Introduction to Java

- Java is a **high level, object-oriented and platform independent** programming language that is designed to have as few implementation dependencies as possible.

- Java is developed by **James Arthur Gosling** at **Sun MicroSystems** and released in **1995**. Later, It was acquired by Oracle Corporation and is widely used for developing applications for desktop, web, and mobile devices.

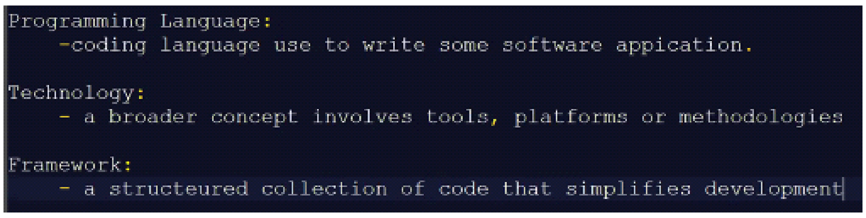
- Java is so popular because it is a **platform independent language, versatile language(used for wide range of applications), largest community support, simplicity, many opportunities available for Java developers in industry.**

- Java follows the principle of **"write once run anywhere" (WORA),** meaning programs can run on any platform with a **Java Virtual Machine(JVM).** Java code can run on all platforms that support Java without the need for recompilation.

- Java makes **writing, compiling, and debugging programming easy**. It helps to **create reusable code** and **modular programs**.

- Java has **both interpreter and compiler**. Java source code is compiled to produce a platform independent bytecode and JVM then interprets this bytecode to execute the code.

- Java is an object oriented language but it is **not purely object oriented language** as it supports the primitive data types as well not just classes and objects.



```
Programming Language:
    -coding language use to write some software appication.

Technology:
    - a broader concept involves tools, platforms or methodologies

Framework:
    - a structeured collection of code that simplifies development
```

Features of Java

- **<u>Platform Independent</u>:** Compiler converts source code to byte code and then the JVM executes the bytecode generated by the compiler. This byte code can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of the byte code. That is why we call java a platform-independent language. Give a real-time example like Minecraft or candy crush saga.

- **<u>Object-Oriented Programming</u>:** Java is an object-oriented language, promoting the use of objects and classes. Organizing the program in the terms of a collection of objects is a way of object-oriented programming, each of which represents an instance of the class. The four main concepts of Object-Oriented programming are:
    - Abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism

- **<u>Simplicity</u>:** Java's syntax is simple and easy to learn, especially for those familiar with C or C++. It eliminates complex features like pointers and multiple inheritances, making it easier to write, debug, and maintain code. It provides hardware abstraction.

- **<u>Robustness</u>:** Java language is robust which means it is a reliable language. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, exception handling, and memory allocation. It prevents the system from collapsing/crashing with help of concepts like exception handling, garbage collector.

Features of Java

- **<u>Security</u>:** In java, we don’t have pointers, so we cannot access out-of-bound arrays i.e. it shows ArrayIndexOutOfBound Exception if we try to do so. That’s why several security flaws like stack corruption or buffer overflow are impossible to exploit in Java. Also, java programs run in an environment that is independent of the os(operating system) environment which makes java programs more secure. It uses garbage collector to dereference the unused instances, making java a secure language.

- **<u>Distributed</u>:** We can create distributed applications using the java programming language. Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java.

- **<u>Multithreading</u>:** Java supports multithreading, enabling the concurrent execution of multiple parts of a program. This feature is particularly useful for applications that require high performance, such as games and real-time simulations.

- **<u>High Performance</u>:**
  Java architecture is defined in such a way that it reduces overhead during the runtime and sometimes java uses Just In Time (JIT) compiler where the compiler compiles code on-demand basis where it only compiles those methods that are called making applications to execute faster.

- **<u>Portable</u>:**
  Java code is portable as java byte code can be executed on various platforms irrespective of the platform on which it is written.

Java 1.8 (Java 8) Features

Java 1.8 (also known as Java 8) was a major release by Oracle, introducing several significant features that enhanced the language and its capabilities. Some of the key features of Java 8 include:

- **Lambda expressions:** Lambda expressions provide a clear and concise way to represent a method of a functional interface using an expression. They enable functional programming in Java and are particularly useful in the collection library to iterate, filter, and extract data.
- **Functional interface:** A functional interface is an interface that contains only one abstract method. Java 8 introduced several built-in functional interfaces such as Consumer, Predicate, Supplier and Function.
- **Stream API:** The Stream API allows functional-style operations on collections of objects. It supports operations like filtering, mapping, and reducing, and can be executed in parallel to improve performance
- **Default and Static Methods in Interfaces:** Java 8 allows interfaces to have default and static methods. Default methods provide a way to add new methods to interfaces without breaking existing implementations.
- **Date and Time API:** Java 8 introduced a new Date and Time API in the java.time package. It provides a comprehensive set of classes for date and time manipulation, such as LocalDate, LocalTime, and LocalDateTime.
- **Optional Class:** The Optional class is used to handle null values more gracefully. It provides methods to check the presence of a value and perform actions accordingly.

Java 1.8 (Java 8) Features

- **Nashorn JavaScript Engine:** Nashorn is a JavaScript engine that allows Java applications to execute JavaScript code dynamically. It can be used via the jjs command-line tool or embedded into Java source code.

- **Collectors Class:** The Collectors class provides various methods to perform reduction operations on streams, such as accumulating elements into collections and summarizing elements.

- **Method References:** Method references provide a shorthand notation for calling methods. They are a more readable alternative to lambda expressions when the lambda expression only calls an existing method.

Java 11 Features

- **Oracle vs. Open JDK:** Java 10 was the last free Oracle JDK release that we could use commercially without a license. Starting with Java 11, there's no free long-term support (LTS) from Oracle. Thankfully, Oracle continues to provide Open JDK releases, which we can download and use without charge.

- **New String methods:** Java 11 adds a few new methods to the String class: isBlank, lines, strip, stripLeading, stripTrailing, and repeat. These methods can reduce the amount of boilerplate involved in manipulating string objects, and save us from having to import libraries.
  In the case of the *strip* methods, they provide similar functionality to the more familiar *trim* method; however, with finer control and Unicode support.

- **New File Methods:** Additionally, it's now easier to read and write Strings from files. We can use the new readString and writeString static methods from the Files class.

- **Collection to an Array:** The java.util.Collection interface contains a new default toArray method which takes an IntFunction argument. This makes it easier to create an array of the right type from a collection.

- **The Not Predicate Method:** A static not method has been added to the Predicate interface. We can use it to negate an existing predicate, much like the negate method.

- **Local-Variable Syntax for Lambda:** Support for using the local variable syntax (var keyword) in lambda parameters was added in Java 11.We can make use of this feature to apply modifiers to our local variables, like defining a type annotation.

Java 11 Features

- **HTTP Client:** The new HTTP client from the java.net.http package was introduced in Java 9. It has now become a standard feature in Java 11. The new HTTP API improves overall performance and provides support for both HTTP/1.1 and HTTP/2

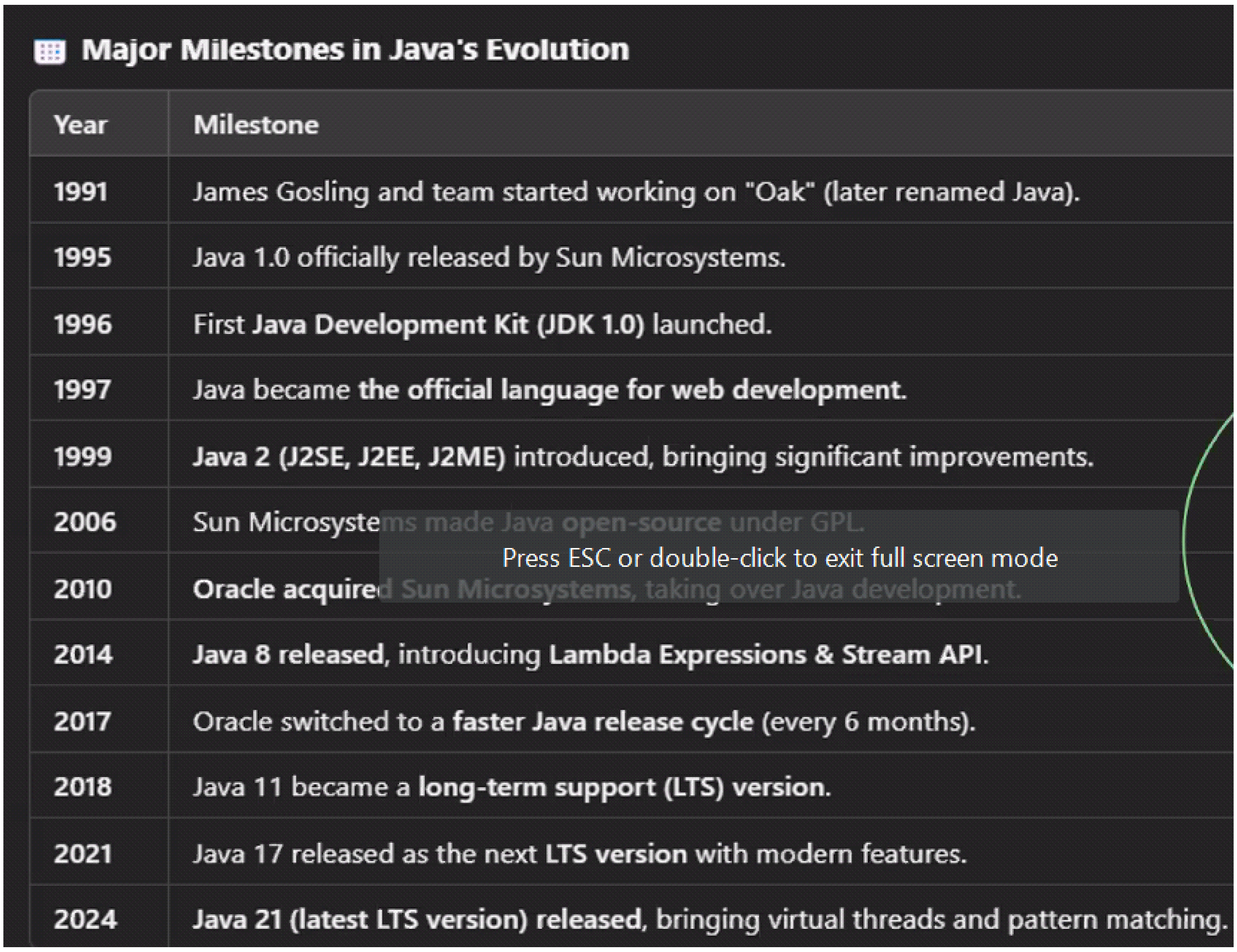
- **Nest Based Access Control:** Java 11 introduces the notion of nestmates and the associated access rules within the JVM. A nest of classes in Java implies both the outer/main class and all its nested classes. Nested classes are linked to the NestMembers attribute, while the outer class is linked to the NestHost attribute. JVM access rules allow access to private members between nestmates; however, in previous Java versions, the reflection API denied the same access. Java 11 fixes this issue and provides means to query the new class file attributes using the reflection API.

- **Running Java Files:** A major change in this version is that we don't need to compile the Java source files with javac explicitly anymore.

- **Epsilon Garbage Collector:** This handles memory allocation but does not have an actual memory reclamation mechanism. Once the available Java heap is exhausted, JVM will shut down. Its goals are Performance testing, Memory pressure testing and last drop latency improvements.

-

Major milestones in Java Evolution



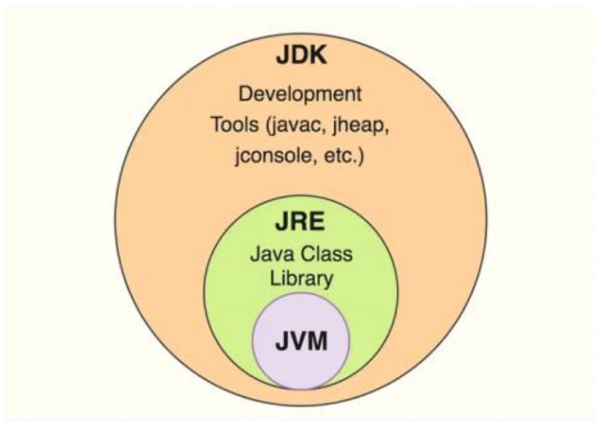
Major Milestones in Java's Evolution

Year	Milestone
1991	James Gosling and team started working on "Oak" (later renamed Java).
1995	Java 1.0 officially released by Sun Microsystems.
1996	First **Java Development Kit (JDK 1.0)** launched.
1997	Java became **the official language for web development.**
1999	**Java 2 (J2SE, J2EE, J2ME)** introduced, bringing significant improvements.
2006	Sun Microsystems made Java **open-source** under GPL.
2010	**Oracle acquired Sun Microsystems**, taking over Java development.
2014	**Java 8 released**, introducing **Lambda Expressions & Stream API.**
2017	Oracle switched to a **faster Java release cycle** (every 6 months).
2018	Java 11 became a **long-term support (LTS) version.**
2021	Java 17 released as the next **LTS version** with modern features.
2024	**Java 21 (latest LTS version) released**, bringing virtual threads and pattern matching.

Press ESC or double-click to exit full screen mode

Java Architecture (JVM, JDK, JRE)

- Java follows a "Write Once, Run Anywhere" (WORA) principle, meaning Java applications can run on any platform without modification. This is possible due to Java Architecture, which consists of three main components:
  - Java Development Kit (JDK)
  - Java Runtime Environment (JRE)
  - Java Virtual Machine (JVM)



- **<u>JDK</u>:** It is a software development environment that is used to develop java applications. The JDK is a complete package that allows developers to write, compile, and debug Java applications. It includes the JRE, along with development tools.

Component	**Description**
JRE (Java Runtime Environment)	Required to run Java applications.
Compiler (javac)	Converts Java source code (.java) into bytecode (.class).
Java Virtual Machine (JVM)	It executes the compiled bytecode.
Java Runtime Environment (JRE)	It includes libraries and JVM which are necessary to run a java applications.
Debugger (jdb)	Helps debug Java programs.
JavaDoc (javadoc)	Generates documentation from Java comments.
Java Archive (jar)	Packages multiple .class files into a single .jar file.
API libraries	Predefined classes and methods for java development
Other Development Tools	Profilers, monitoring tools, jheap, jconsole, etc.

Java Architecture (JVM, JDK, JRE)

- **JRE:** JRE is a part of JDK and it builds a runtime environment where the Java program can be executed. It contains the libraries and software needed by the Java programs to run. It takes the Java code and integrates with the required libraries, and then starts the JVM to execute it. Based on the Operating System, JRE will deploy the relevant code of the JVM. The JRE provides everything needed to run Java applications but does not include development tools like a compiler.

Component	Description
JVM (Java Virtual Machine)	Executes Java bytecode.
Core Libraries (rt.jar)	Essential Java class libraries (e.g., java.lang, java.util).
Java ClassLoader	Loads Java classes into memory.
Garbage Collector	Manages memory automatically.

- **JRE vs JDK vs JVM**
  - JDK = JRE + Development Tools (compiler, debugger, Interpreter, etc.)
  - JRE = JVM + Core Libraries (Only for running Java apps), runtime files (no compiler)
  - JVM = runs java applications by converting bytecode to machine code.

- **JVM:**
  - JVM is the core of Java’s architecture. It is responsible for loading, verifying, and executing Java bytecode.

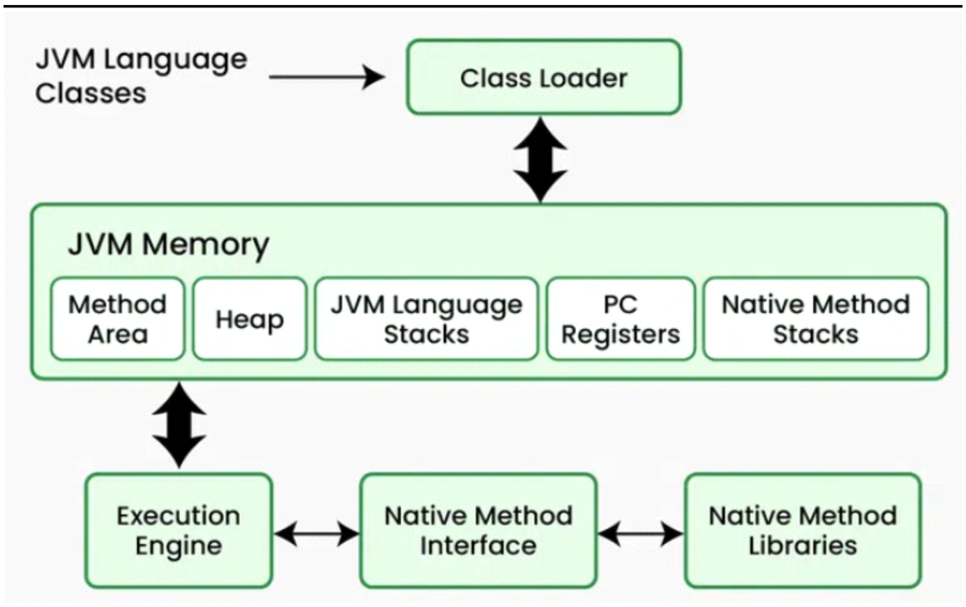
  - It serves as a bridge between Java programs and the underlying operating system.

  - It is an engine that provides a run-time environment to run the Java applications and it is part of JRE.

  - It runs java applications by converting bytecode to machine code.

Java Architecture (JVM, JDK, JRE)

- Java uses the combination of both (compiler and interpreter). source code (.java file) is first compiled into byte code and generates a class file (.class file). Then JVM converts the compiled binary byte code into a specific machine language. In the end, JVM is a specification for a software program that executes code and provides the runtime environment for that code.



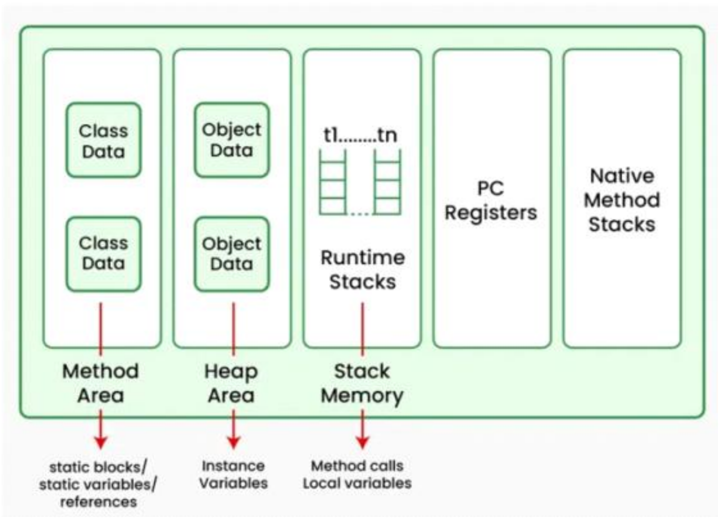
- JVM consists of three main subsystems:
  - Class Loader Subsystem
  - JVM Memory Areas
  - Native Method Interface
  - Execution Engine
  - Native Method Libraries

- <u>**The Class Loader subsystem**</u>**:**
  - It is responsible for loading Java bytecode (.class files) into JVM memory when a program starts. Three Types of Class Loaders:
    - <u>Bootstrap ClassLoader</u> – Loads core Java classes from rt.jar (like java.lang.Object).
    - <u>Extension ClassLoader</u> – Loads classes from the ext directory (java.ext.dirs).
    - <u>Application ClassLoade</u>r – Loads application-specific classes from the classpath.

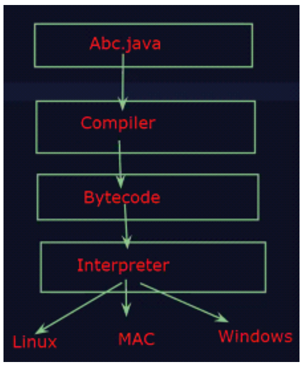
- **Class Loading Process:**
  - **Loading:** Reads .class files from disk or network.
  - **Linking:**
    - Verification: Ensures bytecode follows Java standards.
    - Preparation: Allocates memory for static variables.
    - Resolution: Converts symbolic references to actual memory locations.
  - **Initialization:** Executes static initializers and assigns values.

- **<u>JVM Memory Areas</u>:**
  - JVM divides memory into different areas:

Memory Area	Description
Method Area	Stores class metadata, static variables, references and method code.
Heap	Stores objects and instance variables (shared across threads).
Stack	Stores method call frames and local variables (separate for each thread).
PC Register	Holds the address of the currently executing instruction.
Native Method Stack	Manages native (non-Java) method calls.



- **Execution Engine:**
  - Execution engine executes the ".class" (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:
    - **Interpreter:** It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
    - **Just-In-Time Compiler(JIT):** It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native/machine code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.
    - **Garbage Collector:** It destroys un-referenced objects. Automatically manages memory by reclaiming unused objects.
- **Java Native Interface (JNI):** It is an interface that interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.
- **Java Method Libraries:** These are collections of native libraries required for executing native methods. They include libraries written in languages like C and C++.



Java Development platforms

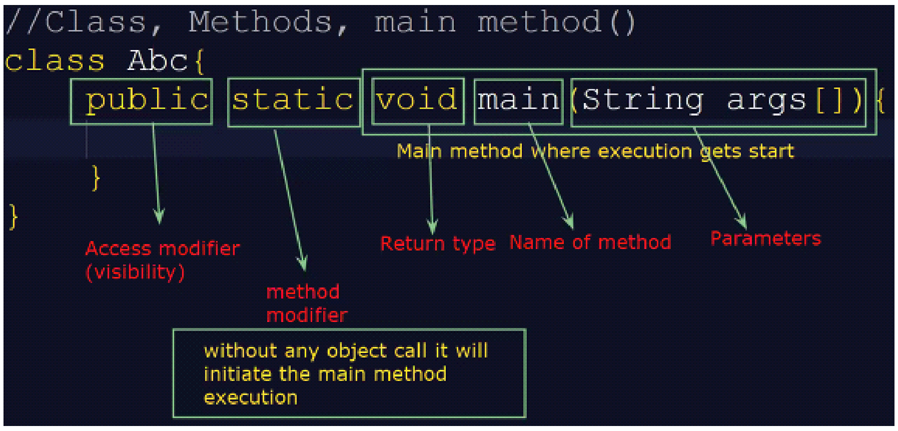
- **J2SE- Java Platform Standard Edition: (local purpose, used by learners)**
  - It has concepts for developing software for Desktop based (standalone) CUI (command user interface) and GUI (graphical user interface) applications, applets, database Interaction application, distributed application, and XML parsing applications.

- **J2EE -Java Platform Enterprise Edition: (Advanced Java, Java for Web development, corporate purpose)**
  - It has concepts to develop software for Web applications, Enterprise applications, and Interoperable applications. These applications are called high-scale applications. Some examples are:- banking and insurance-based applications.

- **J2ME - Java Platform Micro Edition: (Android development, Service Platform)**
  - It has concepts to develop software for consumer electronics, like mobile and electronic level applications. Java ME was popular for developing mobile gaming applications. This edition was called micro because these edition programs are embedded in small chips. The program embedded in the chip is called micro (small).

- **JavaFX - Java Platform Effects:**
  - Used for creating rich internet application (where multimedia is used), designing lightweight user interface applications.
  - Java FX stands for Java platform Effects (Eff=F, ects=X). It provides concepts for developing rich internet applications with more graphics and animations. It’s an extension concept to swing applications of Java SE. The Java FX API is included as part of Java SE software. Just by installing Java SE, we will also get Java FX API.

- **Main method Signature:**



```
//Class, Methods, main method()
class Abc{
    public static void main(String args[]){
    }
}
```

Main method where execution gets start
Access modifier (visibility)
method modifier
Return type
Name of method
Parameters
without any object call it will initiate the main method execution

- **public:** It is an Access modifier, which specifies from where and who can access the method. Making the main() method public makes it globally available. It is made public so that JVM can invoke it from outside the class as it is not present in the current class. If the main method is not public, it's access is restricted.

- **static:**
  - It is a keyword that is when associated with a method, making it a class-related method. The main() method is static so that JVM can invoke it without instantiating the class (i.e. Without creating an object of a class).
  - This also saves the unnecessary wastage of memory which would have been used by the object declared only for calling the main() method by the JVM. If you try to run Java code where main is not static, you will get an error.

- **void:**
  - It is a keyword and is used to specify that a method doesn't return anything.
  - As the main() method doesn't return anything, its return type is void. As soon as the main() method terminates, the Java program terminates too.
  - Hence, it doesn't make any sense to return from the main() method as JVM can't do anything with its return value of it.

Main method signature

- **main:**
  - It is the name of the Java main method. It is the identifier that the JVM looks for as the starting point of the Java program. It's not a keyword.
  - If we change the name while initiating main method, we will get an error.

- **String[] args**
  - It stores Java command-line arguments and is an array of type java.lang.String class.
  - Here, the name of the String array is args but it is not fixed and the user can use any name in place of it.

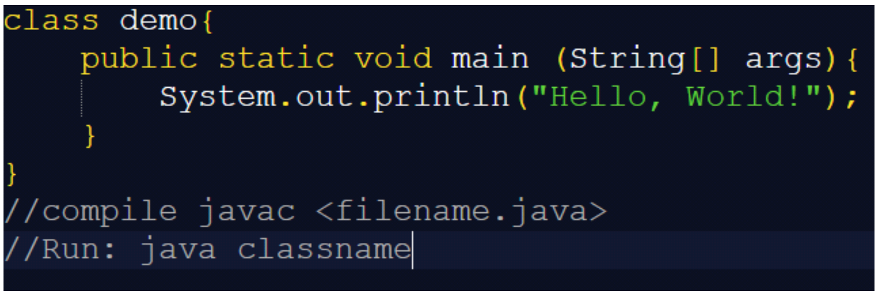
- In this signature, we can change some things.
  - We can interchange words public and static.

    static public void main(String[] args){ }
  - We can put [] (subscript operator) in front of String or args.

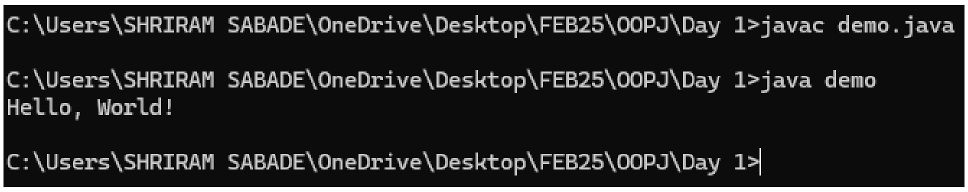
    public static void main(String args[]){ }
  - We can change the identifier of String array in the definition.

    public static void main(String[] cdac){ }

- **Pilot Program:**

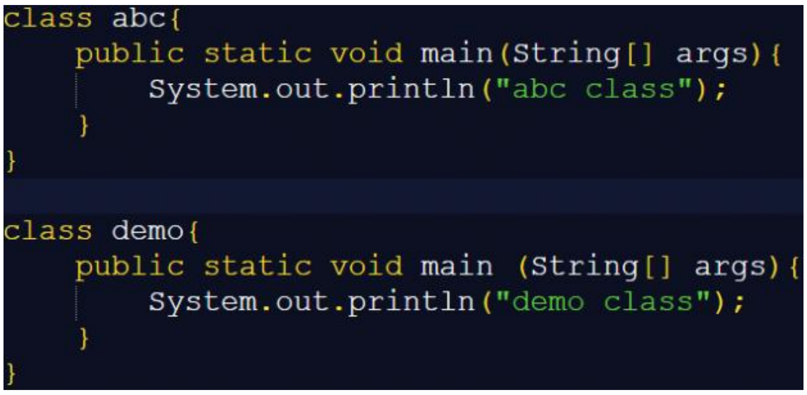


```
class demo{
    public static void main (String[] args){
        System.out.println("Hello, World!");
    }
}
//compile javac <filename.java>
//Run: java classname
```

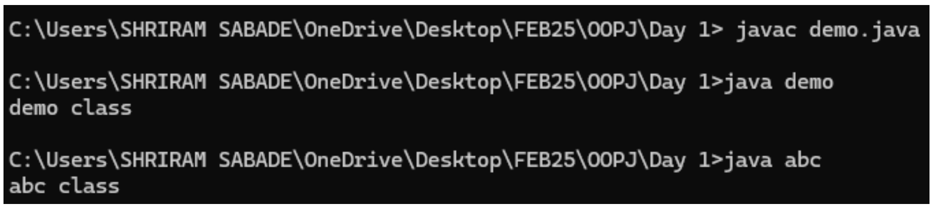


```
C:\Users\SHRIRAM SABADE\OneDrive\Desktop\FEB25\OOPJ\Day 1>javac demo.java

C:\Users\SHRIRAM SABADE\OneDrive\Desktop\FEB25\OOPJ\Day 1>java demo
Hello, World!

C:\Users\SHRIRAM SABADE\OneDrive\Desktop\FEB25\OOPJ\Day 1>
```

- **You can define multiple classes in single java file. If class to be executed doesn't have a similar name as filename, make sure proper class name while running the bytecode.**



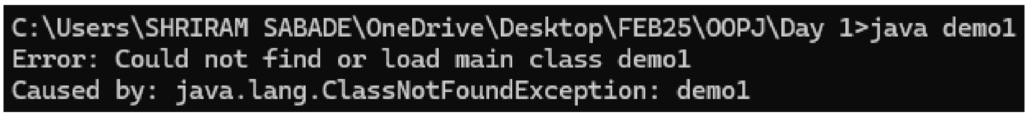
```
class abc{
    public static void main(String[] args){
        System.out.println("abc class");
    }
}

class demo{
    public static void main (String[] args){
        System.out.println("demo class");
    }
}
```



```
C:\Users\SHRIRAM SABADE\OneDrive\Desktop\FEB25\OOPJ\Day 1> javac demo.java

C:\Users\SHRIRAM SABADE\OneDrive\Desktop\FEB25\OOPJ\Day 1>java demo
demo class

C:\Users\SHRIRAM SABADE\OneDrive\Desktop\FEB25\OOPJ\Day 1>java abc
abc class
```

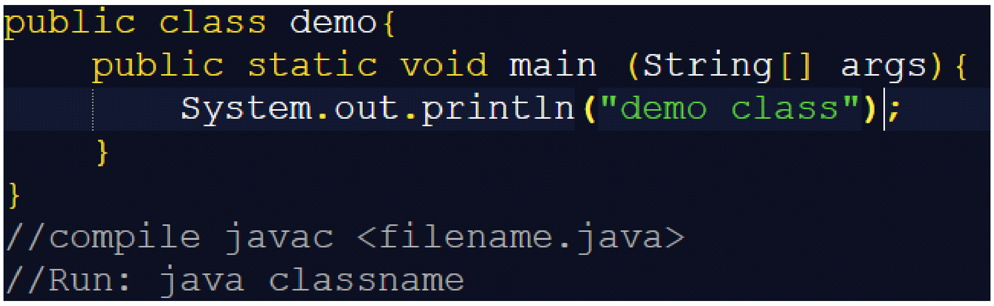
- Suppose the file is saved as demo1.java and we want to execute demo class. In this instance if filename (i.e. demo1) is used instead of class name (i.e. demo), JVM will throw a runtime error called **"Couldn't find or load main class"**. This happens because when demo1.java was compiled, two .class files(bytecodes) were generated with name similar to the classes in the program. Since there is no class with name demo1.java in this code, demo1.class file isn't created, triggering an error.

Pilot Program



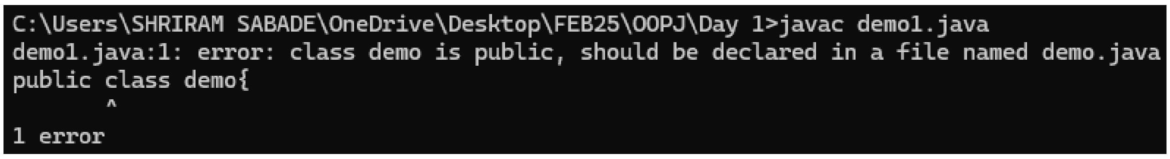
```
C:\Users\SHRIRAM SABADE\OneDrive\Desktop\FEB25\OOPJ\Day 1>java demo1
Error: Could not find or load main class demo1
Caused by: java.lang.ClassNotFoundException: demo1
```

- **Note: If class is declared with public access modifier, class name containing main method must be same as file name.**



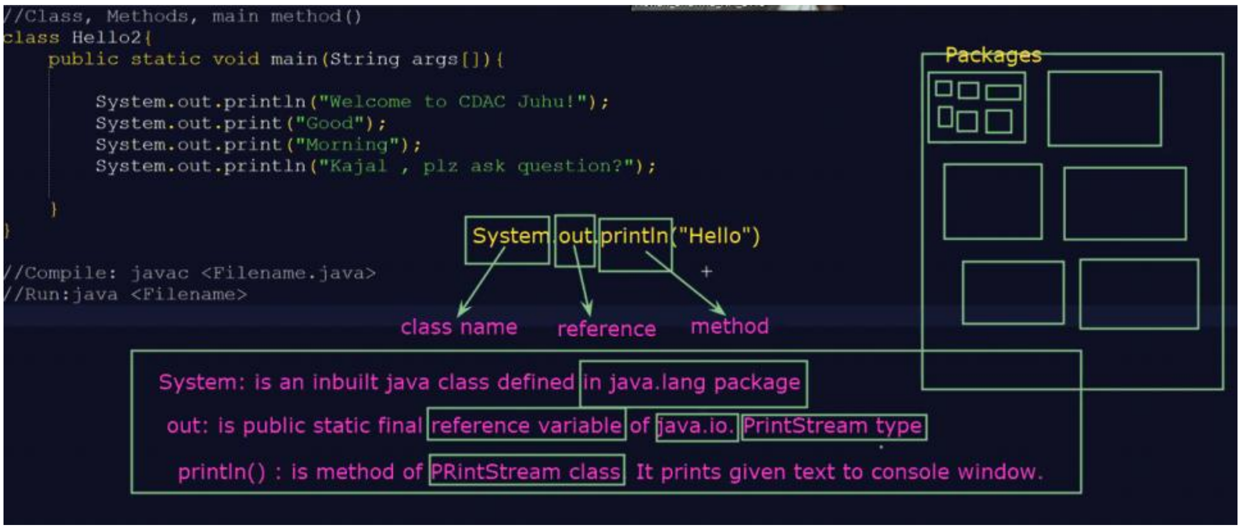
```
public class demo{
    public static void main (String[] args){
        System.out.println("demo class");
    }
}
//compile javac <filename.java>
//Run: java classname
```

- **Above program will only be executed successfully, if filename is demo.java. Anything else will result in a compile time error.**



```
C:\Users\SHRIRAM SABADE\OneDrive\Desktop\FEB25\OOPJ\Day 1>javac demo1.java
demo1.java:1: error: class demo is public, should be declared in a file named demo.java
public class demo{
       ^
1 error
```

println() method



```
//Class, Methods, main method()
class Hello2{
    public static void main(String args[]){

        System.out.println("Welcome to CDAC Juhu!");
        System.out.print("Good");
        System.out.print("Morning");
        System.out.println("Kajal , plz ask question?");

    }
}
//Compile: javac <Filename.java>
//Run:java <Filename>
```

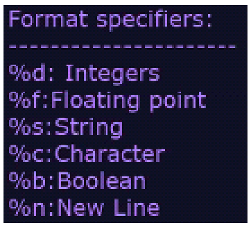
System: is an inbuilt java class defined in java.lang package

out: is public static final reference variable of java.io. PrintStream type

println() : is method of PRintStream class It prints given text to console window.

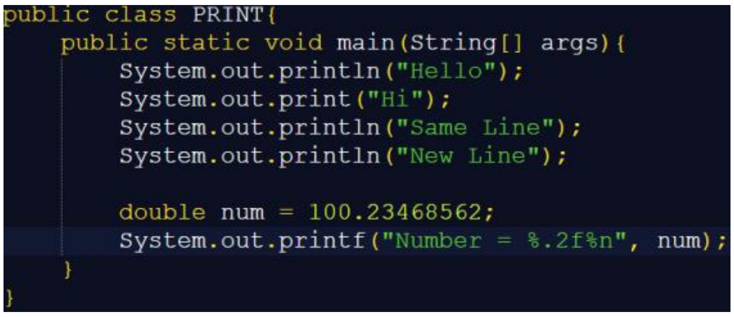
- There are 3 methods in PrintStream class
  - **print()** - prints the data on the current line and doesn't move the cursor to next line.
  - **println()** - prints the data on the current line and moves the cursor to the next line.
  - **printf()** - This method is used to get formatted output.

    Syntax:- System.out.println(format, argument)

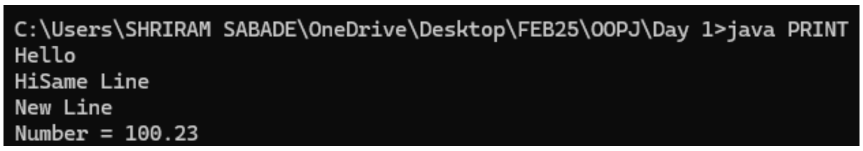


```
Format specifiers:
----------------------
%d: Integers
%f:Floating point
%s:String
%c:Character
%b:Boolean
%n:New Line
```

- Example:



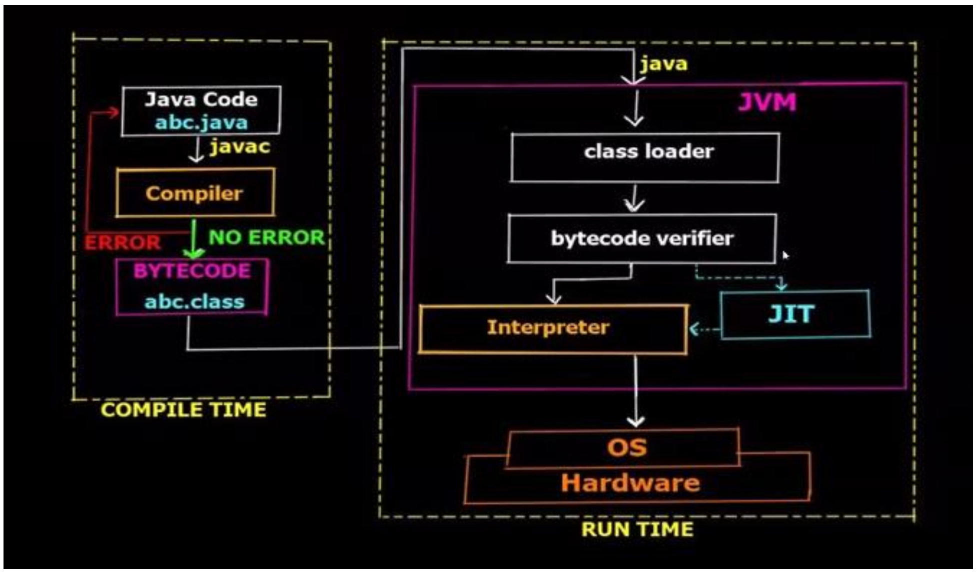
```
public class PRINT{
    public static void main(String[] args){
        System.out.println("Hello");
        System.out.print("Hi");
        System.out.println("Same Line");
        System.out.println("New Line");

        double num = 100.23468562;
        System.out.printf("Number = %.2f%n", num);
    }
}
```



```
C:\Users\SHRIRAM SABADE\OneDrive\Desktop\FEB25\OOPJ\Day 1>java PRINT
Hello
HiSame Line
New Line
Number = 100.23
```

JIT Compiler & Java Program execution

- **JIT (Just in Time) Compiler** converts bytecode into native machine code at runtime to improve performance.
- **Purpose:** Speed up the execution process for frequently used code.
- **Working:**
  - JVM first interprets the bytecode line by line.
  - JIT detects **hotspot methods (frequently used methods)**
  - Convert those methods into **native machine code.**
  - **Stores compiled code** for future reuse.



<u>Fig. Java file execution</u>

Tokens in Java

- In Java, tokens are the smallest elements of a program that are meaningful to the compiler. They are the fundamental building blocks of a Java program and are used to construct expressions and statements. The Java compiler breaks the line of code into text (words) called tokens. These tokens are separated by delimiters, which are not part of the tokens themselves.

- Various types of tokens in java are:
    - Keywords
    - Identifiers
    - Literals
    - Operators
    - Separators

- **Keywords:**
    - Predefined or reserved keywords in programming language.
    - Each keyword is meant to perform a specific function in a program.
    - There are total 55 keywords in Java.
    - They can't be used variable names because by doing so, we are trying to assign new meaning to a keyword, which isn't allowed.
    - E.g.:- break, catch, if, final, static, switch, etc.

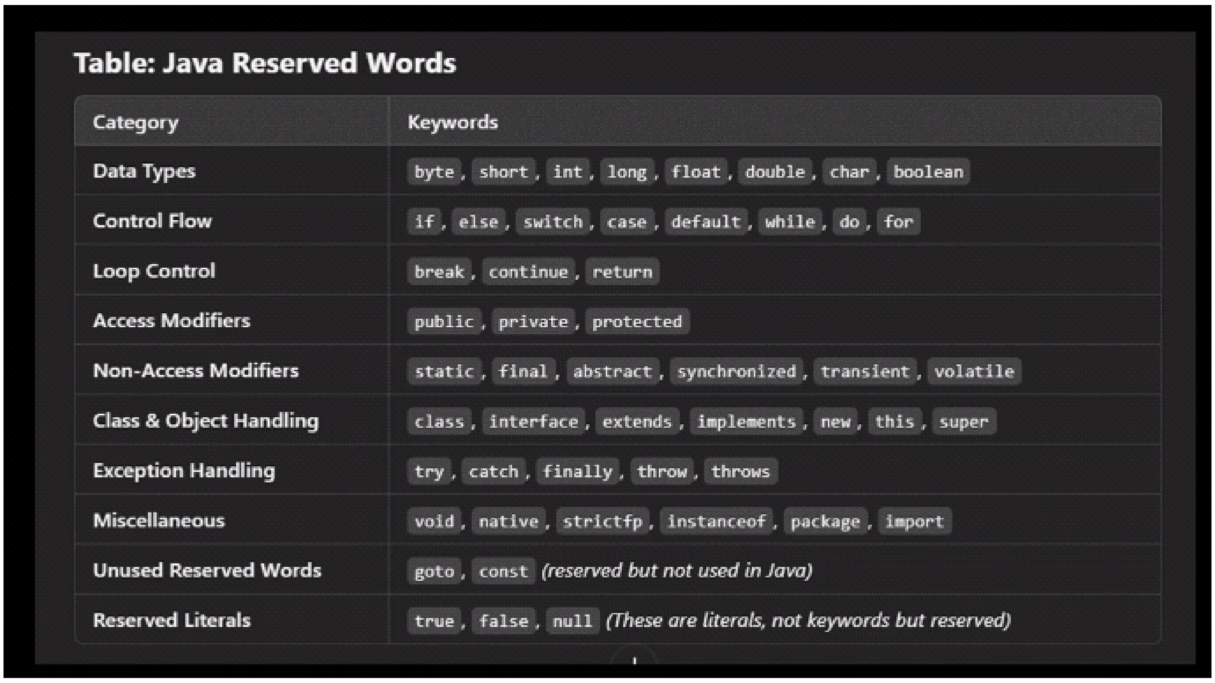


Table: Java Reserved Words

Category	Keywords
Data Types	`byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`
Control Flow	`if`, `else`, `switch`, `case`, `default`, `while`, `do`, `for`
Loop Control	`break`, `continue`, `return`
Access Modifiers	`public`, `private`, `protected`
Non-Access Modifiers	`static`, `final`, `abstract`, `synchronized`, `transient`, `volatile`
Class & Object Handling	`class`, `interface`, `extends`, `implements`, `new`, `this`, `super`
Exception Handling	`try`, `catch`, `finally`, `throw`, `throws`
Miscellaneous	`void`, `native`, `strictfp`, `instanceof`, `package`, `import`
Unused Reserved Words	`goto`, `const` *(reserved but not used in Java)*
Reserved Literals	`true`, `false`, `null` *(These are literals, not keywords but reserved)*

Tokens in Java

- **Identifiers:**
  - An identifier is the name given to various programming elements such as variables, methods, classes, interfaces and packages.
  - It helps us to uniquely identify these entities in Java.
  - Rules for identifier:
    - Must begin with a letter or underscore "_" or a dollar sign "$".
    - Cannot be a Java keyword.
    - Can contain letters, digits (0-9), underscore and dollar sign but cannot start with a digit.
    - Java is a case sensitive language. So, 'A' and 'a' are different identifiers.

- **Constants (literals):**
  - A literal is a fixed value that doesn't change during program execution.
  - Literals are categorized into:
    - **Numeric:**
      - Integer constants:
        - Decimal literals - base 10, 0-10, E.g.:- int x = 110;
        - Octal literals - base 8, 0-7, E.g.:- int x =-010;
        - Hexadecimal literals - base 16, prefix 0x or 0X E.g.:- int x = 0x562;

      - Real constants (floating point literals)
        - Default type: double
        - float: 'f' or 'F' E.g.:- float f = 124.563f;
        - double: 'd' or 'D' E.g.:- double d =   123.256;
        - If f is not mentioned after a floating point literal, java will consider that value as double.

    - **Character:**
      - Character constants
        - Escape characters: \n, \t, \r, \f . . .
        - Character Literals: 'A', 'B', 'a', '$' , "#', . . .
      - String constants.
        - String literal is a sequence of characters enclosed within double quotes("").
        - Java, String is an object of the 'String' class.
        - E.g.:- String s = "Sam";

- Java allows different types of literals like integer literals (E.g.:-100,-25, etc.), Floating-point literals (E.g.:- 3.14, -0.99, etc.), Character Literals (E.g.:-'A','g', etc.), String literals (E.g:-"Hello, World", etc..), and Boolean Literals (E.g.:- true, false).

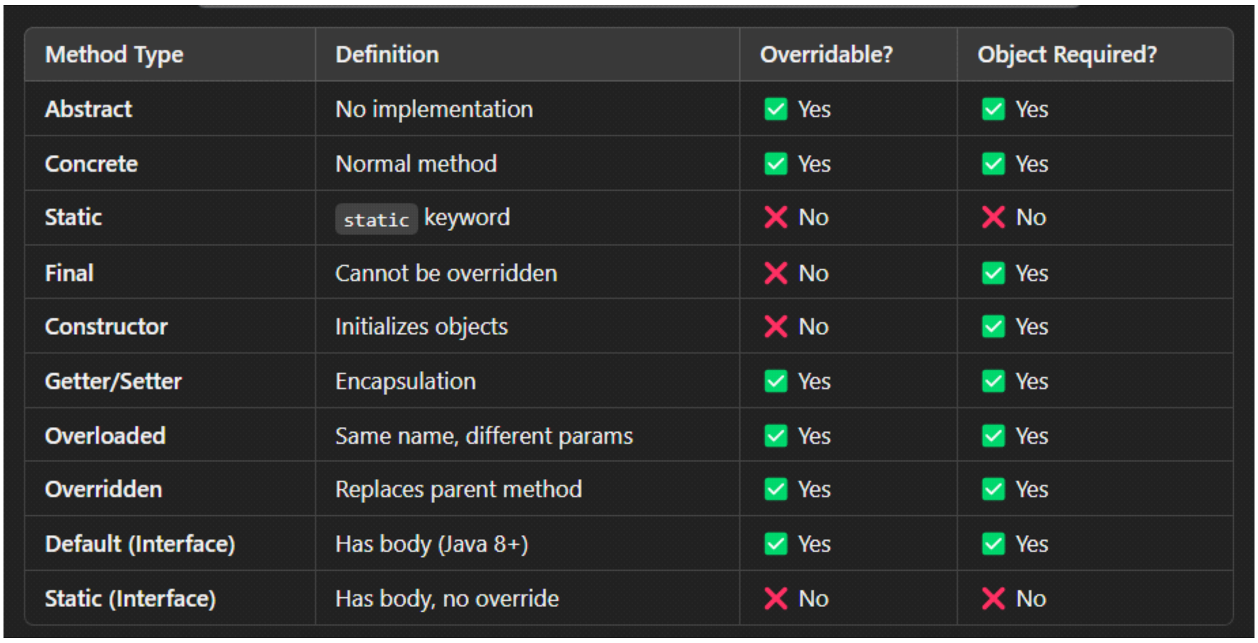
- **Operators:** An operator is a symbol that performs operations on variables and values. Java has several types of operators:

  - **Arithmetic Operators:** + (addition), - (subtraction), * (multiplication), / (division), % (modulus).
  - **Relational (Comparison) Operators:** == (equal to), != (not equal to), > (greater than), < (less than), >=(greater than or equal to), <=(lesser than or equal to).
  - **Logical Operators:** && (AND), || (OR), ! (NOT).
  - **Bitwise Operators:** &(Bitwise AND), |(Bitwise OR), ^(XOR), <<(left shift), >>(right shift) ,>>> (unsigned right shift).
  - **Assignment Operators:** =, +=, -=, *=, /=, %=.
  - **Unary Operators:** +, -, ++ (increment), -- (decrement).
  - **Ternary Operator:** ?: (conditional operator).

- **Separators:** A separator is a symbol used to separate different parts of a Java program. Java uses several separators:

  - **Parentheses () –** Used for method calls and defining precedence in expressions.
  - **Braces {} –** Used to define blocks of code, such as class bodies, methods, and loops.
  - **Brackets [] –** Used for arrays to define indexes.
  - **Semicolon ; –** Used to terminate statements.
  - **Comma , –** Used to separate multiple values in a statements.
  - **Period . –** Used for accessing class members or packages.

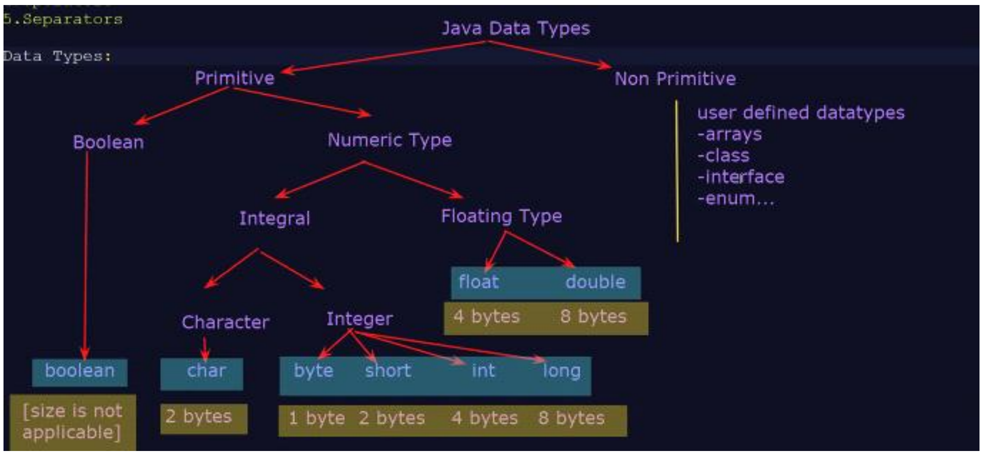
Types of methods



Method Type	Definition	Overridable?	Object Required?
Abstract	No implementation	✅ Yes	✅ Yes
Concrete	Normal method	✅ Yes	✅ Yes
Static	`static` keyword	❌ No	❌ No
Final	Cannot be overridden	❌ No	✅ Yes
Constructor	Initializes objects	❌ No	✅ Yes
Getter/Setter	Encapsulation	✅ Yes	✅ Yes
Overloaded	Same name, different params	✅ Yes	✅ Yes
Overridden	Replaces parent method	✅ Yes	✅ Yes
Default (Interface)	Has body (Java 8+)	✅ Yes	✅ Yes
Static (Interface)	Has body, no override	❌ No	❌ No

Data Types in Java

- There are two variants of data types in Java:
    - Primitive Data types
    - Non-Primitive Data types



- Primitive Data types:

Data Type	Size	Range	Default Value
boolean	Typically 1 byte (8 bits), but not guaranteed.	true/false	false
byte	1 byte	-128 to 127	0
short	2 bytes	-32768 to 32767	0
int	4 bytes	-2,147,483,648 to 2,147,483,647	0
long	8 bytes	2^{-63} to 2^{64}	0
char	2 bytes	Unicode values between 0 to 65,535	\u0000
Float	4 bytes	Up to 7 decimal digits	0.0
double	8 bytes	Up to 16 decimal digits	0.0

- **Note:** 1 and 0 are not equivalent to true and false respectively in Java.

Non-Primitive Data Types in Java

- **String:**
  - Strings are defined as array of characters.
  - Difference between a character array and a string is, the string is designed to hold a sequence of characters in a single variable whereas character array is a collection of separate char type entities.
  - Unlike C/C++, Java Strings are not terminated with null character.

- **Class:**
  - Class is a user defined blueprint. It contains methods which are common to all objects of one type.

- **Objects:**
  - An object is a basic unit of object oriented programming and represents real-life entities.
  - A typical java program creates many objects, which interact by invoking methods.
  - Object consists of state, behavior, and identity.

- **Interface:**
  - Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
  - Interfaces specify what a class must do and not how. It is a blueprint of a class.
  - If a class implements an interface and doesn't provide method bodies for all functions specified in the interface, then class has to be implemented.

Non-Primitive Data Types in Java

- **Array:**
  - An array is a group of similar data type variables that are commonly referred to by a common name.
  - In Java, all arrays are dynamically allocated.
  - Since arrays are objects in java, we can find their length using member length. This is different from C/C++ where we find length using size.
  - A java array variable can also be declared like other variables with [] after the data type.
  - Variables in the array are ordered and each array has indices beginning with 0.
  - Java array can also be used in as static field, a local variable, or a method parameter.
  - The size of an array must be specified by an int value and not long or short.
  - The direct superclass of an array is object.
  - Every array type implements the interfaces cloneable and java.io.serializable.

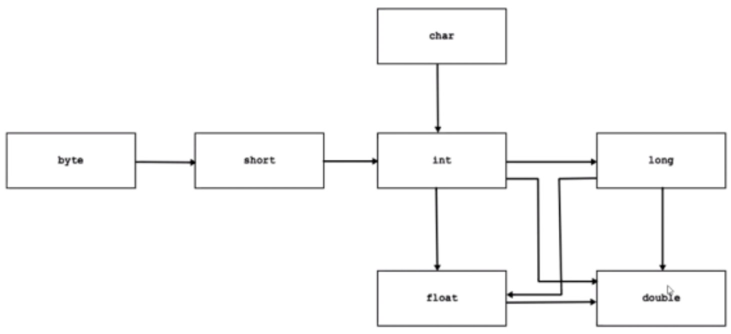
Float v Double

Float	Double
Its size is 4 bytes	Its size is 8 bytes
It has 7 decimal digits precision	It has 15 decimal digits precision
Precision errors might occur while dealing with large numbers	Precision errors won't occur while dealing with large numbers.
This data type supports up to 7 digits of storage.	This data type supports up to 15 digits of storage.
For float data type, the format specifier is %f.	For double data type, the format specifier is %lf.
It is less costly in terms of memory usage.	It is costly in terms of memory usage.
It requires less memory space as compared to double data type.	It needs more resources such as occupying more memory space in comparison to float data type.
It is suitable in graphics libraries for greater processing power because of its small range.	It is suitable to use in the programming language to prevent errors while rounding off the decimal values because of its wide range.
For example -: 3.1415	For E\ Example -: 5.3645803

Type Casting - Implicit Type Casting.

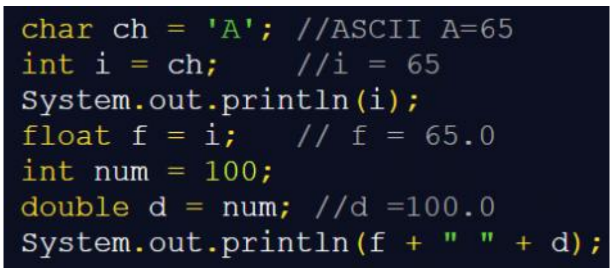
- Typecasting in java is the process of converting one data type to another data type.
- There are two categories of type casting in Java (in context to primitive data types) :
  - Widening type casting
  - Narrowing type casting
- Widening type casting:
  - A lower data type is transformed into higher one by a process known as widening type casting.
  - It is also known as implicit casting or casting down.
  - Since, There is no chance of data loss, it is secure. This type of type casting occurs naturally/automatically.
  - Widening type casting occurs when:
    - The target type must be larger than the source type.
    - Both data types must be compatible with each other.
  - Syntax:-
    <larger-data-type> Variable_name = <smaller-data-type-variable>

Widening

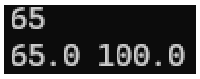


- Widening is the process of converting value of variable of narrower type into wider type.

  - E.g.:-



```
char ch = 'A'; //ASCII A=65
int i = ch;     //i = 65
System.out.println(i);
float f = i;    // f = 65.0
int num = 100;
double d = num; //d =100.0
System.out.println(f + " " + d);
```

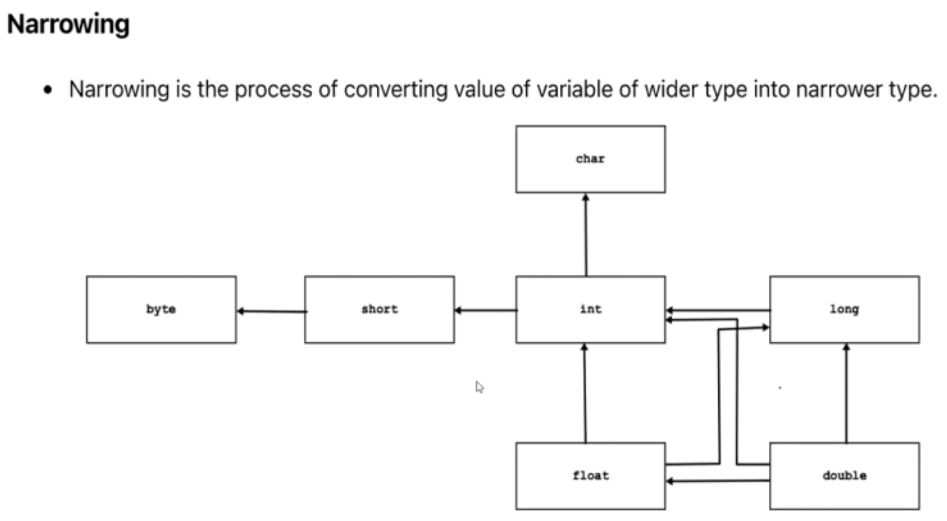


```
65
65.0 100.0
```

Explicit Type Casting

- The process of downsizing a bigger data type into smaller data type is known as explicit type casting.
- It is also called as narrowing type casting or casting up.
- It doesn't happen by itself. If we don't explicitly do it , compile time error occurs.
- It is unsafe as data loss might occur due to lower data type's smaller range of permitted values.
- A cast operator assists in the process of explicit casting.
- During explicit type casting, when larger data type's value (value outside the range of smaller data type) is assigned to smaller data type identifier using caste operator, value will move in cyclic fashion within the range of smaller data type.
- Syntax:

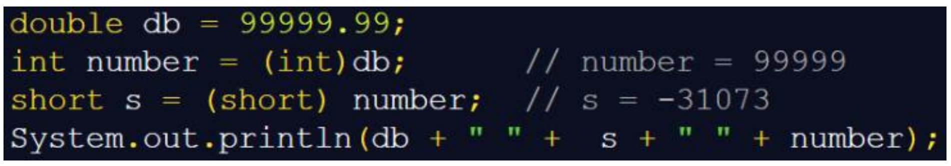
<smaller-data-type> VName = (smaller-data-type) <larger-data-type-variable>



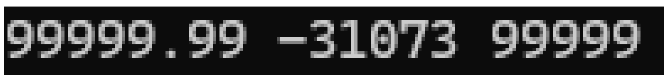
Narrowing

- Narrowing is the process of converting value of variable of wider type into narrower type.

E.g.:-



```
double db = 99999.99;
int number = (int)db;       // number = 99999
short s = (short) number;   // s = -31073
System.out.println(db + " " +  s + " " + number);
```



```
99999.99 -31073 99999
```

Decision making statements

- Decision making statements in java execute a block of code based on a condition.

- A programming language uses control statements ot control the flow of execution of a program based on certain conditions.

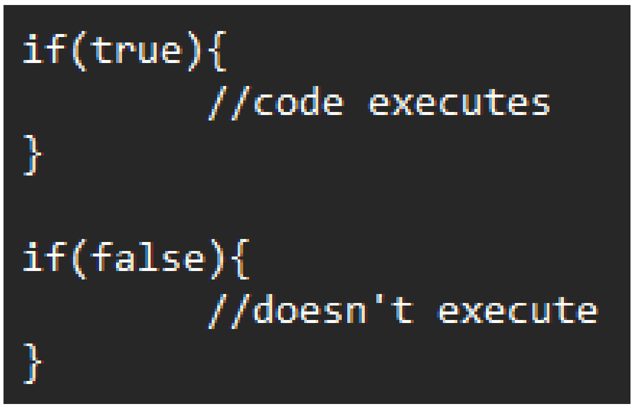
- These statements are also known as conditional statements, or control flow statements.

- Various types of Decision making statements in Java are:
    - If
    - If-else
    - If-else if-if
    - Nested if
    - Switch
    - Jump statements.

- **If Statement:**
    - It executes a block of code if a specified condition evaluates to true.
    - Syntax:

```
if(condition) {
      //code
}
```

    - E.g.:-

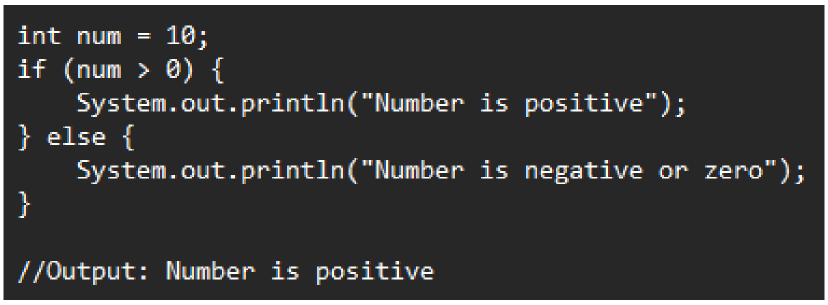


```
if(true){
        //code executes
}

if(false){
        //doesn't execute
}
```

Decision making statements

- **If-else:**
  - It executes a block of code if a specified condition evaluates to true. If the condition is false, another block of code (inside else) is executed.
  - Syntax:

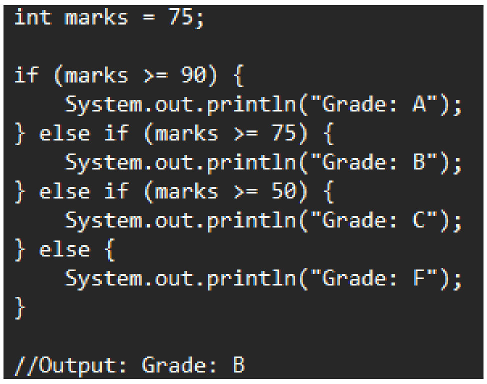
    if (condition) {
       // Code to execute if condition is true
    } else {
       // Code to execute if condition is false
    }
  - E.g.:-



```
int num = 10;
if (num > 0) {
    System.out.println("Number is positive");
} else {
    System.out.println("Number is negative or zero");
}

//Output: Number is positive
```

- **If-else if-else:**
  - When there are multiple conditions to check, we use if-else if-else. The program evaluates conditions from top to bottom and executes the block corresponding to the first true condition.
  - Syntax:

    if (condition) {
       // Code to execute if condition is true
    } else if (condition){
       // Code to execute if condition is true
    } else {
       // Code to execute if condition is false
  - E.g.:-

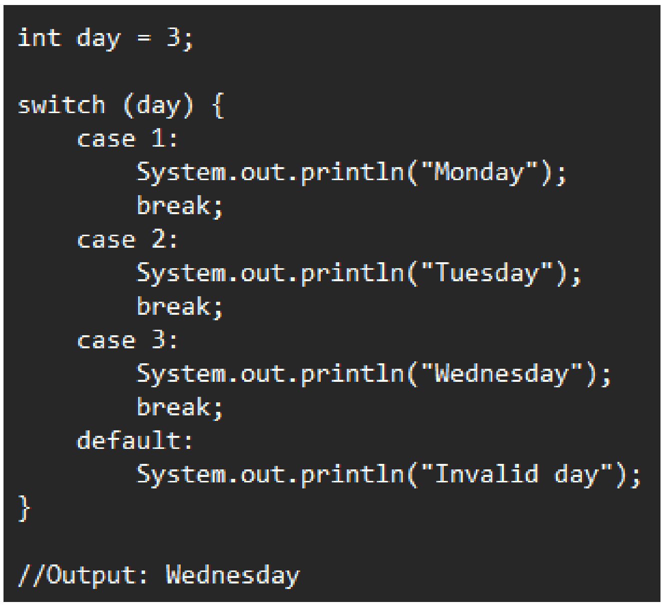


```
int marks = 75;

if (marks >= 90) {
    System.out.println("Grade: A");
} else if (marks >= 75) {
    System.out.println("Grade: B");
} else if (marks >= 50) {
    System.out.println("Grade: C");
} else {
    System.out.println("Grade: F");
}

//Output: Grade: B
```

- **Switch:**
  - The switch statement is an alternative to multiple if-else if conditions when a variable is compared against multiple constant values.
  - Syntax:

```
switch (expression) {
  case value1:
    // Code to execute if expression == value1
    break;
  case value2:
    // Code to execute if expression == value2
    break;
  default:
    // Code to execute if no case matches
}
```

  - E.g.:-



```
int day = 3;

switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}

//Output: Wednesday
```

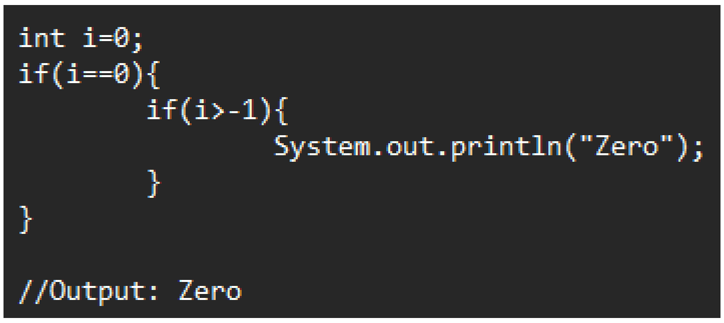
  - When a break statement is not provided in a switch case, it leads to a condition known as "fall-through."
  - Fall-through occurs when execution does not stop at the matched case and continues executing subsequent case blocks until a break statement or the switch block ends.
  - This behavior can be intentional or accidental, depending on the logic.

Decision making statements

- **Nested if statement**:
    - Nested if statements refer to an if statement(s) which is present inside another if statement.
    - Syntax:

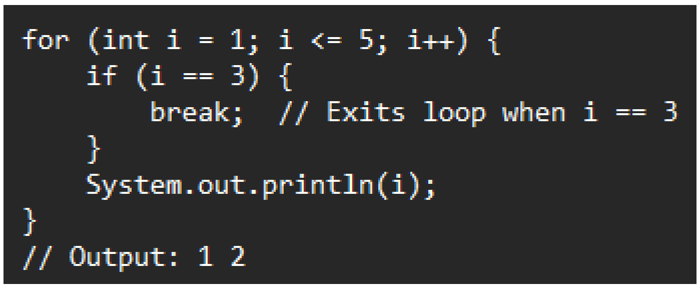
```
if(condition1){
        //executes if condition1 is true
        } if(condition2){
                //executes if condition2 is true
        }
}
```

    - E.g.:-



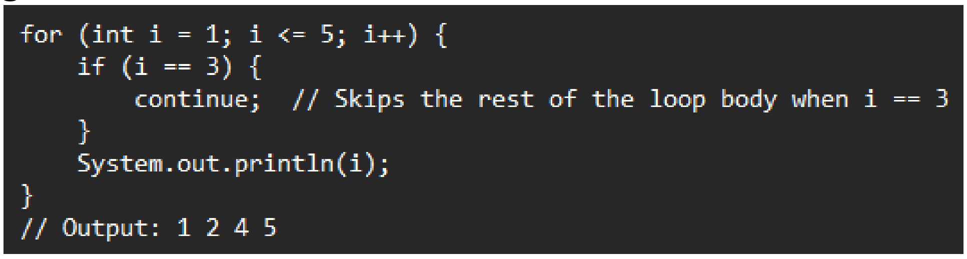
```
int i=0;
if(i==0){
        if(i>-1){
                System.out.println("Zero");
        }
}

//Output: Zero
```

- **Jump Statements:**
    - Jump statements alter the normal flow of execution in a program.
        - **break:**
            - The break statement is used to exit a loop or switch statement prematurely.
              The break statement prevents fall-through (execution of subsequent cases).
            - If break is omitted, execution will continue into the next case.
            - E.g.:-



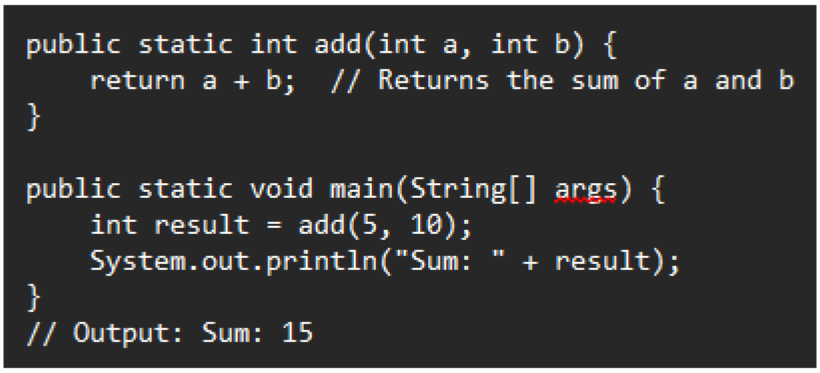
```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        break;  // Exits loop when i == 3
    }
    System.out.println(i);
}
// Output: 1 2
```

- **continue:**
  - The continue statement skips the rest of the current loop iteration and jumps to the next iteration.
  - E.g.:-



```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;  // Skips the rest of the loop body when i == 3
    }
    System.out.println(i);
}
// Output: 1 2 4 5
```

- **return:**
  - The return statement is used to exit a method and optionally return a value.
  - If a method is declared to return a data type but does not provide a return statement with a proper value, a compilation error will occur.
  - E.g.:-



```
public static int add(int a, int b) {
    return a + b;  // Returns the sum of a and b
}

public static void main(String[] args) {
    int result = add(5, 10);
    System.out.println("Sum: " + result);
}
// Output: Sum: 15
```

Loops in Java

- Loops in Java are control structures used to execute a block of code multiple times.

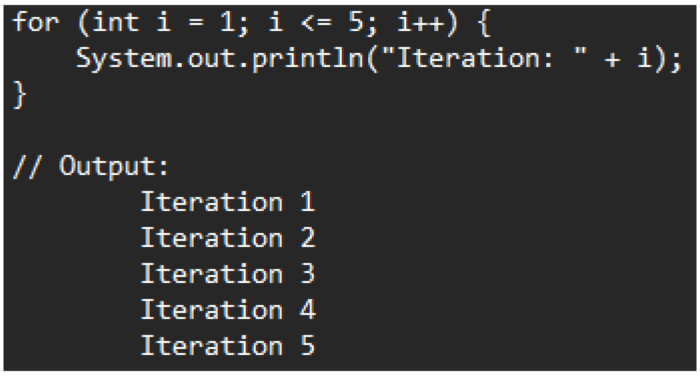
- They help in automating repetitive tasks, making the code more efficient and readable.

- **Off-by-One Error:** An Off-by-One (OBOE) error is a common programming mistake where a loop iterates one time too many or one time too few due to an incorrect boundary condition.

- **Infinite loop:** An infinite loop is a loop that never terminates because the exit condition is either never met or missing. This can lead to a program hanging or consuming excessive resources.

- **For loop:**
    - The for loop in Java is a control flow statement that allows code to be executed repeatedly based on a condition.
    - It is typically used when the number of iterations is known beforehand.
    - Syntax:

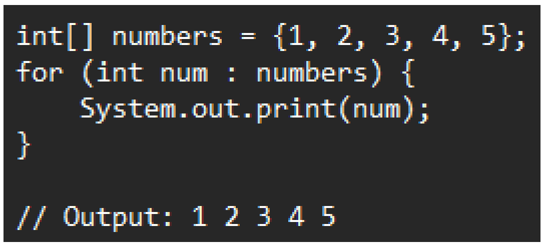
      for (initialization; condition; update) {
          // Code to be executed
      }
    - **Initialization:** A variable is initialized before the loop starts.
    - **Condition:** The loop runs while this condition evaluates to true.
    - **Update:** The loop control variable is updated after each iteration.
    - Initialization is done once, condition is checked n+1 times and updation and code block are executed n times.
    - E.g.:-



```
for (int i = 1; i <= 5; i++) {
    System.out.println("Iteration: " + i);
}

// Output:
        Iteration 1
        Iteration 2
        Iteration 3
        Iteration 4
        Iteration 5
```

- **Foreach loop:**
  - The foreach loop, also known as the enhanced for loop, is used to iterate over elements of an array or a collection without requiring an explicit index.
  - It simplifies iteration by eliminating the need for an explicit counter or index.
  - It is useful for iterating over arrays and collections like ArrayList.
  - Syntax:

    for (Type variable : collection) {
       // Code to be executed
    }
  - E.g.:-

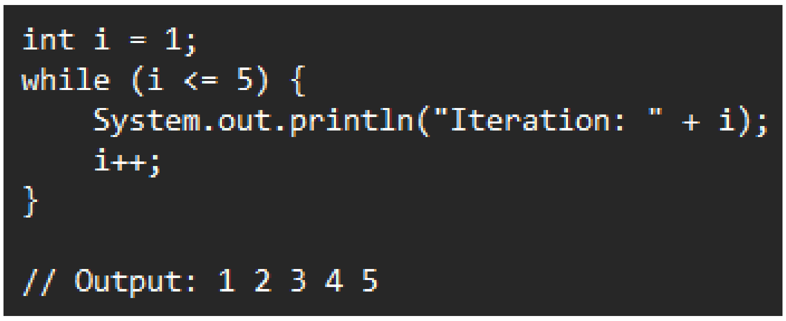


```
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.print(num);
}

// Output: 1 2 3 4 5
```

- **While loop:**
  - The while loop is a control structure that repeatedly executes a block of code as long as the specified condition evaluates to true.
  - The loop checks the condition before executing the block.
  - If the condition is initially false, the loop body may not execute at all.
  - Syntax:

    while (condition) {
       // Code to be executed
    }

- E.g.:-

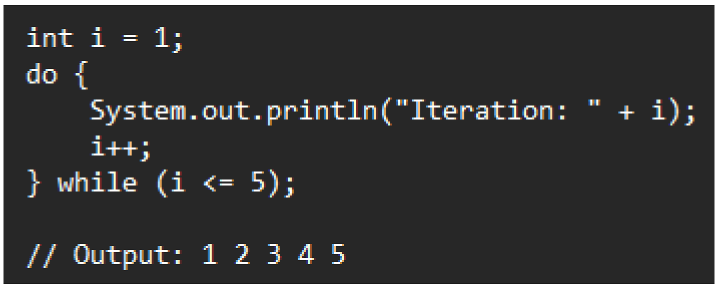


```
int i = 1;
while (i <= 5) {
    System.out.println("Iteration: " + i);
    i++;
}

// Output: 1 2 3 4 5
```

- **do-while Loop:**
  - The do-while loop is a variant of the while loop where the condition is checked **after** executing the loop body.
  - This ensures that the loop runs **at least once**, regardless of the condition.
  - The loop body executes before the condition is checked.
  - Even if the condition is false at the beginning, the loop will execute once.
  - Syntax:

```
do {
    // Code to be executed
} while (condition);
```

  - E.g.:-



```
int i = 1;
do {
    System.out.println("Iteration: " + i);
    i++;
} while (i <= 5);

// Output: 1 2 3 4 5
```

Loop Type	Condition Check	Guaranteed Execution	Best Use Case
for	Before each iteration	No	When number of iterations is known
foreach	N/A (Implicit Iteration)	No	Iterating through collections or arrays
while	Before each iteration	No	When condition needs to be checked first
do-while	After each iteration	Yes (at least once)	When loop should run at least once

Wrapper Class

- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

- A Wrapper class in Java is one whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types.

- **Autoboxing and Unboxing**
    - **Autoboxing:** The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double, etc.

    - **Unboxing:** It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Scanner Class

- The `Scanner` class is used to get user input, and it is found in the `java.util` package.
- To use the Scanner class, create an object of the class.

Method	Description
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user
`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

- Note: If you provide wrong input (e.g. text in a numerical input), you will get an exception/error message (like "InputMismatchException").
- Using the Scanner class in Java is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.
- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
- To read strings, we use nextLine().
- To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string.
- The Scanner class reads an entire line and divides the line into tokens. Tokens are small elements that have some meaning to the Java compiler.

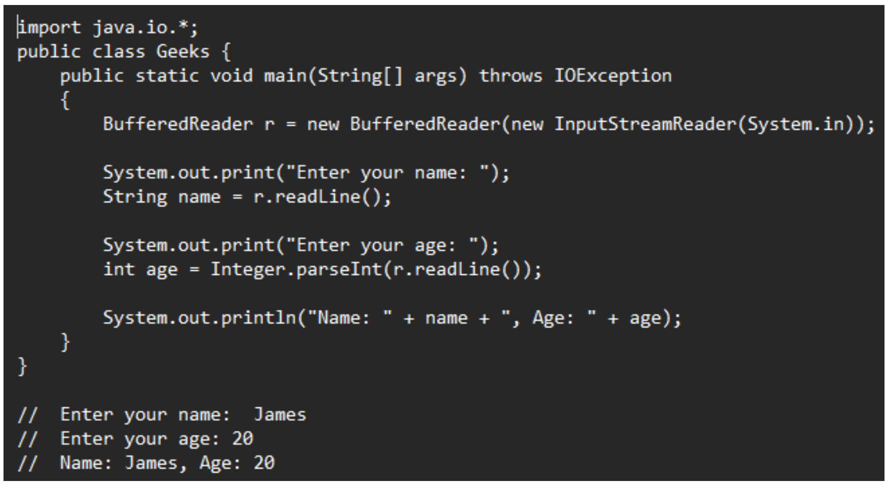
Buffered Reader Class (for input)

- Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

- The main difference between Scanner and BufferedReader is that Scanner Class provides parsing and input reading capabilities with built-in methods for different data types while BufferedReader Class reads text from a character input stream.

- We use BufferedReader when performance is important, especially for efficiently reading large volumes of data or files.

- BufferedReader reads large chunks of data at once, making it ideal for reading from files or processing large amounts of input.



```
import java.io.*;
public class Geeks {
    public static void main(String[] args) throws IOException
    {
        BufferedReader r = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Enter your name: ");
        String name = r.readLine();

        System.out.print("Enter your age: ");
        int age = Integer.parseInt(r.readLine());

        System.out.println("Name: " + name + ", Age: " + age);
    }
}

//  Enter your name:  James
//  Enter your age: 20
//  Name: James, Age: 20
```

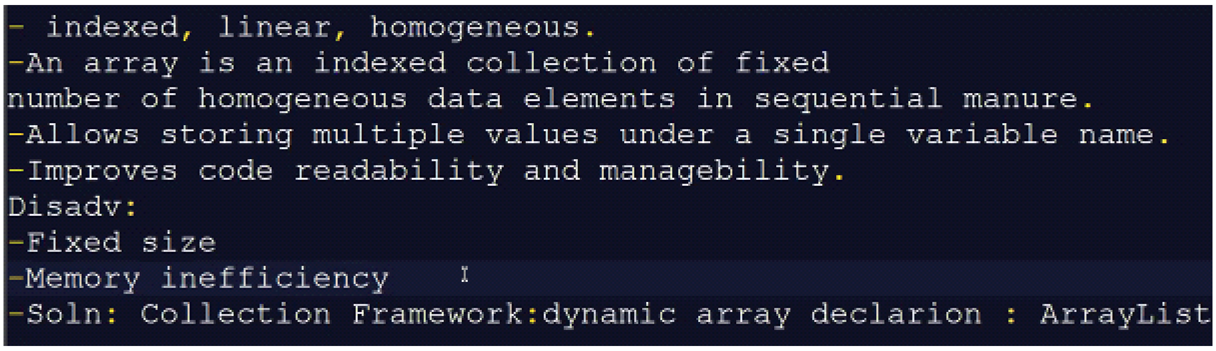
- **readLine():** Reads a line of text. A line is considered to be terminated by any one of a line feed (‘\n’), a carriage return (‘\r’), or a carriage return followed immediately by a line feed.

- **read():** Reads a single character.

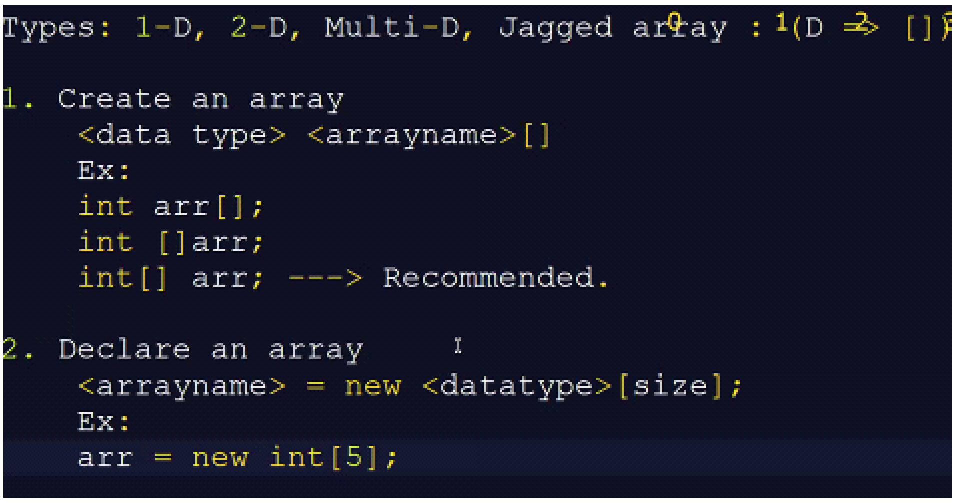
Buffered Reader vs Scanner

Aspect	Scanner	BufferedReader
Package	It is a part of java.util package.	It is a part of java.io package.
Key use	Simple parsing of primitive types and strings	High-performance text reading
Performance	Performance is slower due to parsing overhead and tokenization	Performance is faster due to efficient buffering
Buffer Size	Buffer Size is smaller	Buffer Size is larger
Thread-safe	It is not thread-safe	It is thread-safe
Error Handling	Throws Exception like InputMismatchException	Throws Exception like IOException

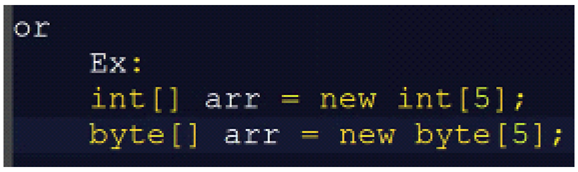
Arrays



```
- indexed, linear, homogeneous.
-An array is an indexed collection of fixed
number of homogeneous data elements in sequential manure.
-Allows storing multiple values under a single variable name.
-Improves code readability and managebility.
Disadv:
-Fixed size
-Memory inefficiency
-Soln: Collection Framework:dynamic array declarion : ArrayList
```



```
Types: 1-D, 2-D, Multi-D, Jagged array : 1(D => []

1. Create an array
    <data type> <arrayname>[]
    Ex:
    int arr[];
    int []arr;
    int[] arr; ---> Recommended.

2. Declare an array
    <arrayname> = new <datatype>[size];
    Ex:
    arr = new int[5];
```



```
or
    Ex:
    int[] arr = new int[5];
    byte[] arr = new byte[5];
```

Object Oriented Programming (OOP)

- Object oriented programming is a programming paradigm based on the concepts of objects.

- Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming.

- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

- Four pillars of OOP are Abstraction, Encapsulation Inheritance and Polymorphism.

- It helps in designing and developing scalable, maintainable and reusable systems.
  - Modularity: Divides the program into objects.
  - Reusability: Inherit existing functionality.
  - Scalability: Easier to manage large applications.
  - Secure: Abstraction and Encapsulation -> we restrict direct access to data.

- Key features of OOP:
  - Class:
    - A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class.
    - It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object.

  - Objects:
    - It is a basic unit of Object-Oriented Programming and represents the real-life entities. An Object is an instance of a Class.

    - When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is alllocated.

    - An object has an identity, state, and behavior. Each object contains data and code to manipulate the data.
    - Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

- Data Abstraction:
    - Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
    - Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not have any idea about how it works.

- Encapsulation:
    - The process of wrapping properties and functions within a single unit is known as encapsulation.
    - In encapsulation, the data in a class is hidden from other classes, which is similar to what data-hiding does. So, the terms "encapsulation" and "data-hiding" are used interchangeably.
    - Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

- Inheritance:
    - It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using extends keyword. Inheritance is also known as "is-a" relationship.

- **Superclass:** The class whose features are inherited is known as superclass (also known as base or parent class).

- **Subclass:** The class that acquires properties and behaviour of other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

- **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.

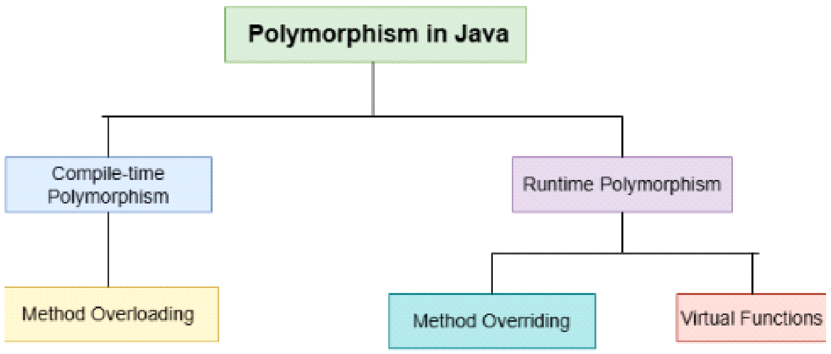
○ Polymorphism:

- The word 'polymorphism' means 'having many forms'.

- It lets you use the same function or method on different types of objects, and they each respond in their own way.

- E.g.:- A person can have different characteristics at the same time. Like a man at the same time is a father, a husband, and an employee.

- In Java Polymorphism is mainly divided into two types - Compile-Time Polymorphism and Runtime Polymorphism



Instance Variable and Reference Variable.

- **Instance :**
  - An instance is a concreate object created from a class.

  - Each instance has its own separate memory allocation.

```
class Car {
    String brand; // Instance variable
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // 'myCar' is an instance of the Car class
        myCar.brand = "Toyota";

        System.out.println(myCar.brand);  // Output: Toyota
    }
}
```

- **Method():** A method is a block of code that performs an action. Methods can take parameters and return values.

- **Instance Variable:**
  - Instance variable represents an attribute of an object. It is declared inside a class but outside the method but inside the class.

  - An instance variable is a variable that belongs to an object (instance) of a class. Each object has its own copy of instance variables, which store unique values for that object.

  - Example:

```
class Car {
    String brand;  // Instance variable
    int speed;     // Instance variable
```

- **Reference Variable:**
  - A reference is a variable that stores the memory address of an object (instance) instead of the actual object itself. In Java, when you create an object, you use a reference variable to access it.

  - It is used to access the object's fields and methods.

  - E.g.:-

```
Car car1 = new Car();  // 'car1' is a reference to a Car object
```

Pillars of OOP

- Four pillars of Object Oriented Programming are:
    - **Abstraction:** Hiding the complex part and showing only the part that is necessary.
    - **Polymorphism:** Ability to implement same thing in different ways.
    - **Inheritance:** Ability to acquire properties and behaviour of another class or interface.
    - **Encapsulation:** Wrapping the whole functionality into a single unit.

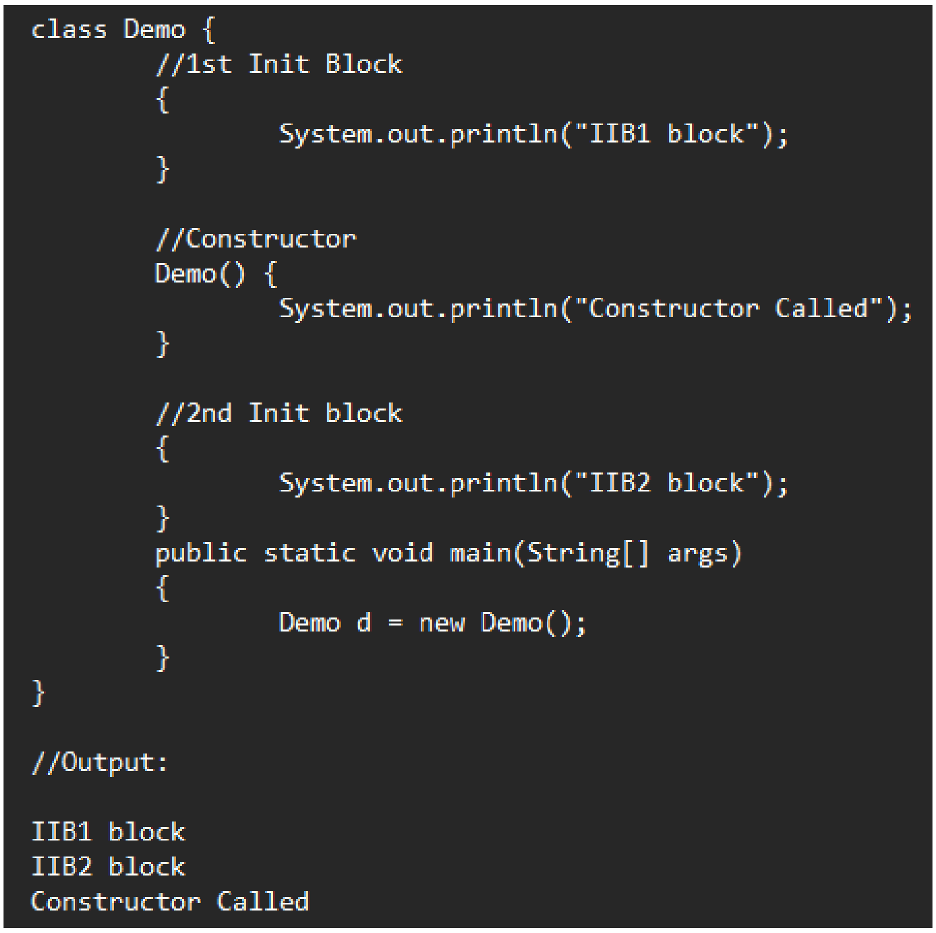
- These pillars can be categorized as:

    - Major Pillars
        - Encapsulation
        - Modularity
        - Abstraction
        - Hierarchy

    - Minor Pillars:
        - Typing
        - Concurrency
        - Persistence

Instance Initializer Block

- **Initialization Block:**
    - An Instance Initialization Block (IIB) in Java is a block of code inside a class that runs each time an object is created, before the constructor is executed.
    - They run each time when the object of the class is created.
    - Initialization blocks are executed whenever the class is initialized and before constructors are invoked.
    - They are typically placed above the constructors within braces.
    - It is not at all necessary to include them in your classes.
    - We can also have multiple IIBs in a single class. If the compiler finds multiple IIBs, then they all are executed from top to bottom i.e. the IIB which is written at the top will be executed first.



```
class Demo {
        //1st Init Block
        {
                System.out.println("IIB1 block");
        }

        //Constructor
        Demo() {
                System.out.println("Constructor Called");
        }

        //2nd Init block
        {
                System.out.println("IIB2 block");
        }
        public static void main(String[] args)
        {
                Demo d = new Demo();
        }
}

//Output:

IIB1 block
IIB2 block
Constructor Called
```

Static Keyword

- The static keyword in Java is mainly used for memory management.

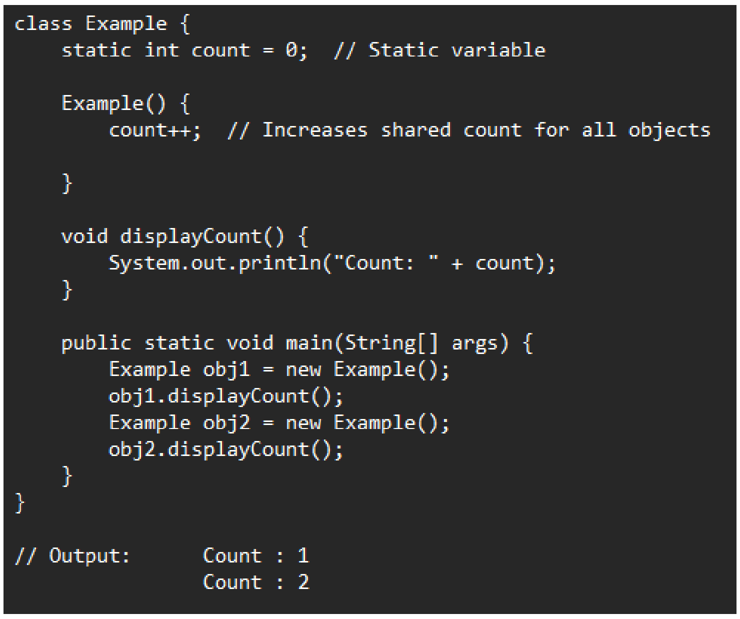
- We use the static keyword for the property that is common to all objects. For example, Suppose there is a class named Student where all students share the same college name. Use static methods for changing static variables.

- The **static keyword** in Java is used to indicate that a variable, method, or block belongs to the class rather than to instances (objects) of the class.

- This means static members are shared among all objects of the class instead of being unique for each instance.

- Any static member can be accessed before any objects of its class are created, and without reference to any object.

- **Static Variable (Class Variable):**
  - When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level.
  - Static variables are, essentially, global variables.
  - A static variable belongs to the class, not individual objects. All instances share the same copy.
  - We can create static variables at the class level only. Static block and static variables are executed in the order they are present in a program.



```java
class Example {
    static int count = 0;  // Static variable

    Example() {
        count++;  // Increases shared count for all objects

    }

    void displayCount() {
        System.out.println("Count: " + count);
    }

    public static void main(String[] args) {
        Example obj1 = new Example();
        obj1.displayCount();
        Example obj2 = new Example();
        obj2.displayCount();
    }
}

// Output:      Count : 1
                Count : 2
```

Static Keyword

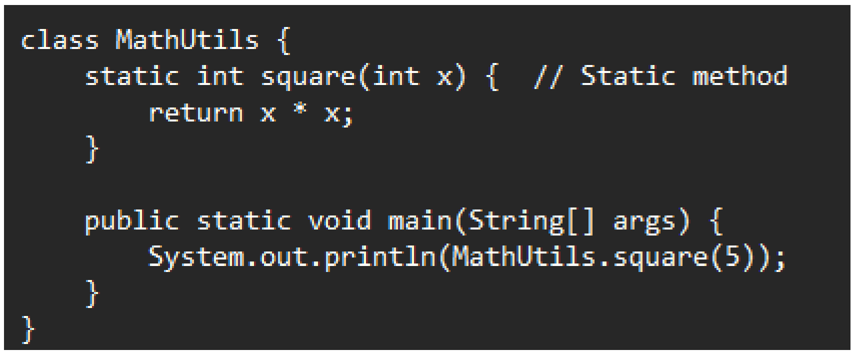
- **Static Methods:**
  - When a method is declared with the static keyword, it is known as the static method. The most common example of a static method is the main( ) method.

  - A static method can be called without creating an object of the class.

  - Methods declared as static have several restrictions:
    - It can only call other static methods. We cannot call non-static methods from static class.
    - It can only access static variables. We cannot use non-static variables in static methods.
    - They cannot refer to **this** or **super** in any way.

  - If we need to access static method of a class we need to call it using class name. E.g.:- Car.Brakes()

  - E.g.:- In below example, We call square(5) without creating an object of MathUtils.

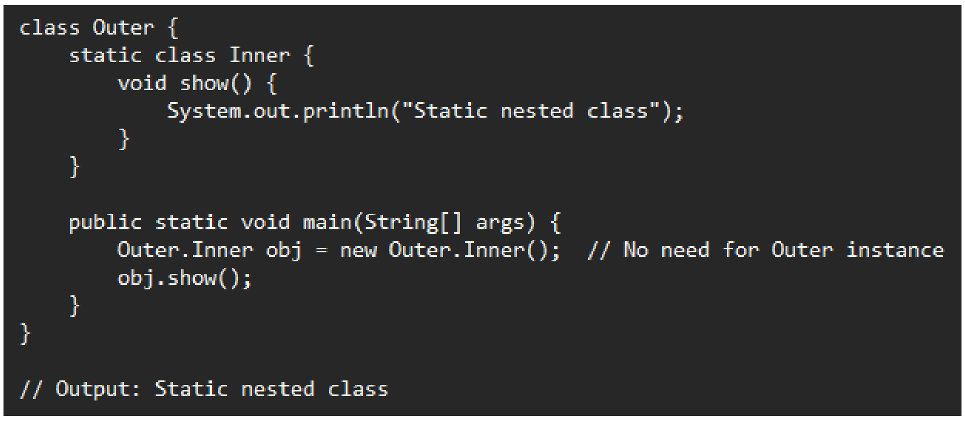


```
class MathUtils {
    static int square(int x) {  // Static method
        return x * x;
    }

    public static void main(String[] args) {
        System.out.println(MathUtils.square(5));
    }
}
```

- **Static Classes:**
  - A class can be made static only if it is a nested class. We cannot declare a top-level class with a static modifier but can declare nested classes as static.

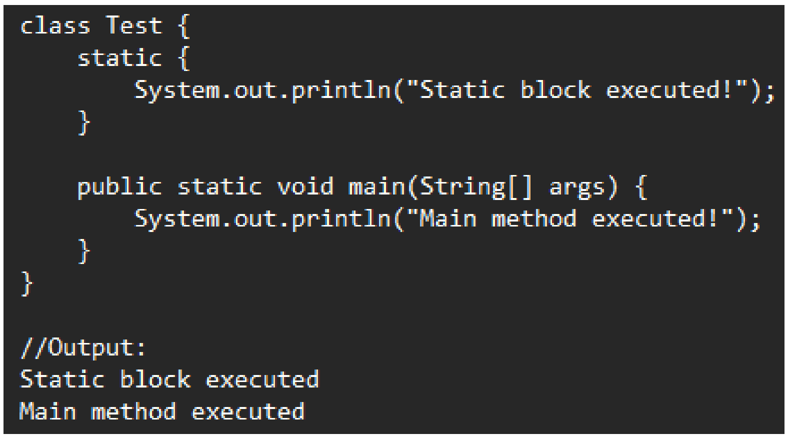
  - Such types of classes are called Nested static classes. Nested static class doesn't need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

- E.g.:-



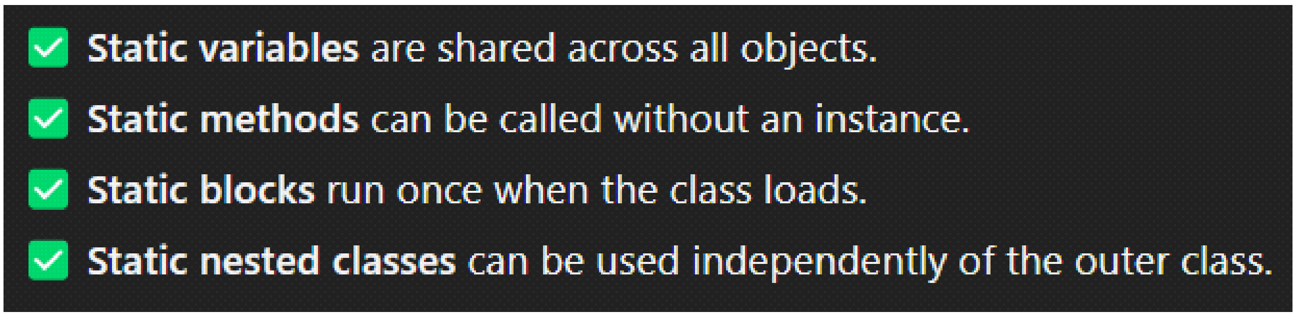
```
class Outer {
    static class Inner {
        void show() {
            System.out.println("Static nested class");
        }
    }

    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner();  // No need for Outer instance
        obj.show();
    }
}

// Output: Static nested class
```

- **Static Block:**
  - A static block in Java is a block of code inside a class that runs once when the class is loaded into memory, before the main method or any object creation.



```
class Test {
    static {
        System.out.println("Static block executed!");
    }

    public static void main(String[] args) {
        System.out.println("Main method executed!");
    }
}

//Output:
Static block executed
Main method executed
```

- **Summary:**



✅ **Static variables** are shared across all objects.

✅ **Static methods** can be called without an instance.

✅ **Static blocks** run once when the class loads.

✅ **Static nested classes** can be used independently of the outer class.

Static Keyword Characteristics

- **Shared memory allocation:** Static variables and methods are allocated memory space only once during the execution of the program. This memory space is shared among all instances of the class, which makes static members useful for maintaining global state or shared functionality.

- **Accessible without object instantiation:** Static members can be accessed without the need to create an instance of the class.

- **Associated with class, not objects:** Static members are associated with the class, not with individual objects. This means that changes to a static member are reflected in all instances of the class, and that you can access static members using the class name rather than an object reference.

- **Cannot access non-static members:** Static methods and variables cannot access non-static members of a class, as they are not associated with any particular instance of the class.

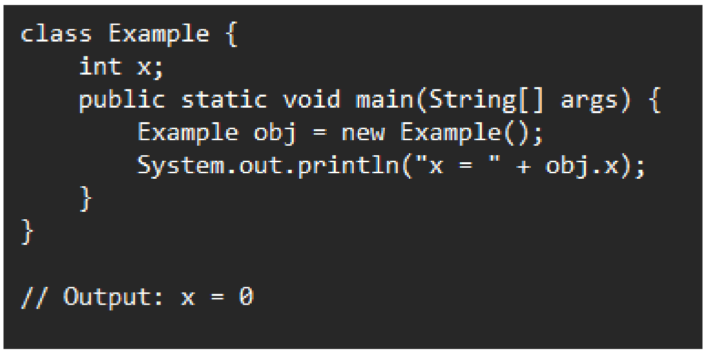
- **Can be overloaded, but not overridden:** Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class. If we try to override them without using @Override keyword, static method in main class will not be called leading to method hidding.

Constructors

- 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.
- Constructor is a block of codes similar to the method. It is called when an instance 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.
- How are **constructor different as compared to method**:
    - Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
    - Constructors do not return any type while method(s) have the return type or void if does not return any value.
    - Constructors are called only once at the time of Object creation while method(s) can be called any number of times.
- A constructor in Java cannot be abstract, final, static, or Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e. which other class can call the constructor.
- 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).
- There are no “return value” statements in the constructor, but the constructor returns the current class instance. We can write ‘return’ inside a constructor.
- Constructors can be overloaded by defining multiple constructors with different parameters (Constructor Overloading).
- Various types of constructors in Java are:
    - Default Constructor
    - Parameterized Constructor
    - Copy Constructor

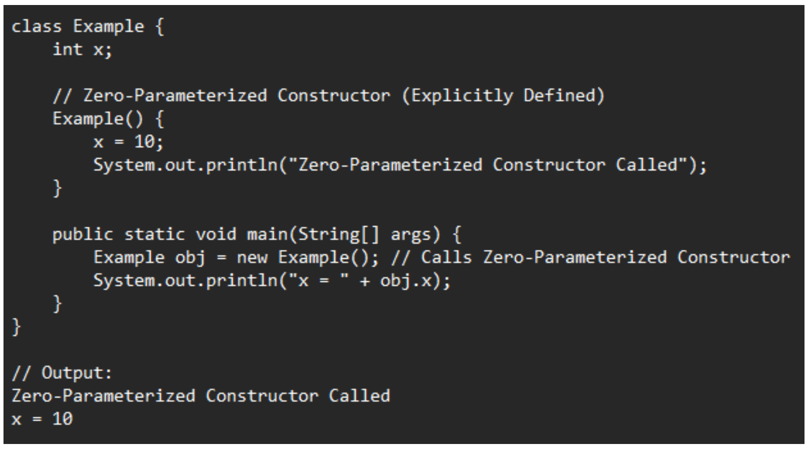
Constructors

- **Default Constructor:**
    - A constructor that has no parameters is known as default constructor. The default constructor can be implicit or explicit.
    - A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor.
    - **Implicit Default Constructor:** If no constructor is defined in a class, the Java compiler automatically provides a default constructor. This constructor doesn't take any parameters and initializes the object with default values, such as 0 for numbers, null for objects. E.g.:-In the following example, Compiler provides a default constructor when object of a class is created.



```
class Example {
    int x;
    public static void main(String[] args) {
        Example obj = new Example();
        System.out.println("x = " + obj.x);
    }
}

// Output: x = 0
```

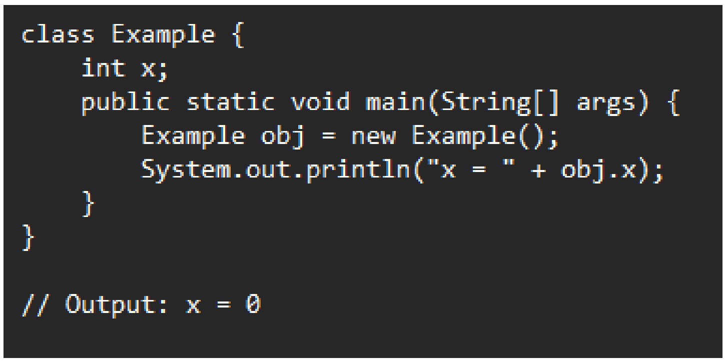
    - **Explicit Default Constructor:** If we define a constructor that takes no parameters, it's called an explicit default constructor. This constructor replaces the one the compiler would normally create automatically. Once you define any constructor (with or without parameters), the compiler no longer provides the default constructor for you. E.g.:-



```
class Example {
    int x;

    // Zero-Parameterized Constructor (Explicitly Defined)
    Example() {
        x = 10;
        System.out.println("Zero-Parameterized Constructor Called");
    }

    public static void main(String[] args) {
        Example obj = new Example(); // Calls Zero-Parameterized Constructor
        System.out.println("x = " + obj.x);
    }
}

// Output:
Zero-Parameterized Constructor Called
x = 10
```

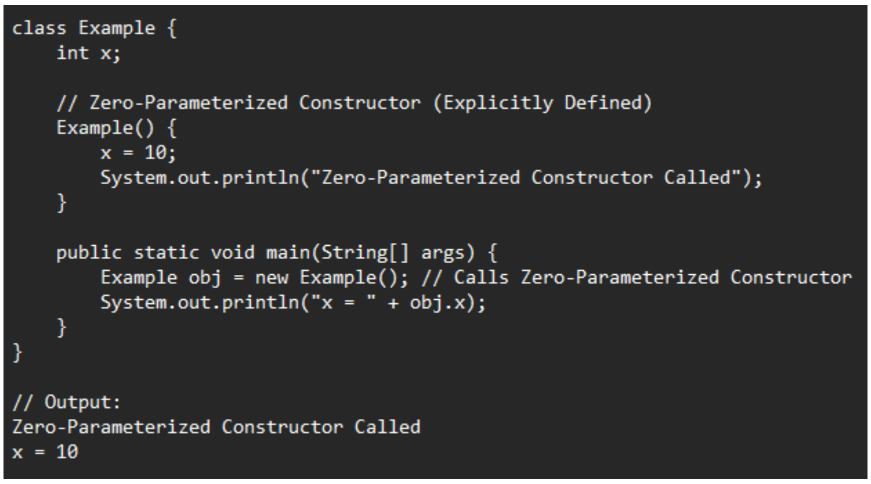
Constructor Types

- **Default Constructor:**
  - A constructor that has no parameters is known as default constructor. The default constructor can be implicit or explicit.
  - A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor.
  - **Implicit Default Constructor:** If no constructor is defined in a class, the Java compiler automatically provides a default constructor. This constructor doesn't take any parameters and initializes the object with default values, such as 0 for numbers, null for objects. E.g.:-In the following example, Compiler provides a default constructor when object of a class is created.



```
class Example {
    int x;
    public static void main(String[] args) {
        Example obj = new Example();
        System.out.println("x = " + obj.x);
    }
}

// Output: x = 0
```

  - **Explicit Default Constructor:** If we define a constructor that takes no parameters, it's called an explicit default constructor. This constructor replaces the one the compiler would normally create automatically. Once you define any constructor (with or without parameters), the compiler no longer provides the default constructor for you. E.g.:-

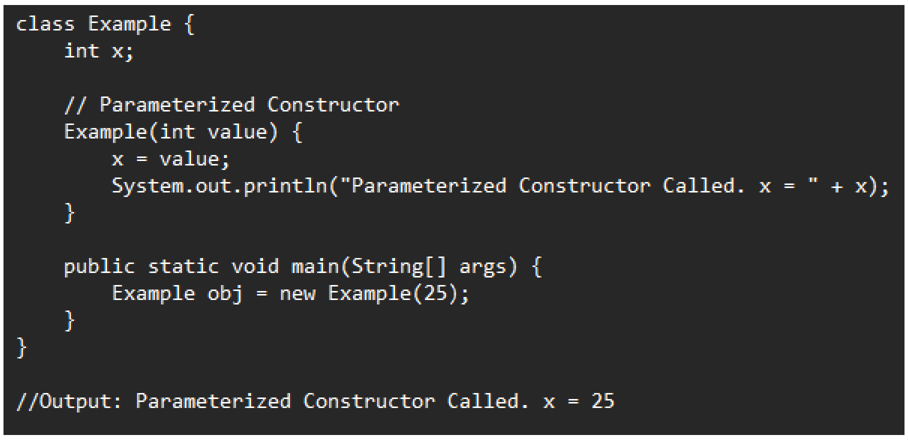


```
class Example {
    int x;

    // Zero-Parameterized Constructor (Explicitly Defined)
    Example() {
        x = 10;
        System.out.println("Zero-Parameterized Constructor Called");
    }

    public static void main(String[] args) {
        Example obj = new Example(); // Calls Zero-Parameterized Constructor
        System.out.println("x = " + obj.x);
    }
}

// Output:
Zero-Parameterized Constructor Called
x = 10
```

Constructor Types

- **Parameterized Constructor:**
  - A constructor that has parameters is known as parameterized constructor. If we want to initialize an object of the class with user defined values, then use a parameterized constructor.

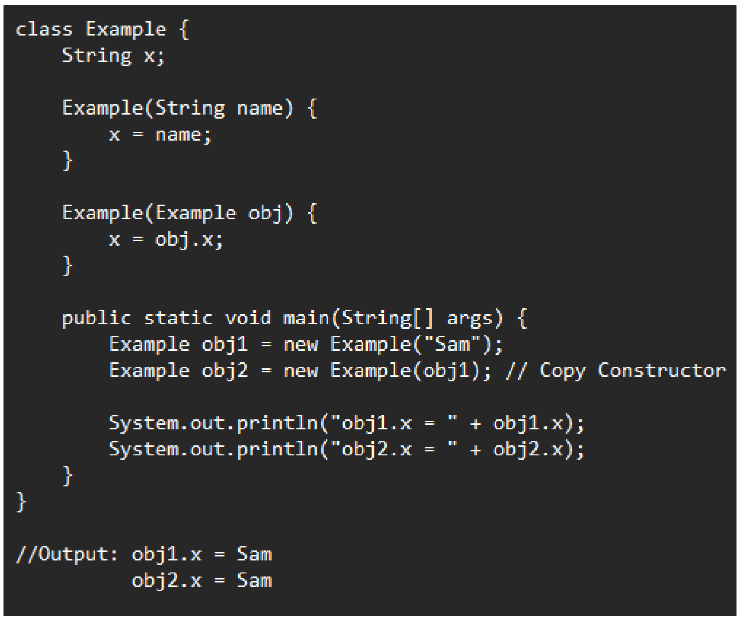
  - E.g.:-



```
class Example {
    int x;

    // Parameterized Constructor
    Example(int value) {
        x = value;
        System.out.println("Parameterized Constructor Called. x = " + x);
    }

    public static void main(String[] args) {
        Example obj = new Example(25);
    }
}

//Output: Parameterized Constructor Called. x = 25
```

- **Copy Constructor:**
  - Unlike other constructors, copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

  - Copy constructor creates a new object by copying an existing object's values



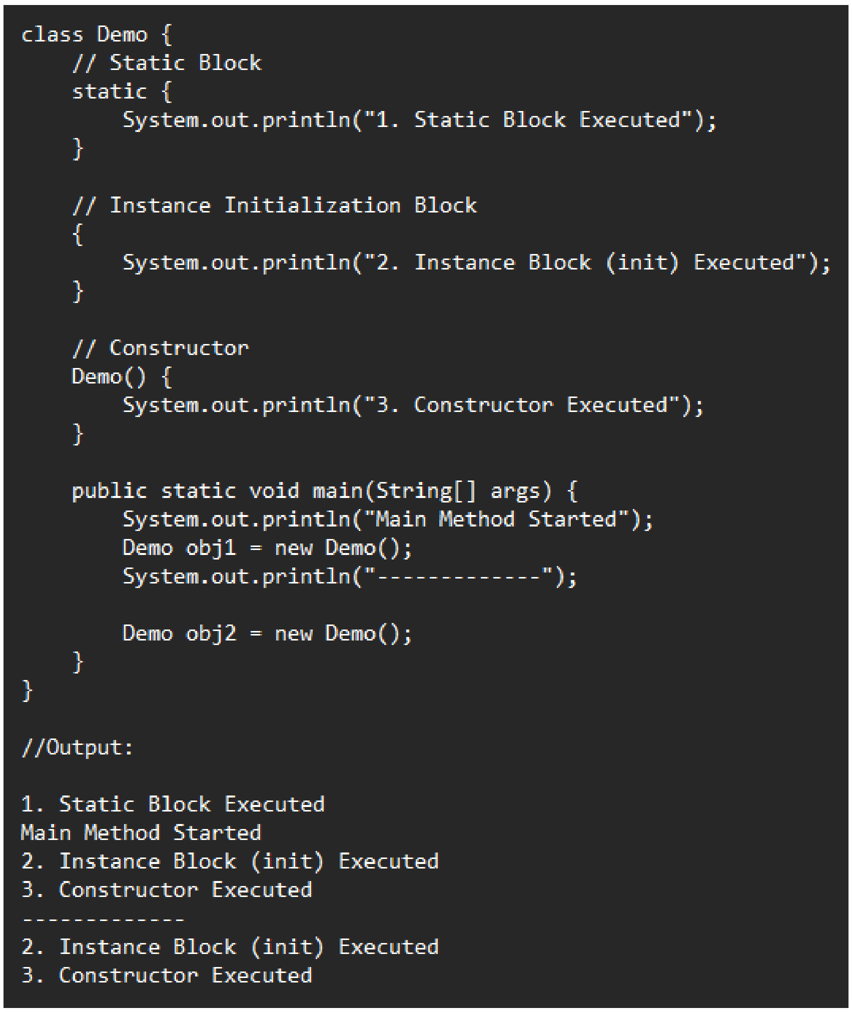
```
class Example {
    String x;

    Example(String name) {
        x = name;
    }

    Example(Example obj) {
        x = obj.x;
    }

    public static void main(String[] args) {
        Example obj1 = new Example("Sam");
        Example obj2 = new Example(obj1); // Copy Constructor

        System.out.println("obj1.x = " + obj1.x);
        System.out.println("obj2.x = " + obj2.x);
    }
}

//Output: obj1.x = Sam
          obj2.x = Sam
```

- **Private Constructor:** Constructor can be declared private. A private constructor is used in restricting object creation.

Flow of execution - Init, static block and Constructor

- In Java, the flow of execution in a class follows a specific order when an object is created. The sequence is:

  - Static Block (Runs Only Once, When Class is Loaded)
  - Instance Initialization Block (init block) (Runs Before Constructor, Every Time an Object is Created)
  - Constructor (Runs Every Time an Object is Created)



```
class Demo {
    // Static Block
    static {
        System.out.println("1. Static Block Executed");
    }

    // Instance Initialization Block
    {
        System.out.println("2. Instance Block (init) Executed");
    }

    // Constructor
    Demo() {
        System.out.println("3. Constructor Executed");
    }

    public static void main(String[] args) {
        System.out.println("Main Method Started");
        Demo obj1 = new Demo();
        System.out.println("-------------");

        Demo obj2 = new Demo();
    }
}

//Output:

1. Static Block Executed
Main Method Started
2. Instance Block (init) Executed
3. Constructor Executed
-------------
2. Instance Block (init) Executed
3. Constructor Executed
```

- **Explanation:**
  - Static block executes only once when the class is loaded into memory.

  - Instance (init) block executes before the constructor, every time an object is created.

  - Constructor executes after the instance block, initializing object-specific properties.

Constructor Overloading

- **Constructor Overloading** is a feature in object-oriented programming where a class can have multiple constructors with different parameter lists. The correct constructor is called based on the number and type of arguments passed during object creation.

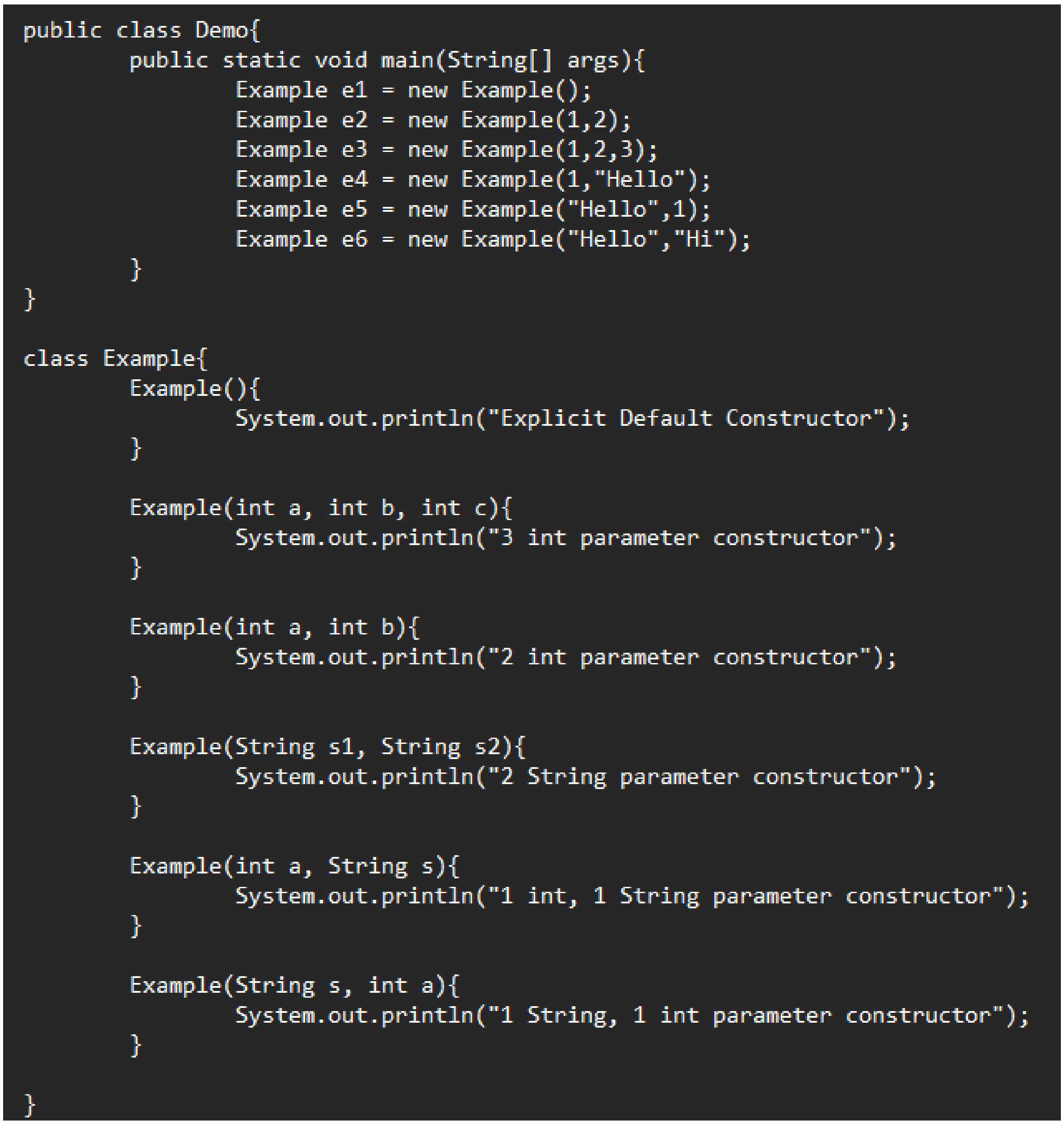
- Rules for Constructor Overloading:
  - Constructors must have the same name as the class.
  - Each constructor must have a unique parameter list (different in type, number, or order).
  - The appropriate constructor is selected at runtime based on the arguments passed.

- Why constructor overloading is performed?
  - Overloaded constructors allow different ways to initialize objects.
  - The constructor executed depends on the arguments provided.
  - It improves flexibility and code readability.

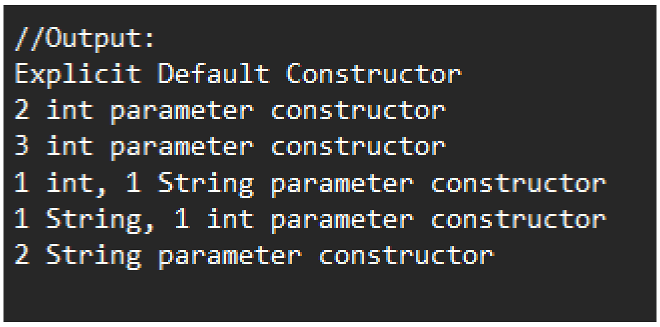
- Ways to overload a constructor:
  - Defining multiple parameterized constructors with different number of parameters.
  - Defining multiple parameterized constructors with different types of parameters.
  - Defining multiple parameterized constructors with different order of parameters.

- Combination of any two or all three ways can also be used to overload a constructor.

Constructor Overloading Example



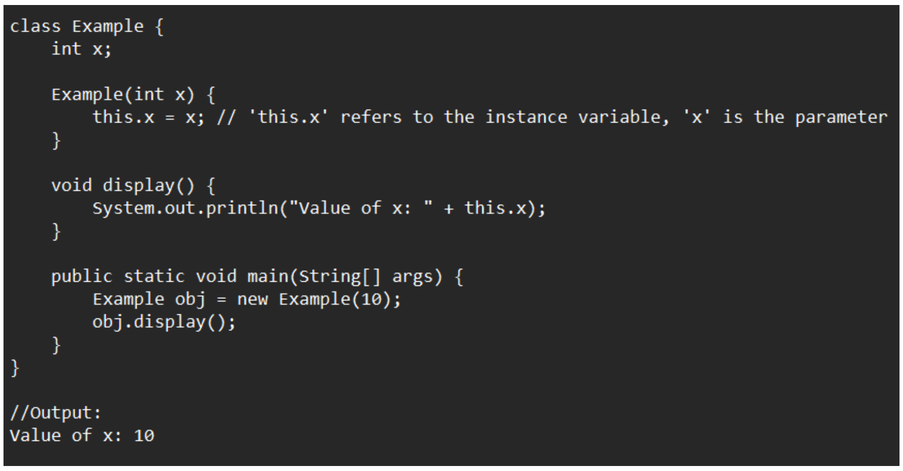
```java
public class Demo{
	public static void main(String[] args){
		Example e1 = new Example();
		Example e2 = new Example(1,2);
		Example e3 = new Example(1,2,3);
		Example e4 = new Example(1,"Hello");
		Example e5 = new Example("Hello",1);
		Example e6 = new Example("Hello","Hi");
	}
}

class Example{
	Example(){
		System.out.println("Explicit Default Constructor");
	}

	Example(int a, int b, int c){
		System.out.println("3 int parameter constructor");
	}

	Example(int a, int b){
		System.out.println("2 int parameter constructor");
	}

	Example(String s1, String s2){
		System.out.println("2 String parameter constructor");
	}

	Example(int a, String s){
		System.out.println("1 int, 1 String parameter constructor");
	}

	Example(String s, int a){
		System.out.println("1 String, 1 int parameter constructor");
	}

}
```



```
//Output:
Explicit Default Constructor
2 int parameter constructor
3 int parameter constructor
1 int, 1 String parameter constructor
1 String, 1 int parameter constructor
2 String parameter constructor
```

this Keyword

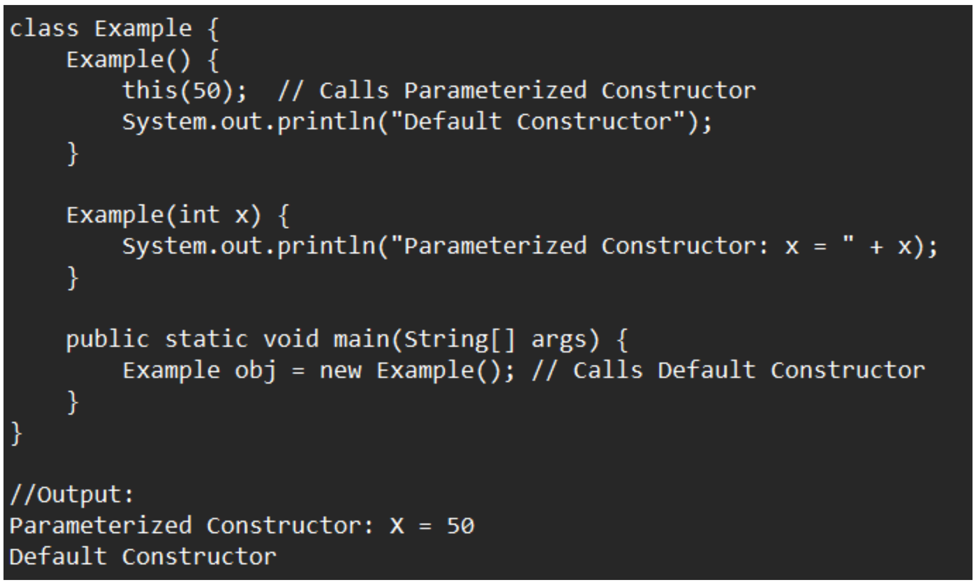
- The this keyword in Java is a reference variable that refers to the current object of a class.
- It is used in various scenarios to differentiate between instance variables, call constructors, and invoke methods.
- In Java, “this” in Java is a keyword that refers to the current object instance.
- It can be used to call current class methods and fields, to pass an instance of the current class as a parameter, and to differentiate between the local and instance variables.
- Using “this” reference can improve code readability and reduce naming conflicts.
- In Java, this is a reference variable that refers to the current object on which the method or constructor is being invoked. It can be used to access instance variables and methods of the current object.
- Uses of this Keyword
  - Referring to Instance Variables (To avoid name conflicts)
  - Calling Another Constructor (Constructor Chaining)
  - Calling a Method of the Same Class
  - Returning the Current Object
  - Passing the Current Object as a Parameter
- **Referring to Instance Variables:** When instance variables and method parameters have the same name, this helps to distinguish between them.



```
class Example {
    int x;

    Example(int x) {
        this.x = x; // 'this.x' refers to the instance variable, 'x' is the parameter
    }

    void display() {
        System.out.println("Value of x: " + this.x);
    }

    public static void main(String[] args) {
        Example obj = new Example(10);
        obj.display();
    }
}

//Output:
Value of x: 10
```

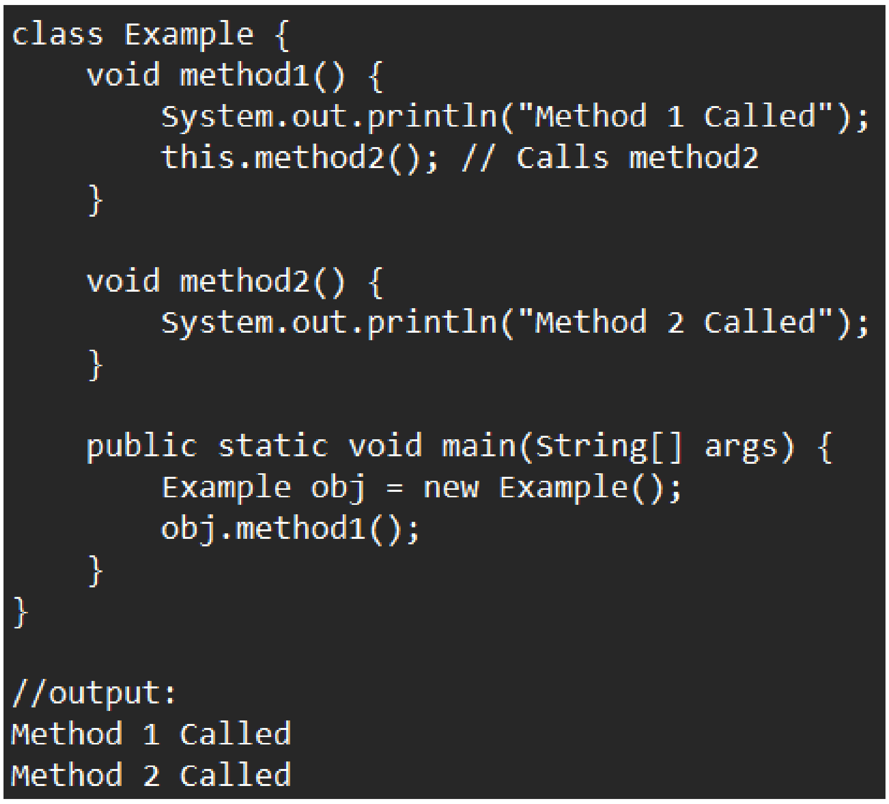
this Keyword

- **Calling Another Constructor (Constructor Chaining): this()** is used to call another constructor within the same class.



```
class Example {
    Example() {
        this(50);  // Calls Parameterized Constructor
        System.out.println("Default Constructor");
    }

    Example(int x) {
        System.out.println("Parameterized Constructor: x = " + x);
    }

    public static void main(String[] args) {
        Example obj = new Example(); // Calls Default Constructor
    }
}

//Output:
Parameterized Constructor: X = 50
Default Constructor
```

- **Calling a Method of the Same Class:** We can use **this** to call another method from the same class. Using this to call another method within the same class helps in code clarity, chaining method calls, and ensuring proper execution flow. However, it is not always required because methods in the same class can be called directly.



```
class Example {
    void method1() {
        System.out.println("Method 1 Called");
        this.method2(); // Calls method2
    }

    void method2() {
        System.out.println("Method 2 Called");
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.method1();
    }
}

//output:
Method 1 Called
Method 2 Called
```

this Keyword

- **Returning the Current Object:** The this keyword can return the current object. Using this to return the current object is useful in method chaining, fluent APIs, and passing the object for further modifications.

```
class Example {
    Example getObject() {
        return this;  // Returning current instance
    }

    void display() {
        System.out.println("Current Object Called");
    }

    public static void main(String[] args) {`
        Example obj = new Example();
        obj.getObject().display();
    }
}

//Output:
Current object Called
```

- **Passing the Current Object as a Parameter:** this can be passed as a parameter to another method.

```
class Example {
    void display(Example obj) {
        System.out.println("Current Object Passed");
    }

    void call() {
        display(this); // Passing current instance
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.call();
    }
}

//Output:
Current Object Passed
```

- When Calling an Overridden Method in a Subclass
  If a subclass overrides a method, this.method() ensures that the current class's version of the method is called instead of an inherited method.

this Keyword

```
class Parent {
    void show() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent {
    void show() {
        System.out.println("Child Method");
    }

    void display() {
        this.show();  // Calls the overridden method of this class
    }

    public static void main(String[] args) {
        Child obj = new Child();
        obj.display();  // Calls Child's show() method
    }
}

//Output: Child Method
```

- Summary:
  - this refers to the current object of a class.
  - Used to differentiate between instance and local variables.
  - Helps in constructor chaining and method invocation.
  - Can return or pass the current object as an argument.

Constructor Chaining

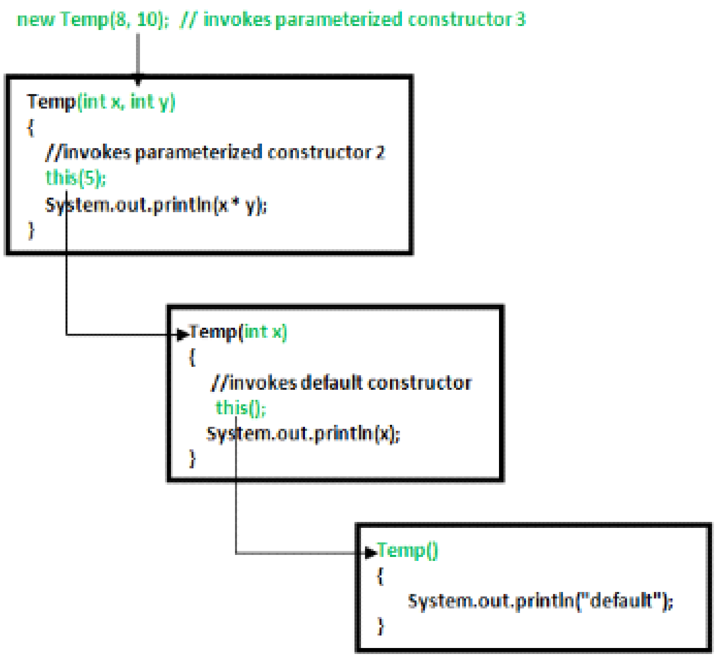
- Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

- One of the main use of constructor chaining is to avoid duplicate codes while having multiple constructor (by means of constructor overloading) and make code more readable.

- Constructor chaining can be done in two ways:
    - **Within same class:** It can be done using this() keyword for constructors in the same class

    - **From base class:** by using super() keyword to call the constructor from the base class.

- Need of constructor chaining
    - This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.



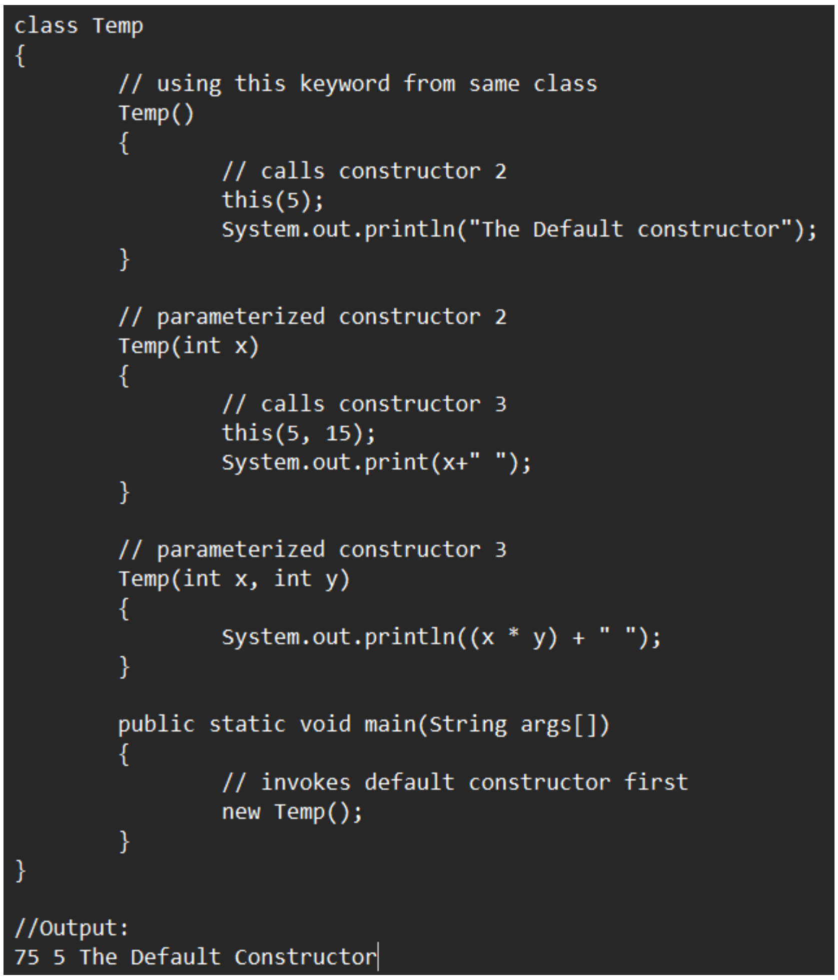
```
new Temp(8, 10);  // invokes parameterized constructor 3
```

```
Temp(int x, int y)
{
   //invokes parameterized constructor 2
   this(5);
   System.out.println(x * y);
}
```

```
Temp(int x)
{
   //invokes default constructor
   this();
   System.out.println(x);
}
```

```
Temp()
{
   System.out.println("default");
}
```

- **Constructor Chaining within the same class using this() keyword:**

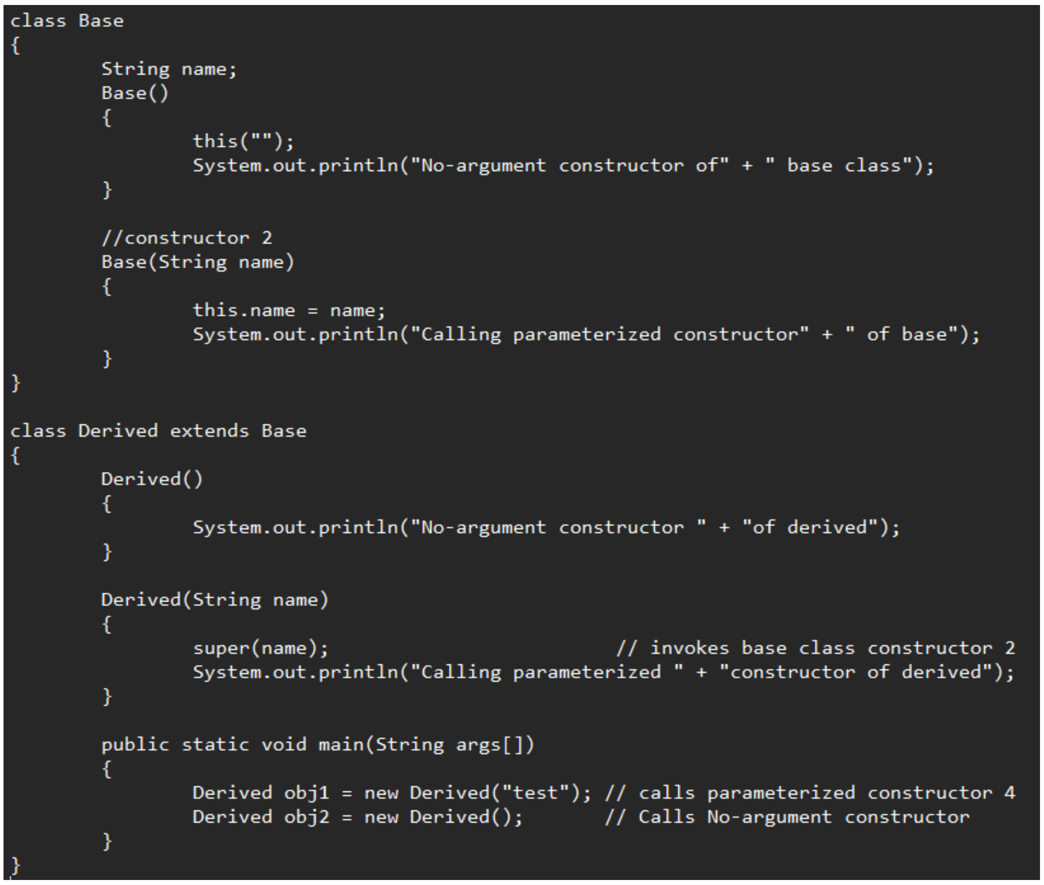


```
class Temp
{
        // using this keyword from same class
        Temp()
        {
                // calls constructor 2
                this(5);
                System.out.println("The Default constructor");
        }

        // parameterized constructor 2
        Temp(int x)
        {
                // calls constructor 3
                this(5, 15);
                System.out.print(x+" ");
        }

        // parameterized constructor 3
        Temp(int x, int y)
        {
                System.out.println((x * y) + " ");
        }

        public static void main(String args[])
        {
                // invokes default constructor first
                new Temp();
        }
}

//Output:
75 5 The Default Constructor
```

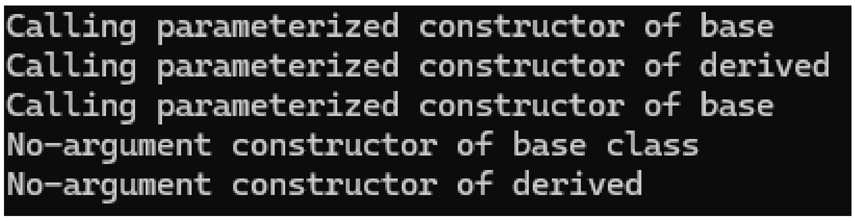
- **Rules of constructor chaining :**
  - The this() expression should always be the first line of the constructor.
  - There should be at-least be one constructor without the this() keyword.
  - Constructor chaining can be achieved in any order.

- **What happens if we change the order of constructors?**
  - Nothing, Constructor chaining can be achieved in any order

- **Constructor Chaining to other class using super() keyword :**



```java
class Base
{
        String name;
        Base()
        {
                this("");
                System.out.println("No-argument constructor of" + " base class");
        }

        //constructor 2
        Base(String name)
        {
                this.name = name;
                System.out.println("Calling parameterized constructor" + " of base");
        }
}

class Derived extends Base
{
        Derived()
        {
                System.out.println("No-argument constructor " + "of derived");
        }

        Derived(String name)
        {
                super(name);                         // invokes base class constructor 2
                System.out.println("Calling parameterized " + "constructor of derived");
        }

        public static void main(String args[])
        {
                Derived obj1 = new Derived("test"); // calls parameterized constructor 4
                Derived obj2 = new Derived();       // Calls No-argument constructor
        }
}
```



```
Calling parameterized constructor of base
Calling parameterized constructor of derived
Calling parameterized constructor of base
No-argument constructor of base class
No-argument constructor of derived
```

  - **Note:** Similar to constructor chaining in same class, super() should be the first line of the constructor as super class's constructor are invoked before the sub class's constructor.

Pass by reference

- In Java, pass by reference means passing the reference (or memory address) of an object to a method, rather than passing a copy of the actual object.

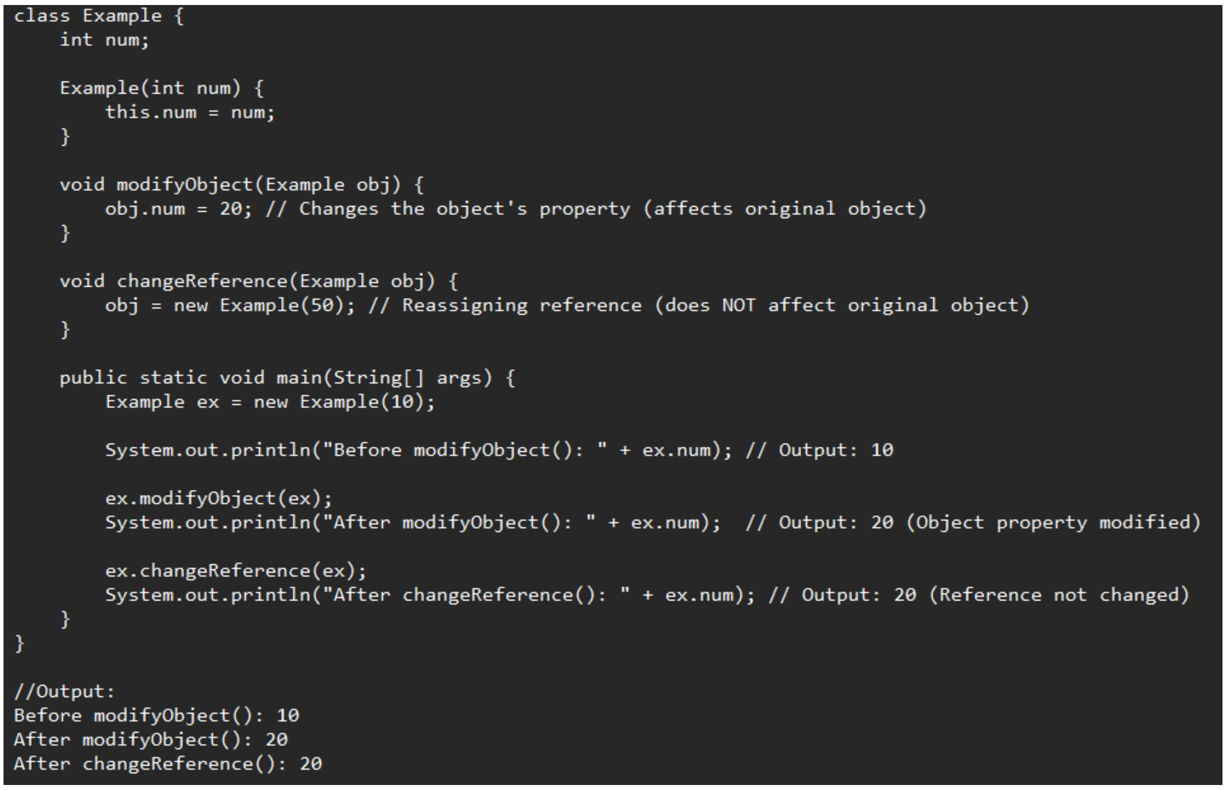
- However, Java **does not support** pass by reference. Java **strictly follows pass by value**, even for object references. This means that when an object is passed to a method, a copy of the reference (not the object itself) is passed.

- The method receives a copy of the reference, allowing changes to the object but not to the reference itself.

- If you reassign the reference inside a method, it does not affect the original object outside the method.

- Why Java Does Not Support Pass by Reference:
    - Primitive Types (int, double, char, etc.)
        - Passed by value: A copy of the actual value is passed to the method.
        - Changes inside the method do not affect the original variable.
    - Objects (Reference Types)
        - The reference (memory address) of the object is passed by value.
        - The method gets a copy of the reference, not the actual reference.
        - Both the original and copied references point to the same object, so modifying the object's properties inside the method affects the original object.
        - However, if you reassign the reference inside the method, it does not change the original reference outside.

Pass by reference Example



```
class Example {
    int num;

    Example(int num) {
        this.num = num;
    }

    void modifyObject(Example obj) {
        obj.num = 20; // Changes the object's property (affects original object)
    }

    void changeReference(Example obj) {
        obj = new Example(50); // Reassigning reference (does NOT affect original object)
    }

    public static void main(String[] args) {
        Example ex = new Example(10);

        System.out.println("Before modifyObject(): " + ex.num); // Output: 10

        ex.modifyObject(ex);
        System.out.println("After modifyObject(): " + ex.num);  // Output: 20 (Object property modified)

        ex.changeReference(ex);
        System.out.println("After changeReference(): " + ex.num); // Output: 20 (Reference not changed)
    }
}

//Output:
Before modifyObject(): 10
After modifyObject(): 20
After changeReference(): 20
```

- Java does not support true pass-by-reference , but it passes object references by value .
- Since both references point to the same object , modifications inside the method reflect in the original object.

Abstraction

- Abstraction is one of the core concepts of Object-Oriented Programming (OOP) in Java. It is used to hide implementation details and only show the essential features of an object.
- Key Points about Abstraction:
  - Hides Implementation: Only relevant details are exposed, while the internal workings are hidden.

  - It is implemented using Abstract Classes & Interfaces. Java provides two ways to achieve abstraction:
    - Abstract Class (0% to 100% abstraction)
    - Interface (100% abstraction in older versions, but from Java 8+, it can have default and static methods)
- **Abstraction Using Abstract Class**
  - An abstract class is a class that cannot be instantiated and is used to provide a common structure for its subclasses. It can have both:
    - Abstract methods (methods without a body, to be implemented by subclasses).
    - Concrete methods (methods with a body, which subclasses can use or override).

  - Why Use Abstract Classes?
    - **Code Reusability:** Common functionality can be defined once and shared by multiple subclasses.
    - **Enforce a Contract:** Ensures that all subclasses implement specific methods.
    - **Achieve Partial Abstraction:** Unlike interfaces (which enforce 100% abstraction in older Java versions), abstract classes allow both abstract and concrete methods.
    - **Support for Polymorphism:** Allows dynamic method dispatch (runtime polymorphism).

Abstraction - abstract keyword

- In Java, abstract is a non-access modifier in java applicable for classes, and methods but not variables. It is used to achieve abstraction.
- In Java, the abstract keyword is used to define abstract classes and methods. Here are some of its key characteristics:
  - **Abstract classes cannot be instantiated:** An abstract class is a class that cannot be instantiated directly. It is meant to be extended by other classes.
  - **Abstract methods do not have a body:** An abstract method is a method that does not have an implementation. It is declared using the abstract keyword and ends with a semicolon instead of a method body. Subclasses of an abstract class must provide a concrete implementation of all abstract methods defined in the parent class.
  - **Abstract classes can have both abstract and concrete methods (methods with implementation).**
  - **Abstract classes can have constructors:** Abstract classes can have constructors, which are used to initialize instance variables and perform other initialization tasks. However, because abstract classes cannot be instantiated directly, their constructors are typically called constructors in concrete subclasses.
  - **Abstract classes can contain instance variables:** Abstract classes can contain instance variables, which can be used by both the abstract class and its subclasses.
  - **Abstract classes can implement interfaces:** Abstract classes can implement interfaces. In this case, the abstract class must provide concrete implementations of all methods defined in the interface.
- By declaring a class or method as abstract, developers can provide a structure for subclassing and ensure that certain methods are implemented in a consistent way across all subclasses.

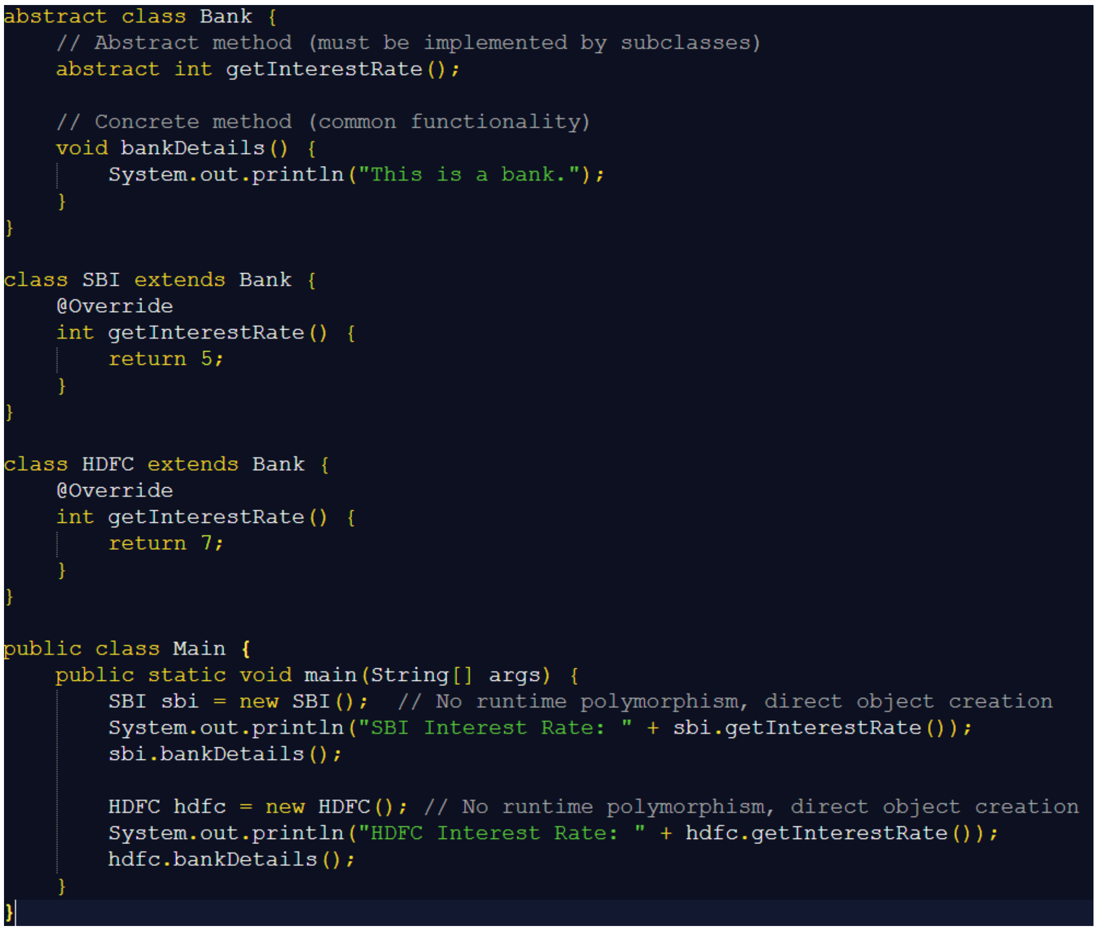
Abstraction - abstract Methods

- Sometimes, we require just method declaration in super-classes. This can be achieved by specifying the **abstract** type modifier.
- These methods are sometimes referred to as **subclass responsibility** because they have no implementation specified in the super-class. Thus, a **subclass must override** them to provide a method definition.
- To declare an abstract method, use this general form:

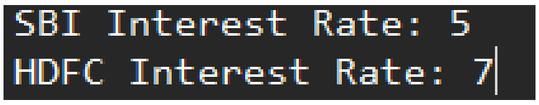
  abstract type method-name(parameter-list);

- **No method body** is present. Any concrete class(i.e. Normal class) that extends an abstract class must override all the abstract methods of the class.
- Any class that contains one or more abstract methods must also be declared abstract
- The following are various illegal combinations of other modifiers for methods with respect to abstract modifiers:
  - final
  - abstract native
  - abstract synchronized
  - abstract static
  - abstract private
- **E.g.:- void speed();**
- Advantages:
  - Provides a way to define a common interface
  - Enables polymorphism
  - Encourages code reuse
  - Provides a way to enforce implementation
  - Enables late binding
- In Java, you will never see a class or method declared with both final and abstract keywords. For classes, final is used to prevent inheritance whereas abstract classes depend upon their child classes for complete implementation. In cases of methods, final is used to prevent overriding whereas abstract methods need to be overridden in sub-classes.

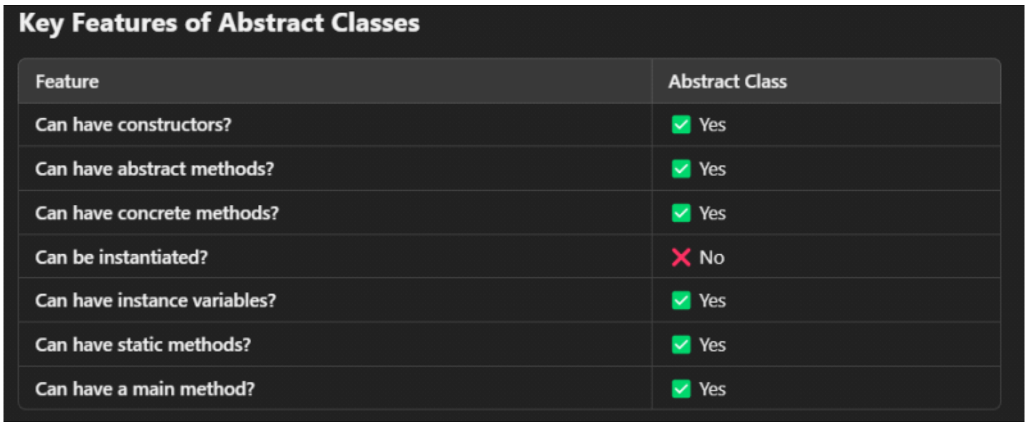
Abstraction



```java
abstract class Bank {
    // Abstract method (must be implemented by subclasses)
    abstract int getInterestRate();

    // Concrete method (common functionality)
    void bankDetails() {
        System.out.println("This is a bank.");
    }
}

class SBI extends Bank {
    @Override
    int getInterestRate() {
        return 5;
    }
}

class HDFC extends Bank {
    @Override
    int getInterestRate() {
        return 7;
    }
}

public class Main {
    public static void main(String[] args) {
        SBI sbi = new SBI();  // No runtime polymorphism, direct object creation
        System.out.println("SBI Interest Rate: " + sbi.getInterestRate());
        sbi.bankDetails();

        HDFC hdfc = new HDFC(); // No runtime polymorphism, direct object creation
        System.out.println("HDFC Interest Rate: " + hdfc.getInterestRate());
        hdfc.bankDetails();
    }
}
```



```
SBI Interest Rate: 5
HDFC Interest Rate: 7
```



Key Features of Abstract Classes

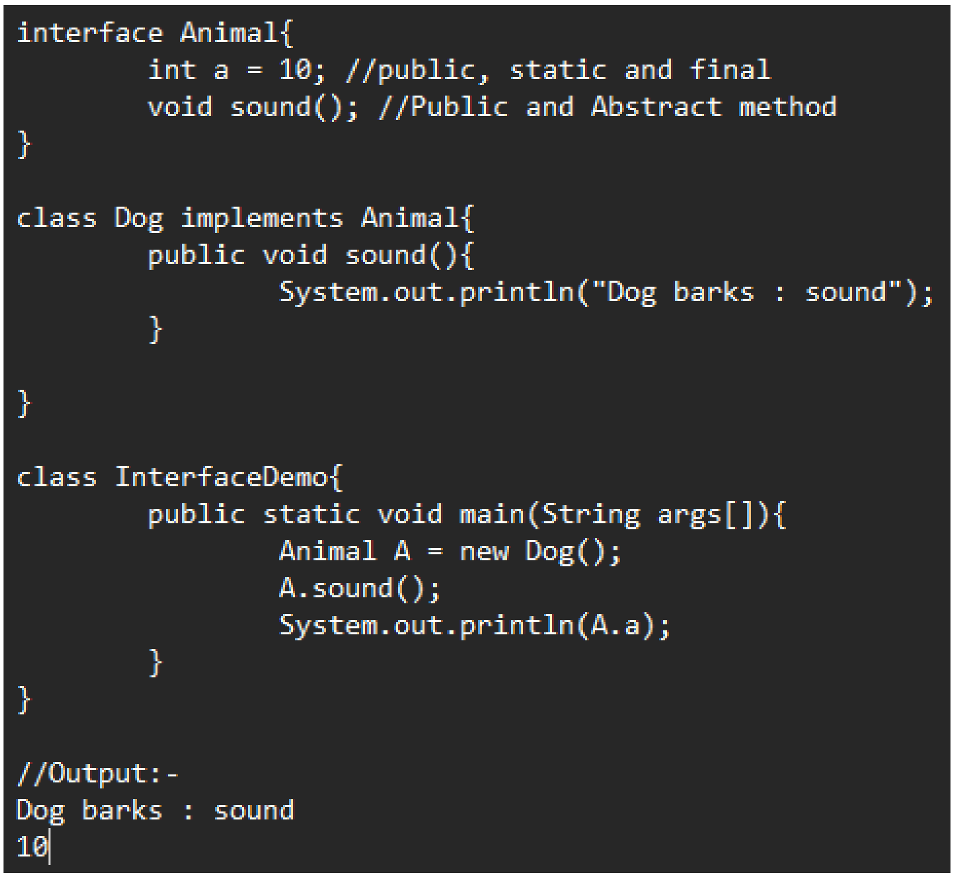
Feature	Abstract Class
Can have constructors?	✅ Yes
Can have abstract methods?	✅ Yes
Can have concrete methods?	✅ Yes
Can be instantiated?	❌ No
Can have instance variables?	✅ Yes
Can have static methods?	✅ Yes
Can have a main method?	✅ Yes

Abstraction - Interface

- **Using an Interface (100% Abstraction):**
  - An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods.
  - The interface in Java is a mechanism to **achieve abstraction** and **multiple inheritance**.
  - By default, variables in an interface are **public, static, and final**.
  - It is also used to achieve **loose coupling**.
  - In other words, interfaces primarily define methods that other classes must implement.
  - An interface in Java defines a set of behaviors that a class can implement, usually representing an **IS-A relationship**, but not always in every scenario.
  - To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body **(abstract)** and are **public** and all fields are **public, static,** and **final** by default. From **Java 8** onwards, interfaces can contain **default** methods and **static** methods with implementations. From **Java 9** onwards, interfaces can have **private** methods.
  - A class that implements an interface must implement all the methods declared in the interface. To implement the interface, use the implements keyword.
  - From **Java 8 onwards**, interfaces **do not provide 100% abstraction** because they can contain default and static methods with implementations.
  - Inside the Interface, **constructors are not allowed**. Inside the interface, **main method is not allowed**.
  - We can’t create an instance **(interface can’t be instantiated)** of the interface. A class can implement more than one interface. An interface can extend to another interface or interfaces (more than one interface).

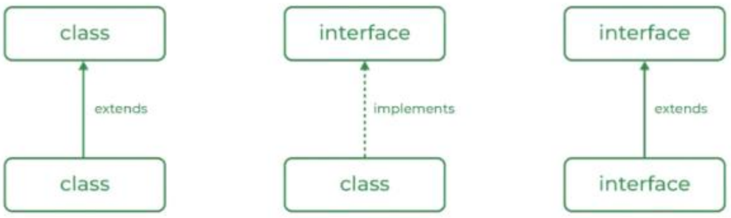
Abstraction - Interface

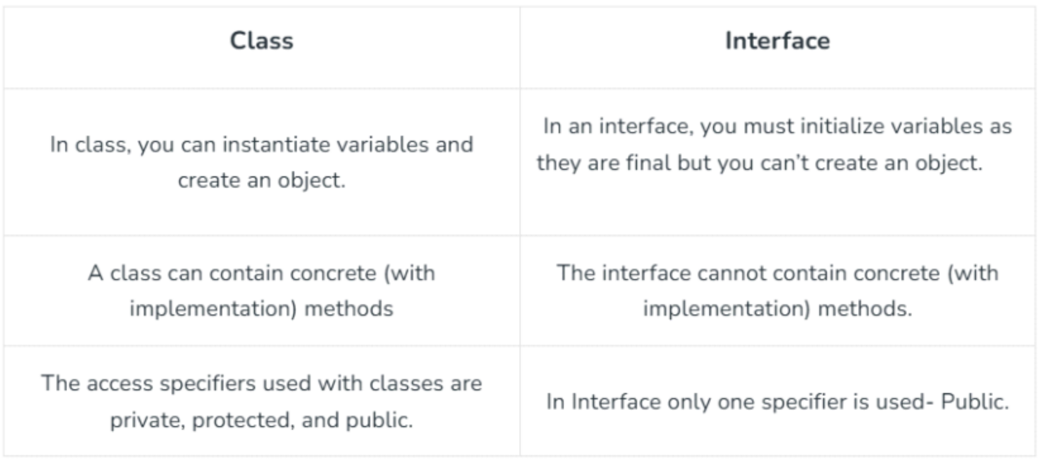
- Example:-



```
interface Animal{
        int a = 10; //public, static and final
        void sound(); //Public and Abstract method
}

class Dog implements Animal{
        public void sound(){
                System.out.println("Dog barks : sound");
        }

}

class InterfaceDemo{
        public static void main(String args[]){
                Animal A = new Dog();
                A.sound();
                System.out.println(A.a);
        }
}

//Output:-
Dog barks : sound
10
```

- **Relationship Between Class and Interface:** A class can extend another class, and similarly, an interface can extend another interface. However, only a class can implement an interface, and the reverse (an interface implementing a class) is not allowed.



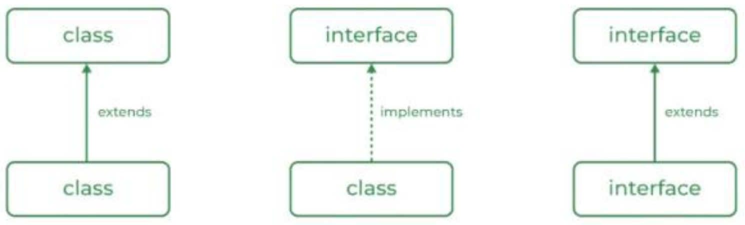


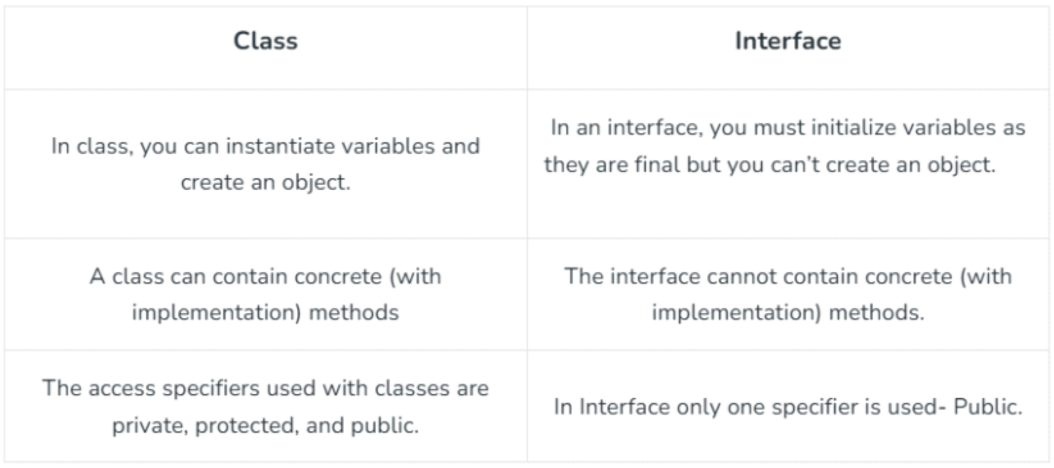
Class	Interface
In class, you can instantiate variables and create an object.	In an interface, you must initialize variables as they are final but you can't create an object.
A class can contain concrete (with implementation) methods	The interface cannot contain concrete (with implementation) methods.
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Abstraction - Nested Interfaces

- Example:-

- **Relationship Between Class and Interface:** A class can extend another class, and similarly, an interface can extend another interface. However, only a class can implement an interface, and the reverse (an interface implementing a class) is not allowed.

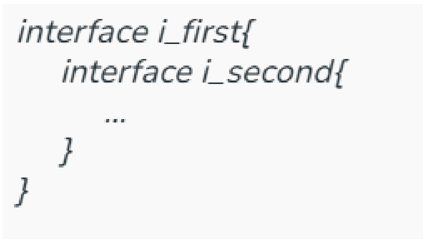




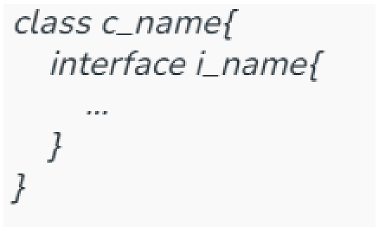
Class	Interface
In class, you can instantiate variables and create an object.	In an interface, you must initialize variables as they are final but you can't create an object.
A class can contain concrete (with implementation) methods	The interface cannot contain concrete (with implementation) methods.
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Abstraction - Interface

- We can declare interfaces as members of a class or another interface. Such an interface is called a member interface or nested interface.
- Interfaces declared outside any class can have only public and default (package-private) access specifiers.
- In Java, nested interfaces (interfaces declared inside a class or another interface) can be declared with the public, protected, package-private (default), or private access specifiers.
- A top-level interface (not nested) can only be declared as public or package-private (default). It cannot be declared as protected or private.
- Syntax:

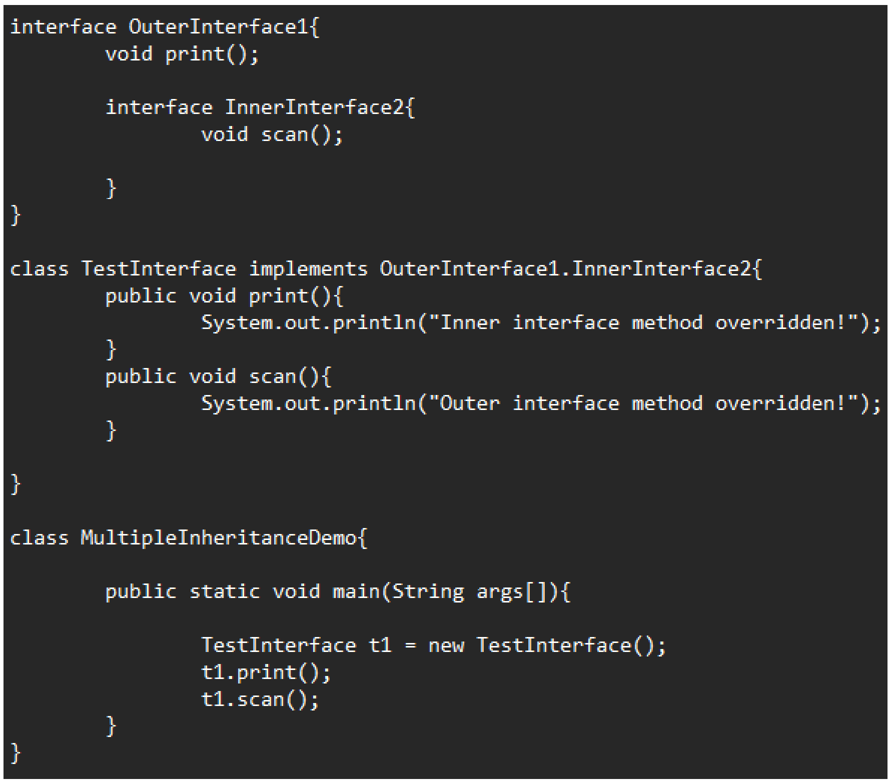


```
interface i_first{
    interface i_second{
        ...
    }
}
```

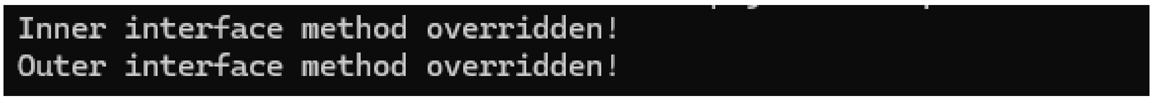


```
class c_name{
    interface i_name{
        ...
    }
}
```

- Example:-

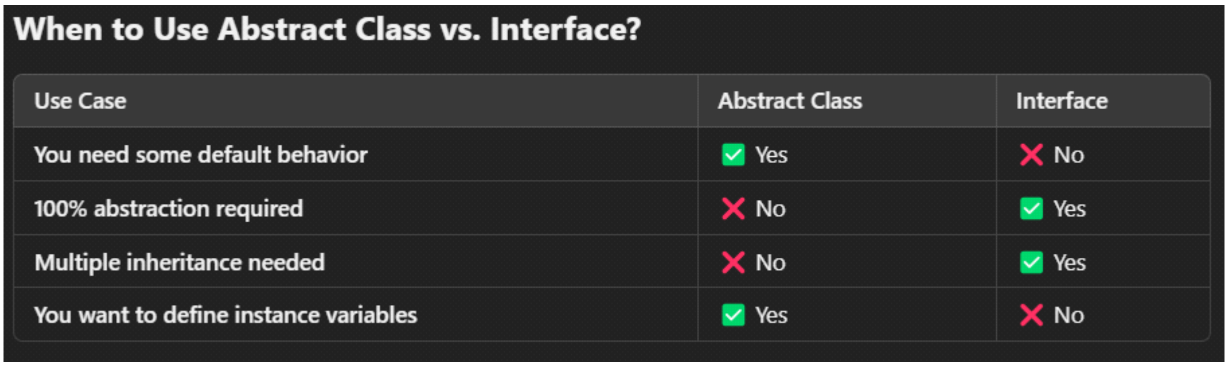


```
interface OuterInterface1{
        void print();

        interface InnerInterface2{
                void scan();

        }
}

class TestInterface implements OuterInterface1.InnerInterface2{
        public void print(){
                System.out.println("Inner interface method overridden!");
        }
        public void scan(){
                System.out.println("Outer interface method overridden!");
        }

}

class MultipleInheritanceDemo{

        public static void main(String args[]){

                TestInterface t1 = new TestInterface();
                t1.print();
                t1.scan();
        }
}
```



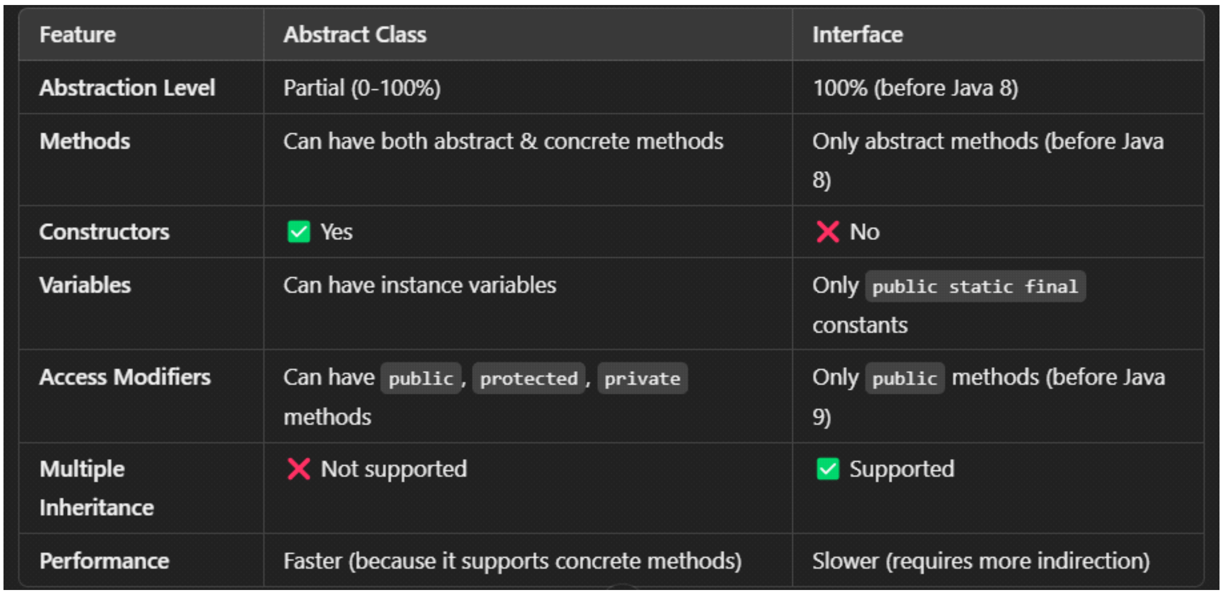
```
Inner interface method overridden!
Outer interface method overridden!
```

Abstract class vs Interface



When to Use Abstract Class vs. Interface?

Use Case	Abstract Class	Interface
You need some default behavior	✅ Yes	❌ No
100% abstraction required	❌ No	✅ Yes
Multiple inheritance needed	❌ No	✅ Yes
You want to define instance variables	✅ Yes	❌ No



Feature	Abstract Class	Interface
Abstraction Level	Partial (0-100%)	100% (before Java 8)
Methods	Can have both abstract & concrete methods	Only abstract methods (before Java 8)
Constructors	✅ Yes	❌ No
Variables	Can have instance variables	Only `public static final` constants
Access Modifiers	Can have `public`, `protected`, `private` methods	Only `public` methods (before Java 9)
Multiple Inheritance	❌ Not supported	✅ Supported
Performance	Faster (because it supports concrete methods)	Slower (requires more indirection)

Inheritance

- It is the mechanism in Java by which one class is allowed to acquire the features (fields and methods) of another class.

- In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

- Need of Inheritance in Java:
  - **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
  - **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
  - **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.

- **Super Class/Parent Class:** The class whose features are inherited is known as a **superclass** or a **base class** or a **parent class.**

- **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.

- The **extends** keyword is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, “extends” refers to increased functionality.

- Various types of inheritance that are supported in Java:
  - Single Inheritance (A → B)
  - Multilevel Inheritance (A → B → C)
  - Hierarchical Inheritance (A → B, A → C)
  - Hybrid Inheritance (one not involving multiple inheritance)

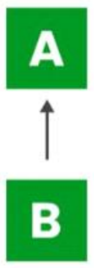
- Java doesn't support multiple inheritance as doing so might result in ambiguity for a compiler. Suppose there are 3 classes A, B, and C respectively. Class C inherits both A and B's properties and features. Both classes A and B have same method named display with same signature. Now when derived class, i.e. C tries to call display, confusion occurs as compiler can't decide which display method to call, leading to ambiguity. To eliminate this problem, Java doesn't support multiple inheritance using classes.

- This is the reason why Java does not directly support hybrid inheritance (a combination of multiple types of inheritance) because multiple inheritance with classes is not allowed. However, hybrid inheritance can be achieved using interfaces.

- We can still implement the concept of multiple inheritance with the help of interfaces because Interfaces do not cause conflicts like multiple class inheritance because they only contain method signatures (except for default and static methods from Java 8+).

- From Java 8+, If two interfaces contain default methods with the same name, a conflict arises. Java requires the implementing class to override the conflicting method.

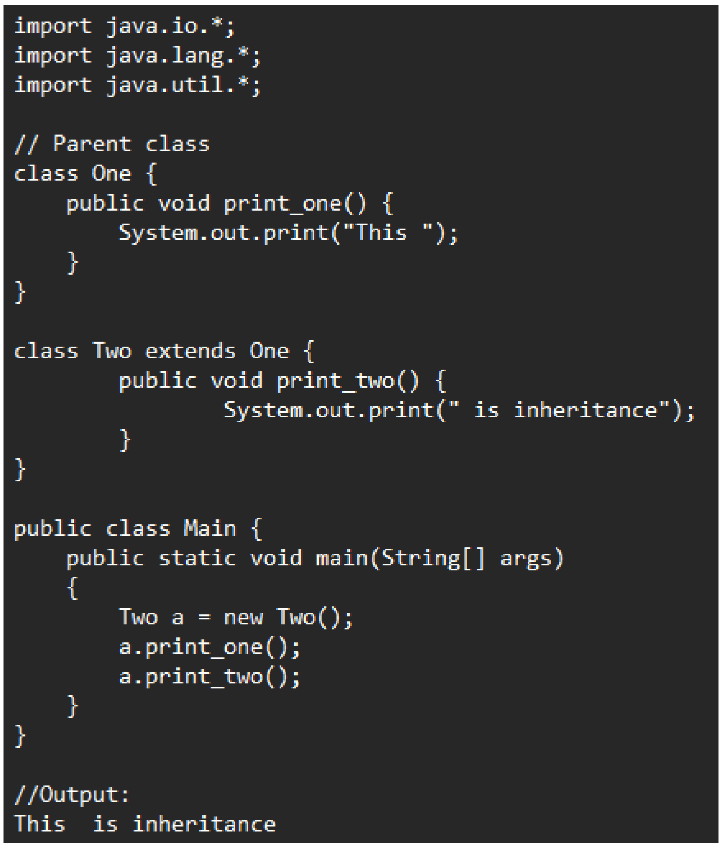
- Single Inheritance:
    - In single inheritance, a sub-class is derived from only one super class.
    - It inherits the properties and behaviour of a single-parent class.
    - It is also known as **simple inheritance**.
    - In the below figure, ‘A’ is a parent class and ‘B’ is a child class. The class ‘B’ inherits all the properties of the class ‘A’.



Single Inheritance

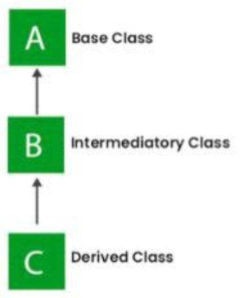
Single Inheritance, Multilevel Inheritance

- Example:



```
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class
class One {
    public void print_one() {
        System.out.print("This ");
    }
}

class Two extends One {
        public void print_two() {
                System.out.print(" is inheritance");
        }
}

public class Main {
    public static void main(String[] args)
    {
        Two a = new Two();
        a.print_one();
        a.print_two();
    }
}

//Output:
This  is inheritance
```

- **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.**



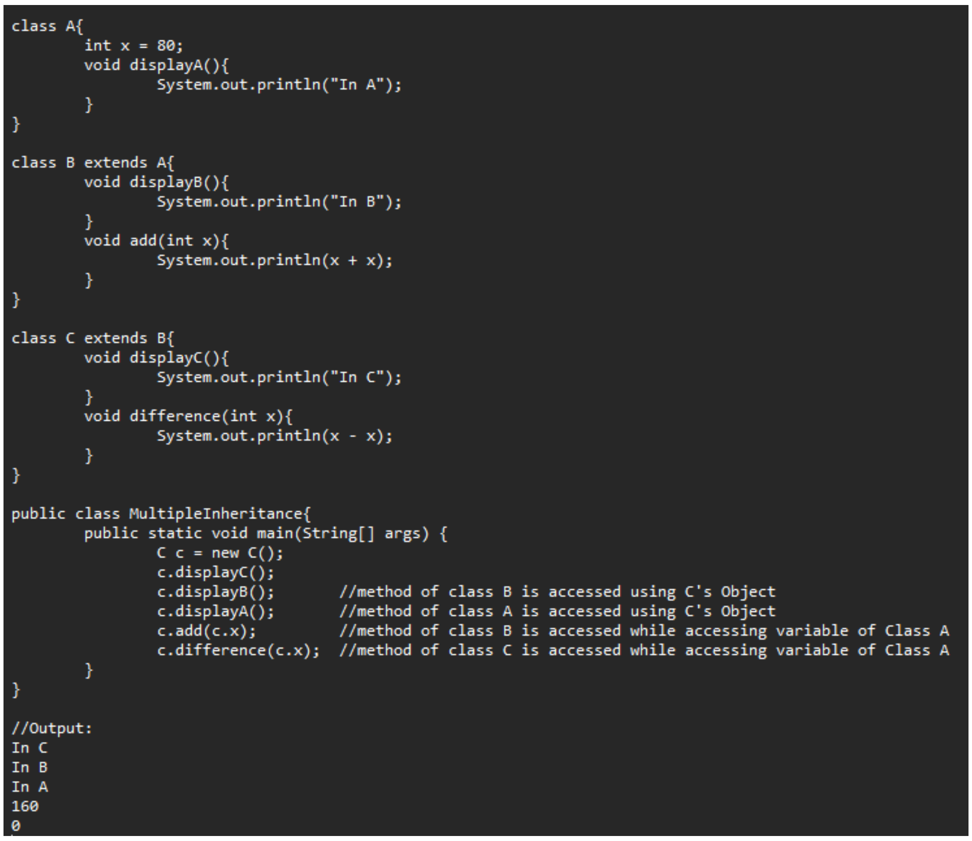
Multilevel Inheritance

- Whenever a new object of derived class is created, constructor of base class is called first, followed by constructor of intermediatory class, and at last constructor of derived class is called.

Constructor Execution Flow: Base --> Intermediatory --> Derived

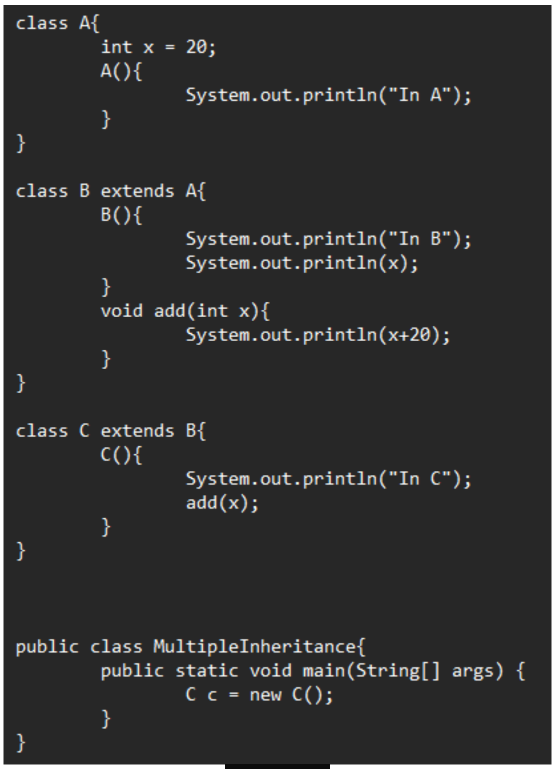
Multilevel Inheritance

- Example 1: Multiple Inheritance Demo

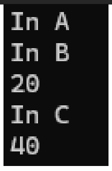


```
class A{
        int x = 80;
        void displayA(){
                System.out.println("In A");
        }
}

class B extends A{
        void displayB(){
                System.out.println("In B");
        }
        void add(int x){
                System.out.println(x + x);
        }
}

class C extends B{
        void displayC(){
                System.out.println("In C");
        }
        void difference(int x){
                System.out.println(x - x);
        }
}

public class MultipleInheritance{
        public static void main(String[] args) {
                C c = new C();
                c.displayC();
                c.displayB();       //method of class B is accessed using C's Object
                c.displayA();       //method of class A is accessed using C's Object
                c.add(c.x);         //method of class B is accessed while accessing variable of Class A
                c.difference(c.x);  //method of class C is accessed while accessing variable of Class A
        }
}

//Output:
In C
In B
In A
160
0
```

- Example 2: Constructor Execution Flow



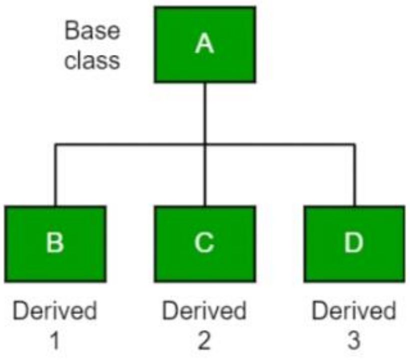
```
class A{
        int x = 20;
        A(){
                System.out.println("In A");
        }
}

class B extends A{
        B(){
                System.out.println("In B");
                System.out.println(x);
        }
        void add(int x){
                System.out.println(x+20);
        }
}

class C extends B{
        C(){
                System.out.println("In C");
                add(x);
        }
}



public class MultipleInheritance{
        public static void main(String[] args) {
                C c = new C();
        }
}
```



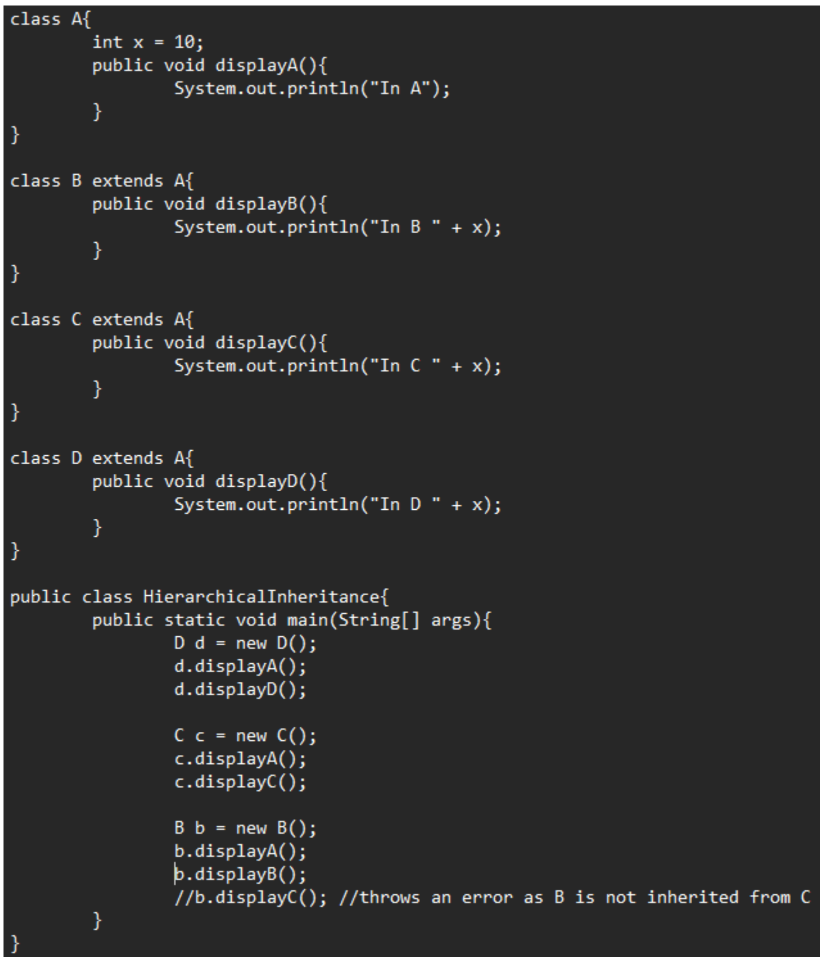
```
In A
In B
20
In C
40
```

Hierarchical Inheritance

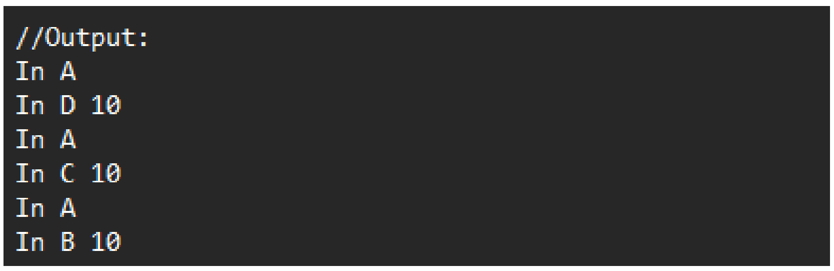
- **Hierarchical Inheritance:**
  - In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.
  - In the below image, class A serves as a base class for the derived classes B, C, and D.



  - Example:-



```
class A{
	int x = 10;
	public void displayA(){
		System.out.println("In A");
	}
}

class B extends A{
	public void displayB(){
		System.out.println("In B " + x);
	}
}

class C extends A{
	public void displayC(){
		System.out.println("In C " + x);
	}
}

class D extends A{
	public void displayD(){
		System.out.println("In D " + x);
	}
}

public class HierarchicalInheritance{
	public static void main(String[] args){
		D d = new D();
		d.displayA();
		d.displayD();

		C c = new C();
		c.displayA();
		c.displayC();

		B b = new B();
		b.displayA();
		b.displayB();
		//b.displayC(); //throws an error as B is not inherited from C
	}
}
```

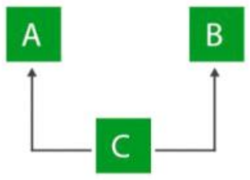


```
//Output:
In A
In D 10
In A
In C 10
In A
In B 10
```

Diamond Problem

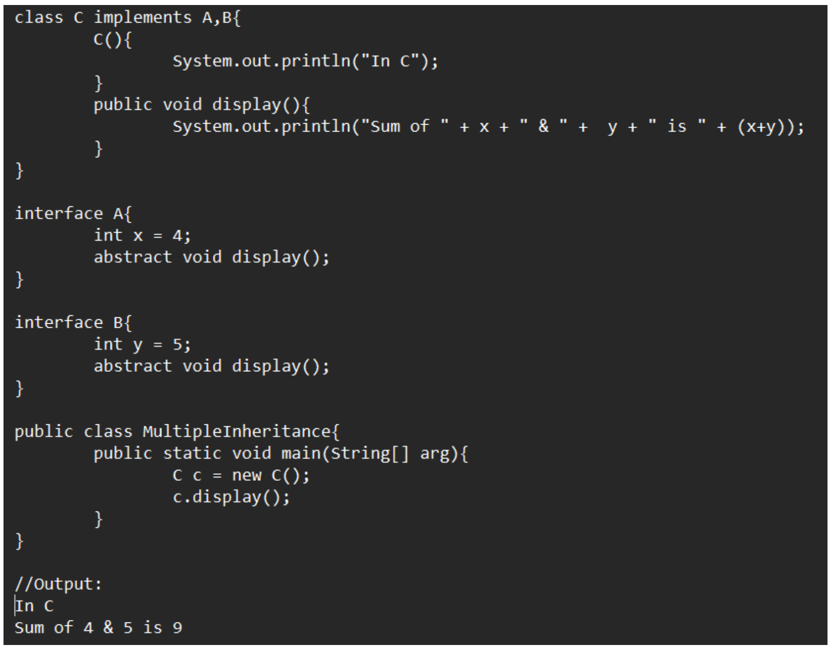
Multiple Inheritance

- **Multiple Inheritance:**
  - In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes.
  - Java doesn't support multiple inheritance as it might lead to ambiguity for a compiler.
  - Suppose there are 3 classes A, B, and C respectively. Class C inherits both A and B's properties and features. Both classes A and B have same method named display with same signature. Now when derived class, i.e. C tries to call display, confusion occurs as compiler can't decide which display method to call, leading to ambiguity. To eliminate this problem, Java doesn't support multiple inheritance using classes.
  - We can still implement the concept of multiple inheritance with the help of interfaces because Interfaces do not cause conflicts like multiple class inheritance because they only contain method signatures (except for default and static methods from Java 8+).



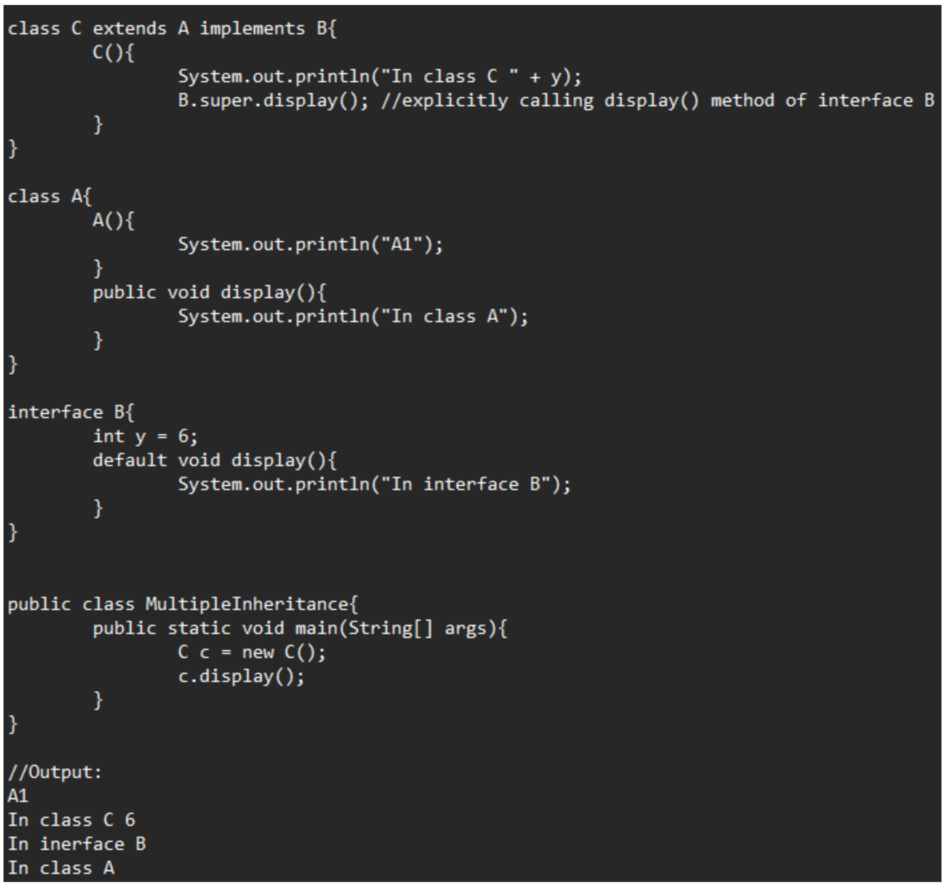
Multiple Inheritance

  - Example:- Implementation with 2 interfaces



```
class C implements A,B{
        C(){
                System.out.println("In C");
        }
        public void display(){
                System.out.println("Sum of " + x + " & " +  y + " is " + (x+y));
        }
}

interface A{
        int x = 4;
        abstract void display();
}

interface B{
        int y = 5;
        abstract void display();
}

public class MultipleInheritance{
        public static void main(String[] arg){
                C c = new C();
                c.display();
        }
}

//Output:
In C
Sum of 4 & 5 is 9
```

- **Multiple Inheritance:**
    - Example:- Implementation with a class and an interface

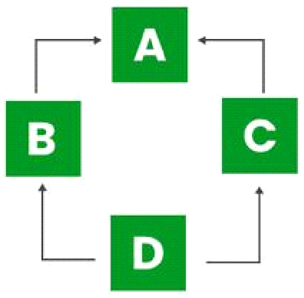


```
class C extends A implements B{
	C(){
		System.out.println("In class C " + y);
		B.super.display(); //explicitly calling display() method of interface B
	}
}

class A{
	A(){
		System.out.println("A1");
	}
	public void display(){
		System.out.println("In class A");
	}
}

interface B{
	int y = 6;
	default void display(){
		System.out.println("In interface B");
	}
}


public class MultipleInheritance{
	public static void main(String[] args){
		C c = new C();
		c.display();
	}
}

//Output:
A1
In class C 6
In inerface B
In class A
```

In above example, we have tried to implement multiple inheritance with the help of a class and an interface. While, this is a form of multiple inheritance, it isn't a true multiple inheritance as it is combination of single inheritance and multiple interface implementation. It is not full-fledged multiple inheritance like in C++ (which allows multiple class inheritance).

Hybrid Inheritance

- **Hybrid Inheritance:**
  - 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.



Hybrid Inheritance