Interview Cake

Java Interview Questions

How to pass the Java programming interview

Chris got the job at **Apple**:

I got the job! Your questions prepared me big time and I felt really relaxed throughout the entire process. I believe in Interview Cake!

Java Interview Questions in This Article

What are Singletons for?

Explain what kinds of problems singletons solve. Then implement a singleton class and test that is in fact a singleton. keep reading »

What's the Difference Between a String Literal and a String Object?

How do I know which one to use in a given context? And should I ever use a naked char array instead? keep reading »

What's the Difference Between an int and a long in Java?

How should I decide which one to use? keep reading »

What's the Difference Between an ArrayList, a LinkedList, and a Vector?

How do I pick which one to use? keep reading »

What are singletons used for in Java applications?

Singletons are classes which can have no more than one object. They're most useful for storing global state across a system.

Some situations where one might use a singleton include:

1. **A system-wide "global value," that many parts of the sytem may need to access—e.g. the software's license number**. Some software requires a valid "license" in order to run. Such sofware might want to make the current license available to different parts of the software system while it's running. A singleton is a good place to store that information, since there's only ever one correct answer to the question "what license are we using?"
2. **Logging**. You might want different loggers with different configurations. For example, you might want a "loud" logger that emails exceptions back to the software maintainer, to alert her of crucial issues, as well as a "quiet" logger that simply logs errors to a file on the user's system. Your software might have several components (read: several *classes*) that want to use the loud logger (e.g. payment-related stuff) and several components that that want to use the *quiet* logger (e.g. caching systems—if the cache fails the system might still run correctly, just more slowly). `java.util.logging.LogManager` manages a set of individual loggers which are singletons. You can access them by name with getLogger(), and you can add new ones with addLogger.

Singletons are contentious these days. Many people believe they should be avoided (http://programmers.stackexchange.com/questions/148108/why-is-global-state-so-evil), or at least be used less often than they generally are.

Even so, implementing a singleton is an interesting coding challenge.

Suppose we wanted a singleton called `InstallationDetails` that stored some information, including the `licenseNumber`. How would we implement this?

We have several options. The first is **lazy**: have the class get or create its instance just in time, as it's requested:

Java

```java
public final class InstallationDetails {
    private static InstallationDetails INSTANCE = null;
    public long licenseNumber;

    // by making the constructor private, we prevent instantiation
    private InstallationDetails() {}

    public static InstallationDetails getInstance() {
        if(INSTANCE == null){
            INSTANCE = new InstallationDetails();
        }
        return INSTANCE;
    }
}
```

To make this thread-safe:

Java

```java
public final class InstallationDetails {
    private static volatile InstallationDetails INSTANCE = null;
    public long licenseNumber;

    // by making the constructor private, we prevent instantiation
    private InstallationDetails() {}

    public static InstallationDetails getInstance() {
        if(INSTANCE == null){
            synchronized (InstallationDetails.class) {
                if(INSTANCE == null){
                    INSTANCE = new InstallationDetails();
                }
            }
        }
        return INSTANCE;
    }
}
```

Another is to **eagerly** have the class instantiate its singleton object even before one is requested:

Java

```java
public final class InstallationDetails {
    private static final InstallationDetails INSTANCE = new InstallationDetails();
    public long licenseNumber;

    // by making the constructor private, we prevent instantiation
    private InstallationDetails() {}

    public static InstallationDetails getInstance() {
        return INSTANCE;
    }
}
```

There's also the initialization-on-demand way:

Java

```java
public final class InstallationDetails {
    public long licenseNumber;

    // by making the constructor private, we prevent instantiation
    private InstallationDetails() {}

    private static class InstallationDetailsHolder {
        private static final InstallationDetails INSTANCE = new InstallationDetails();
    }

    public static InstallationDetails getInstance() {
        return InstallationDetailsHolder.INSTANCE;
    }
}
```

This method is lazy like the first approach, and also thread-safe.

Then there's the enum way:

Java
```java
public enum InstallationDetails {
    INSTANCE;
    public long licenseNumber;
}
```

Regardless of which method we use, we can test that our class is indeed a singleton like so:

Java
```java
public void testInstallationDetailsIsSingleton {
    InstallationDetails obj1 = InstallationDetails.getInstance();
    InstallationDetails obj2 = InstallationDetails.getInstance();

    obj1.licenseNumber = 123;
    obj2.licenseNumber = 456;

    assertTrue(obj1 == obj2);
    assertEquals(456, obj1.licenseNumber);
    assertEquals(456, obj2.licenseNumber);
}
```

In Java, how do I decide whether to use a string literal or a string object?

To hard-code a string in Java, we have two options. A **string literal**:

```java
String username = "CakeLover89";
```

And a **string object**:

```java
String username = new String("CakeLover89");
```

What's different about these two options?

When you use a string literal, the string is **interned**. That means it's stored in the **"string pool"** or "string intern pool". In the string pool, each string is stored no more than once. So if you have two separate variables holding the same string literals in a Java program:

```java
String awayMessage = "I am the cake king.";
String emailSignature = "I am the cake king.";
```

Those two Strings don't just contain the same objects in the same order, they are in fact both pointers to *the same single canonical string in the string pool*. This means they will pass an '==' check.

Java

```java
String awayMessage = "I am the cake king.";
String emailSignature = "I am the cake king.";

awayMessage == emailSignature; // True -- same object!
awayMessage.equals(emailSignature) // True -- same contents
```

If our Strings were instantiated as objects, however, they would not be "interned," so they would remain separate objects (stored outside of the string pool).

Java

```java
String awayMessage = new String("I am the cake king.");
String emailSignature = new String("I am the cake king.");

awayMessage == emailSignature; // False -- different objects!
awayMessage.equals(emailSignature) // True -- same contents
```

> In some languages, like Lisp and Ruby, interned strings are called "symbols."

Can you intern a string "by hand?" Absolutely:

Java

```java
String awayMessage = new String("I am the cake king.");
String emailSignature = new String("I am the cake king.");

// intern those strings!
awayMessage = awayMessage.intern();
emailSignature = emailSignature.intern();

awayMessage == emailSignature; // True -- same object!
awayMessage.equals(emailSignature) // True -- same contents
```

Given this, String literals can be thought of as *syntactic sugar* for instantiating a String and immediately interning it.

So which should you use, string literals or String objects?

You should almost always use String literals. Here's why:

It saves time. Comparing equality of interned strings is a constant-time operation, whereas comparing with .equals() is O(n) time.

It saves space. You can have several variables referencing one string while only storing that set of characters in one canonical place (the string pool).

Use String objects only if you want to be able to have two separate string objects with the same contents.

Bonus: consider storing *sensitive* strings (like passwords) as char arrays. This has a few nice features:

1. A char array can be "zeroed out" when you're done with it, giving *some* assurance that the string has been removed from memory (though it may still exist in some caching layers).
2. If a char array is accidentally printed (e.g. in a debug statement), by default its *address in memory* will be printed, rather than its contents.

The `Swing` library, for example, has a `getPassword()` method which always returns a char[].

What's the difference between an int and a long in Java?

32 bits. The difference is 32 bits :)

ints are 32-bit numbers, while longs are 64-bit numbers. This means ints take up half as much space in memory as longs, but it also means ints can't store as big of numbers as longs.

Here's a full listing of non-decimal number primitive types in Java, along with the maximum and minimum values they can hold:

name	bits	min value	max value
byte	8	-128	127
short	16	$-32,768$	$32,767$
int	32	-2^{31} (~-2 billion)	$2^{31}-1$ (~2 billion)
long	64	-2^{63} (~-9 "billion billion billion")	$2^{63}-1$ (~9 "billion billion billion")

Which should you use? It depends how big you expect your numbers to be. **In general, you should use the data type that's big enough to hold the numbers you expect to store, but no bigger.**

If you choose a type that's *too small*, you risk integer overflow . In Java, integers "silently" overflow—that is, no error is thrown, the integer simply goes from a very large value to a very small value. This can cause some *very* difficult-to-diagnose bugs. In Java 8, you can use `Math.addExact()` and `Math.subtractExact()` to force an exception to be thrown if the operation causes an overflow.

> **What's an example of a time when 32 bits is not enough? When you're counting views on a viral video**. YouTube famously ran into trouble when the Gangnam Style video hit over $2^{31}-1$ views, forcing them to upgrade their view counts from 32-bit to 64-bit numbers (http://arstechnica.com/business/2014/12/gangnam-style-overflows-int_max-forces-youtube-to-go-64-bit/).

Given the threat of integer overflow, one might be tempted to just *always* use longs, "to be safe." But you'd risk using up more space than you needed to. Specifically, if you use longs when you could be using ints, you'll use *twice as much space* as you need to. If you're dealing with a big array of numbers that takes up several gigabytes of space, a space savings of one half is huge.

Another nice side effect of using the correctly-sized data type to store your numbers is that it serves a bit of *documentation* in your code—a reminder to yourself and to other engineers about the specific range of numbers you're expecting for a given variable.

What's the difference between a Java ArrayList, Vector, and LinkedList? How do I pick which one to use?

Linked List vs Dynamic Array

The first difference is that LinkedList is, predictably enough, an implementation of a linked list . ArrayList and Vector, on the other hand, are implementations of dynamic arrays .

So LinkedList has the strengths and weaknesses of a linked list, while ArrayList and Vector have the strengths and weaknesses of a dynamic array. In particular:

Advantages of Dynamic Arrays

1. **getting the item at a specific position/index (`get()`) is faster**. It's $O(1)$ time, vs $O(n)$ time for a linked list

2. **they take up less space than linked lists**. In a linked list, each new "node" is a separate data structure which incurs some space overhead, whereas for a dynamic array each new item is simply another element in the underlying array. This difference is asymptotically insignificant, however—both data structures take $O(n)$ space.
3. **they're more cache friendly**, since the elements are actually next to each-other in memory. This means that reads, especially sequential reads, often end up being much faster. Again, this difference is asymptotically insignificant.

Advantages of Linked Lists

1. **`Iterator.remove()` is faster with a linked list**. It's $O(1)$ time, vs $O(n)$ time for a dynamic array (dynamic arrays have to "scoot over" each subsequent item to fill in the gap created by the removal).
2. **`ListIterator.add()` is faster with a linked list**. It's $O(1)$ time, vs $O(n)$ time for a dynamic array (dynamic arrays have to "scoot over" each subsequent item to make space for the new item).
3. **`add()` is always $O(1)$ time**. Dynamic arrays have an *amortized* $O(1)$ time cost for `add()`, but a *worst case* $O(n)$ time cost, because an `add()` could trigger a doubling of the underlying array.

So which should you use? **Conventional wisdom is that dynamic arrays are *usually* the right choice.** The main exception is if you plan to use `Iterator.remove()` and/or `ListIterator.add()` very heavily and you *don't* plan to use `get()` very often. *Then* a LinkedList might be the right choice, although it may take up more memory and—because it's less cache-friendly—it may have slower reads.

ArrayList vs Vector

So within our options for dynamic array data structures, which one should we choose?

The main difference (though there are others) is that Vector is entirely thread-safe, because it synchronizes on each individual operation.

But **you should almost always use ArrayList, even if you're writing code that needs to be thread-safe**. The reason is that synchronizing on *each operation* is generally not the best way to make your dynamic array thread-safe. Often what you really want is to synchronize a *whole set of operations*, such as looping through the dynamic array, making some modifications as you go. If you're going to be doing multiple operations while looping through a Vector, you'll need to take out a lock for that whole series of operations *anyway*, or else another thread could modify the Vector underneath you, causing a `ConcurrentModificationException`.

For this reason, Vectors are generally considered to be obsolete (http://stackoverflow.com/questions/1386275/why-is-java-vector-class-considered-obsolete-or-deprecated). Use an ArrayList instead, and manage any necessary synchronization by hand.

Ready for more?

If you're ready to start applying these concepts to some problems, check out our mock coding interview questions (/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

MillionGazillion »

I'm making a new search engine called MillionGazillion(tm), and I need help figuring out what data structures to use. keep reading »