Java

 Java is a multi-platform, object-oriented, and network-centric language that can be used as a platform in itself. It is a fast, secure, reliable programming language for coding everything from mobile apps and enterprise software to big data applications and server-side technologies.

## **How does Java work?**

All programming languages are a means to communicate with machines. Machine hardware only responds to electronic communication. High-level programming languages like Java act as a bridge between human language and hardware language. To use Java, a developer needs to understand two things:

### 1. Java language and APIs

This is the front-end communication between the developer and the Java platform.

### 2. Java Virtual Machine

This is the back-end communication between the Java platform and the underlying hardware. Let’s look at each of these in detail below.

When natural programming languages were first developed, they fell into two broad categories, depending on how they communicated with the underlying hardware.

1. **Compilers:**The complete program is written in natural English-like syntax with compilers, and the language then compiles (or translates) the entire code into machine code. The compiled code is then run on the hardware.
2. **Interpreters:**With interpreters, every high-level code statement is interpreted into machine code on the fly.  Written statements are run immediately by the hardware before looking at the next statement.

## **The Java runtime environment**

The Java program was the first language to combine both methods above using a Java Virtual Machine (JVM). The Java code compiler is called the Java Virtual Machine. Any Java file is first compiled into bytecode. Java bytecode can only run in the JVM. The JVM then interprets the bytecode to run it on the underlying hardware platform. So if the application is running on a Windows machine, the JVM will interpret it for Windows. But if it is running on an open-source platform like Linux, the JVM will interpret it for Linux.

**JDK, JRE and JVM**

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)**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) 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 contains:**

* Java Runtime Environment (JRE),
* An interpreter/loader (Java),
* A compiler (javac),
* An archiver (jar) and many more.

### 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.

### 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**

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.

### Role of the JRE in Java programming language

The JRE combines the Java code that you create by using the JDK with additional built-in code called libraries. It then creates a JVM instance, or local copy, that finally runs the Java programs. JVMs are available for multiple operating systems, and the JRE generates a single copy of your Java code that runs on all types of JVMs. In this way, the JRE facilitates platform independence for Java applications. You can write them once and run them anywhere.

## **How does the JRE work?**

The Java Runtime Environment (JRE) runs on top of the operating system, providing additional Java-specific resources. The Java Development Kit (JDK) and JRE interact to create a sustainable runtime environment that runs Java program files on any machine. The JRE uses three core components to work.

### ClassLoader

Java class libraries contain collections of pre-written code that you can call as needed. They simplify the job of Java developers by providing built-in methods for common and non-trivial tasks such as taking input from users, displaying output to users, and more. All Java programs reference several class libraries. The Java ClassLoader dynamically loads all class files necessary into the Java Virtual Machine (JVM) on demand.

### Bytecode verifier

The JDK has a compiler that converts the English-like code you write into a machine language version called Java bytecode. They bytecode verifier in the JRE checks the format and accuracy of the Java code before loading it into the JVM. For example, if the code violates system integrity or access rights, the JRE will not load the class file.

### Interpreter

After the bytecode successfully loads, the Java interpreter creates the JVM instance that runs the Java program on the underlying machine.

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.

**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](https://www.geeksforgeeks.org/object-class-in-java/) class.

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

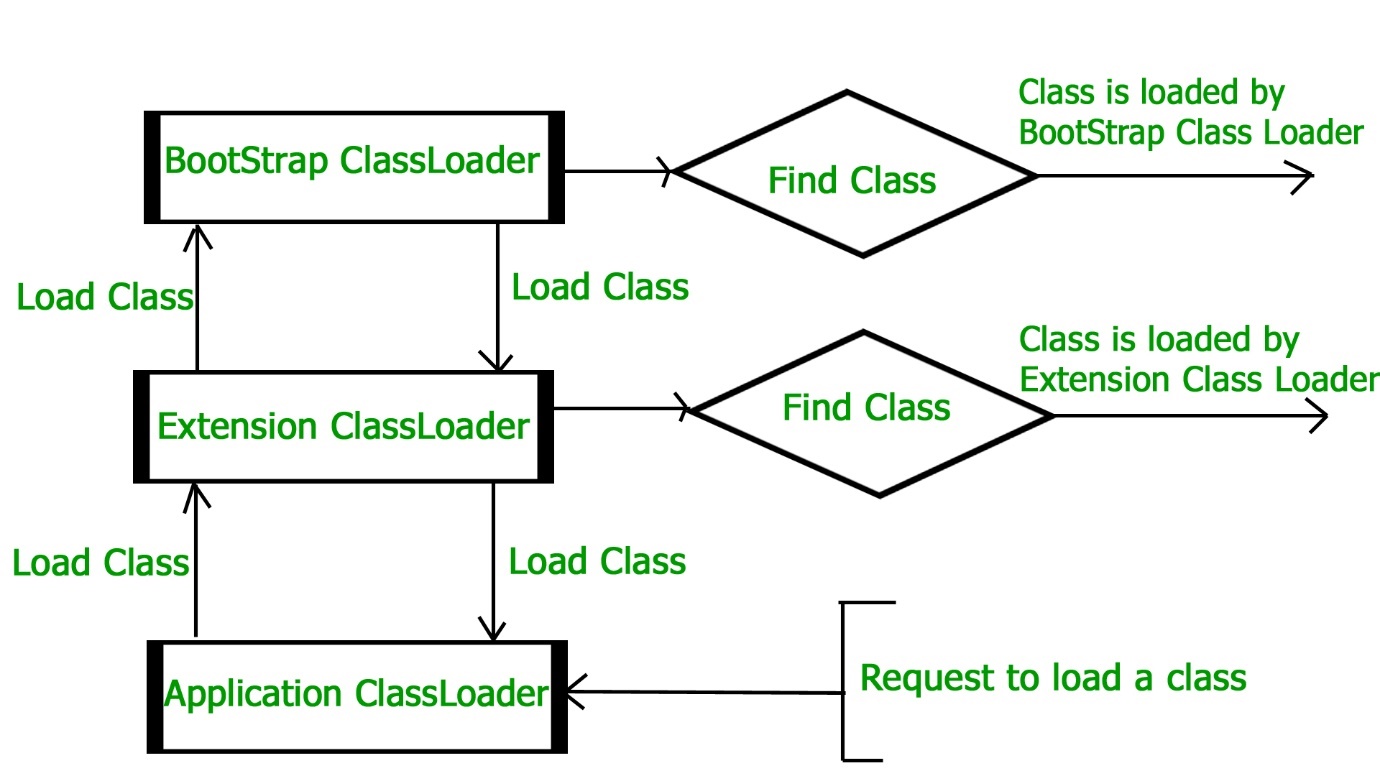
* *Verification*: It ensures the correctness of the *.class* file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError*. This activity is done by the component ByteCodeVerifier. Once this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

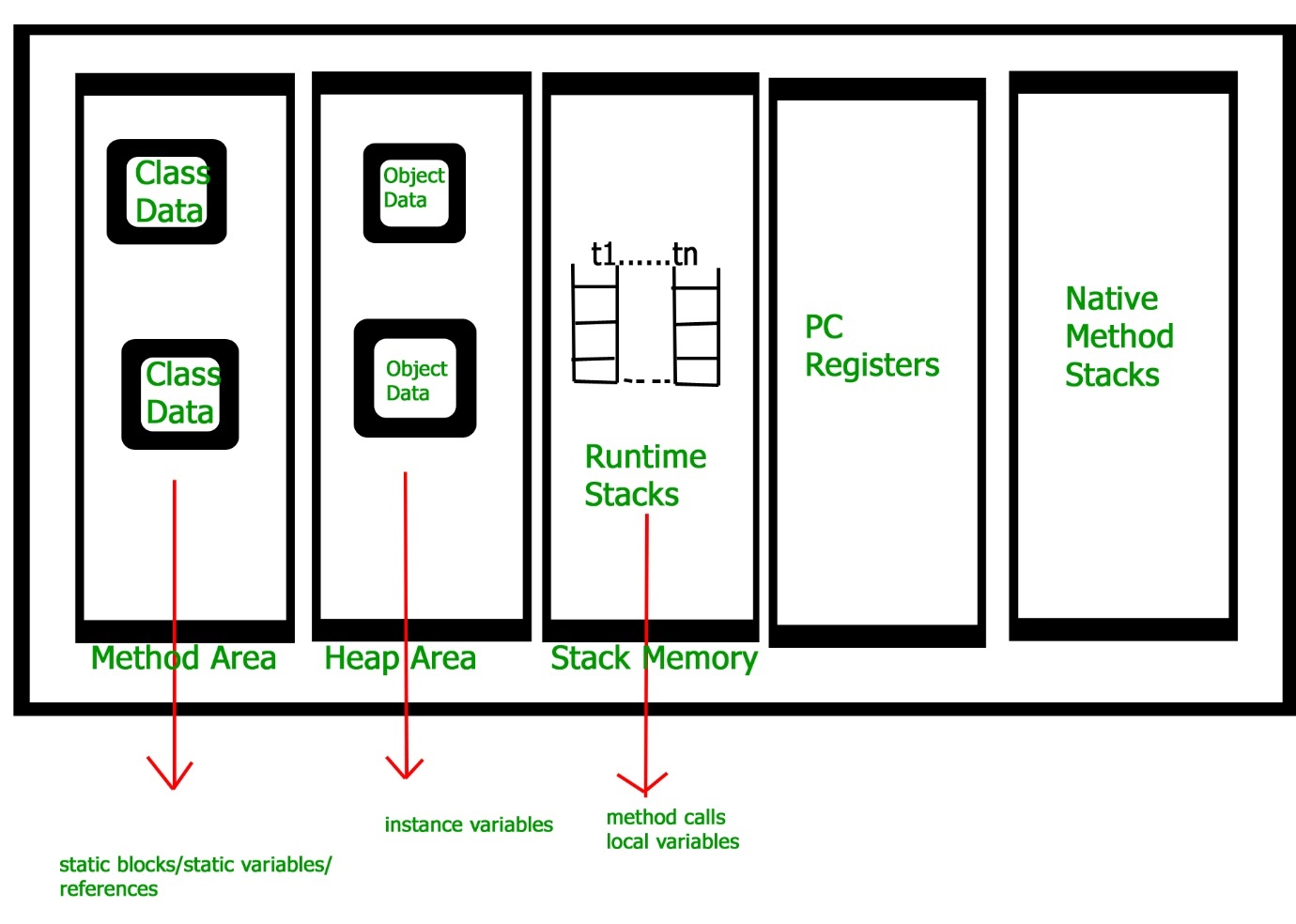
* *Bootstrap class loader*: Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in the “*JAVA\_HOME/jre/lib”* directory. This path is popularly known as the bootstrap path. It is implemented in native languages like C,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

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**

1. **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. From java 8, static variables are now stored in Heap area.
2. **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.
3. **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.
4. **PC Registers:** Store address of current execution instruction of a thread. Obviously, each thread has separate PC Registers.
5. **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](https://www.geeksforgeeks.org/garbage-collection-java/).

**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.

How Java is Platform independent?

Java is platform-independent because it uses a virtual machine. The Java programming language and all APIs are compiled into bytecodes. Bytecodes are effectively platform-independent. The virtual machine takes care of the differences between the bytecodes for the different platforms. The run-time requirements for Java are therefore very small. The Java virtual machine takes care of all hardware-related issues so that no code has to be compiled for different hardware.

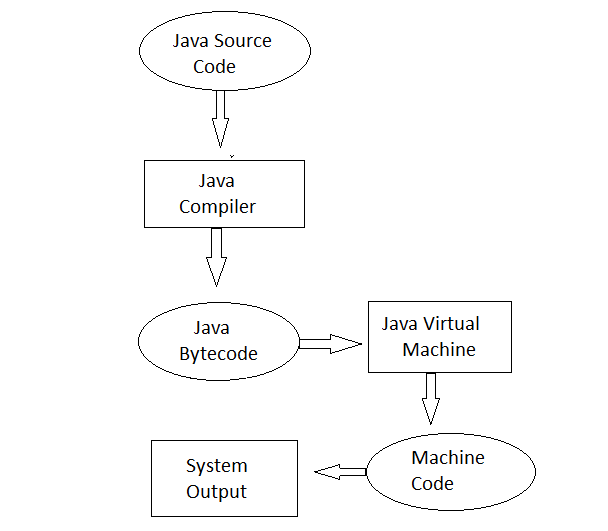
Whenever we enter a statement “javac filename.java” or compile a code in java, javac compiles the code. And it creates an intermediate code called Byte Code.

Bytecode adds this important feature in Java of being independent/ portable as this bytecode is independent of the platforms. This dependency eases the tasks as it can be used in various environments with lesser planning and translation.

This is where java makes difference between all different programming languages. It creates a .class file, which is considered as byte code. Whereas languages like c/c++ will create a natively executable code when they are compiled and thus makes them platform dependent.

Hold on, this is too early for celebration. This byte code is not executable. We need a translator for executing this byte code. And JVM does this job. Generally, JVM resides in the main memory of our computer. Java Virtual Machine acts as an interpreter and then executes the byte code generated by javac.

And now we are done, our code gets executed successfully.



So how does java going to manage it all alone?

And the answer would be, it’s because of the JVM. The byte code generated by source code compilation would run in any operating system, but the JVM present in a machine differs for each operating system. And this is how java is considered a platform-independent programming language.

1. Bootstrap Classloader- Loads the libraries of Java.
2. Extension Classloader- Load the codes in the directories of extensions or any other directory commanded by the system property.
3. Application Classloader- Loads code found on java.class.path eventually mapping these java.class.path to CLASSPATH environment variable.

Should you consider using Java as an embedded programming language?

Interest in the Java language continues unabated. It is clearly the most talked-about language of the year. What may be surprising is that it is being touted as applicable to embedded systems development. In this article we look at when it is appropriate to write embedded software in Java and when it's not. We present the facts behind the Java hype as we describe the language and the development tools that are currently available. We also discuss how these tools fit into a typical embedded development scenario and suggest improvements to the tools that will make them better suited to the needs of embedded programmers.

## Why Java?

Although platform independence has been hailed as Java's greatest strength, it is equally important to note that it is easier to produce bug-free software in Java than in C or C++. Java was designed from the ground up to produce code that is simpler to write and easier to maintain. Though they based their language on the syntax of C, the developers of Java eliminated many of that language's most troublesome features. These features sometimes make C hard to understand and maintain, and they frequently lead to undetected programming errors. Here are just a few of the improvements:

* All of Java's primitive data types have a fixed size. For example, an int is always 32-bits in Java
* Automatic bounds-checking prevents the programmer from writing or reading past the end of an array
* All test conditions must return either true or false. Common mistakes, such as if (x = 3), are detected at compile-time, thus eliminating one entire set of bugs
* Built-in support for strings and string manipulation allows statements like "Hello, " + "world!"

In addition, Java is an object-oriented language similar to C++. This forces software developers to structure their data and functions into logical units called classes. Encapsulation, polymorphism, and inheritance are all available and are used extensively in the built-in class libraries. Java simplifies inheritance by eliminating multiple inheritance and replacing it with interfaces. It also adds many new features that are not available in C++, most notably:

* Automatic garbage collection simplifies dynamic memory management and eliminates memory leaks
* A built-in threads library makes applications written in Java more portable by providing a consistent thread and synchronization interface across all operating systems
* An integrated exception mechanism organizes software exceptions into a logical class hierarchy and does not allow programmers to ignore them

Unfortunately, it is not possible to do everything in Java that you may be accustomed to doing in C. In particular, Java does not allow manipulation of pointers (which are replaced by references). However, much as assembly code can be called from other languages, it is possible to call C/C++ and assembly functions from Java. This technique is known as "native methods". Native methods allow device drivers and other software that manipulates memory or hardware registers directly to be written in another language, without forsaking the benefits of encapsulation.

## Java Development Tools

Before we talk about using Java in embedded systems, it is helpful to have a basic understanding of the Java development paradigm. The relationship between the various Java development tools is illustrated in Figure 1 and described in the following paragraphs.

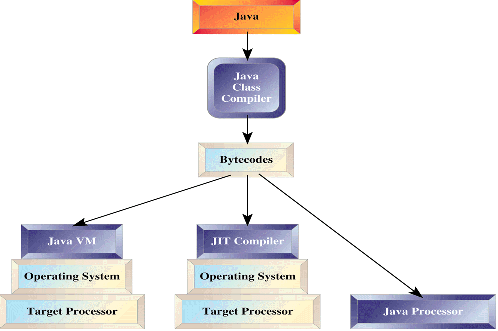


Figure 1. Java development tools

In Java, the compiler is called a class compiler. A class compiler compiles high-level Java code into a set of assembly-like instructions known as Java bytecodes. The bytecodes are the machine language of Java's "virtual machine." The virtual machine idea is central to Java's platform independence. A virtual machine can be any platform-hardware or software-that is capable of understanding and executing Java bytecodes. There are currently three such platforms: Java Virtual Machines, Just-in-Time Compilers, and Java Processors.

The Java Virtual Machine (VM) is a piece of software that translates Java bytecodes to the processor's native opcodes as they are executed. There are many implementations of the Java VM available, but all of them support the same set of bytecodes defined in the Java Virtual Machine Specification. Versions of the Java VM are available for many common hardware platforms. In an embedded system, the Java VM is essentially a set of threads that execute on top of a real-time operating system (RTOS).

Interpreted languages are portable, but slow. Just-in-Time (JIT) compilers were invented to speed the execution of Java programs by speeding up the language translation process. A JIT compiler is a drop-in replacement for the Java VM. The only difference is that it keeps a copy of all previously translated code for potential reuse. That way, the same section of code never has to be reinterpreted. Only new bytecodes that have not been previously executed need to be translated. A typical program with loops and repeated function calls should execute about 10 times faster under a JIT compiler. That makes the speed of programs written in Java comparable to those written in C++.

To speed the execution of Java bytecodes even further, Sun Microsystems has announced a line of Java processors. These are hardware realizations of the Java virtual machine. In other words, each Java processor is a microprocessor with a set of opcodes that are identical to the Java bytecode standard. To eliminate the need for native methods, these chips will have a set of extended bytecodes for accessing memory directly. It is reasonable to expect that these so-called JavaChips will execute Java bytecodes as quickly as any processor can execute its own assembly language. However, since the programs themselves will be written in the high-level Java, it is more realistic to expect performance comparable to C.

## Embeded Java Toolkit

The ideal embedded development kit for Java would integrate the above tools with existing cross-compilers, assemblers, debuggers, and real-time operating systems. The parts of the software written in Java could be coded and tested on the host platform, even before the target hardware becomes available. In addition, Ahead-of-Time Compilers would be available to further speed the execution of Java programs.

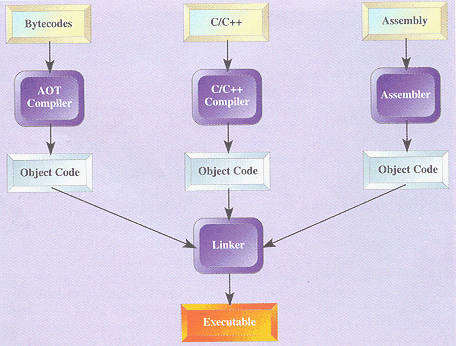
**

Figure 2. Using Java without a virtual machine

An Ahead-of-Time (AOT) compiler is a tool for converting Java bytecodes into the native object code of the target processor. With an AOT compiler, it would be possible to write software in Java but eliminate the run-time overhead of the Java VM or JIT compiler. The virtual machine would in this case be replaced by a smaller piece of software responsible only for garbage collection and exception handling-possibly provided with the RTOS. Java bytecodes that have been compiled with an AOT compiler could be linked to C/C++ and assembly programs using existing tools. A development scenario involving all three languages is shown in Figure 2.

Given the current set of bytecodes, it is unlikely that an entire embedded software project could be written in Java. For one thing, Java has no means of accessing memory or hardware registers directly (the downside of having no pointers). Unless bytecode extensions are introduced or JavaChips somehow supplant traditional embedded processors, there will always need to be device drivers and other pieces of supporting software written in C and assembly.

In a mixed language environment, developers will need an integrated development environment that understands Java's native methods. Ideally, such a tool would handle the details of mixing Java and other languages, support integrated version control and other niceties, and include a powerful multilingual debugger.

The debugger is perhaps the most critical feature-as it is in any development environment. It must be able to switch easily between Java, C/C++, and assembly language modules. In addition, it should be able to seamlessly communicate with: (1) a Java virtual machine running on the development workstation (for simulation), (2) a virtual machine running on the target (for debugging), or (3) an emulator installed on the target hardware (for real-time tracing). All three of these configurations are illustrated in Figure 3.

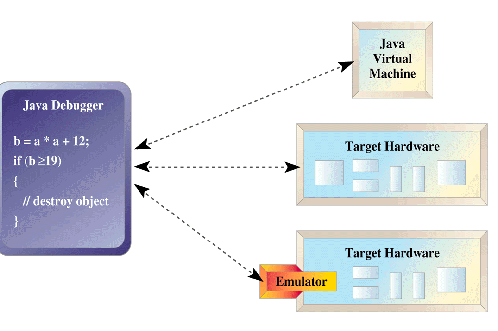


Figure 3. The ideal Java debugging environment

## What Tools are Available?

Although many of the individual components are available today, no single package fills all of the above requirements. JavaChips and AOT compilers will not be ready for some time, and JIT compilers and remote debuggers are not yet available for embedded systems. However, many of the other pieces are beginning to come together.

In particular, Java class compilers are widely available from multiple vendors. They are inexpensive (frequently less than $100 per seat) and can be run on PC, Mac, and UNIX workstations. These same class compilers are used to develop commercial software and Web applets. As a result, the development environments are more sophisticated than most of the tools embedded developers are currently using.

Implementations of the Java VM are available for several real-time operating systems, including Integrated System's pSOS. In addition, Java VM's are under development at Wind River Systems (VxWorks) and Microtec (VRTX). These are three of the most prominent RTOS vendors, so it is reasonable to expect other vendors will follow suit.

Unfortunately, the development of JIT compilers for embedded systems is lagging behind. Although they may be eventually made obsolete by AOT compilers, it would be nice to have them now. We are pleased to note that there is at least some movement toward the introduction of AOT compilers. MetaWare is in the process of developing the first such compiler for 32-bit x86 processors. We feel that the development of AOT compilers for other common embedded processors is crucial to Java's success as an embedded language.

In the area of Java debuggers, there is also good news. A Java Debugger API has been established by Sun and many of the Java development suites include remote debuggers. The current generation of debuggers can communicate over TCP/IP with any virtual machine that supports the debugger API. However, they cannot yet communicate with an embedded target over a serial interface.

## Does Java Make Sense for Embedded Systems?

Embedded developers usually have a different set of criterion for evaluating a language than other software developers. Even if Java is the best language for developers of commercial applications and Web applets, it may still not be appropriate as an embedded language. Among the most important requirements for embedded languages are deterministic behavior, small memory footprint, and efficient execution.

The key issue in real-time systems is deterministic behavior. Unfortunately, the current generation of garbage collectors is inherently non-deterministic. Moreover, garbage collection is an integral part of the Java language. Any variables that are not primitive types are objects. Because garbage collection cannot be eliminated from the language, several groups are working to create deterministic garbage collectors. Developers of real-time systems will not want to use Java until such alternatives become available.

Unfortunately, Java's current memory requirements are an order of magnitude too large for many embedded systems. In fact, systems based on typical 8-bit processors may not even have sufficient address space. The Java VM and core libraries alone require about 200K of code space. This requirement is in addition to those of the underlying operating system, the application itself, and a rather large heap for Java objects (the larger the heap, the more deterministic the garbage collection becomes). For the foreseeable future, it probably only makes sense to use Java in systems based on 16 or 32-bit processors with at least 1MB of RAM.

Java is not the most efficient language either. For now, embedded developers wanting to use Java must make do with a Java VM. Unfortunately, that means slow execution-sometimes less than 10% as fast as a similar program written in C. In the long run, performance will become less of an issue as embedded JIT compilers, AOT compilers, and JavaChips begin to appear.

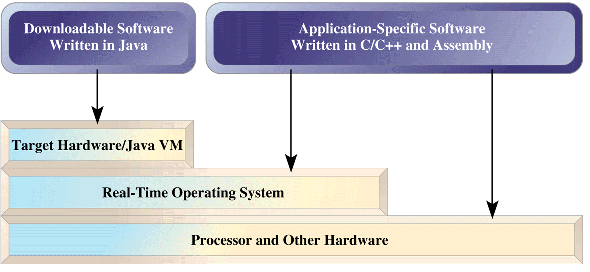


Figure 4. Software designer's view of a set-top box

Despite these deficiencies, however, there are some situations in which Java may already be appropriate. In fact, any sort of embedded device that connects to a network (or the Internet) is a good candidate for Java. Consider a set-top box. In such a device, most of the software would still be written in C/C++ and assembly. That software would control the basic operation of the device: receiving data from the satellite or cable network, decoding the enclosed video and audio streams, and sending images and sounds to the television.

With a Java VM in the set-top box, new functionality could be added to the device after it is manufactured. For example, Java programs could be downloaded over the satellite or cable network while browsing the Internet or during an interactive TV session. These programs would be downloaded over the network in bytecode form. Once received, they could be executed by the Java VM, as shown in Figure 4.

In this scenario, the application-specific software would be developed using current cross-compilers, assemblers, debuggers, and other tools. The RTOS and Java VM are commercial software products that would be linked with the application-specific software and stored in ROM. At the time of manufacture, the Java VM would not have any programs to execute; it would just wait idly for Java bytecodes to arrive over the network. The Java bytecodes would be created in a separate development environment, using only a Java class compiler. Once ready, the Java bytecodes could be stored on a server, where they would be available to any set-top box on the network.

Spring

The Spring Framework (Spring) is an [open-source](https://www.techtarget.com/whatis/definition/open-source) [application](https://www.techtarget.com/searchsoftwarequality/definition/application) [framework](https://www.techtarget.com/whatis/definition/framework) that provides [infrastructure](https://www.techtarget.com/searchdatacenter/definition/infrastructure) support for developing [Java](https://www.theserverside.com/definition/Java) applications. One of the most popular Java Enterprise Edition ([Java EE](https://www.theserverside.com/definition/J2EE-Java-2-Platform-Enterprise-Edition)) frameworks, Spring helps developers create high performing applications using plain old Java [objects](https://www.techtarget.com/searchapparchitecture/definition/object) (POJOs).

IOC

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. We most often use it in the context of object-oriented programming.

In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built in. **If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.**

The advantages of this architecture are:

* decoupling the execution of a task from its implementation
* making it easier to switch between different implementations
* greater modularity of a program
* greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts

We can achieve Inversion of Control through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

### Why Spring?

Java programs are complex and feature many heavyweight [components](https://www.techtarget.com/whatis/definition/component). Heavyweight means the components are dependent on the underlying operating system ([OS](https://www.techtarget.com/whatis/definition/operating-system-OS)) for their appearance and properties.

Spring is considered to be a secure, low-cost and flexible framework. Spring improves coding efficiency and reduces overall application development time because it is lightweight -- efficient at utilizing system resources -- and has a lot of [support](https://www.techtarget.com/whatis/definition/support).

Spring removes tedious configuration work so that developers can focus on writing [business logic](https://www.techtarget.com/whatis/definition/business-logic). Spring handles the infrastructure so developers can focus on the application.

### How Spring works

A [web application](https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app) (layered architecture) commonly includes three [layers](https://www.techtarget.com/searchsoftwarequality/definition/layer):

1. Presentation/view layer ([UI](https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI)) - This is the outermost layer which handles the presentation of content and interaction with the user.
2. Business logic layer - The central layer that deals with the logic of a program.
3. Data access layer - The deep layer that deals with data retrieval from sources.

Each layer is dependent on the other for an application to work. In other words, the presentation layer talks to the business logic layer, which talks to the data access layer. Dependency is what each layer needs to perform its function. A typical application has thousands of [classes](https://www.techtarget.com/whatis/definition/class) and many dependencies.

Without a Spring Framework, application code tends to be tightly coupled (interdependent), which is not considered good coding practice. Loose coupling is ideal because loosely coupled components are independent, meaning changes in one will not affect the operation of others.

Spring’s core logic is [dependency injection](https://www.techtarget.com/searchapparchitecture/definition/dependency-injection). Dependency injection is a programming pattern that allows developers to build more [decoupled architectures](https://www.techtarget.com/whatis/definition/decoupled-architecture). Dependency injection means that Spring understands the different [Java annotations](https://www.theserverside.com/definition/Java-annotations) that a developer puts on top of classes. Spring knows that the developer wants to create an [instance](https://www.techtarget.com/whatis/definition/instance) of a class and that Spring should manage it. Spring also understands the dependency and makes sure that all instances created have properly populated dependencies.

For the Spring Framework to instantiate objects and populate the dependencies, a programmer simply tells Spring which objects to manage and what the dependencies are for each class.

Indexing

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.

Indexes are created using a few database columns.

* The first column is the **Search key** that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly.   
  *Note: The data may or may not be stored in sorted order.*
* The second column is the **Data Reference** or **Pointer** which contains a set of pointers holding the address of the disk block where that particular key value can be found.

Why String is immutable?

We can not be sure of what was Java designers actually thinking while designing String but we can only conclude these reasons based on the advantages we get out of string immutability, Some of which are

1. Existence of String Constant Pool

As discussed in [Why String is Stored in String Constant Pool](https://programmingmitra.com/2018/02/why-string-is-stored-in-constant-pool.html) article, every application creates too many string objects and in order to save JVM from first creating lots of string objects and then garbage collecting them. JVM stores all string objects in a separate memory area called String constant pool and reuses objects from that cached pool.

Whenever we create a string literal JVM first sees if that literal is already present in constant pool or not and if it is there, new reference will start pointing to the same object in SCP.

String a = "Naresh";

String b = "Naresh";

String c = "Naresh";

In above example string object with value Naresh will get created in SCP only once and all reference a, b, c will point to the same object but what if we try to make change in a e.g. a.replace("a", "").

Ideally, a should have value Nresh but b, c should remain unchanged because as an end user we are making the change in a only. And we know a, b, c all are pointing the same object so if we make a change in a, others should also reflect the change.

But string immutability saves us from this scenario and due to the immutability of string object string object Naresh will never change. So when we make any change in a instead of change in string object Naresh JVM creates a new object assign it to a and then make change in that object.

So String pool is only possible because of String's immutability and if String would not have been immutable, then caching string objects and reusing them would not have a possibility because any variable woulds have changed the value and corrupted others.

And That's why it is handled by JVM very specially and have been given a special memory area.

2. Thread Safety

An object is called thread-safe when multiple threads are operating on it but none of them is able to corrupt its state and object hold the same state for every thread at any point in time.

As we an immutable object cannot be modified by anyone after its creation which makes every immutable object is thread safe by default. We do not need to apply any thread safety measures to it such as creating synchronized methods.

So due to its immutable nature string object can be shared by multiple threads and even if it is getting manipulated by many threads it will not change its value.

3. Security

In every application, we need to pass several secrets e.g. user's user-name\passwords, connection URLs and in general, all of this information is passed as the string object.

Now suppose if String would not have been immutable in nature then it would cause a serious security threat to the application because these values are allowed to get changed and if it is allowed then these might get changed due to wrongly written code or any other person who have access to our variable references.

4. Class Loading

As discussed in [Creating objects through Reflection in Java with Example](https://programmingmitra.com/2016/05/creating-objects-through-reflection-in-java-with-example.html), we can use Class.forName("class\_name") method to load a class in memory which again calls other methods to do so. And even JVM uses these methods to load classes.

But if you see clearly all of these methods accepts the class name as a string object so Strings are used in java class loading and immutability provides security that correct class is getting loaded by ClassLoader.

Suppose if String would not have been immutable and we are trying to load java.lang.Object which get changed to org.theft.OurObject in between and now all of our objects have a behavior which someone e can use to unwanted things.

5. HashCode Caching

If we are going to perform any hashing related operations on any object we must override the hashCode() method and try to generate an accurate hashcode by using the state of the object. If an object's state is getting changed which means its hashcode should also change.

Because String is immutable so the value one string object is holding will never get changed which means its hashcode will also not change which gives String class an opportunity to cache its hashcode during object creation.

Yes, String object caches its hashcode at the time of object creation which makes it the great candidate for hashing related operations because hashcode doesn't need to be calculated again which save us some time. This is why String is mostly used as HashMap keys.

Umodifiable and Immutable

An unmodifiable collection is often a wrapper around a modifiable collection *which other code may still have access to*. So while *you* can't make any changes to it if you only have a reference to the unmodifiable collection, you can't rely on the contents not changing.

An *immutable* collection guarantees that *nothing* can change the collection any more. If it wraps a modifiable collection, it makes sure that no other code has access to that modifiable collection. Note that although no code can change which objects the collection contains references to, the objects themselves may still be mutable - creating an immutable collection of StringBuilder doesn't somehow "freeze" those objects.

Basically, the difference is about whether other code may be able to change the collection behind your back.

Basically unModifiable Collection is a view, So indirectly it could still be 'modified' from some other reference that is modifiable. Also as its just a ***readonly view*** of annother collection , When the source collection changes unModifiable Collection will always present with latest values.

However immutable Collection can be treated as a ***readonly copy*** of another collection and can not be modified. In this case when the source collection changes , immutable Collection do not reflect the changes

Because instance variables CANNOT be overridden in Java. In Java, only methods can be overridden.

When you declare a field with the same name as an existing field in a superclass, the new field *hides* the existing field. The existing field from the superclass is still present in the subclass, and can even be used ... subject to the normal Java access rules.

## Using the verify()Method

Whenever we mock a void method, we do not expect a return value. That is why we can only **verify** whether that method is being called or not.

Features of verify():

* Mockito provides us with a verify()method that lets us verify whether the mock void method is being called or not.
* It lets us check the number of methods invocations. So, if the method invocation returns to be zero, we would know that our mock method is not being called.

Method Overloading Variations

1.

widening and boxing of primitive types can not work together.

We are invoking another method with the argument of Long wrapper Object.

The compiler starts searching for the method having the same reference type (Long wrapper class).

Since there is no method having with parameter of Long wrapper class.

So, It searches for a method that can accept the parameter bigger than long primitive data type as an argument.

In this case, it finds a method with float primitive data type and invokes it.

2.

In method overloading, you may come across a situation where a signature takes reference type or a primitive type as a formal argument. The compiler first searches a method with parameter(s) of the same data type(s). If you are using wrapper class Object as an actual argument and the compiler does not find the method with parameter(s) of the same reference type (i.e. class or interface type), then it starts searching a method with parameter(s) having the corresponding primitive data type.

we'll execute [JVM Process Status (jps)](https://docs.oracle.com/en/java/javase/11/tools/jps.html) command to discover the PID process of our application:

Secondly, we get the PID for our application, in this case, the one next to the NetworkDriver. Then, we'll capture the thread dump using [jstack](https://docs.oracle.com/en/java/javase/11/tools/jstack.html). Finally, we'll store the result in a text file:

ReentrantLock and synchronized keyword in Java

1) Difference between ReentrantLock and synchronized keyword is fairness. synchronized keyword doesn't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make ReentrantLock fair by specifying fairness property, while creating instance of ReentrantLock. Fairness property provides lock to longest waiting thread, in case of contention.

2) Second difference between synchronized and Reentrant lock is tryLock() method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread. This reduce blocking of thread waiting for lock in Java application.

3) One more worth noting difference between ReentrantLock and synchronized keyword in Java is, ability to interrupt Thread while waiting for Lock. In case of synchronized keyword, a thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that. ReentrantLock provides a method called lockInterruptibly(), which can be used to interrupt thread when it is waiting for lock. Similarly tryLock() with timeout can be used to timeout if lock is not available in certain time period.

Why we need Serialization?

Let's define serialization first, then we can talk about why it's so useful.

Serialization is simply turning an existing object into a byte array. This byte array represents the class of the object, the version of the object, and the internal state of the object. This byte array can then be used between JVM's running the same code to transmit/read the object.

Why would we want to do this?

There are several reasons:

* Communication: If you have two machines that are running the same code, and they need to communicate, an easy way is for one machine to build an object with information that it would like to transmit, and then serialize that object to the other machine. It's not the best method for communication, but it gets the job done.
* Persistence: If you want to store the state of a particular operation in a database, it can be easily serialized to a byte array, and stored in the database for later retrieval.
* Deep Copy: If you need an exact replica of an Object, and don't want to go to the trouble of writing your own specialized clone() class, simply serializing the object to a byte array, and then de-serializing it to another object achieves this goal.
* Caching: Really just an application of the above, but sometimes an object takes 10 minutes to build, but would only take 10 seconds to de-serialize. So, rather than hold onto the giant object in memory, just cache it out to a file via serialization, and read it in later when it's needed.
* Cross JVM Synchronization: Serialization works across different JVMs that may be running on different architectures.

## What Is serialVersionUID? And Why Should We Declare It?

Suppose we have a class and we have serialized its object to a file on the disk, and due to some new requirements, we added/removed one field from our class. Now, if we try to deserialize the already serialized object, we will get InvalidClassException; why?

We get it because, by default, the JVM associates a version number to each serializable class to control the class versioning. It is used to verify that the serialized and deserialized objects have the same attributes and thus are compatible with deserialization. The version number is maintained in a field called serialVersionUID. If a serializable class doesn't declare a serialVersionUID, the JVM will generate one automatically at run-time.

If we change our class structure, e.g. remove/add fields, that version number also changes, and according to the JVM, our class is not compatible with the class version of the serialized object. That's why we get the exception, but if you really think about it, why should it be thrown just because I added a field? Couldn't the field just be set to its default value and then written out next time?

Yes, it can be done by providing the serialVersionUID field manually and ensuring it is always the same. It is highly recommended that each serializable class declares its serialVersionUID as the generated one is compiler dependent and thus may result in unexpected *InvalidClassExceptions*.

**Compatible Changes :**

Compatible changes are those changes which **does not affect** deSerialization process even if class was updated after being serialized (provided serialVersionUID has been declared)

* **Adding new fields** - We can add new member variables in class.
* **Adding writeObject()/readObject()  methods** - We may add these methods to customize serialization process.
* **Removing writeObject()/readObject() methods** - We may remove these methods and then default customization process will be used.
* **Changing access modifier of a field** - The change to access modifiers i.e. public, default, protected, and private have no effect on the ability of serialization to assign values to the fields.
* **Changing a field from static to non static OR changing transient filed to non transient field**. - it’s like addition of fields.

**InCompatible Changes :**

InCompatible changes are those changes which affect deSerialization process if class was updated after being serialized (provided serialVersionUID has been declared)

* **Deletion of fields.**
* **Changing a nonstatic field to static or  non transient field to transient field. -** it’s equal to deletion of fields.
* **Modifying the writeObject() / readObject() method** - we must not modify these method, though adding or removing them completely is compatible change.

*Java 8u20* has introduced one more *JVM* parameter for reducing the unnecessary use of memory by creating too many instances of the same *String.* This optimizes the heap memory by removing duplicate *String* values to a global single *char[]* array.

We can enable this parameter by adding ***-XX:+UseStringDeduplication*** as a *JVM* parameter

CLASS LOADERS

Class loaders are responsible for**loading Java classes dynamically to the JVM** **(Java Virtual Machine) during runtime.** They're also part of the JRE (Java Runtime Environment). Therefore, the JVM doesn't need to know about the underlying files or file systems in order to run Java programs thanks to class loaders.

Furthermore, these Java classes aren't loaded into memory all at once, but rather when they're required by an application. This is where class loaders come into the picture. They're responsible for loading classes into memory.

The application class loader loads the class where the example method is contained. **An application or system class loader loads our own files in the classpath.**

Next, the extension class loader loads the Logging class.**Extension class loaders load classes that are an extension of the standard core Java classes.**

Finally, the bootstrap class loader loads the ArrayList class. **A bootstrap or primordial class loader is the parent of all the others.**

However, we can see that for the ArrayList, it displays null in the output. **This is because the bootstrap class loader is written in native code, not Java, so it doesn't show up as a Java class.** As a result, the behavior of the bootstrap class loader will differ across JVMs.

### Bootstrap Class Loader

Java classes are loaded by an instance of java.lang.ClassLoader. However, class loaders are classes themselves. So the question is, who loads the java.lang.ClassLoader itself?

This is where the bootstrap or primordial class loader comes into play.

It's mainly responsible for loading JDK internal classes, typically rt.jar and other core libraries located in the $JAVA\_HOME/jre/lib directory. Additionally, the **Bootstrap class loader serves as the parent of all the other ClassLoader instances**.

**This bootstrap class loader is part of the core JVM and is written in native code,** as pointed out in the above example. Different platforms might have different implementations of this particular class loader.

### Extension Class Loader

The **extension class loader is a child of the bootstrap class loader, and takes care of loading the extensions of the standard core Java classes** so that they're available to all applications running on the platform.

The extension class loader loads from the JDK extensions directory, usually the $JAVA\_HOME/lib/ext directory, or any other directory mentioned in the java.ext.dirs system property.

### System Class Loader

The system or application class loader, on the other hand, takes care of loading all the application level classes into the JVM. **It loads files found in the classpath environment variable, -classpath, or -cp command line option**. It's also a child of the extensions class loader.

## How Do Class Loaders Work?

Class loaders are part of the Java Runtime Environment. When the JVM requests a class, the class loader tries to locate the class and load the class definition into the runtime using the fully qualified class name.

The **java.lang.ClassLoader.loadClass() method is responsible for loading the class definition into runtime**. It tries to load the class based on a fully qualified name.

If the class isn't already loaded, it delegates the request to the parent class loader. This process happens recursively.

Eventually, if the parent class loader doesn’t find the class, then the child class will call the java.net.URLClassLoader.findClass() method to look for classes in the file system itself.

If the last child class loader isn't able to load the class either, it throws [java.lang.NoClassDefFoundError or java.lang.ClassNotFoundException.](https://www.baeldung.com/java-classnotfoundexception-and-noclassdeffounderror)

### Delegation Model

Class loaders follow the delegation model, where **on request to find a class or resource, a ClassLoader instance will delegate the search of the class or resource to the parent class loader**.

Let's say we have a request to load an application class into the JVM. The system class loader first delegates the loading of that class to its parent extension class loader, which in turn delegates it to the bootstrap class loader.

### Only if the bootstrap and then the extension class loader are unsuccessful in loading the class, the system class loader tries to load the class itself.

### Creating Our Custom Class Loader

For illustration purposes, let's say we need to load classes from a file using a custom class loader.

**We need to extend the ClassLoader class and override the findClass() method:**

* Singleton pattern ensures one instance of a particular class of per class loader.
* Spring Singleton is  “per container per bean”.

## **Runtime compilation**

The best (most well known) example I'm personally aware of is the [just in time compilation](https://en.wikipedia.org/wiki/Just-in-time_compilation) used by Java. As you might know Java code is being compiled into bytecode which can be interpreted by the Java Virtual Machine. It's therefore different from let's say C++ which is first fully (preprocessed) compiled (and linked) into an executable which can be ran directly by the OS without any virtual machine.

The Java bytecode is instead interpreted by the VM, which maps them to processor specific instructions. That being said the JVM does JIT, which takes that bytecode and compiles it (during runtime) into machine code. Here we arrive at your **second question**. Even in Java it can depend on which JVM you are using but basically there are pieces of code called **hotspots**, the pieces of code that are run frequently and which might be compiled so the application's performance improves. This is done during runtime because the normal compiler does not have (or well might not have) all the necessary data to make a proper judgement which pieces of code are in fact ran frequently. Therefore JIT requires some kind of **runtime statistics** gathering, which is done parallel to the program execution and is done by the **JVM**. What kind of statistics are gathered, what can be optimised (compiled in runtime) etc. depends on the implementation (you obviously cannot do everything a normal compiler would do due to memory and time constraints - guess this partly answers the **first question**? you don't compile everything and usually only a limited set of optimisations are supported in runtime compilation). You can try looking for such info but from my experience usually it's very badly documented and hard to find (at least when it comes to official sources, not presentations/blogs etc.)

## **Runtime linking**

Linker is a different pair of shoes. We cannot use the Java example anymore since it doesn't really have a linker like C or C++ (instead it has a classloader which takes care of loading files and putting it all together).

Usually linking is performed by a linker after the compilation step (static linking), this has pros (no dependencies) and cons (higher memory imprint as we cannot use a shared library, when the library number changes you need to recompile your sources).

Runtime linking (dynamic/late linking) is actually performed by the **OS** and it's the OS linker's job to first load shared libraries and then attach them to a running process. Furthermore there are also different types of dynamic linking: explicit and implicit. This has the benefit of not having to recompile the source when the version number changes since it's dynamic and library sharing but also drawbacks, what if you have different programs that use the same library but require different versions (look for **DLL hell**). So yes those two concepts are also quite different.

Again how it's all done, how it's decided what and how should be linked, is OS specific, for instance Microsoft has the [dynamic-link library](https://en.wikipedia.org/wiki/Dynamic-link_library) concept.

## Difference between Setter and Constructor Injection in Spring framework

Spring supports both setter and constructor Injection, which are two standard ways of injecting dependency on beans managed by IOC constructor. Spring framework doesn't support Interface Injection on which dependency is injected by implementing a particular interface.   
  
In this section, we will see a couple of differences between setter and constructor Injection, which will help you decide when to use setter Injection over constructor Injection in Spring and vice-versa.

1) The fundamental difference between setter and constructor injection, as their name implies, is How dependency is injected.  Setter injection in Spring uses setter methods like setDependency() to inject dependency on any bean managed by Spring's IOC container. On the other hand, constructor injection uses the [constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) to inject dependency on any Spring-managed bean.

2) Because of using the setter method, setter Injection in more readable than constructor injection in Spring configuration file usually applicationContext.xml . Since the setter method has name like setReporotService() by reading Spring XML config file you know which dependency you are setting. While in constructor injection, since it uses an index to inject the dependency, it's not as readable as setter injection and you need to refer either Java documentation or code to find which index corresponds to which property.

3) Another difference between setter vs constructor injection in Spring and one of the drawbacks of setter injection is that it does not ensures [dependency Injection](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html). You can not guarantee that certain dependency is injected or not, which means you may have an object with incomplete dependency. On the other hand, constructor Injection does not allow you to construct an object until your dependencies are ready.

4) One more drawback of setter Injection is Security. By using setter injection, you can [override](http://javarevisited.blogspot.in/2011/12/method-overloading-vs-method-overriding.html) certain dependency which is not possible with constructor injection because every time you call the constructor, a new object is gets created.  
  
  
5) One more difference between Setter and Constructor Injection in Spring, where later can help if there is a circular dependency between two object A and B.

-------------------------------------------------------------------------------------------

How to remove or stop a bean?

We figured out a way to do it by adding a BeanFactoryPostProcessor to the ConfigurableWebApplicationContext in the method customizeContext(). It seems solved our problems.

Code changes are:

protected void customizeContext(ServletContext servletContext, ConfigurableWebApplicationContext applicationContext) {

super.customizeContext(servletContext, applicationContext);

applicationContext.addBeanFactoryPostProcessor(new BootProcessor());

}

class BootProcessor implements BeanFactoryPostProcessor{

@Override

public void postProcessBeanFactory(ConfigurableListableBeanFactory clbf) throws BeansException {

BeanDefinitionRegistry beanDefinitionRegistry = (BeanDefinitionRegistry) clbf;

beanDefinitionRegistry.removeBeanDefinition("testerBean");

}

}

Rest Statelessness

As per the REST architecture, a RESTful Web Service should not keep a client state on the server. This restriction is called Statelessness. It is the responsibility of the client to pass its context to the server and then the server can store this context to process the client's further request. For example, session maintained by server is identified by session identifier passed by the client.

By stateless it means that the **server** does not store any state about the **client session** on the server side.

# [**If REST applications are supposed to be stateless, how do you manage sessions?**](https://stackoverflow.com/questions/3105296/if-rest-applications-are-supposed-to-be-stateless-how-do-you-manage-sessions)

The client session is stored on the client. The server is stateless means that every server can service any client at any time, there is no session affinity or sticky sessions. The relevant session information is stored on the client and passed to the server as needed.

That does not preclude other services that the web server talks to from maintaining state about business objects such as shopping carts, just not about the client's current application/session state.

The **client's** application state should never be stored on the server, but passed around from the **client** to every place that needs it.

That is where the ST in REST comes from, State Transfer. You transfer the state around instead of having the server store it. **This is the only way to scale to millions of concurrent users.** If for no other reason than because millions of sessions is millions of sessions.

The load of session management is amortized across all the clients, the clients store their session state and the servers can service many orders of magnitude or more clients in a stateless fashion.

Even for a service that you think will only need in the 10's of thousands of concurrent users, you still should make your service stateless. Tens of thousands is still tens of thousands and there will be time and space cost associated with it.

Stateless is how the HTTP protocol and the web in general was designed to operate and is an overall simpler implementation and you have a single code path instead of a bunch of server side logic to maintain a bunch of session state.

### There are some very basic implementation principles:

These are principles not implementations, how you meet these principles may vary.

In summary, the [five key principles](https://www.infoq.com/articles/rest-introduction) are:

1. Give every “thing” an ID
2. Link things together
3. Use standard methods
4. Resources with multiple representations
5. Communicate statelessly

ClassNotFoundException Vs NoClassDefFoundError

ClassNotFoundException is an exception that occurs when you try to load a class at run time using Class. forName() or loadClass() methods and mentioned classes are not found in the classpath. NoClassDefFoundError is an error that occurs when a particular class is present at compile time, but was missing at run time.

# Why to use char[] array over a string for storing passwords in Java?

**1. Strings are immutable:** Strings are immutable in Java and therefore if a password is stored as plain text it will be available in memory until Garbage collector clears it and as Strings are used in the String pool for re-usability there are high chances that it will remain in memory for long duration, which is a security threat. Strings are immutable and there is no way that the content of Strings can be changed because any change will produce new String.   
Within an array, the data can be wiped explicitly after its work is completed. The array can be overwritten and the password won’t be present anywhere in the system, even before garbage collection.

**2. Security:** Any one who has access to memory dump can find the password in clear text and that’s another reason to use encrypted password than plain text.  So Storing password in character array clearly mitigates security risk of stealing password.

**3. Log file safety:** With an array, one can explicitly wipe the data , overwrite the array and the password won’t be present anywhere in the system.   
With plain String, there are much higher chances of accidentally printing the password to logs, monitors or some other insecure place. char[] is less vulnerable.

**4. Java Recommendation:** Java has methods like JPasswordField of javax.swing as the method public String getText() which returns String is Deprecated from Java 2 and is replaced by public char[] getPassword() which returns Char Array.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method.

### ****1. DELETE :****

Basically, it is a [Data Manipulation Language Command (DML)](https://www.geeksforgeeks.org/sql-ddl-dql-dml-dcl-tcl-commands/). It is used to delete one or more tuples of a table. With the help of the “DELETE” command, we can either delete all the rows in one go or can delete rows one by one. i.e., we can use it as per the requirement or the condition using the Where clause. It is comparatively slower than the TRUNCATE command. The TRUNCATE command does not remove the structure of the table.

* **SYNTAX –**   
  If we want to delete all the rows of the table:

DELETE from;

* **SYNTAX –**   
  If we want to delete the row of the table as per the condition then we use the WHERE clause,

DELETE from WHERE ;

**Note –** Here we can use the “ROLLBACK” command to restore the tuple because it does not auto-commit.

### ****2. DROP :****

It is a Data Definition Language Command (DDL). It is used to drop the whole table. With the help of the “DROP” command we can drop (delete) the whole structure in one go i.e. it removes the named elements of the schema. By using this command the existence of the whole table is finished or say lost.

* **SYNTAX –**   
  If we want to drop the table:

DROP table ;

**Note –** Here we can’t restore the table by using the “ROLLBACK” command because it auto commits.

### ****3. TRUNCATE :****

It is also a Data Definition Language Command (DDL). It is used to delete all the rows of a relation (table) in one go. With the help of the “TRUNCATE” command, we can’t delete the single row as here WHERE clause is not used. By using this command the existence of all the rows of the table is lost. It is comparatively faster than the delete command as it deletes all the rows fastly.

* **SYNTAX –**   
  If we want to use truncate :

TRUNCATE;

**Note –** Here we can’t restore the tuples of the table by using the “ROLLBACK” command.

**Primary Key**

**Primary Key** is a column that is used to uniquely identify each tuple of the table.

It is used to add integrity constraints to the table. Only one primary key is allowed to be used in a table. Duplicate and NULL (empty) values are not valid in the case of the primary key. Primary keys can be used as foreign keys for other tables too.

Let’s take an example,

We have a table name employee which stores data of employees of a company. The below table shows the contents of the table.

**Unique Key**

**Unique key** is a constraint that is used to uniquely identify a tuple in a table.

Multiple unique keys can present in a table. NULL values are allowed in case of a unique key. These can also be used as foreign keys for another table.

Let’s take an example,

We have a table name employee which stores data of employees of a company. The below table shows the contents of the table.

**Difference between Primary Key and Unique Key**

| **Primary Key** | **Unique Key** |
| --- | --- |
| Unique identifier for rows of a table | Unique identifier for rows of a table when primary key is not present |
| Cannot be NULL | Can be NULL |
| Only one primary key can be present in a table | Multiple Unique Keys can be present in a table |
| present in a table | present in a table |
| Selection using primary key creates clustered index | Selection using unique key creates non-clustered index |

A PRIMARY KEY column is equivalent to UNIQUE and NOT NULL and is indexed column by default.  
It should be UNIQUE because a primary key identifies rows in a table so 2 different row should not have the same key.  
In addition a primary key may be used a FOREIGN KEY in other tables and that's why it cannot be NULL so that the other table can fin the rows in the referenced table.

**Composite Key:**

A composite key is made by the combination of two or more columns in a table that can be used to uniquely identify each row in the table when the columns are combined uniqueness of a row is guaranteed, but when it is taken individually it does not guarantee uniqueness, or it can also be understood as a primary key made by the combination of two or more attributes to uniquely identify every row in a table.

[Static Class](https://www.geeksforgeeks.org/static-class-in-java/)

[Static nested classes](https://www.geeksforgeeks.org/nested-classes-java/) are nested classes that are declared static. In Java, Static classes are a convenient way of grouping classes together. Java doesn’t allow you to create top-level static classes; only nested (inner) classes. That is why a static class is also known as a static inner class or static nested class.

## **new operator in Java**

**new is an operator in Java**, it creates a new object of a type that is known beforehand and allocates memory dynamically for the object

## **newInstance() method in Java**

**newInstance()** method is present in **java.lang.Class** is used to create a new instance of the class dynamically.

Imagine a situation where you load classes dynamically from a remote source and you will not be able to import those classes during compile-time. In those cases we will not able to use the regular [**new keyword**](https://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html) to create the object, we have to go only for the **newInstance()** method.

|  |  |
| --- | --- |
| **new Operator** | **newInstance() method** |
| **Operator** in Java | **Method** present in java.lang.Class |
| **new operator** can be used to create an object if we **know the type of object beforehand** | **newInstance() method** can be used to create an object if we **doesn’t know the type of object beforehand** and we will be getting at **runtime** |
| Can call **any constructor** such as **no-args constructor**and **parameterized constructor** | Can call only the **no-args constructor** and it is **mandatory** to have a **no-args constructor** |
| At **runtime**, if the **.class file** is **not available**, then we will be getting **NoClassDefFoundError** | At **runtime**, if the **.class file** is **not available**, then we will be getting **ClassNotFoundException** |

Serialization is a process of converting Java objects to a byte array so as to preserve those objects into memory because after the execution of a program Java destroys the objects.

**POJO vs Java Bean**

| POJO | Java Bean |
| --- | --- |
| It doesn’t have special restrictions other than those forced by Java language. | It is a special POJO which have some restrictions. |
| It doesn’t provide much control on members. | It provides complete control on members. |
| It can implement Serializable interface. | It should implement serializable interface. |
| Fields can be accessed by their names. | Fields are accessed only by getters and setters. |
| Fields can have any visibility. | Fields have only private visibility. |
| There may/may-not be a no-arg constructor. | It must have a no-arg constructor. |
| It is used when you don’t want to give restriction on your members and give user complete access of your entity | It is used when you want to provide user your entity but only some part of your entity. |

## 

## Optimize queries based on the query optimization guidelines

Follow the SQL best practices to ensure query optimization:

1. Index all the predicates in JOIN, WHERE, ORDER BY and GROUP BY clauses.

WebSphere Commerce typically depends heavily on indexes to improve SQL performance and scalability. Without proper indexes, SQL queries can cause table scans, which causes either performance or locking problems. It is recommended that all predicate columns be indexed. The exception being where column data has very low cardinality.

1. Avoid using functions in predicates.

The index is not used by the database if there is a function on the column. For example:

SELECT \* FROM TABLE1 WHERE UPPER(COL1)='ABC'

As a result of the function UPPER(), the index on COL1 is not used by database optimizers. If the function cannot be avoided in the SQL, you need to create a function-based index in Oracle or generated columns in DB2 to improve performance.

1. Avoid using wildcard (%) at the beginning of a predicate.

The predicate LIKE '%abc' causes full table scan. For example:

SELECT \* FROM TABLE1 WHERE COL1 LIKE '%ABC'

This is a known performance limitation in all databases.

1. Avoid unnecessary columns in SELECT clause.

Specify the columns in the SELECT clause instead of using SELECT \*. The unnecessary columns places extra loads on the database, which slows down not just the single SQL, but the whole system.

1. Use inner join, instead of outer join if possible.

The outer join should only be used if it is necessary. Using outer join limits the database optimization options which typically results in slower SQL execution.

1. DISTINCT and UNION should be used only if it is necessary.

DISTINCT and UNION operators cause sorting, which slows down the SQL execution. Use UNION ALL instead of UNION, if possible, as it is much more efficient.

1. Oracle 10g and 11g requires that the CLOB/BLOB columns must be put at the end of the statements.

Otherwise, it causes failure when the input value size is larger than 1000 characters.

1. The ORDER BY clause is mandatory in SQL if the sorted result set is expected.

The ORDER BY keyword is used to sort the result-set by specified columns. Without the ORDER BY clause, the result set is returned directly without any sorting. The order is not guaranteed. Be aware of the performance impact of adding the ORDER BY clause, as the database needs to sort the result set, resulting in one of the most expensive operations in SQL execution.

Some of the disadvantages of REST are:

- Since there is no contract defined between service and client, it has to be communicated through other means such as documentation or emails.

- Since it works on HTTP, there can't be asynchronous calls.

- Sessions can't be maintained.

REST is purely an HTTP transport based call and you will receive a response say 200 OK on the other side,

SOAP uses two varieties,

* Synchronous Messaging over HTTP
* Asynchronous Messaging over HTTP

With synchronous messaging, the Requestor makes a request and the transport layer code blocks waiting for a response from the Provider. The Requestor receives the response on the same HTTP connection that the Requestor initially established to send the request. Synchronous exchange is usually easier to implement and requires that the Provider be able to generate a response in a short time, specifically in a time less than the HTTP timeout value(generally 120sec). [A single HTTP Connection is used that itself behaves Synchronously]

With asynchronous messaging, the Requestor is able to release transport specific resources once the request is acknowledged by the responder, knowing that a response will eventually be received. When the Provider completes the processing of the message it sends a response back to the Requestor over a new HTTP connection. [Here we utilize two HTTP Connections to implement an asynchronous messaging

* first HTTP Connection is used that for sending the Request and receiving an acknowledgement HTTP Response 200/OK
* second HTTP Connection is used for receiving the callback and responding HTTP Response 200/OK]

# Asynchronous Communication

Asynchronous communication is a widely-used communication method between different processes and systems. In an asynchronous communication, the client sends a request to the server (which requires lengthy processing) and receives a delivery acknowledgment right away. Different from the synchronous communication, this response does not have the required information, yet.

After the client receives the acknowledgment, it continues to do its other tasks, assuming it will eventually be notified when the required information is ready on the server side.

The biggest benefit of asynchronous communication is the increased performance. Since the client does not block its valuable CPU cycles just for waiting, it can deliver more within the same timeframe. Increased decoupling between the client-server interaction will also lead to better scalability.

Decouple client-server

Increased Performance

Scalabality

**HTTPMessageConverters**

we can use message converters to marshall and unmarshall Java Objects to and from JSON and XML over HTTP.

Each *HttpMessageConverter* implementation has one or several associated MIME Types.

When receiving a new request, **Spring will use the “*Accept*” header to determine the media type that it needs to respond with**.

It'll then try to find a registered converter that's capable of handling that specific media type. Finally, it'll use this to convert the entity and send back the response.

The process is similar for receiving a request that contains JSON information. The framework will **use the “*Content-Type*” header to determine the media type of the request body**.

Then it'll search for a HttpMessageConverter that can convert the body sent by the client to a Java Object.

**SNS vs SQS: Primary Differences**

There are some primary differences between SQS and SNS, owing to the way they are designed and the purpose they serve.

* SQS does not push messages to the consumers, instead, consumers have to poll the queue, and as soon as one of them receives a message, the message is out of the queue and no other consumer can access it. This polling inevitably introduces a certain latent delay in message delivery.
* SNS pushes the messages to all its subscribers as soon as it receives it, hence there is no latency and you can easily add subscribers down the line.
* SQS is mainly used to decouple applications or integrate applications.
* SNS is used to broadcast messages and it’s up to the receivers how they interpret and process those messages

## **There are three objects created by the code (including the interned "a")**

String s1 = "a";

String s2 = "a";

String s3 = new String("a");

a) s1 and s2 are just referenced,not objects, and they point to the same String in memory.

b) The "a" is interned and is a compound object: one char[] object and the String object itself. It consisting of two objects in memory.

c) s3, new String("a") produces one more object. The new String("a") does not copy the char[] of "a", it only references it internally. Here is the method signature:

public String2(String original) {

this.value = original.value;

this.hash = original.hash;

}

One interned String ("a") equals 2 Objects. And one new String("a") equals one more object. Net effect from code is three objects.

The **finally block** will always execute even an exception occurred or not in Java. If we call the **System.exit()** method explicitly in the**finally block** then only it will not be executed. There are few situations where the finally will not be executed like **JVM crash**, **power failure**, **software crash** and etc. Other than these conditions, the **finally block** will be always executed.

A plugin is an extension to Maven, something used to produce your artifact

(maven-jar-plugin for an example, is used to, you guess it, make a jar out of your compiled classes and resources).

A dependency is a library that is needed by the application you are building, at compile and/or test and/or runtime time.

Though both are the .jar files, the basic difference between them is plugin is used to execute the operations that we perform in maven,

and dependency is added to the classpath when we execute our operations.

You can’t use the plugin as a dependency because it just adds the plugin to the classpath and doesn't start a compilation.

----------------------------------------------------------------

The main reasons not to use H2 (or HSQLDB, or Derby) for production are:

Probability of critical bugs: compared to the 'big' databases Oracle, IBM DB 2, MS SQL Server, MySQL, PostgreSQL, the Java databases are relatively new and therefore possibly not as stable (have bugs).

Please note this is true for all newer products, including NoSQL databases, and new releases of the 'big' databases. Generally, the more a product is tested, the less the probability of bugs.

Of course it depends on your use case whether it makes sense to pay (possibly a lot of money) for this advantage.

In any case you will need to backup the data, in case of hardware failure for example.

Missing features and optimizations: the 'big' databases have more features and optimizations for special use cases.

Whether or not you need those features is up to you.

Commercial support: it's easier to get support for bigger databases. Please note commercial support is available for H2 as well. HSQLDB also provides commercial support.

IBM used to provide support for Apache Derby (well, IBM Cloudscape), but I believe they stopped.

Singleton: It returns a single bean instance per Spring IoC container.This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object. If no bean scope is specified in the configuration file, singleton is default.

Real world example: connection to a database

Prototype: It returns a new bean instance each time it is requested. It does not store any cache version like singleton.

Real world example: declare configured form elements (a textbox configured to validate names, e-mail addresses for example) and get "living" instances of them for every form being created

Request: It returns a single bean instance per HTTP request.

Real world example: information that should only be valid on one page like the result of a search or the confirmation of an order. The bean will be valid until the page is reloaded.

Session: It returns a single bean instance per HTTP session (User level session).

Real world example: to hold authentication information getting invalidated when the session is closed (by timeout or logout). You can store other user information that you don't want to reload with every request here as well.

GlobalSession: It returns a single bean instance per global HTTP session.

It is only valid in the context of a web-aware Spring ApplicationContext (Application level session).

It is similar to the Session scope and really only makes sense in the context of portlet-based web applications.

The portlet specification defines the notion of a global Session that is shared among all of the various portlets that make up a single portlet web application.

Beans defined at the global session scope are bound to the lifetime of the global portlet Session.

the parameter of Objects.hash is Object.... This has two main consequences:

Primitive values used in the hash code calculation have to be boxed, e.g. this.id is converted from long to Long.

An Object[] has to be created to invoke the method.

The cost of creating of these "unnecessary" objects may add up if hashCode is called frequently.

--------------------------------------------

Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

Docker provides the ability to package and run an application in a loosely isolated environment called a container.

The isolation and security allows you to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host.

You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.

-----------------------------------

Both String and Char[] array are used to store the textual data but choosing one over the other is more difficult. Maybe we can get the idea from the immutability of String why char[] array is preferred over String for storing sensitive information data like password, SSN, etc.

Using the plain string is a much higher chance of accidentally printing the password to logs or some other insecure places where char[] array is less vulnerable.

Since String is immutable, there is no method defined that allow us to change or overwrite the content of the string.

bThis feature makes string objects unstable for storing secure information such as passwords, SSN, etc.

We should always store the secure information in char[] array rather than String.

Since String is immutable if we store the password as plain text it will be available in memory until the garbage collector cleans it. Since string used String Constant Pool (SCP) for re-usability of a string, there will be a pretty chance that it will remain in memory for a long duration.

Since anyone who has access to memory dump can easily find the password in plain text that's another reason should use encrypt password than plain text.

If we notice in Java Swing applications, there is a method of JPasswordField getPassword() which return char[] and the deprecated method getText() which return the password in plain text. So java itself recommending to use the get password() method.

Another reason for storing a password in char[] array, because char[] can be sanitized, for example, after usage one can override a clear password with junk, while String is immutable in Java.

---------------------------------------

widening and boxing of primitive types can not work together.

We are invoking another method with the argument of Long wrapper Object.

The compiler starts searching for the method having the same reference type (Long wrapper class).

Since there is no method having with parameter of Long wrapper class.

So, It searches for a method that can accept the parameter bigger than long primitive data type as an argument.

In this case, it finds a method with float primitive data type and invokes it.

In method overloading, you may come across a situation where a signature takes reference type or a primitive type as a formal argument. The compiler first searches a method with parameter(s) of the same data type(s). If you are using wrapper class Object as an actual argument and the compiler does not find the method with parameter(s) of the same reference type (i.e. class or interface type), then it starts searching a method with parameter(s) having the corresponding primitive data type.

-----------------------------------------

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

UPDATE Customers

SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'

WHERE CustomerID = 1;

DB queries speed up

read replicas and database cache, increase instance size and connection pooling, sharding.

Finalize is an Object class method in Java. The finalize() method is a non-static and protected method of java.lang.object class. In java, the Object class is superclass of all java classes. Being an object class method finalize() method is available for every class in Java. Hence, Garbage Collector can call finalize() method on any java object for clean-up activity.

Finalize method in java is used to release all the resources used by the object before it is deleted/destroyed by the Garbage collector. finalize() is not a reserved keyword in java, it's a method. Once the clean-up activity is done by the finalize() method garbage collector immediately destroys the java object. Java Virtual Machine(JVM) permits invoking of finalize() method only once per object. Once object is finalized JVM sets a flag in the object header to say that it has been finalized, and won't finalize it again. If user tries to use finalize() method for the same object twice, JVM ignores it.

OAuth2

1)Add Jwt dependency – io.jsonwebtoken

Create custom Userdetails object, where we will load the user from repository and convert to spring User object

Add password encoder in Security config

2)We need to create a method to generateToken, which uses jwts library where we set the claims, subject, issued and expiration date and signing algorithm

3)Validate method – which receives token and userDetails

Extract username from token

We will verify username and expiration time

4) In Controller we need to autowire JwtUtil and AuthenticationManager bean.

Extract username and password from the requestbody

Add username and password in the authenticate method of above bean as UsernamePasswordAuthenticationToken object.

5) Add Authentication manager bean in security config as well

And Override configure method with HttpSecurity as argument

JWT consists 3 parts

Header(Signing algorithm), payload(User details and token expire time) and verify signature.

6) Create Jwt filter which extends OncePerRequestFilter

Override dofilterInternal, extract Authorization header from servlet request.

Extract Username and loadUserDetails from Username

Validate token with userDetails

UsernamePasswordAuthenticationToken is set and placed in securityContextholder

7) call doFilter method of filterChain

8) Inject filter in security config and register in the config.

9) sessionCreationPolicy should be stateless

SecurityConfig should extends WebSecurityConfigurerAdaptor and annotate with @EnableWebSecurity (and EnabelGlobalMethodSecurity)

Spring Security

Request redirects to delegatingFilterProxy.

It will extract that user info and convert to Authentication object

That object contains user info as Principal object as well as credential object

For Authentication, the filter will call AutheticationManager object

It will call authenticate method

ProviderManager – it will call support method of each authentication Provider cand check if they support

Then it will call authenticationProvider(multiple) object

It will take help from Userdetails service

It will return valid Authentication object till filter

Filter will place the same in security context holder.

DelegatingFilterProxy -> Authentication object created from request -> Authentication Manager(authenticate) -> Provider Manager -> AuthenticationProvider(Support, authenticate) -> UserDetailsService -> Authenticate Provider will validate and return valid Authentication object -> filter will set that object in security context.

# Reflection in Java

Reflection is an API that is used to examine or modify the behavior of methods, classes, and interfaces at runtime. The required classes for reflection are provided under **java.lang.reflect** package which is essential in order to understand reflection.

* Reflection gives us information about the class to which an object belongs and also the methods of that class that can be executed by using the object.
* Through reflection, we can invoke methods at runtime irrespective of the access specifier used with them.

Arrays and HashMap

* [Array](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html) has values, [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) has keys and values. Whereas you would access a value in array with something like this array[1], you can't do that with a HashMap. You have to call hashmap.get(key) to retrieve a desired value (which means you need a key to directly access the associated value).
* HashMaps cannot have repeating keys. Therefore, if you have a HashMap of keys and values, you can be sure that its keys are unique (values may contain duplicates across the key-value pairs).
* Arrays will maintain your order, and you can sort them. HashMap makes no guarantees on what order will be used. It also makes no guarantees that the order will remain constant over time. So if you're looking for it, you might be better off with [LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html).
* While it is necessary to know a key to retrieve a value from HashMap, you also have a contains(key) method, which is O(1) with a properly implemented hash. It allows you to check whether a key exists in a map. It is quite useful if you store associated data and want to check whether you already have a desired key.

### Procedural Programming

[Procedural Programming](https://www.geeksforgeeks.org/introduction-of-programming-paradigms/) can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedure. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. During a program’s execution, any given procedure might be called at any point, including by other procedures or itself.

### Object-Oriented Programming

[Object-oriented programming](https://www.geeksforgeeks.org/basic-concepts-of-object-oriented-programming-using-c/) can be defined as a programming model which is based upon the concept of objects. Objects contain data in the form of attributes and code in the form of methods. In object-oriented programming, computer programs are designed using the concept of objects that interact with the real world. Object-oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

### The difference between "java.lang.OutOfMemoryError: Java heap space" and "java.lang.OutOfMemoryError: PermGen space"

If you are familiar with different generations on the heap and [How garbage collection works in java](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html) and aware of new, old and permanent generations of heap space then you would have easily figured out this OutOfMemoryError in Java. Permanent generation of the heap is used to store String pool and various Metadata required by JVM related to Class, method and other java primitives.  
  
Since **most of the JVM default size of Perm Space is around "64MB"** you can easily run out of memory if you have too many classes or a huge number of Strings in your project.  
  
  
An important point to remember is that it doesn't depend on **–Xmx**value so no matter how big your total heap size you can run OutOfMemory in perm space. The good thing is you can specify**the size of permanent generation** using JVM options **"-XX: PermSize"** and  **"-XX: MaxPermSize"** based on your project need.

One small thing to remember is that "=" is used to separate parameter and value while specifying the **size of perm space in the heap** while "=" is not required while [**setting maximum heap size in java**](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html), as shown in below example.  
  
**export JVM\_ARGS="-Xmx1024m -XX:MaxPermSize=256m"**

Another reason of "**java.lang.OutOfMemoryError: PermGen**" is memory leak through [Classloaders](http://javarevisited.blogspot.com/2012/12/how-classloader-works-in-java.html) and it’s very often surfaced in WebServer and application server like tomcat, WebSphere, glassfish or WebLogic.   
  
  
In the Application server, different classloaders are used to load different web applications so that you can deploy and undeploy one application without affecting other applications on the same server, but while undeploying if the container somehow keeps a reference of any class loaded by application class loader then that class and all other related class will not be garbage collected and can quickly fill the PermGen space if you deploy and undeploy your application many times.   
  
"*java.lang.OutOfMemoryError: PermGen*” has been observed many times in tomcat in our last project, but the solution of this problem are really tricky because first you need to know which class is causing a memory leak and then you need to fix that. Another reason of OutOfMemoryError in PermGen space is if any thread started by the application doesn't exit when you undeploy your application.

These are just some example of infamous classloader leaks, anybody who is writing code for loading and unloading classes has to be very careful to avoid this. You can also use **visualgc**for monitoring PermGen space, this tool will show the graph of PermGen space and you can see how and when Permanent space getting increased. I suggest using this tool before reaching to any conclusion.

### How to solve java.lang.OutOfMemoryError: Java heap space

1) An easy way to solve OutOfMemoryError in java is to [*increase the maximum heap size*](http://javarevisited.blogspot.com/2011/08/increase-heap-size-maven-ant.html) by using JVM options "-Xmx512M", this will immediately solve your OutOfMemoryError. This is my preferred solution when I get OutOfMemoryError in Eclipse, Maven, or ANT while building a project because based upon the size of the project you can easily run out of Memory.

Here is **an example of increasing maximum heap size of JVM**, Also its better to keep **-Xmx to -Xms**ration either 1:1 or 1:1.5 if you are setting heap size in your java application  
  
**export JVM\_ARGS="-Xms1024m -Xmx1024m"**  
  
2) The second way to resolve OutOfMemoryError in Java is rather hard and comes when you don't have much memory and even after increase the maximum heap size you are still getting java.lang.OutOfMemoryError, in this case, you probably want to profile your application and look for any memory leak.

You can also use [**Eclipse Memory Analyzer**](http://www.eclipse.org/mat/) to examine your heap dump or you can use any profiler like Netbeans or JProbe. This is a tough solution and requires some time to analyze and **find memory leaks**.

### How to solve java.lang.OutOfMemoryError: PermGen space

As explained in the above paragraph this OutOfMemory error in java comes when the Permanent generation of heap is filled up. To fix this OutOfMemoryError in Java, you need to *increase the heap size of the Perm space* by using the JVM option   **"-XX: MaxPermSize".**

You can also specify the initial size of Perm space by using    **"-XX: PermSize"** and by keeping both initial **and maximum Perm Space** you can prevent some full garbage collection that may occur when Perm Space gets re-sized. Here is **how you can specify initial and maximum Perm size in Java**:  
  
**export JVM\_ARGS="-XX:PermSize=64M -XX:MaxPermSize=256m"**  
  
Some time java.lang.OutOfMemoryError  in Java gets tricky and on those cases profiling remains ultimate solution.Though you have the freedom to increase heap size in java, it’s recommended that to follow memory management practices while coding and setting null to any unused references.  
That’s all from me on **OutOfMemoryError in Java** I will try to write more about finding the memory leak in java and using profiler in some other post. Please share what is your approach to solving *java.lang.OutOfMemoryError in Java*.

## What Are the Solutions to StackOverflowError?

There are couple of strategies to address  StackOverflowError.

### 1. Fix the Code

Because of a non-terminating recursive call (as shown in the above example), threads stack size can grow to a large size. In those circumstances, you must fix the source code that is causing recursive looping. When ‘StackOverflowError’ is thrown, it will print the stacktrace of the code that it was recursively executing. This code is a good pointer to start debugging and fixing the issue.

### 2. Increase Thread Stack Size (-Xss)

There might be legitimate reason where a threads stack size needs to be increased. Maybe thread has to execute large number of methods or lot of local variables/created in the methods thread has been executing? In such circumstances, you can increase the thread’s stack size using the JVM argument: ‘-Xss." This argument needs to be passed when you start the application.

# [**Concept behind putting wait(),notify() methods in Object class**](https://stackoverflow.com/questions/17840397/concept-behind-putting-wait-notify-methods-in-object-class)

In the Java language, you wait() on a particular instance of an Object – a monitor assigned to that object to be precise. If you want to send a signal to one thread that is waiting on that specific object instance then you call notify() on that object. If you want to send a signal to all threads that are waiting on that object instance, you use notifyAll() on that object.

If wait() and notify() were on the Thread instead then each thread would have to know the status of every other thread. How would thread1 know that thread2 was waiting for access to a particular resource? If thread1 needed to call thread2.notify() it would have to somehow find out that thread2 was waiting. There would need to be some mechanism for threads to register the resources or actions that they need so others could signal them when stuff was ready or available.

In Java, the object itself is the entity that is shared between threads which allows them to communicate with each other. The threads have no specific knowledge of each other and they can run asynchronously. They run and they lock, wait, and notify on the *object* that they want to get access to. They have no knowledge of other threads and don't need to know their status. They don't need to know that it is thread2 which is waiting for the resource – they just notify on the resource and whomever it is that is waiting (if anyone) will be notified.

In Java, we then use objects as synchronization, mutex, and communication points between threads. We synchronize on an object to get mutex access to an important code block and to synchronize memory. We wait on an object if we are waiting for some condition to change – some resource to become available. We notify on an object if we want to awaken sleeping threads.

// locks should be final objects so the object instance we are synchronizing on,

// never changes

private final Object lock = new Object();

...

// ensure that the thread has a mutex lock on some key code

synchronized (lock) {

...

// i need to wait for other threads to finish with some resource

// this releases the lock and waits on the associated monitor

lock.wait();

...

// i need to signal another thread that some state has changed and they can

// awake and continue to run

lock.notify();

}

There can be any number of lock objects in your program – each locking a particular resource or code segment. You might have 100 lock objects and only 4 threads. As the threads run the various parts of the program, they get exclusive access to one of the lock objects. Again, they don't have to know the running status of the other threads.

This allows you to scale up or down the number of threads running in your software as much as you want. You find that the 4 threads is blocking too much on outside resources, then you can increase the number. Pushing your battered server too hard then reduce the number of running threads. The lock objects ensure mutex and communication between the threads independent of how many threads are running.

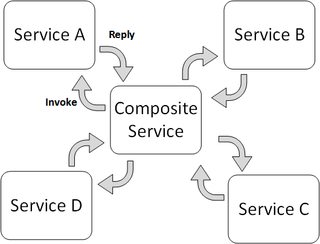
# [**Why use Optional.of over Optional.ofNullable?**](https://stackoverflow.com/questions/31696485/why-use-optional-of-over-optional-ofnullable)

Your question is based on assumption that the code which may throw NullPointerException is worse than the code which may not. This assumption is wrong. If you expect that your foobar is never null due to the program logic, it's much better to use Optional.of(foobar) as you will see a NullPointerException which will indicate that your program has a bug. If you use Optional.ofNullable(foobar) and the foobar happens to be null due to the bug, then your program will silently continue working incorrectly, which may be a bigger disaster. This way an error may occur much later and it would be much harder to understand at which point it went wrong

# Service orchestration

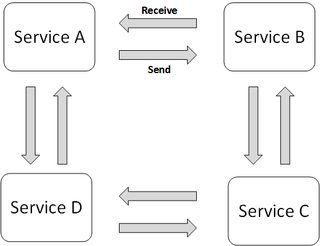
Service orchestration represents a single centralized executable business process (the orchestrator) that coordinates the interaction among different services. The orchestrator is responsible for invoking and combining the services.

The relationship between all the participating services are described by a single endpoint (i.e., the composite service). The orchestration includes the management of transactions between individual services. Orchestration employs a centralized approach for service composition.



# Service Choreography

Service choreography is a global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints. Choreography employs a decentralized approach for service composition.



The choreography describes the interactions between multiple services, where as orchestration represents control from one party's perspective. This means that a **choreography** differs from an **orchestration** with respect to where the logic that controls the interactions between the services involved should reside.

Saga Pattern

[**Saga pattern**](https://javarevisited.blogspot.com/2021/09/microservices-design-patterns-principles.html)**which is an asynchronous pattern** that performs a sequence of transactions in each microservice and publishes messages or events to proceed to the next step. If any step fails in between, compensating steps will be executed by the Saga pattern to reverse the transaction.

JWTs can be used in various ways:

* **Authentication**: When a user successfully logs in using their credentials, an [ID token](https://auth0.com/docs/secure/tokens/id-tokens) is returned. According to the [OpenID Connect (OIDC) specs](https://openid.net/specs/openid-connect-core-1_0.html#IDToken), an ID token is always a JWT.
* **Authorization**: Once a user is successfully logged in, an application may request to access routes, services, or resources (e.g., APIs) on behalf of that user. To do so, in every request, it must pass an Access Token, which may be in the form of a JWT. Single Sign-on (SSO) widely uses JWT because of the small overhead of the format, and its ability to easily be used across different domains.
* **Information Exchange**: JWTs are a good way of securely transmitting information between parties because they can be signed, which means you can be sure that the senders are who they say they are. Additionally, the structure of a JWT allows you to verify that the content hasn't been tampered with.

Monolithic Architecture

Single codebase

Big deployment

Single Jar/war

Difficult to manage

Hard to implement new features

Benefits:

Simple to develop

Easier debugging and testing

Simple to deploy

Transaction management

Challenges:

Complicated in understanding

Making new changes – (not reliable as bugs takes down whole app, smallest change requires new version of app)

New technology barriers – (entire app needs to be redeveloped, maintainability is effected)

Scalability – ( needs to scale whole app )

Use case:

When Application is small

Horizontal and Vertical Scaling

Load Balancer

Balance traffic to all nodes.

Improve responsiveness and availability of architecture.

Consistent hashing algorithm

Microservices

Small, Independent, loosely coupled ( Decouple with bounded context)

Separate codebase (tech agnostic)

Independent deployment

Persisting own data

Well defined API’s

Cloud native approach

Componentization via Services

Organized by business capabilities

Decentralized governance and data management

Design for failure

Highly maintainable and testable

Agility (Time to market is less)

Small and focussed team

Small and separated codebase

Fault Isolation

Scalable

Data Isolation

Challenges

Complexity (many moving parts)

Network problems and latency (communication)

Development and testing ( refactoring is difficult )

Data integrity (Data Consistency - eventual is preferred)

**Database per service pattern**

Loose coupling

Own databases

Polygot persistence

Can’t be accessed directly

Scale independently

Data encapsulated within the service

No affect to other services

**Choosing the right DB**

Structure of data

Query pattern

Amount of scale needed

Cache, Text based, Data warehouse

RDBMS, DocumentDB, ColumnarDB

Red-Black tree

A red-black tree is a self-balancing binary search tree, that is, a binary search tree that automatically maintains some balance.

Each node is assigned a color (red or black). A set of rules specifies how these colors must be arranged (e.g., a red node may not have red children). This arrangement ensures that the tree maintains a certain balance.

Red-Black Tree Properties

The following rules enforce the red-black tree balance:

1. Each node is either red or black.
2. (The root is black.)
3. All NIL leaves are black.
4. A red node must not have red children.
5. All paths from a node to the leaves below contain the same number of black nodes.

AVL Tree

An AVL tree is a balanced binary search tree – that is, a [binary search tree](https://www.happycoders.eu/algorithms/binary-search-tree-java/) in which the heights of the left and right subtrees of each node differ by at most one.

After each insert and delete operation, this invariant is verified, and the balance is restored by AVL rotation if necessary.

### AVL Tree vs. Red Black Tree

Both the AVL tree and the [red-black tree](https://www.happycoders.eu/algorithms/red-black-tree-java/) are self-balancing binary search trees.

In the AVL tree, we perform rebalancing by calculating balance factors and subsequent rotations. The absolute height difference at any node is not greater than 1.

In a red-black tree, nodes are marked by colors (red/black). Rotations occur when certain criteria for color sequences are no longer met. The absolute height difference at a node can be greater than 1. More precisely, the lowest leaf can be up to twice as far from the root as the highest leaf.

These characteristics result in the following differences:

* Searching in the AVL tree is usually faster than in the red-black tree because the AVL tree is better balanced.
* Insertions and deletions, on the other hand, are faster in a red-black tree because it rebalances less frequently.
* AVL trees need an extra byte per node for storing their height. Red-black trees need only one bit per node for the color information. In Java practice, this makes no difference as at least one byte is occupied for the bit anyway.

# Command Query Responsibility Segregation (CQRS)

An RDBMS is often used as the transactional system of record and a text search database such as Elasticsearch or Solr for text search queries. Some applications keep the databases in sync by writing to both simultaneously. Others copy data from the RDBMS to the text search engine periodically. The applications built on this architecture take advantage of the strengths of multiple databases, the transactional properties of the RDBMS, and the querying capabilities of the text database. CQRS generalizes this kind of architecture.

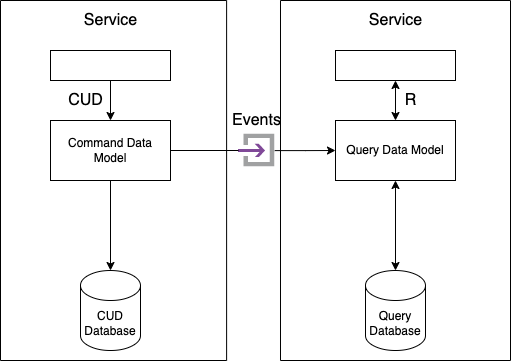
Microservice architectures face three common challenges when implementing queries.

1. Data scattered across multiple services is retrieved using the API composition pattern, resulting in expensive and inefficient in-memory joining.
2. Data is stored in a format or in a database that does not efficiently support the required query by the service that owns the data.
3. Separating concerns means that the service that owns the data shouldn’t be responsible for implementing query operations.

All three problems can be solved by using the CQRS pattern.

The primary goal of CQRS is to segregate or separate concerns. Hence, the persistent data model is split into two parts: the command side and the query side.

Create, update, and delete operations are implemented by the command side modules and the data model. Queries are implemented by the query side module and data model. By subscribing to the events published by the command line, the query side keeps its data model synchronized with the command side



Why Static methods can be accessed by objects?

Generally, public variables can be accessed by everybody, and private variables can only be accessed from within the current instance of the class. In your example you're allowed to access the x variable from the main method, because that method is within the Static class.

If you're wondering why you're allowed to access it from another instance of Static class than the one you're currently in (which generally isn't allowed for private variables), it's simply because static variables don't exist on a per-instance basis, but on a per class basis. This means that the same static variable of *A* can be accessed from all instances of *A*.

If this wasn't the case, nobody would be able to access the private static variable at all, since it doesn't belong to *one* instance, but them all.

Why String argument function gets called we pass null? – similar question

The reason why we get compile time error in the above scenario is, here the method arguments Integer and String both are not primitive data types in Java. That means they accept null values. When we pass a null value to the method1 the compiler gets confused which method it has to select, as both are accepting the null. This compile time error wouldn’t happen unless we intentionally pass null value. For example see the below scenario which we follow generally while coding.

**Microservices Decomposition Pattern**

Decompose by business capability

Services must be cohesive

Services must be loosely coupled

Corresponding to business capabilities

**Domain driven design (DDD)**

Decompose by subdomain

Bounded context – groups of sub domains(closely related scopes – logical boundaries)

Breaks the large complex system into different bounded contexts containing many sub domains

**DevOps**

Jenkins

Launch EC2

Install EPL and JDK for installing Jenkins

Install Jenkins

Build a new job

Integrate Github

Install Git on Jenkins instance

Install github plugin on Jenkins GUI

Configure Git on Jenkins GUI

yum install git

whereis git

Maven

Setup Maven on Jenkins Server

Setup environment variables JAVA\_HOME, M2, M2\_HOME

Install Maven plugin

Configure maven and java

Update .bash\_profiles with env variables

Tomcat server

Setup EC2

Install Java

Configure tomacat

Git – Github – Jenkins – Tomcat

Maven

Setup Docker

Setup EC2

Install docker

Start docker services

**DevOps cycle**

Doing changes on our workstation

Committing changes to git and github

Jenkins pull the latest code

Build the code using Maven

Jenkins will copy the artifacts to Ansible server

Ansible will take artifact and with the help of dockerfile, it will create image

Image is pushed to Docker hub

To deploy a container, dockerhost will communicate with dockerhub and pull the image and create a container out of it.

Ansible playbook is created for creating an image(Customized Tomcat with artifacts copied to webapps folder). One more playbook is created pulling the image and run the container.

In Jenkins commands has been added to execute the ansible playbook.

Kubernetes

Service and deployment manifest files are created.

Another ansible playbook is created to executed service and deployment files of Kubernetes(rollout command is also added for restart)

In Jenkins, that playbook is also executed.

useradd user

visudo – add user under root

vi /etc/ssh/sshd\_config – uncomment Passwordauth and comment other

service sshd reload

sudo su – user

ssh-copy-id ipaddress

## What are containers?

As Hykes described it in his PyCon talk, containers are “self-contained units of software you can deliver from a server over there to a server over there, from your laptop to EC2 to a bare-metal giant server, and it will run in the same way because it is isolated at the process level and has its own file system.”

By simplifying this process, Docker quickly became close to a de facto industry standard for containers. Docker let developers deploy, replicate, move, and back up a workload in one streamlined way, using a set of reusable images to make workloads more portable and flexible than previous methods allowed.

In the virtual machine (VM) world, this could be done by keeping applications separate while running on the same hardware, but in that world each VM requires its own operating system, meaning they are typically large, slow to start up, difficult to move around, and cumbersome to maintain and upgrade. Where containers marked a defined shift from the VM era was by isolating execution environments while sharing the underlying OS kernel, providing a lightweight and speedy option for developers.

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Container images become containers at runtime and in the case of Docker containers – images become containers when they run on [Docker Engine](https://www.docker.com/products/container-runtime). Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging. Docker containers that run on Docker Engine:

* **Standard:** Docker created the industry standard for containers, so they could be portable anywhere
* **Lightweight:** Containers share the machine’s OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs
* **Secure:** Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry

## Docker: the component parts

Docker took off with software developers because it found a novel way to package the tools required to build and launch a container in a more streamlined and simplified way than was previously possible. Broken down into its component parts, Docker comprises Dockerfile, container images, the Docker run utility, Docker Hub, Docker Engine, Docker Compose, and Docker Desktop.

**Dockerfile.**Each Docker container starts with a Dockerfile. This text file provides a set of instructions to build a Docker image, including the operating system, languages, environmental variables, file locations, network ports, and any other components it needs to run.

**Docker image.**Similar to a snapshot in the VM world, a Docker image is a portable, read-only, executable file containing the instructions for creating a container and the specifications for which software components the container will run and how.

**Docker run utility.** Docker’s run utility is the command that launches a container. Each container is an instance of an image, and multiple instances of the same image can be run simultaneously.

**Docker Hub.**Docker Hub is a repository where container images can be stored, shared, and managed. Think of it as Docker’s own version of GitHub, but specifically for containers.

**Docker Engine.**Docker Engine is the core of Docker. It is the underlying client-server technology that creates and runs the containers. The Docker Engine includes a long-running daemon process called dockerd for managing containers, APIs that allow programs to communicate with the Docker daemon, and a command-line interface.

**Docker Compose.**Docker Compose is a command-line tool that uses YAML files to define and run multicontainer Docker applications. It allows you to create, start, stop, and rebuild all the services from your configuration and view the status and log output of all running services.

**Docker Desktop.**All of these component parts are wrapped in Docker’s Desktop application, providing a user-friendly way to build and share containerized applications and microservices.

## Docker advantages

Docker containers provide a way to build applications that are easier to assemble, maintain, and move around than previous methods allowed. That provides several advantages to software developers.

**Docker containers are minimalistic and enable portability.**Docker lets applications and their environments be kept clean and minimal by isolating them, which allows for more granular control and greater portability.

**Docker containers enable composability.**Containers make it easier for developers to compose the building blocks of an application into a modular unit with easily interchangeable parts, which can speed up development cycles, feature releases, and bug fixes.

**Docker containers ease orchestration and scaling.**Because containers are lightweight, developers can launch lots of them for better scaling of services. These clusters of containers do then need to be orchestrated, which is where [Kubernetes](https://www.infoworld.com/article/3268073/what-is-kubernetes-your-next-application-platform.html) typically comes in.

## Docker drawbacks

Containers solve a great many problems, but they don’t solve all developer ills.

**Docker containers are not virtual machines.**Unlike virtual machines, containers use controlled portions of the host operating system’s resources, which means elements aren’t as strictly isolated as they would be on a VM.

**Docker containers don’t provide bare-metal speed.**Containers are significantly more lightweight and closer to the metal than virtual machines, but they do incur some performance overhead. If your workload requires bare-metal speed, a container will get you close but not all the way there.

**Docker containers are stateless and immutable.**Containers boot and run from an image that describes their contents. That image is immutable by default—once created, it doesn’t change. But a container instance is transient. When it is removed from system memory it’s gone forever. If you want your containers to persist state across sessions, like a virtual machine, you need to design for that persistence.

**Microservice Communication Design Patterns**

API gateway pattern

Routing to internal microservices

Managing cross cutting concerns

Aggregate several microservices

3 patterns

Gateway Routing pattern

Gateway Aggregation

Gateway Offloading

Service registry and service aggregator pattern used along with api gateway pattern

**Microservice Asynchronous Communication**

Scalability by kafka

**Microservice Data Management**

Data integrity and consistency challenges

Database per service pattern –

* 1. Helps us to scale independently
  2. Avoid single point of failure
  3. Decompose database into distributed data model
  4. Schema changes can be performed independently
  5. Can pick optimized database for each microservices

Relational Databases

Storing data in related tables

Fixed schema and use sql to query

Support transaction with ACID

NoSQL Databases

Stores unstructured data in key value pairs or json documents

Doesn’t provide ACID

Ease of use, scalable and resilient

Choosing the Right DB

Choose the consistency level

High scalability

High availability

CAP(Consistency, Availability, Partition Tolerance)

Microservices architecture chooses Partition tolerance with high availability and follow Eventual Consistency

Horizontal Scaling - Sharding

Each partition is a separate data store but same schema

Sharding separate the load on different servers with partition keys keys are organized alphabetically

Vertical Scaling

Row splitting

Holds a subset of the columns for table

Columns are divided according to their pattern

Functional Partitioning

Partitioning data by following the bounded context

Database sharding Pattern

Improve scalability

Improve performance by balancing the workload across shards

Dividing shards with partition keys

Materialized View Pattern

Microservice maintains a denormalized copy of the data (as read model) which is maintained by the microservices it depends.

Eliminates synchronous cross service calls

Data will be updated by event based mechanism (eventual consistency)

CQRS

Separate DB for reading and writing

Query to read DB (Materialized view) and command to to write DB.

Commands perform update data and queries perform read data

Commands should be actions with task based operations

Queries is never modifying DB.

Publish an update event using message broker

Read DB eventually synchronize with write DB

Event Sourcing pattern is used here

Event Sourcing

Save all events in DB with sequential ordered of data events

Append each change to a sequential list of events

Event store become a source of truth for the data

Publish/subscribe pattern with publish event

Replay event with latest status of data

Outbox pattern

The event messages are persisted in another outbox table parallel to entity table, a job then publishes events to message broker system

This pattern helps in data consistency

The database update and sending of messages should be atomic

Microservices containers

Containers provide to decouple the applications with all dependencies and isolate applications

We can deploy microservices separately in containers

New features can be added and rollbacked easily

Orchestrators automates the deployments, scaling and operational concerns of containerized workloads across clusters

**ACID**

### ****Atomicity:****

By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.   
—**Abort**: If a transaction aborts, changes made to the database are not visible.   
—**Commit**: If a transaction commits, changes made are visible.

### Consistency:

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database.

### Isolation:

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

### Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

**BASIC**

1. **Basically Available:** Instead of making it compulsory for immediate consistency, BASE-modelled NoSQL databases will ensure the availability of data by spreading and replicating it across the nodes of the database cluster.
2. **Soft State:** Due to the lack of immediate consistency, the data values may change over time. The BASE model breaks off with the concept of a database that obligates its own consistency, delegating that responsibility to developers.
3. **Eventually Consistent**: The fact that BASE does not obligates immediate consistency but it does not mean that it never achieves it. However, until it does, the data reads are still possible (even though they might not reflect reality).

**CAP**

* **Consistency –**   
  Consistency means that the nodes will have the same copies of a replicated data item visible for various transactions. A guarantee that every node in a distributed cluster returns the same, most recent and a successful write. Consistency refers to every client having the same view of the data. There are various types of consistency models. Consistency in CAP refers to sequential consistency, a very strong form of consistency.
* **Availability –**   
  Availability means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed. Every non-failing node returns a response for all the read and write requests in a reasonable amount of time. The key word here is “every”. In simple terms, every node (on either side of a network partition) must be able to respond in a reasonable amount of time.
* **Partition Tolerance –**   
  Partition tolerance means that the system can continue operating even if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other. That means, the system continues to function and upholds its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

**Aspect-Oriented Programming Basics**

AOP and AspectJ deserve comprehensive study in their own right hence I will give a very brief introduction into the main concepts:

* **Aspect**: logic which cuts across multiple parts of an application (exception handling in our case). Fun fact: transaction management (@Transactional) is implemented using AOP in Spring.
* **Join point**: a point during application execution when a cross-cutting concern occurs e.g. method execution, accessing a class field. **Spring AOP only allows method execution join points**.
* **Advice**: dictates when aspect logic is invoked e.g. before method execution, after etc.
* **Pointcut**: a logical condition which needs to be matched so that an aspect is executed. For example, we may want to execute certain logic only for methods of Java package services.

**API design - best practices**

Naming

Pagination

fragmentation

Versioning

setting API bounderies

atomicity

Consistency

service degradation

**Code quality issues**

replace SOP with logger

package naming - use regular expressions

remove unreferenced variables

nullpointer exception could be thrown

method not covered under tests

commented out lines of code should be removed

add comments why method is empty

## Transaction Propagation

Propagation defines our business logic's transaction boundary. Spring manages to start and pause a transaction according to our propagation setting.

### REQUIRED ****Propagation****

REQUIRED is the default propagation. Spring checks if there is an active transaction, and if nothing exists, it creates a new one. Otherwise, the business logic appends to the currently active transaction:

### SUPPORTS ****Propagation****

For SUPPORTS, Spring first checks if an active transaction exists. If a transaction exists, then the existing transaction will be used. If there isn't a transaction, it is executed non-transactional

### MANDATORY ****Propagation****

When the propagation is MANDATORY, if there is an active transaction, then it will be used. If there isn't an active transaction, then Spring throws an exception

### NEVER ****Propagation****

For transactional logic with NEVER propagation, Spring throws an exception if there's an active transaction:

### NOT\_SUPPORTED ****Propagation****

If a current transaction exists, first Spring suspends it, and then the business logic is executed without a transaction:

### REQUIRES\_NEW ****Propagation****

When the propagation is REQUIRES\_NEW, Spring suspends the current transaction if it exists, and then creates a new one:

### NESTED ****Propagation****

For NESTED propagation, Spring checks if a transaction exists, and if so, it marks a save point. This means that if our business logic execution throws an exception, then the transaction rollbacks to this save point. If there's no active transaction, it works like REQUIRED.

## Transaction Isolation

Isolation is one of the common ACID properties: Atomicity, Consistency, Isolation, and Durability. Isolation describes how changes applied by concurrent transactions are visible to each other.

Each isolation level prevents zero or more concurrency side effects on a transaction:

* **Dirty read:** read the uncommitted change of a concurrent transaction
* **Nonrepeatable read**: get different value on re-read of a row if a concurrent transaction updates the same row and commits
* **Phantom read:** get different rows after re-execution of a range query if another transaction adds or removes some rows in the range and commits

### READ\_UNCOMMITTED Isolation

READ\_UNCOMMITTED is the lowest isolation level and allows for the most concurrent access.

As a result, it suffers from all three mentioned concurrency side effects. A transaction with this isolation reads uncommitted data of other concurrent transactions. Also, both non-repeatable and phantom reads can happen. Thus we can get a different result on re-read of a row or re-execution of a range query.

### READ\_COMMITTED Isolation

The second level of isolation, READ\_COMMITTED, prevents dirty reads.

The rest of the concurrency side effects could still happen. So uncommitted changes in concurrent transactions have no impact on us, but if a transaction commits its changes, our result could change by re-querying.

### REPEATABLE\_READ Isolation

The third level of isolation, REPEATABLE\_READ, prevents dirty, and non-repeatable reads. So we are not affected by uncommitted changes in concurrent transactions.

Also, when we re-query for a row, we don't get a different result. However, in the re-execution of range-queries, we may get newly added or removed rows.

### SERIALIZABLE Isolation

SERIALIZABLE is the highest level of isolation. It prevents all mentioned concurrency side effects, but can lead to the lowest concurrent access rate because it executes concurrent calls sequentially.

N+1 Problem

Hibernate N+1 problem occurs when you use FetchType.LAZY for your entity associations. If you perform a query to select n-entities and if you try to call any access method of your entity's lazy association, Hibernate will perform n-additional queries to load lazily fetched objects.

N+1 problem is a performance issue in Object Relational Mapping that fires multiple select queries (N+1 to be exact, where N = number of records in table) in database for a single select query at application layer. Hibernate & Spring Data JPA provides multiple ways to catch and address this performance problem.

To resolve:

### Spring Data JPA Approach

If we are using Spring Data JPA, then we have two options to achieve this - using EntityGraph or using select query with fetch join.

@Query("SELECT p FROM User p LEFT JOIN FETCH p.roles")

List<User> findWithoutNPlusOne();

@EntityGraph(attributePaths = {"roles"})

List<User> findAll();

Dirty Checking Mechanism in Hibernate

Hibernate provides as feature called Automatic Dirty checking whereby changes to a persistent object are automatically saved to the database when the session is flushed or the transaction is committed. So the code does not need to invoke an explicit save or update

By default Hibernate checks all managed entity properties. Every time an entity is loaded, Hibernate makes an additional copy of all entity property values. At flush time, every managed entity property is matched against the loading-time snapshot value:

Even if only one property of a single entity has ever changed, Hibernate will still check all managed entities. For a large number of managed entities, the default dirty checking mechanism may have a significant CPU and memory footprint. Since the initial entity snapshot is held separately, the persistence context requires twice as much memory as all managed entities would normally occupy.

# **Blue/Green Deployments**

A blue/green deployment is a deployment strategy in which you create two separate, but identical environments. One environment (blue) is running the current application version and one environment (green) is running the new application version. Using a blue/green deployment strategy increases application availability and reduces deployment risk by simplifying the rollback process if a deployment fails. Once testing has been completed on the green environment, live application traffic is directed to the green environment and the blue environment is deprecated.

Disadvantages of Indexing

It is common knowledge that judicious use of indexes can help SELECT queries execute significantly faster. This can tempt some database admins (DBAs) to try to milk as much performance gains as possible by adding indexes to every column that might possibly be included in a query. The downside to adding indexes to a table is that they affect the performance of writes. Moreover, improperly created indexes can even adversely affect SELECT queries! Any table configuration where performance suffers due to excessive, improper, or missing indexes is considered to be poor indexing.

# **The Effects of Poor Indexing**

A poor index can be an index created on a column that doesn't provide easier data manipulation or an index created on multiple columns which, rather than speed up queries, slows them down.

If indexes are not created properly, the database has to go through more records in order to retrieve the data requested by a query. Therefore, it uses more hardware resources (processor, memory, disk, and network) and makes fetching the data take longer.

A table without a clustered index can also be considered as a poor indexing practice in some cases. Execution of a SELECT statement, inserting, updating, and deleting records is in most cases slower on a heap table (i.e. a table without a clustered index) than on a clustered one.

# **Choosing Columns For Clustered Indexes**

When you create a table with a primary key (PK) in a relational database, a unique clustered index is automatically created on the primary key column. Although this default action is perfectly acceptable in most cases, this might not be the optimal index for your data.

The columns that make up a clustered index should form a unique, identity, primary key, or any combination where values are increased for each new entry. As clustered indexes sort the records based on the value, using a column already ordered ascending, such as an identity column, is a good choice.

A column whose value changes frequently should not be used for a clustered index. The reason is that each change of the column used for the clustered index requires the records to be reordered. This re-ordering can easily be avoided by using a column that is updated less frequently, or ideally, not updated at all.

Likewise, columns that store large data, such as BLOB columns (text, nvarchar(max), image, etc.), and GUID columns are not ideal for clustered indexes. This is because sorting large values is highly inefficient, and in case of GUID and image columns, doesn't make much sense.

Finally, a clustered index should not be built on a column already used in a unique index.

Mock and Spy

Mocking is a way **to replace a dependency in a unit under test with a stand-in for that dependency**. The stand-in allows the unit under test to be tested without invoking the real dependency.

When Mockito creates a mock, it does so from the *Class* of a Type, not from an actual instance. The mock simply creates **a bare-bones shell instance** of the Class, entirely instrumented to track interactions with it.

On the other hand,**the spy will wrap an existing instance**. It will still behave in the same way as the normal instance; the only difference is that it will also be instrumented to track all the interactions with it.

Size limit of ThreadPool

int numOfCores = Runtime.getRuntime().availableProcessors();

Number of threads = Number of Available Cores \* (1 + Wait time / Service time)

**Waiting time** - is the time spent waiting for IO bound tasks to complete, say waiting for HTTP response from remote service.

(not only IO bound tasks, it could be time waiting to get monitor lock or time when thread is in WAITING/TIMED\_WAITING state)

**Service time** - is the time spent being busy, say processing the HTTP response, marshaling/unmarshaling, any other transformations etc.

Wait time / Service time - this ratio is often called *blocking coefficient*.

A computation-intensive task has a blocking coefficient close to 0, in this case, the number of threads is equal to the number of available cores. If all tasks are computation intensive, then this is all we need. Having more threads will not help.

## The CMS Collector

Following up on the parallel collector is the CMS collector (“*concurrent-mark-sweep*”). This algorithm uses multiple threads (“concurrent”) to scan through the heap (“mark”) for unused objects that can be recycled (“sweep”). This algorithm will enter “stop the world” (STW) mode in two cases: when initializing the initial marking of roots (objects in the old generation that are reachable from thread entry points or static variables) and when the application has changed the state of the heap while the algorithm was running concurrently, forcing it to go back and do some final touches to make sure it has the right objects marked.

The biggest concern when using this collector is encountering **promotion failures** which are instances where a race condition occurs between collecting the young and old generations. If the collector needs to promote young objects to the old generation, but hasn’t had enough time to make space clear it,  it will have to do so first which will result in a full STW collection – the very thing this CMS collector was meant to prevent. To make sure this doesn’t happen you would either increase the size of the old generation (or the entire heap for that matter) or allocate more background threads to the collector for him to compete with the rate of object allocation.

Another downside to this algorithm in comparison to the parallel collector is that it uses more CPU in order to provide the application with higher levels of continuous throughput, by using multiple threads to perform scanning and collection. For most long-running server applications which are adverse to application freezes, that’s usually a good trade off to make. Even so, this algorithm is **not on by default**. You have to specify *XX:+USeParNewGC* to actually enable it. If you’re willing to allocate more CPU resources to avoid application pauses this is the collector you’ll probably want to use, assuming that your heap is less than 4Gb in size.  However, if it’s greater than 4GB, you’ll probably want to use the last algorithm – the G1 Collector.

## **The G1 Collector**

The Garbage first collector (G1) introduced in JDK 7 update 4 was designed to better support heaps larger than 4GB. The G1 collector utilizes multiple background threads to scan through the heap that it divides into regions, spanning from 1MB to 32MB (depending on the size of your heap). G1 collector is geared towards scanning those regions that contain the most garbage objects first, giving it its name (Garbage first). This collector is turned on using the *–XX:+UseG1GC*flag.

This strategy reduced the chance of the heap being depleted before background threads have finished scanning for unused objects, in which case the collector will have to stop the application which will result in a STW collection. The G1 also has another advantage that is that it compacts the heap on-the-go, something the CMS collector only does during full STW collections.

Large heaps have been a fairly contentious area over the past few years with many developers moving away from the single JVM per machine model to more micro-service, componentized architectures with multiple JVMs per machine. This has been driven by many factors including the desire to isolate different application parts, simplifying deployment and avoiding the cost which would usually come with reloading application classes into memory (something which has actually been improved in Java 8).

Even so, one of the biggest drivers to do this when it comes to the JVM stems from the desire to avoid those long “stop the world” pauses (which can take many seconds in a large collection) that occur with large heaps. This has also been accelerated by container technologies like Docker that enable you to deploy multiple apps on the same physical machine with relative ease.

## **Don’t Catch Throwable**

[Throwable](https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html) is the superclass of all exceptions and errors. You can use it in a catch clause, but you should never do it!

If you use Throwable in a catch clause, it will not only catch all exceptions; it will also catch all errors.

Errors are thrown by the JVM to indicate serious problems that are not intended to be handled by an application.

Typical examples for that are the [OutOfMemoryError](https://docs.oracle.com/javase/8/docs/api/java/lang/OutOfMemoryError.html) or the [StackOverflowError](https://docs.oracle.com/javase/8/docs/api/java/lang/StackOverflowError.html). Both are caused by situations that are outside of the control of the application and can’t be handled.

So, better don’t catch a Throwable unless you’re absolutely sure that you’re in an exceptional situation in which you’re able or required to handle an error.

Clean code

Code which is easy to read and understand easily.

Reduce cognitive load

Should be concise and to the point

Should avoid unintuitive names, complex nesting and big code blocks.

Should follow best practices and patterns.

Should be easy to maintainable and extensible.

Places to look

Names

Structure and comments

Functions

Conditionals & error handling

Classes and data structures

## Why Should We Care About Clean Code?

* *Maintainable Codebase*: Any software that we develop has a productive life and during this period will require changes and general maintenance. Clean code **can help develop software that is easy to change and maintain** over time.
* *Easier Troubleshooting*: Software can exhibit unintended behavior due to a variety of internal or external factors. It may often require a quick turnaround in terms of fixes and availability. Software developed with clean coding principles **is easier to troubleshoot for problems**.
* *Faster Onboarding*: Software during its lifetime will see many developers create, update, and maintain it, with developers joining at different points in time. This requires **a quicker onboarding to keep productivity high**, and clean code helps achieve this goal.

## Characteristics of Clean Code

* *Focused*: A piece of **code should be written to solve a specific problem**. It should not do anything strictly not related to solving the given problem. This applies to all levels of abstraction in the codebase like method, class, package, or module.
* *Simple*: This is by far the most important and often ignored characteristic of clean code. The software **design and implementation must be as simple as possible**, which can help us achieve the desired outcomes. Increasing complexity in a codebase makes them error-prone and difficult to read and maintain.
* *Testable*: Clean code, while being simple, must solve the problem at hand. It must be **intuitive and easy to test the codebase, preferably in an automated manner**. This helps establish the baseline behavior of the codebase and makes it easier to change it without breaking anything.

## Clean Coding in Java

### Project Structure

While Java doesn't enforce any project structure, **it's always useful to follow a consistent pattern to organize our source files, tests, configurations, data, and other code artifacts**. Maven, a popular build tool for Java, [prescribes a particular project structure](https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html). While we may not use Maven, it's always nice to stick to a convention.

### Naming Convention

Following **naming conventions can go a long way in making our code readable and hence, maintainable**.

Java [prescribes a set of rules](https://www.oracle.com/technetwork/java/codeconventions-135099.html) to adhere to when it comes to naming anything in Java. A well-formed name does not only help in reading the code, but it also conveys a lot about the intention of the code. Let's take some examples:

* Classes: Class in terms of object-oriented concepts is a blueprint for objects which often represent real-world objects. Hence it's meaningful to use nouns to name classes describing them sufficiently:
* *Variables*: Variables in Java capture the state of the object created from a class. The name of the variable should describe the intent of the variable clearly:
* Methods: Methods in Java are always part of classes and hence generally represent an action on the state of the object created from the class. It's hence [**useful to name methods using verbs**](https://www.baeldung.com/java-pojo-class#javabeans):

### Source File Structure

A source file can contain different elements. While Java **compiler enforces some structure, a large part is fluid**. But adhering to a specific order in which to places elements in a source file can significantly improve code readability. There are multiple popular style-guides to take inspiration from, like one by [Google](https://google.github.io/styleguide/javaguide.html) and another by [Spring](https://github.com/spring-projects/spring-framework/wiki/Code-Style).

A typical ordering of elements in a source file look:

* Package statement
* Import statements
  + All static imports
  + All non-static imports
* Exactly one top-level class
  + Class variables
  + Instance variables
  + Constructors
  + Methods

**Whitespaces**

We all know that it is easier to read and understand short paragraphs compared to a large block of text. It is not very different when it comes to reading code as well. Well-placed and consistent whitespaces and blank lines can enhance the readability of the code.

The idea here is to introduce logical groupings in the code which can help organize thought processes while trying to read it through. There is no one single rule to adopt here but a general set of guidelines and an inherent intention to keep readability at the center of it:

* Two blank lines before starting static blocks, fields, constructors and inner classes
* One blank line after a method signature that is multiline
* A single space separating reserved keywords like if, for, catch from an open parentheses
* A single space separating reserved keywords like else, catch from a closing parentheses

### Indentation

Although quite trivial, almost any developer would vouch for the fact that **a well-indented code is much easier to read and understand**. There is no single convention for code indentation in Java. The key here is to either adopt a popular convention or define a private one and then follow it consistently across the organization.

* A typical best practice is to use four spaces, a unit of indentation. Please note that some guidelines suggest a tab instead of spaces. While there is no absolute best practice here, the key remains consistency!
* Normally, there should be a cap over the line length, but this can be set higher than traditional 80 owing to larger screens developers use today.
* Lastly, since many expressions will not fit into a single line, we must break them consistently:
  + Break method calls after a comma
  + Break expressions before an operator
  + Indent wrapped lines for better readability (we here at Baeldung prefer two spaces)

### Method Parameters

Parameters are essential for methods to work as per specification. But, **a long list of parameters can make it difficult for someone to read and understand the code**. So, where should we draw the line? Let's understand the best practices which may help us:

* Try to restrict the number of parameters a method accepts, three parameters can be one good choice
* Consider [**refactoring**](https://www.baeldung.com/cs/refactoring) the method if it needs more than recommended parameters, typically a long parameter list also indicate that the method may be doing multiple things
* We may consider bundling parameters into custom-types but must be careful not to dump unrelated parameters into a single type
* Finally, while we should use this suggestion to judge the readability of the code, we must not be pedantic about it

**Hardcoding**

Hardcoding values in code can often lead to multiple side effects. For instance, **it can lead to duplication, which makes change more difficult**. It can often lead to undesirable behavior if the values need to be dynamic. In most of the cases, hardcoded values can be refactored in one of the following ways:

* Consider replacing with constants or enums defined within Java
* Or else, replace with constants defined at the class level or in a separate class file
* If possible, replace with values which can be picked from configuration or environme

### Code Comments

[Code comments](https://www.baeldung.com/cs/clean-code-comments) can be **beneficial while reading code to understand the non-trivial aspects**. At the same time, care should be taken to **not include obvious things in the comments**. This can bloat comments making it difficult to read the relevant parts.

Java allows two types of comments: Implementation comments and documentation comments. They have different purposes and different formats, as well.

* Documentation/JavaDoc Comments
  + The audience here is the users of the codebase
  + The details here are typically implementation free, focusing more on the specification
  + Typically useful independent of the codebase
* Implementation/Block Comments
  + The audience here is the developers working on the codebase
  + The details here are implementation-specific
  + Typically useful together with the codebase

So, how should we optimally use them so that they are useful and contextual?

* Comments should only complement a code, if we are not able to understand the code without comments, perhaps we need to refactor it
* We should use block comments rarely, possibly to describe non-trivial design decisions
* We should use JavaDoc comments for most of our classes, interfaces, public and protected methods
* All comments should be well-formed with a proper indentation for readability

**Logging**

Anyone who has ever laid their hands onto production code for debugging has yearned for more logs at some point in time. The **importance of logs can not be over-emphasized in development in general and maintenance in particular**.

There are lots of libraries and frameworks in Java for logging, including SLF4J, Logback. While they make logging pretty trivial in a codebase, care must be given to logging best practices. An otherwise done logging can prove to be a maintenance nightmare instead of any help. Let's go through some of these best practices:

* Avoid excessive logging, think about what information might be of help in troubleshooting
* Choose log levels wisely, we may want to enable log levels selectively on production
* Be very clear and descriptive with contextual data in the log message
* Use external tools for tracing, aggregation, filtering of log messages for faster analytic

# **Why Do Microservices Need an API Gateway?**

[**API Gateway**](https://konghq.com/learning-center/api-gateway)**|**6 MIN READ

#### An API gateway is a software application between a client and a set of backend microservices.

The API Gateway serves as a reverse proxy to accept API calls from the client application, forwarding this traffic to the appropriate service. The API gateway also enforces security and ensures scalability and high availability.

### Centralized access to decentralized microservices

A microservice architecture modularizes the many functions of an application so that each service can focus on implementing a specific business rule. This design pattern makes it easy to develop, test, deploy, and maintain different capabilities of an application. However, this approach also means increased complexity for clients to access those services.

However, an API gateway can handle several API calls simultaneously and route them to different backend services. It can also decompose a single client call into multiple requests to other microservices and aggregate the results when they respond.

### Management and discovery for scalable, distributed services

The elastic nature of the cloud allows for the horizontal scaling of services as demand increases. However, achieving this requires efficient load balancing, easier ways to discover services, and resiliency features like retries and timeouts.

API Gateways can load balancing between replicas of the same service, which allows better resource utilization. Traditional proxies employ basic load balancing algorithms like Random or Round Robin. At the same time, API Gateways can use more sophisticated algorithms like weighted connections, least connections, or even custom implementations that leverage the service registry. These techniques can provide efficient traffic routing. It can do all this while implementing features like retries when a service is down, rerouting requests to healthy service instances, or graceful error handling.

### Abstraction for microservice language and protocol independence

Developers can build different microservices by using another technology stack and protocol. For example, they could develop a service in Java and implement it as a [RESTful](https://en.wikipedia.org/wiki/Representational_state_transfer) API, or write it in Go and implement it with a [gRPC](https://grpc.io/) protocol interface. While this language and protocol independence yields immeasurable flexibility, it also means clients accessing these APIs need to understand and implement different communication protocols.

However, the API Gateway can translate between protocols, allowing clients to call any service with a single protocol. API Gateways effectively hide the service implementations from client applications. As long as the interface definition remains the same, developers can implement changes to the service logic using any technology stack without the client knowing it.

This means a service written in Java can be rewritten in another language like Python and redeployed behind the Gateway. To the client, the interface will remain the same, and it can call the same function with the same parameters and get the same results without realizing the underlying technology and logic have changed.

### Routing to microservices based on deployment strategies

Feature improvements and bug fixes may require special handling at deployment time. Development teams may adopt a deployment strategy like canary release or blue/green deployments. To do so, they need to ensure that they’ve optimized their CI/CD pipelines for these strategies.

API gateways can be configured to route traffic following those strategies, switching or routing traffic between old and new service versions accordingly.

### Traffic control to prevent overloading of resources

Internet-facing services are targets for malicious attacks and exploitation. They need to handle sudden surges in requests from valid users and bad actors. This means request throttling and blacklist capabilities are necessary to keep systems reliable and secure.

API gateways are an effective barrier against Distributed Denial of Service (DDoS) attacks—throttling the number of requests made to affected services, so the service isn’t overwhelmed and protecting it against becoming unresponsive.

## **Common Features and Benefits of API Gateways**

An API Gateway is located at the outer edge of your microservices and acts as a proxy to manage all ingress traffic. From here, the API Gateway handles several features.

### Authentication, Authorization, and Audit

Every request made to your service needs to be authenticated to ensure only valid users have access. An API Gateway authenticates all traffic before routing it to the called service. The API Gateway can perform the authentication itself or use external authentication providers for that task.

Authorization is the process that ensures your users have the necessary permissions to request the action from a service. As with authentication, an API Gateway can use its access control lists (ACL) for this authorization or fetch authorization information from an external service provider.

Since all calls must pass through the API Gateway, this provides system administrators with an invaluable audit trail of who, when, and what—helping to troubleshoot errors, monitor performance, or address security incidents.

### Traffic Management

Traffic management is a broad category core to any API Gateway. The most basic capability offered by the Gateway is a virtual endpoint for API clients, which remains constant over time. This minimizes the need for disruptive change with clients when new versions of services are deployed, leading to an overall better developer experience.

With cloud-native orchestration platforms like Kubernetes, it’s easy to scale services across multiple nodes within a cluster and across clusters. However, this also means ephemeral services with locations (IP address and port) constantly changing. An API Gateway can provide a service registry that keeps track of available instances of each service.

As services integrate with the API Gateway, they can self-register with the service registry and report their availability. It’s also possible to use third-party registration services. With a service registry, the Gateway knows where to route the client request at any time.

### Threat Protection

An API Gateway can protect your services from spikes or DDoS attacks. DevOps teams can smooth and throttle traffic by implementing failure patterns like circuit breaking, protecting services from becoming overwhelmed with requests.

By inspecting incoming requests, an API Gateway can prevent attacks. It can integrate with other applications like Security Information and Event Management (SIEM) systems or fraud detection systems to decide if a request is good or bad. Some of its tasks may include:

* **API firewalling**: mitigates application-level threats like Cross-Site Request Forgery (CSRF) or SQL injection (some API gateways can integrate with WAFs) as well as detects and blocks threats according to the [OWASP Top Ten](https://owasp.org/www-project-top-ten) or by blacklisting bots.
* **Content validation**: ensures each request has the correctly formatted input parameter values for the intended service and adheres to the API’s published interface.
* **Integrity validation**: confirms that requests have not been tampered with and that encrypted data in the request or response body remains confidential.

### Monitoring and Observability

An API gateway can collect metrics, logs, and traces about all inbound traffic passing through it. For example, based on this data, it can track request and response times, traffic patterns, and error trends. It can log important messages in either direction to help troubleshoot errors and improve the observability of a system. The location of the API gateway in the microservice architecture also allows it to add tracing information like correlation ID, enabling visibility of requests through the system.

When API gateways integrate with a unified observability solution, the result is a single pane of glass to check a distributed application’s overall health and performance. An API gateway’s proactive alerting rules can report on the health of services before they become unresponsive.

## **Caveats of using API Gateways**

Though API gateways bring many benefits, there are caveats that organizations should bear in mind.

First, an API Gateway is a new software component that a DevOps (or APIOps) team needs to deploy, configure, and maintain. This means there’s an extra moving part in the architecture, which means additional cost (money, time, effort) and a learning curve.

Adding an extra component between clients and the microservices means an extra network hop for requests. Depending on the network speed, this can have an adverse effect on the service response time.

As the entry point for all your services, an API Gateway could potentially become a single point of failure and take your entire system offline when it becomes unavailable. For this reason, API gateways are often configured to be highly availability (HA).

Similarly, an API Gateway can be a single point of attack for hackers and malicious players. A compromised API Gateway can expose your microservices to further attacks.

When selecting an API Gateway, it’s important to remember that there are pros and cons to all architectural decisions. Likewise, there are also advantages and disadvantages to consider with specific tools. There’s no silver bullet architecture or tool that can provide an extremely effective solution without caveats. An API Gateway is no exception.

**Microservice Principles**

* Scalability
* Availability
* Resiliency
* Flexibility
* Independent, autonomous
* Decentralized governance
* Failure isolation
* Auto-Provisioning
* Continuous delivery through DevOps

Event Source

Events are stored in event source like DB.

Aggregate is stored as a sequence of events

For every state changes, a new event is created and added to the event store

Event store maintains a complete history of the aggregate, so we can reconstruct the historical state of the object.

Why we need default and static methods in interfacers?

In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down.

Default interface methods are an efficient way to deal with this issue. They **allow us to add new methods to an interface that are automatically available in the implementations**. Therefore, we don't need to modify the implementing classes.

In this way, **backward compatibility is neatly preserved** without having to refactor the implementers.

We can solve diamond problem with default methods and interfaces.

The idea behind static interface methods is to provide a simple mechanism that allows us to **increase the degree of**[**cohesion**](https://en.wikipedia.org/wiki/Cohesion_(computer_science)) of a design by putting together related methods in one single place without having to create an object.

**The same can pretty much be done with abstract classes.** The main difference is that **abstract classes can have constructors, state, and behavior**.

Java interface static method is similar to default method except that we can’t override them in the implementation classes. This feature helps us in avoiding undesired results in case of poor implementation in implementation classes.

Can we instantiate abstract classes?

We cannot instantiate an abstract class in Java because it is abstract, it is not complete, hence it cannot be used.

But we can have an anonymous sub class which can be instantiated.

## Reason Why Wait, Notify, and NotifyAll are in Object Class.

Here are some thoughts on why they should not be in Thread class which make sense to me :

1. Wait and notify is not just normal methods or synchronization utility, more than that they are **communication mechanism between two threads in Java**. And Object class is the correct place to make them available for every object if this mechanism is not available via any java keyword like synchronized.

Remember synchronized and wait to notify are two different areas and don’t confuse that they are the same or related. Synchronized is to provide mutual exclusion and ensuring [thread safety of Java class](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html) like race condition while wait and notify are communication mechanism between two thread.  
  
2. **Locks are made available on per Object basis**, which is another reason wait and notify is declared in Object

3.  In Java in order to enter a critical section of code, Threads needs lock and they wait for lock, they don't know which threads hold lock instead they just know the lock is hold by some thread and they should wait for lock instead of knowing which thread is inside the synchronized block and asking them to release lock. this analogy fits with wait and notify being on object class rather than a thread in Java.

Composite Primary Key

A composite primary key, also called a composite key, is a combination of two or more columns to form a primary key for a table.

In JPA, we have two options to define the composite keys: the *@IdClass* and *@EmbeddedId* annotations.

**In order to define the composite primary keys, we should follow some rules:**

* The composite primary key class must be public.
* It must have a no-arg constructor.
* It must define the *equals()* and *hashCode()* methods.
* It must be *Serializable.*

# **Spring Data JPA Projections**

When using [Spring Data JPA](https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa) to implement the persistence layer, the repository typically returns one or more instances of the root class. However, more often than not, we don't need all the properties of the returned objects.

In such cases, we might want to retrieve data as objects of customized types. **These types reflect partial views of the root class, containing only the properties we care about.** This is where projections come in handy.

JPA Example type

Spring Reactive

New Programming paradigm

Asynchronous and non-blocking

Functional style code

Data flow as event stream

Back pressure on data streams

Spring Batch

Components – JobLauncher – Job – step – ( item reader, processor, writer) and jobRepository

In the config class, we will have jobbuilderfactory and stepbuilderfactory

Parallel Processing – Partitioning – multiple threads to process range of data

Fault Tolerance – using skip policy

AWS

AWS Lambda

AWS Lambda is a serverless, event-driven compute service that lets you run code for virtually any type of application or backend service without provisioning or managing servers. You can trigger Lambda from over 200 AWS services and software as a service (SaaS) applications, and only pay for what you use.

Elastic BeanStalk

AWS Elastic Beanstalk is an orchestration service offered by Amazon Web Services for deploying applications which orchestrates various AWS services, including EC2, S3, Simple Notification Service, CloudWatch, autoscaling, and Elastic Load Balancers

Elastic Beanstalk is a service for deploying and scaling web applications and services. Upload your code and Elastic Beanstalk automatically handles the deployment—from capacity provisioning, load balancing, and auto scaling to application health monitoring.

Difference between Lambda and beanstalk

Elastic beanstalk and lambda are very different though some of the features may look similar. At high level, elastic beanstalk deploys a long running application whereas lambda deploys short running code function

1. Lambda can at maximum run for 15 minutes, whereas EB can run continuously. Generally, we deploy websites/apps on EB whereas lambda are generally used for triggered functionality like processing image when image gets uploaded to S3.
2. Lambda can only handle one request at a time whereas number of concurrent requests EB can handle depends on your underlying infrastructure. So, if you are having say 100 requests, 100 lambdas will be created whereas these 100 requests can be handled by one underlying EC2 instance in EB
3. Lambda is serverless (underlying infra is entirely abstracted from developer). Whereas EB is automation over infra provisioning. You can still see your EC2 instances, load balancer, auto scaling group etc. in your AWS console. You can even ssh/rdp to your instance and change running services. AWS EB allows you also to have your custom AMIs.
4. Lambda is having issue of cold starts as in lambda, infra needs to be provisioned on demand by AWS, whereas in EB, you generally have EC2 instances already provisioned to handle your requests.

Kubernetes

[Kubernetes](https://kubernetes.io/docs/concepts/overview/), also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.

#### [**Automated rollouts and rollbacks**](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)

Kubernetes progressively rolls out changes to your application or its configuration, while monitoring application health to ensure it doesn't kill all your instances at the same time. If something goes wrong, Kubernetes will rollback the change for you. Take advantage of a growing ecosystem of deployment solutions.

#### [**Service discovery and load balancing**](https://kubernetes.io/docs/concepts/services-networking/service/)

No need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

#### [**Storage orchestration**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

Automatically mount the storage system of your choice, whether from local storage, a public cloud provider such as [AWS](https://aws.amazon.com/products/storage/) or [GCP](https://cloud.google.com/storage/), or a network storage system such as NFS, iSCSI, Ceph, Cinder.

#### [**Self-healing**](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#how-a-replicaset-works)

Restarts containers that fail, replaces and reschedules containers when nodes die, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.

#### [**Secret and configuration management**](https://kubernetes.io/docs/concepts/configuration/secret/)

Deploy and update secrets and application configuration without rebuilding your image and without exposing secrets in your stack configuration.

#### [**Automatic bin packing**](https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/)

Automatically places containers based on their resource requirements and other constraints, while not sacrificing availability. Mix critical and best-effort workloads in order to drive up utilization and save even more resources.

#### [**Batch execution**](https://kubernetes.io/docs/concepts/workloads/controllers/job/)

In addition to services, Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.

#### [**Horizontal scaling**](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/)

Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage.

#### [**IPv4/IPv6 dual-stack**](https://kubernetes.io/docs/concepts/services-networking/dual-stack/)

Allocation of IPv4 and IPv6 addresses to Pods and Services

#### [**Designed for extensibility**](https://kubernetes.io/docs/concepts/extend-kubernetes/)

Add features to your Kubernetes cluster without changing upstream source code.

NoSQL Databases

NoSQL databases (aka "not only SQL") are non-tabular databases and store data differently than relational tables. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide flexible schemas and scale easily with large amounts of data and high user loads.

### Types of NoSQL databases

Over time, four major [types of NoSQL databases](https://www.mongodb.com/scale/types-of-nosql-databases) emerged: document databases, [key-value databases](https://www.mongodb.com/databases/key-value-database), wide-column stores, and graph databases.

* **Document databases** store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.
* **Key-value databases** are a simpler type of database where each item contains keys and values.
* **Wide-column stores** store data in tables, rows, and dynamic columns.
* **Graph databases** store data in nodes and edges. Nodes typically store information about people, places, and things, while edges store information about the relationships between the nodes.

**Hashcode and Equals**

If two objects are equal according to their equals() method, they must also have the same hash code. If two objects have the same hash code, they do NOT have to be equal too.

Here is the actual wording from the java.util.Object documentation:

* If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
* It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

It is true, that if two objects don't have the same hash then they are not equal. However, hashing is not a way to check equality - so it is wildly incorrect to say that it is a faster way to check equality.

Also, it is also wildly incorrect to say the hashCode function is an efficient way to do anything. This is all up to implementation, but the default implementation for hashCode of a string is very inefficient as the String gets large. It will perform a calculation based on each char of the String, so if you are using large Strings as keys, then this becomes very inefficient; moreso if you have a large number of buckets.

In a Map (HashSet uses a HashMap internally), there are buckets and in each bucket is a linked list. Java uses the hashCode() function to find out which bucket it belongs in (it actually will modify the hash, depending on how many buckets exist). Since two objects may share the same hash, it will iterate through the linked list sequentially next, checking the equals() method to see if the object is a duplicate. Per the java.util.Set documenation:

A collection that contains no duplicate elements.

So, if its hashCode() leads it to a bucket, in which that bucket contains an Object where the .equals() evaluates to true, then the previous Object is overwritten with the new Object. You can probably view here for more information: [How does a Java HashMap handle different objects with the same hash code?](https://stackoverflow.com/questions/6493605/how-does-a-hashmap-work-in-java)

Generally speaking though, it is good practice that if you overwrite the hashCode function, you also overwrite the equals function (if I'm not mistaken, this breaks the contract if you choose not to).