**Introduction to Java**

Java is a class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is intended to let application developers write once, and run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java was first released in 1995 and is widely used for developing applications for desktop, web, and mobile devices. Java is known for its simplicity, robustness, and security features, making it a popular choice for enterprise-level applications.

JAVA was developed by James Gosling at Sun Microsystems Inc in the year 1995 and later acquired by Oracle Corporation. It is a simple programming language. Java makes writing, compiling, and debugging programming easy. It helps to create reusable code and modular programs. Java is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible. A general-purpose programming language made for developers to write once run anywhere that is compiled Java code can run on all platforms that support Java. Java applications are compiled to byte code that can run on any Java Virtual Machine. The syntax of Java is similar to c/c++.

**History**: Java’s history is very interesting. It is a programming language created in 1991. James Gosling, Mike Sheridan, and Patrick Naughton, a team of Sun engineers known as the Green team initiated the Java language in 1991. Sun Microsystems released its first public implementation in 1996 as Java 1.0. It provides no-cost -run-times on popular platforms. Java1.0 compiler was re-written in Java by Arthur Van Hoff to strictly comply with its specifications. With the arrival of Java 2, new versions had multiple configurations built for different types of platforms.

In 1997, Sun Microsystems approached the ISO standards body and later formalized Java, but it soon withdrew from the process. At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System.

Implementation of a Java application program involves a following step. They include:

**1. Creating the program**

**2. Compiling the program**

**3. Running the program**

Remember that, before we begin creating the program, the Java Development Kit (JDK) must be properly installed on our system and also path will be set**.**

**•** **Creating Program**

We can create a program using Text Editor (Notepad) or IDE (NetBeans)

class Test

{

public static void main(String []args)

{

System.out.println(“My First Java Program.”);

}

};

File -> Save -> d:\**Test.java**

**• Compiling the program**

To compile the program, we must run the Java compiler (javac), with the name of the source file on “command prompt” like as follows

If everything is OK, the “javac” compiler creates a file called “Test.class” containing byte code of the program.

**• Running the program**

We need to use the Java Interpreter to run a program.

**Java programming language is named JAVA. Why?**

After the name OAK, the team decided to give it a new name to it and the suggested words were Silk, Jolt, revolutionary, DNA, dynamic, etc. These all names were easy to spell and fun to say, but they all wanted the name to reflect the essence of technology. In accordance with James Gosling, Java the among the top names along with Silk, and since java was a unique name so most of them preferred it.

Java is the name of an island in Indonesia where the first coffee(named java coffee) was produced. And this name was chosen by James Gosling while having coffee near his office. Note that Java is just a name, not an acronym.

**Java Terminology**

Before learning Java, one must be familiar with these common terms of Java.

1**. Java Virtual Machine(JVM**): This is generally referred to as JVM. There are three execution phases of a program. They are written, compile and run the program.

Writing a program is done by a java programmer like you and me.

The compilation is done by the **JAVAC compiler** which is a primary Java compiler included in **the Java development kit (JDK).** It takes the Java program as input and generates **bytecode as output**.

In the Running phase of a program, JVM executes the bytecode generated by the compiler.

Now, we understood that the function of Java Virtual Machine is to execute the bytecode produced by the compiler. Every Operating System has a different JVM but the output they produce after the execution of bytecode is the same across all the operating systems. This is why Java is known as a platform-independent language.

**2. Bytecode in the Development Process:**  As discussed, the Javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. It is saved as .class file by the compiler. To view the bytecode, a disassembler like javap can be used.

**3. Java Development Kit(JDK)**: While we were using the term JDK when we learn about bytecode and JVM. So, as the name suggests, it is a complete Java development kit that includes everything including compiler, Java Runtime Environment (JRE), java debuggers, java docs, etc. For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.

**4. Java Runtime Environment (JRE):** JDK includes JRE. JRE installation on our computers allows the java program to run, however, we cannot compile it. JRE includes a browser, JVM, applet support, and plugins. For running the java program, a computer needs JRE.

**5. Garbage Collector:** In Java, programmers can’t delete the objects. To delete or recollect that memory JVM has a program called Garbage Collector. Garbage Collectors can recollect the objects that are not referenced. So Java makes the life of a programmer easy by handling memory management. However, programmers should be careful about their code whether they are using objects that have been used for a long time. Because Garbage cannot recover the memory of objects being referenced.

**6. ClassPath:** The classpath is the file path where the java runtime and Java compiler look for .class files to load. By default, JDK provides many libraries. If you want to include external libraries they should be added to the classpath.

**Primary/Main Features of Java**

**1. Platform Independent:** Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode 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 bytecode. That is why we call java a platform-independent language.

**2. Object-Oriented Programming Language**: 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**

**3. Simple**: Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, and Explicit memory allocation.

**4. Robust:** Java language is robust which means reliable. 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.

**5. Secure**: 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.

**6. 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. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

**7. Multithreading**: Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU.

**8. Portable**: As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

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

**10. Dynamic flexibility**: Java being completely object-oriented gives us the flexibility to add classes, new methods to existing classes, and even create new classes through sub-classes. Java even supports functions written in other languages such as C, C++ which are referred to as native methods.

**11. Sandbox Execution**: Java programs run in a separate space that allows user to execute their applications without affecting the underlying system with help of a bytecode verifier. Bytecode verifier also provides additional security as its role is to check the code for any violation of access.

**12. Write Once Run Anywhere**: As discussed above java application generates a ‘.class’ file that corresponds to our applications(program) but contains code in binary format. It provides ease t architecture-neutral ease as bytecode is not dependent on any machine architecture. It is the primary reason java is used in the enterprising IT industry globally worldwide.

**13. Power of compilation and interpretation**: Most languages are designed with the purpose of either they are compiled language or they are interpreted language. But java integrates arising enormous power as Java compiler compiles the source code to bytecode and JVM executes this bytecode to machine OS-dependent executable code.

**class :** class keyword is used to declare classes in Java

**public :** It is an access specifier. Public means this function is visible to all.

**static** : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The main() method here is called by JVM, without creating any object for class.

**void :** It is the return type, meaning this function will not return anything.

**main :** main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.

**String[] args** : This is used to signify that the user may opt to enter parameters to the Java Program at command line. We can use both String[] args or String args[]. Java compiler would accept both forms.

**System.out.println** : This is used to print anything on the console like “printf” in C language.

**Example**

// Importing classes from packages

import java.io.\*;

// Main class

public class GFG {

// Main driver method

public static void main(String[] args)

{

// Print statement

System.out.println("Welcome to GeeksforGeeks");

}

}

Output

Welcome to GeeksforGeeks

**// Q: Write a Java program to calculate the sum and product of two given number.**

class Sample {

public static void main(String args[])

{

int a,b,sum=0,product=0;

a=10;

b=20;

sum=a+b;

product=a\*b;

System.out.println("Sum="+sum);

System.out.println("Product="+product);

}

};

**// Q: Write a Java program to accept two numbers from the user and calculate sum and product.**

import java.util.\*;

class Sample {

public static void main(String args[])

{

int a,b,sum=0,product=0;

Scanner scan=new Scanner(System.in);

System.out.println("Enter the First No:");

a=scan.nextInt();

System.out.println("Enter the Second No:");

b=scan.nextInt();

sum=a+b;

product=a\*b;

System.out.println("Sum="+sum);

System.out.println("Product="+product);

}

}

**// Q: Write a Java program to accept two number from the user and calculate the average.**

import java.util.\*;

public class Sample {

public static void main(String args[])

{

double a,b,sum=0,avg=0;

Scanner scan=new Scanner(System.in);

System.out.println("Enter the First No:");

a=scan.nextDouble();

System.out.println("Enter the Second No:")

b=scan.nextDouble();

sum=a+b;

avg=sum/2;

System.out.println("Average="+avg);

}

}

**// Q: Write a java program to convert fahrenheit to celsius and celsius to Fahrenheit using formula c = (5.0/9.0)\*(f-32).**

import java.util.Scanner;

public class Fahrenheit\_Celsius

{

public static void main(String[] args)

{

double c, f;

Scanner s = new Scanner(System.in);

System.out.print("Enter temperature in Fahrenheit:");

f = s.nextDouble();

c = (5.0/9.0)\*(f-32);

System.out.println("Temperature in Celsius:"+c);

}

}

**// Q: Write a Java program to calculate the area of triangle using three sides.**

import java.lang.\*;

import java.util.Scanner;

public class Area

{

public static void main(String[] args)

{

int a,b,c;

double s,area;

Scanner scan = new Scanner(System.in);

System.out.print("Enter the three sides:");

a=scan.nextInt();

b=scan.nextInt();

c=scan.nextInt();

s=(a+b+c)/2;

area=Math.sqrt(s\*(s-a)\*(s-b)\*(s-c));

System.err.println("Area of Triangle:"+area);

}

}

Explanation:

**1. Comments**: Comments are used for explaining code and are used in a similar manner in Java or C or C++. Compilers ignore the comment entries and do not execute them. Comments can be of a single line or multiple lines.

Single line Comments:

Syntax:

// Single line comment

Multi-line comments:

Syntax:

/\* Multi line comments\*/

**2. import java.io.\*:** This means all the classes of the io package can be imported. Java io package provides a set of input and output streams for reading and writing data to files or other input or output sources.

**3. class**: The class contains the data and methods to be used in the program. Methods define the behavior of the class. Class GFG has only one method Main in JAVA.

**4. static void main():** static keyword tells us that this method is accessible without instantiating the class.

**5. void:** keywords tell that this method will not return anything. The main() method is the entry point of our application.

**6. System.in:** This is the standard input stream that is used to read characters from the keyboard or any other standard input device.

**7. System.out**: This is the standard output stream that is used to produce the result of a program on an output device like the computer screen.

**8. println():** This method in Java is also used to display text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.

**9. String []args:** This is the argument passed to the main function which is an array of strings with the array name args. One can choose their own flexible name but this name is used by many developers.

Everything in java is represented in Class as an object including the main function.

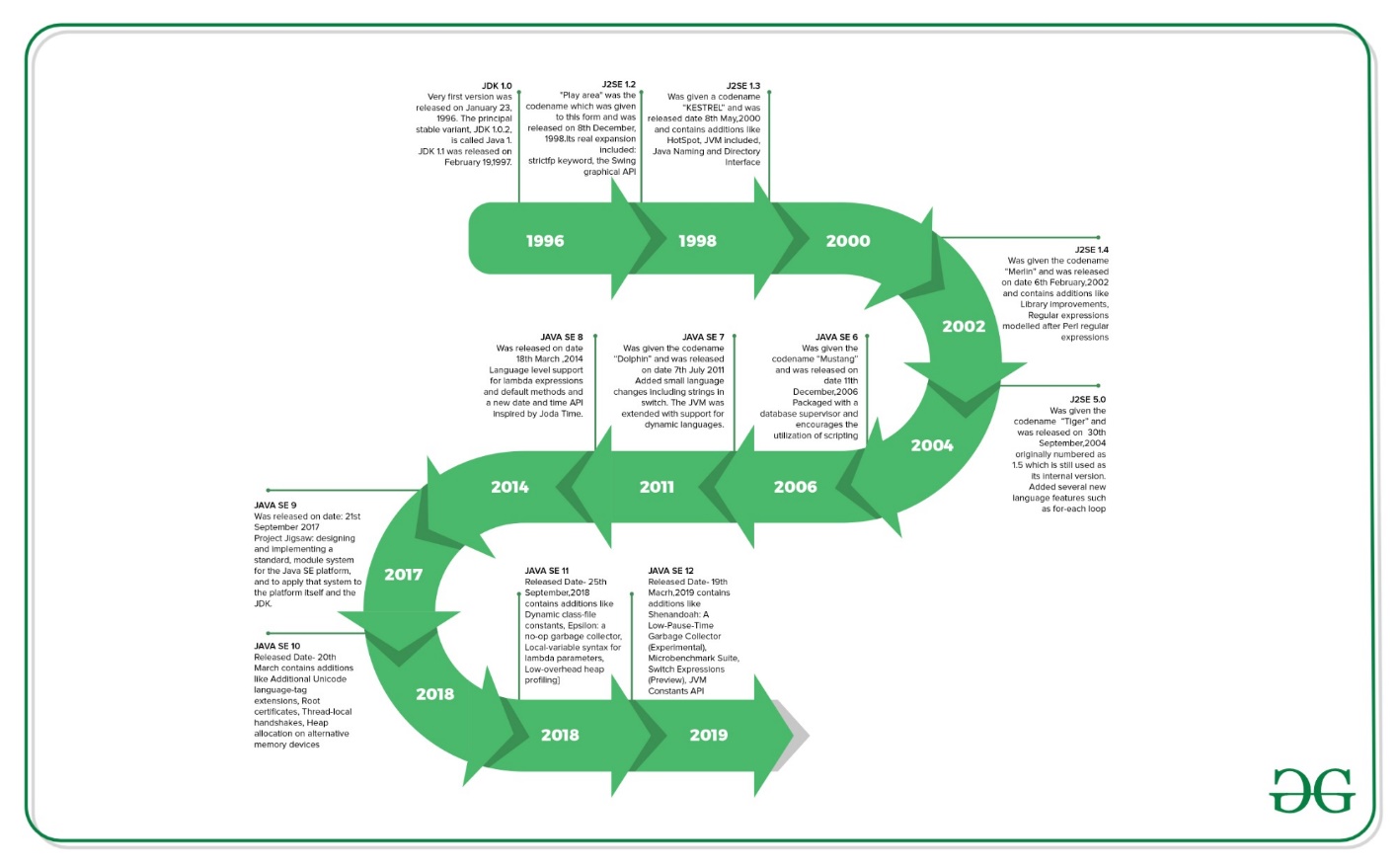
**The Complete History of Java Programming Language**

Java is an Object-Oriented programming language developed by James Gosling in the early 1990s. The team initiated this project to develop a language for digital devices such as set-top boxes, television, etc. Originally C++ was considered to be used in the project but the idea was rejected for several reasons(For instance C++ required more memory). Gosling endeavoured to alter and expand C++ however before long surrendered that for making another stage called Green. James Gosling and his team called their project “Greentalk” and its file extension was .gt and later became to known as “OAK”.

**Why “Oak”?**

The name Oak was used by Gosling after an oak tree that remained outside his office. Also, Oak is an image of solidarity and picked as a national tree of numerous nations like the U.S.A., France, Germany, Romania, etc. But they had to later rename it as “JAVA” as it was already a trademark by Oak Technologies. “JAVA” Gosling and his team did a brainstorm session and after the session, they came up with several names such as JAVA, DNA, SILK, RUBY, etc. Java name was decided after much discussion since it was so unique. The name Java originates from a sort of espresso bean, Java. Gosling came up with this name while having a coffee near his office. Java was created on the principles like Robust, Portable, Platform Independent, High Performance, Multithread, etc. and was called one of the Ten Best Products of 1995 by the TIME MAGAZINE. Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.

The Java language has experienced a few changes since JDK 1.0 just as various augmentations of classes and packages to the standard library. In Addition to the language changes, considerably more sensational changes have been made to the Java Class Library throughout the years, which has developed from a couple of hundred classes in JDK 1.0 to more than three thousand in J2SE 5.



| Version | Release Date | Major changes |
| --- | --- | --- |
| JDK Beta | 1995 |  |
| JDK 1.0 | January 1996 | The Very first version was released on January 23, 1996. The principal stable variant, JDK 1.0.2, is called Java 1. |
| JDK 1.1 | February 1997 | Was released on February 19, 1997. There were many additions in JDK 1.1 as compared to version 1.0 such as   * A broad retooling of the AWT occasion show * Inner classes added to the language * JavaBeans * JDBC * RMI |
| J2SE 1.2 | December 1998 | “Play area” was the codename which was given to this form and was released on 8th December 1998. Its real expansion included: strictfp keyword   * the Swing graphical API was coordinated into the centre classes * Sun’s JVM was outfitted with a JIT compiler out of the blue * Java module * Java IDL, an IDL usage for CORBA interoperability * Collections system |
| J2SE 1.3 | May 2000 | Codename- “KESTREL” Release Date- 8th May 2000 Additions:   * HotSpot JVM included * Java Naming and Directory Interface * JPDA * JavaSound * Synthetic proxy classes |
| J2SE 1.4 | February 2002 | Codename- “Merlin” Release Date- 6th February 2002 Additions: Library improvements   * Regular expressions modelled after Perl regular expressions * The image I/O API for reading and writing images in formats like JPEG and PNG * Integrated XML parser and XSLT processor (JAXP) (specified in JSR 5 and JSR 63) * Preferences API (java.util.prefs)   Public Support and security updates for this version ended in October 2008. |
| J2SE 5.0 | September 2004 | Codename- “Tiger” Release Date- “30th September 2004” Originally numbered as 1.5 which is still used as its internal version. Added several new language features such as:   * for-each loop * Generics * Autoboxing * Var-args |
| JAVA SE 6 | December 2006 | Codename- “Mustang” Released Date- 11th December 2006 Packaged with a database supervisor and encourages the utilization of scripting languages with the JVM. Replaced the name J2SE with java SE and dropped the .0 from the version number. Additions:   * Upgrade of JAXB to version 2.0: Including integration of a StAX parser. * Support for pluggable annotations (JSR 269). * JDBC 4.0 support (JSR 221) |
| JAVA SE 7 | July 2011 | Codename- “Dolphin” Release Date- 7th July 2011 Added small language changes including strings in the switch. The JVM was extended with support for dynamic languages. Additions:   * Compressed 64-bit pointers. * Binary Integer Literals. * Upstream updates to XML and Unicode. |
| JAVA SE 8 | March 2014 | Released Date- 18th March 2014 Language level support for lambda expressions and default methods and a new date and time API inspired by Joda Time. |
| JAVA SE 9 | September 2017 | Release Date: 21st September 2017 Project Jigsaw: designing and implementing a standard, a module system for the Java SE platform, and to apply that system to the platform itself and the JDK. |
| JAVA SE 10 | March 2018 | Released Date- 20th March Addition:   * Additional Unicode language-tag extensions * Root certificates * Thread-local handshakes * Heap allocation on alternative memory devices * Remove the native-header generation tool – javah. * Consolidate the JDK forest into a single repository. |
| JAVA SE 11 | September 2018 | Released Date- 25th September, 2018 Additions-   * Dynamic class-file constants * Epsilon: a no-op garbage collector * The local-variable syntax for lambda parameters * Low-overhead heap profiling * HTTP client (standard) * Transport Layer Security (TLS) 1.3 * Flight recorder |
| *JAVA SE 12* | *March 2019* | *Released Date- 19th March 2019 Additions-*   * *Shenandoah: A Low-Pause-Time Garbage Collector (Experimental)* * *Microbenchmark Suite* * *Switch Expressions (Preview)* * *JVM Constants API* * *One AArch64 Port, Not Two* * *Default CDS Archives* |
| *JAVA SE 13* | *September 2019* | *Released Date – 17th September 2019 Additions-*   * *Text Blocks (Multiline strings).* * *Switch Expressions.* * *Enhanced Thread-local handshakes.* |
| JAVA SE 14 | March 2020 | Released Date – 17th March 2020 Additions-   * Records (new class type for data modeling). * Pattern Matching for instanceof. * Helpful NullPointerExceptions. |
| JAVA SE 15 | September 2020 | Released Date – 15th September 2020 Additions-   * Sealed Classes. * Hidden Classes. * Foreign Function and Memory API (Incubator). |
| JAVA SE 16 | March 2021 | Released Date – 16th March 2021 Additions-   * Records (preview feature). * Pattern Matching for switch (preview feature). * Unix Domain Socket Channel (Incubator). |
| JAVA SE 17 | September 2021 | Released Date – 14th September 2021 Additions-   * Sealed Classes (finalized). * Pattern Matching for instanceof (finalized). * Strong encapsulation of JDK internals by default. * New macOS rendering pipeline. |

C++ vs Java vs **PythonC++ vs Java vs Python**

These three programming languages are the most popular among coders in terms of competitive coding and programming. C++ of today in its efficiency, speed, and memory makes it widely popular among coders. Java is platform-independent. It continues to add considerable value to the world of software development. Python requires less typing and provides new libraries, fast prototyping, and several other new features. Let’s look at the comparison between these popular coding languages.These three programming languages are the most popular among coders in terms of competitive coding and programming. C++ of today in its efficiency, speed, and memory makes it widely popular among code\

C++ Vs Java:

| TOPIC | C++ | Java |
| --- | --- | --- |
| Memory Management | Use of pointers, structures, union | No use of pointers. Supports references, thread and interfaces. |
| Libraries | Comparatively available with low-level functionalities | Wide range of classes for various high-level services |
| Multiple Inheritance | Provide both single and multiple inheritance. | Multiple inheritances is partially done through interfaces |
| Operator Overloading | Supports operator overloading | It doesn’t support this feature |
| Program Handling | Functions and variables can reside outside classes. | Functions and variables reside only in classes, packages are used. |
| Portability | Platform dependent, must be recompiled for different platform | Platform independent, byte code generated works on every OS. |
| Thread Support | No built-in support for threads, depends on libraries. | It has built-in thread support. |

is platform-independent. It continues to add considerable value to the world of software eDatatype | **Python Vs Java:**

Components can be developed in Java and combined to form applications in Python. Let’s see some of the differences between these two popular languages:other

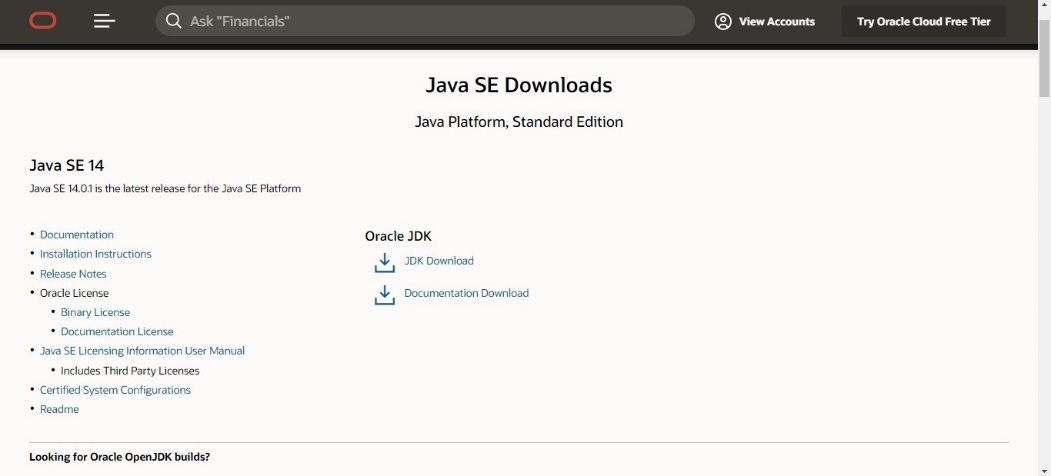
| TOPIC | Java | Python |
| --- | --- | --- |
| Compilation process | Java is both compiled and interpreted language, which is first compiled and then interpreted into a byte code. | Python is an interpreted programming language |
| Code Length | Longer lines of code as compared to python. | 3-5 times shorter than equivalent Java programs. |
| Syntax Complexity | Define particular block by curly braces, end statements by ; | No need of semi colons and curly braces, uses indentation |
| Ease of typing | Strongly typed, need to define the exact datatype of variables | Dynamic, no need to define the exact datatype of variables. |
| Speed of execution | Java is much faster than python in terms of speed. | Expected to run slower than Java programs |
| Multiple Inheritance | Multiple inheritance is partially done through interfaces | Provide both single and multiple inheritance |

**How to Download and Install Java for 64 bit machine?**

Java is one of the most popular and widely used programming languages. It is simple, portable, platform independent language. It has been one of the most popular programming languages for many years. In this article, we will see how to download and install Java on a 64-bit architecture machines.

The following steps can be followed in order to download and install java. All the steps are described below has been performed on the Windows 10 operating system, but the procedure is quite similar to other operating systems as well. Fea

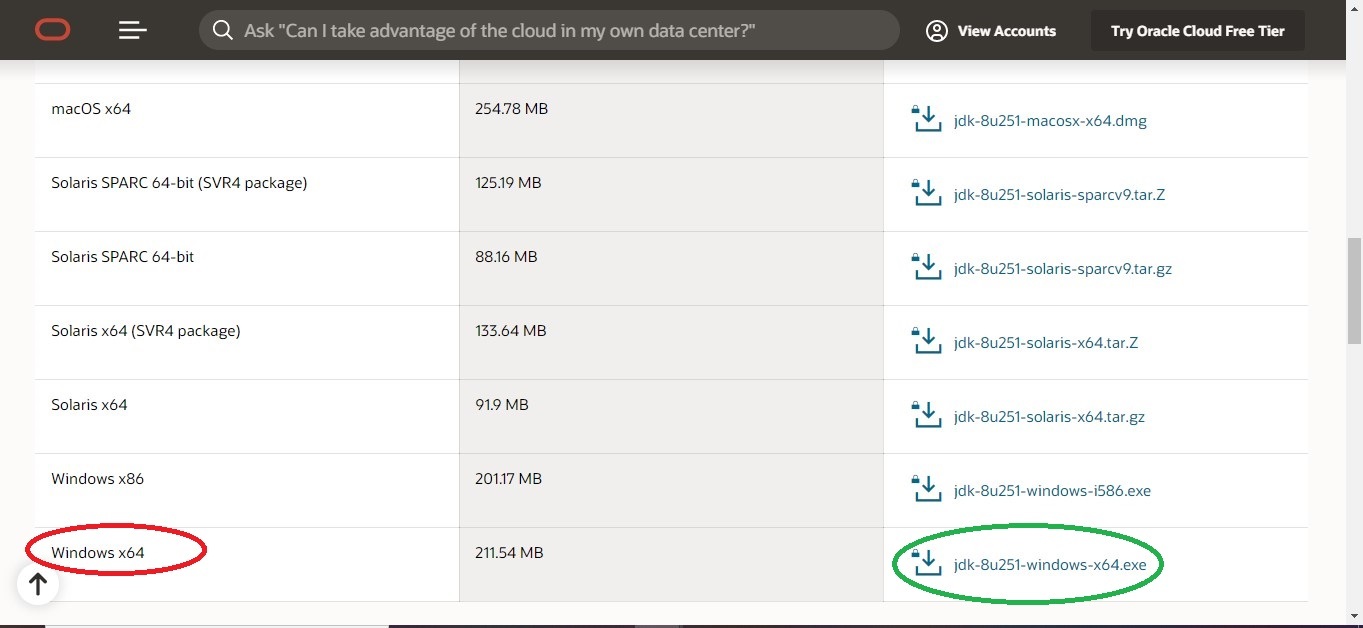
Step 1: Open https://www.oracle.com/java/technologies/javase-downloads.html url in the browser and it will navigate to the official Oracle Java downloads page.

t’s look at the comparison between these popular coding languages.

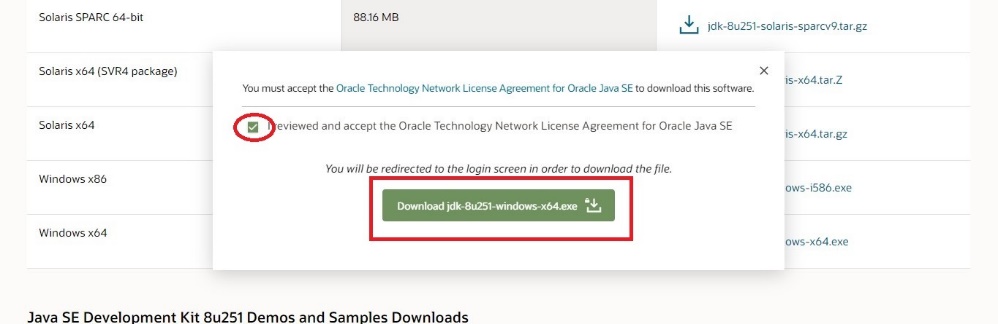
Step 2: Now, scroll to the version of the Java which we want to download and click on JDK Download option as shown below:



Step 3: Scroll down to the page and click on the download button option suitable for your computer Operating system. But for a 64-bit machine, choose the software name ending with x64.



After clicking on the download button, a popup will appear which says that we have to accept Oracle Technology Network License Agreement for Oracle Java SE in order to download this software. Therefore, click on the checkbox and then proceed to download as shown below:



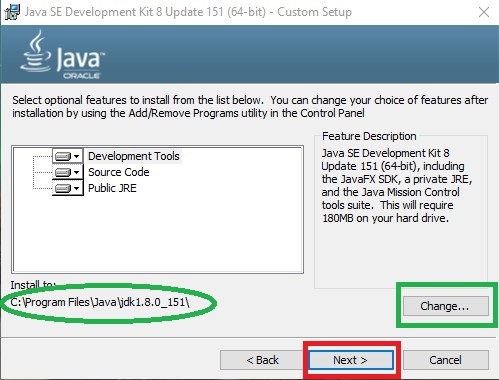
Step 4: We will now be navigated to Oracle Login page. We need to login to the account. As soon we log in, our download will start instantly as shown below:

https://media.geeksforgeeks.org/wp-content/uploads/20200605003606/7EC5F186-04EA-4555-BD06-70722D598E6D.png.jpg

Step 5: After the downloading procedure is complete, we need to run the installer. Once Java installation wizard opens, click on the Next button as shown below:



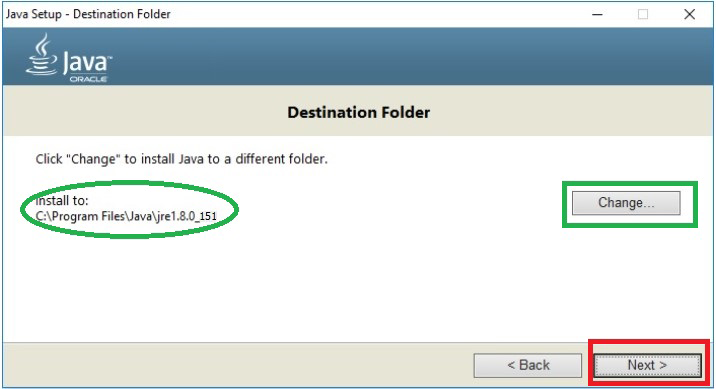
Step 6: Again click on the Next button if we wish to install Java development kit in the default directory(encircled with green color), or we can change this directory by clicking on Change button



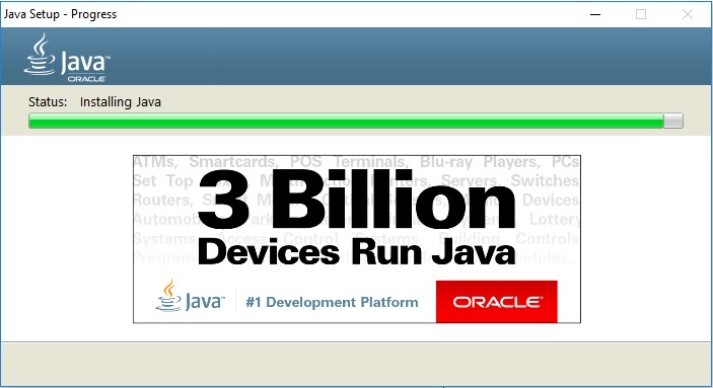
Step 7: The installation will begin as shown below:



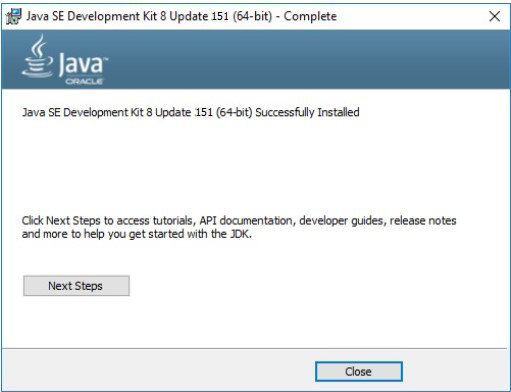
Step 8: Now, it will ask for the installation directory for JRE(Java Runtime Environment). Again we can continue with the default directory or change it accordingly.



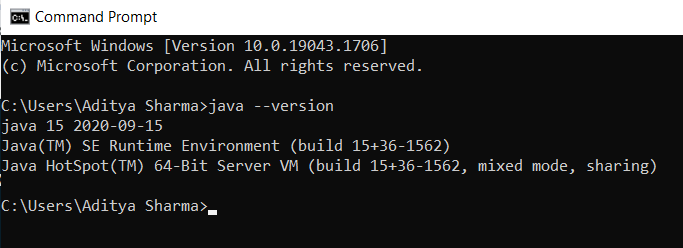
Step 9: The Java installation will be successfully completed as shown below:



Step 10: Finally, we can click on the Close button after the confirmation window appears which saying that the Java is installed.



Step 11: At last, To confirm if everything is set up properly, open cmd and type java –version



Setting up the environment in Java

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, etc. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture. The latest version is Java 21. Below are the environment settings for both Linux and Windows. JVM, JRE, and JDK three are all platform-dependent because the configuration of each Operating System is different. But, Java is platform-independent. Few things must be clear before setting up the environment which can better be perceived from the below image provided as follows:



**JDK(Java Development Kit)**: JDK is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.

**JRE(Java Runtime Environment**): JRE contains the parts of the Java libraries required to run Java programs and is intended for end-users. JRE can be viewed as a subset of JDK.

**JVM: JVM** (Java Virtual Machine) is an abstract machine. It is a specification that provides a runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms.

Now let us discuss the steps for setting up a Java environment with visual aids. Let’s use the Windows operating system to illustrate visual aids.

Steps: Here we will be proposing steps for three different operating systems as listed:

**Windows operating system**

**Linux operating system**

**macOS operating system**

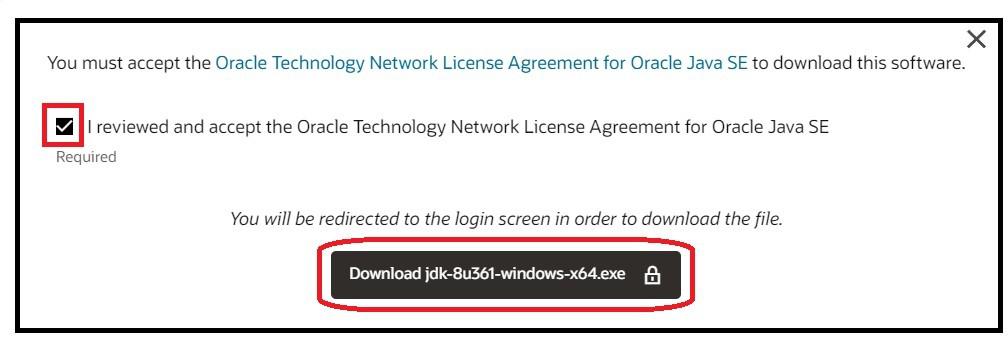
A. **Windows operating systems**

The steps for setting the environment in the Windows operating system are as follows:

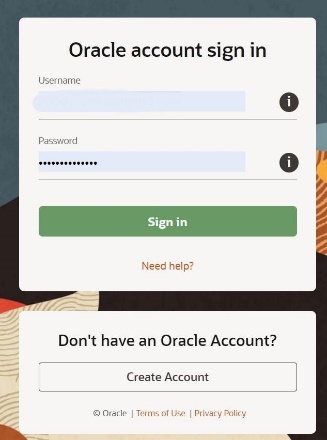
Step 1: Java8 JDK is available at Download Java 8. Click the first link for Windows(32-bit) and the last link for Windows(64-bit) as highlighted below.



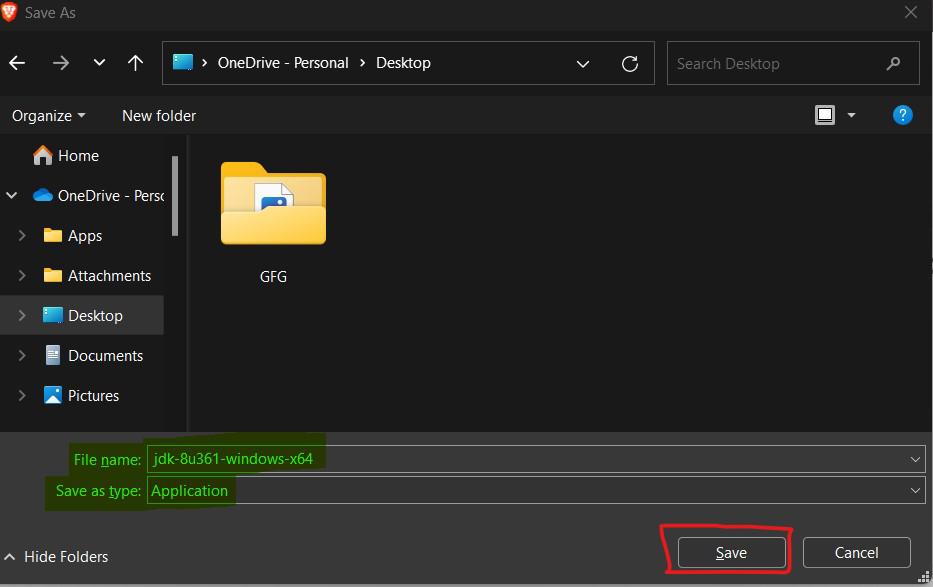
Step 2: Now you will see a download prompt just accept the term and conditions and click on the download button



Step 3: If you have an oracle account then sign in or if don’t then create one and sign in

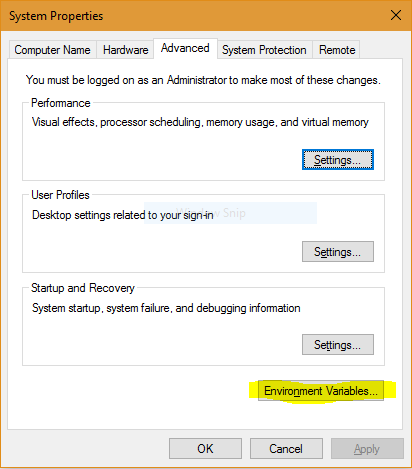


Step 4: Then your download will start automatically after signing in if don’t then click on the previous link again

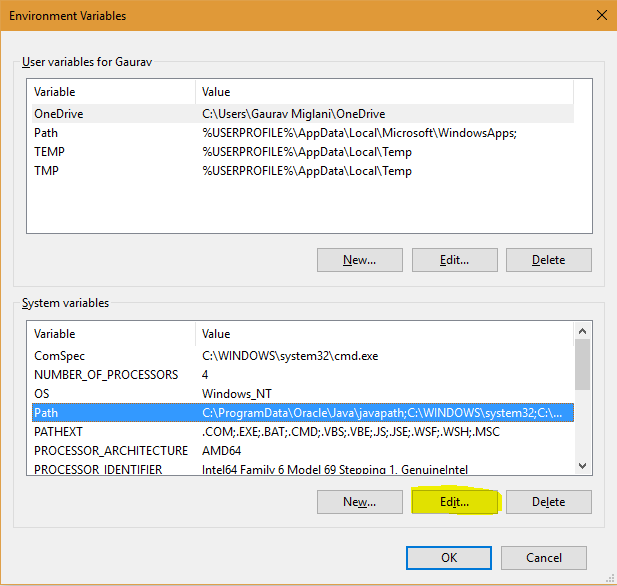


Step 5: After downloading, run the jdk-8u Application(.exe file) and follow the instructions to install Java on your machine. Once you install Java on your device, you have to set up the environment variable.

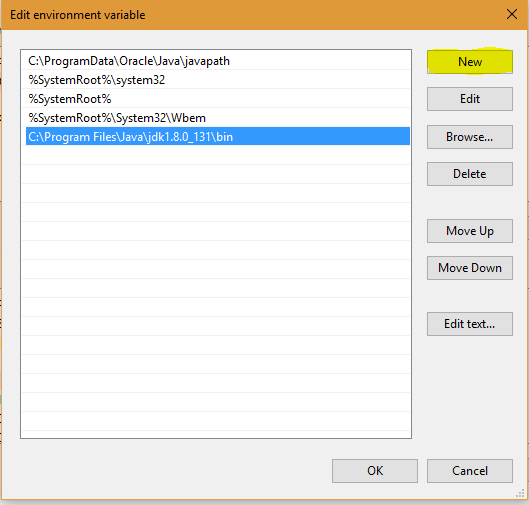
Step 6: Go to Control Panel -> System and Security -> System. Under the Advanced System Setting option click on Environment Variables as highlighted below.



Step 7: Now, you have to alter the “Path” variable under System variables so that it also contains the path to the Java environment. Select the “Path” variable and click on the Edit button as highlighted below.



Step 8: You will see a list of different paths, click on the New button, and then add the path where java is installed. By default, java is installed in “C:\Program Files\Java\jdk\bin” folder OR “C:\Program Files(x86)\Java\jdk\bin”. In case, you have installed java at any other location, then add that path.



Step 9: Click on OK, Save the settings, and you are done !! Now to check whether the installation is done correctly, open the command prompt and type javac -version. You will see that java is running on your machine.

Note: To make sure whether the compiler is set up, type javac in the command prompt. You will see a list related to javac.

* **How to Download and Install Eclipse on Windows?**

Are you starting to code? Then Eclipse will prove helpful for you. The best part about Eclipse software download is that, it’s a free and integrated development environment (IDE) with a user-friendly interface. It provides extensive features, making it ideal for coding projects. It’s famous for supporting multiple programming languages. Eclipse provides a flexible environment, enabling developers to efficiently and collaboratively create software.

In this article, we learn how to use this helpful tool by installing Eclipse on Your Windows PC. Before directly jumping to Eclipse’s installation part. Let’s get more familiar with Eclipse IDE.

**Why Use Eclipse?**

Eclipse is a free open source platform, Integrated Development Environment (IDE) with the help of which applications are made using the Java programming languages and other programming languages are also used such as C/C++, PERL, Python, Ruby, etc.

Eclipse is a preferred choice for developers for several reasons. It provides a user-friendly interface that makes it easy for both beginners and experienced coders. It supports multiple programming languages. It also offers collaborative tools that allow multiple developers to contribute to a project simultaneously.

Eclipse Software comprises of many plug-ins and is designed to be extensible using additional plug-ins. Eclipse IDE can be used for any programming language for which a plug-in is available.

Famous Plugins Used in Eclipse IDE

**Following are some famous plug-ins used in Eclipse IDE :**

The Java Development Tools (JDT) is a plugin that allows Eclipse to be used as a Java IDE.

PyDev is a plugin that allows Eclipse to be used as a Python IDE.

C/C++ Development Tools (CDT) is a plug-in that allows Eclipse to be used as C/C++ development.

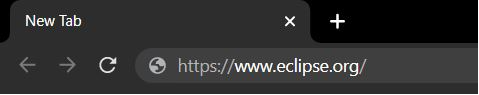
The Scala is a plug-in that allows Eclipse to be used as an IDE to develop Scala applications.

PHPeclipse is a plug-in that allows Eclipse to be used as an IDE to develop PHP applications

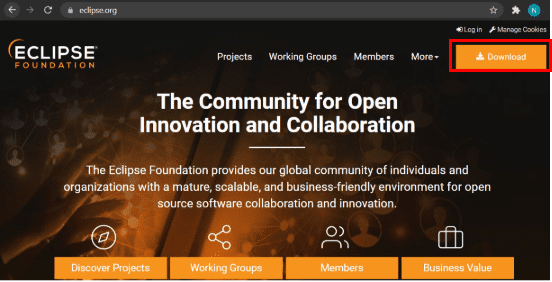
**Steps to Eclipse software download on Windows:**

Now let’s look at step by step process of installing Eclipse IDE on Windows:

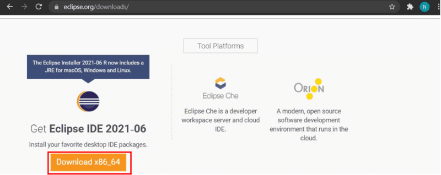
Step 1: In the first step, Open your browser and navigate to this URL.



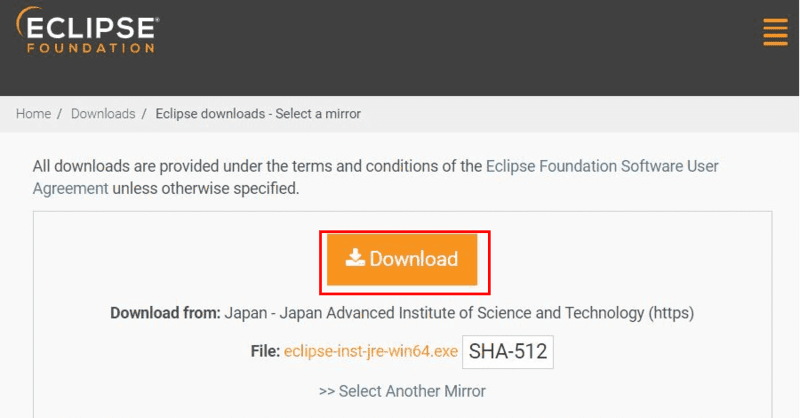
Step 2: Then, click on the “Download” button to download Eclipse IDE.



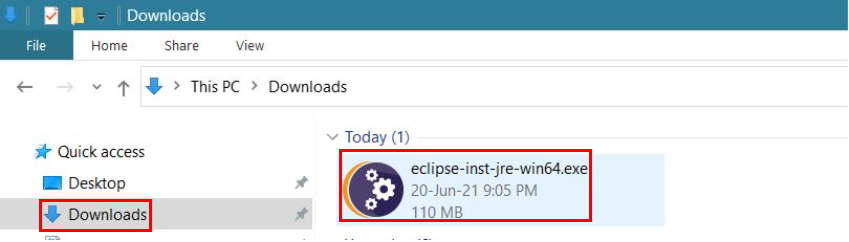
Step 3: Now, click on the “Download x86\_64” button.



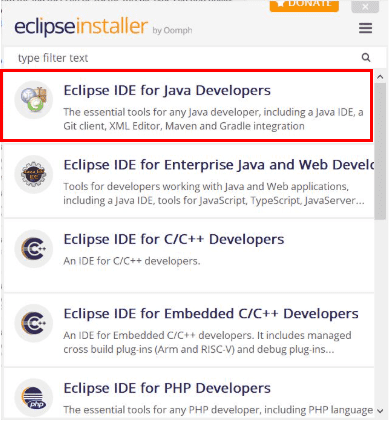
Step 4: Then click on the “Download” button. After clicking on the download button the .exe file for the eclipse will be downloaded.



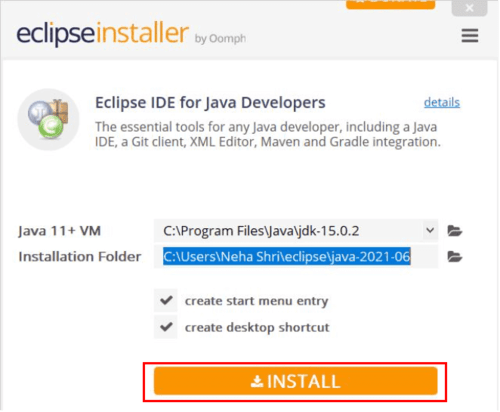
Step 5: Now go to File Explorer and click on “Downloads” after that click on the “eclipse-inst-jre-win64.exe” file for installing Eclipse IDE.



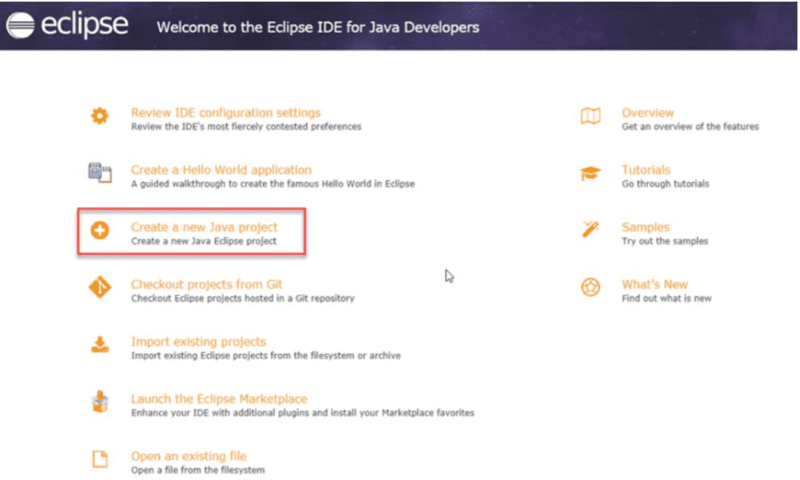
Step 6: Then, click on “Eclipse IDE for Java Developers”.

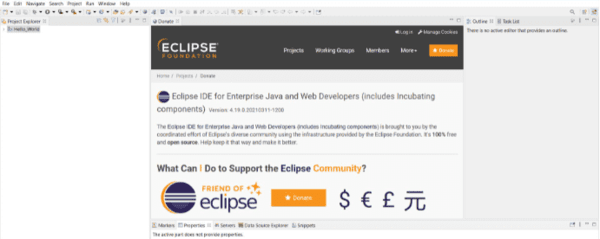


Step 7: Then, click on the “Install” button.



Step 8: Now click on “Create a new Java project”.

  
Now, you are ready to make new Java projects using eclipse IDE and the screen will look like this :

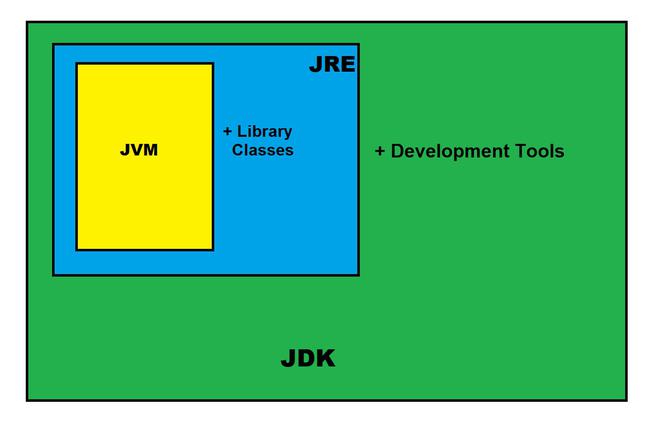


**JDK in Java**

The Java Development Kit (JDK) is a cross-platformed software development environment that offers a collection of tools and libraries necessary for developing Java-based software applications and applets. It is a core package used in Java, along with the JVM (Java Virtual Machine) and the JRE (Java Runtime Environment).

Beginners often get confused with JRE and JDK, if you are only interested in running Java programs on your machine then you can easily do it using Java Runtime Environment. However, if you would like to develop a Java-based software application then along with JRE you may need some additional necessary tools, which is called JDK.

JDK=JRE+Development Tools



The Java Development Kit is an implementation of one of the Java Platform:

Standard Edition (Java SE),

Java Enterprise Edition (Java EE),

Micro Edition (Java ME),

Contents of JDK

The JDK has a private Java Virtual Machine (JVM) and a few other resources necessary for the development of a Java Application

**JDK contains:**

Java Runtime Environment (JRE),

An interpreter/loader (Java),

A compiler (javac),

An archiver (jar) and many more.

The Java Runtime Environment in JDK is usually called Private Runtime because it is separated from the regular JRE and has extra content. The Private Runtime in JDK contains a JVM and all the class libraries present in the production environment, as well as additional libraries useful to developers, e.g, internationalization libraries and the IDL libraries.

**Most Popular JDKs:**

Oracle JDK: the most popular JDK and the main distributor of Java11,

OpenJDK: Ready for use: JDK 15, JDK 14, and JMC,

Azul Systems Zing: efficient and low latency JDK for Linux os,

Azul Systems: based Zulu brand for Linux, Windows, Mac OS X,

IBM J9 JDK: for AIX, Linux, Windows, and many other OS,

Amazon Corretto: the newest option with the no-cost build of OpenJDK and long-term support.

**Set-Up:**

Setting up JDK in your development environment is super easy, just follow the below simple steps.

**Installation of JDK**

Go to this Oracle’s official Download Page through this link

Select the latest JDK version and click Download and add it to your classpath.

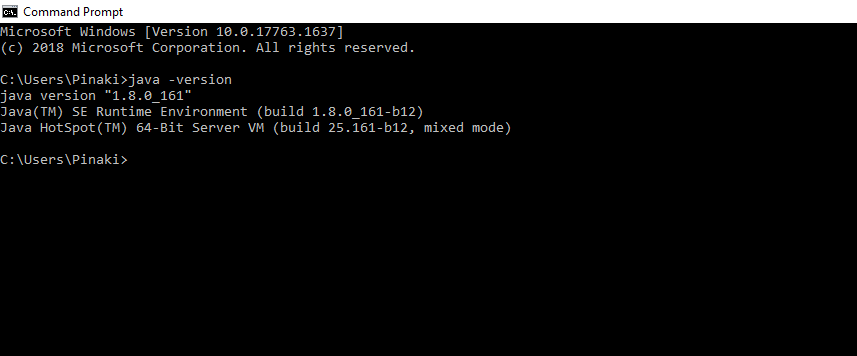
Just check the JDK software is installed or not on your computer at the correct location, for example, at C:\Program Files\Java\jdk11.0.9.

Set JAVA\_HOME for Windows:

Right-click My Computer and select Properties.

Go to the Advanced tab and select Environment Variables, and then edit JAVA\_HOME to point to the exact location where your JDK software is stored, for example, C:\Program Files\Java\jdk11.0.9 is the default location in windows.

Java maintains backward compatibility, so don’t worry just download the latest release and you will get all the old and many new features. After Installing the JDK and JRE adds the java command to your command line. You can verify this through the command prompt by the java -version command. In some cases, you need to restart your system after installing the JDK.



compile and Run Java Code using JDK:

You can use the JDK compiler to convert your Java text file into an executable program. Your Java text segment is converted into bytecode after compilation which carries the .class extension.

First, create a Java text file and save it using a name. Here we are saving the file as Hello.java.

class Hello{

public static void main (String[] args) {

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

}

}

After that just simply use the javac command, which is used for the compilation purpose in Java. Please don’t forget to provide the full path of your java text file to the command line else you will get an error as “The system cannot find the path specified”,

Your command should be similar to the given below example where Hello is the file name and the full path to the file is specified before the file name. The path and javac.exe should be inside the quotes.

“C:\Program Files\Java\jdk-11.0.9\bin\javac.exe” Hello.java

You can notice now that the Hello.class file is being created in the same directory as Hello.java. Now you can run your code by simply using the java Hello command, which will give you the desired result according to your code. Please remember that you don’t have to include the .class to run your code.

C:\Users\Pinaki\Documents>java hello\_world

(Output:) Hello Geek!

**The Jar component:**

JDK contains many useful tools and among them, the most popular after javac is the jar tool. The jar file is nothing but a full pack of Java classes. After creating the .class files, you can put them together in a .jar, which compresses and structures them in a predictable fashion. Now, let’s convert our Hello.class to a jar file.

Before proceeding, please note that you should be in the same directory where the Hello.java file was saved. Now type the command given below in the command line.

**Creating a .jar file**

C:\Users\Pinaki\Documents>”c:\Program Files\Java\jdk-11.0.9\bin\jar.exe**” –create –file Hello.jar** **Hello.class**

Now you can notice that Hello.jar file had been created in the same directory using Hello.class file and jar.exe. You can use the jar file by adding it to your classpath and executing the program inside it. Here the -cp stands for classpath which helps to add the jar to the same classpath.

**Executing the .jar file**

**java -cp hello\_world.jar hello\_world**

**Important Components of JDK**

Below there is a comprehensive list of mostly used components of Jdk which are very useful during the development of a java application.

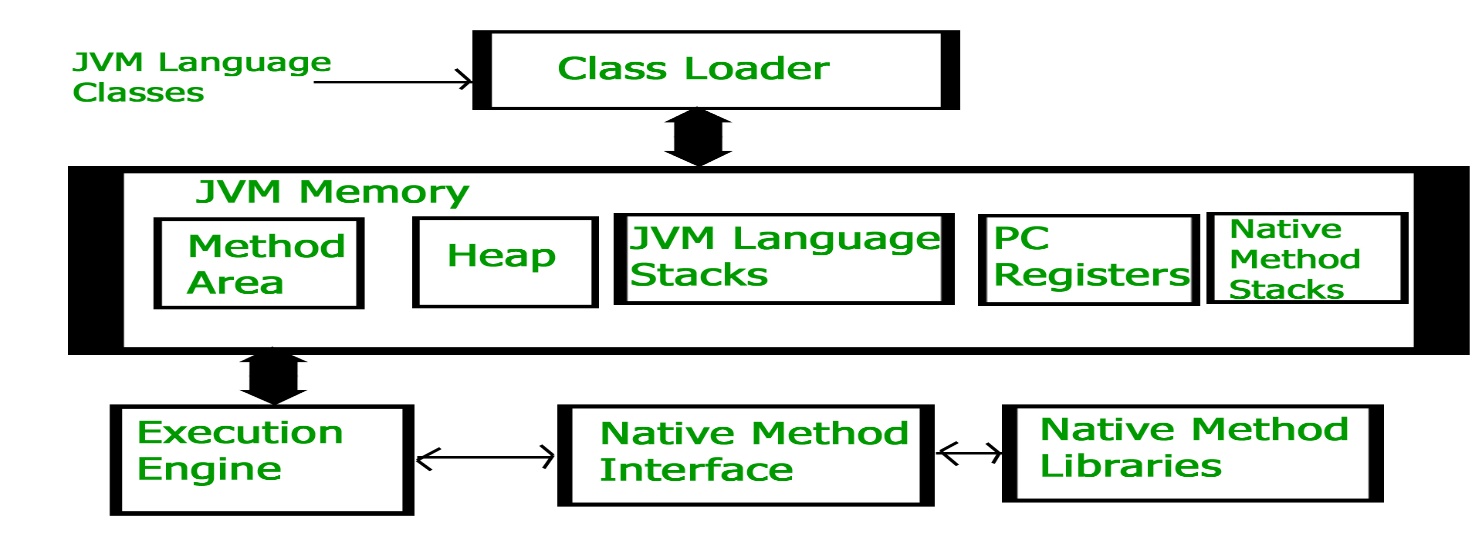
| Component | Use |
| --- | --- |
| javac | Java compiler converts source code into Java bytecode |
| java | The loader of the java apps. |
| javap | Class file disassembler, |
| javadoc | Documentation generator, |
| jar | Java Archiver helps manage JAR files. |
| appletviewer | Debugging of Java applets without a web browser, |
| xjc | Accepts an XML schema and generates Java classes, |
| apt | Annotation-processing tool, |
| jdb | Debugger, |
| jmc | Java Mission Control, |
| JConsole | Monitoring and Management Console, |
| pack200 | JAR compression tool, |
| extcheck | Utility tool to detects JAR file conflicts, |
| idlj | IDL-to-Java compiler, |
| keytool | The keystore manipulating tool, |
| jstatd | jstat daemon (experimental) |
| jstat | JVM statistics monitoring tool |
| jshell | jshell introduced in java 9. |
| jstack | Prints Java stack traces(experimental) |
| jrunscript | Java command-line script shell. |
| jhat | Java Heap Analysis Tool (experimental) |
| jpackage | Generate self-contained application bundles. |
| javaws | Web Start launcher for JNLP applications, |
| javah | C header and stub generator, |
| jarsigner | jar signing and verification tool |
| jinfo | configuration information(experimental) |
| javafxpackager | Package and sign JavaFX applications |

**How JVM Works – JVM Architecture?**

JVM(Java Virtual Machine) acts as a **run-time engine** to run Java applications. JVM is the one that actually calls the **main method** present in a Java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

When we compile a **.java file**, .**class files**(contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.



**Class Loader Subsystem**

It is mainly responsible for three activities.

* Loading
* Linking
* Initialization

**Loading:** The Class loader reads the “.class” file, generate the corresponding binary data and save it in the method area. For each “.class” file, JVM stores the following information in the method area.

The fully qualified name of the loaded class and its immediate parent class.

Whether the “.class” file is related to Class or Interface or Enum.

Modifier, Variables and Method information etc.

After loading the “.class” file, JVM creates an object of **type Class** to represent this file in the heap memory. Please note that this object is of type Class predefined in **java.lang package**. These Class object can be used by the programmer for getting **class level information** like the name of the **class, parent name, methods and variable information etc**. To get this object reference we can use **getClass()** method of Object class.

// A Java program to demonstrate working

// of a Class type obje.ct created by JVM

// to represent .class file in memory.

import java.lang.reflect.Field;

import java.lang.reflect.Method;

// Java code to demonstrate use

// of Class object created by JVM

public class Test {

public static void main(String[] args)

{

Student s1 = new Student();

// Getting hold of Class

// object created by JVM.

Class c1 = s1.getClass();

// Printing type of object using c1.

System.out.println(c1.getName());

// getting all methods in an array

Method m[] = c1.getDeclaredMethods();

for (Method method : m)

System.out.println(method.getName());

// getting all fields in an array

Field f[] = c1.getDeclaredFields();

for (Field field : f)

System.out.println(field.getName());

}

}

// A sample class whose information

// is fetched above using its Class object.

class Student {

private String name;

private int roll\_No;

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public int getRoll\_no() { return roll\_No; }

public void setRoll\_no(int roll\_no)

{

this.roll\_No = roll\_no;

}

}

**Output**

Student

getName

setName

getRoll\_no

setRoll\_no

name

roll\_No

Note: For every loaded “.class” file, only one object of the class is created.

Student s2 = new Student();

// c2 will point to same object where

// c1 is pointing

Class c2 = s2.getClass();

System.out.println(c1==c2); // true

**Linking**: Performs verification, preparation, and (optionally) resolution.

**Verification**: It ensures the correctness of the **.class file** i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run-time exception java.lang.VerifyError. This activity is done by the component ByteCodeVerifier. Once this activity is completed then the class file is ready for compilation.

**Preparation**: JVM allocates memory for class static variables and initializing the memory to default values.

**Resolution: It** is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

**Initialization:** In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy.

In general, there are three class loaders :

**Bootstrap class loader**: Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in the “JAVA\_HOME/jre/lib” directory. This path is popularly known as the bootstrap path. It is implemented in native languages like C, C++.

**Extension class loade**r: It is a child of the bootstrap class loader. It loads the classes present in the extensions directories “JAVA\_HOME/jre/lib/ext”(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the sun.misc.Launcher$ExtClassLoader class.

**System/Application class loader:** It is a child of the extension class loader. It is responsible to load classes from the application classpath. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the sun.misc.Launcher$AppClassLoader class.

// Java code to demonstrate Class Loader subsystem

public class Test {

public static void main(String[] args)

{

// String class is loaded by bootstrap loader, and

// bootstrap loader is not Java object, hence null

System.out.println(String.class.getClassLoader());

// Test class is loaded by Application loader

System.out.println(Test.class.getClassLoader());

}

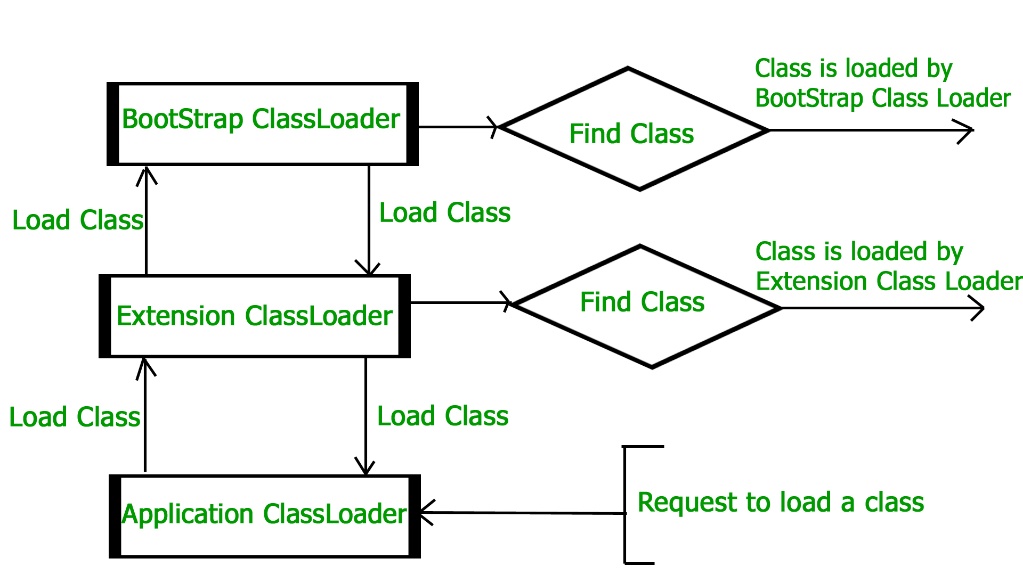
}

**Output**

null

jdk.internal.loader.ClassLoaders$AppClassLoader@8bcc55f

Note: JVM follows the Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to the bootstrap class loader. If a class found in the boot-strap path, the class is loaded otherwise request again transfers to the extension class loader and then to the system class loader. At last, if the system class loader fails to load class, then we get run-time exception java.lang.ClassNotFoundException.



**JVM Memory**

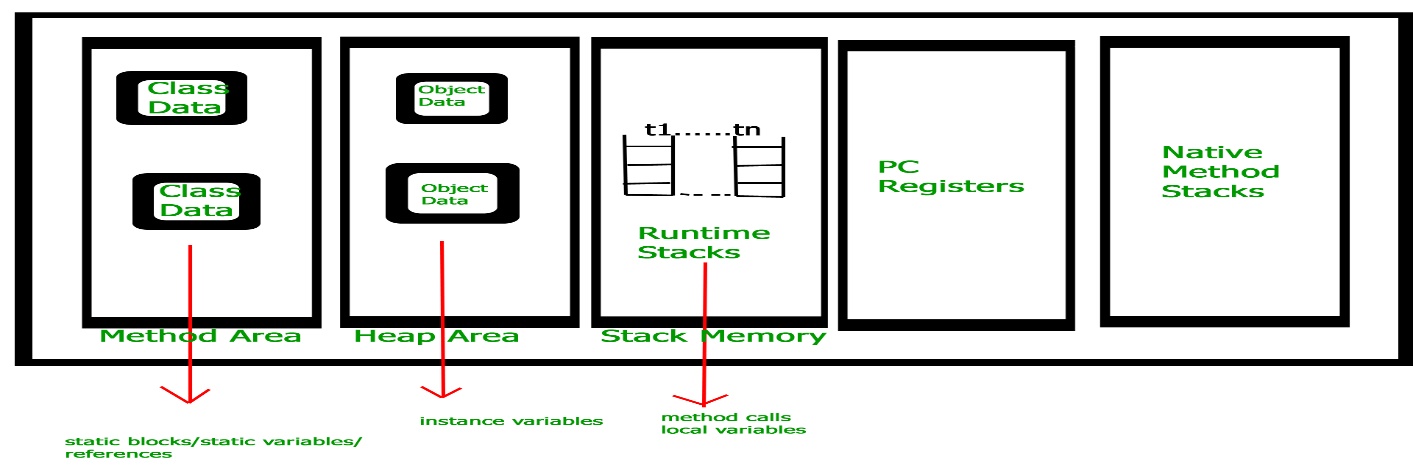
**Method area:** In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.

**Heap area:** Information of all objects is stored in the heap area. There is also one Heap Area per JVM. It is also a shared resource.

**Stack area:** For every thread, JVM creates one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.

**PC Registers**: Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.

**Native method stacks**: For every thread, a separate native stack is created. It stores native method information.



**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 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. For more on Garbage Collector, refer Garbage Collector.

**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.

**Native Method Libraries :** It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

**Differences between JDK, JRE and JVM**

**Java Development Kit (JDK)** is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

Now we need an environment to make a run of our program. Henceforth, JRE stands for “Java Runtime Environment” and may also be written as “Java RTE.” The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files.

Now let us discuss JVM, which stands out for java virtual machines. It is as follows:

A specification where the working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.

An implementation is a computer program that meets the requirements of the JVM specification.

**Runtime Instance** Whenever you write a java command on the command prompt to run the java class, an instance of JVM is created.

Before proceeding to the differences between JDK, JRE, and JVM, let us discuss them in brief first and interrelate them with the image below proposed.



**1. JDK (Java Development Kit)** is a Kit that provides the environment to develop and execute(run) the Java program. JDK is a kit(or package) that includes two things

1. Development Tools(to provide an environment to develop your java programs)
2. JRE (to execute your java program).

**2. JRE (Java Runtime Environment)** is an installation package that provides an environment to only run(not develop) the java program(or application)onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

**3. JVM (Java Virtual Machine)** is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an interpreter.

Now let us discuss the components of **JRE** in order to understand its importance of it and perceive how it actually works. For this let us discuss components.

The **components of JRE** are as follows:

**Deployment technologies**, including deployment, Java Web Start, and Java Plug-in.

* User interface toolkits, including Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag, and drop (DnD), and input methods.
* Integration libraries, including Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP), and scripting.
* Other base libraries, including international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization, and Java for XML Processing (XML JAXP).
* Lang and util base libraries, including lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects, and Regular Expressions.

**Java Virtual Machine (JVM**), including **Java HotSpot Client** and **Server Virtual Machines**.

After having an adequate understanding of the components, now let us discuss the working of JDK. In order to understand how JDK works, let us consider an illustration below as follows:

Illustration:

Consider a java source file saved as ‘Example.java’. The file is compiled into a set of Byte Code that is stored in a “.class” file. Here it will be “Example.class“.



Note: From above, media operation computing during the compile time can be interpreted.

The following actions occur at **runtime** as listed below:

* Class Loader
* Byte Code Verifier
* Interpreter
* Execute the Byte Code

Make appropriate calls to the underlying hardware

Now let us discuss in brief how **JVM works out**. It is as follows:

JVM becomes an instance of JRE at the runtime of a Java program. It is widely known as a runtime interpreter. JVM largely helps in the abstraction of inner implementation from the programmers who make use of libraries for their programs from JDK.

It is mainly responsible for **three activities**.

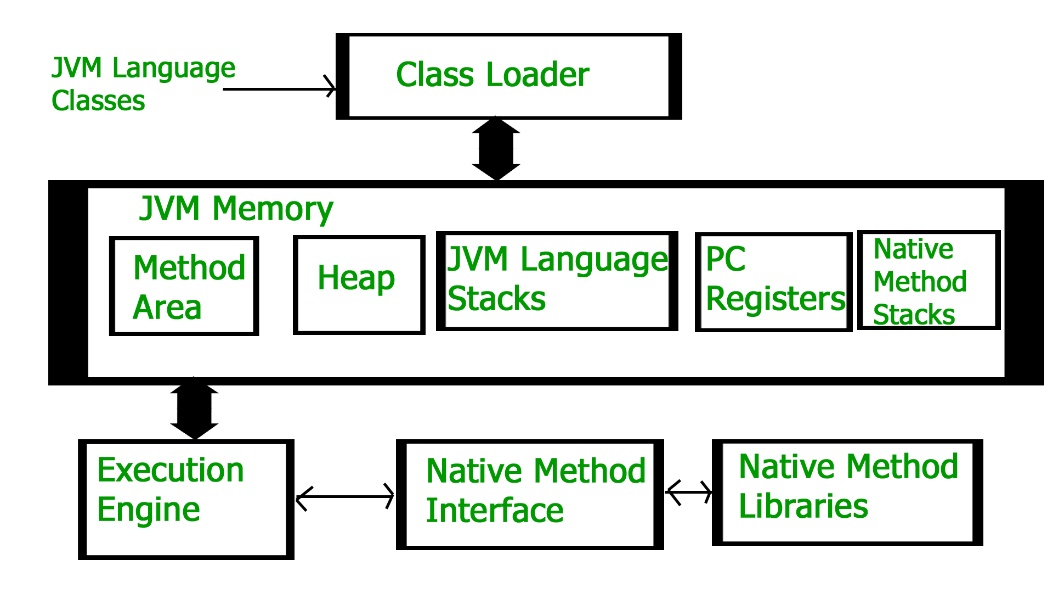
* **Loading**
* **Linking**
* **Initialization**

Similarly, now let us discuss the **working of JRE** which is as follows:

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main method** present in a java code. JVM is a part of JRE(Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustments. This is all possible because of JVM.

When we compile a .java file, .class files(contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.

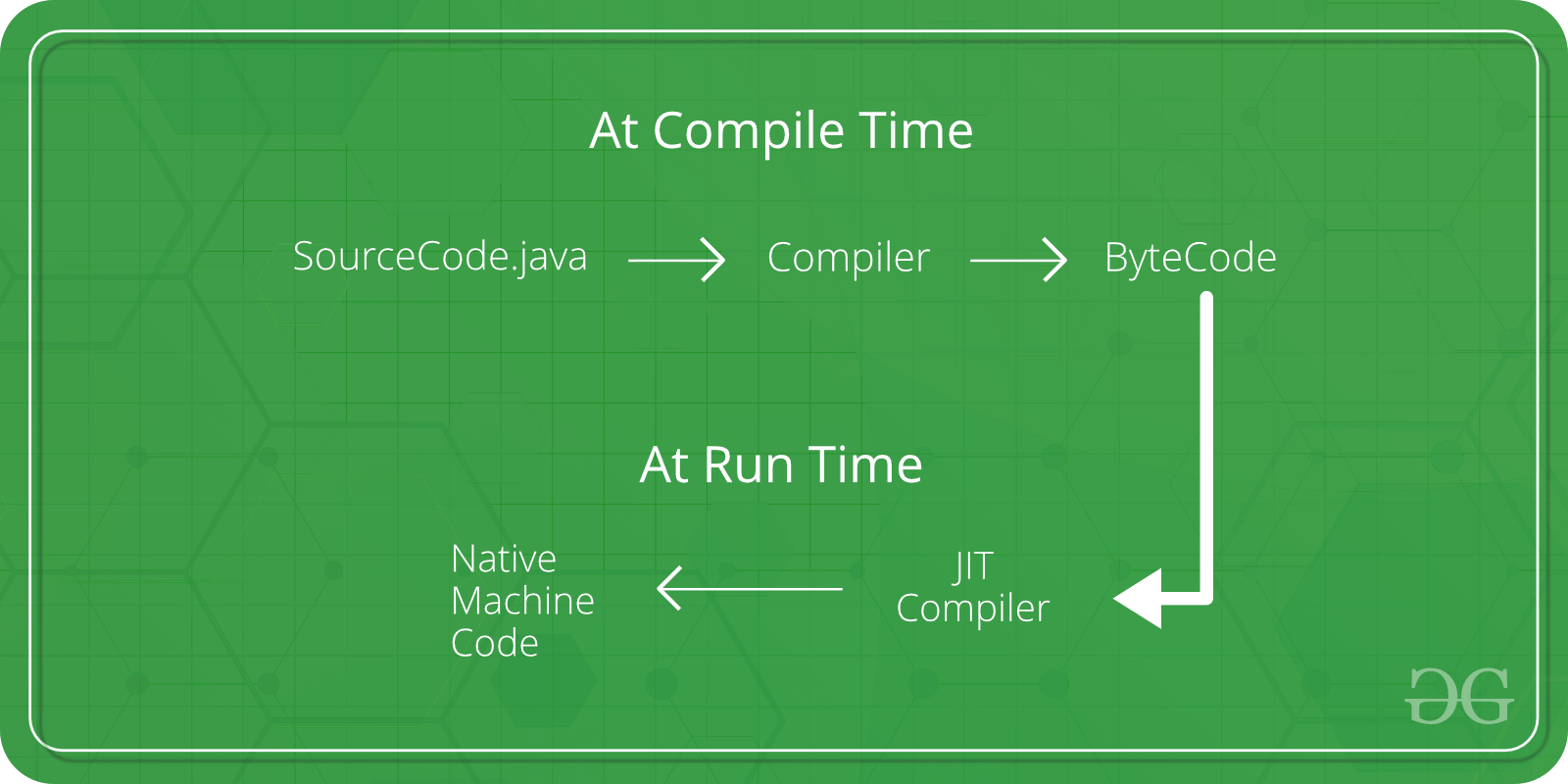


The **JIT** or Just-In-Time compiler is an essential part of the JRE (Java Runtime Environment), that is responsible for performance optimization of java based applications during run time. The compiler is one of the key aspects in deciding the performance of an application for both parties i.e. the end-user and the application developer. Let us check the Just In Time Compiler in Java in more detail.

**Java JIT Compiler**

Bytecode is one of the most important features of java that aids in cross-platform execution. The way of converting **bytecode to native machine language** for execution has a huge impact on its speed of it. These bytecodes have to be interpreted or compiled to proper machine instructions depending on the instruction set architecture. Moreover, these can be directly executed if the instruction architecture is bytecode based. Interpreting the bytecode affects the speed of execution. In order to improve performance, JIT compilers interact with the Java Virtual Machine (JVM) at run time and compile suitable bytecode sequences into native machine code. While using a JIT compiler, the hardware is able to execute the native code, as compared to having the JVM interpret the same sequence of bytecode repeatedly and incurring overhead for the translation process. This subsequently leads to performance gains in the execution speed, unless the compiled methods are executed less frequently.

The JIT compiler is able to perform certain simple optimizations while compiling a series of bytecode to native machine language. Some of these optimizations performed by JIT compilers are data analysis, reduction of memory accesses by register allocation, translation from stack operations to register operations, elimination of common sub-expressions, etc. The greater the degree of optimization done, the more time a JIT compiler spends in the execution stage. Therefore it cannot afford to do all the optimizations that a static compiler is capable of, because of the extra overhead added to the execution time and moreover its view of the program is also restricted.



**Working on JIT Compiler**

Java follows an object-oriented approach, as a result, it consists of classes. These constitute bytecode that is platform neutral and are executed by the JVM across diversified architectures.

At run time, the JVM loads the class files, the semantics of each are determined, and appropriate computations are performed. The additional processor and memory usage during interpretation make a Java application perform slowly as compared to a native application.

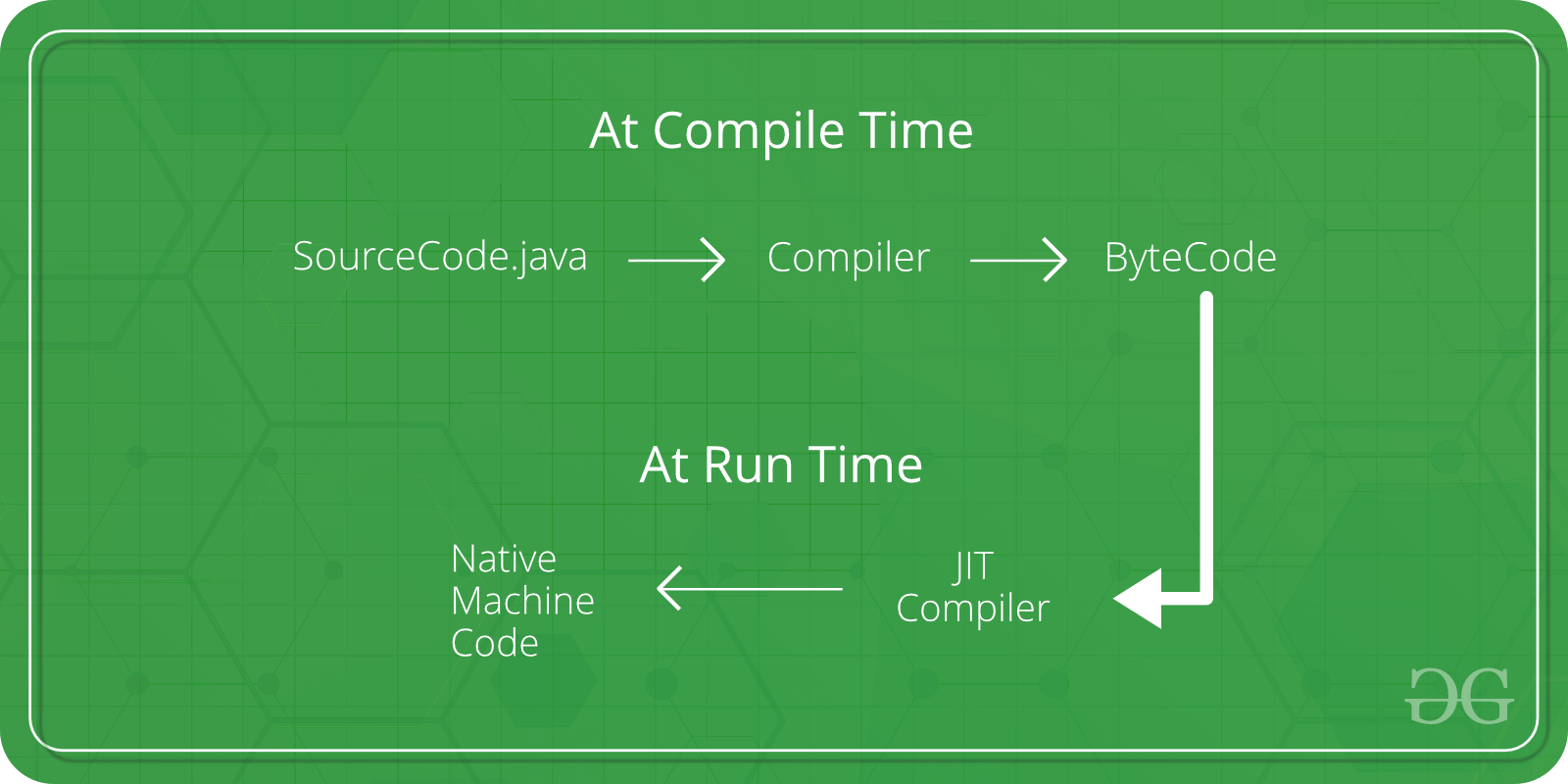
The JIT compiler aids in improving the performance of Java programs by compiling bytecode into native machine code at run time.

The JIT compiler is enabled throughout, while it gets activated when a method is invoked. For a compiled method, the JVM directly calls the compiled code, instead of interpreting it. Theoretically speaking, If compiling did not require any processor time or memory usage, the speed of a native compiler and that of a Java compiler would have been the same.

JIT compilation requires processor time and memory usage. When the java virtual machine first starts up, thousands of methods are invoked. Compiling all these methods can significantly affect startup time, even if the end result is a very good performance optimization.

**Difference between JIT and JVM in Java**

Java Virtual Machine (JVM) is used in the java runtime environment(JRE). The original JVM was conceived as a bytecode interpreter. This may come as a bit of a surprise because of performance problems. Many modern languages are meant to be compiled into CPU-specific, executable code. The fact that the JVM executes a Java program, however, helps address the major issues associated with web-based applications.



The fact that the JVM executes a Java program also helps to make it stable. Since the JVM is in charge, program execution is controlled by it. Therefore, it is possible for the JVM to build a limited execution area called a sandbox that contains the software, preventing the system from getting unlimited access. Protection is also improved by some limitations in the Java language that exists. Java’s JVM architecture includes a class loader, execution engine, memory field, etc.

In order to understand differences, let’s dig down to the components by illustrating the working of JVM alongside.

**ClassLoader:** The class loader has the purpose of loading class files. It helps accomplish three main functions: Loading, Initialization, and Linking.

**JVM language Stacks**: Java memory stores local variables, and partial results of a computation. Each thread has its own JVM stack, created as the thread is created. When the method is invoked, a new frame is created, and then removed.

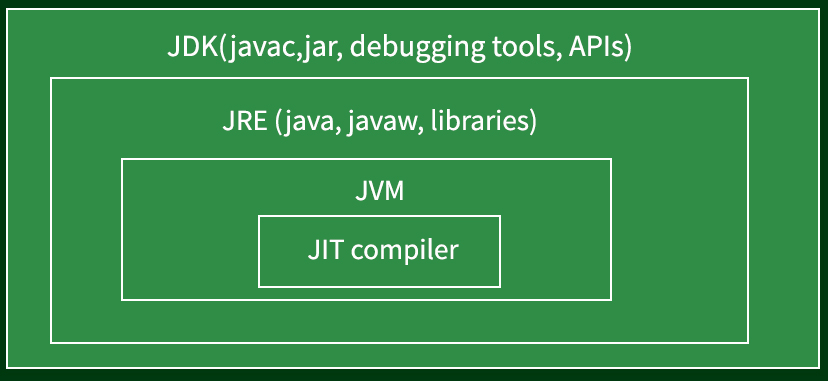
**Method Area:** JVM Method Area specializes in storing the metadata and code-behind files for Java applications.

**PC Registers:** The Java Virtual Machine Instruction address currently being executed is saved by PC registers. Each thread in Java has its own separate PC register.

**Heap:** In a heap are saved all objects, arrays, and instance variables. This memory is shared between several threads.

**Execution Engine:** It is a form of software used for the testing of software, hardware, or complete systems. The test execution engine never carries any information concerning the product being tested.

**Native Method Libraries** which are the Executing Engine needs Native Libraries (C, C++) and the native method interface which is a framework for programming is the Native Method Interface. This enables the Java code that runs in a JVM to call libraries and native applications. Also, the native method stacks have a native code command depending on the native library. It assigns storage to native heaps or uses any stack type.



**Just In Time(JIT) compiler**

While Java was developed as an interpreted language, in order to improve performance, there is nothing about Java that prevents bytecode compilation into native code on the fly. For that reason, not long after Java’s initial release, the HotSpot JVM was released. A just-in-time (JIT) bytecode compiler is included in HotSpot. A Just In Time(JIT) compiler is part of the JVM and on a piece-by-piece demand basis, selected portions of bytecode are compiled into executable code in real-time. That is, as is necessary during execution, a JIT compiler compiles code. In addition, not all bytecode sequences are compiled, only those that will benefit from the compilation. The just-in-time method, however, still yields a major boost in inefficiency. The portability and safety function still exists even though dynamic compilation is applied to bytecode since the JVM is still in control of the execution environment.

In order to understand differences, let’s dig down to the components by illustrating the working of JIT alongside.

Interpreting the bytecode, the standard implementation of the JVM slows the execution of the programs. JIT compilers interact with JVM at runtime to improve performance and compile appropriate bytecode sequences into native machine code.

Hardware is interpreting the code instead of JVM (Java Virtual Machine). This can lead to performance gains in the speed of execution. This can be done per-file, per-function, or maybe on any arbitrary code fragment; the code is often compiled when it’s close to being executed (hence the name “just-in-time”), and then cached and reused later without having to be recompiled. It performs many optimizations: data analysis, translation from stack operations to registry operations, reduction of memory access by registry allocation, elimination of common sub-expressions.

Hence, from the above knowledge, we landed on the conclusive differences between them as mentioned in the table below:

| **JVM** | **JIT** |
| --- | --- |
| JVM stands for Java Virtual Machine. | JIT stands for Just-in-time compilation. |
| JVM was introduced for managing system memory and providing a transportable execution environment for Java-based applications | JIT was invented to improve the performance of JVM after many years of its initial release. |
| JVM consists of many other components like stack area, heap area, etc. | JIT is one of the components of JVM. |
| JVM compiles complete byte code to machine code. | JIT compiles only the reusable byte code to machine code. |
| JVM provides platform independence. | JIT improves the performance of JVM. |

**Difference between Byte Code and Machine Code**

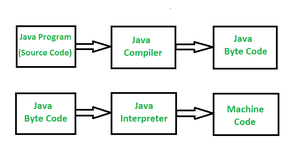
**Byte code** is an intermediate code between the source code and machine code. It is a low-level code that is the result of the compilation of a source code which is written in a high-level language. It is processed by a virtual machine like Java Virtual Machine (JVM).

Byte code is a non-runnable code after it is translated by an interpreter into machine code then it is understandable by the machine. It is compiled to run on JVM, any system having JVM can run it irrespective of their operating system. That’s why Java is platform-independent. Byte code is referred to as a Portable code.

**Machine Code:**

Machine code is a set of instructions that is directly machine-understandable and it is processed by the Central Processing Unit (CPU). Machine code is in binary (0’s and 1’s) format which is completely different from the byte code and source code. It is regarded as the most lowest-level representation of the source code. Machine code is obtained after compilation or interpretation. It is also called machine language.

The below figure illustrates the example of how Java source code is converted to Byte code and then to machine code :



**Difference between Byte Code and Machine Code:**

|  |  |  |
| --- | --- | --- |
| **S.NO.** | **Byte Code** | **Machine Code** |
| 01. | Byte Code consisting of binary, hexadecimal, macro instructions like (new, add, swap, etc) and it is not directly understandable by the CPU. It is designed for efficient execution by software such as a virtual machine.intermediate-level | Machine code consisting of binary instructions that are directly understandable by the CPU. |
| 02. | Byte code is considered as the intermediate-level code. | Machine Code is considered as the low-level code. |
| 03. | Byte code is a non-runnable code generated after compilation of source code and it relies on an interpreter to get executed. | Machine code is a set of instructions in machine language or in binary format and it is directly executed by CPU. |
| 04. | Byte code is executed by the virtual machine then the Central Processing Unit. | Machine code is not executed by a virtual machine it is directly executed by CPU. |
| 05. | Byte code is less specific towards machine than the machine code. | Machine code is more specific towards machine than the byte code. |
| 06. | It is platform-independent as it is dependent on the virtual machine and the system having a virtual machine can be executed irrespective of the platform. | It is not platform independent because the object code of one platform can not be run on the same Operating System. Object varies depending upon system architecture and native instructions associated with the machine. |
| 07. | All the source code need not be converted into byte code for execution by CPU. Some source code written by any specific high-level language is converted into byte code then byte code to object code for execution by CPU. | All the source code must be converted into machine code before it is executed by the CPU. |

**How is Java platform independent?**

The meaning of Java platform-independent is that the Java compiled code(byte code) can run on all operating systems. A program is written in a language that is a human-readable language. It may contain words, phrases, etc which the machine does not understand. For the source code to be understood by the machine, it needs to be in a language understood by machines, typically a machine-level language. So, here comes the role of a compiler. The compiler converts the high-level language (human language) into a format understood by the machines.

Therefore, a compiler is a program that translates the source code for another program from a programming language into executable code. This executable code may be a sequence of machine instructions that can be executed by the CPU directly, or it may be an intermediate representation that is interpreted by a virtual machine. This intermediate representation in Java is the Java Byte Code.

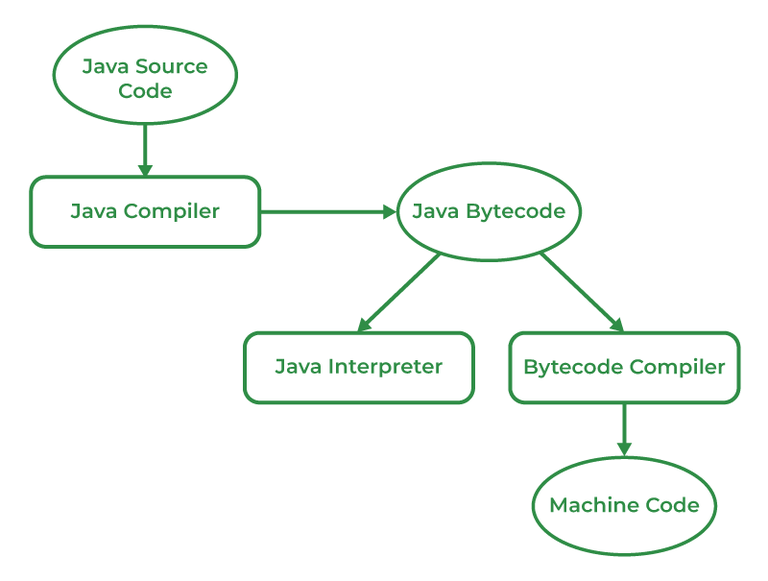
**Step-by-Step Execution of Java Program**

Whenever a program is written in JAVA, the javac compiles it.

The result of the JAVA compiler is the .class file or the bytecode and not the machine’s native code (unlike the C compiler).

The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.

And finally, the program runs to give the desired output.



In the case of C or C++ (languages that are not platform independent), the compiler generates a .exe file which is OS dependent. When we try to run this .exe file on another OS it does not run, since it is OS-dependent and hence is not compatible with the other OS.

**Why Java is platform-independent but JVM is platform dependent?**

In Java, the main point here is that the JVM depends on the operating system – so if you are running Mac OS X you will have a different JVM than if you are running Windows or some other operating system. This fact can be verified by trying to download the JVM for your particular machine – when trying to download it, you will be given a list of JVMs corresponding to different operating systems, and you will obviously pick whichever JVM is targeted for the operating system that you are running. So we can conclude that JVM is platform-dependent and it is the reason why Java is able to become “Platform Independent”.

**important Points:**

In the case of Java, it is the magic of Bytecode that makes it platform-independent.

This adds to an important feature in the JAVA language termed portability. Every system has its own JVM which gets installed automatically when the JDK software is installed. For every operating system separate JVM is available which is capable to read the .class file or byte code.

An important point to be noted is that while JAVA is a platform-independent language, the JVM is platform-dependent. Different JVM is designed for different OS and byte code is able to run on different OS.

Note: As JVM is not platform-dependent because of which Java is not considered completely platform-independent.

**Java Basic Syntax :** Java program is an object-oriented programming language, that means java is the collection of objects, and these objects communicate through method calls to each other to work together.

Basic terminologies in Java

**1. Class:** The class is a blueprint (plan) of the instance of a class (object). It can be defined as a logical template that share common properties and methods.

Example1: Blueprint of the house is class.

Example2: In real world, Alice is an object of the “Human” class.

**2. Object:** The object is an instance of a class. It is an entity that has behavior and state.

Example: Dog, Cat, Monkey etc. are the object of “Animal” class.

Behavior: Running on the road.

**3. Method:** The behavior of an object is the method.

Example: The fuel indicator indicates the amount of fuel left in the car.

**4. Instance variables:** Every object has its own unique set of instance variables. The state of an object is generally created by the values that are assigned to these instance variables.

Example: Steps to compile and run a java program in a console

javac GFG.java

java GFG

import java.util.\*;

public class GFG {

public static void main(String[] args)

{

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

}

}

GeeksforGeeks!

Note: When the class is public, the name of the file has to be the class name.

Syntax:

**1. Comments in Java**

There are three types of comments in Java.

**i. Single line Comment**

// System.out.println("This is an comment.");

**ii. Multi-line Comment**

/\*

System.out.println("This is the first line comment.");

System.out.println("This is the second line comment.");

\*/

**iii. Documentation Comment.** Also called a doc comment.

/\*\* documentation \*/

**2. Source File Name :** The name of a source file should exactly match the public class name with the extension of .java. The name of the file can be a different name if it does not have any public class. Assume you have a public class GFG.

GFG.java // valid syntax

gfg.java // invalid syntax

**3. Case Sensitivity :** Java is a case-sensitive language, which means that the identifiers AB, Ab, aB, and ab are different in Java.

System.out.println("GeeksforGeeks"); // valid syntax

system.out.println("GeeksforGeeks"); // invalid syntax because of the first letter of System keyword is always uppercase.

**. Class Names**

i. The first letter of the class should be in Uppercase (lowercase is allowed but discouraged).

ii. If several words are used to form the name of the class, each inner word’s first letter should be in Uppercase. Underscores are allowed, but not recommended. Also allowed are numbers and currency symbols, although the latter are also discouraged because they are used for a special purpose (for inner and anonymous classes).

class MyJavaProgram // valid syntax

class 1Program // invalid syntax

class My1Program // valid syntax

class $Program // valid syntax, but discouraged

class My$Program // valid syntax, but discouraged (inner class Program inside the class My)

class myJavaProgram // valid syntax, but discouraged

**5. public static void main(String [] args)**

The method main() is the main entry point into a Java program; this is where the processing starts. Also allowed is the signature public static void main(String… args).

**6. Method Names**

i. All the method names should start with a lowercase letter (uppercase is also allowed but lowercase is recommended).

ii. If several words are used to form the name of the method, then each first letter of the inner word should be in Uppercase. Underscores are allowed, but not recommended. Also allowed are digits and currency symbols.

public void employeeRecords() // valid syntax

public void EmployeeRecords() // valid syntax, but discouraged

**7. Identifiers in java :** Identifiers are the names of local variables, instance and class variables, and labels, but also the names for classes, packages, modules and methods. All Unicode characters are valid, not just the ASCII subset.

i. All identifiers can begin with a letter, a currency symbol or an underscore (\_). According to the convention, a letter should be lower case for variables.

ii. The first character of identifiers can be followed by any combination of letters, digits, currency symbols and the underscore. The underscore is not recommended for the names of variables. Constants (static final attributes and enums) should be in all Uppercase letters.

iii. Most importantly identifiers are case-sensitive.

iv. A keyword cannot be used as an identifier since it is a reserved word and has some special meaning.

Legal identifiers: MinNumber, total, ak74, hello\_world, $amount, \_under\_value

Illegal identifiers: 74ak, -amount

**8. White spaces in Java**

A line containing only white spaces, possibly with the comment, is known as a blank line, and the Java compiler totally ignores it.

**9. Access Modifiers:** These modifiers control the scope of class and methods.

Access Modifiers: default, public, protected, private.

Non-access Modifiers: final, abstract, static, transient, synchronized, volatile, native.

**10. Understanding Access Modifiers:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Access Modifier | Within Class | Within Package | Outside Package by subclass only | Outside Package |
| Private | Yes | No | No | No |
| Default | Yes | Yes | No | No |
| Protected | Yes | Yes | Yes | No |
| Public | Yes | Yes | Yes | Yes |

**11. Java Keywords**

Keywords or Reserved words are the words in a language that are used for some internal process or represent some predefined actions. These words are therefore not allowed to use as variable names or objects.

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | assert | boolean | break |
| byte | case | catch | char |
| class | const | continue | default |
| do | double | else | enum |
| extends | final | finally | float |
| for | goto | if | implements |
| import | instanceof | int | interface |
| long | native | new | package |
| private | protected | public | return |
| short | static | strictfp | super |
| switch | synchronized | this | throw |
| throws | transient | try | void |
| volatile | while |  |  |

**Java Hello World Program**

Java is one of the most popular and widely used programming languages and platforms. Java is fast, reliable, and secure. Java is used in every nook and corner from desktop to web applications, scientific supercomputers to gaming consoles, cell phones to the Internet. In this article, we will learn how to write a simple Java Program.

**Steps to Implement Java Program**

Implementation of a Java application program involves the following step. They include:

**Creating the program**

**Compiling the program**

**Running the program**

**1. Creating Programs in Java**

We can create a program using Text Editor (Notepad) or IDE (NetBeans)

class Test

{

public static void main(String []args)

{

System.out.println("My First Java Program.");

}

};

**2. Compiling the Program in Java**

To compile the program, we must run the Java compiler (javac), with the name of the source file on the “command prompt” like as follows

If everything is OK, the “javac” compiler creates a file called “Test.class” containing the byte code of the program.

**3. Running the Program in Java**

We need to use the Java Interpreter to run a program. Java is easy to learn, and its syntax is simple and easy to understand. It is based on C++ (so easier for programmers who know C++).

The process of Java programming can be simplified in three steps:

Create the program by typing it into a text editor and saving it to a file – HelloWorld.java.

Compile it by typing “javac HelloWorld.java” in the terminal window.

Execute (or run) it by typing “java HelloWorld” in the terminal window.

The below-given program is the most simple program of Java printing “Hello World” to the screen. Let us try to understand every bit of code step by step.

// This is a simple Java program.

// FileName : "HelloWorld.java".

class HelloWorld {

// Your program begins with a call to main().

// Prints "Hello, World" to the terminal window.

public static void main(String args[])

{

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

}

}

**Output**

Hello, World

The complexity of the above method

Time Complexity: O(1)

Space Complexity: O(1)

The “Hello World!” program consists of three primary components: the HelloWorld class definition, the main method, and source code comments. The following explanation will provide you with a basic understanding of the code:

**1. Class Definition**

This line uses the keyword class to declare that a new class is being defined.

class HelloWorld {

//

//Statements

}

**2. HelloWorld**

It is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace “{” and the closing curly brace “}“.

**3. main Method**

In the Java programming language, every application must contain a main method. The main function(method) is the entry point of your Java application, and it’s mandatory in a Java program. whose signature in Java is:

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

Explanation of the above syntax

**public:** So that JVM can execute the method from anywhere.

**static:** The main method is to be called without an object. The modifiers are public and static can be written in either order.

**void:** The main method doesn’t return anything.

**main():** Name configured in the JVM. The main method must be inside the class definition. The compiler executes the codes starting always from the main function.

**String[]:** The main method accepts a single argument, i.e., an array of elements of type String.

Like in C/C++, the main method is the entry point for your application and will subsequently invoke all the other methods required by your program.

The next line of code is shown here. Notice that it occurs inside the main() method.

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

This line outputs the string “Hello, World” followed by a new line on the screen. Output is accomplished by the built-in println( ) method. The System is a predefined class that provides access to the system and out is the variable of type output stream connected to the console.

**Comments**

They can either be multiline or single-line comments.

// This is a simple Java program.

// Call this file "HelloWorld.java".

This is a single-line comment. This type of comment must begin with // as in C/C++. For multiline comments, they must begin from /\* and end with \*/.

**Important Points**

The name of the class defined by the program is HelloWorld, which is the same as the name of the file(HelloWorld.java). This is not a coincidence. In Java, all codes must reside inside a class, and there is at most one public class which contains the main() method.

By convention, the name of the main class(a class that contains the main method) should match the name of the file that holds the program.

Every Java program must have a class definition that matches the filename (class name and file name should be same).

**Compiling the Program**

After successfully setting up the environment, we can open a terminal in both Windows/Unix and go to the directory where the file – HelloWorld.java is present.

Now, to compile the HelloWorld program, execute the compiler – javac, to specify the name of the source file on the command line, as shown:

javac HelloWorld.java

The compiler creates a HelloWorld.class (in the current working directory) that contains the bytecode version of the program. Now, to execute our program, JVM(Java Virtual Machine) needs to be called using java, specifying the name of the class file on the command line, as shown:

java HelloWorld

This will print “Hello World” to the terminal screen.



**Java Data Types**

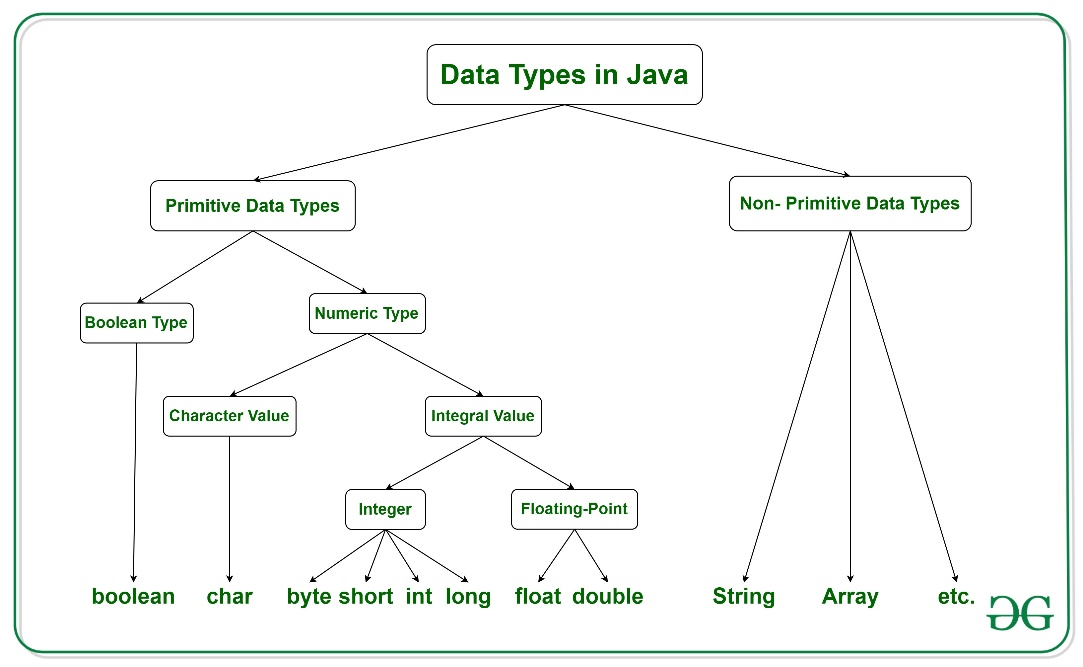
Java is statically typed and also a strongly typed language because, in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the Java data types.

**Data Types in Java**

Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated

**Primitive Data Type**: such as boolean, char, int, short, byte, long, float, and double

**Non-Primitive Data** Type or Object Data type: such as String, Array, etc.



**Primitive Data Types in Java** Primitive data are only single values and have no special capabilities. There are 8 primitive data types. They are depicted below in tabular format below as follows:

| **Type** | **Description** | **Default** | **Size** | **Example Literals** | **Range of values** |
| --- | --- | --- | --- | --- | --- |
| **boolean** | true or false | false | 1 bit | true, false | true, false |
| **byte** | twos-complement integer | 0 | 8 bits | (none) | -128 to 127 |
| **char** | Unicode character | \u0000 | 16 bits | ‘a’, ‘\u0041’, ‘\101’, ‘\\’, ‘\’, ‘\n’, ‘β’ | characters representation of ASCII values  0 to 255 |
| **short** | twos-complement integer | 0 | 16 bits | (none) | -32,768 to 32,767 |
| **int** | twos-complement intger | 0 | 32 bits | -2,-1,0,1,2 | -2,147,483,648  to  2,147,483,647 |
| **long** | twos-complement integer | 0 | 64 bits | -2L,-1L,0L,1L,2L | -9,223,372,036,854,775,808  to  9,223,372,036,854,775,807 |
| **float** | IEEE 754 floating point | 0.0 | 32 bits | 1.23e100f , -1.23e-100f , .3f ,3.14F | upto 7 decimal digits |
| **double** | IEEE 754 floating point | 0.0 | 64 bits | 1.23456e300d , -123456e-300d , 1e1d | upto 16 decimal digit |

Let us discuss and implement each one of the following data types that are as follows:

**1. Boolean Data Type**

Boolean data type represents only one bit of information either true or false which is intended to represent the two truth values of logic and Boolean algebra, but the size of the boolean data type is virtual machine-dependent. Values of type boolean are not converted implicitly or explicitly (with casts) to any other type. But the programmer can easily write conversion code.

**Syntax:**

**boolean booleanVar;**

**Size: Virtual machine dependen**t

**2. Byte Data Type**

The byte data type is an 8-bit signed two’s complement integer. The byte data type is useful for saving memory in large arrays.

**Syntax:**

byte byteVar;

Size: 1 byte (8 bits)

**3. Short Data Type**

The short data type is a 16-bit signed two’s complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.

**Syntax:**

short shortVar;

Size: 2 bytes (16 bits)

**4. Integer Data Type**

It is a 32-bit signed two’s complement integer.

**Syntax:**

int intVar;

Size: 4 bytes ( 32 bits )

Remember: In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has a value in the range [0, 232-1]. Use the Integer class to use the int data type as an unsigned integer.

**5. Long Data Type**

The range of a long is quite large. The long data type is a 64-bit two’s complement integer and is useful for those occasions where an int type is not large enough to hold the desired value. The size of the Long Datatype is 8 bytes (64 bits).

Syntax:

**long longVar;**

Remember: In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 264-1. The Long class also contains methods like comparing Unsigned, divide Unsigned, etc to support arithmetic operations for unsigned long.

**6. Float Data Type**

The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers. The size of the float data type is 4 bytes (32 bits).

**Syntax:**

float floatVar;

**7. Double Data Type**

The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice. The size of the double data type is 8 bytes or 64 bits.

**Syntax:**

double doubleVar;

Note: Both float and double data types were designed especially for scientific calculations, where approximation errors are acceptable. If accuracy is the most prior concern then, it is recommended not to use these data types and use BigDecimal class instead.

It is recommended to go through rounding off errors in java.

**8. Char Data Type**

The char data type is a single 16-bit Unicode character with the size of 2 bytes (16 bits).

Syntax:

char charVar;

**Why is the Size of char 2 bytes in Java?**

So, other languages like C/C++ use only ASCII characters, and to represent all ASCII characters 8 bits is enough. But Java uses the Unicode system not the ASCII code System and to represent the Unicode system 8 bits is not enough to represent all characters so Java uses 2 bytes for characters. Unicode defines a fully international character set that can represent most of the world’s written languages. It is a unification of dozens of character sets, such as Latin, Greek, Cyrillic, Katakana, Arabic, and many more.

Example:

// Java Program to Demonstrate Char Primitive Data Type

// Class

class GFG {

// Main driver method

public static void main(String args[])

{

// Creating and initializing custom character

char a = 'G';

// Integer data type is generally

// used for numeric values

int i = 89;

// use byte and short

// if memory is a constraint

byte b = 4;

// this will give error as number is

// larger than byte range

// byte b1 = 7888888955;

short s = 56;

// this will give error as number is

// larger than short range

// short s1 = 87878787878;

// by default fraction value

// is double in java

double d = 4.355453532;

// for float use 'f' as suffix as standard

float f = 4.7333434f;

// need to hold big range of numbers then we need

// this data type

long l = 12121;

System.out.println("char: " + a);

System.out.println("integer: " + i);

System.out.println("byte: " + b);

System.out.println("short: " + s);

System.out.println("float: " + f);

System.out.println("double: " + d);

System.out.println("long: " + l);

}

}

Output

char: G

integer: 89

byte: 4

short: 56

float: 4.7333436

double: 4.355453532

long: 12121

**Non-Primitive Data Type or Reference Data Types**

The Reference Data Types will contain a memory address of variable values because the reference types won’t store the variable value directly in memory. They are strings, objects, arrays, etc.

**1. Strings**

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

**Syntax: Declaring a string**

<String\_Type> <string\_variable> = “<sequence\_of\_string>”;

**Example:**

// Declare String without using new operator

**String s = "GeeksforGeeks**";

// Declare String using new operator

**String s1 = new String("GeeksforGeeks");**

**2. Class**

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

**Modifiers**: A class can be public or has default access. Refer to access specifiers for classes or interfaces in Java

**Class name:** The name should begin with an initial letter (capitalized by convention).

**Superclass**(if any): The name of the class’s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.

**Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.

Body: The class body is surrounded by braces, { }.

**3. Object** An Object is a basic unit of Object-Oriented Programming and represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

**State:** It is represented by the attributes of an object. It also reflects the properties of an object.

**Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.

**Identity:** It gives a unique name to an object and enables one object to interact with other objects.

**4. 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 the blueprint of the class.
* An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
* If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
* A Java library example is Comparator Interface. If a class implements this interface, then it can be used to sort a collection.
* **5. Array :** An Array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. The following are some important points about Java arrays.
* In Java, all arrays are dynamically allocated. (discussed below)
* 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.
* The variables in the array are ordered and each has an index beginning with 0.
* Java array can also be used as a 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 type is Object.
* Every array type implements the interfaces Cloneable and java.io.Serializable.

**FAQs of Data Types in Java**

1. What are Data Types in Java?

Data types are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases.

**2. What are the 8 Data Types that use in Java?**

There are 8 main primitive data types in java as mentioned below:

boolean

byte

char

short

int

long

float

double

**3. Which is a Primitive Type in Java?**

Primitive data types are the types in java that can store a single value and do not provide any special capability.

**4. Why char uses 2 bytes in Java and what is \u0000?**

Char uses 2 bytes in java because it uses the Unicode system rather than the ASCII system. “\u000” is the lowest range of the Unicode system.

**Difference between the primitive and object data types in Java:**

Now let’s look at a program that demonstrates the difference between the primitive and object data types in Java.

import java.lang.\*;

import java.util.\*;

class GeeksForGeeks {

public static void main(String ar[])

{

System.out.println("PRIMITIVE DATA TYPES\n");

int x = 10;

int y = x;

System.out.print("Initially: ");

System.out.println("x = " + x + ", y = " + y);

// Here the change in the value of y

// will not affect the value of x

y = 30;

System.out.print("After changing y to 30: ");

System.out.println("x = " + x + ", y = " + y);

System.out.println(

"\*\*Only value of y is affected here "

+ "because of Primitive Data Type\n");

System.out.println("REFERENCE DATA TYPES\n");

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

// Here complete reference of c is copied to d

// and both point to same memory in Heap

int[] d = c;

System.out.println("Initially");

System.out.println("Array c: "

+ Arrays.toString(c));

System.out.println("Array d: "

+ Arrays.toString(d))

// Modifying the value at

// index 1 to 50 in array d

System.out.println("\nModifying the value at "

+ "index 1 to 50 in array d\n");

d[1] = 50;

System.out.println("After modification");

System.out.println("Array c: "

+ Arrays.toString(c));

System.out.println("Array d: "

+ Arrays.toString(d));

System.out.println(

"\*\*Here value of c[1] is also affected "

+ "because of Reference Data Type\n");

}

}

**Output**

**PRIMITIVE DATA TYPES**

Initially: x = 10, y = 10

After changing y to 30: x = 10, y = 30

\*\*Only value of y is affected here because of Primitive Data Type

**REFERENCE DATA TYPES**

Initially

Array c: [10, 20, 30, 40]

Array d: [10, 20, 30, 40]

**Modifying the value at index 1 to 50 in array d**

After modification

Array c: [10, 50, 30, 40]

Array d: [10, 50, 30, 40]

\*\*Here value of c[1] is also affected because of Reference Data Type

Let’s look at the difference between the primitive and object data types in a tabular manner as shown below as follows:

| **Properties** | **Primitive data types** | **Objects** |
| --- | --- | --- |
| Origin | Pre-defined data types | User-defined data types |
| Stored structure | Stored in a stack | Reference variable is stored in stack and the original object is stored in heap |
| When copied | Two different variables is created along with different assignment(only values are same) | Two reference variable is created but both are pointing to the same object on the heap |
| When changes are made in the copied variable | Change does not reflect in the original ones. | Changes reflected in the original ones. |
| Default value | Primitive datatypes do not have null as default value | The default value for the reference variable is null |
| Example | byte, short, int, long, float, double, char, boolean | array, string class, interface etc. |

**Java Identifiers**

In Java, identifiers are used for identification purposes. Java Identifiers can be **a class name, method name, variable name, or label.**

**Example of Java Identifiers**

public class Test

{

public static void main(String[] args)

{

int a = 20;

}

}

In the above Java code, we have 5 identifiers namely :

**Test**: class name.

**main:** method name.

**String:** predefined class name.

**args:** variable name.

**a:** variable name.

* **Rules For Defining Java Identifiers**
* There are certain rules for defining a valid Java identifier. These rules must be followed, otherwise, we get a compile-time error. These rules are also valid for other languages like C, and C++.
* The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), ‘$‘(dollar sign) and ‘\_‘ (underscore).For example “geek@” is not a valid Java identifier as it contains a ‘@’ a special character.
* Identifiers should not start with digits([0-9]). For example “123geeks” is not a valid Java identifier.
* Java identifiers are case-sensitive.
* There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.
* Reserved Words can’t be used as an identifier. For example “int while = 20;” is an invalid statement as a while is a reserved word. There are 53 reserved words in Java.

**Examples of valid identifiers :**

* MyVariable
* MYVARIABLE
* myvariable
* x
* i
* x1
* i1
* \_myvariable
* $myvariable
* sum\_of\_array
* geeks123

**Examples of invalid identifiers :**

* My Variable // contains a space
* 123geeks // Begins with a digit
* a+c // plus sign is not an alphanumeric character
* variable-2 // hyphen is not an alphanumeric character
* sum\_&\_difference // ampersand is not an alphanumeric character

**Reserved Words in Java**

Any programming language reserves some words to represent functionalities defined by that language. These words are called reserved words. They can be briefly categorized into two parts: **keywords(50)** and **literals(3). Keywords define functionalities and literals define value**. Identifiers are used by symbol tables in various analyzing phases(like lexical, syntax, and semantic) of a compiler architecture.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | continue | for | protected | transient |
| Assert | Default | Goto | public | Try |
| Boolean | Do | If | Static | throws |
| break | double | implements | strictfp | Package |
| byte | else | import | super | Private |
| case | enum | Interface | Short | switch |
| Catch | Extends | instanceof | return | void |
| Char | Final | Int | synchronized | volatile |
| class | finally | long | throw | Date |
| const | float | Native | This | while |

Note: The keywords const and goto are reserved, even though they are not currently used. In place of const, the final keyword is used. Some keywords like strictfp are included in later versions of Java.

**Operators in Java**

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. In this article, we will learn about Java Operators and learn all their types.

**What are the Java Operators?**

Operators in Java are the symbols used for performing specific operations in Java. Operators make tasks like addition, multiplication, etc which look easy although the implementation of these tasks is quite complex.

**Types of Operators in Java**

There are multiple types of operators in Java all are mentioned below:

* **Arithmetic Operators**
* **Unary Operators**
* **Assignment Operator**
* **Relational Operators**
* **Logical Operators**
* **Ternary Operator**
* **Bitwise Operators**
* **Shift Operators**
* **instance of operator**

**1. Arithmetic Operators**

They are used to perform simple arithmetic operations on primitive data types.

* : Multiplication
* / : Division
* % : Modulo
* + : Addition
  + : Subtraction

Example:

// Java Program to implement

// Arithmetic Operators

import java.io.\*;

// Drive Class

class GFG {

// Main Function

public static void main (String[] args) {

// Arithmetic operators

int a = 10;

int b = 3;

System.out.println("a + b = " + (a + b));

System.out.println("a - b = " + (a - b));

System.out.println("a \* b = " + (a \* b));

System.out.println("a / b = " + (a / b));

System.out.println("a % b = " + (a % b));

}

}

**Output**

a + b = 13

a - b = 7

a \* b = 30

a / b = 3

a % b = 1

**2. Unary Operators**

Unary operators need only one operand. They are used to increment, decrement, or negate a value.

* + : Unary minus, used for negating the values.
* + : Unary plus indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.
* ++ : Increment operator, used for incrementing the value by 1. There are two varieties of increment operators.
* Post-Increment: Value is first used for computing the result and then incremented.
* Pre-Increment: Value is incremented first, and then the result is computed.
  + – : Decrement operator, used for decrementing the value by 1. There are two varieties of decrement operators.
* Post-decrement: Value is first used for computing the result and then decremented.
* Pre-Decrement: The value is decremented first, and then the result is computed.
* ! : Logical not operator, used for inverting a boolean value.

Example:

// Java Program to implement

// Uniary Operators

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main(String[] args)

{

// Interger declared

int a = 10;

int b = 10;

// Using uniary operators

System.out.println("Postincrement : " + (a++));

System.out.println("Preincrement : " + (++a));

System.out.println("Postdecrement : " + (b--));

System.out.println("Predecrement : " + (--b));

}

}

**Output**

Postincrement : 10

Preincrement : 12

Postdecrement : 10

Predecrement : 8

**3. Assignment Operator**

‘=’ Assignment operator is used to assign a value to any variable. It has right-to-left associativity, i.e. value given on the right-hand side of the operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.

The general format of the assignment operator is:

variable = value;

In many cases, the assignment operator can be combined with other operators to build a shorter version of the statement called a Compound Statement. For example, instead of a = a+5, we can write a += 5.

* +=, for adding the left operand with the right operand and then assigning it to the variable on the left.
* -=, for subtracting the right operand from the left operand and then assigning it to the variable on the left.
* \*=, for multiplying the left operand with the right operand and then assigning it to the variable on the left.
* /=, for dividing the left operand by the right operand and then assigning it to the variable on the left.
* %=, for assigning the modulo of the left operand by the right operand and then assigning it to the variable on the left.

Example:

// Java Program to implement

// Assignment Operators

import java.io.\*;

// Driver Class

class GFG {

// Main Function

public static void main(String[] args)

{

// Assignment operators

int f = 7;

System.out.println("f += 3: " + (f += 3));

System.out.println("f -= 2: " + (f -= 2));

System.out.println("f \*= 4: " + (f \*= 4));

System.out.println("f /= 3: " + (f /= 3));

System.out.println("f %= 2: " + (f %= 2));

System.out.println("f &= 0b1010: " + (f &= 0b1010));

System.out.println("f |= 0b1100: " + (f |= 0b1100));

System.out.println("f ^= 0b1010: " + (f ^= 0b1010));

System.out.println("f <<= 2: " + (f <<= 2));

System.out.println("f >>= 1: " + (f >>= 1));

System.out.println("f >>>= 1: " + (f >>>= 1));

}

}

Output

f += 3: 10

f -= 2: 8

f \*= 4: 32

f /= 3: 10

f %= 2: 0

f &= 0b1010: 0

f |= 0b1100: 12

f ^= 0b1010: 6

f <<= 2: 24

f >>= 1: 12

f >>>= 1: 6

**4. Relational Operators**

These operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements. The general format is,

**variable relation\_operator value**

Some of the relational operators are-

1. ==, Equal to returns true if the left-hand side is equal to the right-hand side.
2. !=, Not Equal to returns true if the left-hand side is not equal to the right-hand side.
3. <, less than: returns true if the left-hand side is less than the right-hand side.
4. <=, less than or equal to returns true if the left-hand side is less than or equal to the right-hand side.
5. >, Greater than: returns true if the left-hand side is greater than the right-hand side.
6. >=, Greater than or equal to returns true if the left-hand side is greater than or equal to the right-hand side.

Example:

// Java Program to implement

// Relational Operators

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main(String[] args)

{

// Comparison operators

int a = 10;

int b = 3;

int c = 5;

System.out.println("a > b: " + (a > b));

System.out.println("a < b: " + (a < b));

System.out.println("a >= b: " + (a >= b));

System.out.println("a <= b: " + (a <= b));

System.out.println("a == c: " + (a == c));

System.out.println("a != c: " + (a != c));

}

}

Output

a > b: true

a < b: false

a >= b: true

a <= b: false

a == c: false

a != c: true

**5. Logical Operators**

These operators are used to perform “logical AND” and “logical OR” operations, i.e., a function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Java also has “Logical NOT”, which returns true when the condition is false and vice-versa

Conditional operators are:

1. &&, Logical AND: returns true when both conditions are true.
2. ||, Logical OR: returns true if at least one condition is true.
3. !, Logical NOT: returns true when a condition is false and vice-versa

Example:

// Java Program to implemenet

// Logical operators

import java.io.\*;

// Driver Class

class GFG {

// Main Function

public static void main (String[] args) {

// Logical operators

boolean x = true;

boolean y = false;

System.out.println("x && y: " + (x && y));

System.out.println("x || y: " + (x || y));

System.out.println("!x: " + (!x));

}

}

Output

x && y: false

x || y: true

!x: false

**6. Ternary operator**

The ternary operator is a shorthand version of the if-else statement. It has three operands and hence the name Ternary.

The general format is:

**condition ? if true : if false**

The above statement means that if the condition evaluates to true, then execute the statements after the ‘?’ else execute the statements after the ‘:’.

Example:

// Java program to illustrate

// max of three numbers using

// ternary operator.

public class operators {

public static void main(String[] args)

{

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

// result holds max of three

// numbers

result

= ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);

System.out.println("Max of three numbers = "

+ result);

}

}

Output

Max of three numbers = 30

**7. Bitwise Operators**

These operators are used to perform the manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of the Binary indexed trees.

1. &, Bitwise AND operator: returns bit by bit AND of input values.
2. |, Bitwise OR operator: returns bit by bit OR of input values.
3. ^, Bitwise XOR operator: returns bit-by-bit XOR of input values.
4. ~, Bitwise Complement Operator: This is a unary operator which returns the one’s complement representation of the input value, i.e., with all bits inverted.

// Java Program to implement

// bitwise operators

import java.io.\*;

// Driver class

class GFG {

// main function

public static void main(String[] args)

{

// Bitwise operators

int d = 0b1010;

int e = 0b1100;

System.out.println("d & e: " + (d & e));

System.out.println("d | e: " + (d | e));

System.out.println("d ^ e: " + (d ^ e));

System.out.println("~d: " + (~d));

System.out.println("d << 2: " + (d << 2));

System.out.println("e >> 1: " + (e >> 1));

System.out.println("e >>> 1: " + (e >>> 1));

}

}

Output

d & e: 8

d | e: 14

d ^ e: 6

~d: -11

d << 2: 40

e >> 1: 6

e >>> 1: 6

**8. Shift Operato**rs : These operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively. They can be used when we have to multiply or divide a number by two. General format-

**number shift\_op number\_of\_places\_to\_shift;**

1. <<, Left shift operator: shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as multiplying the number with some power of two.
2. >>, Signed Right shift operator: shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of the initial number. Similar effect to dividing the number with some power of two.
3. >>>, Unsigned Right shift operator: shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

// Java Program to implement

// shift operators

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main(String[] args)

{

int a = 10;

// using left shift

System.out.println("a<<1 : " + (a << 1));

// using right shift

System.out.println("a>>1 : " + (a >> 1));

}

}

Output

a<<1 : 20

a>>1 : 5

**9. instanceof operator :** The instance of the operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass, or an interface. General format-

**object instance of class/subclass/interface**

// Java program to illustrate

// instance of operator

class operators {

public static void main(String[] args)

{

Person obj1 = new Person();

Person obj2 = new Boy();

// As obj is of type person, it is not an

// instance of Boy or interface

System.out.println("obj1 instanceof Person: "

+ (obj1 instanceof Person));

System.out.println("obj1 instanceof Boy: "

+ (obj1 instanceof Boy));

System.out.println("obj1 instanceof MyInterface: "

+ (obj1 instanceof MyInterface));

// Since obj2 is of type boy,

// whose parent class is person

// and it implements the interface Myinterface

// it is instance of all of these classes

System.out.println("obj2 instanceof Person: "

+ (obj2 instanceof Person));

System.out.println("obj2 instanceof Boy: "

+ (obj2 instanceof Boy));

System.out.println("obj2 instanceof MyInterface: "

+ (obj2 instanceof MyInterface));

}

}

class Person {

}

class Boy extends Person implements MyInterface {

}

interface MyInterface {

}

Output

obj1 instanceof Person: true

obj1 instanceof Boy: false

obj1 instanceof MyInterface: false

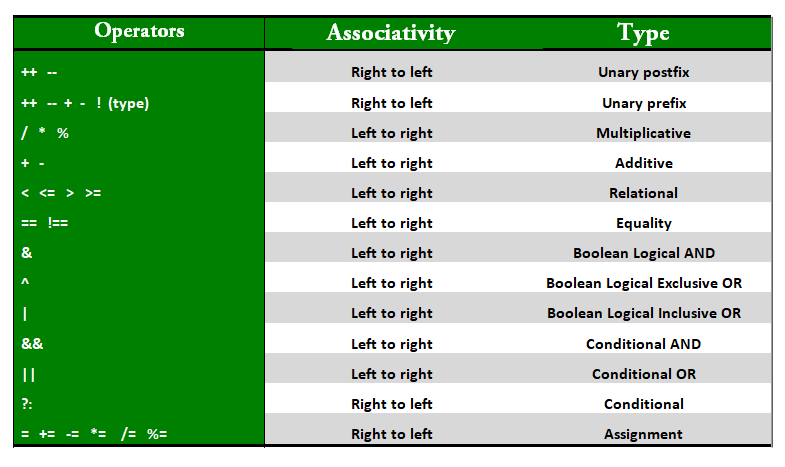
obj2 instanceof Person: true

obj2 instanceof Boy: true

obj2 instanceof MyInterface: true

**Precedence and Associativity of Java Operators**

Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator. In such cases, these rules determine which part of the equation to consider first, as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude, with the top representing the highest precedence and the bottom showing the lowest precedence.



**Interesting Questions about Java Operators**

**1. Precedence and Associativity**:

There is often confusion when it comes to hybrid equations which are equations having multiple operators. The problem is which part to solve first. There is a golden rule to follow in these situations. If the operators have different precedence, solve the higher precedence first. If they have the same precedence, solve according to associativity, that is, either from right to left or from left to right. The explanation of the below program is well written in comments within the program itself.

public class operators {

public static void main(String[] args)

{

int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;

// precedence rules for arithmetic operators.

// (\* = / = %) > (+ = -)

// prints a+(b/d)

System.out.println("a+b/d = " + (a + b / d));

// if same precedence then associative

// rules are followed.

// e/f -> b\*d -> a+(b\*d) -> a+(b\*d)-(e/f)

System.out.println("a+b\*d-e/f = "

+ (a + b \* d - e / f));

}

}

Output

a+b/d = 20

a+b\*d-e/f = 219

**2. Be a Compiler:**

The compiler in our systems uses a lex tool to match the greatest match when generating tokens. This creates a bit of a problem if overlooked. For example, consider the statement a=b+++c; too many of the readers might seem to create a compiler error. But this statement is absolutely correct as the token created by lex is a, =, b, ++, +, c. Therefore, this statement has a similar effect of first assigning b+c to a and then incrementing b. Similarly, a=b+++++c; would generate an error as the tokens generated are a, =, b, ++, ++, +, c. which is actually an error as there is no operand after the second unary operand.

public class operators {

public static void main(String[] args)

{

int a = 20, b = 10, c = 0;

// a=b+++c is compiled as

// b++ +c

// a=b+c then b=b+1

a = b++ + c;

System.out.println("Value of a(b+c), "

+ " b(b+1), c = " + a + ", " + b

+ ", " + c);

// a=b+++++c is compiled as

// b++ ++ +c

// which gives error.

// a=b+++++c;

// System.out.println(b+++++c);

}

}

Output

Value of a(b+c), b(b+1), c = 10, 11, 0

**3. Using + over ():**

When using the + operator inside system.out.println() make sure to do addition using parenthesis. If we write something before doing addition, then string addition takes place, that is, associativity of addition is left to right, and hence integers are added to a string first producing a string, and string objects concatenate when using +. Therefore it can create unwanted results.

public class operators {

public static void main(String[] args)

{

int x = 5, y = 8;

// concatenates x and y as

// first x is added to "concatenation (x+y) = "

// producing "concatenation (x+y) = 5"

// and then 8 is further concatenated.

System.out.println("Concatenation (x+y)= " + x + y);

// addition of x and y

System.out.println("Addition (x+y) = " + (x + y));

}

}

Output

Concatenation (x+y)= 58

Addition (x+y) = 13

**Advantages of Operators in Java**

The advantages of using operators in Java are mentioned below:

* Expressiveness: Operators in Java provide a concise and readable way to perform complex calculations and logical operations.
* Time-Saving: Operators in Java save time by reducing the amount of code required to perform certain tasks.
* Improved Performance: Using operators can improve performance because they are often implemented at the hardware level, making them faster than equivalent Java code.

**Disadvantages of Operators in Java**

The disadvantages of Operators in Java are mentioned below:

* Operator Precedence: Operators in Java have a defined precedence, which can lead to unexpected results if not used properly.
* Type Coercion: Java performs implicit type conversions when using operators, which can lead to unexpected results or errors if not used properly.
* Overloading: Java allows for operator overloading, which can lead to confusion and errors if different classes define the same operator with different behavior.

FAQs in Java Operators

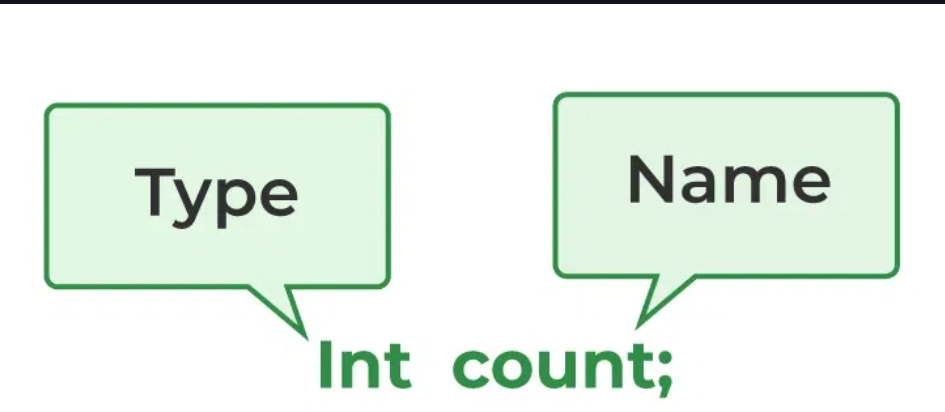
1. What is operators in Java with example?

Operators are the special symbols that are used for performing certain operations. For example, ‘+’ is used for addition where 5+4 will return the value 9.

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

**Variables in Java**

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



From the image, it can be easily perceived that while declaring a variable, we need to take care of two things that are:

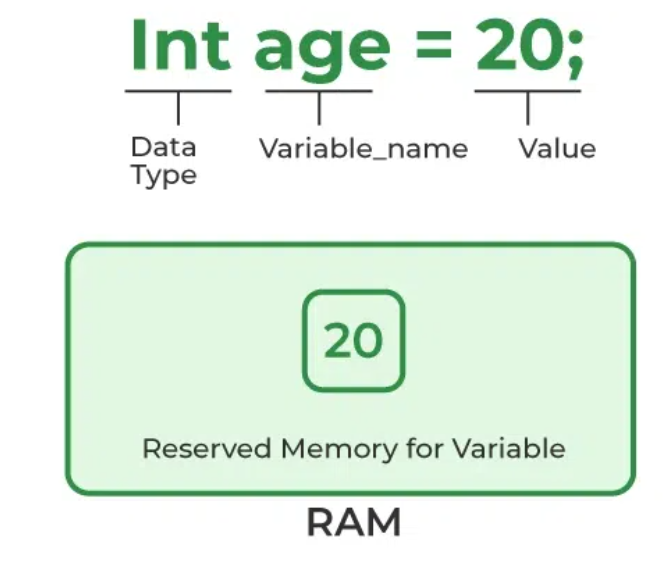
* **datatype**: Type of data that can be stored in this variable.
* **data\_name:** Name was given to the variable.

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

**Variable Initialization**

1. Assigning value by taking input
2. How to Initialize Variables in Java?
3. It can be perceived with the help of 3 components that are as follows:

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



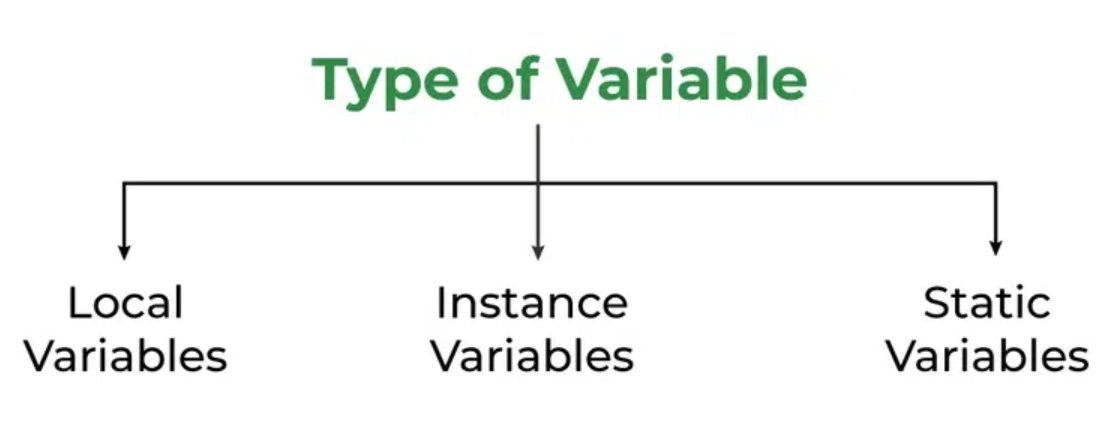
**Illustrations:**

1. // Declaring float variable
2. float simpleInterest;
3. // Declaring and initializing integer variable
4. int time = 10, speed = 20;
5. // Declaring and initializing character variable
6. char var = 'h';

**Types of Variables in Java**

Now let us discuss different types of variables which are listed as follows:

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



Let us discuss the traits of every type of variable listed here in detail.

**1. Local Variables :** A variable defined within a block or method or constructor is called a local variable.

These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.

The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.

Initialization of the local variable is mandatory before using it in the defined scope.

Time Complexity of the Method:

Time Complexity: O(1)

Auxiliary Space: O(1)

Below is the implementation of the above approach:

// Java Program to implement

// Local Variables

import java.io.\*;

class GFG {

public static void main(String[] args)

{

// Declared a Local Variable

int var = 10;

// This variable is local to this main method only

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

}

}

Output

Local Variable: 10

Example :

package a;

public class LocalVariable {

public static void main(String[] args)

{

// x is a local variable

int x = 10;

// message is also a local

// variable

String message = "Hello, world!";

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

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

if (x > 5) {

// result is a

// local variable

String result = "x is greater than 5";

System.out.println(result);

}

// Uncommenting the line below will result in a

// compile-time error System.out.println(result);

for (int i = 0; i < 3; i++) {

String loopMessage

= "Iteration "

+ i; // loopMessage is a local variable

System.out.println(loopMessage);

}

// Uncommenting the line below will result in a

// compile-time error

// System.out.println(loopMessage);

}

}

Output :

message = Hello, world!

x is greater than 5

Iteration 0

Iteration 1

Iteration 2

2. Instance Variables

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.

Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.

Initialization of an instance variable is not mandatory. Its default value is dependent on the data type of variable. For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.

Instance variables can be accessed only by creating objects.

We initialize instance variables using constructors while creating an object. We can also use instance blocks to initialize the instance variables.

The complexity of the method:

Time Complexity: O(1)

Auxiliary Space: O(1)

Below is the implementation of the above approach:

// Java Program to demonstrate

// Instance Variables

import java.io.\*;

class GFG {

// Declared Instance Variable

public String geek;

public int i;

public Integer I;

public GFG()

{

// Default Constructor

// initializing Instance Variable

this.geek = "Shubham Jain";

}

// Main Method

public static void main(String[] args)

{

// Object Creation

GFG name = new GFG();

// Displaying O/P

System.out.println("Geek name is: " + name.geek);

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

+ name.i);

// toString() called internally

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

+ name.I);

}

}

Output

Geek name is: Shubham Jain

Default value for int is 0

Default value for Integer is null

3. Static Variables

Static variables are also known as class variables.

These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor, or block.

Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.

Static variables are created at the start of program execution and destroyed automatically when execution ends.

Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable. For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.

If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won’t halt the program. The compiler will replace the object name with the class name automatically.

If we access a static variable without the class name, the compiler will automatically append the class name. But for accessing the static variable of a different class, we must mention the class name as 2 different classes might have a static variable with the same name.

Static variables cannot be declared locally inside an instance method.

Static blocks can be used to initialize static variables.

The complexity of the method:

Time Complexity: O(1)

Auxiliary Space: O(1)

Below is the implementation of the above approach:

// Java Program to demonstrate

// Static variables

import java.io.\*;

class GFG {

// Declared static variable

public static String geek = "Shubham Jain";

public static void main(String[] args)

{

// geek variable can be accessed without object

// creation Displaying O/P GFG.geek --> using the

// static variable

System.out.println("Geek Name is : " + GFG.geek);

// static int c=0;

// above line,when uncommented,

// will throw an error as static variables cannot be

// declared locally.

}

}

Output

Geek Name is : Shubham Jain

Differences Between the Instance Variables and the Static Variables

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

Each object will have its own copy of an instance variable, whereas we can only have one copy of a static variable per class, irrespective of how many objects we create. Thus, static variables are good for memory management.

Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In the case of a static variable, changes will be reflected in other objects as static variables are common to all objects of a class.

We can access instance variables through object references, and static variables can be accessed directly using the class name.

Instance variables are created when an object is created with the use of the keyword ‘new’ and destroyed when the object is destroyed. Static variables are created when the program starts and destroyed when the program stops.

Syntax: Static and instance variables

class GFG

{

// Static variable

static int a;

// Instance variable

int b;

}

Conclusion

The Important points to remember in the articles are mentioned below:

Variables in Java is a data container that saves the data values during Java program execution.

There are three types of variables in Java Local variables, static variables, and instance variables.

FAQs on Variables in Java

Q1. What are variables in Java?

Variables are the containers in Java that can store data values inside them.

Q2. What are the 3 types of variables in Java?

There are three types of variables in Java are mentioned below:

Local Variables

Static Variables

Instance Variables

Q3. How to declare variables in Java examples?

We can declare variables in java with syntax as mentioned below:

data\_type variable\_name;

Example:

// Integer datatype with var1 name

int var1;

**Scope of Variables In Java**

Scope of a variable is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e.scope of a variable can determined at compile time and independent of function call stack.

Java programs are organized in the form of classes. Every class is part of some package. Java scope rules can be covered under following categories.

**Member Variables (Class Level Scope)**

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let’s take a look at an example:

public class Test

{

// All variables defined directly inside a class

// are member variables

int a;

private String b;

void method1() {....}

int method2() {....}

char c;

}

We can declare class variables anywhere in class, but outside methods.

Access specified of member variables doesn’t affect scope of them within a class.

Member variables can be accessed outside a class with following rules

Modifier Package Subclass World

public Yes Yes Yes

protected Yes Yes No

Default (no

modifier) Yes No No

private No No No

**Local Variables (Method Level Scope)**

Variables declared inside a method have method level scope and can’t be accessed outside the method.

public class Test

{

void method1()

{

// Local variable (Method level scope)

int x;

}

}

**Note** : Local variables don’t exist after method’s execution is over.

Here’s another example of method scope, except this time the variable got passed in as a parameter to the method:

class Test

{

private int x;

public void setX(int x)

{

this.x = x;

}

}

The above code uses this keyword to differentiate between the **local and class variables**.

As an exercise, predict the **output of following Java program**.

public class Test

{

static int x = 11;

private int y = 33;

public void method1(int x)

{

Test t = new Test();

this.x = 22;

y = 44;

System.out.println("Test.x: " + Test.x);

System.out.println("t.x: " + t.x);

System.out.println("t.y: " + t.y);

System.out.println("y: " + y);

}

public static void main(String args[])

{

Test t = new Test();

t.method1(5);

}

}

Output:

Test.x: 22

t.x: 22

t.y: 33

y: 44

**Loop Variables (Block Scope)**

A variable declared inside pair of brackets “{” and “}” in a method has scope within the brackets only.

public class Test

{

public static void main(String args[])

{

{

// The variable x has scope within

// brackets

int x = 10;

System.out.println(x);

}

// Uncommenting below line would produce

// error since variable x is out of scope.

// System.out.println(x);

}

}

**Output:**

10

As another example, consider following program with a for loop.

class Test

{

public static void main(String args[])

{

for (int x = 0; x < 4; x++)

{

System.out.println(x);

}

// Will produce error

System.out.println(x);

}

}

**Output:**

11: error: cannot find symbol

System.out.println(x);

The right way of doing above is,

// Above program after correcting the error

class Test

{

public static void main(String args[])

{

int x;

for (x = 0; x < 4; x++)

{

System.out.println(x);

}

System.out.println(x);

}

}

**Output:**

0

1

2

3

4

Let’s look at tricky example of loop scope. Predict the output of following program. You may be surprised if you are regular C/C++ programmer.

class Test

{

public static void main(String args[])

{

int a = 5;

for (int a = 0; a < 5; a++)

{

System.out.println(a);

}

}

}

Output :

6: error: variable a is already defined in method go(int)

for (int a = 0; a < 5; a++)

^

1 error

Note:- In C++, it will run. But in java it is an error because in java, the name of the variable of inner and outer loop must be different.

A similar program in C++ works. See this.

As an exercise, predict the output of the following Java program.

class Test

{

public static void main(String args[])

{

{

int x = 5;

{

int x = 10;

System.out.println(x);

}

}

}

}

Q. From the above knowledge, tell whether the below code will run or not.

class Test {

public static void main(String args[])

{

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

System.out.println(i);

}

int i = 20;

System.out.println(i);

}

}

Output :

1

2

3

4

5

6

7

8

9

10

20

Yes, it will run!

* See the program carefully, inner loop will terminate before the outer loop variable is declared.So the inner loop variable is destroyed first and then the new variable of same name has been created.

**Some Important Points about Variable scope in Java:**

* In general, a set of curly brackets { } defines a scope.
* In Java we can usually access a variable as long as it was defined within the same set of brackets as the code we are writing or within any curly brackets inside of the curly brackets where the variable was defined.
* Any variable defined in a class outside of any method can be used by all member methods.
* When a method has the same local variable as a member, “this” keyword can be used to reference the current class variable.
* For a variable to be read after the termination of a loop, It must be declared before the body of the loop.

**Wrapper Classes in Java**

A Wrapper class in Java is a class 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. In other words, we can wrap a **primitive value** into a **wrapper class** object. Let’s check on the wrapper classes in Java with examples:

**Need of Wrapper Classes**

1. There are certain needs for using the Wrapper class in Java as mentioned below:
2. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
3. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
4. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
5. An object is needed to support synchronization in multithreading.

**Advantages of Wrapper Classes**

* Collections allowed only object data.
* On object data we can call multiple methods compareTo(), equals(), toString()
* Cloning process only objects
* Object data allowed null values.
* Serialization can allow only object data.

Below are given examples of wrapper classes in Java with their corresponding Primitive data types in Java.

**Primitive Data Types and their Corresponding Wrapper Class**

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

**Autoboxing and Unboxing**

**1. 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.**

// Java program to demonstrate Autoboxing

import java.util.ArrayList;

class Autoboxing {

public static void main(String[] args)

{

char ch = 'a';

// Autoboxing- primitive to Character object

// conversion

Character a = ch;

ArrayList<Integer> arrayList= new ArrayList<Integer>();

// Autoboxing because ArrayList stores only objects

arrayList.add(25);

// printing the values from object

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

}

}

Output

25

**2. 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.**

Example:

// Java program to demonstrate Unboxing

import java.util.ArrayList;

class Unboxing {

public static void main(String[] args)

{

Character ch = 'a';

// unboxing - Character object to primitive

// conversion

char a = ch;

ArrayList<Integer> arrayList = new ArrayList<Integer>();

arrayList.add(24);

// unboxing because get method returns an Integer

// object

int num = arrayList.get(0);

// printing the values from primitive data types

System.out.println(num);

}

}

Output

24

**Java Wrapper Classes Example**

// Java program to demonstrate Wrapping and UnWrapping

// in Classes

import java.io.\*;

class GFG {

public static void main(String[] args)

{

// byte data type

byte a = 1;

// wrapping around Byte object

Byte byteobj = new Byte(a);

// Use with Java 9

// Byte byteobj = Byte.valueOf(a);

// int data type

int b = 10;

// wrapping around Integer object

Integer intobj = new Integer(b);

// Use with Java 9

// Integer intobj = Integer.valueOf(b);

// float data type

float c = 18.6f;

// wrapping around Float object

Float floatobj = new Float(c);

// Use with Java 9

// Float floatobj = Float.valueOf(c);

// double data type

double d = 250.5;

// Wrapping around Double object

Double doubleobj = new Double(d);

// Use with Java 9

// Double doubleobj = Double.valueOf(d);

// char data type

char e = 'a';

// wrapping around Character object

Character charobj = e;

// printing the values from objects

System.out.println(

"Values of Wrapper objects (printing as objects)");

System.out.println("\nByte object byteobj: "

+ byteobj);

System.out.println("\nInteger object intobj: "

+ intobj);

System.out.println("\nFloat object floatobj: "

+ floatobj);

System.out.println("\nDouble object doubleobj: "

+ doubleobj);

System.out.println("\nCharacter object charobj: "

+ charobj);

// objects to data types (retrieving data types from

// objects) unwrapping objects to primitive data

// types

byte bv = byteobj;

int iv = intobj;

float fv = floatobj;

double dv = doubleobj;

char cv = charobj;

// printing the values from data types

System.out.println(

"\nUnwrapped values (printing as data types)");

System.out.println("\nbyte value, bv: " + bv);

System.out.println("\nint value, iv: " + iv);

System.out.println("\nfloat value, fv: " + fv);

System.out.println("\ndouble value, dv: " + dv);

System.out.println("\nchar value, cv: " + cv);

}

}

**Output**

Values of Wrapper objects (printing as objects)

Byte object byteobj: 1

Integer object intobj: 10

Float object floatobj: 18.6

Double object doubleobj: 250.5

Character object charobj: a

Unwrapped values (printing as data types)

byte value, bv: 1

int value, iv: 10

float value, fv: 18.6

double value, dv: 250.5

char value, cv: a

**Custom Wrapper Classes in Java**

Java Wrapper classes wrap the primitive data types. We can create a class that wraps data inside it. So let us check how to create our own custom wrapper class in Java. It can be implemented for creating certain structures like queues, stacks, etc.

Example:

// Java Program to implement

// Custom wrapper class

import java.io.\*;

// wrapper class

class Maximum {

private int maxi = 0;

private int size = 0;

public void insert(int x)

{

this.size++;

if (x <= this.maxi)

return;

this.maxi = x;

}

public int top() { return this.maxi; }

public int elementNumber() { return this.size; }

};

//

class GFG {

public static void main(String[] args)

Maximum x = new Maximum();

x.insert(12);

x.insert(3);

x.insert(23);

System.out.println("Maximum element: " + x.top());

System.out.println("Number of elements inserted: "

+ x.elementNumber());

}

}

**Output**

Maximum element: 23

Number of elements inserted: 3

**FAQs in Wrapper Class**

1. Which are the wrapper classes in Java?

A Wrapper class in Java is a class whose object wraps or contains primitive data types.

2. Why use the wrapper class in Java?

The wrapper class in Java is used to convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method.

3. What are the 8 wrapper classes in Java?

**There are 8 Wrapper classes in Java these are Boolean, Byte, Short, Character, Integer, Long, Float, Double.**

* **Input/Output in Java**

**How to Take Input From User in Java?**

Java brings various Streams with its I/O package that helps the user perform all the Java input-output operations. These streams support all types of objects, data types, characters, files, etc. to fully execute the I/O operations. Input in Java can be with certain methods mentioned below in the article.

* **Methods to Take Input in Java**

There are two ways by which we can take Java input from the user or from a file

1. **BufferedReader Class**
2. **Scanner Class**

**1. Using BufferedReader Class for String Input In Java**

It is a simple class that is used to read a sequence of characters. It has a simple function that reads a character another read which reads, an array of characters, and a readLine() function which reads a line.

1. **InputStreamReader()** is a function that converts the input stream of bytes into a stream of characters so that it can be read as BufferedReader expects a stream of characters. BufferedReader can throw checked Exceptions.

Below is the implementation of the above approach:

**Java ex 1:**

// Java Program for taking user

// input using BufferedReader Class

import java.io.\*;

class GFG {

// Main Method

public static void main(String[] args)

throws IOException

{

// Creating BufferedReader Object

// InputStreamReader converts bytes to

// stream of character

**BufferedReader bfn = new BufferedReader(**

**new InputStreamReader(System.in));**

// String reading internally

String str = bfn.readLine();

// Integer reading internally

int it = Integer.parseInt(bfn.readLine());

// Printing String

System.out.println("Entered String : " + str);

// Printing Integer

System.out.println("Entered Integer : " + it);

}

}

Output

Mayank Solanki

888

Entered String : Mayank Solanki

Entered Integer : 888

Using Buffer Reader Class To Read the Input

Below is the implementation of the above approach:

**Java ex2:**

/\*package whatever //do not write package name here \*/

import java.io.\*;

import java.io.BufferedReader;

import java.io.InputStreamReader;

class Easy {

public static void main(String[] args)

{

// creating the instance of class BufferedReader

BufferedReader reader = new BufferedReader(

new InputStreamReader(System.in));

String name;

try {

System.out.println("Enter your name");

name = reader.readLine(); // taking string input

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

}

catch (Exception e) {

}

}

}

Output:

Enter your name:

Geeks

Name=Geeks

**2. Using Scanner Class for Taking Input in Java**

It is an advanced version of BufferedReader which was added in later versions of Java. The scanner can read formatted input. It has different functions for different types of data types.

* The scanner is much easier to read as we don’t have to write throws as there is no exception thrown by it.
* It was added in later versions of Java
* It contains predefined functions to read an Integer, Character, and other data types as well.
* Syntax of Scanner class
* Scanner scn = new Scanner(System.in);
* Importing Scanner Class
* ‘To use the Scanner we need to import the Scanner class from the util package as

1. import java.util.Scanner;
2. Inbuilt Scanner functions are as follows:
3. Integer: nextInt()
4. Float: nextFloat()
5. String : next() and nextLine()
6. Hence, in the case of Integer and String in Scanner, we don’t require parsing as we did require in BufferedReader.

Java

// Java Program to show how to take

// input from user using Scanner Class

import java.util.\*;

class GFG {

public static void main(String[] args)

{

// Scanner definition

Scanner scn = new Scanner(System.in);

// input is a string ( one word )

// read by next() function

String str1 = scn.next();

// print String

System.out.println("Entered String str1 : " + str1);

// input is a String ( complete Sentence )

// read by nextLine()function

String str2 = scn.nextLine();

// print string

System.out.println("Entered String str2 : " + str2);

// input is an Integer

// read by nextInt() function

int x = scn.nextInt();

// print integer

System.out.println("Entered Integer : " + x);

// input is a floatingValue

// read by nextFloat() function

float f = scn.nextFloat();

// print floating value

System.out.println("Entered FloatValue : " + f);

}

}

Output :

Entered String str1 : Geeks

Entered String str2 : Geeks For Geeks

Entered Integer : 123

Entered FloatValue : 123.090

**Example 2:**

**Java**

// Java Program to implement

// Scanner Class to take input

import java.io.\*;

import java.util.Scanner;

// Driver Class

class Easy {

// main function

public static void main(String[] args)

{

// creating the instance of class Scanner

Scanner obj = new Scanner(System.in);

String name;

int rollno;

float marks;

System.out.println("Enter your name");

// taking string input

name = obj.nextLine();

System.out.println("Enter your rollno");

// taking integer input

rollno = obj.nextInt();

System.out.println("Enter your marks");

// taking float input

marks = obj.nextFloat();

// printing the output

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

System.out.println("Rollno=" + rollno);

System.out.println("Marks=" + marks);

}

}

Output

Enter your name

Geeks

Enter your rollno

5

Enter your marks

84.60

Name=Geeks

Rollno=5

Marks=84.60

**Differences Between BufferedReader and Scanner**

* BufferedReader is a very basic way to read the input generally used to read the stream of characters. It gives an edge over Scanner as it is faster than Scanner because Scanner does lots of post-processing for parsing the input; as seen in nextInt(), nextFloat()
* BufferedReader is more flexible as we can specify the size of stream input to be read. (In general, it is there that BufferedReader reads larger input than Scanner)
* These two factors come into play when we are reading larger input. In general, the Scanner Class serves the input.
* BufferedReader is preferred as it is synchronized. While dealing with multiple threads it is preferred.
* For decent input, and easy readability. The Scanner is preferred over BufferedReader.

**Scanner Class in Java**

* In Java, Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings.
* 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.

**Java Scanner Input Types**

* Scanner class helps to take the standard input stream in Java. So, we need some methods to extract data from the stream. Methods used for extracting data are mentioned below:

| **Method** | **Description** |
| --- | --- |
| **[nextBoolean()](https://www.geeksforgeeks.org/scanner-nextboolean-method-in-java-with-examples/)** | Used for reading Boolean value |
| [**nextByte()**](https://www.geeksforgeeks.org/scanner-nextbyte-method-in-java-with-examples/) | Used for reading Byte value |
| [**nextDouble()**](https://www.geeksforgeeks.org/scanner-nextdouble-method-in-java-with-examples/) | Used for reading Double value |
| [**nextFloat()**](https://www.geeksforgeeks.org/scanner-nextfloat-method-in-java-with-examples/) | Used for reading Float value |
| [**nextInt()**](https://www.geeksforgeeks.org/scanner-nextint-method-in-java-with-examples/) | Used for reading Int value |
| [**nextLine()**](https://www.geeksforgeeks.org/scanner-nextline-method-in-java-with-examples/) | Used for reading Line value |
| [**nextLong()**](https://www.geeksforgeeks.org/scanner-nextlong-method-in-java-with-examples/) | Used for reading Long value |
| [**nextShort()**](https://www.geeksforgeeks.org/scanner-nextshort-method-in-java-with-examples/) | Used for reading Short value |

**Examples of Java Scanner Class**

**Example 1:**

// Java program to read data of various types

// using Scanner class.

import java.util.Scanner;

// Driver Class

public class ScannerDemo1 {

// main function

public static void main(String[] args)

{

// Declare the object and initialize with

// predefined standard input object

Scanner sc = new Scanner(System.in);

// String input

String name = sc.nextLine();

// Character input

char gender = sc.next().charAt(0);

// Numerical data input

// byte, short and float can be read

// using similar-named functions.

int age = sc.nextInt();

long mobileNo = sc.nextLong();

double cgpa = sc.nextDouble();

// Print the values to check if the input was

// correctly obtained.

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

System.out.println("Gender: " + gender);

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

System.out.println("Mobile Number: " + mobileNo);

System.out.println("CGPA: " + cgpa);

}

}

Input

Geek

F

40

9876543210

9.9

Output

Name: Geek

Gender: F

Age: 40

Mobile Number: 9876543210

CGPA: 9.9

Sometimes, we have to check if the next value we read is of a certain type or if the input has ended (EOF marker encountered).

Then, we check if the scanner’s input is of the type we want with the help of hasNextXYZ() functions where XYZ is the type we are interested in. The function returns true if the scanner has a token of that type, otherwise false. For example, in the below code, we have used hasNextInt(). To check for a string, we use hasNextLine(). Similarly, to check for a single character, we use hasNext().charAt(0).

Example 2:

Let us look at the code snippet to read some numbers from the console and print their mean.

// Java program to read some values using Scanner

// class and print their mean.

import java.util.Scanner;

public class ScannerDemo2 {

public static void main(String[] args)

{

// Declare an object and initialize with

// predefined standard input object

Scanner sc = new Scanner(System.in);

// Initialize sum and count of input elements

int sum = 0, count = 0;

// Check if an int value is available

while (sc.hasNextInt()) {

// Read an int value

int num = sc.nextInt();

sum += num;

count++;

}

if (count > 0) {

int mean = sum / count;

System.out.println("Mean: " + mean);

}

else {

System.out.println(

"No integers were input. Mean cannot be calculated.");

}

}

}

Input

1 2 3 4 5

Output

**Mean: 3**

**Important Points About Java Scanner Class**

* 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. For example, Suppose there is an input string: How are you
* In this case, the scanner object will read the entire line and divides the string into tokens: “How”, “are” and “you”. The object then iterates over each token and reads each token using its different methods.

**Java.io.BufferedReader Class in Java**

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. The buffer size may be specified, or the default size may be used. The default is large enough for most purposes. In general, each read request made by a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.

**Constructors of BufferedReader Class**

| **Constructor** | **Action Performed** |
| --- | --- |
| BufferedReader(Reader in) | Creates a buffering character-input stream that uses a default-sized input buffer |
| BufferedReader(Reader in, int sz) | Creates a buffering character-input stream that uses an input buffer of the specified size. |

**Methods of BufferedReader Class**

| **Method Name** | **Action** |
| --- | --- |
| [close()](https://www.geeksforgeeks.org/bufferedreader-close-method-in-java-with-examples/#:~:text=The%20close()%20method%20of,associated%20with%20the%20stream%20operations.&text=Parameters%3A%20This%20method%20does%20not,does%20not%20return%20any%20value.) | Closes the stream and releases any system resources associated with it.Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. Closing a previously closed stream has no effect. |
| [mark()](https://www.geeksforgeeks.org/bufferedreader-mark-method-in-java-with-examples/) | Marks the present position in the stream. Subsequent calls to reset() will attempt to reposition the stream to this point. |
| [markSupported()](https://www.geeksforgeeks.org/bufferedreader-marksupported-method-in-java-with-examples/) | Tells whether this stream supports the mark() operation, which it does. |
| [read()](https://www.geeksforgeeks.org/bufferedreader-read-method-in-java-with-examples/) | Reads a single character. |
| read(char[] cbuf, int off, int len) | Reads characters into a portion of an array. This method implements the general contract of the corresponding read method of the Reader class. As an additional convenience, it attempts to read as many characters as possible by repeatedly invoking the read method of the underlying stream. |
| [readLine()](https://www.geeksforgeeks.org/bufferedreader-readline-method-in-java-with-examples/) | 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. |
| [ready()](https://www.geeksforgeeks.org/bufferedreader-ready-method-in-java-with-examples/) | Tells whether this stream is ready to be read. |
| [reset()](https://www.geeksforgeeks.org/bufferedreader-reset-method-in-java-with-examples/) | Resets the stream to the most recent mark. |
| [skip(long)](https://www.geeksforgeeks.org/bufferedreader-skiplong-method-in-java-with-examples/) | Skips characters. |

**Implementation: The content inside the file is as follows:**

This is first line

this is second line

Example

// Java Program to Illustrate BufferedReader Class

// Via Its Methods

// Importing required classes

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

// Class

class GFG {

// Main driver method

public static void main(String[] args)

throws IOException

{

// Creating object of FileReader and BufferedReader

// class

FileReader fr = new FileReader("file.txt");

BufferedReader br = new BufferedReader(fr);

char c[] = new char[20];

// Illustrating markSupported() method

if (br.markSupported()) {

// Print statement

System.out.println( "mark() method is supported");

// Illustrating mark method

br.mark(100);

}

// File Contents is as follows:

// This is first line

// this is second line

// Skipping 8 characters

br.skip(8);

// Illustrating ready() method

if (br.ready()) {

// Illustrating readLine() method

System.out.println(br.readLine());

// Illustrating read(char c[],int off,int len)

br.read(c);

for (int i = 0; i < 20; i++) {

System.out.print(c[i]);

}

System.out.println();

// Illustrating reset() method

br.reset();

for (int i = 0; i < 8; i++) {

// Illustrating read() method

System.out.print((char)br.read());

}

}

}

}

Output:

mark() method is supported

first line

this is second line

This is

**Difference Between Scanner and Buffered Reader Class in Java**

In Java, Scanner and **Buffered Reader** class are sources that serve as ways of reading inputs. Scanner class is a simple text scanner that can parse primitive types and strings. It internally uses regular expressions to read different types while on the other hand BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of the sequence of characters

The eccentric difference lies in reading different ways of taking input via the next() method that is justified in the below programs over a similar input set.

**Example 1:**

// Java Program to Illustrate Scanner Class

// Importing Scanner class from

// java.util package

import java.util.Scanner;

// Main class

class GFG {

// Main driver method

public static void main(String args[])

{

// Creating object of Scanner class to

// read input from keyboard

Scanner scn = new Scanner(System.in);

System.out.println("Enter an integer & a String");

// Using nextInt() to parse integer values

int a = scn.nextInt();

// Using nextLine() to parse string values

String b = scn.nextLine();

// Display name and age entered above

System.out.printf("You have entered:- " + a + " "

+ "and name as " + b);

}

}

Output:

Enter an integer & a String

10 John

You have entered:- 10 and name as John

Let us try the same using Buffer class and the same Input below as follows:

**Example 2:**

// Java Program to Illustrate BufferedReader Class

// Importing required class

import java.io.\*;

// Main class

class GFG {

// Main driver method

public static void main(String args[])

throws IOException

{

// Creating object of class inside main() method

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.println("Enter an integer");

// Taking integer input

int a = Integer.parseInt(br.readLine());

System.out.println("Enter a String");

String b = br.readLine();

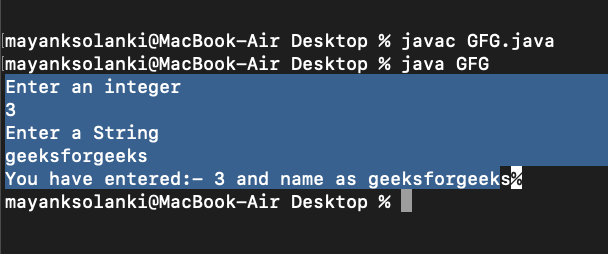
// Printing input entities above

System.out.printf("You have entered:- " + a+ " and name as " + b);

}

}

Output:



**Outputs explanation**: In Scanner class if we call **nextLine()** method after any one of the seven **nextXXX()** method then the **nextLine()** does not read values from console and cursor will not come into console it will skip that step. The **nextXXX()** methods are **nextInt(), nextFloat(), nextByte(), nextShort(), nextDouble(), nextLong(), next().**

In **BufferReader** class there is no such type of problem. This problem occurs only for the **Scanner class**, due to **nextXXX()** methods ignoring newline character and **nextLine()** only reads till the first newline character. If we use one more call of **nextLine()** method between **nextXXX() and nextLine(),** then this problem will not occur because **nextLine()** will consume the newline character.

* **Following are the Major Differences between Scanner and BufferedReader Class in Java**

1. BufferedReader is synchronous while Scanner is not. BufferedReader should be used if we are working with multiple threads.
2. BufferedReader has a significantly larger buffer memory than Scanner.
3. The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it’s more than enough.
4. BufferedReader is a bit faster as compared to Scanner because the Scanner does the parsing of input data and BufferedReader simply reads a sequence of characters.

**Ways to read input from console in Java**

In Java, there are **four different ways** for reading input from the user in the command line environment(console).

**1.Using Buffered Reader Class**

This is the Java classical method to take input, Introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line.

The input is buffered for efficient reading.

The wrapping code is hard to remember.

**Implementation:**

// Java program to demonstrate BufferedReader

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

public class Test {

public static void main(String[] args)

throws IOException

{

// Enter data using BufferReader

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

// Reading data using readLine

String name = reader.readLine();

// Printing the read line

System.out.println(name);

}

}

**Input:**

Geek

Output:

Auxiliary Space : O(1)

Geek

**Note:** To read other types, we use functions like Integer.parseInt(), Double.parseDouble(). To read multiple values, we use split().

**2. Using Scanner Class**

This is probably the most preferred method to take input. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however, it is also can be used to read input from the user in the command line.

* Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
* Regular expressions can be used to find tokens.
* The reading methods are not synchronized
* To see more differences, please see this article.

// Java program to demonstrate working of Scanner in Java

import java.util.Scanner;

class GetInputFromUser {

public static void main(String args[])

{

// Using Scanner for Getting Input from User

Scanner in = new Scanner(System.in);

String s = in.nextLine();

System.out.println("You entered string " + s);

int a = in.nextInt();

System.out.println("You entered integer " + a);

float b = in.nextFloat();

System.out.println("You entered float " + b);

}

}

**Input:**

GeeksforGeeks

12

3.4

**Output:**

You entered string GeeksforGeeks

You entered integer 12

You entered float 3.4

1. **Using Console Class**

It has been becoming a preferred way for reading user’s input from the command line. In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like System.out.printf()).

Advantages:

* Reading password without echoing the entered characters.
* Reading methods are synchronized.
* Format string syntax can be used.
* Does not work in non-interactive environment (such as in an IDE).

// Java program to demonstrate working of System.console()

// Note that this program does not work on IDEs as

// System.console() may require console

public class Sample {

public static void main(String[] args)

{

// Using Console to input data from user

String name = System.console().readLine();

System.out.println("You entered string " + name);

}

}

**Input:**

GeeksforGeeks

Output:

You entered string GeeksforGeeks

**4. Using Command line argument**

Most used user input for competitive coding. The command-line arguments are stored in the String format. The parseInt method of the Integer class converts string argument into Integer. Similarly, for float and others during execution. The usage of args[] comes into existence in this input form. The passing of information takes place during the program run. The command line is given to args[]. These programs have to be run on cmd.

Code:

**// Program to check for command line arguments**

class Hello {

public static void main(String[] args)

{

// check if length of args array is

// greater than 0

if (args.length > 0) {

System.out.println(

"The command line arguments are:");

// iterating the args array and printing

// the command line arguments

for (String val : args)

System.out.println(val);

}

else

System.out.println("No command line "

+ "arguments found.");

}

}

**Command Line Arguments**:

javac GFG1.java

java Main Hello World

**Output:**

The command line arguments are:

Hello

World

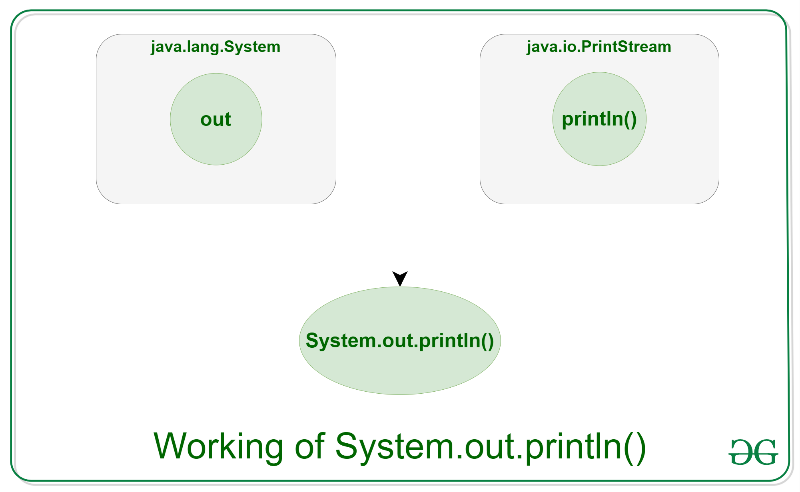
**System.out.println in Java**

Java **System.out.println()** is used to print an argument that is passed to it.

* Parts of System.out.println()

The statement can be broken into **3 parts** which can be understood separately:

* **System**: It is a final class defined in the java.lang package.
* **out:** This is an instance of PrintStream type, which is a public and static member field of the System class.
* **println():** As all instances of the PrintStream class have a public method println(), we can invoke the same on out as well. This is an upgraded version of print(). It prints any argument passed to it and adds a new line to the output. We can assume that System.out represents the Standard Output Stream.



**Syntax:**

System.out.println(parameter)

**Parameters:** The parameter might be anything that the user wishes to print on the output screen.

Example of Java System.out.println()

**Example 1:**

Below is the implementation of System.out.println :

// Java code to illustrate

// System.out.println();

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main(String[] args)

{

System.out.println("Welcome");

System.out.println("To");

System.out.println("GeeksforGeeks");

}

}

Output

Welcome

To

GeeksforGeeks

**Example 2:**

Below is the implementation of System.out.println :

// Java code to illustrate

// System.out.println();

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main(String[] args)

{

// Declaring variable

int num1 = 10, num2 = 20, sum;

// Printing the variables

System.out.print("The addition of ");

System.out.print(

num1 + " and " + num2 + " is: ");

// Printing the result after operation

System.out.println(num1 + num2);

}

}

**Output**

The addition of 10 and 20 is: 30

Just like System.out, Java provides us with two other standard or default input-output streams:

* **System.in:** This is the standard input stream that is used to read characters from the keyboard or any other standard input device. Example:
* **InputStreamReader inp = new InputStreamReader(System.in);**
* **System.err:** This is the standard error stream that is used to output all the error data that a program might throw, on a computer screen or any standard output device.

**Example:**

* System.err.print("Error");
* Overloads of println() method
* As we know, Method Overloading in Java allows different methods to have the same name, but different signatures or parameters where each signature can differ by the number of input parameters or type of input parameters or both. From the use of println() we observed that it is a single method of PrintStream class that allows the users to print various types of elements by accepting different type and number of parameters.

**For example:**

System.out.println(),

System.out.println(int),

System.out.println(double),

System.out.println(string),

System.out.println(character),

etc.

**PrintStream** has around 10 different overloads of println() method that are invoked based on the type of parameters passed by the user.

Example:

// Java code to illustrate method

// overloading in println()

import java.io.\*;

// Driver Class

class PrintLN {

// main function

public static void main(String[] args)

{

// Declaring different datatypes

int num = 10;

char ch = 'G';

String str = "GeeksforGeeks";

double d = 10.2;

float f = 13.5f;

boolean bool = true;

// Various overloads of println() method

System.out.println();

System.out.println(num);

System.out.println(ch);

System.out.println(str);

System.out.println(d);

System.out.println(f);

System.out.println(bool);

System.out.println("Hello");

}

}

Output

10

G

GeeksforGeeks

10.2

13.5

true

Hello

**Difference between System.out.print() and System.out.println()**

**System.out.print()**

This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here. This method must take atleast one parameter else it will throw an error.

**System.out.println()**

This method prints the text on the console and the cursor remains at the start of the next line at the console. The next printing takes place from the next line. This method may or may not take any parameter.

Example:

// Java code to illustrate difference

// between print() and println()

import java.io.\*;

// Driver Class

class Demo\_print {

// main function

public static void main(String[] args)

{

System.out.println("Using print()");

// using print()

// all are printed in the

// same line

System.out.print("GfG! ");

System.out.print("GfG! ");

System.out.print("GfG! ");

System.out.println();

System.out.println();

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

// using println()

// all are printed in the

// different line

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

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

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

}

}

Output:

**Using print()**

**GfG! GfG! GfG!**

**Using println()**

**GfG!**

**GfG!**

**GfG!**

* **Performance Analysis of System.out.println()**

**println()** is a method that helps display output on a console. This might be dependent on various factors that drives the performance of this method. The message passed using println() is passed to the server’s console where kernel time is required to execute the task. Kernel time refers to the CPU time. Since println() is a synchronized method, so when multiple threads are passed could lead to the low-performance issue. System.out.println() is a slow operation as it incurs heavy overhead on the machine compared to most IO operations. There is an alternative way of performing output operations by invoking PrintWriter or the BufferedWriter class. They are fast as compared to the println() of the PrintStream class.

* **Difference between print() and println() in Java**

**print():** print() method in Java is used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here. Various print() methods:

**void print(boolean b)** – Prints a boolean value. void print(char c) – Prints a character. void print(char[] s) – Prints an array of characters. void print(double d) – Prints a double-precision floating-point number. void print(float f) – Prints a floating-point number. void print(int i) – Prints an integer. void print(long l) – Prints a long integer. void print(Object obj) – Prints an object. void print(String s) – Prints a string.

**Example:**

import java.io.\*;

class GFG {

public static void main(String[] args)

{

// The cursor will remain

// just after the 1

System.out.print("GfG1");

// This will be printed

// just after the GfG2

System.out.print("GfG2");

}

}

Output:

GfG1GfG2

* **println():** println() method in Java is also used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the start of the next line at the console. The next printing takes place from next line. Various println() methods:
* **void println()** – Terminates the current line by writing the line separator string.
* **void println(boolean x)** – Prints a boolean and then terminate the line.
* **void println(char x)** – Prints a character and then terminate the line.
* **void println(char[] x)** – Prints an array of characters and then terminate the line.
* **void println(double x)** – Prints a double and then terminate the line.
* **void println(float x)** – Prints a float and then terminate the line. void println(int x) – Prints an integer and then terminate the line.
* **void println(long x)** – Prints a long and then terminate the line. void println(Object x) – Prints an Object and then terminate the line. void println(String x) – Prints a String and then terminate the line.

**Example:**

import java.io.\*;

class GFG {

public static void main(String[] args)

{

// The cursor will after GFG1

// will at the start

// of the next line

System.out.println("GfG1");

// This will be printed at the

// start of the next line

System.out.println("GfG2");

}

}

Output:

GfG1

GfG2

Related Articles:

**Difference between print() and println()**

* Input-Output in Java
* PrintStream println() method in Java with Examples
* Redirecting System.out.println() output to a file in Java

| **println()** | **print()** |
| --- | --- |
| It adds new line after the message gets displayed. | It does not add any new line. |
| It can work without arguments. | This method only works with argument, otherwise it is a syntax error. |

**Formatted Output in Java using printf()**

Sometimes in programming, it is essential to print the output in a given specified format. Most users are familiar with the printf function in C. Let us discuss how we can Formatting Output with printf() in Java in this article.

**Formatting Using Java Printf()**

printf() uses format specifiers for formatting. There are certain data types are mentioned below:

1. **For Number Formatting**
2. **Formatting Decimal Numbers**
3. **For Boolean Formatting**
4. **For String Formatting**
5. **For Char Formatting**
6. **For Date and Time Formatting**

**i). For Number Formatting**

The number itself includes Integer, Long, etc. The formatting Specifier used is %d.

Below is the implementation of the above method:

// Java Program to demonstrate

// Use of printf to

// Formatting Integer

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main (String[] args) {

int a=10000;

//System.out.printf("%.d%n",a);

System.out.printf("%,d%n",a);

}

}

Output

10,000

**ii). For Decimal Number Formatting**

Decimal Number Formatting can be done using print() and format specifier %f .

Below is the implementation of the above method:

// Java Programs to demonstrate

// Use of Printf() for decimal

// Number Formatting

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main(String[] args)

{

// declaring double

double a = 3.14159265359;

// Printing Double Value with

// different Formatting

System.out.printf("%f\n", a);

System.out.printf("%5.3f\n", a);

System.out.printf("%5.2f\n", a);

}

}

Output

3.141593

3.142

3.14

**iii). For Boolean Formatting**

Boolean Formatting can be done using **printf** and **( ‘%b’ or ‘%B’ )** depending upon the result needed.

**Below is the implementation of the above method:**

// Java Programs to demonstrate

// Use of Printf() for decimal

// Boolean Formatting

import java.io.\*;

// Driver Function

class GFG {

// main function

public static void main(String[] args)

{

int a = 10;

Boolean b = true, c = false;

Integer d = null;

// Fromatting Done using printf

System.out.printf("%b\n", a);

System.out.printf("%B\n", b);

System.out.printf("%b\n", c);

System.out.printf("%B\n", d);

}

}

Output

true

TRUE

false

FALSE

**iv). For Char Formatting**

Char Formatting is easy to understand as it need printf() and Charracter format specifier used are **‘%c’ and ‘%C’.**

Below is the implementation of the above method:

// Java Program to Formatt

//

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main(String[] args)

{

char c = 'g';

// Formatting Done

System.out.printf("%c\n", c);

// Converting into Uppercase

System.out.printf("%C\n", c);

}

}

Output

g

G

**v). For String Formatting**

String Formatting requires the knowledge of Strings and format specifier used ‘%s’ and ‘%S’.

Below is the implementation of the above method:

// Java Program to implement

// Printf() for String Formatting

import java.io.\*;

// Driver Class

class GFG {

// main function

public static void main(String[] args)

{

String str = "geeksforgeeks";

// Formatting from lowercase to

// Uppercase

System.out.printf("%s \n", str);

System.out.printf("%S \n", str);

str = "GFG";

// Vice-versa not possible

System.out.printf("%S \n", str);

System.out.printf("%s \n", str);

}

}

Output

geeksforgeeks

GEEKSFORGEEKS

GFG

GFG

**vi). For Date and Time Formatting**

Formatting of Date and Time is not as easy as the data-type used above. It uses more than simple format specifier knowledge can be observed in the example mentioned below.

Below is the implementation of the above method:

// Java Program to demonstrate use of

// printf() for formatting Date-time

import java.io.\*;

import java.util.\*;

// Driver Class

class GFG {

// main function

public static void main(String[] args)

{

Date time = new Date();

System.out.printf("Current Time: %tT\n", time);

// Another Method with all of them Hour

// minutes and seconds seperated

System.out.printf("Hours: %tH Minutes: %tM Seconds: %tS\n",

time,time, time);

// Another Method to print the time

// Followed by am/pm , time in milliseconds

// nanoseconds and time-zone offset

System.out.printf("%1$tH:%1$tM:%1$tS %1$tp %1$tL %1$tN %1$tz %n",

time);

}

}

Output

Current Time: 11:32:36

Hours: 11 Minutes: 32 Seconds: 36

11:32:36 am 198 198000000 +0000

**Note:** System.out.format() is equivalent to printf() and can also be used.

Other Methods for Formatting

**1. Formatting using DecimalFormat class**

DecimalFormat is used to format decimal numbers.

Below is the implementation of the above method:

// Java program to demonstrate working of DecimalFormat

import java.text.DecimalFormat;

// Driver Class

class JavaFormatter2 {

// main function

public static void main(String args[])

{

double num = 123.4567;

// prints only numeric part of a floating number

DecimalFormat ft = new DecimalFormat("####");

System.out.println("Without fraction part: num = "

+ ft.format(num));

// this will print it upto 2 decimal places

ft = new DecimalFormat("#.##");

System.out.println("Formatted to Give precision: num = "

+ ft.format(num));

// automatically appends zero to the rightmost part

// of decimal instead of #,we use digit 0

ft = new DecimalFormat("#.000000");

System.out.println("appended zeroes to right: num = "

+ ft.format(num));

// automatically appends zero to the leftmost of

// decimal number instead of #,we use digit 0

ft = new DecimalFormat("00000.00");

System.out.println("formatting Numeric part : num = "

+ ft.format(num));

// formatting money in dollars

double income = 23456.789;

ft = new DecimalFormat("$###,###.##");

System.out.println("your Formatted Dream Income : "

+ ft.format(income));

}

}

Output

Without fraction part: num = 123

Formatted to Give precision: num = 123.46

appended zeroes to right: num = 123.456700

formatting Numeric part : num = 00123.46

your Formatted Dream Income : $23,456.79

**2. Formatting dates and parsing using SimpleDateFormat class**

This class is present in java.text package.

Below is the implementation of the above method:

// Java program to demonstrate working of SimpleDateFormat

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Date;

// Driver Class

class Formatter3 {

// main function

public static void main(String args[])

throws ParseException

{

// Formatting as per given pattern in the argument

SimpleDateFormat ft = new SimpleDateFormat("dd-MM-yyyy");

String str = ft.format(new Date());

System.out.println("Formatted Date : " + str);

// parsing a given String

str = "02/18/1995";

ft = new SimpleDateFormat("MM/dd/yyyy");

Date date = ft.parse(str);

// this will print the date as per parsed string

System.out.println("Parsed Date : " + date);

}

}

**Output**

Formatted Date : 24-01-2022

Parsed Date : Sat Feb 18 00:00:00 UTC 1995

**Fast I/O in Java in Competitive Programming**

Using Java in competitive programming is not something many people would suggest just because of its slow input and output, and well indeed it is slow.

In this article, we have discussed some ways to get around the difficulty and change the verdict from TLE to (in most cases) AC.

Example: Input:

7 3

1

51

966369

7

9

999996

11

Output:

4

1. **Scanner Class** (easy, less typing, but not recommended very slow, refer this for reasons of slowness):

In most of the cases, we get TLE while using scanner class. It uses built-in nextInt(), nextLong(), nextDouble methods to read the desired object after initiating scanner object with the input stream(e.g. System.in). The following program many times gets time limit exceeded verdict and therefore not of much use.

// Working program using Scanner

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.util.Scanner;

public class Main {

public static void main(String[] args)

{

Scanner s = new Scanner(System.in);

int n = s.nextInt();

int k = s.nextInt();

int count = 0;

while (n-- > 0) {

int x = s.nextInt();

if (x % k == 0)

count++;

}

System.out.println(count);

}

}

Learn Data Structures & Algorithms with GeeksforGeeks

**2. BufferedReader** (fast, but not recommended as it requires a lot of typing):

The **Java.io.BufferedReader** class reads text from a character-input stream, buffering characters to provide for the efficient reading of characters, arrays, and lines. With this method, we will have to parse the value every time for the desired type. Reading multiple words from a single line adds to its complexity because of the use of Stringtokenizer and hence this is not recommended. These get accepted with a running time of approx 0.89 s.but still as you can see it requires a lot of typing altogether and therefore method 3 is recommended.

**// Working program using BufferedReader**

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.StringTokenizer;

public class Main {

public static void main(String[] args)

throws IOException

{

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

StringTokenizer st = new StringTokenizer(br.readLine());

int n = Integer.parseInt(st.nextToken());

int k = Integer.parseInt(st.nextToken());

int count = 0;

while (n-- > 0) {

int x = Integer.parseInt(br.readLine());

if (x % k == 0)

count++;

}

System.out.println(count);

}

}

Learn Data Structures & Algorithms with GeeksforGeeks

**3.Userdefined FastReader Class** (which uses bufferedReader and StringTokenizer):

This method uses the time advantage of BufferedReader and StringTokenizer and the advantage of user-defined methods for less typing and therefore a faster input altogether. These get accepted with a time of 1.23 s and this method is very much recommended as it is easy to remember and is fast enough to meet the needs of most of the question in competitive coding.

// Working program with FastReader

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.Scanner;

import java.util.StringTokenizer;

public class Main {

static class FastReader {

BufferedReader br;

StringTokenizer st;

public FastReader()

{

br = new BufferedReader(

new InputStreamReader(System.in));

}

String next()

{

while (st == null || !st.hasMoreElements()) {

try {

st = new StringTokenizer(br.readLine());

}

catch (IOException e) {

e.printStackTrace();

}

}

return st.nextToken();

}

int nextInt() { return Integer.parseInt(next()); }

long nextLong() { return Long.parseLong(next()); }

double nextDouble()

{

return Double.parseDouble(next());

}

String nextLine()

{

String str = "";

try {

if(st.hasMoreTokens()){

str = st.nextToken("\n");

}

else{

str = br.readLine();

}

}

catch (IOException e) {

e.printStackTrace();

}

return str;

}

}

public static void main(String[] args)

{

FastReader s = new FastReader();

int n = s.nextInt();

int k = s.nextInt();

int count = 0;

while (n-- > 0) {

int x = s.nextInt();

if (x % k == 0)

count++;

}

System.out.println(count);

}

}

**4.Using Reader Class:**

There is yet another fast way through the problem, I would say the fastest way but is not recommended since it requires very cumbersome methods in its implementation. It uses inputDataStream to read through the stream of data and uses read() method and nextInt() methods for taking inputs. This is by far the fastest ways of taking input but is difficult to remember and is cumbersome in its approach. Below is the sample program using this method. These get accepted with a surprising time of just 0.28 s. Although this is ultra-fast, it is clearly not an easy method to remember.

// Working program using Reader Class

import java.io.DataInputStream;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.Scanner;

import java.util.StringTokenizer;

public class Main {

static class Reader {

final private int BUFFER\_SIZE = 1 << 16;

private DataInputStream din;

private byte[] buffer;

private int bufferPointer, bytesRead;

public Reader()

{

din = new DataInputStream(System.in);

buffer = new byte[BUFFER\_SIZE];

bufferPointer = bytesRead = 0;

}

public Reader(String file\_name) throws IOException

{

din = new DataInputStream(

new FileInputStream(file\_name));

buffer = new byte[BUFFER\_SIZE];

bufferPointer = bytesRead = 0;

}

public String readLine() throws IOException

{

byte[] buf = new byte[64]; // line length

int cnt = 0, c;

while ((c = read()) != -1) {

if (c == '\n') {

if (cnt != 0) {

break;

}

else {

continue;

}

}

buf[cnt++] = (byte)c;

}

return new String(buf, 0, cnt);

}

public int nextInt() throws IOException

{

int ret = 0;

byte c = read();

while (c <= ' ') {

c = read();

}

boolean neg = (c == '-');

if (neg)

c = read();

do {

ret = ret \* 10 + c - '0';

} while ((c = read()) >= '0' && c <= '9');

if (neg)

return -ret;

return ret;

}

public long nextLong() throws IOException

{

long ret = 0;

byte c = read();

while (c <= ' ')

c = read();

boolean neg = (c == '-');

if (neg)

c = read();

do {

ret = ret \* 10 + c - '0';

} while ((c = read()) >= '0' && c <= '9');

if (neg)

return -ret;

return ret;

}

public double nextDouble() throws IOException

{

double ret = 0, div = 1;

byte c = read();

while (c <= ' ')

c = read();

boolean neg = (c == '-');

if (neg)

c = read();

do {

ret = ret \* 10 + c - '0';

} while ((c = read()) >= '0' && c <= '9');

if (c == '.') {

while ((c = read()) >= '0' && c <= '9') {

ret += (c - '0') / (div \*= 10);

}

}

if (neg)

return -ret;

return ret;

}

private void fillBuffer() throws IOException

{

bytesRead = din.read(buffer, bufferPointer = 0,

BUFFER\_SIZE);

if (bytesRead == -1)

buffer[0] = -1;

}

private byte read() throws IOException

{

if (bufferPointer == bytesRead)

fillBuffer();

return buffer[bufferPointer++];

}

public void close() throws IOException

{

if (din == null)

return;

din.close();

}

}

public static void main(String[] args)

throws IOException

{

Reader s = new Reader();

int n = s.nextInt();

int k = s.nextInt();

int count = 0;

while (n-- > 0) {

int x = s.nextInt();

if (x % k == 0)

count++;

}

System.out.println(count);

}

}

**Efficiently Reading Input For Competitive Programming using Java 8**

As we all know, while solving any CP problems, the very first step is collecting input or reading input. A common mistake we all make is spending too much time on writing code and compile-time as well. In Java, it is recommended to use BufferedReader over Scanner to accept input from the user. Why? It is discussed in one of our previous articles here. (Also, the issues associated with the java.util.Scanner is available) Yet for a better understanding, we will go through both the implementations in this article.

**Ways of Reading Inputs**

1. Using Scanner class
2. Using BufferedReader class
3. Using BufferedReader class with help of streams (More optimized)

Now let us discuss ways of reading individually to depth by providing clean java programs and perceiving the output generated from the custom input.

The java.util.Scanner class provides inbuilt methods to read primitive data from the console along with the lines of text. In the below code snippet let’s understand how it is done.

**Example**

// Java Program Illustrating Reading Input

// Using Scanner class

// Importing Arrays and Scanner class

// from java.util package

import java.util.Arrays;

import java.util.Scanner;

// Main class

class GFG {

// Main driver method

public static void main(String[] args)

{

// Creating object of Scanner class

Scanner scan = new Scanner(System.in);

// Basic Input Reading

int a = scan.nextInt();

float b = scan.nextFloat();

System.out.println("Integer value: " + a);

System.out.println("Float value: " + b);

// Space Separated Input Reading

int[] arr = new int[5];

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

arr[i] = scan.nextInt();

}

System.out.println(Arrays.toString(arr));

}

}

**Way 1: Simple Scanner Input Reading**

The java.util.Scanner class provides inbuilt methods to read primitive data from the console along with the lines of text. In the below code snippet let’s understand how it is done.

Example

// Java Program Illustrating Reading Input

// Using Scanner class

// Importing Arrays and Scanner class

// from java.util package

import java.util.Arrays;

import java.util.Scanner;

// Main class

class GFG {

// Main driver method

public static void main(String[] args)

{

// Creating object of Scanner class

Scanner scan = new Scanner(System.in);

// Basic Input Reading

int a = scan.nextInt();

float b = scan.nextFloat();

System.out.println("Integer value: " + a);

System.out.println("Float value: " + b);

// Space Separated Input Reading

int[] arr = new int[5];

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

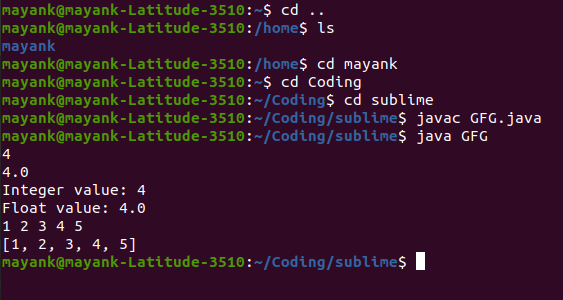
arr[i] = scan.nextInt();

}

System.out.println(Arrays.toString(arr));

}

}



From the above Linux shell output we can conclude that input is given as is follows:

4

5.6

1 2 3 4 5

The output generated is as follows:

Integer value: 4

Float value: 5.6

[1, 2, 3, 4, 5]

The above example illustrates the most common approach used by the majority of programmers while solving competitive programming problems. But what if we can enhance our code a bit to make it faster and reliable?

**Method 2: Simple BufferedReader Input Reading**

java.io.BufferedReader class does not provide any method to read primitive data inputs. Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of the sequence of characters. Although it throws a checked exception known as IOException. Let us see how to handle that exception and read input from the user. Consider custom input as below as follows:

Input:

4

5.6

1 2 3 4 5

**Example**

// Java Program Illustrating Reading Input

// Using

// Importing required classes

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.Arrays;

// Main class

class GFG {

// Main driver method

public static void main(String[] args)

throws IOException

{

// Reading input via BufferedReader class

BufferedReader br = new BufferedReader(

new InputStreamReader(System.in));

// Basic Input Reading

int a = Integer.parseInt(br.readLine());

float b = Float.parseFloat(br.readLine());

// Print above input values in console

System.out.println("Integer value: " + a);

System.out.println("Float value: " + b);

// Space Separated Input Reading

int[] arr = new int[5];

String[] strArr = br.readLine().split(" ");

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

arr[i] = Integer.parseInt(strArr[i]);

}

// Printing the elements in array

// using toString() method

System.out.println(Arrays.toString(arr));

}

}

Learn Data Structures & Algorithms with GeeksforGeeks

Output:

Integer value: 4

Float value: 5.6

[1, 2, 3, 4, 5]

The above example illustrates another common approach used to read the data while solving competitive programming problems. So is this enough? What if we can enhance it even more? Yes. It is possible. Stay tuned.

**Method 3: Enhanced way for reading separated data using BufferedReader via Streams**

In the previous examples, we have seen while reading space-separated data we stored it first in a String array, and then we iterated over elements and then used java typecasting to convert it to the required data type. How about a single line of code making this possible? Yes. Java 8’s stream library provides a variety of functions to make it easy and optimized. Consider custom input as below as follows:

Input:

34 55 78 43 78 43 22

94 67 96 32 79 6 33

Example

// Java Program to Read Separated Data

// Using BufferedReader class voa enhanced for loopd

// Importing required classes

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

// Main class

class GFG {

// Main driver method

public static void main(String[] args)

throws IOException

{

// Reading input separated by space

BufferedReader br = new BufferedReader(

new InputStreamReader(System.in));

// Storing in array

int[] arr = Stream.of(br.readLine().split(" "))

.mapToInt(Integer::parseInt)

.toArray();

System.out.println(Arrays.toString(arr));

// Using streams concepts to parse map to integer

// later on collecting via Collectors via toList()

// method and storing it an integer list

List<Integer> arrayList

= Stream.of(br.readLine().split(" "))

.mapToInt(Integer::parseInt)

.boxed()

.collect(Collectors.toList());

// Print the above List as created

System.out.println(arrayList);

}

}

Learn Data Structures & Algorithms with GeeksforGeeks

Output:

[34, 55, 78, 43, 78, 43, 22]

[94, 67, 96, 32, 79, 6, 33]

The above example illustrates how we can read separated input and store it into the required data structure using a single line of code. Using java8 there might be a possibility that programmers are comfortable with List collection. That’s why it is covered. Now, let us understand code word by word.

**Storing into int array:**

1**. java.util.stream.Stream.of()** – Creates stream of string array passed

2**. br.readLine().split(” “)** – Converts input string into string array based on separator value. (Blank Space – ” ” in example)

3**. mapToInt(Integer::parseInt)** – Converts String element into the required data type using suitable mapper function (Integer’s parseInt() in example)

4. **toArray()** – converts the stream of int elements into an array

* **Storing into the List Collection:**

**1. java.util.stream.Stream.of()** – Creates stream of string array passed

**2. br.readLine().split(” “)** – Converts input string into string array based on separator value. (Blank Space – ” ” in example)

**3. mapToInt(Integer::parseInt)** – Converts String element into the required data type using suitable mapper function (Integer’s parseInt() in example)

**4. boxed()** – boxes the stream to Integer elements

**5. collect(Collectors.toList())** – creates a collection of Integer elements and converts it to the java.util.List Collection.