***Tell me a little about the history of Java as a language?***

Java was created by James Gosling at Sun Microsystem (later acquired by Oracle) and first appeared in 1991. It is one of the most widely used **general purpose concurrent, class-based and object-oriented** programming languages.

**Question # 2**

***What is meant by the statement Java is platform independent?***

Java works on the principle of ***write once and run anywhere***. Once a Java program is written, it gets compiled into what is known as the ***byte code***, which can then be run on any Java Virtual Machine or JVM for short.

Compilation to bytecode is the magic behind Java's interoperability. Different operating systems and hardware architectures have JVMs custom designed for themselves and all JVMs can run the same bytecode. Therefore, if you write a Java program on Linux, it will run seamlessly on a JVM designed for Windows operating system, making code agnostic to the underlying hardware and OS.

Java's Portability

**Question # 3**

***What is meant by the Java platform?***

Java is both a programming language as well as a software platform. The platform consists of:

* Java Virtual Machine (JVM)
* Java Application Programming Interface (Java API)

Oracle's [Java Standard Edition](https://www.oracle.com/technetwork/java/javase/overview/index.html) is an example of the Java platform, which comes in different versions. Similarly, [Java Platform, Enterprise Edition](https://www.oracle.com/technetwork/java/javaee/overview/index.html) is another Java platform targeting enterprise. The platform includes the *Java Language Specification* and the *Java Virtual Machine Specification*. Oracle's corresponding JDKs are implementations of the platforms that can be downloaded and used.

Java Platform

**Question # 4**

***What is the Java language specification?***

In general, **language specification is an official document that gives detailed description of the syntax and semantics of a particular language.** Java publishes Java language specifications and the virtual machine specifications for various editions. Note the following:

* Every specification may not necessarily have an implementation in the JDK (for example JPA, JAX-RS).
* Third party vendors can also participate in implementing the specification.
* An actual implementation of the specification can be a superset of the original specification i.e. it can offer more than the specification but shouldn't contradict the specification.

For example, here is the [*link*](https://docs.oracle.com/javase/specs/) to the Java SE specifications.

**Question # 5**

***What is the Java API?***

Java API is a **large collection of ready-made software components that provide many useful capabilities that comes prepackaged with the Java development kit.** It is grouped into libraries of related classes and interfaces known as packages.

These APIs enable Java programs to access the local file system, the network and other basic functionality which one would otherwise have had to program oneself. These APIs cut down development time.

The standard Java APIs come bundled with the Java Runtime Environment (JRE) or with the Java SDK which also includes a JRE.

**Question # 6**

***What is the Java Runtime Environment?***

The Java Runtime Environment (JRE) includes the **Java Virtual Machine and the standard Java APIs (core classes and supporting files.)**. The JRE contains just enough to execute a Java application, but not to compile it.

**Question # 7**

***What is the Java Development Kit or JDK?***

The Java Software Development Kit (Java SDK) is **JRE plus the Java compiler, and a set of other tools.**

Java Development Kit (JDK) consists of Java Runtime Environment (JRE) along with tools to compile and debug Java code for developing Java applications. JRE consists of Java platform libraries, Java Virtual Machine (JVM), Java Plugin and Java Web Start to run Java applications. JRE as a stand-alone does not contain compilers and debugging tools.

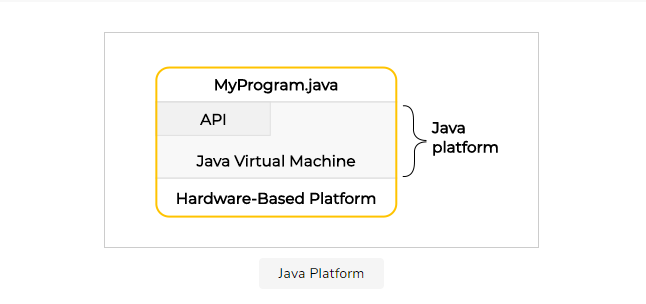
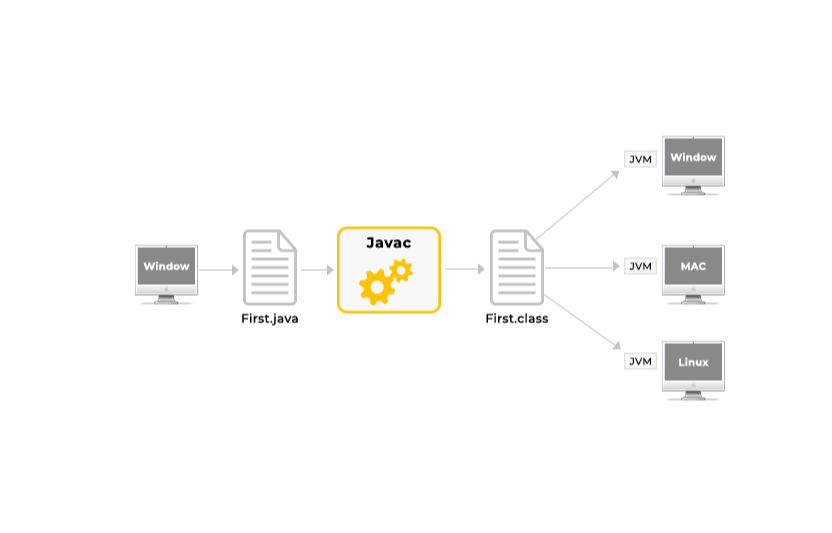
If you need to develop Java programs you need the full Java SDK. The JRE is not enough for program development. Only the full Java SDK contains the Java compiler which turns your **.java** source files into byte code **.class** files.

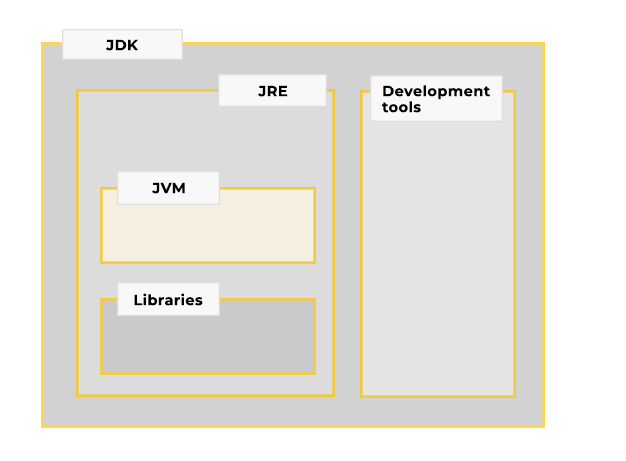
JDK vs JRE vs JVM

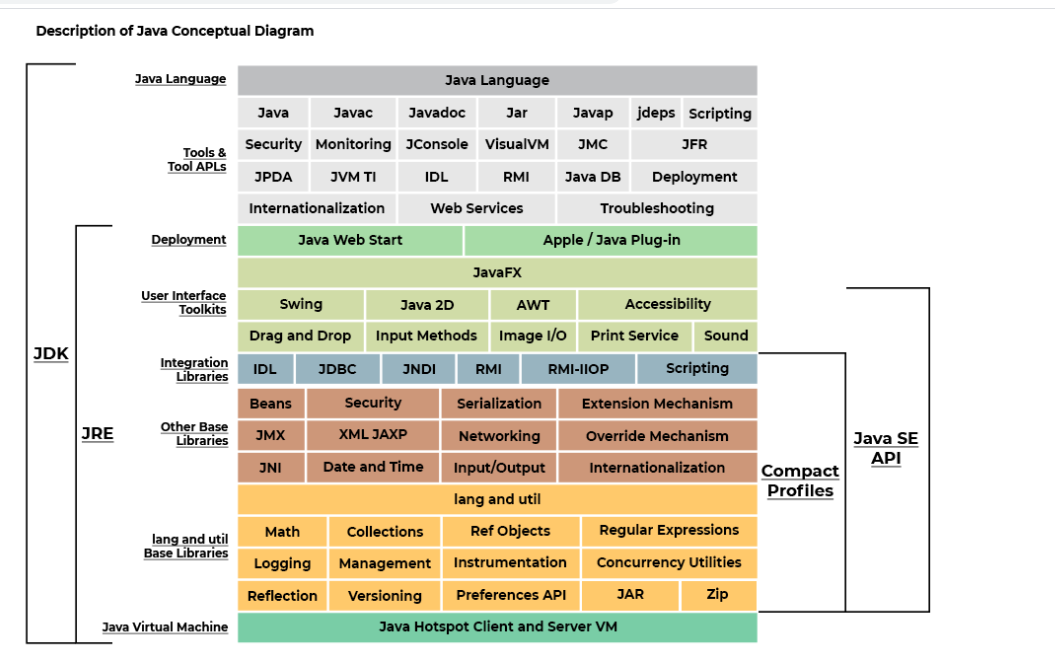
**Question # 8**

***Does Java's platform independence affect its performance?***

As a platform-independent environment, the Java platform can be a bit slower than native code. **Converting byte code to machine code incurs a performance penalty. Additionally, Java is a memory managed language which requires cleaning up of the memory space periodically by the platform, resulting in reduced application throughput.** However, advancements in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.







***What is the Java Virtual Machine or JVM?***

**The Java virtual machine is an implementation of a specification, detailing the behavior expected of a JVM.** Any implementation that conforms to the JVM specification should be able to run code compiled into Java bytecode irrespective of the language in which the code was originally written. The Java Virtual Machine is implemented for several different operating systems, like Windows, Mac OS, Linux, IBM mainframes, Solaris etc. Thus, if your Java program can run on a Java Virtual Machine on Windows, it can normally also run on a Java Virtual Machine on Mac OS or Linux. Note that the JVM is a program itself that can be invoked on the command line and instructed to execute a file containing java bytecode.

In the Java programming language, all source code is first written in plain text files ending with the **.java** extension. Those source files are then compiled into **.class** files by the **javac** compiler. A **.class** file does not contain code that is native to your processor; it instead contains bytecodes — the machine language of the Java Virtual Machine. The java launcher tool then runs your application with an instance of the Java Virtual Machine.

The JVM is by definition a virtual machine or an abstract computer, i. e. a software machine that simulates what a real machine does. Like a real machine, it has an instruction set, a virtual computer architecture and an execution model. It is capable of running code written with this virtual instruction set, pretty much like a real machine can run machine code.

**In practice, JRE is the implementation of the Java Virtual Machine. The JRE contains the JVM and java binaries and other classes to execute any program successfully.**

**Question # 2**

***Describe JVM architecture?***

The Java Virtual Machine consists of three components:

* **Class Loader Subsystem:** The part of a Java virtual machine implementation that takes care of finding and loading types is called the class loader subsystem. The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. (See a more detailed explanation under the class loader section.)
* **Runtime Data Area:** The memory areas allocated by the JVM are called Runtime Data Area. These consist of method area, heap area, stack, pc registers and native stack. (See a more detailed explanation under the memory management section.)
* **Execution Engine:** The execution is responsible for the actual execution of the bytecode. It consists of three components: interpreter, just-in-time compiler and the garbage collector.

JVM Architecture

**Question # 3**

***If we launch two Java programs on the same machine, how many instances of JVM would be created?***

A runtime instance of the Java virtual machine runs a single Java application. When a Java application starts, a runtime instance is born. When the application completes, the instance dies. **If you start three Java applications at the same time, on the same computer, using the same concrete implementation, you'll get three Java virtual machine instances. Each Java application runs inside its own Java virtual machine.**

**Question # 4**

***Can you give a few examples of JVM implementations?***

A large amount of Java development work takes place on Windows, Solaris, Linux, and FreeBSD, primarily with the Oracle JVMs. In fact, Oracle's **Hotspot** implementation of JVM is used as a reference. Additionally, there are 32 and 64-bit varities of JVM. Some implementations of the JVM have been discontinued by their sponsors and aren't active. Some active implementations of the JVM are listed below:

* [Amazon's Corretto JVM implementation](https://aws.amazon.com/corretto/)
* [CACAO JVM implementation](http://www.cacaojvm.org/)
* [IBM's OpenJ9 JVM implementation](https://www.eclipse.org/openj9/)
* [Azul System's Zulu JVM implementation](https://www.azul.com/downloads/zulu/)

[JRockit](https://en.wikipedia.org/wiki/JRockit) is a discontinued implementation of the JVM. Twitter and SAP also have their implementations of the JVM specifications.

**Question # 5**

***What are JVM languages?***

JVM runs bytecode. The Java compiler converts code written in the Java language to bytecode. Similarly, other languages can take advantage of the Java platform if they have a compiler to convert code written in their respective languages to bytecode. For instance, [**Jython**](http://www.jython.org/) is an implementation of the Python language for the Java platform and a program written in Jython can run on any Java platform. [**JRuby**](https://www.jruby.org/) is another example, which is an implementation of the Ruby programming language atop the JVM. **Such languages that can run on the Java platform are called JVM languages.** Here's a [list](https://en.wikipedia.org/wiki/List_of_JVM_languages) on Wikipedia.

**Question # 6**

***What is Java Hotspot?***

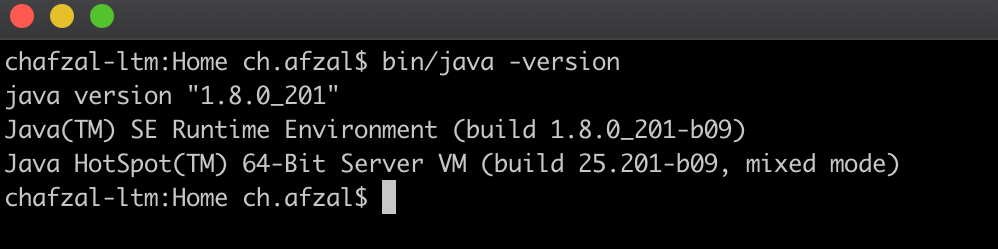
HotSpot is one of the most popular implementations of the JVM concept. It was originally developed by Sun and now is owned by Oracle. There are two editions available:

* Oracle Hotspot
* OpenJDK Hotspot

Sun open sourced and donated the Hotspot JVM source code which became the OpenJDK project. Implementations by OpenJDK serve as reference implementation.

There's not too many differences between the two as Oracle's Hotspot is based on OpenJDK's Hotspot project and comes with additional bells and whistles for paying customers. **The JVM implementation is called HotSpot because it continually analyzes the program's performance for hot spots of code (code paths which are executed repeatedly). The repeating code paths are compiled into very highly optimized native machine code for faster execution.**

You can check the installed JVM version on your machine using the **java -version** command. On my machine the output appears below:



The output shows that the version of the JRE as well as the JVM Hotspot version. Note separate Hotspot JVM implementations exist for server and client environments. In the above screenshot, the JVM identifies itself as the *Server VM*. The Java HotSpot Client VM has been specially tuned to reduce application start-up time and memory footprint, making it particularly well suited for client environments. The Java HotSpot Server VM is similar to the Java HotSpot Client VM, except that it has been specially tuned to maximize peak operating speed. It is intended for running long-running server applications, for which having the fastest possible operating speed is generally more important than having the fastest possible start-up time. These two solutions share the Java HotSpot runtime environment code base but use different compilers that are suited to the distinctly unique performance characteristics of clients and servers. On a 64-bit capable JDK, only the Java Hotspot Server VM is supported.

**Question # 7**

***What is the execution engine?***

The execution engine is responsible for executing bytecode. The execution engine is one part of the virtual machines that can vary in different JVM implementations. The most used JVMs have three components of the execution engine:

* Interpreter
* Just in Time Compiler
* Garbage Collector

The simplest kind of execution engine just interprets the bytecodes one at a time.

Another kind of execution engine, one that is faster but requires more memory, comes with a just-in-time compiler. In this scheme, the bytecodes of a method are compiled to native machine code the first time the method is invoked. The native machine code for the method is then cached, so it can be re-used the next time that same method is invoked.

The third type of execution engine is an adaptive optimizer. In this approach, the virtual machine starts by interpreting bytecodes, but monitors the activity of the running program and identifies the most heavily used areas of code. As the program runs, the virtual machine compiles to native and optimizes just these heavily used areas. The rest of the of code, which is not heavily used, remains as bytecodes which the virtual machine continues to interpret.

The garbage collector is discussed at length in the memory management section of the course.

**Question # 8**

***What is the Java interpreter?***

**The Java Interpreter can be thought of as a translator that converts Java bytecode into native machine code. The translation of bytecodes to native machine code is done line by line.**

The Java interpreter is actually a part of the JVM. For each hardware architecture, a different Java bytecode interpreter is needed. When a computer has a Java bytecode interpreter, it can run any Java bytecode program, and the same program can be run on any computer that has such an interpreter.

When Java bytecode is executed by an interpreter, the execution will always be slower than the execution of the same program compiled into native machine language. This problem is mitigated by just-in-time (JIT) compilers for executing Java bytecode.

Java Interpreter

**Question # 9**

***Explain the working of the JIT compiler?***

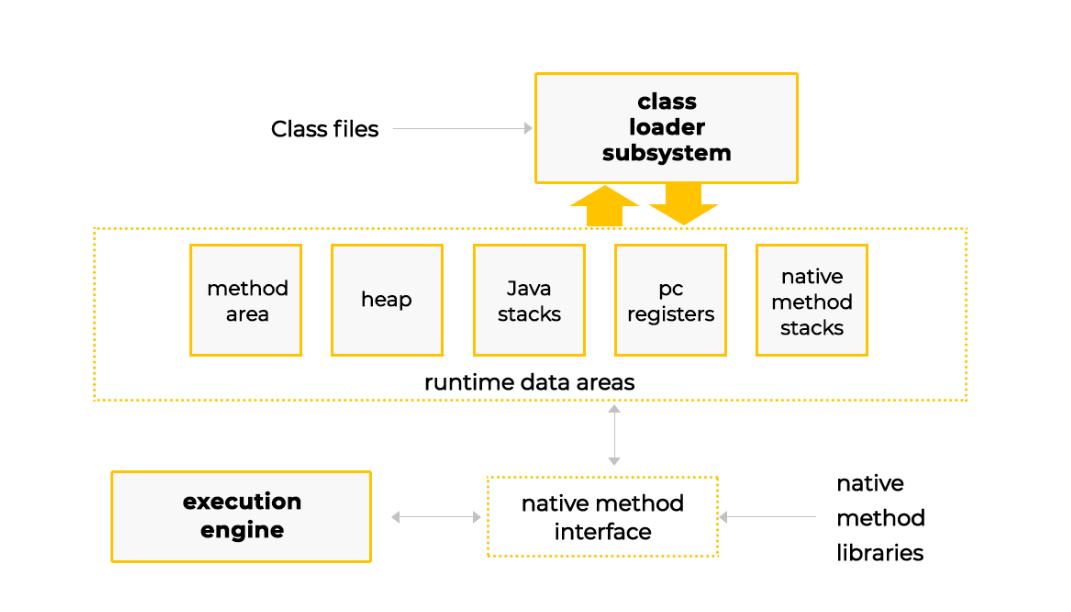
**A JIT compiler runs after the program has started and compiles the code (usually bytecode or some kind of VM instructions) on the fly (or just-in-time, as it's called) into a form that's usually faster, typically the host CPU's native instruction set. A JIT compiler has access to dynamic runtime information whereas a standard compiler doesn't and can make better optimizations like inlining functions that are used frequently. This is in contrast to a traditional compiler that compiles all the code to machine language before the program is run for the first time.**

Java programs consist of classes, which contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures. **The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time.** The exact behavior of the JIT compiler is hard to predict and documentation is scarce, however, the general theme on which JITs work is presented here. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if the compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.

JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance. In practice, methods are not compiled the first time they are called. For each method, the JVM maintains an invocation count, which starts at a predefined compilation threshold value and is decremented every time the method is called. When the invocation count reaches zero, a just-in-time compilation for the method is triggered. Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all. You can disable the JIT compiler, in which case the entire Java program will be interpreted.

JIT is not actually part of the JVM standard, it is, nonetheless, an essential component of Java.

Usually JIT compilers employ a sophisticated, low-cost, sampling-based technique to identify which functions merit optimization. A "sampler thread" wakes up at periodic intervals and checks the status of several application threads. It identifies what each thread is executing and notes some of the execution history. This information is tracked for all the methods and when it is perceived that a method is experiencing heavy use or in other words, becomes *hot* — that method is slated for optimization. Usually, a flurry of such optimization opportunities occurs in the application’s early run stages, with the rate slowing down as execution continues.



#### Question # 1

***What is Java SE?***

The *SE* stands for standard edition. We know that the Java platform is a suite of programs that facilitate developing and running programs written in the Java programming language. All Java platforms consist of a Java Virtual Machine (JVM) and an application programming interface (API). Don't confuse the Java programming language with the Java platform. The programming language is *one* part of the Java platform. Since Java is independent of the underlying OS and hardware, there exist different versions of the platform that target specific devices and use-cases.

A Java platform edition is defined by a specification that describes APIs and their interactions. For instance, the Java SE edition specifications can be seen [here](https://docs.oracle.com/javase/specs/). Any vendor is free to implement the specification. For instance, there's one called [Zulu](https://www.azul.com/zulu-enterprise-landing/), which is Java SE 11 compliant. Similarly, Oracle has their own implementation that can be downloaded from [here](https://www.oracle.com/technetwork/java/javase/downloads/index.html).

As a developer or a user, you'll be required to download either a JRE or JDK belonging to the edition you intend to use. The Java Standard Edition contains the basic Java APIs for standalone desktop and command line applications and can be used on desktop PCs, servers, and similar devices.

#### Question # 2

***What is OpenJDK?***

Open Java development kit is a free and open-source implementation of the Java Platform, Standard Edition (Java SE), and the base for the Oracle JDK. OpenJDK is the official reference implementation for Java Standard Edition from Java SE 7. In fact, Oracle JDK’s builds from OpenJDK source code and so do other vendors. There is no major technical difference between Oracle JDK and OpenJDK. Most of the vendors of JDK are written on top of OpenJDK by doing a few tweaks to replace either licensed proprietary parts or more high-performance items that only work on specific OS components.

The OpenJDK project came into being as a result of Sun open sourcing Java

#### Question # 3

***What is Java EE?***

Java EE is a specification describing a collection of technologies and APIs for the Java platform designed to support **enterprise** applications which can generally be classified as large-scale, multi-tiered, distributed, transactional, and highly-available applications for mission-critical business requirements. Technologies implementing and complying with Java EE specification include for instance GlassFish or IBM's WebSphere.

These implementations are the so-called *Java EE Containers*. When you hear people saying that "GlassFish is a Java EE 1.8 implementation" they mean that GlassFish (a Java program written using the Java Standard Edition classes) provides all the features that the Java EE 8 family of specifications define. A Java EE application needs a Java EE compatible server (such as GlassFish or Websphere) in addition to the JVM to run. Note, the Java EE platform is built on top of the Java SE platform and runs on the Java SE runtime i.e. Java SE JRE.

#### Question # 4

***What is Java ME?***

**The *ME* stands for micro edition. The Java ME platform provides an API and a small-footprint virtual machine for running Java programming language applications on small devices, like mobile phones.** The API is a subset of the Java SE API, along with special class libraries useful for small device application development. Java ME applications are often clients of Java EE platform services.

#### Question # 5

***What is Java FX?***

JavaFX is a platform for creating rich internet applications using a lightweight user-interface API. JavaFX applications use hardware-accelerated graphics and media engines to take advantage of higher-performance clients and a modern look-and-feel as well as high-level APIs for connecting to networked data sources. JavaFX applications may be clients of Java EE platform services. It's speculated that the *FX* in the name refers to special "EFF-ECTS".

#### Question # 6

***What is JCP?***

**The Java Community Process (JCP), established in 1998, is a formalized mechanism that allows interested parties to develop standard technical specifications for Java technology.** Anyone can become a JCP Member by filling a form available at the JCP website. JCP membership for organizations and commercial entities requires annual fees but is free for individuals.

#### Question # 7

***What is JSR?***

The JCP involves the use of **Java Specification Requests (JSRs) – the formal documents that describe proposed specifications and technologies for adding to the Java platform.** Formal public reviews of JSRs take place before a JSR becomes final and the JCP Executive Committee votes on it. A final JSR provides a reference implementation that is a free implementation of the technology in source code form and a Technology Compatibility Kit to verify the API specification.

#### Question # 8

***What is TCK?***

**A Technology Compatibility Kit (TCK) is a suite of tests that nominally checks if a claimed implementation of a Java Specification Request (JSR) is compliant.** The Technology Compatibility Kit for a particular Java platform is called Java Compatibility Kit (JCK). It is an extensive test suite used by Oracle and licensees to ensure compatible implementations of the platform.

Under the JCP process, release of a new or revised technology specification must contain three primary components:

* **Specification:** A written specification of technology. There are different kinds of specifications, for example platform editions, profiles and optional packages.
* **Reference Implementation (RI):** The prototype or proof of concept implementation of the specification. The RI is required to pass the TCK.
* **Technology Compatibility Kit (TCK):** A test kit that Java technology implementors can use to ensure that their work is conformant with the technology specification. The TCK must test all aspects of a specification that impact how conformant an implementation of that specification would be, such as the public API and all elements of the specification. A vendor's implementation of a specification is only considered conformant if the implementation passes the TCK.

#### Question # 1

***What files do we write Java code in?***

Java code is written in files with the extension **.java**

#### Question # 2

***What are .class files?***

***or***

***What is Java bytecode?***

The java compiler compiles the **.java** files into **.class** files. The **.class** files contain what is known as the ***java bytecode***. Then the bytecode can be run by the JVM. **Java bytecode is the instruction set of the Java virtual machine.**

#### Question # 3

***What are .jar files?***

A JAR (Java Archive) is a package file format used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution. A jar file is built on the ZIP format and typically has a .jar file extension. The jar tool can be used to create **.jar** file.

#### Question # 4

***What is a fat jar?***

***Fat jar,* or *uber jar,* is a jar which contains all project class files and classes from all the libraries, on which the project depends.**

#### Question # 5

***What is JCL?***

JCL stands for Java Class Library. It is a set of dynamically loadable libraries that Java applications can call at run time. JCL includes fundamental classes such as **java.lang.String**, **java.lang.Thread**, **java.util.ArrayList** and all the other classes from Java API. Almost all of JCL is stored in a single Java archive file called **rt.jar** (classes.jar on mac) runtime jar for short, which is provided with JRE and JDK distributions. Starting from Java 9 the **rt.jar** file has been broken into several smaller modules.

#### Question # 6

***What is a .hprof file?***

HProf is a tool built into JDK for profiling the CPU and heap usage within a JVM. A Java process crash may produce an **.hprof** file containing a heap dump of the process at the time of the failure.

#### Commonly Used Java Tools

|  |  |
| --- | --- |
| Name | Used For |
| **java** | Launches a java application. |
| **javac** | The Java programming language compiler, javac, reads source files written in the Java programming language, and compiles them into bytecode class files. |
| **javadoc** | Javadoc is a tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, interfaces, constructors, methods, and fields. |
| **jar** | combines multiple files into a single JAR archive file. The combined files could include **.class** files, image and sound files etc. |
| **javap** | The javap command disassembles one or more class files. For instance, it can be used to view public, protected and private members of a class. |
| **JPDA** | The Java Platform Debugger Architecture is a collection of APIs to debug Java code. |
| **jConsole** | JConsole is a graphical monitoring tool to monitor Java Virtual Machine and Java applications both on a local or remote machine. |
| **Java VisualVM** | VisualVM is a tool that provides a visual interface for viewing detailed information about Java applications while they are running on a Java Virtual Machine. It helps in troubleshooting and profiling these applications. |
| **jcmd** | It is a comprehensive JDK tool for troubleshooting JVM applications. It can be used to get heap dumps, stack traces, retrieving garbage collector statistics etc. |
| **jmap** | map is a tool to print statistics about the memory in a running JVM. We can use it for local or remote processes. It can also be used to generate heap dumps. |
| **jstack** | This tool can be used to retrieve stack traces of all Java threads running within a target JVM. |
| **jstat** | This tool is used to monitor JVM statistics, which can also be observed using the visual tools. |
| **jinfo** | This tool prints Java configuration information for a given Java process or core file or a remote debug server. Configuration information includes Java System properties and Java virtual machine command line flags. |

# sJava Tools & Files

Common file types and tools used in Java development.

#### Question # 1

***What files do we write Java code in?***

Java code is written in files with the extension **.java**

#### Question # 2

***What are .class files?***

***or***

***What is Java bytecode?***

The java compiler compiles the **.java** files into **.class** files. The **.class** files contain what is known as the ***java bytecode***. Then the bytecode can be run by the JVM. **Java bytecode is the instruction set of the Java virtual machine.**

#### Question # 3

***What are .jar files?***

A JAR (Java Archive) is a package file format used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution. A jar file is built on the ZIP format and typically has a .jar file extension. The jar tool can be used to create **.jar** file.

#### Question # 4

***What is a fat jar?***

***Fat jar,* or *uber jar,* is a jar which contains all project class files and classes from all the libraries, on which the project depends.**

#### Question # 5

***What is JCL?***

JCL stands for Java Class Library. It is a set of dynamically loadable libraries that Java applications can call at run time. JCL includes fundamental classes such as **java.lang.String**, **java.lang.Thread**, **java.util.ArrayList** and all the other classes from Java API. Almost all of JCL is stored in a single Java archive file called **rt.jar** (classes.jar on mac) runtime jar for short, which is provided with JRE and JDK distributions. Starting from Java 9 the **rt.jar** file has been broken into several smaller modules.

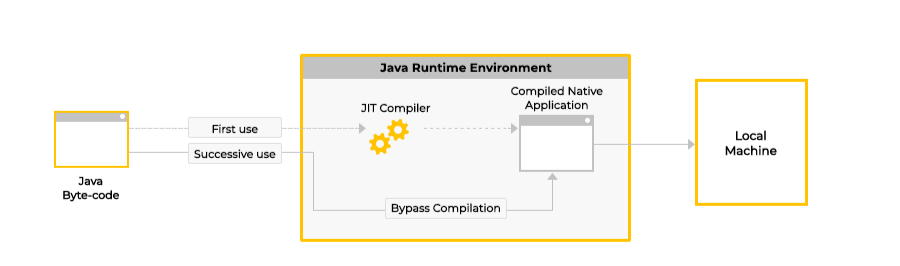
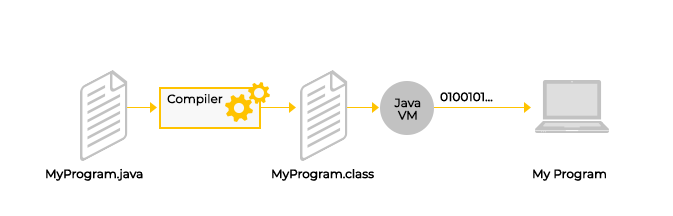
#### Question # 6

***What is a .hprof file?***

HProf is a tool built into JDK for profiling the CPU and heap usage within a JVM. A Java process crash may produce an **.hprof** file containing a heap dump of the process at the time of the failure.

#### Commonly Used Java Tools

|  |  |
| --- | --- |
| Name | Used For |
| **java** | Launches a java application. |
| **javac** | The Java programming language compiler, javac, reads source files written in the Java programming language, and compiles them into bytecode class files. |
| **javadoc** | Javadoc is a tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, interfaces, constructors, methods, and fields. |
| **jar** | combines multiple files into a single JAR archive file. The combined files could include **.class** files, image and sound files etc. |
| **javap** | The javap command disassembles one or more class files. For instance, it can be used to view public, protected and private members of a class. |
| **JPDA** | The Java Platform Debugger Architecture is a collection of APIs to debug Java code. |
| **jConsole** | JConsole is a graphical monitoring tool to monitor Java Virtual Machine and Java applications both on a local or remote machine. |
| **Java VisualVM** | VisualVM is a tool that provides a visual interface for viewing detailed information about Java applications while they are running on a Java Virtual Machine. It helps in troubleshooting and profiling these applications. |
| **jcmd** | It is a comprehensive JDK tool for troubleshooting JVM applications. It can be used to get heap dumps, stack traces, retrieving garbage collector statistics etc. |
| **jmap** | map is a tool to print statistics about the memory in a running JVM. We can use it for local or remote processes. It can also be used to generate heap dumps. |
| **jstack** | This tool can be used to retrieve stack traces of all Java threads running within a target JVM. |
| **jstat** | This tool is used to monitor JVM statistics, which can also be observed using the visual tools. |
| **jinfo** | This tool prints Java configuration information for a given Java process or core file or a remote debug server. Configuration information includes Java System properties and Java virtual machine command line flags. |

s

# Variable Argument Passing

This lesson explains how variable number of arguments can be passed to methods.

#### Question # 1

***How can we pass multiple or variable number of arguments to a method on each invocation call?***

We can pass variable number of arguments to a method using varargs feature. Below is an example of passing multiple arguments of the same type to a method.

##### Varargs Example

**public void childrenNames(String... names) {  
        for (int i = 0; i < names.length; i++)  
            System.out.println(names[i]);  
    }**

Note:

* The type name is followed by three dots, a space, and then the variable name.
* The varargs variable is treated like an array.
* The varargs variable must appear at the last in the method signature.
* As a consequence of the above, there can only be a single varargs in a method signature.

The above method can be invoked as follows:

##### Invoking Varargs Method

**childrenNames();  
        childrenNames("jane");          
        childrenNames("jane", "tom", "peter");**

**Varargs method can also be invoked without any arguments.**

1

2

3

4

5

6

7

8

9

10

11

12

13

class Demonstration {

    public static void main( String args[] ) {

        childrenNames();

        childrenNames("tom", "nancy");

        childrenNames("trump", "obama", "modi");

    }

    public static void childrenNames(String... names) {

        for (int i = 0; i < names.length; i++)

            System.out.println(names[i]);

    }

}

# Pass by Value

This lesson explains how Java works as pass by value.

#### Question # 1

***Is Java both pass by reference and pass by value?***

**Q**

###### A)

True

**Your Answer**

###### B)

False

Explanation

Java is pass by value only. Even reference data types are passed by value.

#### Great, you got it right!

Retake Quiz

#### Question # 2

***Give an example of pass by value?***

When primitive data types are passed to methods, their values are copied and sent over. Any changes made within the receiving method to the received values don't affect or change the original values. Consider the below snippet

##### Pass By Value Example

    void passByValue() {  
        float gravity = 9.8f;  
        receiveByVale(gravity);  
        System.out.println("Gravity acceleration = " + gravity);  
    }  
   
    void receiveByVale(float gravity) {  
        // Attempt to change gravity  
        gravity = 10f;  
    }

The above code will not change the value of the variable **gravity** in the method **passByValue**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

class PassByValueExample {

    public static void main( String args[] ) {

      (new PassByValueExample()).passByValue();

    }

    void passByValue() {

        float gravity = 9.8f;

        receiveByVale(gravity);

        System.out.println("Gravity acceleration = " + gravity);

    }

    void receiveByVale(float gravity) {

        // Attempt to change gravity

        gravity = 10f;

    }

}





RUN

SAVERESET

###### Output

1.761s

Gravity acceleration = 9.8

**Remember Java's method passing always work as pass by value!**

# Pass by Reference

This lesson explains the concept of pass by reference.

#### Question # 1

***What is passing by reference?***

**Passing by reference doesn't apply to Java!** Reference data type parameters, such as objects, are also passed into methods by value. This means that when the method returns, *the passed-in reference still references the same object as before*.

To understand the concept of passing by reference, we'll go through an example. Consider the below code snippet.

**public class SuperList {  
   
    // Constructor  
    public SuperList(int n) {  
1.       List<Integer> superList;  
2.       allocate(superList, n);  
    }  
   
    // Method that does initialization  
    void allocate(List<Integer> list, int n) {  
3.       list = new ArrayList<>(n);  
    }  
}**

What should happen when we initialize an object of class **SuperList**? It will be null, which may seem counterintuitive.

* Consider **superList** to be a holder that will hold a value of null on line 1.
* On line 2, we are passing a value of null and not the variable **superList** itself. This is a very important distinction to realize.
* When program control, reaches line 3, the **list** variable is not the variable **superList**. In fact, it's a brand-new variable (holder) which receives a value of null.
* Line 3 also initializes the **list** variable to an object of **ArrayList** and the **list** variable will hold the reference or the address of the **ArrayList** object in the memory(heap).
* When the program control returns to line 2, **superList** is still null because it was never passed in and assigned the **ArrayList** object.

In Java, we are *copying* the reference or the address the reference data type variable holds and passing it, and not the actual variable.

Consider the below diagram for further clarity.

Note that objects are always created in heap memory and the program *variables* are only references or addresses to them. So, when we pass a reference data type, the address of the object in the heap memory is copied and passed along. The receiving method can use the reference or the address to manipulate the object in the heap.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

import java.util.\*;

class Demonstration {

    public static void main( String args[] ) {

        SuperList obj = new SuperList(5);

        System.out.println("superList = " + obj.sList);

    }

}

class SuperList {

    public List<Integer> sList;

    public SuperList(int n) {

      List<Integer> superList = null;

      allocate(superList, n);

      sList = superList;

    }

    void allocate(List<Integer> list, int n) {

    }

}





RUN

SAVERESET

#### Question # 2

***What will be the output of the run method for the IntegerSwap class below?***

**public class IntegerSwap {  
   
    public void run() {  
        Integer x = 5;  
        Integer y = 9;  
        System.out.println("Before Swap x: " + x + " y: " + y);  
        swap(x, y);  
        System.out.println("After Swap x: " + x + " y: " + y);  
    }  
   
    private void swap(Integer a, Integer b) {  
        Integer temp = a;  
        a = b;  
        b = temp;  
    }  
}**

The output for the two print statements will exactly be the same, i.e. there will be no swapping. Follow the diagram below to understand why.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

class IntegerSwap {

    public static void main( String args[] ) {

        (new IntegerSwap()).run();

    }

    public void run() {

        Integer x = 5;

        Integer y = 9;

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

        swap(x, y);

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

    }

    private void swap(Integer a, Integer b) {

        Integer temp = a;

        a = b;

        b = temp;

    }

}





RUN

SAVERESET

As you can see from the diagram, *a* and *b* appear as stack variables holding addresses of integer object locations. Once the program control returns back to the run method, the *x* and *y* keep pointing to the same integer objects in heap because they passed in the *references* or the *addresses* of the integer objects and not themselves.

#### Question # 3

***What value will be printed from the following snippet?***

**String[] students = new String[10];  
        String studentName = "You are an awesome developer";  
        students[0] = studentName;  
        studentName = null;  
        System.out.println(students[0]);**

**Q**

**Your Answer**

###### A)

You are an awesome developer

###### B)

null

#### Great, you got it right!

Retake Quiz

1

2

3

4

5

6

7

8

9

10

11

class Demonstration {

    public static void main( String args[] ) {

        String[] students = new String[10];

        String studentName = "You are an awesome developer";

        students[0] = studentName;

        studentName = null;

        System.out.println(students[0]);

    }

}





RUN

SAVERESET

**COMPLETED**

[**←    Back**](https://www.educative.io/courses/java-interview-handbook/q2MqxywOyrp)



# Method Overloading

Discusses method overloading in Java.

#### Question # 1

***Consider the following overloaded methods and determine which method will be invoked for the call myOverloadedMethod(5)?***

**void myOverloadedMethod(long arg) {  
        System.out.println("Method with long invoked");  
    }  
   
    void myOverloadedMethod(int arg) {  
        System.out.println("Method with int invoked");  
    }**

**Q**

**Your Answer**

###### A)

Method with **int** invoked

###### B)

Method with **long** invoked

#### Great, you got it right!

Retake Quiz

1

2

3

4

5

6

7

8

9

10

11

12

13

class Demonstration {

    public static void main( String args[] ) {

        myOverloadedMethod(5);

    }

    static void myOverloadedMethod(long arg) {

        System.out.println("Method with long invoked");

    }

    static void myOverloadedMethod(int arg) {

        System.out.println("Method with int invoked");

    }

}





RUN

SAVERESET

#### Question # 2

***How does method overloading match work in Java?***

Methods of a class can be overloaded in Java by:

* Changing the number of parameters
* Changing the type of the parameters passed into the methods

Note that methods can't be overloaded by changing the return types of the methods, as it may cause ambiguity. While overloading has nothing to do with polymorphism, Java programmers also refer to method overloading as Compile Time Polymorphism because the method that is going to get called will be decided at compile time.

**The compiler uses the name of the method and the types of the argument expressions to locate methods that are both accessible and applicable. There may be more than one such method, in which case the *most specific one is chosen*.** Typically, varargs methods are the last chosen, if they compete with other candidate methods because they are considered less specific than the ones receiving the same parameter type.

#### Question # 3

***Can the main method be overloaded?***

Yes, the static **main** method can be overloaded. But only **public static void main(String[] args)** will be used when your class is launched by the JVM even if you specify one or two command-line arguments. However, programmatically one can invoke the overloaded versions of the **main** method.

1

2

3

4

5

6

7

8

9

10

11

12

13

class Demonstration {

    public static void main( String args[] ) {

        System.out.println( "Traditional main method" );

    }

    public static void main( String singleArg) {

        System.out.println( "Method with single arg" );

    }

    public static void main( ) {

        System.out.println( "Method with no args" );

    }

}





RUN

SAVERESET

# Accessibility

This lesson discusses the various accessibility levels in Java.

#### Question # 1

***How can access to classes, class variables and methods be controlled?***

Access level modifiers determine whether other classes can use a particular field or invoke a particular method in another class. There are two levels of access control, the top level and the member level.

#### Question # 2

***List the access modifiers in Java?***

**There are four accessibility levels in Java.** They are listed below in order of increasing restrictiveness:

* public
* package private
* protected
* private

#### Question # 3

***What are the top-level access modifiers?***

The top-level access modifiers apply to top level classes. **Classes can be marked either public or package private**.

* A public class is accessible across different packages.
* A package private class is only visible to other classes within the same package.

All the four access modifiers can be applied at member level.

#### Question # 4

***Explain the public modifier***

The **public** modifier is the least restrictive modifier. Any class marked public or class member marked public would be accessible everywhere.

#### Question # 5

***How can we mark an entity package private in Java?***

There's no explicit modifier for package private. **In the absence of any modifier the class or member variables are package private. A member marked package private is only visible within its own package.** Consider the class below.

##### Package Private Example

// class can be accessed by other classes within the same  
// package but not outside of it.  
class IamPackagePrivateClass {  
   
    int IamPackagePrivate;  
    private int IamPrivate;  
   
    public IamPackagePrivate(int a, int b) {  
        this.IamPackagePrivate = a;  
        this.IamPrivate = b;  
    }  
}

Package private is a slightly wider form of private. One nice thing about package-private is that you can use it to give access to methods you would otherwise consider private to unit test classes. So, if you use helper classes which have no other use but to help your public classes do something clients need, it makes sense to make them package private as you want to keep things as simple as possible for users of the library.

#### Question # 6

***Explain the protected access modifier?***

The **protected** modifier specifies that a member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another or the same package.

#### Question # 7

***Explain the private modifier?***

The **private** modifier specifies that the member can only be accessed in its own class. Note that top level classes can't be marked private or protected but nested ones can be.

#### Question # 8

***Is it a good idea to make classes and fields public?***

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

* Use the most restrictive access level that makes sense for a particular member.
* Use private unless you have a good reason not to.
* Avoid public fields except for constants.
* Note public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

#### Access Modifier Table

The table below, lists how the different access modifiers affect visibility within different entities in Java.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Marked with Modifier** | **Visible Within Class** | **Visible within Package** | **Visible within Subclass** | **Visible within World** |
| **public** | yes | yes | yes | yes |
| **protected** | yes | yes | yes | no |
| **package private** | yes | yes | no | no |
| **private** | yes | no | no | no |

**COMPLETED**

[**←    Back**](https://www.educative.io/courses/java-interview-handbook/xlxWYp8YvyJ)

