BE EXPERT IN JAVA

PART-1

By Ummed Singh

Ummed Singh

Be Expert in Java

Learn Java programming and become expert

BookRix GmbH & Co. KG
81371 Munich

Table of Content

Java Programming

Java serves as both a programming language and a platform. It is a sophisticated, secure, and object-oriented programming language. Sun Microsystems (now a subsidiary of Oracle) developed Java in 1995, and its conception is credited to James Gosling, often referred to as the father of Java. Originally named Oak, the language underwent a name change to Java when it was discovered that the name Oak was already claimed by another company. Regarding its platform designation, Java qualifies as such because it can operate within various hardware or software environments. This is due to its inclusion of a runtime environment (JRE) and API, which enables it to function as a self-sustained platform for running applications.

In Java, the "main" method serves as the entry point of the program and is executed when the program runs.

Here's a brief explanation of the code:

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

class Simple: This line begins the definition of the class named "Simple."

public static void main(String args[]): This line defines the main method, which is declared as "public," meaning it can be accessed from outside the class. The keyword "static" indicates that this method belongs to the class itself, rather than to an instance of the class. The keyword means that the method does not return any value. The parameter String args[] is an array of strings and is used to pass command-line arguments to the program. However, in this example, we are not utilizing the command-line arguments.

System.out.println("Hello Java");: This line prints the string "Hello Java" to the standard output (usually the console). The System.out.println method is used to display text on the screen, and in this case, it will print "Hello Java" followed by a newline.

When this Java program is executed, it will simply display "Hello Java" as the output. This is a common introductory example used in Java to demonstrate the basic structure of a Java program and how to print text to the console.

Application used in Java

Indeed, Java's versatility and widespread adoption have led it to be utilized across a vast array of devices and applications. Here are some examples of the various domains where Java finds application:

Desktop Applications: Java is commonly used to develop desktop applications like Adobe Acrobat Reader, media players, antivirus software, and more. Its "Write Once, Run Anywhere" capability allows these applications to be platform-independent, meaning they can run on different operating systems without modification.

Web Applications: Many web applications are powered by Java on the server-side. Websites like irctc.co.in and javatpoint.com may use Java in the backend to handle server-side logic, manage databases, and process user requests.

Enterprise Applications: Java is widely used in developing enterprise-level applications, especially in the banking and finance sector. These applications require robustness, scalability, and security, which Java provides.

Mobile Applications: Java was traditionally the primary language for developing Android applications. However, as of my knowledge cutoff in September 2021, Kotlin has become the recommended language for Android development. Nevertheless, Java still plays a significant role in the Android ecosystem.

Embedded Systems: Java is also utilized in embedded systems, which are specialized computing devices integrated into larger systems or products. Java's portability makes it suitable for various embedded applications.

Smart Cards: Java Card technology enables Java applications to run on smart cards, which are used for secure transactions and data storage in various applications.

Robotics: Java is applied in robotics for programming and controlling robots due to its object-oriented nature and support for concurrent programming.

Games: Java is used in game development, especially for mobile games. Game developers often utilize Java for creating interactive and entertaining gaming experiences.

Java's "Write Once, Run Anywhere" capability, along with its strong community support, extensive libraries, and mature ecosystem, has contributed to its widespread usage in various domains and on a diverse range of devices. As a result, the claim of 3 billion devices running Java, as stated by Sun Microsystems (prior to its acquisition by Oracle), demonstrates its significant impact on the world of technology.

Number of Java Applications

Java programming enables the creation of diverse application types, encompassing the following four categories:

Standalone Application:

Standalone applications, also known as window-based or desktop applications, are conventional software that necessitates installation on individual machines. Illustrative examples of standalone applications comprise media players, antivirus software, and more. For crafting standalone applications in Java, developers commonly employ AWT and Swing, which are libraries tailored for this purpose.

Web Application:

Web applications operate on the server-side and generate dynamic web pages. Employing technologies like Servlets, JSP (JavaServer Pages), Struts, Spring, Hibernate, JSF (JavaServer Faces), among others, developers construct web applications in Java. These applications enable users to interact with websites and perform various tasks over the internet.

Enterprise Application:

Enterprise applications exhibit a distributed nature, exemplified by banking applications and other similar systems. They are equipped with essential attributes such as robust security, load balancing, and clustering to cater to the complex demands of enterprises. Java's EJB (Enterprise JavaBeans) is specifically designed for creating such enterprise-level applications.

Mobile Application:

Mobile applications are designed specifically for mobile devices, offering functionality tailored to smartphones and tablets. Currently, Java ME (Micro Edition) and Android are the prominent technologies utilized for developing mobile applications in the Java ecosystem. These applications enhance the capabilities of mobile devices and provide users with a wide range of features and services.

Number of Java platforms

Java offers four distinct platforms or editions, each tailored to specific application domains:

Java SE (Java Standard Edition):

Java SE serves as the foundational Java platform for general-purpose programming. It encompasses a wide array of Java programming APIs, including java.lang, java.io, java.net, java.util, java.sql, java.math, and more. Developers utilize Java SE for core programming topics, such as Object-Oriented Programming (OOPs), String manipulation, Regular Expressions (Regex), Exception handling, Inner classes, Multithreading, Input/Output streams, Networking, Graphical User Interface (GUI) development with AWT and Swing, Reflection, Collections, and many others.

Java EE (Java Enterprise Edition):

Java EE is an enterprise-focused platform primarily employed for developing web and enterprise applications. Built on top of Java SE, Java EE extends the capabilities of the standard edition to cater to the demands of large-scale, distributed applications. Topics covered in Java EE include Servlets, JavaServer Pages (JSP), Web Services, Enterprise JavaBeans (EJB), Java Persistence API (JPA), and more. It offers features like robust security, transaction management, and scalability to meet the needs of enterprise-level systems.

Java ME (Java Micro Edition):

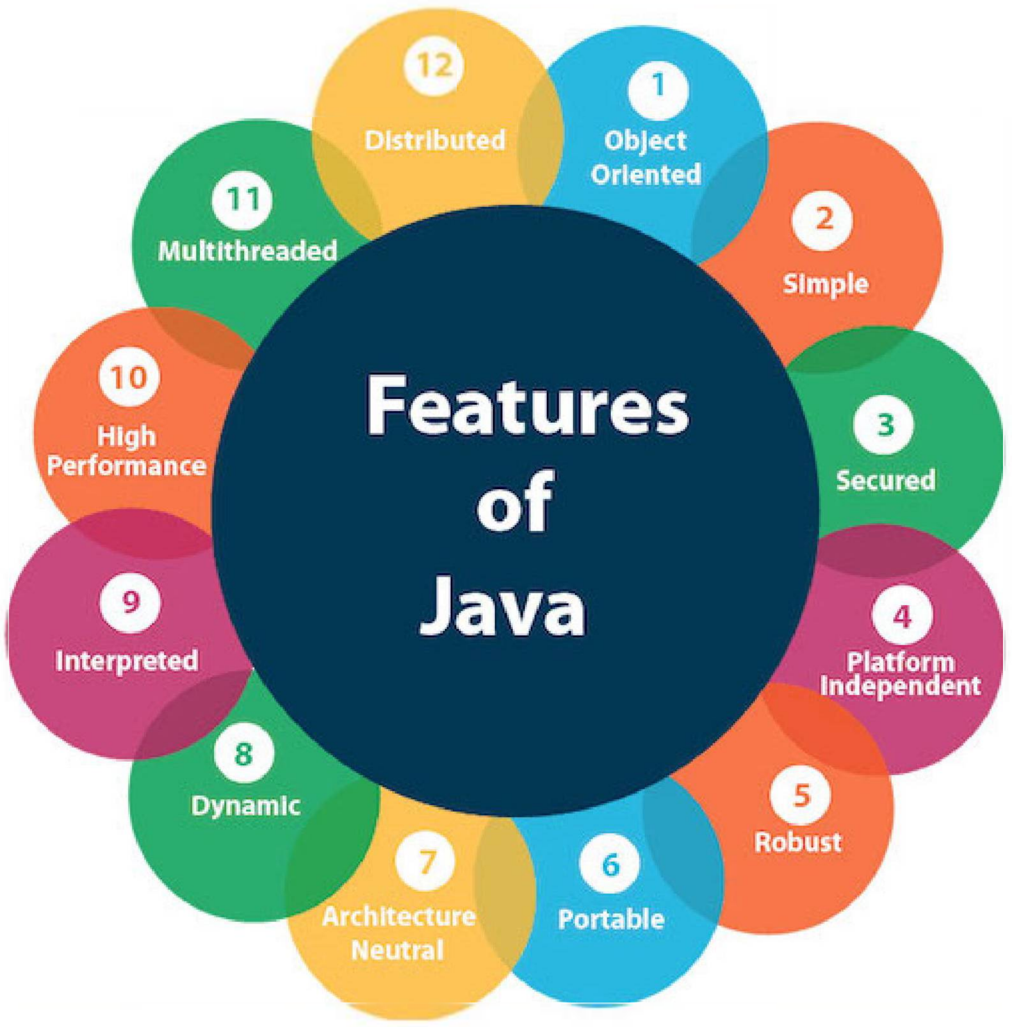
Java ME is a micro-platform designed specifically for mobile applications and resource-constrained devices. It enables developers to create applications for mobile phones, embedded systems, and other small-scale devices. Java ME focuses on optimization for limited resources while providing a subset of Java SE functionalities suitable for mobile environments.

JavaFX:

JavaFX is a platform dedicated to developing Rich Internet Applications (RIAs). It offers a lightweight user interface (UI) API for creating visually appealing and interactive web-based applications. JavaFX supports rich media, 2D and 3D graphics, animations, and multimedia components, making it ideal for creating engaging user experiences on various platforms.

Each platform serves distinct purposes, providing Java developers with a range of tools and capabilities to address a wide spectrum of application requirements, from desktop and web applications to mobile and rich internet applications.

Java Features



Indeed, Java's design philosophy prioritized key features to make it a popular and versatile programming language. The following are the essential features, often referred to as Java buzzwords, that contribute to Java's widespread adoption and success:

Simple:

Java is designed to be straightforward and easy to learn. It emphasizes clean syntax and reduces complexities, making it accessible to both beginner and experienced developers.

Object-Oriented:

Java follows the object-oriented programming (OOP) paradigm, promoting modularity and code reusability through the use of classes, objects, inheritance, and encapsulation.

Portable:

Java's "Write Once, Run Anywhere" (WORA) capability allows Java programs to be compiled into platform-independent bytecode. This bytecode can then be executed on any system with a Java Virtual Machine (JVM), enhancing portability.

Platform Independent:

Due to its portability, Java can run on any platform with a compatible JVM, irrespective of the underlying hardware and operating system.

Secured:

Java incorporates various security measures to create a safe computing environment. Features like the Java Security Manager and the Bytecode Verifier help prevent unauthorized access and malicious activities.

Robust:

Java enforces strong type checking, exception handling, and memory management, reducing the risk of system crashes and errors. This robustness enhances the reliability of Java applications.

Architecture Neutral:

Java's bytecode is platform-independent, making it architecture-neutral. The same compiled code can run on different processor architectures.

Interpreted:

Java source code is compiled into bytecode, which is then interpreted by the JVM at runtime. This interpretation allows for adaptability across various systems without recompilation.

High Performance:

While Java is an interpreted language, it also employs Just-In-Time (JIT) compilation techniques, which dynamically translate bytecode into native machine code for improved performance.

Multithreaded:

Java supports multithreading, enabling concurrent execution of multiple threads within a single program. This feature is beneficial for creating efficient, responsive applications.

Distributed:

Java supports distributed computing through its networking capabilities, allowing developers to build applications that can communicate over networks.

Dynamic:

Java's dynamic features include reflection, which enables examining and modifying program structures during runtime. This feature facilitates advanced meta-programming and code manipulation.

Program first java code

let's write a simple "Hello, World!" program in Java. This is a classic introductory program in programming that prints the phrase "Hello, World!" to the console

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

To run this program, follow these steps:

Install Java Development Kit (JDK) on your computer if you haven't already. You can download the latest JDK from the official Oracle website or adopt an open-source distribution like OpenJDK.

Open a text editor (such as Notepad on Windows or TextEdit on macOS) or a code editor (e.g., Visual Studio Code, IntelliJ IDEA, Eclipse) and paste the above Java code into a new file.

Save the file with a .java extension, for example, "HelloWorld.java." The filename should match the class name followed by the .java extension.

Open a terminal or command prompt and navigate to the directory where you saved the "HelloWorld.java" file.

Compile the Java code by running the following command in the terminal:

```
javac HelloWorld.java
```

This will generate a bytecode file named "HelloWorld.class" in the same directory.

Finally, run the compiled Java program using the following command:

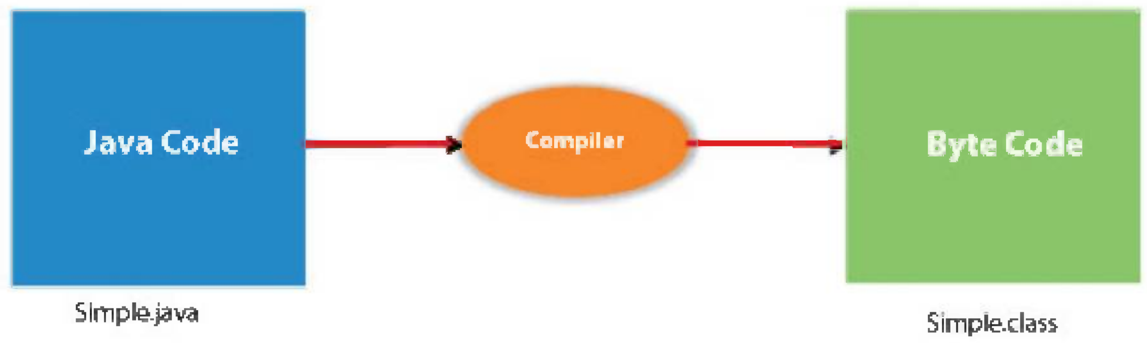
```
java HelloWorld
```

You should see the output "Hello, World!" displayed in the console.

Congratulations! You have successfully written and executed your first Java program. The "Hello, World!" program is the simplest way to ensure that your Java environment is set up correctly and ready for more complex programming tasks.

Compilation Flow:

When we compile a Java program using the "javac" tool, the Java compiler transforms the source code into bytecode.



Parameters Explained in First Java Program: Let's delve into the significance of each keyword and expression used in the initial Java program.

"class" Keyword: The "class" keyword is utilized to declare a class in Java, serving as a blueprint for creating objects.

"public" Keyword: "public" is an access modifier that denotes visibility, making elements accessible to all parts of the program.

"static" Keyword: "static" is a keyword applied to declare a static method. Static methods can be invoked directly without creating objects. The "main()" method is executed by the JVM and does not require an object, conserving memory.

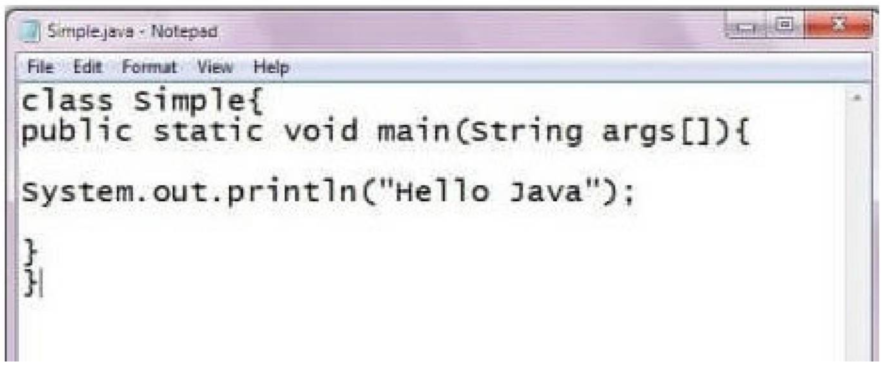
"void" Keyword: "void" is the return type of a method, indicating that the method does not return any value.

"main" Method: "main" serves as the entry point of the program, where the execution commences.

"String[] args" or "String args[]": The "String[] args" or "String args[]" is used for command-line arguments, allowing the program to accept input from the command line.

"System.out.println()": "System.out.println()" is employed to print a statement to the console. In this expression, "System" refers to a class, "out" is an object of the PrintStream class, and "println()" is a method of the PrintStream class. It enables printing output to the standard output (console).

To write a simple program, open Notepad through the Start menu -> All Programs -> Accessories -> Notepad, and input the Java code as illustrated above.

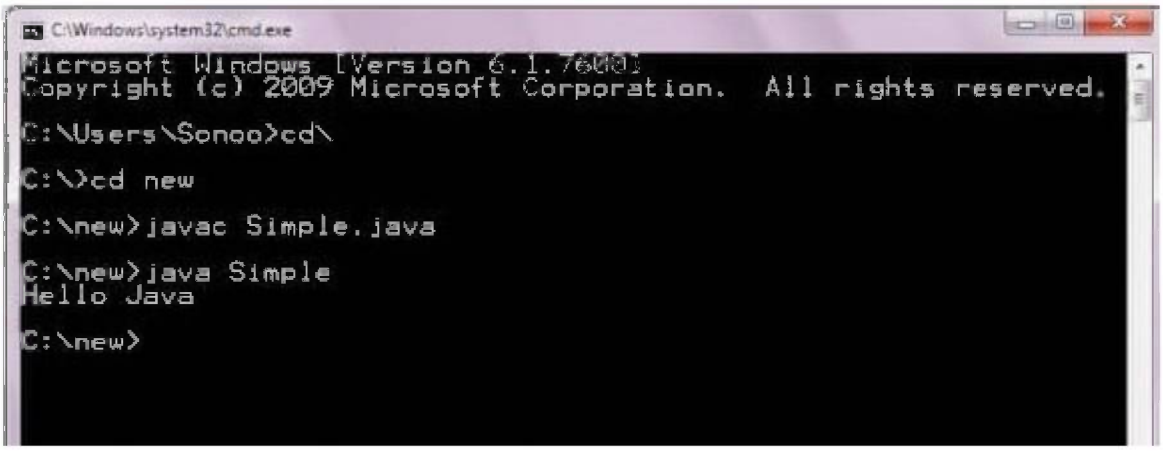


```
class Simple{
public static void main(String args[]){

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

}
}
```

After saving the program as "Simple.java," compile and run it by opening the command prompt through the Start menu -> All Programs -> Accessories -> Command Prompt. Ensure that you are in the correct directory where the "Simple.java" file is saved.



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Sonoo>cd\

C:\>cd new

C:\new>javac Simple.java

C:\new>java Simple
Hello Java

C:\new>
```

To compile the program, use the command:

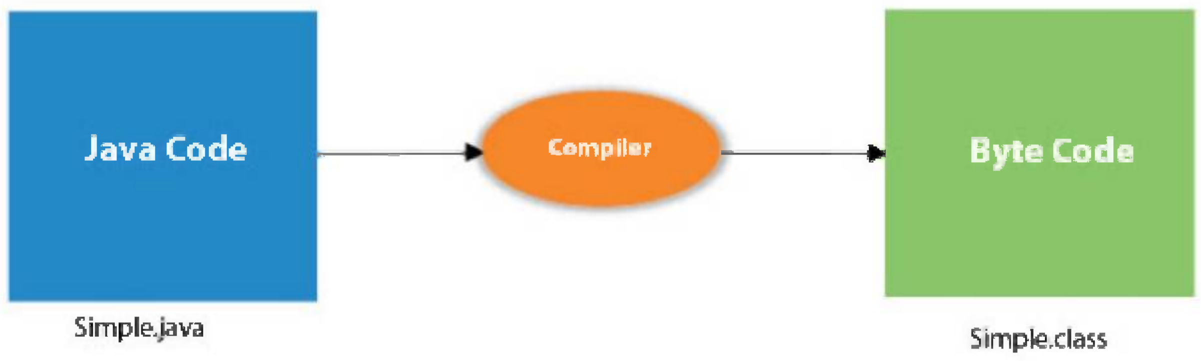
javac Simple.java

To execute the program, use the command:

java Simple

Upon successful execution, the output will be displayed in the command prompt.

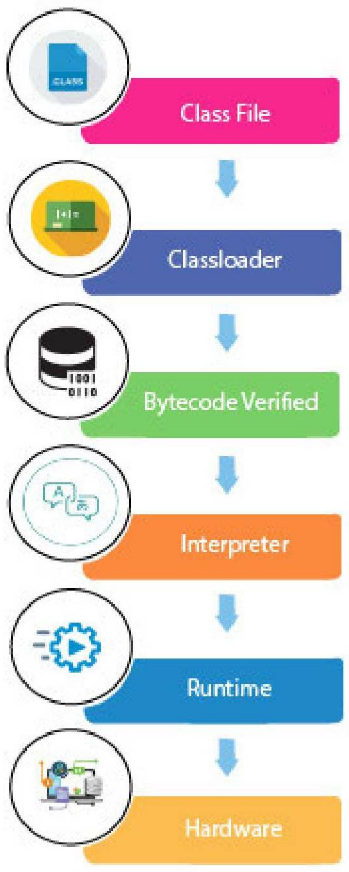
Compile time



Java code file doesn’t interact with operating system. It directly converts from java code to bytecode through compiler.

Runtime

Below steps are being performed during runtime.



Establish Path in java

To set the permanent path of JDK in Windows, follow these steps:

Right-click on "My Computer" (or "This PC") and select "Properties."

In the System Properties window, click on the "Advanced" tab.

Click on the "Environment Variables" button at the bottom of the window.

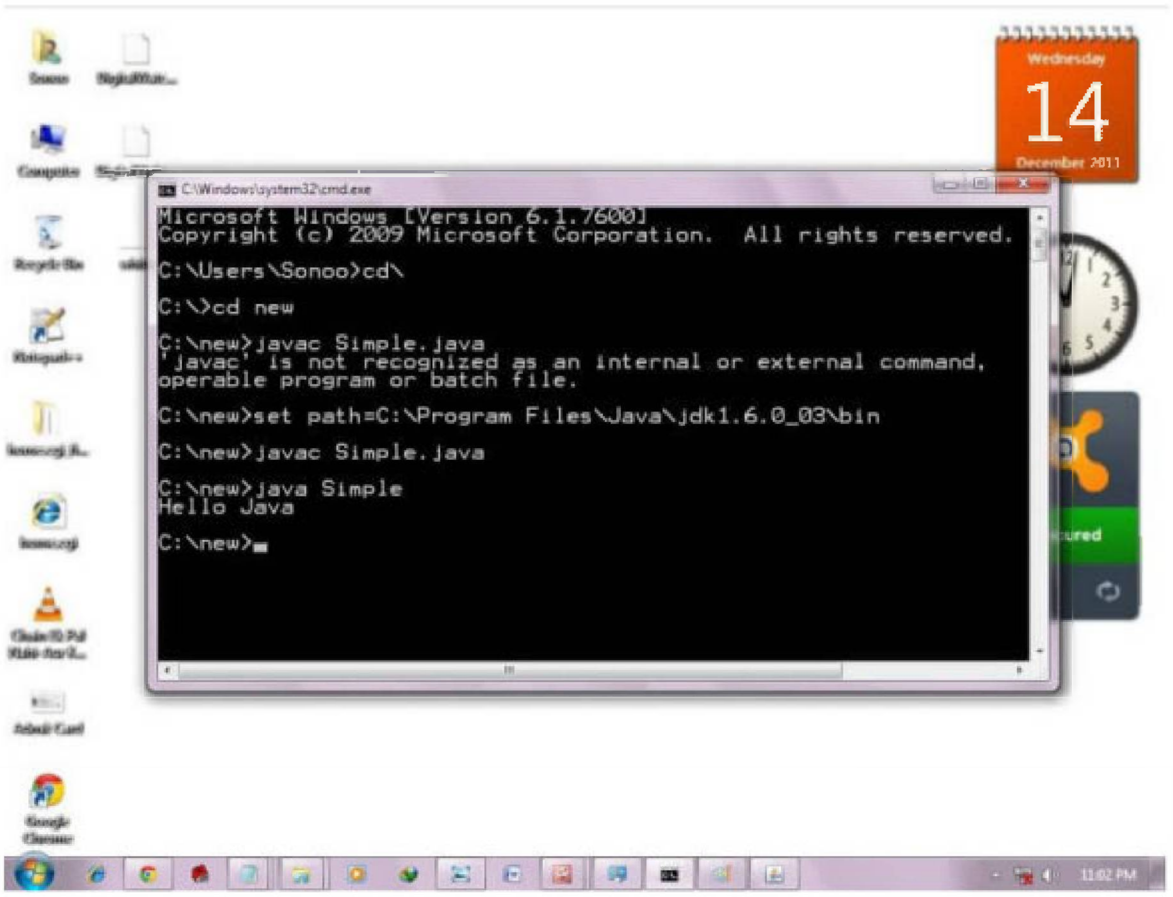
In the Environment Variables window, you will see two sections: User variables and System variables. You can set the path either for the current user (User variables) or for all users (System variables).

For setting the path for the current user (User variables):

In the User variables section, click on the "New" button.

In the "Variable Name" field, enter "PATH" (without quotes).

In the "Variable Value" field, enter the full path to the "bin" directory of your JDK installation. For example: "C:Program FilesJavajdk1.8.0_221bin" (without quotes).



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Sonoo>cd\

C:\>cd new

C:\new>javac Simple.java
'javac' is not recognized as an internal or external command,
operable program or batch file.

C:\new>set path=C:\Program Files\Java\jdk1.6.0_03\bin

C:\new>javac Simple.java

C:\new>java Simple
Hello Java

C:\new>
```

Click "OK" to save the new variable.

For setting the path for all users (System variables):

In the System variables section, scroll down to find the "Path" variable and select it.

Click on the "Edit" button.

In the "Variable Value" field, go to the end of the line and append a semicolon (;) if it's not already there (to separate it from other paths).

Then add the full path to the "bin" directory of your JDK installation. For example: "C:Program FilesJavajdk1.8.0_221bin" (without quotes).

Click "OK" to save the changes.

After setting the path, click "OK" in all the open windows to close them. Now, the JDK path is permanently set, and you can use Java tools like javac and java from any command prompt or terminal window without needing to specify the full path to the JDK/bin directory. The system will automatically find and execute these tools from the specified path.

Setting the Java path in Linux is indeed similar to setting it in Windows, but the syntax and tools used are different. In Linux, you can set the Java path using the "export" command. Here's how to do it:

First, open the terminal on your Linux machine. Locate the directory where you have installed the JDK. For example, if you have installed JDK in "/opt/jdk-11.0.1" directory, you should find the "bin" folder inside it, which contains the Java executables. To set the Java path temporarily for the current terminal session, use the following command:

```
export PATH=$PATH:/opt/jdk-11.0.1/bin/
```

This command adds the "bin" directory of the JDK to the existing PATH environment variable. Now, you can use Java tools (javac, java, etc.) from the terminal without specifying the full path to the JDK/bin directory. If you want to set the Java path permanently for all terminal sessions, you can add the "export" command to your shell configuration file. The specific file depends on the shell you are using (bash, zsh, etc.). For example, for the bash shell, you can add the export command to the ".bashrc" or ".bash_profile" file in your home directory:

```
echo 'export PATH=$PATH:/opt/jdk-11.0.1/bin/' >> ~/.bashrc
```

After adding this line to the file, you need to reload the shell configuration by either opening a new terminal or running the following command:

```
source ~/.bashrc
```

Now, the Java path is set in Linux, and you can use Java tools from any terminal session without specifying the full path to the JDK/bin directory. The system will automatically find and execute these tools from the specified path.

JDK vs JRE vs JVM

JDK, JRE, and JVM are essential components of the Java platform, each serving distinct purposes in the execution and development of Java applications. Let's explore the differences between them:

JDK (Java Development Kit):

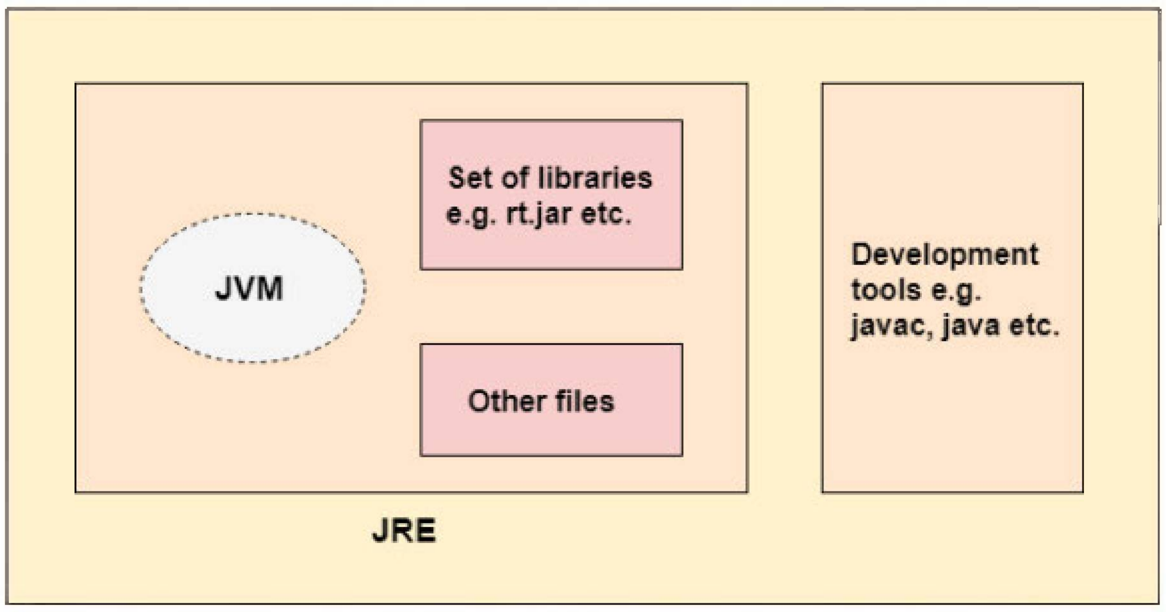
JDK is a comprehensive software development kit provided by Oracle (previously Sun Microsystems) for developing Java applications. It includes everything necessary for Java development, such as compiler, debugger, libraries, and documentation. Key components of JDK are:

Java Compiler (javac): It translates Java source code (.java files) into bytecode (.class files) that can be executed by the JVM.

Java Runtime Environment (JRE): JDK includes a private JRE specifically for development purposes.

Development Tools: JDK provides a set of tools for development, debugging, and testing Java applications.

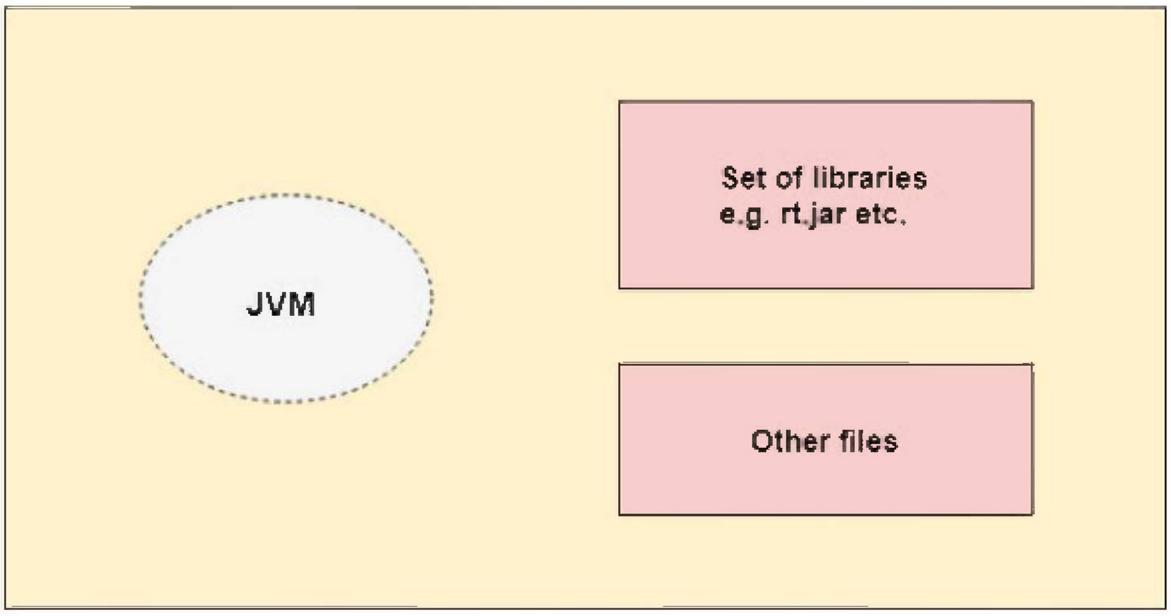
Java API Libraries: It includes the standard Java class libraries and APIs for various functionalities.


Set of libraries e.g. rt.jar etc.
JVM
Other files
Development tools e.g. javac, java etc.
JRE

JDK

JRE (Java Runtime Environment):

JRE is a subset of the JDK and is required to run Java applications. It contains the necessary runtime libraries, the Java Virtual Machine (JVM), and other components to execute Java bytecode. JRE does not include development tools such as the compiler or debugger. When you run a Java application, it relies on the JRE to execute the bytecode on the specific operating system. Users who only need to run Java applications on their systems usually install JRE.



JRE

JVM (Java Virtual Machine):

JVM is the runtime engine that executes Java bytecode. It is responsible for translating bytecode into machine code that can be executed by the underlying hardware. The JVM provides platform independence to Java, allowing Java programs to run on any device or operating system with a compatible JVM implementation. Different platforms (Windows, Linux, macOS, etc.) have their own JVM implementations that adhere to the Java Virtual Machine Specification. In summary, JDK is used for Java development and includes the compiler, runtime libraries, and development tools. JRE is required to run Java applications and includes the JVM and runtime libraries but lacks development tools. JVM is the runtime engine that executes Java bytecode, providing platform independence for Java applications.

Java Virtual Machine (JVM) Architecture

JVM (Java Virtual Machine) embodies an abstract entity, acting as a specification that furnishes a runtime environment for the execution of Java bytecode. JVMs are accessible on multiple hardware and software platforms, rendering them platform-dependent.

What is JVM?

JVM encompasses the following aspects:

Specification: It prescribes the functioning of the Java Virtual Machine while granting independence to the implementation providers for algorithm selection. Oracle and other companies offer diverse JVM implementations.

Implementation: Its realization is known as JRE (Java Runtime Environment), which is a concrete implementation of the JVM.

Runtime Instance: Upon executing the "java" command in the command prompt to run a Java class, an instance of the JVM is created.

JVM Operations:

The JVM undertakes the following operations:

Loading code

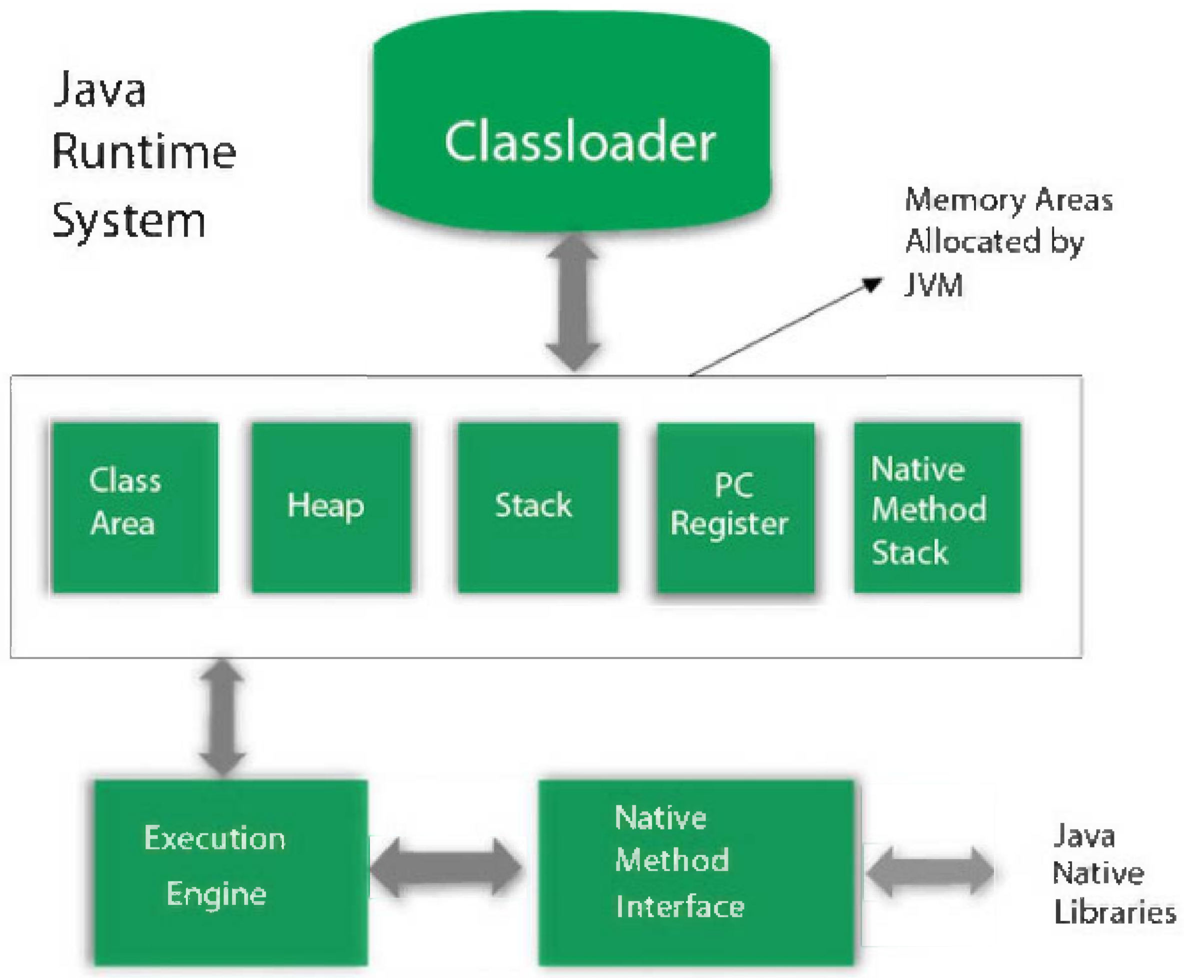
Verifying code

Executing code

Providing a runtime environment

JVM Architecture:

The internal structure of the JVM comprises several components, including:



Classloader: A subsystem of the JVM responsible for loading class files during program execution. It comprises three built-in classloaders:

Bootstrap ClassLoader: The first classloader, the parent of Extension classloader, loads core Java Standard Edition classes (e.g., java.lang, java.net, java.util) from the rt.jar file.

Extension ClassLoader: The child of Bootstrap classloader, loads jar files from $JAVA_HOME/jre/lib/ext directory.

System/Application ClassLoader: The child of Extension classloader, loads class files from the classpath, where the default classpath is set to the current directory.

To illustrate, let's print the classloader name in an example:

```
public class ClassLoaderExample {
public static void main(String[] args) {
Class c = ClassLoaderExample.class;
System.out.println(c.getClassLoader()); // Prints the classloader name
of the current class (Application/System classloader).
System.out.println(String.class.getClassLoader()); // Prints null as
String class is loaded by Bootstrap classloader.
}
}
```

In summary, JVM serves as an abstract specification and runtime environment for Java bytecode execution, and its concrete implementation is realized through JRE. It is a critical component in enabling Java's platform independence and making Java applications portable across various hardware and software platforms.

```
// Example to print the classloader name
public class ClassLoaderExample {
public static void main(String[] args) {
// Print the classloader name of the current class (Application/System
classloader will load this class)
Class currentClass = ClassLoaderExample.class;
System.out.println(currentClass.getClassLoader());

// If we print the classloader name of the String class, it will print null
because String is an in-built class
// found in rt.jar and is loaded by the Bootstrap classloader
System.out.println(String.class.getClassLoader());
}
}
```

Output:

```
sun.misc.Launcher$AppClassLoader@4e0e2f2a
null
```

The Java Virtual Machine (JVM) is a crucial part of the Java platform, responsible for executing Java bytecode. It comprises various components, which we can explore:

Class(Method) Area: The Class(Method) Area stores per-class structures like the runtime constant pool, method and field data, and code for methods.

Heap: The Heap is the runtime data area in which objects are allocated and deallocated. It plays a vital role in memory management for Java applications.

Stack: The Java Stack stores frames, containing local variables and partial results. It is involved in method invocation and return. Each thread has its private JVM stack, created alongside the thread. A new frame is created for each method invocation and destroyed upon completion.

Program Counter Register: The Program Counter (PC) register holds the address of the currently executing Java virtual machine instruction.

Native Method Stack: The Native Method Stack contains native methods used in the application, implemented in other languages like C, C++, or Assembly. JNI (Java Native Interface) facilitates communication with such external code.

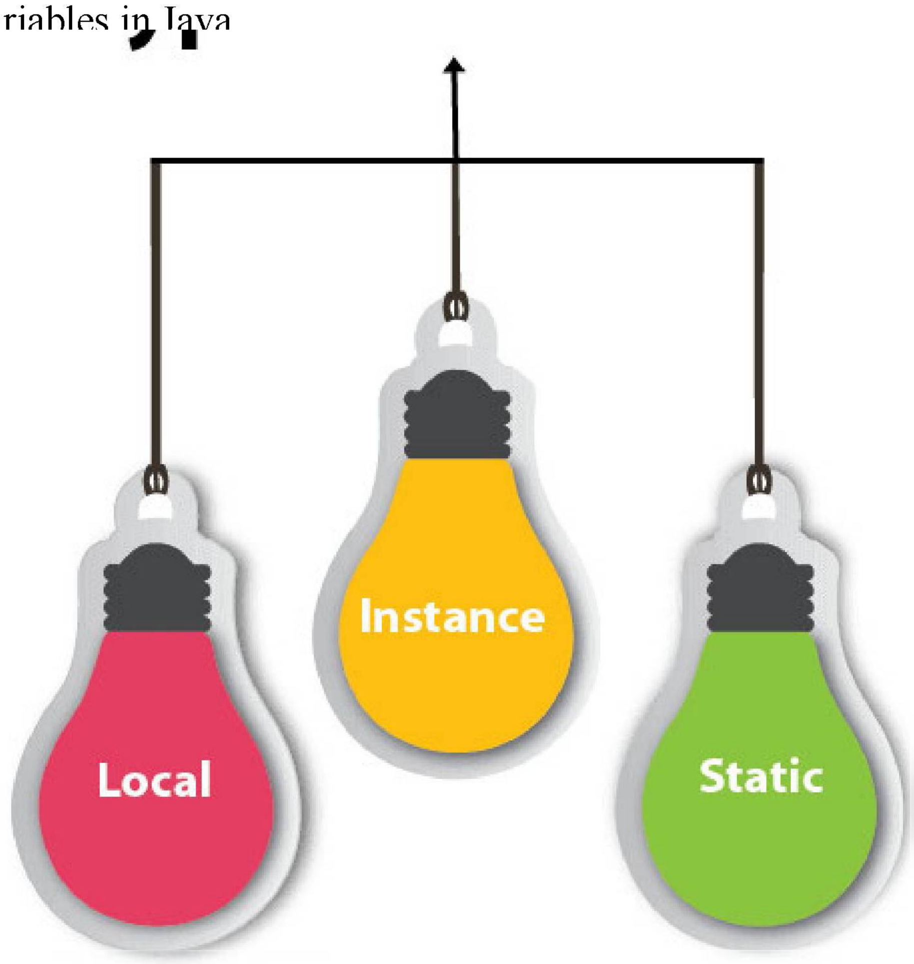
Execution Engine: The Execution Engine includes a virtual processor and two key components:

Interpreter: It reads the bytecode stream and executes the instructions.

Just-In-Time (JIT) Compiler: This component optimizes performance by compiling bytecode parts with similar functionality on-the-fly, reducing compilation time.

Java Native Interface (JNI): JNI is a framework allowing Java applications to interact with code written in other languages, like C, C++, or Assembly. It provides an interface for communication and integration between Java and external applications or libraries. In summary, the JVM acts as an abstract machine, executing Java bytecode and providing a runtime environment. It consists of multiple components working together to ensure efficient execution and platform independence for Java applications.

Variables in Java



Variables in Java serve as containers that hold values during program execution and are assigned a specific data type. There are three types of variables in Java: local, instance, and static.

Local Variable: A local variable is declared inside the body of a method and can only be used within that method. Other methods in the class are not aware of its existence. Local variables cannot be defined with the "static" keyword.

Example:

```
void exampleMethod() {
int localVar = 10; // Local variable
}
```

Instance Variable: An instance variable is declared inside the class but outside the body of a method. It is not declared as static and has instance-specific values, unique to each object of the class.

Example:

```
public class ExampleClass {
int instanceVar; // Instance variable
public static void main(String[] args) {
ExampleClass obj1 = new ExampleClass();
obj1.instanceVar = 10;
ExampleClass obj2 = new ExampleClass();
obj2.instanceVar = 20;
}
}
```

Static Variable: A static variable is declared with the "static" keyword and is shared among all instances of the class. There is only one copy of the static variable, and its memory allocation happens when the class is loaded into memory.

Example:

```
public class ExampleClass {
static int staticVar; // Static variable
public static void main(String[] args) {
ExampleClass obj1 = new ExampleClass();
obj1.staticVar = 10;
ExampleClass obj2 = new ExampleClass();
obj2.staticVar = 20;
System.out.println(obj1.staticVar); // Output: 20 (shared value)
System.out.println(obj2.staticVar); // Output: 20 (shared value)


}
}
```

In addition to the explanations, the examples demonstrate different scenarios, such as arithmetic operations and typecasting, to further illustrate the concepts of variables in Java.

Java data types

Data types in Java specify the nature of data that can be stored in a variable. Java has two categories of data types:

Primitive Data Types:

These are the fundamental data types in Java, representing basic building blocks of data manipulation. There are 8 primitive data types in Java:

boolean: Represents a boolean value, which can be either true or false.

byte: Represents an 8-bit signed integer, with values ranging from -128 to 127.

char: Represents a single 16-bit Unicode character, denoted within single quotes ('a', 'b', '1', etc.).

short: Represents a 16-bit signed integer, with values ranging from -32,768 to 32,767.

int: Represents a 32-bit signed integer, with values ranging from -2^31 to 2^31 - 1.

long: Represents a 64-bit signed integer, with values ranging from -2^63 to 2^63 - 1.

float: Represents a 32-bit floating-point number, used for storing decimal numbers with moderate precision.

double: Represents a 64-bit floating-point number, used for storing decimal numbers with higher precision.

Examples:

```
boolean isStudent = true;
byte age = 25;
char grade = 'A';
short distance = 10000;
int count = 1000000;
```

```
long population = 7700000000L; // Note the 'L' suffix to denote long literal.
float weight = 68.5f; // Note the 'f' suffix to denote float literal.
double pi = 3.14159265359;
```

Non-Primitive Data Types:

These data types are also known as reference data types and include Classes, Interfaces, and Arrays. Unlike primitive data types, non-primitive data types do not directly store values but instead hold references to objects in memory.

Examples:

```
// Class declaration
class Person {
String name;
int age;
}
// Creating an object of the Person class
Person person1 = new Person();
person1.name = "John";
person1.age = 30;
// Array declaration and initialization
int[] numbers = {1, 2, 3, 4, 5};
```

In summary, Java's data types play a crucial role in defining the nature of data that can be stored in variables. Primitive data types handle basic values directly, while non-primitive data types deal with references to objects and more complex data structures.

What is Unicode System?

Java uses the Unicode system for character encoding to ensure that it can handle a wide range of written languages and characters from different scripts across the world. Before Unicode, there were multiple language standards, each with its own character encoding scheme, leading to inconsistencies and compatibility issues.

By adopting the Unicode standard, Java addresses the following problems:

Universal Character Representation: Unicode provides a single, unified character set that covers most of the world's languages, symbols, and characters. This ensures that Java can represent and process text from various languages without any ambiguity.

Consistency in Character Representation: In the previous encoding schemes, the same code value might correspond to different characters in different standards. Unicode eliminates this ambiguity, allowing for consistent character representation.

Fixed-Length Encoding: Unicode ensures fixed-length encoding for characters. In many older encodings, characters had variable lengths, leading to complexities in text processing and storage. With Unicode, each character is represented by a fixed number of bytes (usually two bytes for basic characters) known as code units.

Multilingual Support: Java's adoption of Unicode makes it suitable for developing multilingual applications, where texts from different languages and scripts are used together.

Unicode employs 16-bit encoding, allowing for representation of characters using two bytes. This means that Java also uses two bytes to store characters, allowing it to handle a vast range of characters from different languages and scripts.

Java uses the Unicode escape sequence to represent characters in the form of 'uXXXX', where XXXX is the hexadecimal representation of the Unicode code point. For example, 'u0041' represents the uppercase letter 'A'.

Java operators

Operators in Java are symbols used to perform various operations on data. Java supports a wide range of operators that serve different purposes. Here are the main types of operators in Java:

Unary Operator: Unary operators work on a single operand. The unary operators in Java are:

+ (Unary Plus): Represents the positive value of the operand.

- (Unary Minus): Represents the negation of the operand.

++ (Increment): Increments the value of the operand by 1.

-- (Decrement): Decrements the value of the operand by 1.

! (Logical NOT): Inverts the boolean value of the operand.

Arithmetic Operator: Arithmetic operators perform basic arithmetic operations. The arithmetic operators in Java are:

+ (Addition): Adds two operands.

- (Subtraction): Subtracts the right operand from the left operand.

* (Multiplication): Multiplies two operands.

/ (Division): Divides the left operand by the right operand.

% (Modulus): Returns the remainder after division.

Shift Operator: Shift operators perform bit-wise shifting of the binary representation of the operands. The shift operators in Java are:

<< (Left Shift): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.

>> (Right Shift): Shifts the bits of the left operand to the right by the number of positions specified by the right operand. The leftmost bits are filled with the sign bit (for signed data types) or with 0 (for unsigned data types).

>>> (Unsigned Right Shift): Shifts the bits of the left operand to the right by the number of positions specified by the right operand. The leftmost bits are always filled with 0.

Relational Operator: Relational operators are used to compare operands and return a boolean result. The relational operators in Java are:

== (Equal to): Checks if two operands are equal.

!= (Not Equal to): Checks if two operands are not equal.

> (Greater than): Checks if the left operand is greater than the right operand.

< (Less than): Checks if the left operand is less than the right operand.

>= (Greater than or Equal to): Checks if the left operand is greater than or equal to the right operand.

<= (Less than or Equal to): Checks if the left operand is less than or equal to the right operand.

Bitwise Operator: Bitwise operators perform bit-wise operations on the binary representation of the operands. The bitwise operators in Java are:

& (Bitwise AND): Performs bitwise AND operation between the bits of the operands.

| (Bitwise OR): Performs bitwise OR operation between the bits of the operands.

^ (Bitwise XOR): Performs bitwise exclusive OR operation between the bits of the operands.

~ (Bitwise Complement): Inverts all the bits of the operand.

Logical Operator: Logical operators perform logical operations and return a boolean result. The logical operators in Java are:

&& (Logical AND): Returns true if both operands are true.

|| (Logical OR): Returns true if at least one of the operands is true.

! (Logical NOT): Inverts the boolean value of the operand.

Ternary Operator: The ternary operator (? :) is a conditional operator that evaluates a boolean expression and returns one of two values based on the result of the evaluation.

Assignment Operator: The assignment operator (=) is used to assign values to variables.

In summary, Java supports a rich set of operators that facilitate various computations and comparisons, allowing developers to perform complex operations efficiently.

Keywords used in JAVA

Java Keywords or Reserved Words are predefined and have specific meanings in the Java language. They cannot be used as identifiers (e.g., variable names, class names, etc.) in Java programs. Below is a list of Java keywords with unique alternate words and examples:

abstract: Declares an abstract class with abstract and non-abstract methods.

```
abstract class Shape {
abstract void draw();
void display() {
System.out.println("Displaying Shape");
}
}
```

boolean: Declares a variable of boolean data type that holds true or false.

```
boolean isStudent = true;
```

break: Terminates the current loop or switch statement.

```
for (int i = 0; i < 10; i++) {
if (i == 5) {
break;
}
System.out.print(i + " ");
}
// Output: 0 1 2 3 4
```

byte: Declares a variable that can hold an 8-bit data value.

```
byte age = 25;
```

case: Used in switch statements to mark different blocks of code.

```
switch (day) {
case 1:
System.out.println("Monday");
break;
case 2:
System.out.println("Tuesday");
break;
// ...
}
```

catch: Catches and handles exceptions generated by try statements.

```
try {
// code that may throw an exception
} catch (Exception e) {
// handle the exception
}
```

char: Declares a variable that can hold a single 16-bit Unicode character.

```
char grade = 'A';
```

class: Declares a class.

```
class Person {
// class members
}
```

continue: Skips the remaining code in the loop and continues the next iteration.

```
for (int i = 0; i < 10; i++) {
if (i % 2 == 0) {
continue;
}

System.out.print(i + " ");
}
```

```
// Output: 1 3 5 7 9
```

default: Specifies the default block of code in a switch statement.

```
switch (day) {
case 1:
System.out.println("Monday");
break;
case 2:
System.out.println("Tuesday");
break;
default:
System.out.println("Other Day");
}
```

do: The Java do keyword is used to declare a loop that iterates a part of the program several times. It is often used when the loop needs to be executed at least once, regardless of the condition.

```
int count = 0;
do {
System.out.println("Count: " + count);
count++;
} while (count < 5);
```

double: The Java double keyword is used to declare a variable that can hold a 64-bit floating-point number, which is a double-precision number.

else: The Java else keyword is used to indicate an alternative branch in an if statement. The code block under else is executed if the condition of the if statement evaluates to false.

```
int num = 10;
if (num > 0) {
System.out.println("Positive");
} else {
System.out.println("Non-positive");
}
```

enum: The Java enum keyword is used to define a fixed set of constants. Enums can have constructors (which are always private or default) and methods.

```
enum Days {
MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
Days today = Days.MONDAY;
class Animal { // Base class
// class members
}
class Dog extends Animal { // Derived class
// additional members specific to Dog
}
```

final: The Java final keyword is used to indicate that a variable holds a constant value that cannot be changed. It is also used with methods to prevent method overriding and with classes to prevent class inheritance.

```
final int MAX_VALUE = 100;
final void printMessage() {
System.out.println("This is a final method.");
}
```

finally: The Java finally keyword indicates a block of code in a try-catch structure that is always executed, regardless of whether an exception is handled or not.

```
try {
// code that may throw an exception
} catch (Exception e) {
// handle the exception
} finally {
// code that is always executed
}
```

float: The Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.

```
float temperature = 25.5f;
```

for: The Java for keyword is used to start a for loop, which is used to execute a set of instructions or functions repeatedly when a certain condition becomes true. If the number of iterations is known in advance, it is recommended to use a for loop.

```
for (int i = 0; i < 5; i++) {
System.out.println("Iteration: " + i);
}
```

if: The Java if keyword is used to test a condition. It executes the code block under the if statement if the condition evaluates to true.

```
int x = 10;
if (x > 0) {
System.out.println("Positive");
}
```

implements: The Java implements keyword is used to implement an interface in a class. When a class implements an interface, it must provide concrete (non-abstract) implementations for all the methods declared in that interface.

```
interface Vehicle {
void start();
void stop();
}
class Car implements Vehicle {
public void start() {
System.out.println("Car started.");
}
public void stop() {
System.out.println("Car stopped.");
}
}
```

import: The Java import keyword is used to make classes and interfaces available and accessible to the current source code. It allows you to use classes from other packages without using their fully qualified names.

```
import java.util.ArrayList;
import java.util.List;
public class MyClass {
List names = new ArrayList<>();
}
```

instanceof: The Java instanceof keyword is used to test whether an object is an instance of a specific class or implements a particular interface. It is often used in conjunction with conditional statements or type casting.

```
class Animal { }
class Dog extends Animal { }


Animal animal = new Dog();
if (animal instanceof Dog) {
System.out.println("It's a dog!");
}
```

int: The Java int keyword is used to declare a variable that can hold a 32-bit signed integer (whole number).

```
int age = 25;
```

interface: The Java interface keyword is used to declare an interface. An interface in Java can only have abstract methods (methods without a body) and constant variables. It provides a way to achieve abstraction and multiple inheritance.

```
interface Shape {
void draw();
double area();
}
```

```
class Circle implements Shape {
// Implementing the methods of the Shape interface
public void draw() { /* implementation */ }
public double area() { /* implementation */ }
}
```

long: The Java long keyword is used to declare a variable that can hold a 64-bit integer (long integer).

```
long population = 7854321098L;
```

native: The Java native keyword is used to indicate that a method is implemented in native code using JNI (Java Native Interface). It allows Java programs to interact with native applications or libraries written in other programming languages like C or C++.

```
public class NativeExample {


// Native method declaration
public native void nativeMethod();
}
```

new: The Java new keyword is used to create new objects of a class. It dynamically allocates memory for the object and calls its constructor to initialize the object.

```
class Person {
String name;
int age;
}
Person person = new Person();
person.name = "John";
person.age = 30;
```

null: The Java null keyword is used to indicate that a reference variable does not refer to any object. It is commonly used to initialize reference variables or to check if an object exists before accessing its members.

```
String name = null; // The name variable does not refer to any object.
```

package: The Java package keyword is used to declare a Java package, which is a way to organize and group related classes and interfaces together. It helps in avoiding naming conflicts and provides better organization of code.

```
package com.example.myapp;
// The package declaration should be the first line of a Java source file.
```

private: The Java private keyword is an access modifier used to indicate that a method or variable may be accessed only within the class in which it is declared. It restricts access from other classes.

```
class MyClass {
private int count;
private void increment() {
count++;
}
}
```

protected: The Java protected keyword is an access modifier that allows a method or variable to be accessible within the same package and outside the package, but only through inheritance. It cannot be applied to class-level elements.

```
package com.example.myapp;
class MyBaseClass {
protected int number;
}
```

public: The Java public keyword is an access modifier used to indicate that an item (class, method, or variable) is accessible from anywhere in the program. It has the widest scope among all other access modifiers.

```
public class Calculator {
public int add(int a, int b) {
return a + b;
}
}
```

return: The Java return keyword is used to exit from a method and return a value (if the method has a non-void return type). It can also be used to return early from a method before its completion.

```
public int multiply(int a, int b) {

return a * b; // Returns the result of multiplication
}
```

short: The Java short keyword is used to declare a variable that can hold a 16-bit signed integer (short integer).

```
short temperature = 25;
```

static: The Java static keyword is used to indicate that a variable or method belongs to the class itself rather than an instance of the class. Static members are associated with the class and are shared among all instances of the class.

```
class MyClass {
static int count; // Static variable shared among all instances of
MyClass
static void printMessage() {
System.out.println("Hello, World!");
}
}
```

strictfp: The Java strictfp keyword is used to ensure that floating-point calculations are performed according to the IEEE 754 standard. It provides strict floating-point precision, making the calculations more predictable and portable across different platforms.

```
strictfp class Calculator {
// All floating-point calculations in this class are performed strictly
according to IEEE 754 standard.
}
```

super: The Java super keyword is a reference variable that is used to refer to the immediate parent class objects or invoke the immediate parent class method. It is often used in the context of inheritance to access or call superclass members.

```
class Animal {
void sound() {
System.out.println("Animal makes a sound.");
}
}
class Dog extends Animal {
void sound() {
super.sound(); // Calls the sound() method of the superclass (Animal)
System.out.println("Dog barks.");
}
}
```

switch: The Java switch keyword contains a switch statement that allows the program to execute different blocks of code based on the value of a variable or an expression. It tests the equality of the variable against multiple values.

```
int dayOfWeek = 3;
switch (dayOfWeek) {
case 1:
System.out.println("Sunday");
break;
case 2:
System.out.println("Monday");
break;
case 3:
System.out.println("Tuesday");
break;
// Other cases...
default:
```

```
System.out.println("Invalid day");
}
```

synchronized: The Java synchronized keyword is used to specify that a method or block of code is synchronized, meaning that only one thread can execute it at a time. It is used to ensure thread safety in multithreaded environments.

```
class Counter {
private int count;
// Synchronized method
synchronized void increment() {
count++;
}
}
```

this: The Java this keyword is a reference variable that refers to the current object within a method or constructor of that object. It is often used to access instance variables or invoke other constructors within the same class.

```
class Student {
String name;
Student(String name) {
this.name = name; // 'this' refers to the current object being constructed
}
}
```

throw: The Java throw keyword is used to explicitly throw an exception. It is mainly used to throw custom exceptions or propagate existing exceptions to the calling code for handling.

```
void divide(int dividend, int divisor) {
if (divisor == 0) {

throw new ArithmeticException("Divisor cannot be zero");
}
// Perform the division
}
```

throws: The Java throws keyword is used to declare that a method may throw a checked exception. It is used in the method signature to indicate that the method may not handle the exception and will pass it to the calling method for handling.

```
void readFile() throws IOException {
// Code that reads data from a file and may throw IOException
}
```

transient: The Java transient keyword is used in serialization to indicate that a data member of a class should not be serialized when the object is converted to a byte stream. It is used to exclude sensitive or unnecessary data from being serialized.

```
class User implements Serializable {
String username;
transient String password; // Password will not be serialized
}
```

try: The Java try keyword is used to start a block of code that will be tested for exceptions. It is used in conjunction with catch or finally blocks to handle exceptions that may occur during the execution of the try block.

```
try {
// Code that may throw an exception
} catch (Exception e) {
// Exception handling code
}
```

void: The Java void keyword is used to specify that a method does not have a return value. It is used in the method signature to indicate that the method will not return any data after its execution.

```
void printMessage() {
System.out.println("Hello, World!");
}
```

volatile: The Java volatile keyword is used to indicate that a variable may be accessed and modified by multiple threads simultaneously. It ensures that any change made to the volatile variable is immediately visible to all other threads, preventing thread caching of the variable's value.

```
class SharedResource {
volatile int count;
}
```

while: The Java while keyword is used to start a while loop. The while loop iterates a block of code as long as a specified condition is true. If the number of iterations is not fixed or known beforehand, it is recommended to use the while loop.

```
int count = 0;
while (count < 5) {
System.out.println("Iteration: " + count);
count++;
}
```

Publisher:
BookRix GmbH & Co. KG
Implerstraße 24
81371 Munich
Germany

Text: Ummed Singh
Images: Mitesh Srivastav
Cover: Lokesh Pandey
Editing: Ajay Sharma
Proofreading: Ummed Singh
Layout: Mahesh Babu
Translation: Anand Kumar

Publication Date: July 23rd 2023

https://www.bookrix.com/-qwfb55cf9b2bce5

ISBN: 978-3-7554-4749-8