TOP JAVA INTERVIEW QUESTIONS—ANSWERED

When preparing for a job interview, it is a good move to study industry-specific questions and be prepared on the best way to answer them. This is especially crucial when you are dealing with technical knowledge, such as programming languages like Java.

Java is consistently rated among the most popular programming languages. With its portability and ability to run across multiple platforms, it's no wonder Java's such a highly sought-after programming language—by companies and developers alike.

If you are looking for a career in Java programming, prospective employers will ask you a variety of questions to gauge your knowledge of the language, ranging from the basics to advanced understandings of its technicalities and benefits. Or, if you're a hiring manager or recruiter, you'll want to know some basic and more advanced Java concepts to get the most out of your candidate interviews.

We've put together some top Java questions to help you prepare and feel more confident during the interview process. Whether you're seeking a job or looking to hire, this article will help you out.

Why Java?

Java is one of the most popular programming languages used to create APIs and platforms. It was designed for flexibility, allowing developers to write code that would run on any machine, regardless of architecture or platform. Java is used by a significant majority of companies across the globe.

While these top Java interview questions only scratch the surface, they will offer a broad scope of the type of knowledge you will need in order to prepare for your interview.

Basic Java Questions

Q. Why is Java considered platform independent?

Java's motto is WORA, which stands for "write once, run anywhere". Java is considered platform independent because, thanks to bytecode, it can run on any and all operating systems: Mac, Linux, or Windows. This is beneficial in a networked environment because companies typically use many kinds of computers and devices.

Q. What are JVM, JRE, and JDK?

- **Java Virtual Machine (JVM)** is an abstract machine that provides the runtime environment for the Java bytecode.
- **Java Runtime Environment (JRE)** refers to the specific runtime environment in which the Java bytecode can be executed. JRE implements the JVM and provides all support files that the JVM uses during runtime.
- **Java Development Kit (JDK)** is the tool that gathers, documents, and packages Java programs. It includes both the JRE and necessary development tools.

Q. What is the difference between equals() and == ?

The **equals()** method is used to compare the values of two objects. The **==** **equality operator** is used to compare primitives and objects.

Q. What is hashing?

Hashing is a technique of converting a large string to a small string. A shorter value speeds up indexing and searching. Hashing converts an object into integer form, using the **hashCode()**method. You must write the hashCode() properly for accurate HashMap performance.

Q. What is HashMap?

HashMap provides the basic implementation of the mapping interface of Java. It stores the data in **(Key, Value)** pairs and in order to access a value, one must know its key. HashMap's name is derived from its use of the hashing technique.

Q. What differentiates HashMap and HashTable?

In **HashMap**, the methods are not synchronized and there is no thread safety. The iterator is used to iterate the values and it allows one null key along with multiple null values.

In **HashTable**, the key methods are synchronized and there is thread safety. The enumerator iterates the values, and it doesn't allow anything that is null. Performance is slow, especially compared to HashMap.

Q. What is the difference between ArrayList and vector?

An **ArrayList** is not synchronized, making it much faster. ArrayList can only use **Iterator** for traversing

an ArrayList.

A **vector** is synchronized, which slows it down. However, it is thread safe and limited to one thread at a time. HashTable and vector are the only classes that use both **Enumeration** and **Iterator**.

Q. How are inner classes and sub-classes different?

An **inner class** is a class that is nested within another class. It has access rights for the class that is nesting it. It can access all variables and methods defined in the **outer class**.

A **sub-class** is a class that is derived from another class. It can access all public and protected methods and fields of its super class.

Q. What are the various access modifiers for Java classes?

Access specifiers are the keywords used before a class name that defines its access scope. In Java, the four modifiers are:

- **Private**. The variable is not available outside the class, so outside members cannot access the private members. (Remember, classes and interfaces cannot be private.)
- **Public**. The method or variable can be accessed by all the other classes in the project.
- **Default**. Visible to the classes with the package.
- **Protected**. The variable can be accessed within the same package classes and sub-class of any other packages. These cannot be used for class or interfaces.

Q. What is data encapsulation and what's its significance?

Encapsulation is a concept in Object Oriented Programming for combining properties and methods in a single unit. Encapsulation helps programmers follow a modular approach for software development—each object has its own set of methods and variables, and it serves its functions independent of other objects.

Q. What are the three types of loops?

Looping is used to execute a statement or a block of statements repeatedly. The three types of loops in Java are:

- **For loops** execute statements repeatedly for a given number of times.
- **While loops** are used when certain statements need to be executed repeatedly until a condition is fulfilled. The condition is checked before executing statements.
- **Do while loops** are similar to while loops except they are checked after executing a block of statements.

Q. How is an infinite loop declared?

An **infinite loop** is an instruction sequence that loops endlessly when a terminating condition isn't met. It can be broken by defining any breaking logic in the body of the statement blocks.

Q. What is an exception?

An **exception** is an event that occurs during a program's execution that disrupts the normal flow of the program. After a method throws an exception, the runtime system attempts to find something to handle it. If that exception can't be handled, the execution is terminated before it completes the task.

Q. What are the types of exceptions?

There are two types of exceptions: checked and unchecked.

Checked exceptions are checked by the compiler at the time of compilation. For example, classes that extend throwable class (minus runtime exception or error). They must either declare the exception using throws keyword or get surrounded by the appropriate try/catch:

- IOException
- SQLException
- DataAccessException
- ClassNotFoundException
- InvocationTargetException
- MalformedURLException

Unchecked Exceptions are not verified by the compiler at the time of compilation. These exceptions are a result of bad programming; thefore, the program won't give a compilation error. All unchecked exceptions are direct sub-classes of RuntimeException class:

- NullPointerException
- ArrayIndexOutOfBound
- IllegalArgumentException
- IllegalStateException

Complex Java Questions

Use these questions to gauge your understanding of more advanced Java concepts.

Q. Java does not support multiple inheritance. How did this prevent the "diamond problem"? Why is this no longer true?

Java does not allow **multiple inheritance** for classes, only for interfaces. In the past, this prevented the "diamond problem" more typically seen in C++: it was possible only to inherit an implementation from the single parent class.

In Java 8, which premiered in 2014, and subsequent versions, a class *can* inherit a method implementation from either its parent class or any of its interfaces. This increases the chance that the compiler rejects the compilation.

Q. Why is reflection used?

Reflection makes it possible to inspect classes, interfaces, fields, and methods at runtime without knowing the names of the classes, methods etc., at compile time. Reflection describes code that is

able to inspect other code in the same system. It also makes it possible to instantiate new objects, invoke methods, and get/set field values using reflection. Reflection provides the ability to make modifications at runtime by making use of introspection.

Q. Does Java support operator overloading?

Operator overloading is not supported in Java. This is intentional, as the creators of Java wanted to maintain simplicity and prevent people from using operators in a confusing way. (For example, allowing multiple meanings to come up for the same operator.)

Q. Why is string immutable?

An **immutable object** is an object whose internal state, once created, remains constant. Once the object is assigned to a variable, the object can be neither updated nor mutated. Keeping **string** immutable supports and benefits several activities, including caching, security, synchronization, and performance.

Java String Pool is the special memory region where strong are stored by the JVM. Because strings are immutable, the JVM optimizes the amount of memory allocated for them by storing only a single copy of each string in the pool, preserving crucial memory resources.

Immutable strings also make sense for security purposes, as it is easier to operate with sensitive code when values don't change. Consider the alternative: If strings were mutable, by the time you execute any updates, you aren't able to ensure the security of the string as it may have been compromised between integrity checks.

Being immutable automatically makes the string safe. The string won't be changed when accessed from multiple threads. In collections that use hash implementations, immutability enhances their performance.

Q. How do you override the static method in Java? Does creating the same method in the subclass result in a compile-time error?

Java does not allow for an **override** of the static method. You can, however, declare the method with the same signature in the sub-class. This isn't considered an override, however, as there is no run-time polymorphism.

If a **derived class** defines a static method with the same signature as a static method in the base class, then the method in the derived class hides the method in the base class. It would not be a compile-time error to declare the exact same method in a subclass; this would be known as **method hiding**.

Q. How do you fix a non-serializable member within a serializable class?

The **object serialization process** converts an object into a binary format that can then be sent, via the network, to any other running JVM. Attempts to serialize that class will fail with **NotSerializableException**.

If all fields in a class are serializable, simply add **implements serializable** to the class declaration. If not, you can repeat the process for that field's class into the object graph. If there is a class that can't,

or shouldn't, be made serializable, add the transient keyword to the field declaration. This tells the JVM to ignore the specified field when serializing.