java API offers classes for common tasks like networking, file I/O, GUI development, and database interactions.
* Developers can leverage these libraries to streamline development and avoid reinventing the wheel.

**8. Multithreading:**

* Java allows creating multiple threads of execution within a program.
* This enables concurrent processing, improving performance for tasks that can benefit from parallelism.

**9. Rich Development Tools and IDEs (Integrated Development Environments):**

* A wide range of development tools and IDEs are available for Java, providing features like code completion, syntax highlighting, debugging, and project management.
* These tools enhance developer productivity and improve the overall development experience.

**10. Large and Active Community:**

* Java boasts a large and active community of developers worldwide.
* This community provides extensive online resources, tutorials, forums, and libraries, making it easier for beginners to learn and experienced developers to find solutions.

**Beyond the Basics:**

As you delve deeper into Java, you'll encounter even more advanced features:

* **Annotations**: Provide metadata for code, used for reflection and configuration.
* **Generics**: Enable creating type-safe collections and methods that can work with various data types.
* **Lambda Expressions (Java 8+):** Provide concise syntax for defining anonymous functions, promoting functional programming style.
* **Java Modules (Java 9+):** Modularize large codebases, improving maintainability and reducing classpath conflicts.

**Modern programming vs Non-Modern Programming**

The world of programming is constantly evolving, with new languages, frameworks, and practices emerging all the time. Here's a breakdown of the key differences between modern and non-modern programming approaches:

**Paradigms and Languages:**

**Modern:** Leans towards high-level languages that focus on readability, maintainability, and developer productivity. Examples include Python, Java, JavaScript (with frameworks), C#, and Go. These languages often embrace paradigms like object-oriented programming (OOP) and functional programming, promoting code reusability and modularity.

**Non-Modern:** Often relies on lower-level languages closer to the machine's instruction set, like C, C++, and Assembly. These languages offer more control over hardware resources but require a deeper understanding of computer architecture and can be more error-prone.

**Development Tools and Practices:**

**Modern**: Emphasizes integrated development environments (IDEs) that provide features like code completion, syntax highlighting, debugging tools, and version control integration. This makes development faster and less error-prone. Version control systems like Git allow for collaboration and easy tracking of code changes. Unit testing frameworks are widely used to ensure code quality and functionality.

**Non-Modern**: May use simpler text editors for code creation. Debuggers might be more basic, and version control might be manual or involve cumbersome systems. Unit testing might be less emphasized.

**Focus and Applications:**

**Modern**: Often targets higher-level functionalities like web development, mobile app development, data science, machine learning, and cloud computing. Frameworks and libraries provide pre-built components and functionalities, allowing developers to focus on core logic.

**Non-Modern:** Concentrates more on system programming, embedded systems, device drivers, and performance-critical applications. Developers have more control over hardware interactions but need to handle low-level details.

**Security:**

**Modern**: Languages often have built-in features for memory management and type safety, reducing the risk of security vulnerabilities like buffer overflows that can be common in non-modern languages.

**Non-Modern:** Security is more dependent on the programmer's expertise in handling memory allocation and data types.

It's important to note that these are generalizations. There's still a place for non-modern languages in specific scenarios where performance or hardware control is critical. Modern languages are constantly incorporating features and paradigms from non-modern approaches for better optimization when needed.

**Here's an analogy:**

Imagine building a house. Non-modern programming is like using bricks and mortar directly, giving you fine-grained control but requiring a lot of effort. Modern programming provides pre-fabricated walls and pre-built plumbing systems, making construction faster and easier while still allowing customization.

**Role Of Java in Front – End and Back – End**

**Back-End Development**

Java shines in back-end development, where it excels at creating robust, scalable, and secure server-side applications. Here's a breakdown of the key technologies:

**Java Platform, Standard Edition (Java SE):** The foundation for back-end development with Java. It provides core libraries for I/O operations, networking, database interaction, multithreading, and more.

**Java Servlet API:** A cornerstone of back-end Java development, enabling creation of web servlets that handle HTTP requests and responses. Servlets interact with databases, session management, and other back-end services.

**JavaServer Pages (JSP):** A templating language that simplifies servlet development by combining HTML, Java code snippets (called scriptlets), and expressions within a single file. This allows for dynamic content generation based on user input or server-side data.

**Java Frameworks:** Streamline back-end development by providing pre-built components and patterns. Popular choices include:

**Spring Framework:** A comprehensive, modular framework for building enterprise-grade applications, offering features like dependency injection, security, and data access.

**Jakarta EE (formerly Java EE):** A collection of specifications for developing web services, enterprise applications, and distributed systems. Includes technologies like Java Persistence API (JPA) for database interaction, Java API for RESTful Web Services (JAX-RS) for building REST APIs, and Java Message Service (JMS) for asynchronous communication.

**Databases:** Relational (MySQL, PostgreSQL, Oracle) and NoSQL (MongoDB, Cassandra) databases can be accessed from Java using libraries like JDBC (Java Database Connectivity).

**Message Queues:** Frameworks like Apache ActiveMQ or RabbitMQ enable asynchronous communication between applications.

**Cloud Platforms:** Java applications can be deployed on cloud platforms like AWS, Azure, or GCP, leveraging their scalability, security, and other services.

**Front-End Development (Limited Role)**

While Java isn't traditionally considered a front-end language, there are some niche use cases where it can be involved:

**Java Applets (Deprecated):** In the past, Java applets could be embedded in web pages to provide interactive elements. However, security concerns and the rise of more mature front-end technologies like JavaScript have led to their decline.

**GWT (Google Web Toolkit):** A framework that allows developers to write front-end code in Java and generate JavaScript code for better performance or leveraging existing Java skills. While GWT is still usable, it's not as widely adopted due to the popularity of modern JavaScript frameworks.

**Key Considerations**

**Front-End vs. Back-End Focus:** Java's strength lies primarily in back-end development. If your primary focus is front-end (user interface, interactivity), languages like JavaScript (with frameworks like React, Angular, or Vue.js) are more suitable.

**Full-Stack Development:** Java can be part of a full-stack developer's toolset, allowing them to work on both back-end and some niche front-end use cases.

**Project Requirements:** The choice of technologies ultimately depends on your project's specific requirements. Java remains a solid choice for robust back-end development, while other languages and frameworks might be more appropriate for front-end tasks.

*In conclusion, Java is a powerful back-end development language, and while it has some limited front-end applications, it's generally not the primary choice for front-end work. Understanding these distinctions will guide you in selecting the most suitable tools for your project.*

Procedural Programming V.S Object-oriented programming

|  |  |  |
| --- | --- | --- |
| **On the basis of** | **Procedural Programming** | **Object-oriented programming** |
| **Definition** | It is a programming language that is derived from structure programming and based upon the concept of calling procedures. It follows a step-by-step approach to break down a task into a set of variables and routines via a sequence of instructions. | Object-oriented programming is a computer programming design philosophy or methodology that organizes/ models software design around data or objects rather than functions and logic. |
| **Security** | It is less secure than OOPs. | Data hiding is possible in object-oriented programming due to abstraction. So, it is more secure than procedural programming. |
| **Approach** | It follows a top-down approach. | It follows a bottom-up approach. |
| **Data movement** | In procedural programming, data moves freely within the system from one function to another. | In OOP, objects can move and communicate with each other via member functions. |
| **Orientation** | It is structure/procedure-oriented. | It is object-oriented. |
| **Access modifiers** | There are no access modifiers in procedural programming. | The access modifiers in OOP are named private, public, and protected. |
| **Inheritance** | Procedural programming does not have the concept of inheritance. | There is a feature of inheritance in object-oriented programming. |
| **Code reusability** | There is no code reusability present in procedural programming. | It offers code reusability by using the feature of inheritance. |
| **Overloading** | Overloading is not possible in procedural programming. | In OOP, there is a concept of function overloading and operator overloading. |
| **Importance** | It gives importance to functions over data. | It gives importance to data over functions. |
| **Virtual class** | In procedural programming, there are no virtual classes. | In OOP, there is an appearance of virtual classes in inheritance. |
| **Complex problems** | It is not appropriate for complex problems. | It is appropriate for complex problems. |
| **Data hiding** | There is no proper way for data hiding. | There is a possibility of data hiding. |
| **Program division** | In Procedural programming, a program is divided into small programs that are referred to as functions. | In OOP, a program is divided into small parts that are referred to as objects. |
| **Examples** | Examples of Procedural programming include C, Fortran, Pascal, and VB. | Examples of object-oriented programming are - .NET, C#, Python, Java, VB.NET, and C++. |

Introduction Of Eclipse (IDE)

note : Please double click on the Icon to get the file “Introduction to Eclipse”.



Unit 2

Language Constructors, Classes And Objects

Topics of this unit

1. Variables, types, and type Declaration
2. Types of Datatypes
3. Types of Operators
4. Conditional Statements
5. Performed Input and Output using Scanner Class (CLI)
6. Looping Statements
7. Switch Case
8. Arrays
9. Method Creation
10. Accessing Class Members

Variable, Types and Type Declaration

Variable

In Java, Variables are the data containers that save the data values during Java program execution. Every Variable in Java is assigned a data type that designates the type and quantity of value it can hold. A variable is a memory location name for the data.

1. In Java variable is a name given to a memory location.
2. It is the basic unit of storage in a program.
3. The value stored in a variable can be changed during program execution.
4. Variables in Java are only a name given to a memory location.
5. All the operations done on the variable affect that memory location.
6. In Java, all variables must be declared before use.

Declaring a variable

Syntax

datatype variableName;

Example

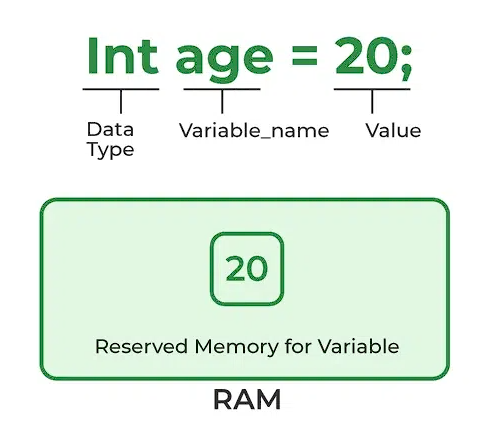
int number;

// Here number is a variable of int type which can only be stored integer number only.

Variable Initialization

In this way, a name can only be given to a memory location. It can be assigned values in two ways:

* Variable Initialization
* Assigning value by taking input



Initialization of a Variable

* **datatype**: Type of data that can be stored in this variable.
* **variable\_** Name given to the variable.
* **value**: It is the initial value stored in the variable.

Types of Variable in Java

1. Local Variables
2. Instance Variables
3. Static Variables
4. **Local Variables**

* A variable defined within a block or method or constructor is called a local variable.
* These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
* The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
* Initialization of the local variable is mandatory before using it in the defined scope.

Example

class DemoLocal {

public static void main(String[] args)

{

// Declared a Local Variable

int var = 10;

// This variable is local to this main method only

System.out.println("Local Variable: " + var);

}

}

1. Instance Variable

* Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.
* As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
* Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
* Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.
* Instance variables can be accessed only by creating objects.
* We initialize instance variables using [constructors](https://www.geeksforgeeks.org/constructors-in-java/) while creating an object. We can also use [instance blocks](https://www.geeksforgeeks.org/using-instance-blocks-in-java/) to initialize the instance variables.

Example

package in.GeneralPrograms;

public class InstanceVariable {

// Java Program to demonstrate

// Instance Variables

// Declared Instance Variable

public String value;

public int i;

public Integer I;

public InstanceVariable()

{

// Default Constructor

// initializing Instance Variable

this.value = "Shubham Jain";

}

// Main Method

public static void main(String[] args)

{

// Object Creation

InstanceVariable name = new InstanceVariable();

// Displaying O/P

System.out.println("Student name is: " + name.value);

System.out.println("Default value for int is "

+ name.i);

// toString() called internally

System.out.println("Default value for Integer is "

+ name.I);

}

}

1. Static Variable

* Static variables are also known as class variables.
* These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.
* Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
* Static variables are created at the start of program execution and destroyed automatically when execution ends.
* Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable. For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.
* If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won’t halt the program. The compiler will replace the object name with the class name automatically.
* If we access a static variable without the class name, the compiler will automatically append the class name. But for accessing the static variable of a different class, we must mention the class name as 2 different classes might have a static variable with the same name.
* Static variables cannot be declared locally inside an instance method.
* Static blocks can be used to initialize static variables.

Example

package in.GeneralPrograms;

public class StaticVariable {

// This is static variable

public static String name="Pushpa";

public static void main(String[] args) {

System.out.println(name);

}

}

Difference Between Instance Variable and Static Variable

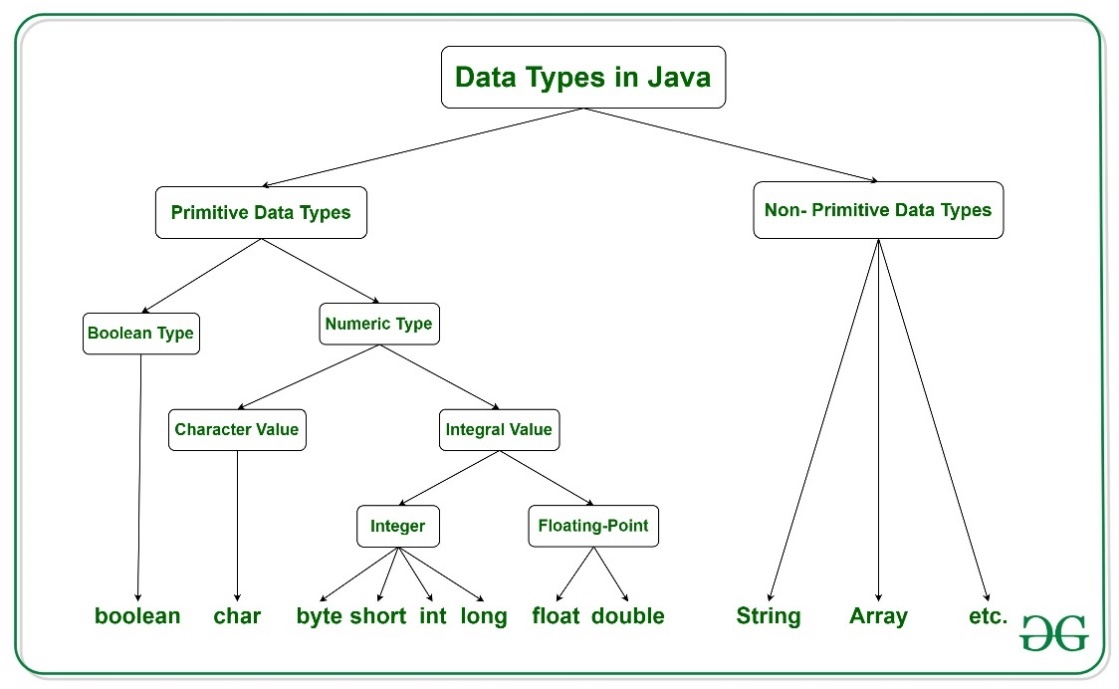
Now let us discuss the differences between the Instance variables and the Static variables:

* Each object will have its own copy of an instance variable, whereas we can only have one copy of a static variable per class, irrespective of how many objects we create. Thus, static variables are good for memory management.
* Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In the case of a static variable, changes will be reflected in other objects as static variables are common to all objects of a class.
* We can access instance variables through object references, and static variables can be accessed directly using the class name.
* Instance variables are created when an object is created with the use of the keyword ‘new’ and destroyed when the object is destroyed. Static variables are created when the program starts and destroyed when the program stops.

Types of Datatypes

In Java, there are two main categories of data types:

1. **Primitive Data Types:** These are the fundamental building blocks for storing data in Java. They are predefined by the language and provide efficient storage and manipulation of basic values. Java offers eight primitive data types:
2. **byte**: Stores 8-bit signed integers (-128 to 127).
3. **short**: Stores 16-bit signed integers (-32,768 to 32,767).
4. **int**: Stores 32-bit signed integers (-2,147,483,648 to 2,147,483,647). This is the most commonly used integer type.
5. **long**: Stores 64-bit signed integers (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807).
6. **float**: Stores single-precision 32-bit floating-point numbers for decimal values.
7. **double**: Stores double-precision 64-bit floating-point numbers for more precise decimal values.
8. **boolean**: Stores Boolean values (true or false).
9. **char**: Stores a single 16-bit Unicode character.
10. **Non-Primitive Data Types**: These are more complex data types that are created by programmers. They are not predefined by the language and offer more flexibility for representing real-world entities and data structures. Here are some common examples:
11. **String**: Represents a sequence of characters. It's immutable (unchangeable) in Java.
12. **Arrays**: Ordered collections of elements of the same data type.
13. **Classes**: Blueprints for creating objects. Objects encapsulate data (member variables) and behavior (methods).
14. **Interfaces**: Define contracts that classes can implement, specifying methods without providing implementations.



Key Points:

* Primitive data types are stored directly in memory and are generally faster to access and manipulate than non-primitive data types.
* Non-primitive data types provide more flexibility and structure for complex data but may have some overhead associated with object creation and memory management.

Choosing the Right Data Type:

* Select the primitive data type that can efficiently store the data you need, considering the range of values and memory usage.
* Use non-primitive data types when you need to represent complex data structures, objects, or collections of data.

Types Of Operator

Java supports eight main types of operators that perform various operations on data and control the flow of your program. Here's a breakdown of these operators:

1. Arithmetic Operators:

Used for performing mathematical calculations on numeric data types (byte, short, int, long, float, double).

* Addition (+)
  + **Working:** Adds two operands (numbers or variables holding numeric values).
  + **Example:** int sum = 10 + 20; // sum will be 30
* Subtraction (-)
  + **Working:** Subtracts the second operand from the first operand.
  + **Example:** int difference = 50 - 15; // difference will be 35
* Multiplication (\*)
  + **Working:** Multiplies two operands.
  + **Example:** int product = 5 \* 8; // product will be 40
* Division (/)
  + **Working:** Divides the first operand (dividend) by the second operand (divisor). In Java, division of integers results in an integer (truncates the decimal part). For floating-point division, use float or double data types.
  + **Example:** int result = 25 / 4; // result will be 6 (integer division)
* Modulo (%)
  + **Working:** Returns the remainder after dividing the first operand by the second operand.
  + **Example:** int remainder = 17 % 3; // remainder will be 1

1. Assignment Operators:

Used to assign values to variables.

* Simple Assignment (=)
  + **Working:** Assigns the value of the right operand to the variable on the left.
  + **Example:** int x = 10;
* Combined Assignment Operators (e.g., +=, -=, \*=, etc.)
  + **Working:** Perform the specified operation (addition, subtraction, multiplication, etc.) with the current value of the variable and the assigned value.
  + **Example:** x += 5; // equivalent to x = x + 5;

1. Relational Operators:

Used for comparing values and resulting in boolean expressions (true or false).

* Equal to (==)

**Working:** Checks if two operands are equal. Returns true if they are equal, false otherwise.

**Example:** int a = 10; boolean isEqual = (a == 10); // isEqual will be true

* Not equal to (!=)

**Working:** Checks if two operands are not equal. Returns true if they are not equal, false otherwise.

**Example:** boolean isNotEqual = (a != 15); // isNotEqual will be true

* Less than (<)

**Working:** Checks if the first operand is less than the second operand. Returns true if it is less than, false otherwise.

**Example:** boolean isLessThan = (a < 20); // isLessThan will be true

* Greater than (>)

**Working:** Checks if the first operand is greater than the second operand. Returns true if it is greater than, false otherwise.

**Example:** boolean isGreaterThan = (a > 5); // isGreaterThan will be true

* Less than or equal to (<=)

**Working:** Checks if the first operand is less than or equal to the second operand. Returns true if it is less than or equal, false otherwise.

**Example:** boolean isLessOrEqualTo = (a <= 10); // isLessOrEqualTo will be true

* Greater than or equal to (>=)

**Working:** Checks if the first operand is greater than or equal to the second operand. Returns true if it is greater than or equal, false otherwise.

**Example:** boolean isGreaterThanOrEqualTo = (a >= 5); // isGreaterThanOrEqualTo will be true

1. Logical Operators:

Used to combine boolean expressions and perform logical operations.

* Logical AND (&&)
  + **Working:** Returns true only if both operands are true. If at least one operand is false, the expression evaluates to false.
* Logical OR (||)
  + **Working:** Returns true if at least one operand is true. If both operands are false, the expression evaluates to false.
* Logical NOT (!)
  + **Working:** Inverts the logical state of the operand. If the operand is true, it becomes false, and vice versa.
  + **Example:** boolean isNegative = !(a > 0); // isNegative will be true if a is not greater than 0

1. Bitwise Operators:

Used to perform bit-level operations on integer data types (byte, short, int, long).

* Bitwise AND (&)
  + **Working:** Performs a bitwise AND operation on two operands. Sets each bit in the result to 1 only if the corresponding bits in both operands are 1.
  + **Example:** int result = 10 & 13; // result will be 8 (binary: 1010 & 1101 = 1000)
* Bitwise OR (|)
  + **Working:** Performs a bitwise OR operation on two operands. Sets each bit in the result to 1 if at least one corresponding bit in either operand is 1.
  + **Example:** int result = 10 | 13; // result will be 15 (binary: 1010 | 1101 = 1111)
* Bitwise XOR (Exclusive OR) (^)
  + **Working:** Performs a bitwise XOR operation on two operands. Sets each bit in the result to 1 if the corresponding bits in the operands are different.
  + **Example:** int result = 10 ^ 13; // result will be 5 (binary: 1010 ^ 1101 = 0101)
* Bitwise Complement (NOT) (~)
  + **Working:** Inverts all bits in the operand.
  + **Example:** int inverted = ~10; // inverted will be -11 (binary: ~1010 = 0101 (inverted) which is -11 in two's complement representation)

1. Shift Operators:

Used to shift bits left or right in integer data types.

* Left Shift (<<)
  + **Working:** Shifts the bits in the left operand to the left by the number of positions specified by the right operand. Zeros are filled in on the right side.
  + **Example:** int result = 5 << 2; // result will be 20 (5 shifted left by 2 positions: 0101 << 2 = 10100)
* Signed Right Shift (>>)
  + **Working:** Shifts the bits in the left operand to the right by the number of positions specified by the right operand. In a signed right shift, the sign bit (leftmost bit) is replicated on the left side during the shift.
  + **Example:** int result = 20 >> 2; // result will be 5 (20 shifted right by 2 positions: 10100 >> 2 = 0101)
* Unsigned Right Shift (>>>)
  + **Working:** Similar to signed right shift, but zeros are filled in on the left side during the shift (useful for unsigned integers).
  + **Example:** int result = 20 >>> 2; // result will be 5 (unsigned 20 shifted right by 2 positions: 000010100 >>> 2 = 00000101)

1. Ternary Operator (Conditional Operator):

A shorthand way to write an if-else statement in a single expression.

* Ternary Operator ( condition ? expression\_if\_true : expression\_if\_false )
  + **Working:** Provides a shorthand way to write an if-else statement in a single expression. Evaluates a condition and returns one of two expressions based on the outcome (true or false).
  + **Example:** int largest = (a > b) ? a : b; // largest will be the larger of a and b

1. instanceof Operator:

Used to check if an object is an instance of a particular class or its subclass.

* instanceof
  + **Working:** Checks if an object reference refers to an object of a particular class or its subclass. Useful for dynamic type checking at runtime.
  + **Example:** if (obj instanceof MyClass) { ... } // This block will execute only if obj refers to an object of MyClass or its subclass.

1. **Increment and Decrement Operator**

**Increment (++)**

* **Prefix Increment (e.g., ++x):**
  + Example:

int counter = 3;

int result = ++counter; // counter becomes 4 first, then result is assigned 4

System.out.println("counter: " + counter); // Prints "counter: 4"

System.out.println("result: " + result); // Prints "result: 4"

Explanation:   
In this example, counter is incremented to 4 first, then that new value (4) is assigned to result.

* **Postfix Increment (e.g., x++):**
  + Example:

int lives = 5;

int damageTaken = lives++; // damageTaken gets the original value (5), then lives is incremented to 6

System.out.println("lives: " + lives); // Prints "lives: 6"

System.out.println("damageTaken: " + damageTaken); // Prints "damageTaken: 5"

Explanation:   
Here, the original value of lives (5) is assigned to damageTaken. Then, after the expression is evaluated, lives is incremented to 6.

**Decrement (--)**

* **Prefix Decrement (e.g., --x):**
  + Example:

int health = 100;

int enemyAttack = --health; // health is decremented to 99 first, then enemyAttack is assigned 99

System.out.println("health: " + health); // Prints "health: 99"

System.out.println("enemyAttack: " + enemyAttack); // Prints "enemyAttack: 99"

Explanation:   
Similar to prefix increment, health is decremented to 99 first, then that new value is assigned to enemyAttack.

* **Postfix Decrement (e.g., x--):**
  + Example:

int level = 3;

int usedPower = level--; // usedPower gets the original value (3), then level is decremented to 2

System.out.println("level: " + level); // Prints "level: 2"

System.out.println("usedPower: " + usedPower); // Prints "usedPower: 3"

Explanation:   
The original value of level (3) is assigned to usedPower. Afterward, level is decremented to 2.

Control Statements

Generally, control statements are divided into 3 parts.

1. Selection statements (Decision Making)
   1. If
   2. If - else
   3. If - else - if
   4. nestd – if
   5. Switch
2. Iteration Statements (Looping)
   1. For
   2. While
   3. Do-while
3. Jump Statements (Branching)
   1. Break
   2. Continue
   3. return

Selection Statements

* if **Statement:** Executes a block of code only if a specified condition is true.
  + Syntax: if (condition) { ... }
  + Example:

int age = 25;

if (age >= 18) {

System.out.println("You are eligible to vote.");

}

* if-else **Statement:** Provides an alternative block of code to execute if the condition in the if statement is false.
  + Syntax: if (condition) { ... } else { ... }
  + Example:

double grade = 87.5;

if (grade >= 90) {

System.out.println("Excellent work!");

} else {

System.out.println("Good effort, keep practicing!");

}

* if-else if **Statement:** Allows you to check multiple conditions sequentially.
  + Syntax: if (condition1) { ... } else if (condition2) { ... } else { ... }
  + Example:

String day = "Tuesday";

if (day.equals("Monday")) {

System.out.println("Start of the week!");

} else if (day.equals("Friday")) {

System.out.println("Almost weekend!");

} else {

System.out.println("It's a weekday.");

}

* Nested – if

Statement: Provides a nesting of if conditions

Examples

public class VotingEligibility {

public static void main(String[] args) {

int age = 20;

boolean isCitizen = true;

// Check if age is eligible and then citizenship

if (age >= 18) {

if (isCitizen) {

System.out.println("You are eligible to vote.");

} else {

System.out.println("You must be a citizen to vote.");

}

} else {

System.out.println("You must be 18 or older to vote.");

}

}

}

* switch **Statement:** Provides a multi-way branching mechanism based on the value of an expression.
  + Syntax: switch (expression) { case value1: ... break; case value2: ... break; default: ... }
  + Example:

char option = 'C';

switch (option) {

case 'A':

System.out.println("You chose option A.");

break;

case 'B':

System.out.println("You chose option B.");

break;

default:

System.out.println("Invalid option selected.");

}

Iteration Statements

* for **Loop:** Provides a concise way to iterate over a block of code a specific number of times or based on a counter variable.
  + Syntax: for (initialization; condition; increment/decrement) { ... }
  + Example:

for (int i = 0; i < 5; i++) {

System.out.println("Iteration number: " + (i + 1)); // +1 for 1-based counting

}

* while **Loop:** Executes a block of code repeatedly as long as a specified condition is true.
  + Syntax: while (condition) { ... }
  + Example:

int count = 1;

while (count <= 3) {

System.out.println("Count: " + count);

count++;

}

* do-while **Loop:** Similar to while, but the code block is executed at least once, even if the condition is initially false.
  + Syntax: do { ... } while (condition);
  + Example:

int input = 0;

do {

System.out.println("Enter a positive number (or 0 to exit): ");

input = scanner.nextInt();

} while (input <= 0);

Jump Statements

* break **Statement:** Used within loops or switch statements to exit the loop or switch block prematurely.
  + Example:

for (int i = 0; i < 10; i++) {

if (i == 7) {

System.out.println("Reached iteration 7 and exiting the loop.");

break;

}

System.out.println(i);

}

* continue **Statement:** Used within loops to skip the remaining code in the current iteration and proceed to the next iteration.
  + Example:

for (int num = 1; num <= 10; num++) {

if (num % 2 == 0) {

continue; // Skip even numbers

}

System.out.println(num);

}

* return   
  **Statement:** this statement is used in method when we return any value at the caller then we use return statements
  + Example:

public class SimpleCalculator {

public static int add(int num1, int num2) {

// Perform addition and store the result

int sum = num1 + num2;

// Return the calculated sum using the return statement

return sum;

}

public static void main(String[] args) {

int a = 5;

int b = 10;

// Call the add method to perform addition

int result = add(a, b);

// Print the result obtained from the method

System.out.println("The sum of " + a + " and " + b + " is: " + result);

}

}

Performed Input and Output using **Scanner** Class (CLI)

To taking input in our program there are several ways are present in java to do that, but we use universal way using Scanner class to get input and output in our class.

**1. Using Scanner Class (CLI):**

This is the most common approach for interactive input from the console. You saw an example in the previous response. Here's a breakdown:

* Import the Scanner class from java.util.
* Create a Scanner object, typically initialized with System.in as the input source.
* Use methods like nextLine() to read entire lines of text or nextInt(), nextDouble(), etc., to read specific data types.

**2. Command-Line Arguments:**

These are arguments passed directly when running your Java program from the command line. You can access them within your program using the args array in the main method:

* Run your program with arguments following the program name (e.g., java MyProgram argument1 argument2).
* In your code, access args in the main method to retrieve these arguments as strings. You might need to parse them to desired data types (e.g., Integer.parseInt(args[0]) for an integer).

**3. BufferedReader Class:**

This class, along with InputStreamReader, provides a character-by-character or line-by-line input stream reading capability. It offers more control compared to Scanner:

* Import BufferedReader and InputStreamReader from java.io.
* Create an InputStreamReader object with System.in as the input source.
* Wrap the InputStreamReader with a BufferedReader for efficient reading.
* Use methods like readLine() to read lines or read() for individual characters.

**4. Console Class (Java 1.6+):**

This class offers methods specifically designed for console input/output (introduced in Java 1.6). It provides a more platform-independent approach compared to System.in and System.out.

* Import the Console class from java.io.
* Check if a console is available using System.console() != null.
* If available, use methods like readLine() or readPassword() for secure password input.

**5. Third-Party Libraries:**

Libraries like Apache Commons CLI or JLine can provide more advanced command-line parsing capabilities, argument validation, and user interaction features.

**Choosing the Right Method:**

* For simple, interactive input from the console during program execution, the Scanner class is generally the easiest and most common choice.
* For processing command-line arguments passed when running the program, accessing those arguments within the main method is the way to go.
* If you need more control over character or line-by-line input reading, BufferedReader might be a better option.
* The Console class is useful for platform-independent console input/output specifically.
* Consider using third-party libraries if you need advanced command-line parsing or user interaction features beyond the basic functionalities of the built-in options.

Using Scanner Class

Example

import java.util.Scanner;

public class InputOutputExample {

public static void main(String[] args) {

// Create a Scanner object to read input from the console

Scanner scanner = new Scanner(System.in);

// Prompt the user for their name

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

String name = scanner.nextLine(); // Read the entire line of input (including spaces)

// Prompt the user for their age

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

int age = scanner.nextInt(); // Read the next integer

// Print a greeting message using the retrieved information

System.out.println("Hello, " + name + "! You are " + age + " years old.");

// Close the Scanner to avoid resource leaks

scanner.close();

}

}

Explanation

1. **Import Scanner:** We import the Scanner class from the java.util package.
2. **Create Scanner Object:** Inside the main method, we create a Scanner object named scanner and initialize it with System.in as the input source (standard input, which is the keyboard in this case).
3. **Prompt and Read Name:**
   * We use System.out.print to display a message "Enter your name: " without a newline character.
   * We use scanner.nextLine() to read the entire line of text entered by the user, including spaces, and store it in the name variable.
4. **Prompt and Read Age:**
   * We use System.out.print again to prompt the user for their age.
   * This time, we use scanner.nextInt() to read the next integer value entered by the user and store it in the age variable.
5. **Output Greeting:** We use System.out.println to print a greeting message that incorporates the user's name and age retrieved from the input.
6. **Close Scanner:** It's good practice to close the Scanner object using scanner.close() to release any associated resources.

**Running the Program:**

1. Save the code as a .java file (e.g., InputOutputExample.java).
2. Compile the code using a Java compiler (e.g., javac InputOutputExample.java).
3. Run the compiled program using the java command (e.g., java InputOutputExample).

Arrays In Java

Arrays are a fundamental data structure in Java that store a collection of elements of the same data type under a single variable name. They offer an efficient way to manage related data and improve code readability. Here's a detailed explanation:

**1. Declaration:**

* **Syntax:**

dataType[] arrayName = new dataType[size];

Use code [with caution.](https://gemini.google.com/faq#coding)

* dataType: This specifies the data type of elements in the array, such as int, double, String, or any custom class type.
* arrayName: This is a user-defined identifier that refers to the entire array. Choose a meaningful name that reflects the data it holds.
* size: This is an integer value that determines the number of elements the array can hold. It must be a positive value, and the size is fixed once the array is created.
* **Example:**

int[] numbers = new int[10]; // Array named "numbers" can hold 10 integers

String[] colors = {"red", "green", "blue"}; // Array named "colors" initialized with 3 strings

**2. Initialization:**

Arrays can be initialized in two ways:

* **During declaration:**

int[] ages = {25, 30, 18, 42}; // Values assigned to elements during declaration

Use code [with caution.](https://gemini.google.com/faq#coding)

* **After declaration:**

double[] prices = new double[5];

prices[0] = 9.99;

prices[1] = 12.50;

// ... assign values to remaining elements

**3. Accessing Elements:**

Individual elements within an array are accessed using their **index**. Arrays are zero-based, meaning the first element has an index of 0, the second element has an index of 1, and so on.

// Accessing element at index 3 (fourth element)

int fourthAge = ages[3]; // fourthAge will be 42

// Modifying an element

colors[1] = "yellow"; // Change the second element from "green" to "yellow"

**Important points to remember when accessing elements:**

* Trying to access an element with an index outside the array bounds (less than 0 or greater than or equal to the size) will result in an ArrayIndexOutOfBoundsException.
* It's generally good practice to validate user input or calculations before using them as array indices to avoid such exceptions.

**4. Length of an Array:**

The length property of an array provides the number of elements it can hold. This value is fixed after creation.

System.out.println("Number of elements in 'numbers' array: " + numbers.length); // Output: 10

**5. Types of Arrays in Java:**

There are two main types of arrays in Java:

**a) Single-dimensional Arrays:**

* These are the most common type, holding elements of a single data type arranged in a linear sequence.
* Example: int[] numbers, String[] names.

**b) Multi-dimensional Arrays:**

* These are arrays of arrays, allowing you to represent data with multiple dimensions (2D, 3D, etc.). They are useful for storing data in grids or matrices.
* Example: A 2D array to represent a matrix can be declared as int[][] matrix = new int[rows][columns].

**6. Common Array Operations:**

Java provides various built-in methods for working with arrays:

* Arrays.sort(array)**:** Sorts the elements of the array in ascending order.
* Arrays.copyOf(array, newSize)**:** Creates a copy of the array with a specified new size.
* Arrays.fill(array, value)**:** Fills all elements of the array with a specific value.

**7. Iterating over Arrays:**

You can iterate through the elements of an array using loops like for or while loops, accessing each element by its index within the loop.

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

System.out.println("Color at index " + i + ": " + colors[i]);

}

**8. Advantages of Arrays:**

* **Efficient memory management:** Elements of the same data type are stored contiguously in memory, improving access speed.
* **Organized data storage:** Arrays provide a structured way to group related data under a single variable name.
* **Improved code readability:** Using arrays can make code more concise and easier to understand compared to using multiple variables for individual data elements.

**9. Disadvantages:**

* **Fixed size:** Once created, the size of an array cannot be changed. If you need to store a dynamically changing number of elements, arrays become cumbersome.
* **Primitive data type restriction:** Single-dimensional arrays can only hold elements of a single data type. If you need to store heterogeneous data (elements of different data types), arrays become less suitable.
* **Memory limitations:** For very large datasets, arrays can consume a significant amount of memory, and managing large arrays can be less efficient.

**10. Alternatives to Arrays:**

Depending on your specific needs, here are some alternatives to consider:

* **ArrayList (from Java Collections Framework):** This is a dynamic array that can automatically resize as needed. It can hold elements of different data types.
* **LinkedList:** This is another dynamic data structure that stores elements in a linked list fashion, offering efficient insertion and deletion operations, but with slower random access compared to arrays.
* **Custom data structures:** You can create your own data structures that cater to your specific data organization and manipulation requirements.

**Choosing the Right Data Structure:**

The best data structure for your program depends on the specific needs of your application. Here are some factors to consider:

* **Data type:** Do you need to store elements of a single data type or a mix of data types?
* **Size:** Is the size of the data collection fixed or dynamic?
* **Operations:** What operations do you need to perform on the data (e.g., frequent insertions/deletions, random access)?
* **Performance:** How critical is fast access or modification of elements?

## CLASS

A class is a user-defined **blueprint** or **prototype** from which objects are created. It represents the set of **properties** or **methods** that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code. In general, class declarations can include these components in order:

note: Remember that highlighted text is the key point

1. **Modifiers**: The class can be either public or have default access. If no modifier is specified, it has default access. It is also known as **Access specifier** or **Visibility mode.**

**public** class Main {

**public** static void main(String[] args) {

System.out.println("Hello and welcome!");

}

}

**Access Modifiers / Access Specifiers / Visibility Modes**

In Java, methods and data members can be encapsulated by the following four access modifiers. The access modifiers are listed according to their restrictiveness order.

1) **private** (accessible within the class where defined)

2) **default or package-private** (when no access modifier is specified)

3) **protected** (accessible only to classes that subclass your class directly within the current or different package)

4) **public** (accessible from any class)

However, the classes and interfaces themselves can have only two access modifiers when declared outside any other class.

1) public

2) default (when no access modifier is specified)

Note: Nested interfaces and classes can have all access modifiers.

Note: We cannot declare class/interface with private or protected access modifiers.

1. **Class**  It should begin with the initial letter capitalized by convention. The name of class would be anything it depends upon what type of work will we perform by using the program. Ex if we want to make a class in which we have methods related to arithmetic calculations then we just simply name our class **calculations.**

public class **Main** {

public static void main(String[] args) {

System.out.println("Hello and welcome!");

}

}

1. **Superclass (if any):** If the class has a parent (superclass), its name is preceded by the keyword extends. A class can only extend (subclass) one parent, If a class doesn’t extends any class then by default it extends **Objects** class { parent class of all the classes} implicitly. We will further discuss this topic in the Inheritance topic.

// this is a parent class

class **first**{

public void display(){

System.out.println(“This is parent class”);

}

//This is child class

public class Main extends **first** {

public static void main(String[] args) {

System.out.println("Hello and welcome!");

}

}

note: child class has capabilities to access most of its parent class members except these.

1. constructors of parent class
2. static members of the parent class
3. private members of its parent class
4. static and non-static block of its parent class
5. **Interfaces (if any):** If the class implements interfaces, their names are listed after the implements keyword, separated by commas. A class can implement more than one interface. In Java, an interface specifies the behavior of a class by providing an abstract type. As one of Java's core concepts, abstraction, polymorphism, and multiple inheritance are supported through this technology. Interfaces are used in Java to achieve abstraction. By using the implemented keyword, a Java class can implement an interface.

**interface** **A** {

public void implementDisplayFunction();

}

public class Main **implements** **A** {

public void inplementDisplayFunction(){

System.out.println(“This method is successfully implemented”);

}

public static void main(String[] args) {

System.out.println("Hello and welcome!");

}

}

1. **Body**: The class body, where you define fields, methods, constructors, etc., is surrounded by braces { }.

public **class** Main extends first **{**

public static void main(String[] args) {

System.out.println("Hello and welcome!");

}

**}**

1. **Methods** : A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping it, which is why they are considered time savers. In Java, every method must be part of some class, which is different from languages like C, C++, and Python.

Example

public void **multiplication**(int a, int b){

return a\*b;

}

public void **addition**(int a, int b){

return a\*b;

}

public void **average**(int a, int b){

return (a+b)/2;

}

## OBJECT

An object is a basic unit of Object-Oriented Programming that represents **real-life entities**. In Java, an object is a self-contained entity that contains both data (also known as attributes or properties) and behavior (methods or functions). Objects are instances of classes, which act as blueprints for creating objects with specific properties and behaviors. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

1. **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
2. **Behaviour**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. **Identity**: It is a unique name given to an object that enables it to interact with other objects.

Mainly objects are classified into two types.

1. Normal Object
2. Anonymous Objects

A. Normal Objects

Let's have a look how object creation is performed in Java .

* // Syntax of java class which we used in program

class sample {

// sample code

}

* // Syntax of object creation

// Syntax

className objectName = new Constructor();

// Example of object creation of sample class

**sample object1 = new sample();** // that’s how object is created

note: there are some steps to create Java objects remember all these points follow consecutively to creating Java objects.

1. **sample** : This is the class name, it means that the instance is a reference to object 1 because object 1 follows its class name.
2. **object1** :This is the name of the object, it would be anything it is not mandatory to give the name object.
3. **=** : This is an assignment operator that is used to assign right side variable value in left side variable look once at the Example.
4. **new :** The new keyword in Java is used to create an instance of a class, also known as an object. The new keyword in Java is used to allocate memory for the object on the heap, the memory space where objects are stored. The new keyword in Java calls the constructor of a class to initialize the object's state.
5. **Sample() :** This is non parameterized constructor of the sample class which is used to assign default values to its data members.

A diagram of a basic language

Description automatically generated with medium confidence

There are the following three steps to create an object of a class. here we use a sample class to demonstrate the object creation.

1. Declaration = A variable declaration with a variable name with an object type.

className **objectName**;

//Here we declared the object of the className class .

1. Instantiation = Java provides the new keyword to create an object of the class.

className objectName = **new** ;

//At this point we instantiate an object but note that here all the variables associated with this object are not initialized.

1. Initialization = The new keyword is followed by a call to a constructor. The call initializes a new object. There are the following three ways to initialize an object in Java:

className objectName = new **className()**;

//At that time our object is successfully created, initiated and by using **classname()** variables{ data members } are also initialized with their default values.

**There are three ways available to initialize an object**

1. **By reference variable:** It means storing data into an object. Multiple objects can be created and store information in it through reference variable.

public class sample {

// these are instance variable

byte age;

String name;

int enrollNo;

// this is display method

void displayResult(){

System.out.println(name);

System.out.println(age);

System.out.println(enrollNo);

}

// this is main method controller of java program

public static void main(String[] args) {

// here is object creation performed

sample sampleStudent1 = new sample();

// instance variable initialized with reference variable and here sampleStudent1 is the reference variable of sample class

sampleStudent1.name = "Puneet Superstar";

sampleStudent1.age = 32;

sampleStudent1.enrollNo = 45236891;

// display method calling using object to display result

sampleStudent1.displayResult();

}

}

1. **By method:** Initializing the objects by invoking the methods.

public class sample {

// these are instance variable

byte age;

String name;

int enrollNo;

// this is display method

void displayResult(){

System.out.println(name);

System.out.println(age);

System.out.println(enrollNo);

}

void initializingVariable(int enrollNo, byte age, String name){

this.enrollNo=enrollNo;

this.age=age;

this.name=name;

}

// this is main method controller of java program

public static void main(String[] args) {

// here is object creation performed

sample sampleStudent1 = new sample();

// at this time instance variable is initialized with java normal parameterized method

sampleStudent1.initializingVariable(46532894,(byte)27,"Rahul" );

// display method calling using object to display result

sampleStudent1.displayResult();

}

}

1. **By constructor:** Constructors can also be used to create an object.

public class sample {

// these are instance variable

byte age;

String name;

int enrollNo;

// this is parameterized constructor

sample(byte age, String name, int enrollNo){

this.name=name;

this.age=age;

this.enrollNo=enrollNo;

}

// this is display method

void displayResult(){

System.out.println(name);

System.out.println(age);

System.out.println(enrollNo);

}

// this is main method controller of java program

public static void main(String[] args) {

// here is object creation performed at same time

// object is also initialized with the help of parameterized constructor

sample sampleStudent1 = new sample((byte) 26,"Pradeep",57789656);

// display method calling using object to display result

sampleStudent1.displayResult();

}

}

Why we use objects in Java?

Below are some points which shows benefits of objects in a language

1. **Encapsulation**: Objects encapsulate data and behaviour into a single unit, which helps to organize and maintain code. Encapsulation also helps to prevent data manipulation from outside the object.
2. **Reusability**: Objects can be reused in different parts of the program, reducing redundancy and promoting code reusability.
3. **Modularity**: Objects provide modularity in Java programs by breaking down complex systems into smaller, more manageable entities.
4. **Inheritance**: Objects can inherit properties and behaviors from other objects (classes), allowing for code reusability and promoting a hierarchical structure in the program.
5. **Polymorphism**: Objects can exhibit different behaviours depending on the context, allowing for flexibility and extensibility in the code.

Different Ways to Create an Object in Java

There are various ways to create an object in Java:

1. Using new keyword // already done
2. Using newInsatnce() Method // used in advanced java
3. Using clone() Method // discussed below
4. Using Deserialization // used in advanced
5. Using Factory method // used in advanced

note: not for beginner.

Creating Object using clone Method

Why we use clone method?

clone() is a method in the Java programming language for object duplication. In Java, objects are manipulated through reference variables, and there is no operator for copying an object—the assignment operator duplicates the reference, not the object. The clone() method provides this missing functionality.

**Method 1**

// using try – catch block

1. Make sure the class you want to create a clone method for implements the Cloneable interface. This interface does not contain any methods, but serves as a marker for the Object.clone() method to know that it is safe to do a field-by-field copy of instances of that class.
2. Override the clone() method in the class you want to make cloneable. The clone() method is protected in the Object class, so you will need to make it public in your class. Here is an Example of how you could override the clone() method:

public class MyClass implements Cloneable {

// this is instance variable

private int myField;

// this is clone @overridded function

public Object clone() {

try {

return super.clone();

} catch (CloneNotSupportedException e) {

throw new AssertionError();

}

}

// Other methods and fields go here

}

1. In the clone() method, call super.clone(), which will create a shallow copy of the object. If you have any mutable fields in your class, you may also need to create a deep copy of those fields to ensure that changes to the original object do not affect the cloned object.
2. Handle the CloneNotSupportedException by throwing an AssertionError since Cloneable interface guarantees that clone() will be supported.
3. To use the clone() method, you can simply call it on an instance of your class, like so:

MyClass original = new MyClass();

MyClass cloned = (MyClass) original.clone();

1. Please note that the clone() method only creates a shallow copy of the object, so if your class has any complex data structures or references, you may need to implement a custom deep copy method to ensure that the cloned object is completely independent of the original object.

**Method 2**

**// without try - catch using throws keyword (simpler one)**

public class sample implements Cloneable{

String name="Sushil";

int age=22;

char gender='F';

public void displayValues(sample obj){

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\* This object is belongs to -> ["+obj.getClass().getSimpleName()+"] class");

System.out.println("Name -> "+obj.name);

System.out.println("Age -> "+obj.age);

System.out.println("Gender -> "+obj.gender);

System.out.println("Object Hash Id-> "+obj.hashCode()+"\n\n");

}

public static void main(String[] args) throws CloneNotSupportedException,AssertionError{

sample obj1 = new sample();

sample clonedObject = (sample) obj1.clone();

// display original object values

obj1.displayValues(obj1);

// display cloned object value

obj1.displayValues(clonedObject);

}

}

Program Output.

// Original object Output

\*\*\*\*\*\*\*\*\*\*\*\*\* This object is belongs to -> [sample] class

Name -> Sushil

Age -> 22

Gender -> F

Object Hash Id-> 122883338

// cloned object Output

\*\*\*\*\*\*\*\*\*\*\*\*\* This object is belongs to -> [sample] class

Name -> Sushil

Age -> 22

Gender -> F

Object Hash Id-> 666641942

Description of Program

1. First we create an class called sample then we implement an cloneable interface in our class to get the property of object cloning.
2. Cloneable is an interface that is used to create the exact copy of an object. It exists in java. lang package. A class must implement the Cloneable interface if we want to create the clone of the class object.
3. Then we create three instance variables (class level variables) called name, age and gender.
4. Then we create a public method with void type called displayValues() method which display the attributes of our objects and its values.
5. Then there we have an main() method attached with throws keywords to handled the exception at the runtime by DEH(Default Exception Handler).
6. The CloneNotSupportedException is an exception in Java that is thrown to indicate that the clone() method in class Object was called to clone an object, but that object's class does not implement the Cloneable interface.
7. In Java, an AssertionError is a subclass of the Error class and it is thrown when an assertion fails. Assertions in Java are typically used to check that preconditions, postconditions, or invariants hold true. As we said below, an AssertionError is thrown when an assertion statement fails to evaluate to true.
8. Then we create an original object called obj1 of our class after that we immediately create an another object called clonedObject with clone method { remember he we not used new keyword to create and allocate new memory in our primary memory for object but at we use cloned method of Object class to assigning and creating memory internally}.
9. Then we pass both the objects in displayValue() function to display the object attributes but here you can see that we add two extra line in function for checking the actual result ether our clone method is working or not.
   1. So in first line we check the object class name to verify that this object is belonging to same class we want or not.
   2. And second one is print the object Hash ID to verify that both objects are different memory space and have not used same memory references.

B. Anonymous Object

Java allows us to create an anonymous object. It means, we can create an object without name. Therefore, we can say that an object that has no reference is known as anonymous object. The disadvantage of an anonymous object is that it can be used at the time of object creation only. It is good to use an anonymous object if we want to use object once.

public class sample {

// these are instance variable

byte age;

String name;

int enrollNo;

// this is parameterized constructor

sample(byte age, String name, int enrollNo){

this.name=name;

this.age=age;

this.enrollNo=enrollNo;

}

// this is display method

void displayResult(){

System.out.println(name);

System.out.println(age);

System.out.println(enrollNo);

}

// this is main method controller of java program

public static void main(String[] args) {

/\* by the using of Anonymous object there is no needed

to have any reference it's all managed by jvm \*/

new sample((byte) 26,"Dolly Chaiwala",45329956).displayResult();

}

}

Creating multiple objects at once

To creating multiple objects at once we must follow same syntax as we declared multiple variables or once.

Declaring multiple variables at once

int a =10, b = 20, c = 30;

Declaring multiple objects at once

sample obj4= new sample(), obj5 = new sample(), obj6 = new sample();

// by using this we create multiple object at once

# OBJECT REFRENCE

Object reference variable simply stores the address of the memory location in the heap at which the object is created and instance variables of that particular object is stored. It's analogous to a variable containing a remote control(having instance variables and methods of that object as keys) mapped to a specific TV(object).so for calling a method or using an instance variable of that object you need a variable that will help you do so…like remote for that specific TV is used to call the TV channels (methods). Therefore, Object reference variable is that variable, which is used to handle (call methods and instance variables) that particular object.

public class sample {

public static void main(String[] args) {

// this is object reference

sample **objReference** = new sample();

} }

Object references in Java help us in various ways:

1. **Memory management:** Object references help Java manage memory efficiently by allowing objects to be created, accessed, and destroyed as needed. Java automatically handles memory management by using garbage collection to reclaim memory no longer in use.
2. **Code reusability:** Object references allow us to create classes and objects that can be reused in different parts of our program. We can create a single class and use it to create multiple objects with different values.
3. **Data sharing**: Object references enable objects to share data and communicate with each other in Java programs. Objects can pass references to other objects as parameters, allowing them to share and manipulate data.
4. **Flexibility:** Object references in Java provide flexibility in creating complex data structures and relationships between objects. We can use references to create hierarchies, dependencies, and associations between objects in our program.

Overall, object references in Java are essential for creating flexible, efficient, and reusable code. They allow us to work with objects, manage memory, share data, and create complex functionalities in our programs.

# METHODS

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping it, which is why they are considered time savers. In Java, every method must be part of some class, which is different from languages like C, C++, and Python.

* A method is like a function i.e. used to expose the behavior of an object.
* It is a set of codes that perform a particular task.

Syntax

<access\_modifier> <return\_type> <method\_name>( list\_of\_parameters)

{

//body

}

Advantage of Method

* Code Reusability
* Code Optimization

**Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list. The method signature is used to differentiate between these methods during compilation. Using method overloading, developers can provide multiple ways of invoking a method, making the code more flexible and easier to use.

A diagram of a method

Description automatically generated

1. Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:
   1. Public: The method is accessible by all classes when we use public specifier in our application.
   2. Private: When we use a private access specifier, the method is accessible only in the classes in which it is defined.
   3. Protected: When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
   4. Default: When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.
2. **Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.
3. **Method**  It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be subtraction(). A method is invoked by its name.
4. **Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.
5. **Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Types of Methods in Java

Classification of method on the base of two types (Understanding purpose only)

1. Based on Return Type:
   1. Parametrized with Return Type
   2. Parametrized with No Return Type
   3. No-Parameterized with Return Type
   4. No-Parameterized with No Return Type
2. Based on Defined Type:
   1. Predefined Methods
   2. User-defined Methods

**note: all these methods are frequently used in java. these are also used depend upon requirement type so look at once onto these also.**

1. Static method
2. Instance method / concrete methods.
3. Getter and setter
4. Abstract method
5. Anonymous method
6. Var-args method
7. Final method
8. Synchronized method.
9. Native method
10. Factory method
11. Default methods
12. Constructor methods
13. Recursive methods

Based on Return Type

1. Parametrized with Return Type

These types of methods can take parameter and return a value when execution of function is complete

Example

public class sample {

public int parameterizedWithReturnType(int number){

return number;

}

public static void main(String[] args) {

sample obj1 = new sample();

System.out.println(obj1.parameterizedWithReturnType(50));

}

}

1. Parametrized with No Return Type

These types of method can take parameter but they cannot return any value

Example

public class sample {

public void parameterizedWithNoReturnType(int number){

System.out.println(“The number is -> ”+number);

}

public static void main(String[] args) {

sample obj1 = new sample();

obj1.parameterizedWithNoReturnType(50);

}

}

1. No-Parameterized with Return Type

These types of methods cannot take parameter but it returns value

Example

public class sample {

int i=152;

public int noParameterizedWithReturnType(){

return this.i;

}

public static void main(String[] args) {

sample obj1 = new sample();

System.out.println(obj1.noParameterizedWithReturnType());

}

}

1. No-Parameterized with No Return Type

These type of method cannot take any value and also they cannot return any value

Example

public class sample {

String course = "Core Java";

public void noParameterizedWithNoReturnType(){

System.out.println("The Course is -> "+course);

}

public static void main(String[] args) {

sample obj1 = new sample();

obj1.noParameterizedWithNoReturnType();

}

}

Based on Defined Type

note: in general methods are actually divided into two types predefined and user defined but all others here showed further classification in methods.

1. Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point.

Example

public class Demo {

public static void main(String[] args) {

// using the max() method of Math class

System.out.print("The maximum number is: " + Math.max(9,7));

} }

2. User-defined Method

The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

1. **Instance Method / Concrete Methods:** Instance methods are the methods defined under a class and we can call such functions only after creating an object of that class. In fact, the call to the instance method is made through the created object itself.
   1. Instance methods are related to an object rather than a class as they can be invoked after creating an object of the class.
   2. We can override an instance method as they are resolved using dynamic binding during run time.
   3. The instance methods are stored in a single memory location.

Syntax:

// Instance Method

Access\_modifier return\_type method\_name(){

body // instance area

}

Example

public void addNumbers(){

// method body

}

There are two types of instance method:

* Accessor Method / Getters Method
* Mutator Method / Setter Methods

**Accessor Method:** The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word get. It is also known as getters. It returns the value of the private field. It is used to get the value of the private field.

Example

public int getId()

{

return Id;

}

**Mutator Method:** The method(s) read the instance variable(s) and also modify the values. We can easily identify it because the method is prefixed with the word set. It is also known as setters or modifiers. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

Example

public void setRoll(int roll)

{

this.roll = roll;

}

// This is demo program to display the working of **accessor** and **mutator**

public class Student {

private int roll;

private String name;

//accessor method

public int getRoll() {

return roll;

}

//mutator method

public void setRoll(int roll) {

this.roll = roll;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public void display() {

System.out.println("Roll no.: "+roll);

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

} }

2. **Static Method:** A method with the "static" keyword is known as a static method. In other words, a method that belongs to a class rather than an instance (object) of a class is called a static method. We can create a static method by using the keyword "static" before the method name.

The main advantage of a static method is that it can be called without creating an object. It can access static data members and even modify their values. Static methods are used to create instance methods that operate on the class level. They are invoked using the class name. The most common Example of a static method is the main() method.

Syntax

Access\_modifiers static\_keyword returntype methodName(){

// method body

}

Example:

//Static Method

public static void display(){

// body of method

}

Abstract Method

The method without a method body is called an abstract method. An abstract method consists only of its declaration. It is always declared in an abstract class. If a class contains an abstract method, it must be an abstract class itself. To create an abstract method, we use the keyword "abstract." If we want to provide a body for an abstract method, we need to extend the abstract class with a normal class and override the abstract method. If the extended class does not override the abstract method, it becomes an abstract class automatically. Note that abstract classes cannot be instantiated or used to create objects because they have abstract methods.

Syntax

abstract void method\_name();

Example

abstract class Demo //abstract class

{

//abstract method declaration

abstract void display(); }

public class MyClass extends Demo

{

//method impelmentation

void display() {

System.out.println("Abstract method?"); }

public static void main(String args[])

{

//creating object of abstract class

Demo obj = new MyClass();

//invoking abstract method

obj.display();

}

}

Output:

Abstract method...

Anonymous methods

In Java, anonymous functions, also known as lambda expressions. It introduced in Java 8 as a way to provide more concise and readable code. They allow us to define a function in a single line of code without having to explicitly define a class or interface.

An anonymous function is a function that has no name and is not bound to an identifier. It is typically defined inline and can be passed as an argument to another function or returned as a value from a function. In Java, anonymous functions are implemented as lambda expressions, which are a shorthand way of defining an anonymous function.

syntax

(parameters) -> { body }

Example

(int a, int b) -> { return a + b; }

Var-args Methods

Let’s suppose you are creating a Java method. However, you are not sure how many arguments your method is going to accept. To address this problem, Java 1.5 introduced varargs.

Varargs is a short name for variable arguments. In Java, an argument of a method can accept arbitrary number of values. This argument that can accept variable number of values is called varargs.

syntax

accessModifier methodName(datatype… arg) {

// method body

}

In order to define vararg, ... (three dots) is used in the formal parameter of a method.

A method that takes variable number of arguments is called a variable-arity method, or simply a varargs method.

Example of without var-args

First, let’s look at the Example without using varargs:

class NoVararg {

public int sumNumber(int a, int b){

return a+b;

}

public int sumNumber(int a, int b, int c){

return a+b+c;

}

public static void main( String[] args ) {

NoVararg obj = new NoVararg();

System.out.println(obj.sumNumber(1, 2));

System.out.println(obj.sumNumber(1, 2, 3));

}

}

When you run this program, the Output will be:

3

6

As you can clearly see, you had to overload sumNumber() method to make it work for 3 arguments. What if the user wants to add 5 numbers or 10 or 100?. This can be handled in a neat way with the use of varargs. Let’s see a code Example:

Example: Working of varargs

class VarargExample {

public int sumNumber(int ... args){

System.out.println("argument length: " + args.length);

int sum = 0;

for(int x: args){

sum += x;

}

return sum;

}

public static void main( String[] args ) {

VarargExample ex = new VarargExample();

int sum2 = ex.sumNumber(2, 4);

System.out.println("sum2 = " + sum2);

int sum3 = ex.sumNumber(1, 3, 5);

System.out.println("sum3 = " + sum3);

int sum4 = ex.sumNumber(1, 3, 5, 7);

System.out.println("sum4 = " + sum4);

}

}

Output:

argument length: 2

sum2 = 6

argument length: 3

sum3 = 9

argument length: 4

sum4 = 16

Final Methods

The final modifier for finalizing the implementations of classes, methods, and variables. We can declare a method as final, once you declare a method final it cannot be overridden. So, you cannot modify a final method from a sub class. The main intention of making a method final would be that the content of the method should not be changed by any outside.

Example

public class FinalMethodExample {

public final void display(){

System.out.println("Hello welcome to Tutorialspoint");

}

public static void main(String args[]){

new FinalMethodExample().display();

}

class Sample extends FinalMethodExample{

public void display(){

System.out.println("hi");

}

}

}

Output

FinalMethodExample.java:12: error: display() in FinalMethodExample.Sample cannot override display() in FinalMethodExample

public void display(){

^

overridden method is final

1 error

Synchronized Method

A synchronized method is a method declared with the synchronized keyword. When a thread invokes a synchronized method, it automatically acquires the lock associated with the object on which the method is called. This ensures that only one thread can execute the synchronized method at a time.

Here's an Example to demonstrate the usage of synchronized methods:

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

public synchronized int getCount() {

return count;

}

}

In the above Example, both the increment() and getCount() methods are declared as synchronized. This guarantees that only one thread can invoke these methods on a Counter object at a time, ensuring thread safety.

Benefits of Using the synchronized Keyword

1. Thread Safety: The synchronized keyword provides a simple and effective way to ensure thread safety in Java programs. By synchronizing access to shared resources, it prevents data inconsistencies and race conditions that can occur when multiple threads access the same data concurrently.
2. Mutual Exclusion: The synchronized keyword enforces mutual exclusion, allowing only one thread to execute the synchronized block or method at a time. This eliminates conflicts and ensures that critical sections of code are executed atomically.
3. Memory Visibility: In addition to thread safety, the synchronized keyword also guarantees memory visibility. When a thread releases the lock, it flushes any changes made to shared variables, making them visible to other threads. This ensures that the most up-to-date values are observed by all threads.
4. Simplicity: The synchronized keyword provides a straightforward way to synchronize code without the need for manual lock management or complex synchronization mechanisms. It is built into the Java language and can be easily understood and applied.

Native Method

Native methods are Java™ methods that start in a language other than Java. Native methods can access system-specific functions and APIs that are not available directly in Java.

The use of native methods limits the portability of an application, because it involves system-specific code. Native methods can either be new native code statements or native code statements that call existing native code.

Once you decide that a native method is required, it may have to interoperate with the Java virtual machine where it runs. The Java Native Interface (JNI) facilitates this interoperability in a platform-neutral way.

The JNI is a set of interfaces that permit a native method to interoperate with the Java virtual machine in numerous ways. For Example, the JNI includes interfaces that create new objects and call methods, get fields and set fields, process exceptions, and manipulate strings and arrays.

Example

public class DateTimeUtils {

public native String getSystemTime();

static {

System.loadLibrary("nativedatetimeutils");

} }

Testing Native Method

public class DateTimeUtilsManualTest {

@BeforeClass

public static void setUpClass() {

// .. load other dependent libraries

System.loadLibrary("nativedatetimeutils");

}

@Test

public void givenNativeLibsLoaded\_thenNativeMethodIsAccessible() {

DateTimeUtils dateTimeUtils = new DateTimeUtils();

LOG.info("System time is : " + dateTimeUtils.getSystemTime());

assertNotNull(dateTimeUtils.getSystemTime());

}

}

Output

[main] INFO c.b.n.DateTimeUtilsManualTest - System time is : Wed Dec 19 11:34:02 2018

Factory Methods

**Definition 1**

In Java, the Factory Method is a creational design pattern that provides an interface for creating objects without explicitly specifying the class to instantiate. It allows subclasses to decide which class to instantiate, giving them the flexibility to provide their own implementation.

A Factory Method involves defining an abstract method in a superclass, which subclasses must override to provide their own concrete implementation for object creation. The subclasses can then choose to create objects of their own classes or other derived classes, depending on the specific requirements.

**Definition 2**

The factory method creates and returns the objects to the client. A factory method may accept an input that denotes the type of object that needs to be created. Factory methods belong to a specific design pattern called "Factory pattern" which is a way to dynamically return an object of a class it belongs to, at run-time based on the user's choice.

The main benefits of using the Factory Method pattern include:

1. Decoupling the creation of objects from the client code, which enhances flexibility and maintainability.

2. Allowing for the dynamic selection of object types at runtime, based on the subclass's implementation.

3. Promoting the "Open-Closed Principle," which states that software entities should be open for extension but closed for modification.

Default Method in Java

In Java, a default method is a method that is added to an existing interface, allowing it to have an implementation. Default methods were introduced in Java 8 as part of the language's evolution to support the addition of new features to interfaces without breaking existing code.

Before default methods, interfaces in Java could only declare abstract methods, which must be implemented by any class that implements the interface. With default methods, interfaces can now include methods with an implementation, providing a default behavior that can be overridden or used as is by implementing classes.

Default methods are denoted by the "default" keyword before the method signature. They can be invoked using the interface name followed by the method name. However, if a class implements an interface with a default method and provides its own implementation for that method, the class should override the default method using the "@Override" annotation to ensure the correct method is being overridden.

Default methods play a crucial role in the implementation of some design patterns, such as the Bean Stream API, where they provide default methods for various operations that can be customized by implementing classes.

Constructor Methods

Constructors in Java have the same name as the class they belong to, and they can be overloaded to provide different ways of initializing objects. This means that a class can have multiple constructors with different parameter types and/or different numbers of parameters.

When you create an object using the "new" keyword, the Java compiler automatically selects the appropriate constructor based on the provided arguments. If no constructor is explicitly defined, the Java compiler provides a default no-argument constructor, which initializes the instance variables to their default values.

Constructors play a vital role in object-oriented programming, as they ensure that objects are properly initialized and ready for use when they are created. They help maintain consistency in object creation and allow for custom initialization based on the object's requirements.

Recursive Methods

In Java, a recursive method is a function or subroutine that calls itself within its own definition. It solves a problem by breaking it down into smaller subproblems of the same kind, which are then solved by further recursive calls. Recursive methods are particularly useful for problems that can be naturally divided into smaller, similar instances of the same problem. They help simplify complex algorithms and can lead to more elegant and efficient solutions. However, it is crucial to ensure proper termination conditions to avoid infinite recursion, which can lead to stack overflow errors.

Naming a Method

A method name is typically a single word that should be a verb in lowercase or a multi-word, that begins with a verb in lowercase followed by an adjective, noun. After the first word, the first letter of each word should be capitalized.

Rules to Name a Method:

1. While defining a method, remember that the method name must be a verb and start with a lowercase letter.
2. If the method name has more than two words, the first name must be a verb followed by an adjective or noun.
3. In the multi-word method name, the first letter of each word must be in uppercase except the first word. For Example, findSum, computeMax, setX, and getX.

Passing Parameters to a method

There are some cases when we don’t know the number of parameters to be passed or an unexpected case to use more parameters than declared number of parameters. In such cases we can use

1. Passing Array as an Argument
2. Passing Variable-arguments as an Argument
3. Passing Object as an argument
4. Passing Normal Arguments
5. Passing Array as an Argument

public class sample {

// receiving array parameters in loopingArray() function

public void loopingArray(int arr[]){

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

System.out.println("Result "+"-> "+arr[i]+" -> Step "+i);

}

}

public static void main(String[] args) {

// creating object

sample objReference = new sample();

// creating static array

int sampleArray[]= {10,20,45,32,65,78,96,54,78,52,45,12,56,32,85};

// sending arguments to loopingArray() functiion

objReference.loopingArray(sampleArray);

}

}

1. Passing Variable-arguments as an Argument

public class sample {

// receiving variable-length parameters

public void averageOf(int ... numbers){

int i =0,sum=0;

while (i<numbers.length){

sum=sum+numbers[i];

i++;

}

System.out.println("Average of input is -> "+sum/numbers.length);

}

public static void main(String[] args) {

// object creation

sample objReference = new sample();

// passing variable-length arguments

objReference.averageOf(20,50,46,32,58,97,58,63,25,41,20);

}

}

1. Passing Object as an argument

public class sample {

// Initialized instance variable at time of declaration

String name = "Manoj tripathi",designation = "Manager";

int empId = 77885544;

long mobileNumber = 7586954525l;

// function which takes object as an parameter of same class

public void displayData(sample obj){

System.out.println("Name -> "+obj.name);

System.out.println("EmpId -> "+obj.empId);

System.out.println("Designation -> "+obj.designation);

System.out.println("MobileNumber -> "+obj.mobileNumber);

}

public static void main(String[] args) {

sample objReference = new sample();

// passing object as an argument to display function

objReference.displayData(objReference);

}

}

1. Passing Normal Arguments

public class sample {

// taking parameter

public void areaOfRectangle(int length, int breadth){

System.out.println("Area of Rectangle is -> "+length\*breadth);

}

public static void main(String[] args) {

// object creation

sample obj1 = new sample();

// passing arguments of an areaOfRectangle function

obj1.areaOfRectangle(40,50);

}

}

Memory Allocation for Methods Calls

Whenever we call a method, the Java Virtual Machine (JVM) allocates memory in two main ways - Heap and Stack.

Heap

Heap memory in Java dynamically allocates memory for objects and JRE classes at runtime. Objects created within methods are stored in the heap, with references managed in the stack. When heap space is full, Java throws java.lang.OutOfMemoryError. Accessing the heap memory is slower than accessing the stack memory.

Unlike the stack, heap memory isn't automatically deallocated; the Garbage Collector handles this task to maintain memory efficiency. However, the heap isn't inherently threadsafe and requires proper synchronization. Thus, managing heap memory efficiently is crucial to prevent memory errors and ensure application stability in Java programs.

Stack

Methods calls are implemented through a stack. It is used for static memory allocation and execution of a thread. Whenever a method is called a stack frame is created within the stack area and after that, the arguments passed to and the local variables and value to be returned by this called method are stored in this stack frame and when execution of the called method is finished, the stack frame is popped out. There is a stack pointer register that tracks the top of the stack which is adjusted accordingly. It used Last In First Out (LIFO) algorithm to manage the stack.

There are several advantages to using methods in Java, including:

1. Reusability: Methods allow you to write code once and use it many times, making your code more modular and easier to maintain.
2. Abstraction: Methods allow you to abstract away complex logic and provide a simple interface for others to use. This makes your code more readable and easier to understand.
3. Improved readability: By breaking up your code into smaller, well-named methods, you can make your code more readable and easier to understand.
4. Encapsulation: Methods allow you to encapsulate complex logic and data, making it easier to manage and maintain.
5. Separation of concerns: By using methods, you can separate different parts of your code and assign different responsibilities to different methods, improving the structure and organization of your code.
6. Improved modularity: Methods allow you to break up your code into smaller, more manageable units, improving the modularity of your code.
7. Improved testability: By breaking up your code into smaller, more manageable units, you can make it easier to test and debug your code.
8. Improved performance: By organizing your code into well-structured methods, you can improve performance by reducing the amount of code that needs to be executed and by making it easier to cache and optimize your code.

# CONSTRUCTORS

Java constructors or constructors in Java is a terminology used to construct something in our programs. A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

In Java, a Constructor is a block of codes similar to the method. It is called when an instance(Object) of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

Rules for constructor

1. Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
2. Constructors do not return any type while method(s) have the return type or void if does not return any value.
3. Constructors are called only once at the time of Object creation while method(s) can be called any number of times.
4. Constructors cannot be inherited in child class.
5. A constructor in Java can not be abstract, final, static, or Synchronized.
6. Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.
7. The first line of a constructor is a call to super() or this(), (a call to a constructor of a super-class or an overloaded constructor), if you don’t type in the call to super in your constructor the compiler will provide you with a non-argument call to super at the first line of your code, the super constructor must be called to create an object:
8. If you think your class is not a subclass it actually is, every class in Java is the subclass of a class object even if you don’t say extends object in your class definition.

Need of Constructors in Java?

Think of a Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object(i.e Box will now exist in the computer’s memory), then can a box be there with no value defined for its dimensions? The answer is No.

So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

Types of Constructors in Java

1. Non-parameterized Constructor / Default Constructor
2. parameterized Constructor
3. Copy Constructor

Non-parameterized construct / Default Constructor

In Java, a non-parameterized constructor, also known as a default constructor, is a constructor that doesn't take any parameters. It's often used to initialize the newly created objects with default values or perform any necessary setup. if there is no constructor in Java class then the Java compiler provides the default construct itself But if programmer has written any constructor in the class then compiler will not insert default constructor in that class.

Example

public class TestingConstructor {

// instance variable

int a,b;

// Default constructor

TestingConstructor(){}

//Setter Method for Assigning values to variable

public void setA(int number){

a=number;

}

public void setB(int number){

b=number;

}

// Display Method for displaying Values of objects

private void display(){

System.out.println("Value of A -> "+a+"\nValue of B -> "+b);

}

public static void main(String[] args) {

// Creating objects of TestingConstructor class with default constructor

TestingConstructor nonparameterizedConstructor = new TestingConstructor();

System.out.println("Default Constructor Running ------------------");

// Run display method to see the value of variables

nonparameterizedConstructor.display();

System.out.println("\nAfter Running Setter Method ------------------");

// running setters method

nonparameterizedConstructor.setA(100);

nonparameterizedConstructor.setB(1000);

nonparameterizedConstructor.display();

}

}

Output

Default Constructor Running ------------------

Value of A -> 0

Value of B -> 0

After Running Setter Method ------------------

Value of A -> 100

Value of B -> 1000

Explanation

In this Example, the MyClass class has a non-parameterized constructor MyClass(), which initializes the value variable with a default value of 0. This constructor is invoked when an object of MyClass is created without any arguments.

Parameterized Constructor

parameterized constructors help to create objects with states defined by the programmer. Objects in Java can be initialized with the default constructor or by a parameterized constructor. Initializing objects with parameterized constructors requires the same number and order of arguments to be passed by the user concerning the parameterized constructor being used. The objects created by using parameterized constructors can be unique with different data member values or states. One can have any number of parameterized constructors in a class. The parameterized constructors differ in terms of the parameters they hold. The compiler would not create a default constructor if the programmer creates their own constructor.

Example

public class TestingConstructor {

// instance variable

int a,b;

// parameterized constructor

TestingConstructor(int a, int b){

this.a=a;

this.b=b;

}

// Display Method for displaying Values of objects

private void display(){

System.out.println("Value of A -> "+a+"\nValue of B -> "+b);

}

public static void main(String[] args) {

// Creating objects of TestingConstructor class with parameterized constructor

TestingConstructor nonparameterizedConstructor = new TestingConstructor(500,600);

System.out.println("parameterized Constructor Running ------------------");

// Run display method to see the value of variables

nonparameterizedConstructor.display();

}

}

Output

parameterized Constructor Running ------------------

Value of A -> 500

Value of B -> 600

Copy constructor

A copy constructor in a Java class is a constructor that creates an object using another object of the same Java class. That’s helpful when we want to copy a complex object that has several fields, or when we want to make a deep copy of an existing object.

In Java, a copy constructor is not a built-in language feature like in some other programming languages, such as C++. However, the concept of a copy constructor can be implemented in Java using the clone() method from the Object class. The clone() method creates a shallow copy of an object, meaning that the new object references the same variables as the original object, rather than creating a completely new, independent copy.

To create a custom copy constructor-like functionality in Java, you can override the Object's clone() method in your class. This allows you to create a new object with the same data as the original, but it requires careful implementation to ensure proper copying of all relevant data members and handling of any resources or special conditions specific to your class.

Example

public class TestingConstructor implements Cloneable{

// instance variable

int a,b;

// Non - parameterized / Default Constructor

TestingConstructor(){}

// parameterized constructor

TestingConstructor(int a, int b){

this.a=a;

this.b=b;

}

// Copy Constructor

TestingConstructor(TestingConstructor obj){

this.a = obj.a;

this.b = obj.b;

}

public Object clone(TestingConstructor ob) throws CloneNotSupportedException,AssertionError{

TestingConstructor clonedObj = (TestingConstructor) super.clone();

clonedObj.a=ob.a;

clonedObj.b=ob.b;

return clonedObj;

}

// Display Method for displaying Values of objects

private void display(TestingConstructor obj){

System.out.println("Value of A -> "+obj.a+"\nValue of B -> "+obj.b);

System.out.println("Object Id -> "+obj);

System.out.println("Object Hash-code -> "+obj.Hash-code());

}

public static void main(String[] args) throws CloneNotSupportedException {

// Creating objects

TestingConstructor normalObject = new TestingConstructor(123,321);

System.out.println("\nDisplay Original Object ------------------");

// Run display method to see the value of variables

normalObject.display(normalObject);

// Cloning Object using Copy Constructor

TestingConstructor copyConstructorObject = new TestingConstructor(normalObject);

System.out.println("\nDisplay Cloned Object using Copy Constructor ------------------");

normalObject.display(copyConstructorObject);

// Cloning Object using Object.clone() method

System.out.println("\nDisplay Cloned Object using Object.clone Method ------------------");

TestingConstructor objectClonedBYMethod ;

objectClonedBYMethod = (TestingConstructor) normalObject.clone(normalObject);

normalObject.display(objectClonedBYMethod);

}

}

Output

Display Original Object ------------------

Value of A -> 123

Value of B -> 321

Object Id -> TestingConstructor@52cc8049

Object Hashcode -> 1389133897

Display Cloned Object using Copy Constructor ------------------

Value of A -> 123

Value of B -> 321

Object Id -> TestingConstructor@506e1b77

Object Hashcode -> 1349393271

Display Cloned Object using Object.clone Method ------------------

Value of A -> 123

Value of B -> 321

Object Id -> TestingConstructor@4fca772d

Object Hashcode -> 1338668845

what if we just simple assign one object to another ?

public class CloningObjectDirectUsingAssignment {

int a,b;

CloningObjectDirectUsingAssignment(){}

CloningObjectDirectUsingAssignment(int a, int b){

this.a=a;

this.b=b;

}

// Display Method for displaying Values of objects

private void display(CloningObjectDirectUsingAssignment obj){

System.out.println("Value of A -> "+obj.a+"\nValue of B -> "+obj.b);

System.out.println("Object Id -> "+obj);

System.out.println("Object Hash-code -> "+obj.Hash-code());

}

public static void main(String[] args) {

System.out.println("\nOriginal Object ----------------------------");

CloningObjectDirectUsingAssignment objOreginal = new CloningObjectDirectUsingAssignment(10,30);

objOreginal.display(objOreginal);

System.out.println("\nCloned Object using Assignment ----------------");

CloningObjectDirectUsingAssignment clonedObj = new CloningObjectDirectUsingAssignment();

clonedObj=objOreginal;

objOreginal.display(clonedObj);

}

}

Output

Original Object ----------------------------

Value of A -> 10

Value of B -> 30

Object Id -> CloningObjectDirectUsingAssignment@52cc8049

Object Hash-code -> 1389133897

Cloned Object using Assignment ----------------

Value of A -> 10

Value of B -> 30

Object Id -> CloningObjectDirectUsingAssignment@52cc8049

Object Hash-code -> 1389133897

"Note: In this case, we can easily observe that both the Object ID and Object Hash-code are similar. The Object ID and Object Hash-code are exactly the same for both objects because when we create an object using the 'new' keyword, a separate copy of the object is created in memory. However, when we assign objects like this, no new object is created; instead, a reference variable of the same class is created, and the address of the first object is simply copied into the second one."

Accessing Class Members

In Java, accessing class members refers to how you interact with the building blocks that define a class. These class members can be:

* **Fields (or member variables):** These store data associated with the class and represent the object's state.
* **Methods:** These are functions defined within the class that perform specific actions or calculations on the object's data (fields) or interact with other objects.

There are two main ways to access class members:

**1. Using the Dot (.) Operator:**

* **Within the same class:** You can directly access any member (field or method) declared within the same class using the dot (.) operator. This is the most common way to access members when working inside the class itself.

public class Person {

private String name;

public int age;

public void setName(String newName) {

name = newName; // Accessing the name field using the dot operator

}

public String getName() {

return name; // Accessing the name field using the dot operator

}

}

In this example: - The name field is private (accessible only within the class) and accessed using the dot operator within the setName and getName methods. - The age field is public (accessible from anywhere) and can be accessed directly using the dot operator throughout the class.

**2. Using the Object Reference:**

* **From outside the class:** If you want to access members of a class from outside the class definition (in another class or the main method), you need to create an object of that class. Then, you use the object reference followed by the dot (.) operator to access the members.

public class Main {

public static void main(String[] args) {

Person person1 = new Person(); // Create an object of Person class

person1.setName("Alice"); // Accessing the setName method using object reference

person1.age = 30; // Accessing the public age field using object reference

System.out.println("Name: " + person1.getName()); // Accessing the getName method using object reference

}

}

Here: - We create an object person1 of the Person class. - To access the setName method and age field (assuming it's public), we use the person1 object reference followed by the dot operator.

**Key Points:**

* Access modifiers (public, private, protected) determine the visibility and accessibility of class members.
* Private members are only accessible within the class, while public members can be accessed from anywhere.
* Protected members are accessible within the class, its subclasses (in the same package or different packages), and the same package.
* The this keyword can be used within methods to refer to the current object instance.

Unit 3

Inheritance And Polymorphism

Topics of this unit

1. Definition Of Inheritance
2. Protected Data, Private Data, Public Data
3. Constructor Chaining
4. Order Of Invocation
5. Types Of Inheritance
   1. Single Inheritance
   2. Multilevel Inheritance
   3. Hierarchical Inheritance
   4. Hybrid Inheritance
6. Method & Constructor Overloading
7. Method Overriding
8. Up-Casting, And Down-Casting

# INHERITANCE

[Inheritance](https://en.wikipedia.org/wiki/Inheritance_%28object-oriented_programming%29) makes it possible to create a child class that inherits the fields and methods of the parent class. The child class can override the values and methods of the parent class, but it’s not necessary. It can also add new data and functionality to its parent.

Parent classes are also called superclasses or base classes, while child classes are known as subclasses or derived classes as well. Java uses the extends keyword to implement the principle of inheritance in code.

**Inheritance in Java:**

* A class (child class) can extend another class (parent class) by inheriting its features
* Implements the [DRY (Don’t Repeat Yourself)](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) programming principle
* Improves code reusability
* Multi-level inheritance is allowed in Java (a child class can have its own child class as well)
* Multiple inheritances are not allowed in Java (a class can’t extend more than one class)

Why Do We Need Java Inheritance?

1. Code Reusability: The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
2. Method Overriding: Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
3. Abstraction: The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

**Important Terminologies Used in Java Inheritance**

1. **Class:**Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. **Super Class/Parent Class:**The class whose features are inherited is known as a superclass(or a base class or a parent class).
3. **Sub Class/Child Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
4. **Reusability:**Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Access Modifiers / Visibility Modes

In Java, access modifiers control the visibility and accessibility of a class's members (variables, methods, constructors). These modifiers determine which parts of your code can access and modify the data and functionality within a class. Here's a breakdown of the three common access modifiers:

1. Private
2. Protected
3. Public
4. Default
5. Private:

**Scope**: Within the class where it's declared.

**Accessibility**: Only accessible by methods within the same class.

**Use Case:** Ideal for member variables that encapsulate the internal state of an object and should not be directly modified from outside the class. By keeping them private, you control how they are accessed and manipulated, promoting data integrity.

Example:

class Person {

private String name;

private int age;

// Methods within the class can access and modify private members

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

Explanation

In this example, name and age are private. Outside code cannot directly access or modify them. The setName and getName methods provide controlled access to the private data.

1. Protected:

**Scope**: Within the class where it's declared, along with all subclasses (derived classes) in the same package or different packages.

**Accessibility**: Accessible by methods within the same class, its subclasses (even in different packages), and the package itself.

**Use Case:** Useful for member variables and methods that need to be shared by subclasses but should not be directly accessed from outside the class hierarchy. It promotes code reuse within the inheritance hierarchy while maintaining some encapsulation.

Example:

class Animal {

protected String name;

public void makeSound() {

System.out.println("Generic animal sound");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Woof!");

}

public void printName() {

System.out.println("Dog's " + name); // Accessing protected member from subclass

}

}

Explanation

Here, name is protected in Animal. Subclasses like Dog can access and potentially modify it (though careful design is recommended for protected members).

1. Public:

**Scope**: Accessible from anywhere in your program.

**Accessibility**: Accessible by any code that has a reference to the class.

**Use Case:** Used for member variables and methods that need to be directly accessed and used from outside the class. However, overuse of public members can reduce encapsulation and make code less maintainable.

Example:

public class Message {

public String content;

public void displayMessage() {

System.out.println(content);

}

}

Explanation

In this example, content and displayMessage are public. Any code with a reference to a Message object can access and modify content or call displayMessage.

Choosing the Right Modifier:

* Use private for internal data that should not be directly modified from outside the class.
* Use protected for members that need to be shared and potentially modified by subclasses, but not directly accessed from outside the inheritance hierarchy.
* Use public sparingly for members that truly need to be accessible from anywhere in your program.
* By effectively using access modifiers, you can promote data encapsulation, code reusability, and maintainability in your Java applications.

1. Default

In Java, the default access modifier, also referred to as package-private, offers a middle ground between private and public access for class members (variables, methods, constructors). Here's a breakdown of its characteristics:

**Scope**: Within the package where the class is declared.

**Accessibility**: Accessible by all classes within the same package but not directly accessible from outside the package.

**Use Cases:**

* Promoting Encapsulation: While not as restrictive as private, default access prevents direct access from outside the package, promoting a degree of encapsulation. You can control how members are accessed and modified by using package-specific helper classes or interfaces.
* Code Organization: When multiple classes within a package collaborate closely, using default access can keep implementation details hidden from other packages, promoting modularity and organization.

Example:

package com.example.util; // Package declaration

class Utility {

int helperValue; // Default access modifier

void performAction() {

System.out.println("Performing action using helperValue: " + helperValue);

}

}

class DataProcessor { // Within the same package

void processData() {

Utility util = new Utility();

util.helperValue = 10; // Accessing default member from same package

util.performAction();

}

}

Explanation

* helperValue and performAction in Utility have default access.
* DataProcessor (in the same package) can access and use helperValue and performAction.
* Code outside the com.example.util package cannot directly access these members.

Key Points:

* If no access modifier is explicitly declared, the default access modifier applies.
* It provides a balance between encapsulation and ease of collaboration within a package.
* Overusing default access can make code less reusable, so consider public or protected access if broader accessibility is needed.

Comparison with Other Access Modifiers:

|  |  |  |
| --- | --- | --- |
| Modifier | Scope | Accessibility |
| Private | Within the class | Only accessible by methods within the same class |
| Protected | Within the class, | Accessible by methods within the same class, its subclasses |
|  | subclasses (same/diff. package) | (even in different packages), and the package itself |
| Public | Anywhere in the program | Accessible by any code that has a reference to the class |
| Default | Within the package | Accessible by all classes within the same package |

A table with text on it

Description automatically generated

Constructor Chaining

In object-oriented programming, constructor chaining is a powerful mechanism that allows you to call one constructor from another constructor within the same class or from a subclass to its parent class constructor. This approach streamlines object initialization by promoting code reusability and reducing redundancy.

Key Concepts:

1. Constructors: Special member functions in a class that are invoked automatically when an object of that class is created. Their primary purpose is to initialize the object's state (member variables).
2. Overloaded Constructors: A class can have multiple constructors with different parameter lists, enabling flexibility in object creation based on the required initialization.
3. Inheritance: A fundamental object-oriented concept where classes can inherit properties and behaviors from parent classes.

Types of Constructor Chaining:

1. Chaining Constructors Within the Same Class (Using this()):
   1. This type of chaining is used to create overloaded constructors that share some common initialization logic and then specialize in specific scenarios.
   2. The first line of the constructor must be a call to this(), which references another constructor within the same class.
   3. this() arguments are passed to the invoked constructor, allowing you to provide default values for some parameters or perform partial initialization.

Example

Java

class Point {

int x;

int y;

Point() {

this(0, 0); // Calls the constructor with (0, 0)

}

Point(int x) {

this(x, 0); // Calls the constructor with (x, 0)

}

Point(int x, int y) {

this.x = x;

this.y = y;

}

}

Explanation

* The default constructor Point() calls this(0, 0), ensuring that both x and y are initialized to 0.
* The constructor Point(int x) calls this(x, 0), setting x to the provided value and y to 0.
* The full constructor Point(int x, int y) takes care of initializing both x and y.

1. Chaining from Subclass to Parent Class Constructor (Using super()):
   1. In inheritance hierarchies, constructors are chained automatically when creating an object of a subclass.
   2. The first line of a subclass constructor must be a call to super(), which invokes the constructor of the immediate parent class.
   3. This ensures that the parent class's member variables are initialized before the subclass's specific logic is executed.

Example (Java):

class ColoredPoint extends Point {

String color;

ColoredPoint(int x, int y, String color) {

super(x, y); // Calls the parent class constructor Point(x, y)

this.color = color;

}

}

Explanation

* When a ColoredPoint object is created, the ColoredPoint constructor first calls super(x, y), delegating the initialization of x and y to the Point class constructor.
* Then, it proceeds to initialize the color member variable specific to ColoredPoint.

Benefits of Constructor Chaining:

1. Reduced Code Duplication: By calling a common constructor from others, you avoid repeating initialization logic, improving code maintainability.
2. Improved Readability: Constructor chaining often makes code more concise and easier to understand, as the flow of initialization becomes clearer.
3. Enforced Base Class Initialization: When inheriting, chaining to the parent class constructor guarantees that the base class's state is properly set up before subclass-specific logic executes.

Constructor chaining is a valuable technique for creating well-structured and efficient object initialization code in object-oriented programming. By effectively using this() and super() keywords, you can promote code reusability, readability, and maintainability in your projects.

Order Of Invocation in constructor Chaining

In constructor chaining, the order of invocation is strictly defined for both types of chaining (within the same class using this() and across classes using super()):

1. Within the Same Class (Using this()):

* The first line of a constructor that uses this() must be a call to another constructor within the same class.
* This ensures that the common initialization logic happens first, followed by any specific initialization for the current constructor.
* Arguments passed to this() are used to initialize the constructor being called.

Example:

class Point {

int x;

int y;

Point() {

this(0, 0); // Calls the constructor with (0, 0)

}

Point(int x) {

this(x, 0); // Calls the constructor with (x, 0)

}

Point(int x, int y) {

this.x = x;

this.y = y;

}

}

Explanation

* When Point() is called, this(0, 0) ensures both x and y are initialized to 0.
* When Point(int x) is called, this(x, 0) sets x and initializes y to 0.

1. Across Classes (Using super()):

* In inheritance hierarchies, constructors are chained automatically when creating an object of a subclass.
* The first line of a subclass constructor must be a call to super(), which invokes the constructor of the immediate parent class.
* This guarantees that the parent class's member variables are initialized before the subclass's specific logic is executed.

Example

class ColoredPoint extends Point {

String color;

ColoredPoint(int x, int y, String color) {

super(x, y); // Calls the parent class constructor Point(x, y)

this.color = color;

}

}

Explanation

* When a ColoredPoint object is created, super(x, y) in the ColoredPoint constructor calls the Point(x, y) constructor, initializing x and y.
* Then, this.color = color sets the color member variable specific to ColoredPoint.

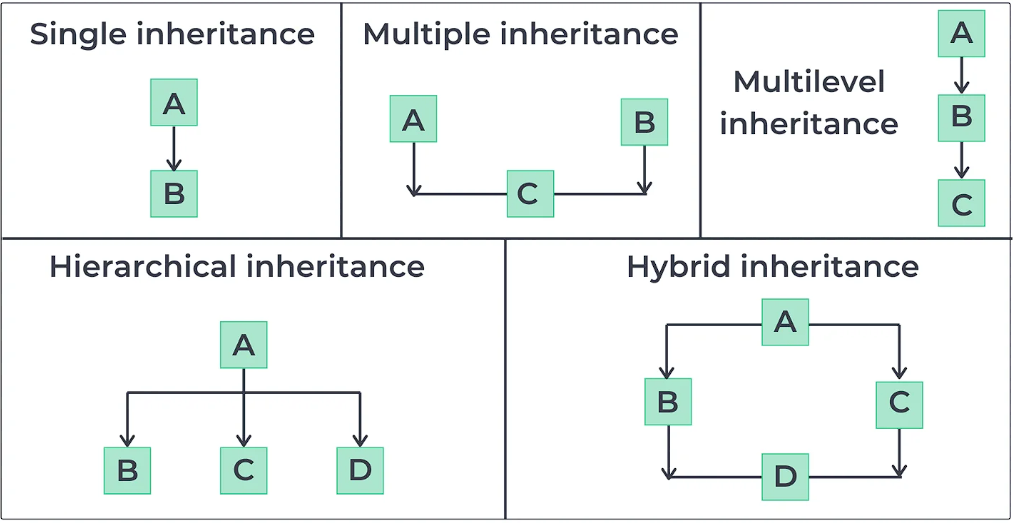
Remember:

* this() and super() calls can only be used within constructors, and only one of each can be used per constructor.
* The order of invocation is crucial: this() calls first within a class, and super() calls first across classes.

Understanding this order of invocation ensures proper initialization of objects in your object-oriented programs using constructor chaining.

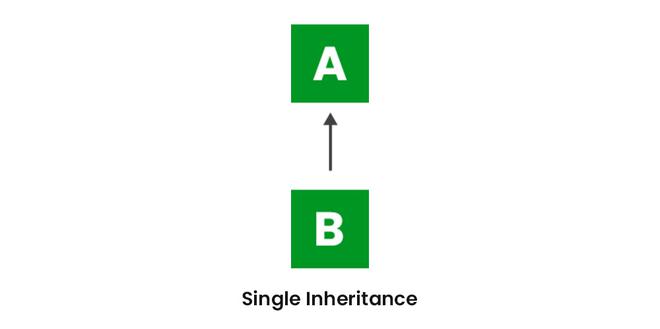
Java Inheritance Types

1. Single-level Inheritance
2. Multi-level Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance (Through Interfaces)
5. Hybrid Inheritance (Through Interfaces)



1. Single level Inheritance

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



Example

package in.InheritancePrograms;

// Parent class (Superclass)

public class Animal {

private String name;

public Animal(String name) {

this.name = name;

}

public String getName() {

return name;

}

// Common method for all animals

public void makeSound() {

System.out.println("The animal makes a sound.");

}

}

// Child class (Subclass)

class Dog extends Animal {

public Dog(String name) {

super(name);

}

// Overriding the makeSound() method

@Override

public void makeSound() {

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

}

}

class Main {

public static void main(String[] args) {

// Create a Dog object

Dog myDog = new Dog("Fido");

// Call the makeSound() method from the Animal class

myDog.makeSound();

// Print the dog's name

System.out.println("Dog's " + myDog.getName());

}

}

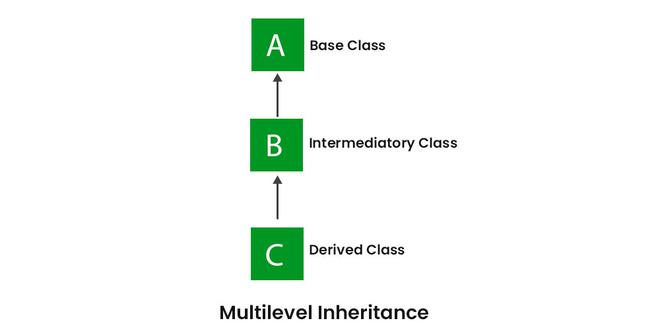
Explanation

In this program, the "Animal" class serves as the parent class or superclass, while the "Dog" class extends the "Animal" class as its child class or subclass. The "Dog" class inherits the properties and methods from the "Animal" class and overrides the "makeSound()" method to provide a more specific implementation.

The main class demonstrates the usage of the "Dog" class and its inherited functionality. In this case, the "makeSound()" method is called on the "myDog" object, which results in the dog-specific sound being printed instead of the generic animal sound.

1. Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent’s members.



Example

// Parent class (Superclass)

public class Animal {

private String name;

public Animal(String name) {

this.name = name;

}

public String getName() {

return name;

}

// Common method for all animals

public void makeSound() {

System.out.println("The animal makes a sound.");

}

}

// Second Level Parent class (Superclass)

public class Mammal extends Animal {

public Mammal(String name) {

super(name);

}

// Overriding the makeSound() method

@Override

public void makeSound() {

System.out.println("The mammal makes a specific mammal sound.");

}

}

// Child class (Subclass)

public class Dog extends Mammal {

public Dog(String name) {

super(name);

}

// Overriding the makeSound() method

@Override

public void makeSound() {

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

}

}

public class Main {

public static void main(String[] args) {

// Create a Dog object

Dog myDog = new Dog("Fido");

// Call the makeSound() method from the Animal class

myDog.makeSound();

// Print the dog's name

System.out.println("Dog's " + myDog.getName());

}

}

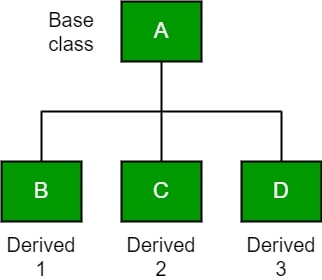
Explanation

In this program, the "Animal" class is the first parent class, followed by the "Mammal" class, which extends the "Animal" class. The "Dog" class then extends the "Mammal" class. The "Dog" class inherits properties and methods from both the "Mammal" and "Animal" classes.

The main class demonstrates the usage of the "Dog" class and its inherited functionality. In this case, the "makeSound()" method is called on the "myDog" object, which results in the dog-specific sound being printed instead of the generic mammal or animal sound.

1. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.



Example

// Parent class (Superclass)

public class Shape {

private String name;

public Shape(String name) {

this.name = name;

}

public String getName() {

return name;

}

// Common method for all shapes

public void draw() {

System.out.println(name + " shape is being drawn.");

}

}

// Child class (Subclass)

public class TwoDimensionalShape extends Shape {

public TwoDimensionalShape(String name) {

super(name);

}

// Overriding the draw() method

@Override

public void draw() {

System.out.println(name + " two-dimensional shape is being drawn.");

}

}

// Subclass of TwoDimensionalShape

public class Rectangle extends TwoDimensionalShape {

public Rectangle(String name) {

super(name);

}

// Overriding the draw() method

@Override

public void draw() {

System.out.println("Rectangle is being drawn.");

}

}

public class Main {

public static void main(String[] args) {

// Create a Rectangle object

Rectangle myRectangle = new Rectangle("Red Rectangle");

// Call the draw() method from the Shape class

myRectangle.draw();

// Print the rectangle's name

System.out.println("Rectangle's " + myRectangle.getName());

}

}

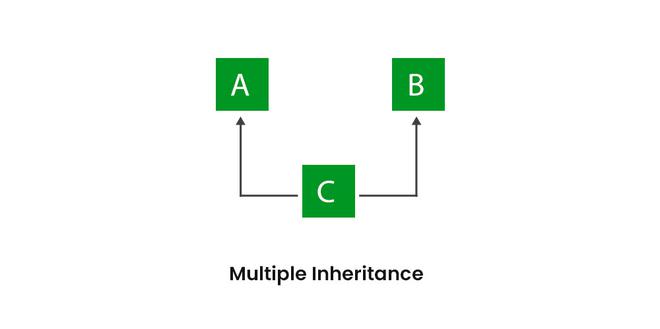
Explanation

In this program, the "Shape" class is the parent class, followed by the "TwoDimensionalShape" class, which extends the "Shape" class. The "Rectangle" class then extends the "TwoDimensionalShape" class. The "Rectangle" class inherits properties and methods from both the "TwoDimensionalShape" and "Shape" classes.

The main class demonstrates the usage of the "Rectangle" class and its inherited functionality. In this case, the "draw()" method is called on the "myRectangle" object, which results in the rectangle-specific drawing being printed instead of the generic two-dimensional shape or shape drawing.

1. Multiple Inheritance (Through Interfaces)

In [Multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/), one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support [multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. In Java, we can achieve multiple inheritances only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/). In the image below, Class C is derived from interfaces A and B.



Example

// Interface 1 (Superclass)

interface Animal {

public void makeSound();

}

// Interface 2 (Superclass)

interface Mammal {

public void giveBirth();

}

// Class implementing both interfaces (Subclass)

public class Dog implements Animal, Mammal {

@Override

public void makeSound() {

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

}

@Override

public void giveBirth() {

System.out.println("Dogs give birth to puppies.");

}

}

public class Main {

public static void main(String[] args) {

Dog myDog = new Dog();

// Call the methods from the implemented interfaces

myDog.makeSound();

myDog.giveBirth();

}

}

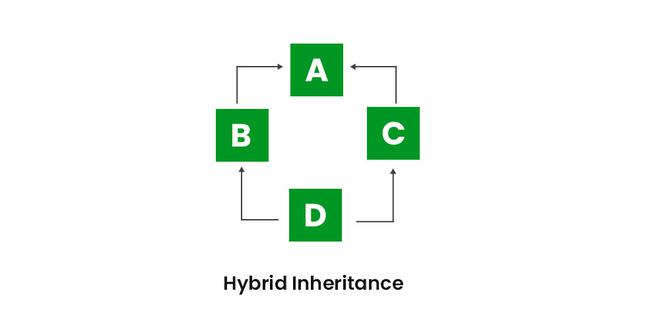
Explanation

In this program, the "Animal" and "Mammal" interfaces serve as the parent classes, and the "Dog" class implements both interfaces. The "Dog" class provides the implementations for the methods declared in the interfaces.

The main class demonstrates the usage of the "Dog" class and its implementation of the methods from the implemented interfaces. In this case, the "makeSound()" and "giveBirth()" methods are called on the "myDog" object, which prints the respective method's content.

1. Hybrid Inheritance (Through Interfaces)

It is a mix of two or more of the above types of inheritance. Since Java doesn’t support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.  
However, it is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.



Example

// Interface 1 (Superclass)

interface Animal {

public void makeSound();

}

// Interface 2 (Superclass)

interface Mammal {

public void giveBirth();

}

// Class implementing an interface and extending a parent class (Subclass)

public class Dog extends ParentClass implements Animal, Mammal {

@Override

public void makeSound() {

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

}

@Override

public void giveBirth() {

System.out.println("Dogs give birth to puppies.");

}

}

// Parent class

abstract class ParentClass {

public abstract void commonMethod();

}

public class Main {

public static void main(String[] args) {

Dog myDog = new Dog();

// Call the methods from the implemented interfaces and the parent class

myDog.makeSound();

myDog.giveBirth();

myDog.commonMethod();

}

}

Explanation

In this program, the "Animal" and "Mammal" interfaces serve as the first set of parent classes, and the "ParentClass" serves as the second parent class. The "Dog" class extends the "ParentClass" and implements both interfaces. The "Dog" class provides the implementations for the methods declared in the interfaces and the abstract method in the parent class.

The main class demonstrates the usage of the "Dog" class and its implementation of the methods from the implemented interfaces and the parent class. In this case, the "makeSound()", "giveBirth()", and "commonMethod()" methods are called on the "myDog" object, which prints the respective method's content.

Advantages and Disadvantages of Inheritance

Advantages

1. Java inheritance enables code reusability and saves time.
2. Inheritance in Java provides the extensibility of inheriting parent class methods to the child class.
3. With Java inheritance, the parent class method overriding the child class is possible.

Disadvantages

1. The inherited methods lag in performance.
2. Some of the data members of the parent class may not be of any use—as a result, they waste memory.
3. Inheritance causes strong coupling between parent and child classes. This means, either of the two (Parent class or Child class) become incapable of being used independent of each other.

Problems with Inheritances

1. **Diamond Problem:** In multiple inheritance with concrete classes (not recommended), if two superclasses have a common superclass and both are inherited by a subclass, a diamond-shaped inheritance hierarchy is formed. This can lead to ambiguity about which method to call from the common superclass. Interfaces are preferred to avoid this issue because interfaces have an abstract methods and we must have to override method in implemented class so common method which conflicts is overridden in child class that’s how interfaces solves the Diamon case problem.
2. **Accessibility:** When inheriting, visibility of members (private, protected, public) from the superclass affects accessibility in the subclass. Generally, private members are not accessible directly, while protected members can be accessed by subclasses.

Relationship In Inheritance

In object-oriented programming, one of the fundamental concepts is inheritance. In Java, inheritance allows us to create new classes based on existing ones, inheriting their properties and behaviors. The relationship between classes is often referred to as an "is-a" relationship. In this section, we will explore what an is-a relationship is and how it is implemented in Java.

Types of Relationship

* IS-A Relationship (Inheritance)
* HAS-A Relationship (Classes and Objects)

1. Is-A Relationship (Inheritance):
2. Represents a hierarchical connection between classes.
3. Involves inheritance, where a subclass ("child" class) inherits properties (member variables) and methods (member functions) from a superclass ("parent" class).
4. The subclass is considered a specialization of the superclass. It can add new properties and methods, or redefine inherited ones to provide more specific behavior.
5. This relationship uses the phrase "is-a" because a subclass instance can be treated as an instance of its superclass.

Example:

class Animal {

public void makeSound() {

System.out.println("Generic animal sound");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Woof!");

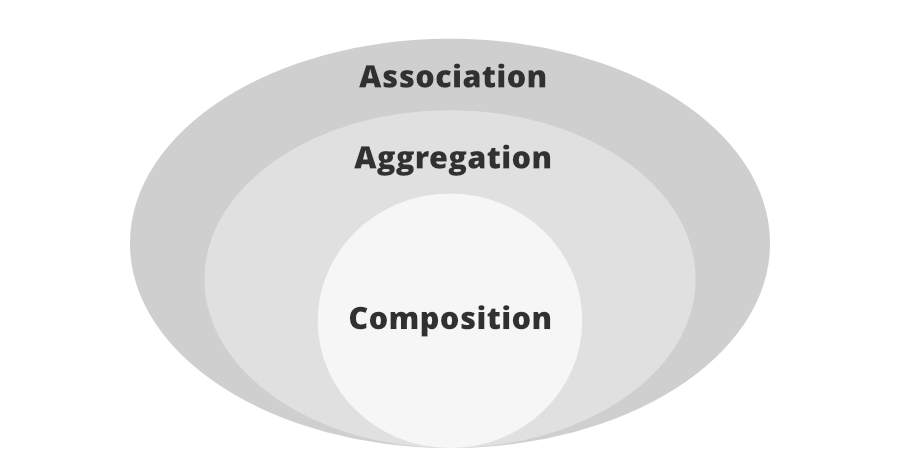
}

}

Explanation

* Dog is-a kind of Animal.
* A Dog object inherits the makeSound method from Animal.
* You can use a Dog object with an Animal reference variable, but not vice versa (due to potential loss of information).

1. Has-A Relationship (Composition):
2. Represents a compositional connection between classes.
3. Involves one class (whole) having a reference to an instance of another class (part) as a member variable.
4. The whole class contains or aggregates the part class.
5. This relationship uses the phrase "has-a" because the whole class has a member variable that holds a reference to an instance of another class.



Association

[Association](https://javapapers.com/oops/association-aggregation-composition-abstraction-generalization-realization-dependency/) means the act of establishing a relationship between two unrelated classes. For Example, when you declare two fields of different types (e.g. Car and Bicycle) within the same class and make them interact with each other, you have created an association.

**Association in Java:**

* Two separate classes are associated through their objects
* The two classes are unrelated, each can exist without the other one
* Can be a one-to-one, one-to-many, many-to-one, or many-to-many relationship

Aggregation

[Aggregation](https://beginnersbook.com/2013/05/aggregation/) is a narrower kind of association. It occurs when there’s a one-way (HAS-A) relationship between the two classes we associate through their objects.

For Example, every Passenger has a Car, but a Car doesn’t necessarily have a Passenger. When you declare the Passenger class, you can create a field of the Car type that shows which car the passenger belongs to. Then, when you instantiate a new Passenger object, you can access the data stored in the related Car as well.

**Aggregation in Java:**

* One-directional association
* Represents a HAS-A relationship between two classes
* Only one class is dependent on the other

Composition

[Composition](https://www.infoworld.com/article/3029325/application-development/exploring-association-aggregation-and-composition-in-oop.html) is a stricter form of aggregation. It occurs when the two classes you associate are mutually dependent and can’t exist without each other.

For Example, take a Car and an Engine class. A Car cannot run without an Engine, while an Engine also can’t function without being built into a Car. This kind of relationship between objects is also called a PART-OF relationship.

**Composition in Java:**

* A restricted form of aggregation
* Represents a PART-OF relationship between two classes
* Both classes are dependent on each other
* If one class ceases to exist, the other can’t survive alone

Example:

class Car {

private Engine engine; // Car has-a Engine

public Car(Engine engine) {

this.engine = engine;

}

public void start() {

engine.start(); // Car uses Engine's functionality

}

}

class Engine {

public void start() {

System.out.println("Engine starting...");

}

}

Explanation

* Car has-a Engine.
* A Car object has an engine member variable that references an Engine object.
* The Car class uses the Engine object's start method to start the car.

Comparison of Relationships

|  |  |  |
| --- | --- | --- |
| Feature | Is-A Relationship (Inheritance) | Has-A Relationship (Composition) |
| Relationship | Hierarchical | Compositional |
| Inheritance | Yes | No |
| Specialization | Subclass is a specialized version of superclass | Whole class can be composed of different parts |
| Code Reusability | Reuses code from superclass | Promotes modularity and code organization |
| Example | Dog is-a Animal | Car has-a Engine |

Choosing the Right Relationship:

* Use inheritance when there's a clear "is-a" relationship between classes. It promotes code reuse and a well-defined class hierarchy.
* Use composition when one class needs to contain or aggregate another class as a part of its functionality. It promotes modularity and flexibility in object composition.
* By understanding these relationships, you can design object-oriented programs that are well-structured, maintainable, and leverage the power of code reuse and modularity.

Super Keyword

The super keyword is a unique keyword that refers to an immediate parent class's object it is used to access and refer to the parent class's methods, properties, and constructors in the context of a subclass (child class). It helps in differentiating between the parent class and the child class's versions of the same method or property. If you create an instance of the child class, then the super keyword implicitly refers to the parent class instance.

1. "super" keyword in Java refers to the parent class or superclass of the current class.
2. super is to be the first statement when it usage.
3. It is used to access and invoke methods or variables from the parent class.
4. Helps in code reusability and maintaining consistency across classes.
5. Assists in overriding parent class methods in child classes.
6. Useful when you want to access parent class constructors, methods, or variables that have been overridden or hidden in the child class.
7. Enhances encapsulation and inheritance principles in object-oriented programming.

Example of super

class Person{

int id;

String name;

Person(int id,String name){

this.id=id;

this.name=name;

}

}

class Emp extends Person{

float salary;

Emp(int id,String name,float salary){

super(id,name);//reusing parent constructor

this.salary=salary;

}

void display(){System.out.println(id+" "+name+" "+salary);}

}

class TestSuper5{

public static void main(String[] args){

Emp e1=new Emp(1,"ankit",45000f);

e1.display();

}}

Method Overloading

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

## **Advantage of method overloading**

Method overloading increases the readability of the program.

Overload method of two types

1. Numbers of parameters
2. Different Datatypes
3. **Number of Parameters:** Methods can have the same name but accept a different number of arguments.
4. **Data Types of Parameters:** Methods can have the same name but accept arguments of different data types.

This flexibility provides several advantages:

* **Improved Readability:** Code becomes more intuitive as method names reflect their functionality, even with varying inputs.
* **Increased Reusability:** You can create generic methods with the same name and tailor their behavior based on the provided arguments.
* **Enhanced Code Maintainability:** Overloading promotes modularity and reduces code duplication.

**Benefits of Method Overloading:**

* **Improved Readability:** By using the same method name for related functionalities with different inputs, the code becomes more intuitive and easier to understand.
* **Increased Reusability:** You can create generic methods with the same name and customize their behavior based on the provided arguments.
* **Flexibility:** Method overloading allows you to handle various data types and input scenarios within a single class.

1. **Overloading by Changing the Number of Arguments**

Here's an example in Java:

Java

class Calculator {

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

public String add(String a, String b) {

return a.concat(b); // Concatenates strings

}

}

In this Calculator class:

* The add method can be called with two integer arguments to perform integer addition.
* The same add method can be called with two double arguments for double-precision addition.
* It can also be called with two String arguments to concatenate them.

Java uses method resolution to identify the correct add method to invoke based on the number and types of arguments provided at call time.

1. **Overloading by Changing Data Types**

Here's another example demonstrating overloading based on data type variations:

Java

class Shape {

public double area(Circle circle) {

return Math.PI \* circle.radius \* circle.radius;

}

public double area(Rectangle rectangle) {

return rectangle.length \* rectangle.width;

}

}

The Shape class has two area methods:

* The first area method takes a Circle object as an argument and calculates its area using the circle's radius.
* The second area method takes a Rectangle object and calculates its area using its length and width.

This approach allows for calculating areas of different shapes using the same method name but with specific object types.

**Key Points to Remember:**

* Method overloading decisions are made based on the **number and data types** of arguments at call time, not the return type.
* Methods cannot be overloaded solely based on the return type.
* Overloaded methods cannot have the same access modifiers (e.g., public, private) within a class.

**How Java Distinguishes Overloaded Methods:**

When you call a method with a specific set of arguments, Java uses a process called **method resolution** to determine which overloaded method to invoke. Here's the order of precedence:

1. **Exact match:** If there's a method with the exact number and data types of arguments you provide, that method is called. (e.g., add(1, 2))
2. **Promotion:** If an exact match isn't found, Java considers promoting numeric types (e.g., int to double). So, add(10.5, 5) would call the add(double, double) method.
3. **Boxing/Unboxing:** For primitive types and their corresponding wrapper classes (e.g., int and Integer), Java can perform boxing/unboxing to convert between them. This allows some flexibility in argument types.

**Important Considerations:**

* Method overloading cannot be achieved based on return type alone. Methods must differ in parameter characteristics.
* Overloaded methods cannot have the same access specifier (e.g., public, private) within a class.

Constructor Overloading

Constructor overloading is a programming concept in object-oriented programming languages, such as Java, where a class can have multiple constructors with the same name but with different sets of parameters. This allows you to create objects of the class with varying initial states by providing different arguments when creating an instance.

When you define multiple constructors in a class, each constructor has a unique combination of parameters, which can include different data types, numbers of arguments, or a mix of both. This enables you to initialize the object's state based on the provided arguments, offering flexibility in object creation.

Rules of construct Overloading

1. Constructor calling must be the first statement of the constructor in Java.
2. If we have defined any parameterized constructor, then the compiler will not create a default constructor. and vice versa if we don’t define any constructor, the compiler creates the default constructor(also known as no-arg constructor) by default during compilation
3. Recursive constructor calling is invalid in Java.

Example

public class Rectangle {

private double width;

private double height;

// Constructor with width and height parameters

public Rectangle(double width, double height) {

this.width = width;

this.height = height;

}

// Constructor with only width parameter

public Rectangle(double width) {

this.width = width;

this.height = 1.0;

}

public static void main(String[] args) {

// This is two parameterized construct

Rectangle recObj1 = new Rectangle(30,10);

// This is single parameterized construct

Rectangle recObj2 = new Rectangle(10);

}

}

Difference between Java Constructor and Java Method

|  |  |
| --- | --- |
| Constructor | Method |
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |

Copy Constructor vs .Clone

1. In Java, we can also use the clone method to create an object from an existing object. However, the copy constructor has some advantages over the clone method:
2. The copy constructor is much easier to implement. We do not need to implement the Cloneable interface and handle CloneNotSupportedException.
3. The clone method returns a general Object reference. Therefore, we need to typecast it to the appropriate type.
4. We can not assign a value to a final field in the clone method. However, we can do so in the copy constructor.

Using this() in Constructor Overloading

this() reference can be used during constructor overloading to call the default constructor implicitly from the parameterized constructor. by using this keyword we can call any parameter constructor explicitly

Example

public class ConstructorOverloading {

// no-parameterized constructor

ConstructorOverloading(){

this(10);

System.out.println("This is Default Constructor");

}

// Single parameterized constructor

ConstructorOverloading(int a){

System.out.println("This is one Parameterized Constructor");

}

// Two parameterized constructor

ConstructorOverloading(String a,int b){

this(10,20,30);

System.out.println("This is Two Parameterized Constructor");

}

// Three parameterized constructor

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

this();

System.out.println("This is Three parameterized Constructor");

}

public static void main(String[] args) {

ConstructorOverloading obj = new ConstructorOverloading("Hello",10);

}

}

Output

This is one Parameterized Constructor

This is Default Constructor

This is Three parameterized Constructor

This is Two Parameterized Constructor

Explanation

This program demonstrates constructor overloading in Java. The class "ConstructorOverloading" contains multiple constructors with the same name but different sets of parameters.

1. The default constructor:

ConstructorOverloading(){

this(10);

System.out.println("This is Default Constructor");

}

Here, the default constructor calls another constructor with a parameter (10) using the "this" keyword. It also prints the message "This is Default Constructor."

1. One-parameterized constructor:

ConstructorOverloading(int a){

System.out.println("This is one Parameterized Constructor");

}

This constructor takes an integer parameter 'a' and prints the message "This is one Parameterized Constructor."

1. Two-parameterized constructor:

ConstructorOverloading(String a,int b){

this(10,20,30);

System.out.println("This is Two Parameterized Constructor");

}

This constructor takes a String parameter 'a' and an integer parameter 'b.' It calls another constructor with three parameters (10, 20, 30) using the "this" keyword and prints the message "This is Two Parameterized Constructor."

1. Three-parameterized constructor:

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

this();

System.out.println("This is Three parameterized Constructor");

}

This constructor takes three integer parameters 'a,' 'b,' and 'c.' It calls the default constructor using the "this" keyword and prints the message "This is Three parameterized Constructor."

In the "main" method, an object 'obj' of the "ConstructorOverloading" class is created, passing a String ("Hello") and an integer (10) as arguments. This invokes the two-parameterized constructor, which in turn calls the three-parameterized constructor, the default constructor, and finally prints the message "This is Two Parameterized Constructor."

Method Overriding

Method overriding, also known as "method redefinition," is a crucial concept in object-oriented programming (OOP) that allows a subclass or child class to provide its implementation of a method that is already defined in its parent class or superclass. This feature is supported in languages like Java and C++.

1. Basics: Method overriding occurs when a subclass provides a new implementation for a method that is already present in its parent class. The subclass method should have the same method signature (method name, parameter types, and return type) as the parent class method.
2. Purpose: Overriding enables polymorphism, which is a fundamental aspect of OOP. It allows for more flexible and dynamic behavior in the child class, as it can provide an alternative implementation of a method without changing the parent class's structure.
3. Access Modifiers: The overriding method in the child class can have a more relaxed access modifier than the parent class method, but it cannot be more restrictive. For example, if the parent method is public, the child method can be public, protected, or even package-private (without any access modifier), but it cannot be private.
4. Overriding vs. Overloading: While both concepts involve methods, overriding deals with methods in different classes with the same signature, and overloading deals with methods within the same class with different signatures.
5. Dynamic Binding: Overriding methods utilize dynamic binding, where the actual method to be executed is determined at runtime based on the object's class. This is also known as late binding.
6. Overriding Rules: The overriding method should have the same return type (either the same or subtype), and it should have the same method name and parameter types. Additionally, it must specify the "override" keyword, which is optional in Java but required in C++.
7. Overridden Method Invocation: To invoke the parent class's original method from the child class, you can use the "super" keyword, followed by the method name. This is particularly useful when you want to use functionality from the parent class alongside the child class's custom implementation.
8. Overriding and Polymorphism: Overriding plays a vital role in achieving polymorphism, which allows objects to be treated as objects of their parent class or interface types. This flexibility enables dynamic behavior based on the actual object's type at runtime.

Example:

class Parent {

void print(int a) {

System.out.println("Parent's print(int) method");

}

}

class Child extends Parent {

void print(String a) {

System.out.println("Child's print(String) method");

}

void print(int a) {

super.print(a); // Invoking parent class's print(int) method

System.out.println("Child's print(int) method");

}

}

public class Main {

public static void main(String[] args) {

Child myChild = new Child();

myChild.print(10);

}

}

note: In this example, the "super.print(a)" statement invokes the parent class's "print(int)" method this will also demonstrate how parent class method is called using super.

Up-casting And Down-casting

Upcasting and downcasting in Java are fundamental concepts related to type casting and inheritance. They deal with how you handle references to objects of different classes in a class hierarchy.

1. Upcasting (Widening Conversion):
   1. Upcasting refers to converting a reference variable of a subclass type to a reference variable of a superclass type.
   2. It's also known as widening conversion because the superclass is generally considered more general (wider) than the subclass (more specific).
   3. Upcasting is safe and often implicit (automatic) in Java.

Why Upcasting?

* Promotes code flexibility: You can write code that works with objects of different subclasses as long as they share a common superclass. This is particularly useful for collections and polymorphism scenarios.
* Enhances abstraction: By using superclass references, you isolate the core functionalities from subclass-specific details, leading to cleaner code.

Example:

class Animal {

public void makeSound() {

System.out.println("Generic animal sound");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Woof!");

}

}

public class Main {

public static void main(String[] args) {

// Upcasting implicitly

Animal animalRef = new Dog(); // Upcast Dog object to Animal reference

animalRef.makeSound(); // Calls Animal's makeSound() (Generic sound)

// Upcasting explicitly (not strictly necessary in this case)

Animal anotherAnimalRef = (Animal) new Dog(); // Explicit cast

anotherAnimalRef.makeSound(); // Calls Animal's makeSound() (Generic sound)

}

}

Explanation

* A Dog object is created.
* The Dog object is assigned to an Animal reference variable (animalRef). This is upcasting.
* When makeSound is called on animalRef, the method from the Animal class is invoked (generic sound), even though the actual object is a Dog. This is because the reference variable type (Animal) determines which method is accessible.

1. Down-casting (Narrowing Conversion):
2. Downcasting refers to converting a reference variable of a superclass type to a reference variable of a subclass type.
3. It's also known as narrowing conversion because the subclass is more specific (narrower) than the superclass (more general).
4. Downcasting is not always safe and requires an explicit cast in Java.

Why Downcasting?

* Access subclass-specific methods: Upcasting allows access to superclass methods, but sometimes you need methods specific to the subclass. Downcasting allows you to regain access to those methods.
* Type safety: Although not always necessary, downcasting can sometimes clarify the intended type of an object for better code readability. However, use it cautiously to avoid runtime errors.

Example (Illustrative - May cause ClassCastException):

Animal animalRef = new Dog(); // Upcasting (safe)

// Assuming the context guarantees the object is actually a Dog

Dog downcastedDog = (Dog) animalRef; // Downcasting (explicit cast)

downcastedDog.fetch(); // Calls Dog's fetch() method (subclass-specific)

// Potential runtime error if animalRef doesn't point to a Dog object

Animal anotherAnimalRef = new Animal(); // Upcasting

Dog riskyDowncast = (Dog) anotherAnimalRef; // Risky downcast (may throw ClassCastException)

riskyDowncast.fetch(); // Potential ClassCastException if anotherAnimalRef doesn't refer to a Dog

Important Considerations:

* Downcasting should be used judiciously. Always ensure the object you're downcasting is actually of the intended subclass type. If not, a ClassCastException will be thrown at runtime.
* Upcasting is generally safer and often preferred for promoting code flexibility and abstraction.
* By understanding upcasting and downcasting effectively, you can leverage the power of inheritance and polymorphism in your Java applications while maintaining type safety and avoiding runtime errors.

Unit 4

Abstraction and Interfaces

Topics of this unit includes

1. Key Points Of Abstract Class & Interface,
2. Difference Between An Abstract Class & Interface,
3. Implementation Of Multiple Inheritance Through Interface.

# ABSTRACTION

1. Abstraction aims to hide complexity from users and show them only relevant information. For Example, if you’re driving a car, you don’t need to know about its internal workings.
   1. The same is true of Java classes. You can hide internal implementation details using abstract classes or interfaces. On the abstract level, you only need to define the method signatures (name and parameter list) and let each class implement them in their own way.
2. Abstraction is a process of hiding the implementation details and showing only functionality to the user.
   1. Another way, it shows only essential things to the user and hides the internal details, for Example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

# ABSTRACT CLASS

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

1. An abstract class must be declared with an abstract keyword.
2. An abstract class may or may not have all abstract methods. Some of them can be concrete methods.
3. It can have default constructors, parameterized constructors and static methods also.
4. It can have final methods which will force the subclass not to change the body of the method.
5. An abstract method is a method that is declared without implementation.
6. A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
7. It cannot be instantiated. There can be no object of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.

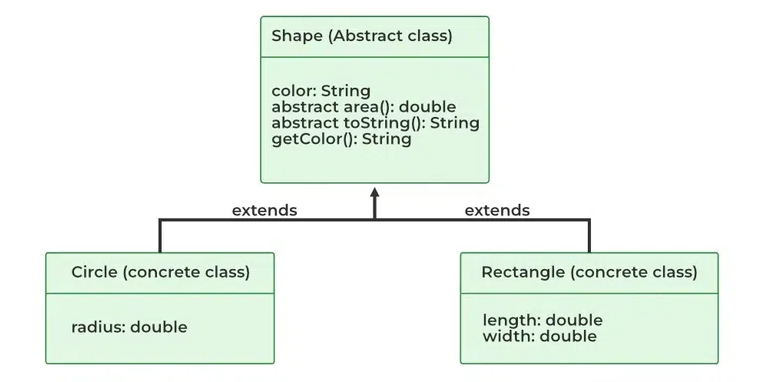
Algorithm to implement abstraction in Java

1. Determine the classes or interfaces that will be part of the abstraction.
2. Create an abstract class or interface that defines the common behaviours and properties of these classes.
3. Define abstract methods within the abstract class or interface that do not have any implementation details.
4. Implement concrete classes that extend the abstract class or implement the interface.
5. Override the abstract methods in the concrete classes to provide their specific implementations.
6. Use the concrete classes to implement the program logic.

When to use abstract classes and abstract methods?

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic “shape” Example, perhaps used in a computer-aided design system or game simulation. The base type is “shape” and each shape has a color, size, and so on. From this, specific types of shapes are derived(inherited)-circle, square, triangle, and so on — each of which may have additional characteristics and behaviours. For Example, certain shapes can be flipped. Some behaviours may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.



Example

abstract class { }

Abstract Method in Java

Abstract methods are declared using the "abstract" keyword and are followed by the method signature, which includes the method name, parameter types (if any), and the return type. However, the method body (the code that performs the actual functionality) is not provided.

Some rules

1. If there is an abstract method in a class, that class must be abstract.
2. If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

**Example**

**abstract void printStatus();**

**Example of abstract class**

abstract class Bike{

abstract void run();

}

class Honda4 extends Bike{

void run(){System.out.println("running safely");}

public static void main(String args[]){

Bike obj = new Honda4();

obj.run();

}

}

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

//Example of an abstract class that has abstract and non-abstract methods

abstract class Bike{

Bike(){System.out.println("bike is created");}

abstract void run();

void changeGear(){System.out.println("gear changed");}

}

//Creating a Child class which inherits Abstract class

class Honda extends Bike{

void run(){System.out.println("running safely..");}

}

//Creating a Test class which calls abstract and non-abstract methods

class TestAbstraction2{

public static void main(String args[]){

Bike obj = new Honda();

obj.run();

obj.changeGear();

}

}

Output

bike is created

running safely..

gear changed

Advantages of Abstraction

1. It reduces the complexity of viewing things.
2. Avoids code duplication and increases reusability.
3. Helps to increase the security of an application or program as only essential details are provided to the user.
4. It improves the maintainability of the application.
5. It improves the modularity of the application.
6. The enhancement will become very easy because without affecting end-users we can able to perform any type of changes in our internal system.
7. Improves code reusability and maintainability.
8. Hides implementation details and exposes only relevant information.
9. Provides a clear and simple interface to the user.
10. Increases security by preventing access to internal class details.
11. Supports modularity, as complex systems can be divided into smaller and more manageable parts.
12. Abstraction provides a way to hide the complexity of implementation details from the user, making it easier to understand and use.
13. Abstraction allows for flexibility in the implementation of a program, as changes to the underlying implementation details can be made without affecting the user-facing interface.
14. Abstraction enables modularity and separation of concerns, making code more maintainable and easier to debug.

Disadvantages of Abstraction in Java

1. Abstraction can make it more difficult to understand how the system works.
2. It can lead to increased complexity, especially if not used properly.
3. This may limit the flexibility of the implementation.
4. Abstraction can add unnecessary complexity to code if not used appropriately, leading to increased development time and effort.
5. Abstraction can make it harder to debug and understand code, particularly for those unfamiliar with the abstraction layers and implementation details.
6. Overuse of abstraction can result in decreased performance due to the additional layers of code and indirection.
7. What is the Difference between **Encapsulation and Data Abstraction?**

*Here are some key difference b/w encapsulation and abstration:*

|  |  |
| --- | --- |
| Encapsulation | Abstraction |
| Encapsulation is data hiding(information hiding) | Abstraction is detailed hiding(implementation hiding). |
| Encapsulation groups together data and methods that act upon the data | Data Abstraction deal with exposing the interface to the user and hiding the details of implementation |
| Encapsulated classes are Java classes that follow data hiding and abstraction | Implementation of abstraction is done using abstract classes and interface |
| Encapsulation is a procedure that takes place at the implementation level | abstraction is a design-level process |

1. What is a real-life Example of data abstraction?

***Television remote control****is an excellent real-life Example of abstraction. It simplifies the interaction with a TV by hiding the complexity behind simple buttons and symbols, making it easy without needing to understand the technical details of how the TV functions.*

1. What is the Difference between Abstract Classes and Interfaces in Java?

*Here are some key difference b/w Abstract Classes and Interfaces in Java:*

|  |  |
| --- | --- |
| Abstract Class | Interfaces |
| Abstract classes support abstract and Non-abstract methods | Interface supports have abstract methods only. |
| Doesn’t support Multiple Inheritance | Supports Multiple Inheritance |
| Abstract classes can be extended by Java classes and multiple interfaces | The interface can be extended by Java interface only. |
| Abstract class members in Java can be private, protected, etc. | Interfaces are public by default. |
| **Example:**  public abstract class Vechicle{ public abstract void drive() } | **Example:**  public interface Animal{ void speak(); } |

# Interfaces

An interface is a 100% abstract class. It can have **public**, **static**, **final** by default and **abstract methods**. It’s frequently referred to as a blueprint of a class as well. Java interfaces allow you to implement multiple inheritances in your code, as a class can implement any number of interfaces. Classes can access an interface with the implements keyword. A class that implements an interface must implement all the methods declared in the interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

1. By interface, we can support the functionality of multiple inheritance.
2. Interfaces are used to achieve abstraction by allowing you to define a contract without revealing the implementation details.
3. Interfaces can contain constants (final and static variables), abstract methods (without a body), default methods (with a body since Java 8), and static methods.
4. A class can implement multiple interfaces, allowing it to support multiple contracts or behaviors.
5. Interfaces cannot be instantiated; you can only create objects of classes that implement the interface.
6. Interfaces are useful for achieving loose coupling between classes, promoting code reusability, and enabling polymorphism.
7. If overriding an interface the method visibility is wider or equivalent then abstract.
8. Java.lang.Object is an implicit interface implemented by all classes in Java, providing basic functionalities like toString(), equals(), and hashCode().
9. Interfaces can be nested within other interfaces or classes.
10. Since Java 8, interfaces can have default methods, allowing for backward compatibility when updating an interface's behavior without breaking existing implementations.
11. Interfaces can also contain static methods, which can be invoked without creating an object of the class that implements the interface.
12. Interfaces can be extended (inherited) by other interfaces, creating a hierarchy of related interfaces.
13. Private methods can be implemented static or non-static. This means that in an interface we are able to create private methods to encapsulate code from both default and static public method signatures.

Benefits of Private Methods in Interfaces

Touched on in the previous section, interfaces are able to use private methods to hide details on implementation from classes that implement the interface. As a result, one of the main benefits of having these in interfaces is encapsulation.

Another benefit is (as with private methods in general) that there is less duplication and more re-usable code added to interfaces for methods with similar functionality.

Example

In the Example, we define two interfaces: Animal with two abstract methods (interface methods are abstract by default) and Bird with two static fields and an abstract method.

interface Animal {

public void eat();

public void sound();

}

interface Bird {

int numberOfLegs = 2;

String outerCovering = "feather";

public void fly();

}

The class Eagle implements both interfaces. It defines its own functionality for the three abstract methods. The eat() and sound() methods come from the Animal class, while fly() comes from Bird.

class Eagle implements Animal, Bird {

public void eat() {

System.out.println("Eats reptiles and amphibians.");

}

public void sound() {

System.out.println("Has a high-pitched whistling sound.");

}

public void fly() {

System.out.println("Flies up to 10,000 feet.");

} }

In the TestEagleInterfaces test class, we instantiate a new Eagle object (called myEagle) and print out all the fields and methods to the console.

As static fields (numberOfLegs and outerCovering) don’t belong to a specific object but to the interface, we need to access them from the Bird interface instead of the myEagle object.

class TestEagleInterfaces {

public static void main(String[] args) {

Eagle myEagle = new Eagle();

myEagle.eat();

myEagle.sound();

myEagle.fly();

System.out.println("Number of legs: " + Bird.numberOfLegs);

System.out.println("Outer covering: " + Bird.outerCovering);

} }

The Java console returns all the information we wanted to access:

[Console Output of TestEagleInterfaces]

Eats reptiles and amphibians.

Has a high-pitched whistling sound.

Flies up to 10,000 feet.

Number of legs: 2

Outer covering: feather

Differences between a Class and an Interface

**The following table lists all the major differences between an interface and a class in Java language:**

|  |  |
| --- | --- |
| Class | Interface |
| The keyword used to create a class is “class” | The keyword used to create an interface is “interface” |
| A class can be instantiated i.e., objects of a class can be created. | An Interface cannot be instantiated i.e. objects cannot be created. |
| Classes do not support multiple inheritance. | The interface supports multiple [inheritance.](https://www.geeksforgeeks.org/inheritance-in-java/) |
| It can be inherited from another class. | It cannot inherit a class. |
| It can be inherited by another class using the keyword ‘extends’. | It can be inherited by a class by using the keyword ‘implements’ and it can be inherited by an interface using the keyword ‘extends’. |
| It can contain constructors. | It cannot contain constructors. |
| It cannot contain abstract methods. | It contains abstract methods only. |
| Variables and methods in a class can be declared using any access specifier(public, private, default, protected). | All variables and methods in an interface are declared as public. |
| Variables in a class can be static, final, or neither. | All variables are static and final. |

# **Difference between abstract class and interface**

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| Abstract class | Interface |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

Example

//Creating interface that has 4 methods

interface A{

void a();//bydefault, public and abstract

void b();

void c();

void d();

}

//Creating abstract class that provides the implementation of one method of A interface

abstract class B implements A{

public void c(){System.out.println("I am C");}

}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods

class M extends B{

public void a(){System.out.println("I am a");}

public void b(){System.out.println("I am b");}

public void d(){System.out.println("I am d");}

}

//Creating a test class that calls the methods of A interface

class Test5{

public static void main(String args[]){

A a=new M();

a.a();

a.b();

a.c();

a.d();

}}

Implementation of Multiple Inheritance Using Interface

In Java, achieving true multiple inheritance with classes is not possible due to the "diamond problem." This problem arises when a subclass inherits from two parent classes that have the same method signature. The compiler wouldn't know which implementation to use, leading to ambiguity.

However, you can effectively achieve a similar functionality using interfaces. Here's how:

**1. Define Interfaces:**

* Create separate interfaces outlining the functionalities you want the implementing class to inherit. Each interface can define methods specific to its purpose.

interface Animal {

void makeSound();

}

interface Swimmer {

void swim();

}

**2. Implement Interfaces in a Class:**

* A class can implement multiple interfaces, inheriting the methods declared in each interface. The class needs to provide implementations for all inherited methods.

public class Dolphin implements Animal, Swimmer {

@Override

public void makeSound() {

System.out.println("Squeals and clicks!");

}

@Override

public void swim() {

System.out.println("Gliding through the water!");

}

}

**3. Using the Implementing Class:**

* You can create an object of the implementing class and call the inherited methods from any of the implemented interfaces.

Dolphin dolly = new Dolphin();

dolly.makeSound(); // Output: Squeals and clicks!

dolly.swim(); // Output: Gliding through the water!

Use code [with caution.](https://gemini.google.com/faq#coding)

**Benefits of using Interfaces for Multiple Inheritance:**

* **Clearer Separation of Concerns:** Interfaces promote better code organization by separating functionalities into well-defined contracts.
* **Improved Maintainability:** Changes to an interface affect all implementing classes, ensuring consistency.
* **Flexibility:** Classes can implement multiple interfaces, inheriting functionalities from various sources.
* **Avoids the Diamond Problem:** Interfaces prevent ambiguity that arises with multiple class inheritance.

**Remember:** Interfaces cannot have member variables, and methods are by default abstract (without implementation). The implementing class provides the concrete implementations for these methods.

# Encapsulation

1. A green and white pill

   Description automatically generatedEncapsulation in Java is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Java Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.
2. Encapsulation helps with data security, allowing you to protect the data stored in a class from system-wide access. As the name suggests, it safeguards the internal contents of a class like a capsule.

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.

**Encapsulation in Java:**

* Restricts direct access to data members (fields) of a class
* Fields are set to private
* Each field has a getter and setter method
* Getter methods return the field
* Setter methods let us change the value of the field

Example: how encapsulation works in practice

With encapsulation, you can protect the fields of a class. To do so, you need to declare the fields as private and provide access to them with getter and setter methods.

The Animal class below is fully encapsulated. It has three private fields, and each has its own pair of getter and setter methods.

class Animal {

private String name;

private double averageWeight;

private int numberOfLegs;

// Getter methods

public String getName() {

return name;

}

public double getAverageWeight() {

return averageWeight;

}

public int getNumberOfLegs() {

return numberOfLegs;

}

// Setter methods

public void setName(String name) {

this.name = name;

}

public void setAverageWeight(double averageWeight) {

this.averageWeight = averageWeight;

}

public void setNumberOfLegs(int numberOfLegs) {

this.numberOfLegs = numberOfLegs;

}

}

The TestAnimal class first creates a new Animal object (called myAnimal), then defines a value for each field with the setter methods, and finally prints out the values using the getter methods.

class TestAnimal {

public static void main(String[] args) {

Animal myAnimal = new Animal();

myAnimal.setName("Eagle");

myAnimal.setAverageWeight(1.5);

myAnimal.setNumberOfLegs(2);

System.out.println(" " + myAnimal.getName());

System.out.println("Average weight: " + myAnimal.getAverageWeight() + "kg");

System.out.println("Number of legs: " + myAnimal.getNumberOfLegs());

}

}

As you can see below, the Java console returns all the values we have set with the setter methods:

[Console Output of TestAnimal]

Eagle

Average weight: 1.5kg

Number of legs: 2

1. Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.
2. Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.
3. As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a combination of data-hiding and abstraction.
4. Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
5. It is more defined with the setter and getter method.

**Advantages of Encapsulation**

1. **Data Hiding:**it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
2. **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirements. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
3. **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
4. **Testing code is easy:** Encapsulated code is easy to test for unit testing.
5. **Freedom to programmer in implementing the details of the system:**This is one of the major advantage of encapsulation that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

Disadvantages of Encapsulation in Java

1. Can lead to increased complexity, especially if not used properly.
2. Can make it more difficult to understand how the system works.
3. May limit the flexibility of the implementation.

Example of Encapsulation

public class Encapsulation {

// Java program to demonstrate

// Java encapsulation

// private variables declared

// these can only be accessed by

// public methods of class

private String studentName;

private int studentRoll;

private int studentAge;

// get method for age to access

// private variable studentAge

public int getAge() { return studentAge; }

// get method for name to access

// private variable studentName

public String getName() { return studentName; }

// get method for roll to access

// private variable studentRoll

public int getRoll() { return studentRoll; }

// set method for age to access

// private variable studentage

public void setAge(int newAge) { studentAge = newAge; }

// set method for name to access

// private variable studentName

public void setName(String newName)

{

studentName = newName;

}

// set method for roll to access

// private variable studentRoll

public void setRoll(int newRoll) { studentRoll = newRoll; }

}

class TestEncapsulation {

public static void main(String[] args)

{

Encapsulation obj = new Encapsulation();

// setting values of the variables

obj.setName("Harsh");

obj.setAge(19);

obj.setRoll(51);

// Displaying values of the variables

System.out.println("Student " + obj.getName());

System.out.println("Student age: " + obj.getAge());

System.out.println("Student roll: " + obj.getRoll());

// Direct access of studentRoll is not possible

// due to encapsulation

// System.out.println("student's roll: " +

// obj.studentName);

}

}

Output

Student Harsh

Student age: 19

Student roll: 51

Polymorphism

In Java, polymorphism refers to the ability of objects of different classes to respond to the same method call in distinct ways. It's a powerful concept that enhances code flexibility and promotes code reusability. There are two main types of polymorphism in Java:

1. Compile-Time Polymorphism / Static Polymorphism (Method Overloading):

1. Occurs at compile time, when the Java compiler determines the appropriate method to call based on the number and/or data types of the arguments provided in the method call.
2. Involves defining multiple methods within the same class that share the same name but have different parameter lists (number, types, or order of arguments).
3. The compiler selects the most specific method that matches the arguments used in the call.

Example:

class MathOperations {

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

}

Explanation

In this example, the add method is overloaded to handle both integer and double addition. When you call add(5, 3), the compiler chooses the first add method for integers. Similarly, add(3.14, 2.71) would use the second add method for doubles.

2. Runtime Polymorphism / Dynamic Polymorphism (Method Overriding):

1. Occurs at runtime, when the actual object's type determines the method to be invoked.
2. Achieved through inheritance: a subclass (derived class) redefines a method inherited from its parent class (base class).
3. When you call the overridden method using a reference variable of the parent class type that points to an object of the subclass, the subclass's implementation is executed at runtime (dynamic binding).

Example:

class Animal {

public void makeSound() {

System.out.println("Generic animal sound");

}}

class Dog extends Animal {

@Override // Optional annotation to indicate overriding

public void makeSound() {

System.out.println("Woof!");

}

}

Explanation

Here, the makeSound method is overridden in the Dog class. When you call makeSound on an Animal reference variable that refers to a Dog object, the Dog class's makeSound implementation is called at runtime, resulting in "Woof!" being printed.

Key Points:

1. Method overloading promotes flexibility in how methods are called based on arguments, while method overriding allows for specialized behavior in subclasses.
2. Inheritance is a prerequisite for method overriding.
3. Polymorphism enables you to write generic code that can work with objects of different classes if they share a common base class or interface.

note: By effectively using these polymorphism techniques, you can create more adaptable and maintainable Java code.

Unit 5

Exception Handling

Topics of exception handling

1. Definition Of Exception Handling
2. Implementation Of Keywords Like Try, Catch, Finally, Throws And Throws.
3. Importance Of Exception Handling In Practical Implementation Of Live Projects.

Exception handling

Exception handling is a crucial mechanism in Java (and most programming languages) for managing unexpected situations that may arise during program execution. These unexpected situations are called exceptions.

**1. What are Exceptions?**

Exceptions are runtime errors or unexpected events that can disrupt the normal flow of your program. They can occur due to various reasons, such as:

* User input errors (e.g., entering an invalid number when expecting an integer)
* File access issues (e.g., trying to read a non-existent file)
* Network problems (e.g., loss of connection while communicating over a network)
* Internal errors within the program (e.g., attempting to divide by zero)

**2. Why is Exception Handling Important?**

Without proper exception handling, these errors can cause your program to crash abruptly, terminate unexpectedly, or even produce incorrect results. Exception handling allows you to:

* **Gracefully handle errors:** You can provide informative error messages to the user or take corrective actions when exceptions occur.
* **Improve program stability:** By anticipating and handling potential exceptions, you prevent program crashes and make your code more robust.
* **Enhance code maintainability:** Exception handling code clearly separates normal program flow from error handling logic, making code easier to understand and maintain.

**3. Core Concepts of Exception Handling:**

* **Try-Catch Block:** This is the fundamental structure for handling exceptions.  
  + The try block contains the code that might potentially throw an exception.
  + The catch block(s) specify how to handle exceptions of specific types or a broader category (all exceptions).
* **Throwing Exceptions:** When an error condition occurs, you can explicitly throw an exception using the throw keyword followed by the exception object. This signals the problem to a higher level in the program.
* **Exception Hierarchy:** Java provides a built-in hierarchy of exception classes. Specific exceptions inherit from more general ones, allowing you to catch exceptions at different levels of detail.

**4. Example:**

public class Calculator {

public static int divide(int numerator, int denominator) {

try {

// Code that might throw an exception (division by zero)

return numerator / denominator;

} catch (ArithmeticException e) {

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

return -1; // Indicate an error condition (optional)

}

}

public static void main(String[] args) {

int result = divide(10, 2);

System.out.println("Result: " + result); // Output: Result: 5

result = divide(10, 0);

System.out.println("Result: " + result); // Output: Error: Cannot divide by zero!

// Result: -1 (optional)

}

}

In this example:

* The divide method performs division.
* The try block attempts the division.
* The catch block specifically handles ArithmeticException (thrown when dividing by zero).
* If division by zero occurs, the catch block prints an error message and optionally returns an error code (-1).
* The main method demonstrates calling divide with valid and invalid inputs.

Keywords of Exception Handling

* **Try Block:** This block contains the code that might throw an exception.
* **Catch Block(s):** These blocks define how to handle exceptions of specific types or a broader category (all exceptions). Each catch block is associated with an exception class.
* **Finally Block (Optional):** This block executes regardless of whether an exception occurs or not. It's typically used for cleanup tasks like closing files or releasing resources.
* **Throw Keyword:** This keyword is used to explicitly throw an exception object from a method, signaling an error condition to a higher level in the program.
* **Throws Keyword:** This keyword is used in method signatures to declare the exceptions that a method might throw. It informs callers about potential exceptions to handle.

Explained With Example

**a) try Block:**

try {

// Code that might throw an exception

}

Use code [with caution.](https://gemini.google.com/faq#coding)

* The try block identifies the code section that could potentially throw an exception.
* If no exception occurs within the try block, the program continues execution normally.

**b) catch Block(s):**

try {

// Code that might throw an exception

} catch (ExceptionType1 e) {

// Handle exceptions of type ExceptionType1

} catch (ExceptionType2 e) {

// Handle exceptions of type ExceptionType2

}

* A catch block follows a try block and specifies how to handle exceptions of a particular type (like ArithmeticException).
* You can have multiple catch blocks to handle different exception types.
* The e parameter in the catch block is an exception object that provides information about the error.
* The code within the catch block defines the actions to take when the corresponding exception type is thrown within the try block.

**c) finally Block (Optional):**

try {

// Code that might throw an exception

} catch (ExceptionType e) {

// Handle exceptions

} finally {

// Code that always executes (cleanup)

}

* The finally block is optional but can be very useful.
* It executes **always**, regardless of whether an exception occurs or not, and even if the program exits abruptly due to an unhandled exception.
* This is typically used for resource management tasks like closing files, database connections, or releasing network resources to prevent resource leaks.

**d) throw Keyword:**

public int divide(int numerator, int denominator) {

if (denominator == 0) {

throw new ArithmeticException("Division by zero!");

}

// ... perform division

}

* The throw keyword is used to explicitly throw an exception object from a method.
* This signals an error condition to the calling code.
* You can throw any exception object, including custom exceptions you create.

**e) throws Keyword:**

public int divide(int numerator, int denominator) throws ArithmeticException {

// ... perform division

}

* The throws keyword is used in a method signature to declare the exceptions that a method might throw.
* This informs the caller about potential exceptions that need to be handled.
* If an exception is thrown within the method, it propagates up the call stack until a catch block for that exception type is found.

**4. Putting It Together:**

public class Calculator {

public static int divide(int numerator, int denominator) throws ArithmeticException {

try {

return numerator / denominator;

} catch (ArithmeticException e) {

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

return -1; // Indicate error (optional)

} finally {

System.out.println("Division operation completed."); // Always executes

}

}

public static void main(String[] args) {

try {

int result = divide(10, 2);

System.out.println("Result: " + result); // Output: Result: 5

} catch (ArithmeticException e) {

// Handle exception if division by zero occurs in main

Difference Between Final, Finally and Finalizer

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| Key | final | finally | finalize |
| Definition | final is the keyword and access modifier which is used to apply restrictions on a class, method or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java which is used to perform clean up processing just before object is garbage collected. |
| Applicable to | Final keyword is used with the classes, methods and variables. | Finally block is always related to the try and catch block in exception handling. | finalize() method is used with the objects. |
| Functionality | * Once declared, final variable becomes constant and cannot be modified. * final method cannot be overridden by sub class. * final class cannot be inherited. | * finally block runs the important code even if exception occurs or not. * finally block cleans up all the resources used in try block | * finalize method performs the cleaning activities with respect to the object before its destruction. |
| Execution | Final method is executed only when we call it. | * Finally block is executed as soon as the try-catch block is executed. * It's execution is not dependant on the exception. | finalize method is executed just before the object is destroyed. |

Difference between throw and throws

The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method.

There are many differences between [throw](https://www.javatpoint.com/throw-keyword) and [throws](https://www.javatpoint.com/throws-keyword-and-difference-between-throw-and-throws) keywords. A list of differences between throw and throws are given below:

|  |  |  |
| --- | --- | --- |
| Basis of Differences | throw | throws |
| Definition | Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code. | Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code. |
| Keyword | Type of exception Using **throw** keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only. | Using throws keyword, we can declare both checked and unchecked exceptions. However, the **throws** keyword can be used to propagate checked exceptions only. |
| Syntax | The throw keyword is followed by an instance of Exception to be thrown. | The throws keyword is followed by class names of Exceptions to be thrown. |
| Declaration | throw is used within the method. | throws is used with the method signature. |
| Internal implementation | We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions. | We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException. |

Example of Throw keyword

public class TestThrow {

//defining a method

public static void checkNum(int num) {

if (num < 1) {

throw new ArithmeticException("\nNumber is negative, cannot calculate square");

}

else {

System.out.println("Square of " + num + " is " + (num\*num));

}

}

//main method

public static void main(String[] args) {

TestThrow obj = new TestThrow();

obj.checkNum(-3);

System.out.println("Rest of the code..");

}

}

Example of Throws

public class TestThrows {

//defining a method

public static int divideNum(int m, int n) throws ArithmeticException {

int div = m / n;

return div;

}

//main method

public static void main(String[] args) {

TestThrows obj = new TestThrows();

try {

System.out.println(obj.divideNum(45, 0));

}

catch (ArithmeticException e){

System.out.println("\nNumber cannot be divided by 0");

}

System.out.println("Rest of the code..");

}

}

Explaining Try-catch Block with program

note – To run this program just copy the code and create a file ATM.java and copy the entire code and run it. 😊

ATM Machine Program

package in.GeneralPrograms;

import java.util.InputMismatchException;

import java.util.Scanner;

public class ATM {

private int balance = 10000; *// Initial balance*

private Scanner scanner = new Scanner(System.in);

public static void main(String[] args) {

ATM atm = new ATM();

atm.run();

}

public void run() {

int choice;

do {

System.out.println("\n\*\* ATM Machine \*\*");

System.out.println("1. Check Balance");

System.out.println("2. Withdraw Cash");

System.out.println("3. Deposit Cash");

System.out.println("4. Exit");

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

try {

choice = scanner.nextInt();

} catch (InputMismatchException e) {

System.err.println("Invalid input. Please enter a number (1-4).");

scanner.nextLine(); *// Clear invalid input*

choice = -1; *// Set choice to invalid value to avoid processing*

}

switch (choice) {

case 1:

checkBalance();

break;

case 2:

withdrawCash();

break;

case 3:

depositCash();

break;

case 4:

System.out.println("Thank you for using ATM.");

break;

default:

System.out.println("Invalid choice. Please try again.");

}

} while (choice != 4);

}

private void checkBalance() {

System.out.println("Your current balance is: $" + balance);

}

private void withdrawCash() {

System.out.print("Enter amount to withdraw: $");

int amount;

try {

amount = scanner.nextInt();

} catch (InputMismatchException e) {

System.err.println("Invalid input. Please enter a number.");

scanner.nextLine(); *// Clear invalid input*

return; *// Exit withdraw operation on invalid input*

}

if (amount > balance) {

System.out.println("Insufficient funds. You cannot withdraw more than $" + balance);

} else {

balance -= amount;

System.out.println("Withdrawn $" + amount + ". New balance: $" + balance);

}

}

private void depositCash() {

System.out.print("Enter amount to deposit: $");

int amount;

try {

amount = scanner.nextInt();

} catch (InputMismatchException e) {

System.err.println("Invalid input. Please enter a number.");

scanner.nextLine(); *// Clear invalid input*

return; *// Exit deposit operation on invalid input*

}

if (amount <= 0) {

System.out.println("Invalid amount. Please enter a positive value.");

} else {

balance += amount;

System.out.println("Deposited $" + amount + ". New balance: $" + balance);

}

}

}

Explanation

* The program uses a **Scanner** object to get user input.
* A **do-while** loop keeps the ATM menu running until the user chooses to exit (option 4).
* A **try-catch** block is used within the menu selection (**choice**) to handle potential **InputMismatchException** if the user enters a non-numeric value.
* Each menu option has a separate method for better organization.
* The **withdrawCash** method checks for insufficient funds before withdrawing and throws an informative message.
* The **depositCash** method validates the deposit amount to ensure it's positive.
* Scanner resources are not explicitly closed in this example for simplicity, but consider using a **try-with-resources** statement for production code to ensure proper resource management.

This program demonstrates basic functionalities of an ATM with exception handling for user input validation and handling scenarios like insufficient funds. You can further enhance it with features like PIN verification, transaction history, and error handling for other potential exceptions.