# JAVA

Java is a Programming Language developed by SUN Microsystems initially termed as OAK.

NEED : The need to develop a portable and secure programming language which will be platform independent.

LOGIC : Compile once run everywhere.

ISSUES with C, C++ : They were platform dependent and need separate compilers for different OS and architecture.

HOW JAVA WORKS ?

Java uses both compiler and interpreter to succeed its motto.

Java code (.java) 🡪 uses compiler (javac) 🡪byte code (.class) (which targets java interpreter and not any machine. Intermediate language which stores all java keywords in sign language which occupies 1byte memory in RAM) 🡪 JVM (Java virtual machine) an interpreter which converts byte code into machine language which is specific to OS.

Here we have used an additional layer called JVM which sits on the machine OS and byte code can be executed only by JVM. Developing a platform specific interpreter was cheap compared to a platform specific compiler. So SUN Microsystems as choosen this option.

Terminology –

JDK – Java development kit

**The JDK allows developers to create Java programs that can be executed and run by the JVM and JRE**

JRE – Java runtime environment

**The JRE is the on-disk part of Java that creates the JVM.**

JVM – Java virtual machine

**The JVM is the Java platform component that executes programs.**

We often confuse the Java Development Kit and the Java Runtime Environment. The distinction is that the **JDK is a package of tools for developing Java-based software, whereas the JRE is a package of tools for running Java code.**

The JRE can be used as a standalone component to simply run Java programs, but it's also part of the JDK. The JDK requires a JRE because running Java programs is part of developing them.

**Class String Handling in JAVA –**

Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable. There are many differences between StringBuffer and StringBuilder. The StringBuilder class is introduced since JDK 1.5.

Most inefficient way is to use String as the memory is not deallocated for a modified string, i.e., once a string is modified a new instance is created rather than modifying the same instance.

|  |  |
| --- | --- |
| **StringBuffer** | **StringBuilder** |
| StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

### StringBuider vs StringBuffer

In Summary, following are notable difference between StringBuffer and StringBuilder in Java  
  
1) StringBuilder is non synchronized version of StringBuffer class. Methods in StringBuilder e.g. all overloaded version of append() method is not synchronized.  
  
2) StringBuilder is definitely faster than StringBuffer because of no overhead of acquiring and releasing locks associated with synchronized methods.  
  
3) StringBuffer is thread-safe and StringBuilder is not. You can not share Instances of StringBuilder class between multiple threads. If such synchronization is required then it is better to use StringBuffer class.  
  
4) StringBuffer is old class, its there in JDK from very first release, while StringBuilder is relatively newer class, introduced much later in release of JDK 1.5  
  
5) Another interesting fact to know about both of this class is that, when you do String concatenation using + operator, Java internally convert that call to corresponding StringBuilder append() method class. For example "one" + "two" + "three" will be converted to new StringBuilder().append("one").append("two").append("three"). Only problem is that it initialize StringBuilder with default capacity, which means expensive array copy operation, when StringBuilder get resized.

**Abstract Classes and Interfaces –**

Abstract Class is more fundamental and related class to the extending/subclass. The class which extends the abstract class is related to the abstract class (example – Machine (abstract class) relates more to Classes like Car and Camera).

Interfaces are not classes, they contain only method definitions and the extending classes need to implement these methods. Interfaces are not fundamentally related to the extending class. (example – Interface INFO is implemented by Machine, Person, World. These classes have common functionality like showing their info but the classes are not related to the interface fundamentally)

Based on the above 2 statements, a decision should be made on whether to create a Interface or Abstract.

Differences between Abstract and Interface –

|  |  |
| --- | --- |
| **Interface** | **Abstract** |
| Not a class, so cannot instantiate an interface. You can create a reference to an interface using anonymous inner class but not create an object by instantiating. | Same as interface. But abstract class is a class though. |
| Only method definitions allowed | Can have both method definitions and implementations |
| A subclass can implement multiple interfaces as they are not fundamentally related. | A subclass can extend only one Abstract class as they are fundamentally related. Also as abstract class is a class Java does not support multiple inheritance. |
| Implementing an interface consumes very little CPU, because it's not a class, just a bunch of names, and therefore there isn't any expensive look-up to do. It's great when it matters, such as in embedded devices. | Abstract classes, unlike interfaces, are classes. They are more expensive to use, because there is a look-up to do when you inherit from them. |
| Interface has only static and final variables. | Abstract class can have final, non-final, static and non-static variables. |
| Members of a Java interface are public by default. | A Java abstract class can have class members like private, protected, etc. |

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

**Buffered Reader vs Scanner**

BufferedRedaer can only read String but Scanner can read both String and other data types like int, float, long, double, float etc. This functional difference drives several other differences in their usage.