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.

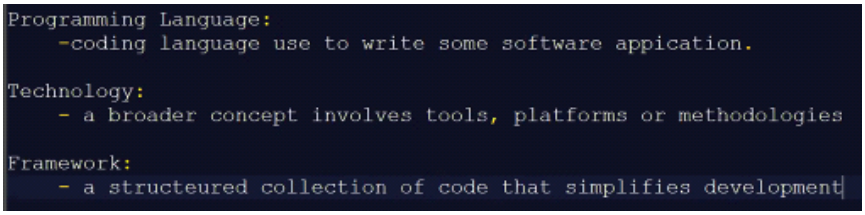
- Java is so popular because it is a **platform independent language, versatile language(used for wide range of applications), largest community support, simplicity, many opportunities available for Java developers in industry.**

- 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.



Features of Java

- **Platform Independent:** 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.

- **Object-Oriented Programming:** 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

- **Simplicity:** 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.

- **Robustness:** 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

- **Security:** 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.

- **Distributed:** 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.

- **Multithreading:** 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.

- **High Performance:**
  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.

- **Portable:**
  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.

- **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

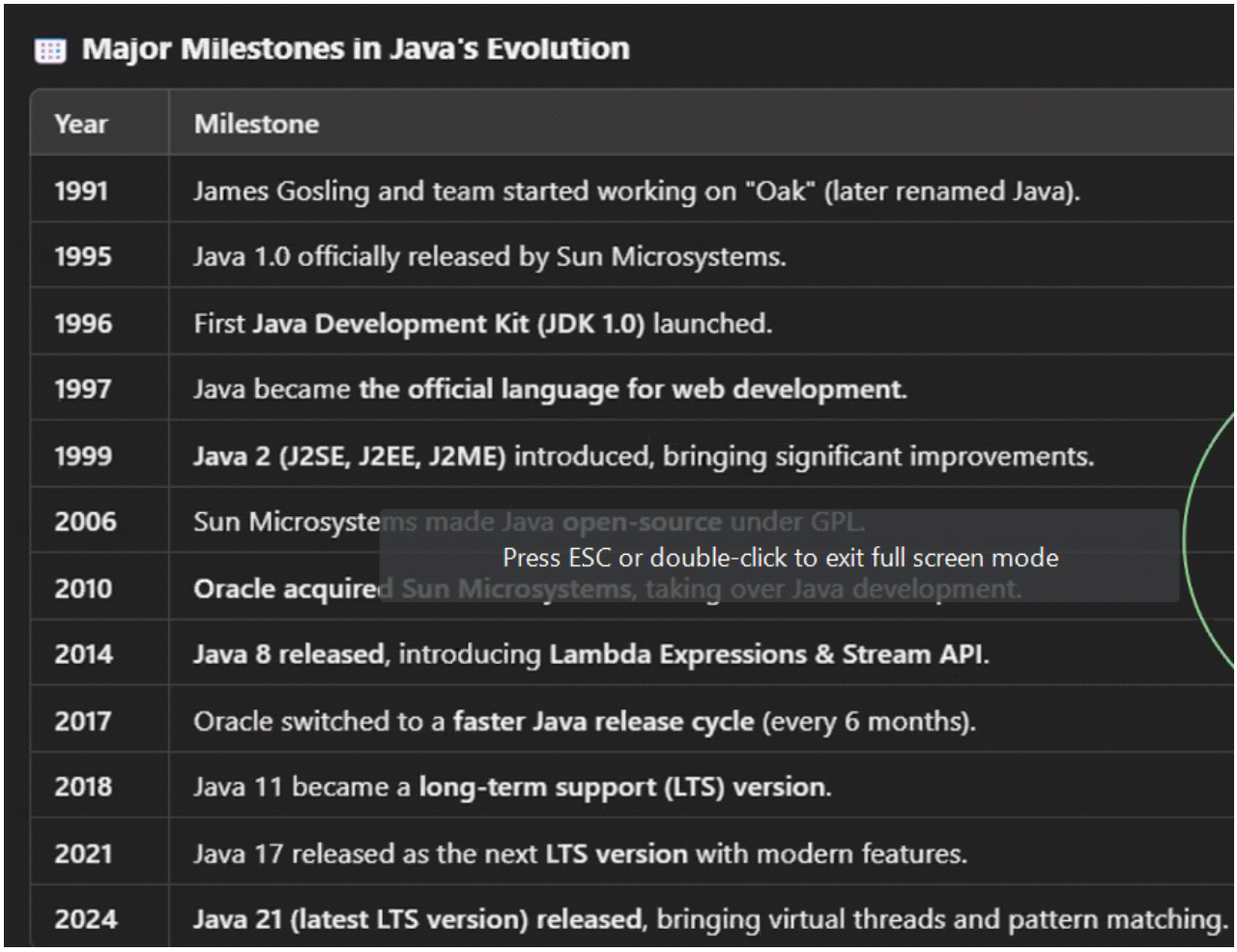
- **Nest Based Access Control:** Java 11 introduces the notion of nestmates and the associated access rules within the JVM. A nest of classes in Java implies both the outer/main class and all its nested classes. Nested classes are linked to the NestMembers attribute, while the outer class is linked to 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

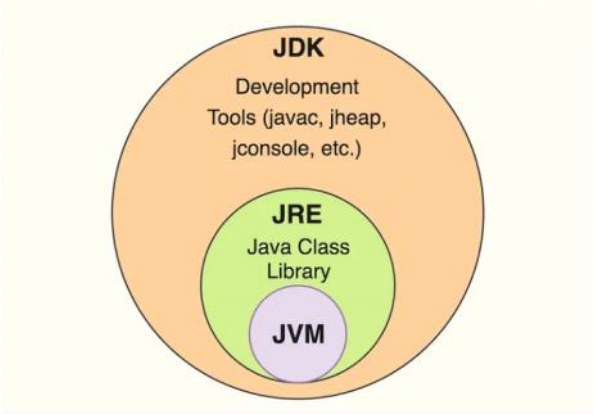


Major Milestones in Java's Evolution

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 Microsyste**ms made Java open-source under GPL.**
2010	**Oracle acquire**d 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.

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)



- **JDK:** 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.

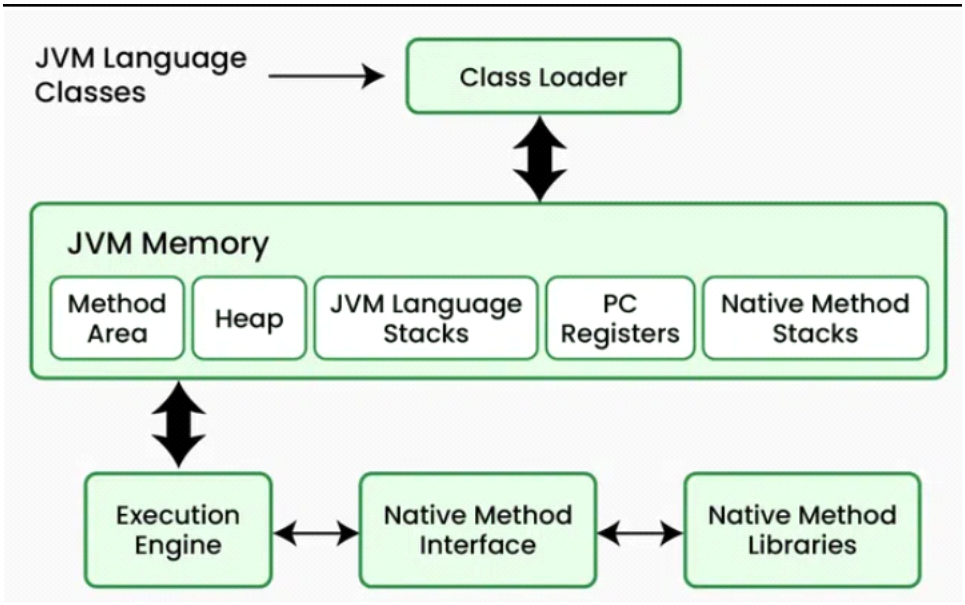
- **JRE:** JRE is a part of JDK and it builds a runtime environment where the Java program can be executed. It contains the libraries and software needed by the Java programs to run. It takes the Java code and integrates with the required libraries, and then starts the JVM to execute it. Based on the 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 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.



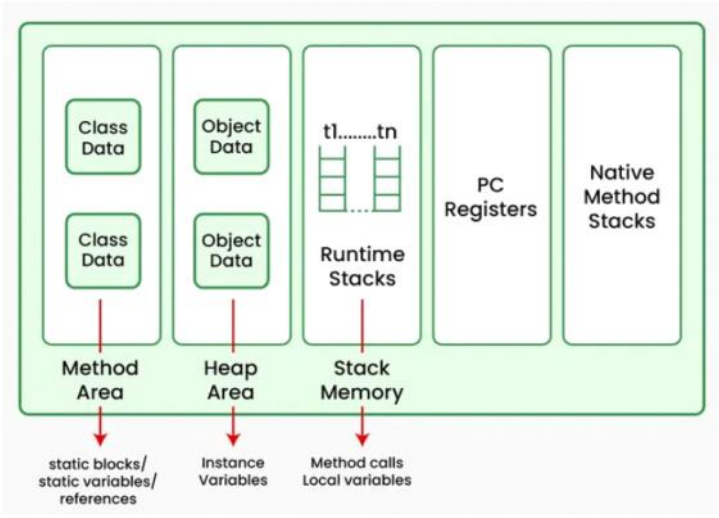
- JVM consists of three main subsystems:
  - Class Loader Subsystem
  - JVM Memory Areas
  - Native Method Interface
  - Execution Engine
  - Native Method Libraries

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

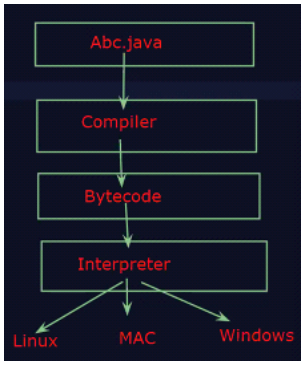
- **Class Loading Process:**
  - **Loading:** Reads .class files from disk or network.
  - **Linking:**
    - Verification: Ensures bytecode follows Java standards.
    - Preparation: Allocates memory for static variables.
    - Resolution: Converts symbolic references to actual memory locations.
  -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

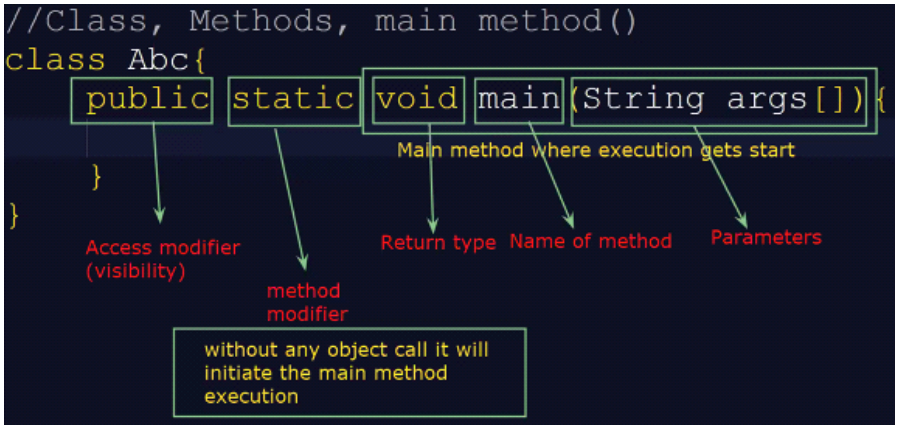
- **J2EE -Java Platform Enterprise Edition: (Advanced Java, Java for Web development, corporate purpose)**
  - It has concepts to develop software for Web applications, Enterprise applications, and Interoperable applications. These applications are called high-scale applications. Some examples are:-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

- **Main method Signature:**



- **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:**

```java
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.**

```java
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.

```
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
```

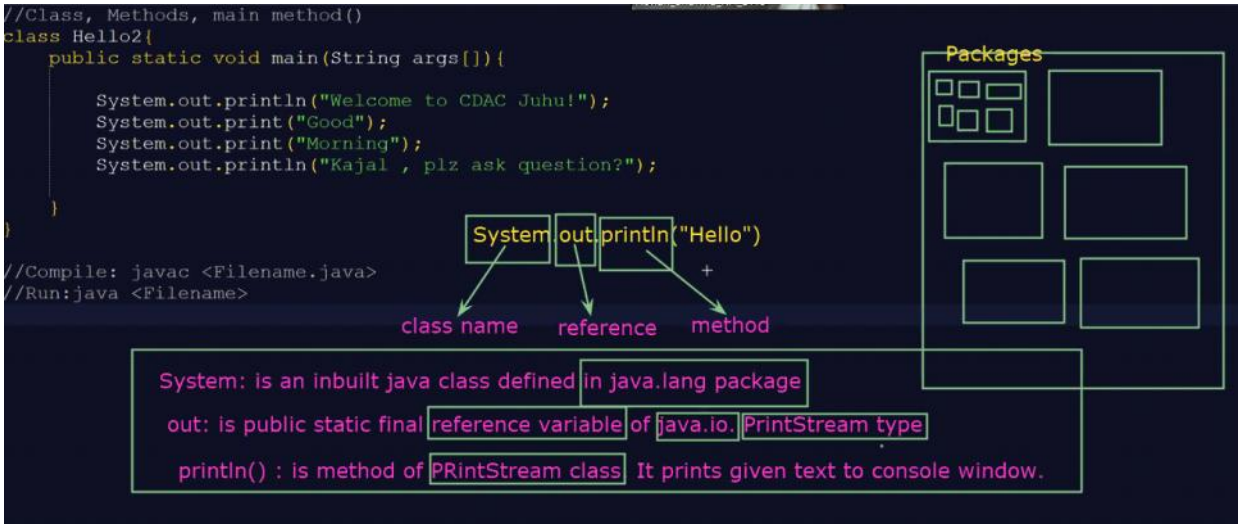
- **Note: If class is declared with public access modifier, class name containing main method must be same as file name.**

```java
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



- 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 (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.

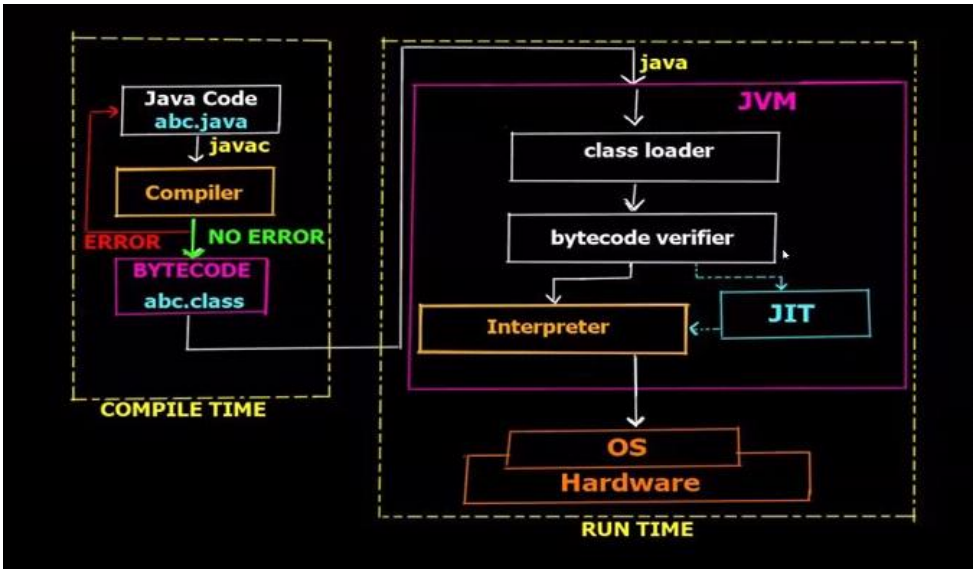


Fig. Java file execution

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:

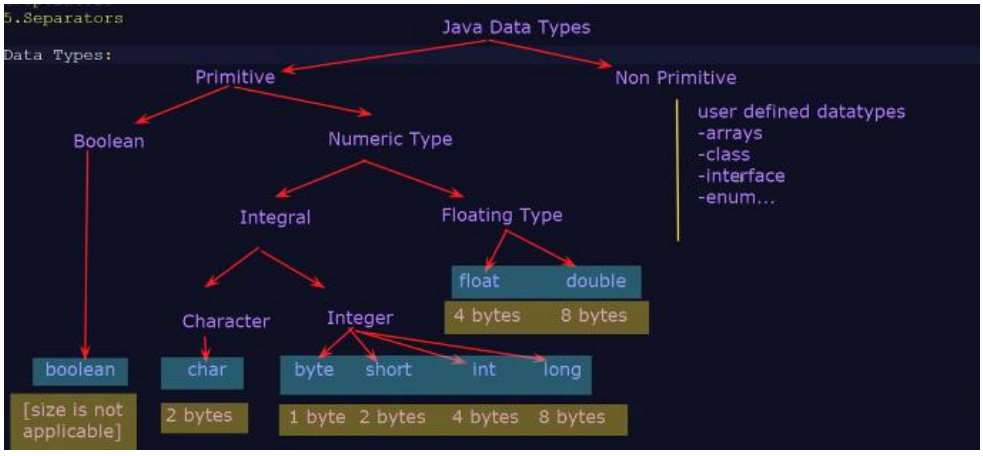
  - **Arithmetic Operators:** + (addition), - (subtraction), * (multiplication), / (division), % (modulus).
  - **Relational (Comparison) Operators:** == (equal to), != (not equal to), > (greater than), < (less than), >=(greater than or equal to), <=(lesser than or equal to).
  - **Logical Operators:** && (AND), || (OR), ! (NOT).
  - **Bitwise Operators:** &, |, ^, <<, >>.
  - **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.

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

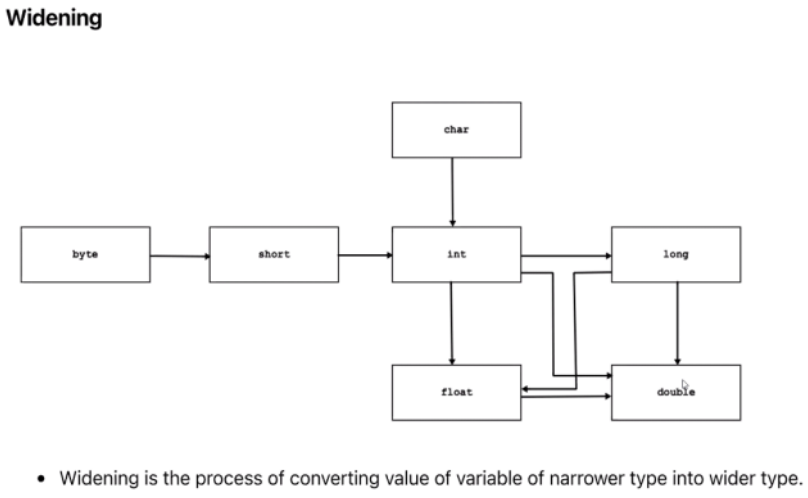
- **Array:**
  - An array is a group of similar data type variables that are commonly referred to by a common name.
  - In Java, all arrays are dynamically allocated.
  - Since arrays are objects in java, we can find their length using member length. This is different from C/C++ where we find length 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 is the process of converting value of variable of narrower type into wider type.

  - E.g.:-

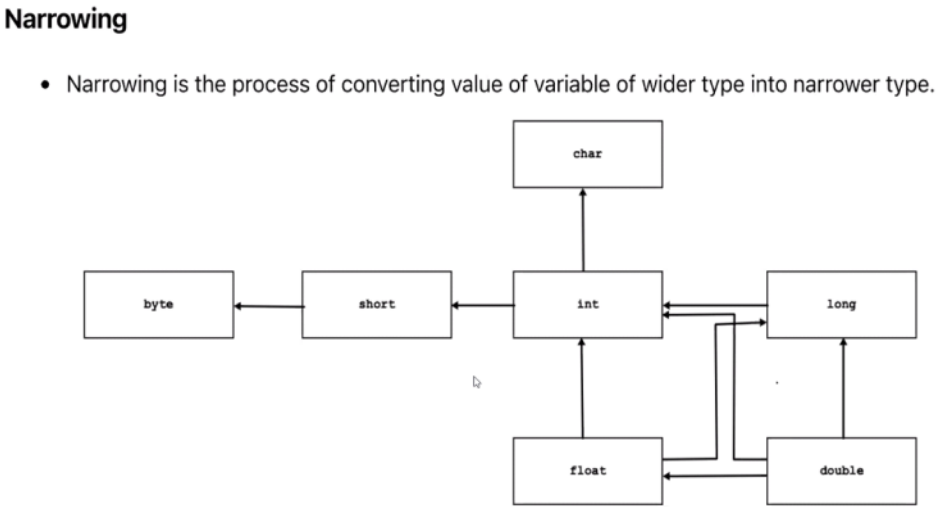
```
char ch = 'A'; //ASCII A=65
int i = ch;    //i = 65
System.out.println(i);
float f = i;   // f = 65.0
int num = 100;
double d = num; //d =100.0
System.out.println(f + " " + d);
```

```
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>



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
```