

JAVA INTERVIEW BOOTCAMP

THE COMPLETE GUIDE TO FINDING AND
LANDING YOUR NEXT JAVA DEVELOPER
ROLE

SAM ATKINSON

Java Interview Bootcamp

The practical guide to the Java interview process

Sam Atkinson

This book is for sale at <http://leanpub.com/javainterviewbootcamp>

This version was published on 2016-05-21



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2014 - 2016 Sam Atkinson

To Laura. Thank you for (just about) putting up with me whilst I've been writing this and building the website. You're amazing.

To Mike & Dan for being an awesome support network and listening to me talk on and on and on about CFIQ all the time.

To Dave. Your incredible and tireless work as editor has made this a much better book than it would have been otherwise.

Contents

Introduction	1
Core Java review	3
Object oriented programming	4
Basics	4
Data structures	8
The collection types	9
Collection implementations	10
Multithreading and collections	15
Java exceptions	19
Core questions	19
JVM and garbage collection	21
New generation	23
Old generation	24
PermGen	25
From the code	25
JVM tuning	25
Diagnosis	26
Threading	31
Introduction to threading	31
The atomic classes	35
Immutability	35
Thread methods	36
Object methods	37
Deadlock	39
Futures, callables and executors	42
Big O Notation	45
What on earth is Big O?	45
Why does this matter?	46

CONTENTS

Examples	46
How to figure out Big O in an interview	49
Sample question	49
Conclusion	51

Introduction

Welcome to Java Interview Bootcamp. Firstly, I'd just like to say a huge thank you to you for buying this book. I hope you will enjoy reading the book, but more importantly that it helps you to get your next amazing job.

Helping Java devs through the interview process is something I'm really passionate about. In my career I've interviewed hundreds of people from the most junior developers to the most senior technical leads. There have been some consistent themes throughout that really drove my desire to write this book and create www.corejavainterviewquestions.com. What I have learnt can be summed up in two key points; people are terrible at preparing for interviews, and they do not know how to give a good impression during an interview.

Programming interviews are both a skill and an art and as such they require practice and time. I have discovered most people simply don't take the time to prepare. A huge proportion of the developers I've interviewed struggle to answer basic java questions. They seem intentionally stand offish and make no effort to come across as personable. These are simple things that can be remediated with preparation. By purchasing this book you are already ahead of most of your competitors because you know that you need to take the time to study beforehand.

This book is split into two sections, interview process and core java revision. When someone is interviewing you they are looking to ensure that you are technically capable, but they also want to know that you are someone they can work with day to day. In the first half of the book we cover the different types of interview and what you can do as a candidate to make the best impression possible on your future employer.

The second half of the book is a ground-up revision of core java. The simple fact is in most interviews the technical questions bare little resemblance to our day-to-day programming. I've found most interview processes don't even involve a programming stage. Instead companies love to ask obscure threading questions or ask you to write java code on a whiteboard to reverse a string. Whether these are good interview questions or not isn't the subject of this book. The simple fact is you have to be prepared for them, which means spending some time to revise the basic core java principles.

I hope you find the everything you need in here. If you have any questions, queries, or just want to have a chat then you can email me on hello@corejavainterviewquestions.com. I will always respond.

If you're not already a member, don't forget to head to www.corejavainterviewquestions.com and sign up to the mailing list. It's packed with tons of valuable information and goodies.

Send me an email with your receipt for this book to hello@corejavainterviewquestions.com and I will send you back 2 example resumes and a practice interview too.

And last but not least: good luck with all your interviews. I'm sure you're going to knock them out the park.

Sam Atkinson, author of Java Interview Bootcamp.

Core Java review

Now you're fully equipped to excel at the softer skills of interviewing it's time to learn how to answer the technical stuff. There is an unlimited selection of questions which an interviewer could ask you; the Java universe is huge. So where do you start?

The traditional method is to google around for lists of interview questions (that may even be how you found this book). The simple fact is that generally the questions will be much more in-depth than what most of what you will find- you're never going to get asked a simple list of questions. You really need to understand the areas you're talking about. That's what this section will give you; an in depth revision guide to the core areas of Java that you need to know for most interviews.

Read this half all the way through, then come back to the sections you feel you really struggle with. It's great if you can remember the esoteric facts, but you really want to concentrate on the areas you feel are your weakest. Perhaps you've never worked on a highly multi-threaded application. In that case, spend the time re-reading about Threading. Come up with practical coding examples; nothing helps lock it into memory like actually coding a solution!

Object oriented programming

Basics

Questions on object oriented programming (OOP) are exceptionally common on interviews and with good reasons. For junior developers it can clearly differentiate between someone who doesn't really understand programming and a candidate who could be a potential star but simply lacks in experience. For more senior candidates I find it can help to identify developers who understand good design and how code should be put together. There is a huge array of questions (pun intended) and it is well worth significant study. Fundamentally you will understand all of the topics covered in this chapter from the work you do in your day to day job; however being able to code away on your machine and being able to explain the concepts are very different things. We all know (hopefully) that we should aim for low coupling and high cohesion. But can you explain what that means? Can you give a concise answer? Take the time to really think about the questions below. Often answers will vary based on personal preference and experience. That's fine, as long as you have a convincing answer!

Q: Java is an object oriented language. What does that mean?

There are a number of different programming paradigms of which OOP is but one. Another would be functional programming (which is becoming very popular with languages like Scala). Programs are modelled as objects which are analogous to real world objects. For example, when modelling a car you could create a Car object, which would possess 4 Wheel objects and an Engine object. An object is defined by the data it contains and the actions that can be performed on it or with it. The four key principles of OOP are Abstraction, Encapsulation, Inheritance and Polymorphism.

Q: What is a class? What is an object?

A class can be thought of like a template; it defines what an object will look like, specifically what fields and methods it will contain. An object is said to be an instance of a class- where the fields will contain data. For example, a "TShirt" class may have a colour and size attribute. A T-Shirt object could have the colour red and the size XL. There could be many instances of an object with different or identical values for the fields.

Q: Explain Abstraction, Encapsulation, Inheritance and Polymorphism

One of the key features of Java is the idea of abstraction and inheritance. Inheritance allows the creation of a class which will have the same fields and methods of another class (and possibly more), and also be considered of the same type. For example, a Car has four wheels and an engine. Using inheritance we can create a TeslaCar, which is still a Car but also has extra fields (such as a cool touchscreen), or a FormulaOneCar, which would also be a car but have an extra field for it's spoiler. This allows us two special features: - If I have a list of cars and I want to see what colour each one

is, I don't care if it's a TeslaCar or a FormulaOneCar. I just want to call the Car's ".colour()" method. Because TeslaCar "is-a" Car, we know that it has this method available on it. - If the class we are inheriting from already has the implementation of the method, we do not need to write it ourselves (although we can if we want to).

The ability to define or override a methods implementation in a subclass results in Polymorphism. When executing a method the implementation used is not the one specified by the variable type but that of the variable instance.

```
1    public static void main(String[] args) {
2        NumberPrinter numberPrinter = new EvenNumberPrinter();
3        numberPrinter.printSomeNumbers();
4    }
5
6    private static class EvenNumberPrinter extends NumberPrinter {
7        public void printSomeNumbers() {
8            System.out.println("2468");
9        }
10   }
11
12   private static class NumberPrinter {
13       public void printSomeNumbers() {
14           System.out.println("123456");
15       }
16   }
```

This would print out "2468" despite the type being a NumberPrinter type.

This is also a good example of method overriding, where a subclass provides a different implementation to that of its superclass. It is possible to call non-abstract methods on the super type using the keyword super, e.g. from EvenNumberPrinter's printSomeNumbers() it would be ok to call super.printSomeNumbers():

```
1    public void printSomeNumbers() {
2        super.printSomeNumbers();
3        System.out.println("2468");
4    }
```

This would result in

```
1      123456
2      2468
```

An Interface is a class with no implementation. To continue our analogy, if Car was an interface with the method `.colour()`, that is simply specifying that all Cars will have a `colour()` method (and makes no statement about how it will be implemented). You cannot create an object from an interface as it has no implementation or fields.

An abstract class is another type of class. It may have some methods which have not been implemented (which will be labelled abstract), but it may also have some methods which have been implemented. Abstract classes also cannot be used to create an object, as some implementation code will be missing.

An object can only be instantiated based on a full class (e.g. not abstract, not an interface). In Java, a class can implement between zero or many interfaces, or it can extend zero or one abstract or concrete classes only.

Encapsulation is the principal of data hiding. An object has internal state that it does not want modifying by other objects, so it can hide this. The only properties and behaviours that can be accessed on an object are those that are deliberately exposed by it's creator. This is accomplished in Java using access modifiers; public, protected, private and default.

Q: What are the different access modifiers in Java and what do they mean?

- public: accessible to everyone in the JVM
- private: only accessible from inside the class.
- protected: accessible by any class in the same package or any subclass in any package
- default: when no visibility modifier is present. accessible from any class in the same package only.

Q: What is meant when it is said we favour low coupling and high cohesion?

This is one of the key design principals in OOP. Low coupling refers to how different components interact; that could be objects, modules, systems, any level. We want components to have a very limited dependency on other components. This makes it quick and cheap to refactor and make changes to. This is usually accomplished by creating clean interfaces between components and using encapsulation to hide the internal implementation. Low coupling is one of the most important ideas for creating systems that are flexible and manageable.

Cohesion refers to the implementation of the components; in a highly cohesive system things are built in such a way that related items are put together and unrelated ones are separate. To return to the Car analogy, it makes sense that the methods for driving a car such as `turn()`, `accelerate()` and `brake()` exist on Car. They are cohesive and belong together. On the other hand, if the Car had a `clean()` method on it that would not be cohesive. A car doesn't clean itself; that is done by something else, perhaps a CarWash or an Owner. High Cohesion is achieved by ensuring that fields and behaviours that don't belong together are separated to a better place.

These terms are often referred to together because cohesive systems with low coupling tend to be more flexible and maintainable.

Q: What is the difference between method overloading and method overriding?

In Java, it is possible for multiple methods to have the same name. They are differentiated by having a different number, type and/or ordering of parameters. This is known as method overloading.

Overriding is the creation of a different implementation of a method in a subclass.

Q: What is a constructor?

A constructor is a special method which is named identical to the class, e.g. a Car class will have a constructor called Car(). A constructor does not specify a return type as it implicitly returns an object of the type of that class, in this case a Car object. It is used to build an instance of that class. If one is not specified there will be a default public no-arg constructor provided. It is possible to overload constructors, and all the access modifiers can be applied to a constructor.

If your class contains any final fields they must be assigned a value by the end of the constructor or your program will not compile.

Q: What is static in Java?

Using the static keyword effectively declares that something belongs to the class, not to the object instance. The static variable is shared by all instances of the object. A static method belongs to the class, meaning that it cannot access any non-static field values. It also means it cannot be overridden. We can also have static code blocks which are executed when a class is loaded into memory:

```
1 static{
2     System.out.println("Hello!");
3     //maybe initialise some static variables here
4 }
```

Static can also be used for nested classes.

Q: What does it mean when we say java does not support multiple inheritance? Is this a good thing?

Java cannot extend functionality from more than one concrete or abstract class. If both parent classes had a jump() method, it would be unclear which functionality the caller would need to use. We can implement multiple interfaces however as the the implementation occurs in our actual class so this problem does not occur.

Q: If you wanted to prevent your methods or classes from being overridden, how would you do this?

By declaring something as final you prevent it from being overridden. Nothing is perfect though, as crafty developers can always use Reflection to get around this, or alternatively just copy and paste your code into their own version of the class. It is rarely a good idea to prevent your methods or classes being overridden, and you should code defensively to reflect this.

Data structures

Java data structures interview questions will come up in your interview. At it's nexus being a programmer is about handling data, transforming and outputting it from one format to another, which is why it is so important to understand what's going on under the hood and how it can affect your application. It never fails to confound me how many people can't tell what collection is used to back an ArrayList (hint: it's in the name). From the interviewer's perspective, questions on data structures reveal a lot of information about the candidate. They show if the candidate has a core understanding of how java works. Even better it provides a platform to lead to a wide ranging set of questions on design, algorithms and a whole lot more. In day to day java programming, you're going to spend most of your time with ArrayLists, LinkedLists and HashMaps. As a result it makes sense to review collections and algorithms you haven't looked at for a while as, in my experience, companies love asking this sort of question.

- **Q: What is the Java Collection Framework?**
- **Q: What Collection types are there in Java?**
- **Q: What are the basic interface types?**
- **Q: What are the differences between List, Set and Map?**

This is the bread and butter stuff that everyone who uses Java should know. The collections framework is simply the set of collection types that are included as part of core java. There are three types of collection you will deal with in Java; Set and List (both of which implement Collection) and Maps (which implement Map and technically aren't part of core Collections). Each type of collection exhibits certain characteristics which define it's usage. The key features of a collection are:

- Elements can be added or removed
- The collection will have a size that can be queried
- The collection may or may not contain duplicates
- It may or may not provide ordering
- It may or may not provide positional access (e.g. get me item at location 6)

The behaviour of these methods varies based on implementation (as you would expect from an interface). What happens on if you call *remove()* on a collection for an object that doesn't exist? It depends on the implementation.

The collection types

Set: A set has no duplicates and no guaranteed order. Because of this, it does not provide positional access. Implements Collection. Example implementations include TreeSet and HashSet.

List: A list may or may not contain duplicates and also guarantees order, allowing positional access. Implements Collection. Example implementations include ArrayList and LinkedList.

Map: A map is slightly different as it contains key-value pairs as opposed to specific object. The key of a map may not contain duplicates. A map has no guaranteed order and no positional access. Does not implement Collection. Example implementations include HashMap and TreeMap.

But what does this mean with regards to interviews? You're almost certainly going to get a question about the differences between the types so make an effort to learn them. More importantly the smart interviewee understands what the implication of these features are. Why would you choose one over the other? What are the implications?

Which to choose?

In reality, whether you choose a Set, List or Map will depend on the structure of your data. If you won't have duplicates and you don't need order, then your choice will lie with a set. If you need to enshrine order into your data, then you will choose List. If you have key-value pairs, usually if you want to associate two different objects or an object has an obvious identifier, then you will want to choose a map.

Examples

- A collection of colours would be best put into a set. There is no ordering to the elements, and there are no duplicates; You can't have two copies of "Red"!
- The batting order of a cricket team would be good to put in a list; you want to retain the order of the objects.
- A collection of web sessions would be best in a map; the unique session ID would make a good key/reference to the real object.

Understanding why you choose a collection is vitally important for giving the best impression in interviews. I've had candidates come in and tell me they just use ArrayList and HashMap because that's just the default implementation they choose to use. In reality, this maybe isn't a bad thing. Most times we need to use collections there are no major latency or access requirements. It doesn't matter. However, people who are asking you these questions in interviews want to know that you understand how things work under the covers.

Q: What are the main concerns when choosing a collection?

When you're handling a collection, you care about speed, specifically

- Speed of access
- Speed of adding a new element
- Speed of removing an element
- Speed of iteration

On top of this, it isn't just a case of "which is faster". There is also consistency. Some implementations will guarantee the speed of access, whereas others will have variable speed. How that speed is determined depends on the implementation. The important thing to understand is the speed of the relative collection implementations to each other, and how that influences your selection.

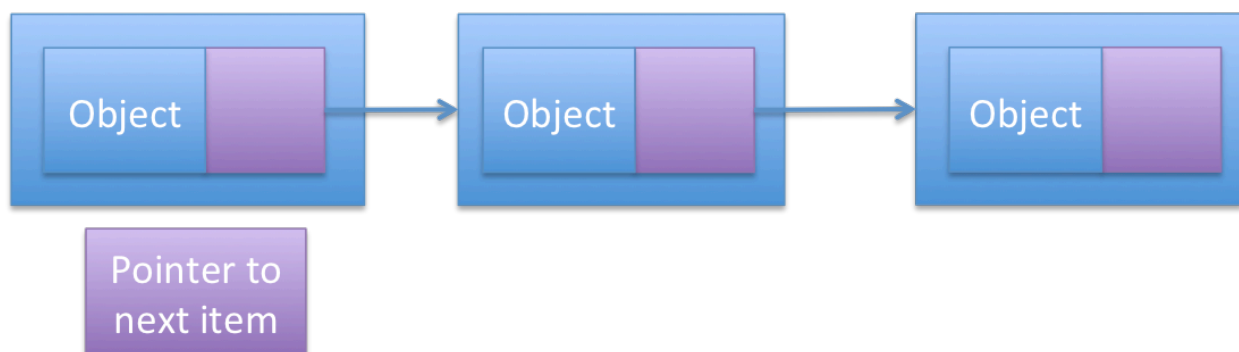
Collection implementations

- Q: Why would I choose `LinkedList` over an `ArrayList`?
- Q: Which is faster, `TreeSet` or `HashSet`?
- Q: How does a `LinkedList` work?

The collection implementations tend to be based on either a *Linked* implementation, a *Tree* implementation or a *Hash* implementation. There are entire textbooks based on this, but what you need to know is how it affects your collection choices.

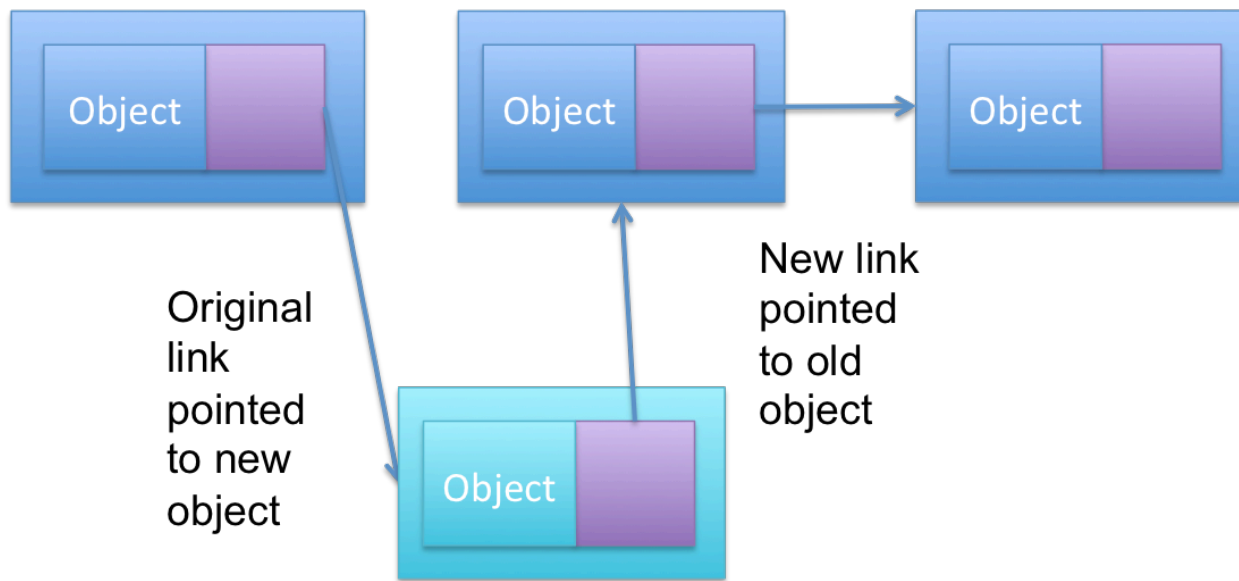
Linked implementation

Example: `LinkedList`, `LinkedHashSet` Under the covers, each item is held in a "node". Each node has a pointer to the next item in the collection like so.



What does this mean for speed?

When adding an item into a linked connection it's quick, irrelevant of where you're adding the node. If you have a list of 3 items and want to add a new one in position 1, the operation is simply to point position 2 to the new item, and the new item to the old position 2.



That's pretty fast! No copying collections, no moving things, no reordering. The same is true for removals; simply point node 0 to node 2 and you've removed node 1.

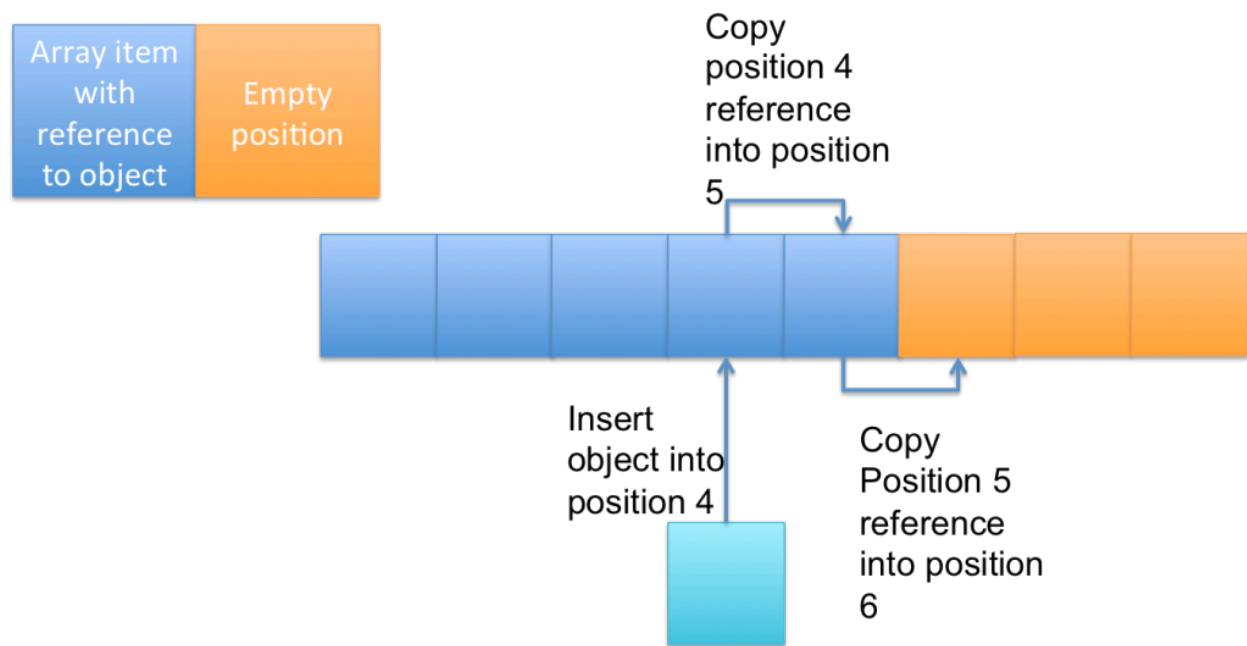
Conversely when accessing objects it is relatively slow. If you have a list of 1000 items, and you want item 999, you need to traverse every single node to get to that item. This also means that the access speed is not consistent. It's quicker to access element 1 than it is element 1000. Bear in mind that when adding a new object you need to first traverse to the node before, so this can have some impact on speed.

So linked collections such as `LinkedList` are great when you need fast addition/removal and access time is less of a concern. **If you're going to be adding/remove items from your collection a lot, then this is the option for you.**

Array implementation

Example: `ArrayList`. `ArrayList` is the only `Array` based implementation in the collection classes, but is often used to compare to `LinkedList` so it's important to understand. As alluded to previously, `ArrayLists` are backed by an `Array`. This has interesting implications.

When adding an item into the middle of an `ArrayList`, the structure needs to copy all of the items to shift them down the `Array`. This can be slow, and is not guaranteed time (depending on how many elements need to be copied).



The same applies when removing objects; all object references after it have to be copied forward one space. There is an even worse case. On creating an ArrayList it starts with a fixed length Array (whose length you can set in the constructor, or the default is 10). If the capacity needs to increase over this, the collection has to create a **brand new array** of greater length and copy all the references to it which can be really slow.

Where ArrayList succeeds is access speed; as the array is in contiguous memory it means that if you request object 999 of 1000, you can calculate exactly where in memory the reference is with no need to traverse the full collection. This gives constant time access which is **fast**.

So, ArrayLists make a great choice if you have a set of data that is **unlikely to be modified significantly**, and you need speedy read access.

Hash implementation

Examples: HashSet, HashMap. The HashMap is a complicated beast in itself and is a good choice for an interview question as it is easy to add extending questions to.

Q: How does a HashMap work?

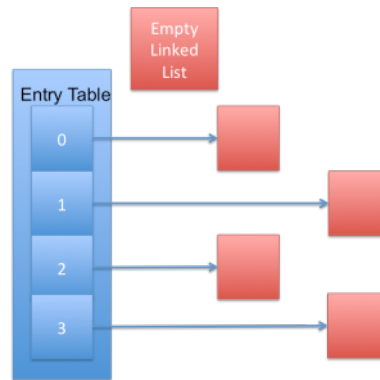
Q: Why does the hashCode of the key matter?

Q: What happens if two objects have the same hashCode?

Q: How does the Map select the correct object if there is a clash?

Interestingly HashSet is backed by HashMap, so for the purpose of this I will just discuss the latter. A HashMap works like this: under the covers a HashMap is an array of references (called the Entry

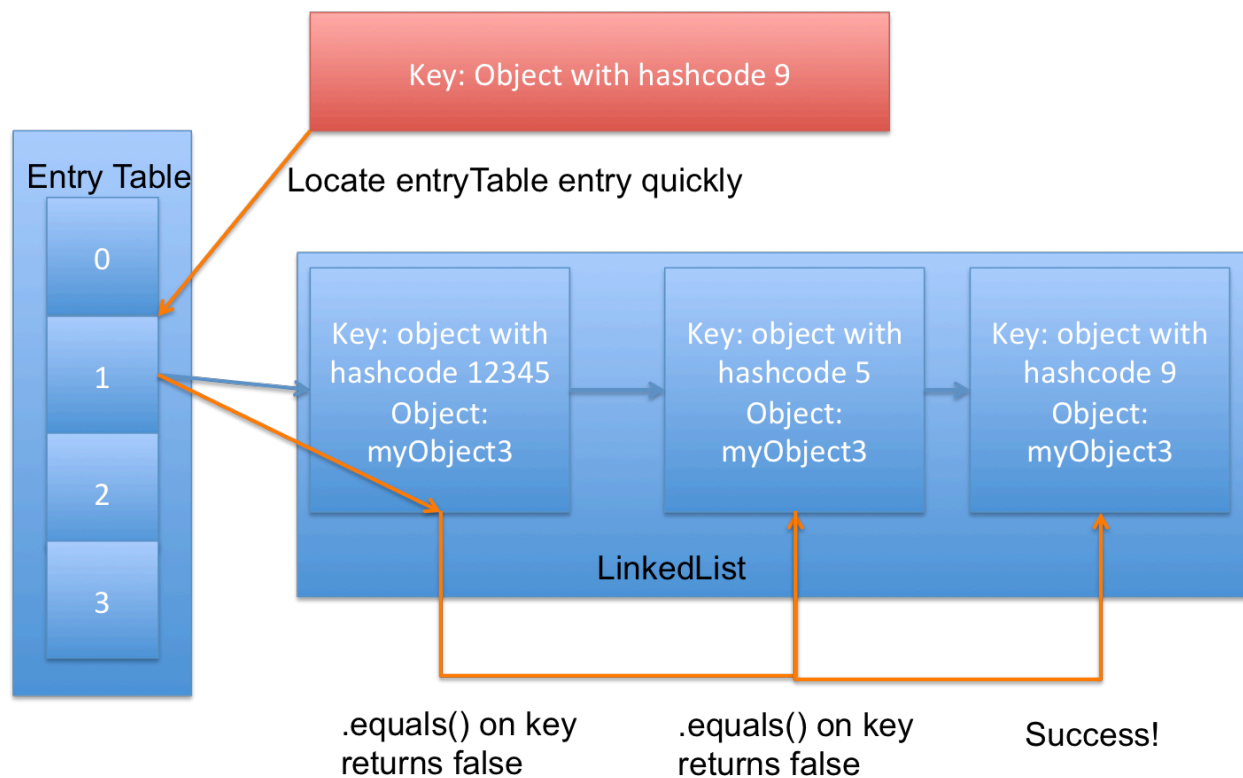
Table, which defaults to 16 entries) to a group of LinkedLists (called Buckets) where HashEntries are stored. A HashEntry is an object containing the key and associated object.



All instances of Object have a method, `hashCode()`, which returns an int. The hashCode is usually a value derived from the properties of an object. The hashCode returned from an object should be consistent and equal objects must return the same hashCode. This property can then be used to determine which bucket to put it in based on the index of the entry table; for example, in the above example with 4 spaces in the entry table, if I have a key in with hashCode 123456, I can do $123456 \% 4 = 1$, so I would place my object in bucket 1.

This makes object retrieval very quick. In my example, if I try to retrieve the object for my key then all I need to do is the same mod calculation to find the bucket and it's the first entry. Exceptionally quick.

This can get more complicated. What if I have two more objects, hashCode of "5" and "9"? They both have a (modulus 4) of 1 so would both be put into bucket 1 too. Now when I use my "9" key to retrieve my object the HashMap has to iterate through a LinkedList, calling `.equals()` on the keys of each HashEntry.



This is slow! Fortunately, this shouldn't happen often. Although collisions will naturally occur, the intention is that objects will be spread evenly across the entry table with minimal collisions. This relies on a well implemented hashCode function. If the hashCode is poor (e.g., always return 0) then you will hit collisions.

To further try and reduce the chance of this, each HashMap has a load factor (which defaults to 0.75). If the entry table is more full than the load factor (e.g. if more than 75% of the entry table has entries) then, similar to the ArrayList, the HashMap will double the size of the entry table and reorder the entries. This is known as rehashing.

Tree implementation

Examples: TreeMap, TreeSet. The Tree implementation is completely different to any of the data structures we've considered before and is considerably more complicated. It is based on a Red Black tree, a complicated algorithm which you're unlikely to get asked about (if you're really keen then watch http://www.csanimated.com/animation.php?t=Red-black_tree). Values are stored in an ordered way, determined either by the natural ordering of the object or using a **Comparator** passed in as part of the constructor. This means on all implementations the collection will re-sort itself to maintain this natural order. When retrieving items it is necessary to navigate the tree, potentially traversing a large number of nodes for sizeable collections. The important thing to note from your perspective: **this is slower than HashMap**. Considerably slower. If speed is the goal then always

use a `HashMap` or `HashSet`. However, if ordering is important then `TreeMap`/`TreeSet` should be your collection of choice.

Multithreading and collections

Collections and Threading are intrinsically linked; many of the core problems related to Threading are regarding shared data access. How can we access data from multiple threads in a fast yet safe fashion?

How do I create a Thread Safe collection?

The easiest way to make something threadsafe is to make it immutable. An immutable object is one whose state cannot be altered. If it cannot change then it is safe for all threads to read it without any problems. This can be achieved in Java using the `Collections.unmodifiable` methods.

```
1 Collections.unmodifiableCollection
2 Collections.unmodifiableSet
3 Collections.unmodifiableSortedSet
4 Collections.unmodifiableList
5 Collections.unmodifiableCollection
6 Collections.unmodifiableMap
7 Collections.unmodifiableSortedMap
```

All of these collections act as a wrapper around a standard collection and guarantee access safety; any attempts to modify the collection will result in an `UnsupportedOperationException`. However, this is only a *view* on the data. If everyone is accessing via the unmodifiable version, then this is thread safe; however the underlying collection could still change. For example:

```
1 LinkedList<String> words = new LinkedList<>();
2 words.add("Hello");
3 words.add("There");
4 List<String> unmodifiableList = Collections.unmodifiableList(words);
5 System.out.println(unmodifiableList);
6 words.add("Cheating");
7 System.out.println(unmodifiableList);
8 //result:
9 //[Hello, There]
10 //[Hello, There, Cheating]
```

As a result this isn't really immutable. Such things as a truly `ImmutableList` do exist in third party libraries such as Google Guava, but not in the core language. It's also important to note that

immutable collections don't normally satisfy the needs of our programs; often data needs to be written and read, so we need a bigger solution.

In Java 5, the `java.util.concurrent` package was introduced with the specific aim of making life easier for dealing with collections in a thread safe way.

Q: Tell me about the `java.util.concurrent` package that was introduced in Java 5. Why is it important? Why does it matter?

The guys who write Java are smarter than you or me (or they should be). Instead of everyone having to write their own implementation to ensure thread safety, making sure the right things were locked in the right places without destroying performance, `java.util.concurrent` provides a number of collection types which guarantee thread safety with the minimum performance impact.

It also introduced a bunch of other cool Thread related features, which will be covered in the Threading chapter.

The queues

- `ConcurrentLinkedDeque`
- `ConcurrentLinkedQueue`
- `LinkedBlockingQueue`
- `LinkedBlockingDeque`
- `LinkedTransferQueue`
- `PriorityBlockingQueue`
- `ArrayBlockingQueue`

That's a lot of queues (and double ended queues, a deque)! The key things to note:

- A Blocking queue will block a consumer/producer when the queue is empty/full. This is a necessary mechanism to allow a producer/consumer pattern. The `BlockingQueue` interface has `put()` and `take()` methods to allow for this.
- A Transfer queue is a blocking queue, with extra methods to allow a consumer to wait for a message to be consumed.
- A Priority queue will order elements based on their natural ordering or using a comparator passed in at construction time.

All of the implementations use Compare and Set (CAS) operations, not synchronized blocks to ensure thread safety. This ensures no risk of deadlock and Threads do not get stuck waiting, and is also significantly faster. All collections based on CAS operations, such as the above, have weakly consistent iterators. This means that they will not throw `ConcurrentModificationExceptions` but they make no guarantee to reflect the correct state; from the documentation:

The returned iterator is a “weakly consistent” iterator that will never throw `ConcurrentModificationException` and guarantees to traverse elements as they existed upon construction of the iterator, and may (but is not guaranteed to) reflect any modifications subsequent to construction.

The CopyOnWrites

Q: How can ConcurrentModificationException be avoided when iterating a collection?

- CopyOnWriteArrayList
- CopyOnWriteArraySet

I'm a big fan of well named Classes, and this is a great example. By copying the entire collection on every write it guarantees immutability; if I have an iterator looping through a collection then it can guarantee that the collection will not be modified as it is being traversed as all modifications will result in a copy of the data being created. This can be an expensive operation so these collections lend themselves better to data that has frequent access but infrequent update. An iterator which relies on the data being copied, such as in this case, is known as a fail safe iterator.

ConcurrentHashMap

Q: How is ConcurrentHashMap implemented?

ConcurrentHashMap is a highly efficient, thread safe implementation of HashMap. As discussed earlier, a HashMap is made up of segments, a number of lists (defaulting to 16) which contain the data which are accessed based on hashCode. In ConcurrentHashMap read operations are generally not blocking which allows for it to be very speedy; writes only block by segment. This means that the other 15 segments can be read without issue if an update is being written.

Q: Should I use a HashTable or a ConcurrentHashMap?

HashTable is a synchronized implementation of HashMap but it is from an old version of Java and should no longer be used. All operations are synchronized in HashTable which means it's performance is very poor relative to ConcurrentHashMap.

Q: Are collections such as ConcurrentLinkedQueue and ConcurrentHashMap completely Thread safe?

The answer is both yes and no. The implementation is completely ThreadSafe and can be used as a replacement for their non Thread safe counterparts. However, it is very possible to write code which access the collection in a non Thread safe manner.

```
1 Map<String, String> map = new ConcurrentHashMap<>();
2 map.put("a", "1");
3 map.put("b", "2");
4 //....Set off a bunch of other threads....
5 if (!map.containsKey("c"))
6     map.put("c", "3");
7 else
8     map.get("c");
```

The contains -> put and contains -> get operations are not atomic. Between checking for existence and putting/getting anything can happen in the collection. The collection is still thread safe, our access simply isn't. We need to ensure any operations where we check state and then execute are done in an atomic or synchronized way. The above code could be rectified by making use of the `putIfAbsent` method to create atomicity; however if for example you wanted to perform an action to a list based on it's size you may need to lock the collection to ensure it happens as a single operation.

Java exceptions

Java exceptions are one of my favourite questions to ask during telephone interviews. They're one of the simpler parts of Java so I would expect most candidates to be able to give good answers. It's normally easy to weed out those who don't as not worth my time. However what I really like about exceptions as interview fodder is that they allow the combination of a technical core java question with an opinion piece. Checked exceptions are actually a really controversial part of Java and it's good to see which side of the fence a candidate falls on, or whether they are even aware that the fence is there.

Core questions

Q: What is an Exception in Java?

I don't normally ask this question as it's so basic but it's good to make sure you've got a standard answer in your pocket. Exceptions are a way to programmatically convey system and programming errors. All exceptions inherit from Throwable. When something goes wrong you can use the throw keyword to fire an exception.

Q: There are 2 types of exception in Java, what are they and what's the difference?

It still amazes me how much the "2 types" question throws people (yep, throws, I did it again, another pun). This is as basic as core java gets. If you don't get this then it tells me that you didn't learn core java properly (or at all) which will probably act as a shaky base for the rest of your knowledge. Get this answer right. The two types of exception are **checked** and **unchecked**. I've heard some strange answers in my time including "Runtime and NullPointerException"! A checked exception is one that forces you to catch it. It forms part of the API or contract. Anyone using code that throws a checked exception can see that as it is declared on the method and they are forced to handle it using a try/catch block. Unchecked exceptions do not need to be caught and do not notify anyone using the code that it could be thrown.

Q: How do I know if an Exception class is checked or unchecked?

All exceptions are checked exceptions except those that inherit from `java.lang.RuntimeException`.

Q: What does "finally" do?

Exceptions are caught using a try-catch block. However following the catch you can have a finally block. This block is guaranteed to execute no matter what happens (short of someone pulling out the plug), such as if an exception is thrown in the catch block. This is really useful to ensure the clean up of resources such as Connections.

Q: Can I throw multiple exceptions from the same method? How do I catch them?

Yes you can, each exception (or a parent of that class) just need listing after the “throws” in the method signature. You can catch them using multiple catch blocks, or by catching a parent exception class e.g. you can just catch Exception which will catch all exceptions. This is in theory bad practice as you lose the nuance of what went wrong and specifically the ability to recover. You can also catch multiple exceptions in a single catch block as of Java 7. When catching exceptions it is important to do so in order from **most specific to least specific**. If your catch block catches Exception first and then IOException second, the first catch block will always catch any exception coming through and the IOException will be rendered useless (and in fact it will cause a compiler error saying the exception has already been caught.)

Q: What do you think of Checked Exceptions, are they a good thing?

Q: When you’re writing your own code and you need to throw a custom exception, do you create a checked or unchecked exception?

Q: C# does not have checked exceptions. Can you tell me why this might be? Who do you think was right, Java or C sharp?

I love these questions so much. It’s a really good way to stretch a candidate and to see if they understand how to architect systems. Having an appropriate policy on exceptions is key to maintaining a clean code base, and you’ll quickly figure out who’s thought about the issue and who hasn’t. A must read before your interview is this interview with the creator of C# on [why he didn’t include checked exceptions in the language](http://www.artima.com/intv/handcuffs.html).¹ The argument in general is that checked exceptions cause ugly code in Java. Handling is incredibly verbose, and normally most developers don’t know how to handle them or are too lazy to. This results in empty catch blocks or just log statements. This results in a brittle system where exceptions get swallowed never to be heard from again and the system can get into a brittle state without telling anyone. On the positive side for checked, it declares on the API what it’s going to do so consumers can handle it. This is good if you’re handing out a library. However a lot of developers use this for evil, and control the flow of their system using exceptions. Exceptions are for exceptional circumstances only.

As with most discussion questions the important part is not that the developer agrees with your opinion but that they are capable of forming an opinion and explaining it. I personally fall heavily on the side that checked exceptions are terrible and developers misuse them. In my code all checked exceptions get wrapped in an application specific RuntimeException and thrown up the stack where I’ll normally have an exception guard to catch and log the issue. However if a candidate says they use checked exceptions in their code and can explain why with an understanding of the problems then they get a pass. Just make sure you’ve thought about where you lie on the issue before you go in for your interview so you can articulate it to the interviewer. And if you’ve never thought about it before, then remember checked exceptions are evil :).

¹<http://www.artima.com/intv/handcuffs.html>

JVM and garbage collection

The Java Virtual Machine is the achilles heel of most developers and can cause even the most seasoned developers to come unstuck. The simple fact is that unless something is going wrong, we don't normally care about it. Maybe we tune it a little when the application goes live but after that it remains untouched until something goes wrong. This makes it a very difficult subject to excel in during interviews. Even worse, interviewers love to ask questions about it. Everyone should have a basic knowledge of the JVM to be able to do their job but often people recruiting are looking for someone who knows how to fix a problem like a memory leak when it happens.

In this guide we take a ground up approach to the JVM and garbage collection so you can feel some level of confidence going into your big day.

Q: What is the JVM? Why is it a good thing? What is “write once, run anywhere”? Are there negatives?

JVM stands for Java Virtual Machine. Java code is compiled down into an intermediary language called byte code. The Java Virtual Machine is then responsible for executing this byte code. This is unlike languages such as C++ which are compiled directly to native code for a specific platform.

This is what gives Java its ‘write once, run anywhere’ ability. In a language which compiles directly to native you would have to compile and test the application separately on every platform on which you wish it to run. There would likely be several issues with libraries, ensuring they are available on all of the platforms for example. Every new platform would require new compilation and new testing. This is time consuming and expensive.

On the other hand a java program can be run on any system where a Java Virtual Machine is available. The JVM acts as the intermediary layer and handles the OS specific details which means that as developers we shouldn't need to worry about it. In reality there are still some kinks between different operating systems, but these are relatively small. This makes it quicker and easier to develop, and means developers can write software on windows laptops that may be destined for other platforms. I only need to write my software once and it is available on a huge variety of platforms, from Android to Solaris.

In theory this is at the cost of speed. The extra layer of the JVM means it is slower than direct-to-tin languages like C. However java has been making a lot of progress in recent years, and given the many other benefits such as ease of use, it is being used more and more often for low latency applications.

The other benefit of the JVM is that any language that can compile down to byte code can run on it, not just java. Languages like Groovy, Scala and Clojure are all JVM based languages. This also means the languages can easily use libraries written in other languages. As a Scala developer I can use Java libraries in my applications as it all runs on the same platform.

The separation from the real hardware also means the code is sandboxed, limiting the amount of damage it can do to a host computer. Security is a great benefit of the JVM.

There is another interesting facet to consider; not all JVMs are built equal. There are a number of different implementations beyond the standard JVM implementation from Oracle. JRockit is renowned for being an exceptionally quick JVM. OpenJDK is an open source equivalent. There are tons of JVM implementations available. Whilst this choice is ultimately a good thing, all of the JVMs may behave slightly differently. A number of areas of the Java specification are left intentionally vague with regards to their implementation and each VM may do things differently. This can result in a bug which only manifests in a certain VM in a certain platform. These can be some of the hardest bugs to figure out.

From a developer perspective, the JVM offers a number of benefits, specifically around memory management and performance optimisation.

Q: What is JIT?

JIT stands for “Just in Time”. As discussed, the JVM executes bytecode. However, if it determines a section of code is being run frequently it can optionally compile a section down to native code to increase the speed of execution. The smallest block that can be JIT compiled is a method. By default, a piece of code needs to be executed 1500 times for it to be JIT compiled although this is configurable. This leads to the concept of “warming up” the JVM. It will be at it’s most performant the longer it runs as these optimisations occur. On the downside, JIT compilation is not free; it takes time and resource when it occurs.

Q: What do we mean when we say memory is managed in Java? What is the Garbage Collector?

In languages like C the developer has direct access to memory. The code references memory space addresses. This can be difficult and dangerous, and can result in damaging memory leaks. In Java all memory is managed. As a programmer we deal exclusively in objects and primitives and have no concept of what is happening underneath with regards to memory and pointers. Most importantly, Java has the concept of a garbage collector. When objects are no longer needed the JVM will automatically identify and clear the memory space for us.

Q: What are the benefits and negatives of the Garbage Collector?

On the positive side:

- The developer can worry much less about memory management and concentrate on actual problem solving. Although memory leaks are still technically possible they are much less common.
- The GC has a lot of smart algorithms for memory management which work automatically in the background. Contrary to popular belief, these can often be better at determining when best to perform GC than when collecting manually.

On the negative side

- When a garbage collection occurs it has an effect on the application performance, notably slowing it down or stopping it. In so called “Stop the world” garbage collections the rest of the application will freeze whilst this occurs. This is can be unacceptable depending on the application requirements, although GC tuning can minimise or even remove any impact.
- Although it’s possible to do a lot of tuning with the garbage collector, you cannot specify when or how the application performs GC.

Q: What is “Stop the World”?

When a GC happens it is necessary to completely pause the threads in an application whilst collection occurs. This is known as Stop The World. For most applications long pauses are not acceptable. As a result it is important to tune the garbage collector to minimise the impact of collections to be acceptable for the application.

Q: How does Generational GC work? Why do we use generational GC? How is the Java Heap structured?

It is important to understand how the Java Heap works to be able to answer questions about GC. All objects are stored on the Heap (as opposed to the Stack, where variables and methods are stored along with references to objects in the heap). Garbage Collection is the process of removing objects which are no longer needed from the Heap and returning the space for general consumption. Almost all GCs are “generational”, where the Heap is divided into a number of sections, or generations. This has proven significantly more optimal which is why almost all collectors use this pattern.

New generation

Q: What is the New Generation? How does it help to minimise the impact of GC?

Most applications have a high volume of short lived objects. Analyzing all objects in an application during a GC would be slow and time consuming, so it therefore makes sense to separate the shortlived objects so that they can be quickly collected. As a result all new objects are placed into the new generation. New gen is split up further:

- **Eden Space:** all new objects are placed in here. When it becomes full, a minor GC occurs. all objects that are still referenced are then promoted to a **survivor space**
- **Survivor spaces:** The implementation of survivor spaces varies based on the JVM but the premise is the same. Each GC of the New Generation increments the age of objects in the survivor space. When an object has survived a sufficient number of minor GCs (defaults vary but normally start at 15) it will then be promoted to the Old Generation. Some implementations use two survivor spaces, a From space and a To space. During each collection these will swap roles, with all promoted Eden objects and surviving objects moved to the To space, leaving From empty.

Q: What is a minor GC?

A garbage collection in the NewGen is known as a **minor GC**. One of the benefits of using a New Generation is the reduction of the impact of fragmentation. When an object is garbage collected, it leaves a gap in the memory where it was. We can compact the remaining objects (a stop-the-world scenario) or we can leave them and slot new objects in. By having a Generational GC we limit the amount that this happens in the Old Generation as it is generally more stable which is good for improving latencies by reducing stop the world. However if we do not compact we may find objects cannot just fit in the spaces inbetween, perhaps due to size concerns. If this is the case then you will see objects failing to be promoted from New Generation.

Old generation

Q: Explain what the old generation is?

Any objects that survive from survivor spaces in the New Generation are promoted to the Old Generation. The Old Generation is usually much larger than the New Generation. When a GC occurs in old gen it is known as a **full GC**. Full GCs are also stop-the-world and tend to take longer, which is why most JVM tuning occurs here. There are a number of different algorithms available for Garbage Collection, and it is possible to use different algorithms for new and old gen.

Q: What type of Garbage Collectors are there?*Serial GC*

Designed when computers only had one CPU and stops the entire application whilst GC occurs. It uses **mark-sweep-compact**. The collector goes through all of the objects and marks which objects are available for Garbage Collection, before clearing them out and then copying all of the objects into contiguous space (so therefore has no fragmentation).

Parallel GC

Similar to Serial, except that it uses multiple threads to perform the GC in order to be faster.

Concurrent Mark Sweep (CMS)

CMS GC minimises pauses by doing most of the GC related work concurrently with the processing of the application. This minimises the amount of time when the application has to completely pause and so lends itself much better to applications which are sensitive to this. CMS is a non compacting algorithm which can lead to fragmentation problems. The CMS collector uses Parallel GC for the young generation.

G1GC (garbage first garbage collector)

A concurrent parallel collector that is viewed as the long term replacement for CMS and does not suffer from the same fragmentation problems as CMS.

Q: Which is better? Serial, Parallel or CMS?

It depends entirely on the application. Each one is tailored to the requirements of the application. Serial is better if you're on a single CPU, or in a scenario where there are more VMs running on the machine than CPUs. Parallel is a throughput collector and really good if you have a lot of work to do but you're ok with pauses. CMS/G1GC is the best of the options if you need consistent responsiveness with minimal pauses.

PermGen

Q: What is the PermGen?

The PermGen is where the JVM stores the metadata about classes. It no longer exists in Java 8, having been replaced with metaspace. Generally the PermGen doesn't require any tuning above ensuring it has enough space, although it is possible to have leaks if Classes are not being unloaded properly.

From the code

Q: Can you tell the system to perform a garbage collection?

This is an interesting question: the answer is both yes and no. We can use the call "System.gc()" to suggest to the JVM to perform a garbage collection. However, there is no guarantee this will do anything. As a java developer, we don't know for certain what JVM our code is being run in. The JVM spec makes no guarantees on what will happen when this method is called. There is even a startup flag, -XX:+DisableExplicitGC, which will stop this from doing anything.

It is considered bad practice to use System.gc().

Q: What does finalize() do?

finalize() is a method on java.lang.Object so exists on all objects. The default implementation does nothing. It is called by the garbage collector when it determines there are no more references to the object. As a result there are no guarantees the code will ever be executed and so should not be used to execute actual functionality. Instead it is used for clean up, such as file references. It will never be called more than once on an object (by the JVM).

JVM tuning

Q: What flags can I use to tune the JVM and GC?

There are textbooks available on tuning the JVM for optimal Garbage Collection. You'll never know them all! Nonetheless it's good to know a few for the purpose of interview.

-XX:-UseConcMarkSweepGC: Use the CMS collector for the Old Gen.

-XX:-UseParallelGC: Use Parallel GC for New Gen

-XX:-UseParallelOldGC: Use Parallel GC for Old and New Gen.

-XX:-HeapDumpOnOutOfMemoryError: Create a thread dump when the application runs out of memory. Very useful for diagnostics.

-XX:-PrintGCDetails: Log out details of Garbage Collection.

-Xms512m: Sets the initial heap size to 512m -Xmx1024m: Sets the maximum heap size to 1024m

-XX:NewSize and -XX:MaxNewSize: Specifically set the default and max size of the New Generation

-XX:NewRatio=3: Set the size of the Young Generation as a ratio of the size of the Old Generation.

-XX:SurvivorRatio=10: Set the size of Eden space relative to the size of a survivor space.

Diagnosis

Whilst all of the questions above are very good to know to show you have a basic understanding of how the JVM works, one of the most standard questions during an interview is this:

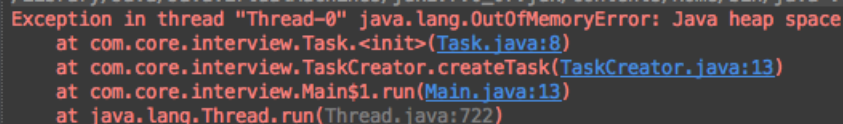
Have you ever experienced a memory leak? How did you diagnose it?

This is a difficult question to answer for most people as although they may have done it, chances are it was a long time ago and isn't something you can recall easily. The best way to prepare is to actually try and write an application with a memory leak and attempt to diagnosis it. Below I have created a ridiculous example of a memory leak which will allow us to go step by step through the process of identifying the problem. **I strongly advise you download the code and follow through this process.** It is much more likely to be committed to your memory if you actually do this process.

```
1  import java.util.ArrayDeque;
2  import java.util.Deque;
3  public class Main {
4      public static void main(String[] args) {
5          TaskList taskList = new TaskList();
6          final TaskCreator taskCreator = new TaskCreator(taskList);
7          new Thread(new Runnable() {
8              @Override
9              public void run() {
10                 for (int i = 0; i < 100000; i++) {
11                     taskCreator.createTask();
12                 }
13             }
14         }).start();
15     }
16     private static class TaskCreator {
```

```
17     private TaskList taskList;
18     public TaskCreator(TaskList taskList) {
19         this.taskList = taskList;
20     }
21     public void createTask() {
22         taskList.addTask(new Task());
23     }
24 }
25 private static class TaskList {
26     private Deque<Task> tasks = new ArrayDeque<Task>();
27     public void addTask(Task task) {
28         tasks.add(task);
29         tasks.peek().execute();//Memory leak!
30     }
31 }
32 private static class Task {
33     private Object[] array = new Object[1000];
34     public void execute() {
35         //dostuff
36     }
37 }
38 }
```

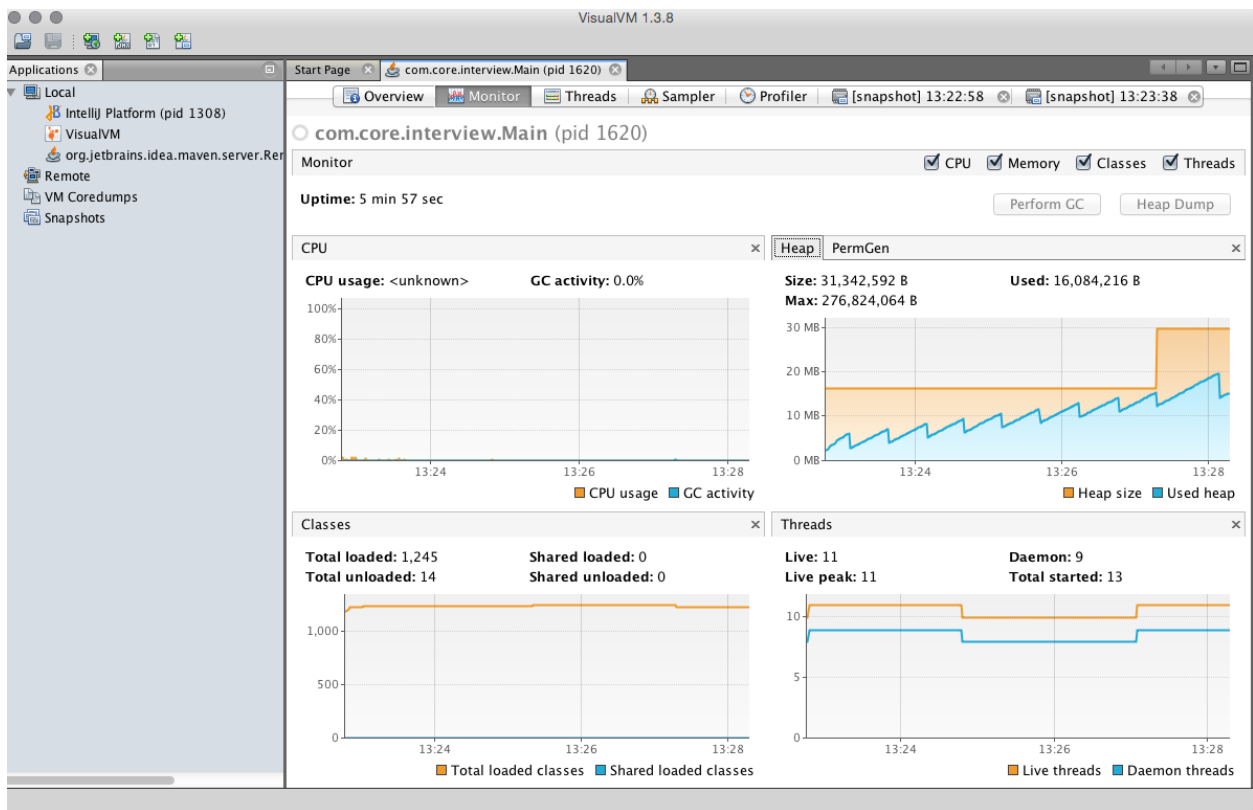
In the above very contrived example, the application executes tasks put onto a Deque. However when we run this we get an out of memory! What could it possibly be?



```
Exception in thread "Thread-0" java.lang.OutOfMemoryError: Java heap space
at com.core.interview.Task.<init>(Task.java:8)
at com.core.interview.TaskCreator.createTask(TaskCreator.java:13)
at com.core.interview.Main$1.run(Main.java:13)
at java.lang.Thread.run(Thread.java:722)
```

To find out we need to use a profiler. A profiler allows us to look at exactly what is going on the VM. There are a number of options available. VisualVM (<https://visualvm.java.net/download.html>) is free and allows basic profiling. For a more complete tool suite there are a number of options but my personal favourite is Yourkit. It has an amazing array of tools to help you with diagnosis and analysis. However the principles used are generally the same.

I started running my application locally, then fired up VisualVM and selected the process. You can then watch exactly what's going on in the heap, permgen etc.



You can see on the heap (top right) the tell tail signs of a memory leak. The application sawtooths, which is not a problem per se, but the memory is consistently going up and not returning to a base level. This smells like a memory leak. But how can we tell what's going on? If we head over to the Sampler tab we can get a clear indication of what is sitting on our heap.

Class Name - Allocated Objects	Bytes Allocated [%]	Bytes Allocated	Objects Allocated
<code>java.lang.Object[]</code>		1,586,944 B (26.8%)	7,584 (9.6%)
<code>char[]</code>		950,568 B (16%)	11,509 (14.6%)
<code>byte[]</code>		847,208 B (14.3%)	5,311 (6.7%)
<code>int[]</code>		592,904 B (10%)	3,218 (4.1%)
<code>java.lang.String</code>		195,600 B (3.3%)	8,150 (10.3%)
<code>java.lang.Class</code>		166,704 B (2.8%)	1,383 (1.8%)
<code>java.io.ObjectStreamClass\$WeakClassKey</code>		146,624 B (2.5%)	4,582 (5.8%)
<code>java.util.TreeMap\$Entry</code>		132,560 B (2.2%)	3,314 (4.2%)

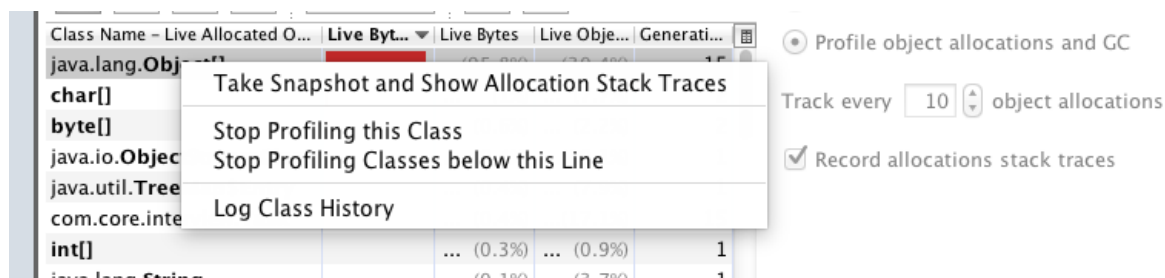
Those Object arrays look a bit odd. But how do we know if that's the problem? Visual VM allows us to take snapshots, like a photograph of the memory at that time. The above screenshot is a snapshot from after the application had only been running for a little bit. The next snapshot a couple of minutes later confirms this:

Class Name – Allocated Objects	Bytes Allocated [%]	Bytes Allocated	Objects Allocated
java.lang.Object[]		2,910,952 B (59.2%)	1,944 (5%)
char[]		385,184 B (7.8%)	5,519 (14.2%)
byte[]		286,208 B (5.8%)	1,577 (4%)
java.lang.Class		167,600 B (3.4%)	1,391 (3.6%)
int[]		166,504 B (3.4%)	2,621 (6.7%)
java.lang.String		130,008 B (2.6%)	5,417 (13.9%)
short[]		109,104 B (2.2%)	1,858 (4.8%)
int[][]		108,080 B (2.2%)	2,097 (5.4%)
java.lang.reflect.Method		90,800 B (1.8%)	1,135 (2.9%)
java.util.HashMap\$Entry[]		41,360 B (0.8%)	465 (1.2%)

We can actually compare these directly by selecting both in the menu and selecting compare.



There's definitely something funky going on with the array of objects. How can we figure out the leak though? By using the profile tab. If I go to profile, and in settings enable "record allocations stack traces" we can then find out where the leak has come from.



By now taking snapshot and showing allocation traces we can see where the object arrays are being instantiated.

Method Name – Allocation Call Tree	Live Bytes ...	Live Bytes	Live Objects	Allocated Obj...	Avg. Age	Generations
java.lang.Object[]		11,... (100%)	7,9... (100%)	20,558	2.3	15
com.core.interview.Task.<init> ()		11,...(98.5%)	2,9...(36.6%)	2,926	6.4	15
java.io.ObjectOutputStream.default		65,... (0.5%)	3,5...(44.7%)	12,442	0.0	1
java.io.ObjectOutputStream\$Handle		57,... (0.5%)	66 (0.8%)	195	0.0	1
com.sun.jmx.mbeanserver.DefaultM		17,... (0.1%)	325 (4.1%)	1,132	0.0	1
java.io.ObjectStreamClass.invokeW		9,4... (0.1%)	392 (4.9%)	1,435	0.0	1
java.io.ObjectOutputStream\$Handle		7,6... (0.1%)	136 (1.7%)	443	0.0	1
java.io.ObjectInputStream\$HandleT		4,4... (0%)	80 (1%)	261	0.0	1
java.io.ObjectOutputStream\$Replac		3,6... (0%)	66 (0.8%)	223	0.0	1
java.util.ArrayList.<init> (int)		3,6... (0%)	68 (0.9%)	238	0.0	1

Looks like there are thousands of Task objects holding references to Object arrays! But what is holding onto these Task items?

If we go back to the “Monitor” tab we can create a heap dump. If we double click on the Object[] in the heap dump it will show us all instances in the application, and in the bottom right panel we can identify where the reference is.

The screenshot shows the IntelliJ IDEA heap dump viewer. The top bar indicates the selected object is `java.lang.Object[]` with 41,929 instances and a total size of 330,685,104 bytes. A link to "Compute Retained Sizes" is available.

The "Instances" panel on the left shows a list of object instances. The selected instance is `#33558 65,536 items` with a size of 524,312 bytes. Other instances are listed with their sizes (e.g., #30: 8,216 bytes, #56: 8,024 bytes).

The "Fields" panel on the right shows the structure of the selected `Object[]` instance. It contains a `this` field of type `Object[]` pointing to `#33558 65,536 items`, and several `<items ...>` fields, each of type `Object` and containing 500 items.

The "References" panel at the bottom right shows the references to the selected object. It lists a `this` field of type `Object[]` pointing to `#33558 65,536 items`, and an `elements` field of type `ArrayDeque` pointing to `#33`. The `tasks` field of type `TaskList` is highlighted, pointing to `class TaskList`.

It looks like `TaskList` is the culprit! If we take a look at the code we can see what the problem is.

```
1 tasks.peek().execute();
```

We’re never clearing the reference after we’ve finished with it! If we change this to use `poll()` then the memory leak is fixed.

Whilst clearly this is a very contrived example, going through the steps will refresh your memory for if you are asked to explain how you would identify memory leaks in an application. Look for memory continuing to increase despite GCs happening, take memory snapshot and compare them to see which Objects may be candidates for not being released, and use a heap dump to analyse what is holding references to them.

Threading

As a developer the single best thing we can do for producing clean, understandable code that won't fall over is to avoid putting any multi threading into your code base. The best programmers can break most problems down to be solved without complex threading. Poor programmers will throw threading in to fix a problem they don't understand or to prematurely optimise. Threading questions are exceptionally popular for interviews so it's important to be able to demonstrate a sound understanding of how threads work in Java, when and why we use them, and how to actually program them in a safe manner. In this article we will tackle threading from the ground up and cover what you need to know to be able to answer any questions that come up.

Introduction to threading

Q: What is a Thread?

Q: How do we create Threads in java?

Threads allow us to do more than one thing at once. Without Threads code is executed in a linear fashion, one command after another. By using threads we can perform multiple actions at the same time. This takes advantage of the fact that computers now have multiple processors as well as the ability to time slice tasks, so multiple threads can even be run on a single processors. There are 2 main ways to create a new Thread in Java. Firstly, you can extend the Thread class and override the run method to execute your code.

```
1 public class SayHello extends Thread {
2
3     public static void main(String[] args) {
4         new SayHello().start();
5     }
6
7     @Override
8     public void run() {
9         for (int i = 0; i < 100; i++) {
10             System.out.println("Hi there!");
11         }
12     }
13 }
```

Generally we avoid this. Java doesn't have multiple inheritance so this option will limit your ability to extend anything else. More importantly, it's just not a very pretty way of doing it. As good developers we aim to favour composition over inheritance. Option two is much nicer, allowing us to implement the Runnable interface and pass this to a new Thread object. The interface has one method on it, run.

```
1 public class SayHelloRunner implements Runnable {
2     public static void main(String[] args) {
3         new Thread(new SayHelloRunner()).start();
4     }
5
6     @Override
7     public void run() {
8         for (int i = 0; i < 100; i++) {
9             System.out.println("Hi there!");
10        }
11    }
12 }
```

We pass the new Runnable to a Thread to start it. We always use **start()** to run the thread; using **run()** will run execute in the same thread as the caller.

Q: How do we stop a Thread in java?

There is actually no API level way to stop a Thread reliably in java. The only guarantee is that when the code being executed in the thread completes, the thread will finish. It is therefore down to the developer to implement a way of telling a Thread to stop.

```
1 public class SayHelloRunner implements Runnable {
2     private volatile boolean running = true;
3
4     public void stopIt(){
5         running = false;
6     }
7
8     @Override
9     public void run() {
10        while(running)
11            System.out.println("Hi there!");
12    }
13 }
```

As you can see in the above example the flag to check if the thread is running or not has the keyword **volatile**.

Q:What is the volatile keyword?

In modern processors a lot goes on to try and optimise memory and throughput such as *caching*. This is where regularly accessed variables are kept in a small amount of memory close to a specific processor to increase processing speed; it reduces the amount the processor has to go to disk to get information (which can be very slow). However, in a multithreaded environment this can be very dangerous as it means that two threads could see a different version of a variable if it has been updated in cache and not written back to memory; *processor A* may think `int i = 2` whereas *processor B* thinks `int i = 4`. This is where **volatile** comes in. It tells Java that this variable could be accessed by multiple threads and therefore should not be cached, thus guaranteeing the value is correct when accessing. It also stops the compiler from trying to optimise the code by reordering it.

The downside is that there is a performance penalty on using volatile variables. There is a greater distance to travel to get access to the information as it cannot be stored in the cache. However volatile is usually a much better option speed wise than using a synchronize block as it will not cause threads to block and as such is much faster than other options for synchronisation.

Q: Explain the synchronized keyword. What does it do? Why do we need it?

Synchronized is a keyword in java that can be used as a block around a piece of code or as part of the method signature.

```
1 public class Bank {
2     private int funds = 100;
3
4     public void deposit(int money){
5         synchronized (this){
6             funds += money;
7         }
8     }
9
10    public synchronized void withdraw(int money){
11        if(funds > money)
12            funds -= money;
13    }
14 }
```

Synchronized exists in Java to allow multiple threads which can both see an object to safely access it to ensure that the data is correct. The classic example is that of the bank account. Imagine if you remove the synchronisation from the above example. Thread one attempts to remove 100 from the account. At the exact same time, Thread two attempts to remove 100 from the account. For both threads when checking if there are sufficient funds the if statement returns true, resulting in them both withdrawing and a resultant balance of -100 which is not allowed. By using synchronized only a single thread can access the section of code in the synchronized block, which can help us to ensure correct state.

When a synchronized keyword is placed on a method such as on `withdraw(int money)`, it has the same effect as wrapping all of a method's code in `synchronized(this)` (which is the case in the `deposit` method in the example). If any Thread tries to execute any synchronized method on the object it will be blocked.

Q: What are the downsides of using synchronized? How can we avoid or mitigate them?

Locks are slow. Really slow. Particularly when using the synchronized keyword, it can have a huge effect on performance. The reason for this is explained wonderfully in a [paper LMAX created²](http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf) on how they built Disruptor, a super low latency concurrency component. It is a brilliant read for anyone who wants to get into the fine details of threading and mechanical sympathy.

“Locks are incredibly expensive because they require arbitration when contended. This arbitration is achieved by a context switch to the operating system kernel which will suspend threads waiting on a lock until it is released. During such a context switch, as well as releasing control to the operating system which may decide to do other house-keeping tasks while it has control, execution context can lose previously cached data and instructions. This can have a serious performance impact on modern processors.”

You don't need to learn the contents of the details of the quote; it should be sufficient to say that along with the impact of threads being blocked whilst they wait for a resource, there is an impact at an OS level which causes huge performance damage. In the same paper an experiment is carried out to see the impact on latency of different types of lock; without locking the process took 300ms, whereas with 2 threads contesting a lock it took 224,000ms. Getting threading right is hard but getting it wrong has huge consequences.

If you insist on using synchronized then minimising the size of the code block that is synchronized is a good start. The smaller this is will help to minimise impact. Even better, you can lock on specific objects, or use a lock object if using a primitive, so that the entire object does not need to be contended.

```
1 public class Bank {
2     private int funds = 100;
3     private Object fundLock;
4
5     private UserDetails details = new UserDetails();
6
7     public void deposit(int money){
8         synchronized (fundLock){ //Lock Object needs to be acquired to update fu\
9 nds
10             funds += money;
11         }
12     }
13 }
```

²<http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf>

```
14     public void setUsername(){
15         synchronized (details){ //Lock on the details object mean that the detail\
16 ls object will block, but Bank will not.
17             details.setName("Bob");
18         }
19         System.out.println("His name is Bob");
20     }
21
22 }
```

Ideally though we want to avoid using synchronized where possible.

Q: What other options do we have for creating Thread safe code?

As previously discussed, making a variable volatile is more performant and less complex. However Java has a number of classes that aid us with threading so that we can do it in an efficient way.

The atomic classes

Q: What is CAS? What benefits does it have over Synchronized?

Java has a number of Atomic classes including AtomicInteger, AtomicDouble and AtomicBoolean. These are completely thread safe and do not involve locking, but instead use Compare and Swap (CAS) operations. A CAS operation is an atomic one (it happens as one single operation) where during an update we check that the field value has not changed from when we decided to update it (it hasn't been modified by another thread) and if so it sets the value to the new amount. This is exceptionally quick; CAS is actually an operation available on most processors and no blocking is required. It is still slower than single threaded (the LMAX experiment concluded it was 100x slower) but it is the best available away of ensuring thread safe access to values.

It is important to understand the difference between volatile and CAS. if you try and perform any operation that uses the original value it will render the volatile keyword useless. For example, $i++$ breaks down into $i=i + 1$. In the time between i being read ($i + 1$) and it being written ($i = \text{result}$) another thread can jump in between and update the value. Because the Atomic classes checks and writes values as an atomic operation, it is a much safer option.

Immutability

Immutable objects are brilliant. An immutable object is an object whose state cannot be changed after creation. By this definition all immutable objects are thread-safe. Programming to use immutable objects can be difficult and is a paradigm shift from standard object oriented programming. Functional languages like Scala make heavy use of immutable objects which allows them to scale well and be highly concurrent. The more we can rely on immutable objects the less threading we will need.

Thread methods

Q: What does `yield()` do?

The `Thread` class has a whole bunch of methods on it which you need to know for your interview. Interviewers love testing people on their ability to remember the details around these methods. It is best to read up on them before an interview.

`Yield` gives up the processor. Imagine you have a single CPU only. If a thread refuses to allow other threads CPU time they will remain permanently blocked. By calling `yield()` the thread is generously saying “who else wants a turn?”. In reality it is unreliable and isn’t normally used. The implementation can be different on any system or JVM, and can often be different between Java versions. There is no guarantee of what other thread will get the next access, nor when your thread will next get an opportunity to run.

Q: What does `interrupt()` do?

You may have noticed that a number of `Thread` related methods (most commonly `Thread.sleep()`) force you to catch an `InterruptedException`. If we need to suggest to a `Thread` it needs to stop for whatever reason we can do this by calling `interrupt()` on it. This sets the “interrupted” flag on the target thread. If the target thread chooses to check this flag (using the `interrupted()` or `isInterrupted()`) then it can optionally throw an `Exception` (usually `InterruptedException`). It is predicated on the target `Thread` actually checking this. However there are a number of methods, such as `Thread.sleep()` which automatically poll for this flag. If the target thread is running one of these then an `InterruptedException` will be thrown straight away.

Q: What does `join()` do?

`join()` will simply tell the currently running thread to wait until completion of whichever thread `join()` is called on completes.

```
1 public class SayHelloRunner implements Runnable {
2
3     public static void main(String[] args) throws InterruptedException {
4         Thread thread = new Thread(new SayHelloRunner());
5         thread.start();
6         thread.join();
7         System.exit(3);
8     }
9
10    @Override
11    public void run() {
12        int i = 0;
13        while(i < 10000){
14            System.out.println(i);
15            i++;
16        }
17    }
18 }
```

```
16     }
17 }
18 }
```

In this example if we did not have the `thread.join()` the application will exit without printing anything out. The `join()` call effectively makes the call synchronous; it wants the Thread to finish before proceeding.

Object methods

Q: What do `wait()`, `notify()` and `notifyAll()` do?

`wait()`, `notify()` and `notifyAll()` are used as a means of inter-thread communication. When acquiring a lock on an object it may not be in the required state; perhaps a resource isn't set yet, a value isn't correct. We can use `wait()` to put the thread to sleep until something changes. When that something does change, the awaiting clients can be notified by calling `notify()` or `notifyAll()` on the object that is being waited for. If all of your waiting threads could in theory take action with the new information then use `notifyAll()`. However if there is a new lock to be acquired (so only one waiting Thread can take action), then call just `notify()`.

Example:

```
1 public class Example {
2
3     public static void main(String[] args) {
4
5         ResourceCarrier carrier = new ResourceCarrier();
6         ThingNeedingResource thingNeedingResource
7         = new ThingNeedingResource(carrier);
8         ThingNeedingResource thingNeedingResource2
9         = new ThingNeedingResource(carrier);
10        ThingNeedingResource thingNeedingResource3
11        = new ThingNeedingResource(carrier);
12        ResourceCreator resourceCreator = new ResourceCreator(carrier);
13
14        new Thread(thingNeedingResource).start();
15        new Thread(thingNeedingResource2).start();
16        new Thread(thingNeedingResource3).start();
17        new Thread(resourceCreator).start();
18    }
19
20 }public class ResourceCarrier {
```

```
21     private boolean resourceReady;
22
23
24
25     public boolean isResourceReady() {
26         return resourceReady;
27     }
28
29     public void resourceIsReady() {
30         resourceReady = true;
31     }
32 }
33 }
34 public class ResourceCreator implements Runnable {
35     private ResourceCarrier carrier;
36
37     public ResourceCreator(ResourceCarrier carrier) {
38
39         this.carrier = carrier;
40     }
41
42     @Override
43     public void run() {
44         try {
45             Thread.sleep(2000);
46         } catch (InterruptedException e) {
47             e.printStackTrace();
48         }
49         synchronized (carrier) {
50             carrier.resourceIsReady();
51             carrier.notifyAll();
52         }
53     }
54 }
55 package com.core.interview.thread;
56
57
58 public class ThingNeedingResource implements Runnable {
59
60     private ResourceCarrier carrier;
61
62     public ThingNeedingResource(ResourceCarrier carrier){
```

```
63
64     this.carrier = carrier;
65 }
66 @Override
67 public void run() {
68     synchronized (carrier){
69         while(!carrier.isResourceReady()){
70             try {
71                 System.out.println("Waiting for Resource");
72                 carrier.wait();
73             } catch (InterruptedException e) {
74                 e.printStackTrace();
75             }
76         }
77         System.out.println("haz resource");
78     }
79 }
80 }
81
82 Sample output:
83 Waiting for Resource
84 Waiting for Resource
85 Waiting for Resource
86 haz resource
87 haz resource
88 haz resource
```

In this example we have a wrapper around a resource called “Resource Carrier” and 3 threads that want access to its resource. When it acquires the lock it sees the resource isn’t available and goes into wait mode by calling `wait()` on the `ResourceCarrier` object. The `ResourceCreator` swoops in later to create the resource and calls `notify` on the `ResourceCarrier`, at which point the three threads spring back to life.

Deadlock

Q: What is a deadlock?

Q: How do we prevent deadlocks?

Many candidates completely fluff the answer to deadlock questions. It’s a very common interview question, and it’s an easy enough concept to understand, but it can be tricky to explain (particularly over the phone).

A deadlock occurs when two or more threads are awaiting the actions of each other which prevents any further processing from happening. For example, Thread A has the lock on Object 1 and attempts to require a lock on Object 2 to continue. Simultaneously, Thread B has a lock on Object 2 and attempts to acquire a lock on Object 1. As a result both Threads are kept busy waiting for the other one, which will never release. This is a deadlock.

The easiest way to prevent deadlock is to ensure that the ordering of acquisition for locks is consistent. If both Threads try to acquire Object 1 then Object 2 in that order then deadlock will not occur. It is important to code defensively when acquiring locks. Even better is to make synchronized blocks as small as possible and where possible avoid locking multiple objects.

Q: What is Livelock?

In deadlock, the Threads will stop working and await the resource to become free. In Livelock the separate Threads do not stop; they will continue working trying to resolve the block. However, by doing so the blockage continues and will not resolve. This can happen in situations where, if a Thread cannot access the resources it requires it will go back to the start of processing and retry the work in a loop until success. Imagine the classic bank account example. If there is a simultaneous transaction from Account A to Account B and Account B to Account A then both accounts are debited, but when the crediting of the account takes place the account is already locked. The transactions are reverted and reattempted. This will continue ad infinitum with the work being attempted; hence, “live” lock.

Q: What is Thread Priority?

Thread has a method on it, `setPriority()` which takes an int representing the priority level. Thread has some constants on it you can use: `Thread.MIN_PRIORITY`, `Thread.MAX_PRIORITY` and `Thread.NORM_PRIORITY`.

However, like all things threading and JVM related, you cannot rely on it to behave in a consistent fashion particularly across different systems and operating systems. The actual way priority levels vary from system to system, and the CPU can choose to behave in any way it wants. Maybe for priorities above a certain level it will give a dedicated percentage of time. In the same fashion, you don't know what other processes are being run. You may have a background thread you set to a low value, but if there is some other process running on the machine at a slightly higher priority your Thread may never get run.

Nonetheless, it can work well. Check the following example:

```
1  public static void main(String[] args) {
2      Thread thread = new Thread(new Counter(1));
3      thread.setPriority(Thread.MAX_PRIORITY);
4      Thread threadTwo = new Thread(new Counter(2));
5      threadTwo.setPriority(Thread.MIN_PRIORITY);
6      thread.start();
7      threadTwo.start();
8  }
9
10 private static class Counter implements Runnable {
11     private int id;
12
13     public Counter(int i) {
14         id = i;
15     }
16
17     @Override
18     public void run() {
19         for (int i = 0; i < 10; i++) {
20             System.out.println("Thread " + id + " - " + i);
21         }
22     }
23 }
```

This simple program launches two Threads which count to ten in a loop. If I remove the priority setting lines, the output is jumbled:

```
1  Thread 2 - 0
2  Thread 1 - 0
3  Thread 1 - 1
4  Thread 1 - 2
5  Thread 1 - 3
6  Thread 1 - 4
7  Thread 2 - 1
8  Thread 2 - 2
9  Thread 2 - 3
10 Thread 2 - 4
11 Thread 2 - 5
12 Thread 2 - 6
13 Thread 1 - 5
14 Thread 1 - 6
15 Thread 2 - 7
```

```
16 Thread 2 - 8
17 Thread 1 - 7
18 Thread 1 - 8
19 Thread 1 - 9
20 Thread 2 - 9
```

However, if I run the program as it's shown above:

```
1 Thread 1 - 0
2 Thread 1 - 1
3 Thread 1 - 2
4 Thread 1 - 3
5 Thread 1 - 4
6 Thread 1 - 5
7 Thread 1 - 6
8 Thread 1 - 7
9 Thread 1 - 8
10 Thread 1 - 9
11 Thread 2 - 0
12 Thread 2 - 1
13 Thread 2 - 2
14 Thread 2 - 3
15 Thread 2 - 4
16 Thread 2 - 5
17 Thread 2 - 6
18 Thread 2 - 7
19 Thread 2 - 8
20 Thread 2 - 9
```

Running on my machine this isn't consistent though, and does sometime jumble slightly. The way Threads work will never be consistent across machines or JVMs and it is important to write code that does not rely on a JVM's behaviour.

Q: What is Thread starvation?

If a Thread cannot get CPU time because of other Threads this is known as starvation. Imagine an extreme example like the above, with a number of high priority Threads and a single low priority Thread. The low priority is unlikely to get any CPU time and will be suffering from starvation.

Futures, callables and executors

Q: What is a ThreadPool? Why is it better to use them instead of manually creating Threads?

The creation and maintenance of Threads is expensive and time consuming. With a ThreadPool a group of Threads are available which can be called on to execute tasks. The Threads are created up-front, thus reducing the overhead at execution time. Once a task has been executed the Thread is returned to the pool to execute another task. If a Thread dies for some reason it will be replaced in the ThreadPool, removing the need to write complex Thread management code.

There is the added benefit that it makes it easy to control the number of tasks happening in parallel. For example, if we have a webserver we may want to limit the number of parallel tasks so that a huge burst of traffic wouldn't stop other background tasks from happening.

Q: How do you create a ThreadPool?

Normally ThreadPools are created using Executors.

```
1 ExecutorService executorService1 = Executors.newSingleThreadExecutor();
2 ExecutorService executorService2 = Executors.newFixedThreadPool(10);
3 ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
```

A single thread executor will create a single thread; all tasks will be executed sequentially. The same could be achieved using `Executors.newFixedThreadPool(1)`.

`Executors.newFixedThreadPool` will create a ThreadPool with exactly the number of Threads specified. Those threads will live for the lifetime of the JVM unless the Executor is shutdown using `shutdown()` (which will allow currently queued tasks to finish) and `shutdownNow()` (which will cancel any outstanding tasks).

Tasks can be run on `ExecutorServices` by either submitting a `Callable` or a `Runnable`.

Q: What is the difference between Callable and Runnable? What is a Future?

A `Runnable` cannot return a value and it cannot throw a checked exception, both features available in `Callable`. Often a return value is very useful when building multithreaded applications. When we submit a task to an executor we don't know when it will be executed; it could be instantly or at any time in the future- the whole process is asynchronous. Futures allow this to happen. Futures have one main method: `get()` and it's overloaded version, `get(long timeout, TimeUnit unit)`. When called, it will block the current Thread whilst it awaits the result (if the result is not currently available).

There is no benefit/negative to using `Callable` over `Runnable` or vice versa; it all depends on whether a return value is needed.


```
1 public static void main(String[] args) throws InterruptedException, ExecutionExc\
2 eption {
3     Callable<String> callable = new StringCallable();
4     Callable<String> callable2 = new StringCallable();
5     Callable<String> callable3 = new StringCallable();
6     Callable<String> callable4 = new StringCallable();
7
8     ExecutorService executorService = Executors.newFixedThreadPool(4);
9     List<Future<String>> futures = executorService.invokeAll(asList(callable, ca\
10 llable2, callable3, callable4));
11     for (Future<String> future : futures) {
12         System.out.println(future.get());
13     }
14 }
15 //pool-1-thread-1
16 //pool-1-thread-2
17 //pool-1-thread-3
18 //pool-1-thread-4
```

Q: What is ThreadLocal?

ThreadLocal allows a different instance of a variable for each Thread that accesses it. Normally this will be a static variable used to store state like a UserID or session. This can also be particularly useful when an object is not Thread Safe but you want to avoid synchronising.

Big O Notation

I hate big O notation. For as long as I can remember it's been my biggest achilles heel. It's just something I've never managed to successfully motivate myself to learn about despite knowing it's going to come up in every single interview. I'll get asked to implement an algorithm which I'll do via brute force to start with: "What is the Big O of that?". "I don't know Big O but I know it's slow".

It's fairly obvious, but this is the wrong approach. Take the time to really read this chapter and maybe even do some extra research. This seems to be a question that comes up consistently at interviews so it's worth taking the time.

What on earth is Big O?

Big O is the way of measuring the efficiency of an algorithm and how well it scales based on the size of the dataset. Imagine you have a list of 10 objects, and you want to sort them in order. There's a whole bunch of algorithms you can use to make that happen, but not all algorithms are built equal. Some are quicker than others but more importantly the speed of an algorithm can vary depending on how many items it's dealing with. Big O is a way of measuring how an algorithm scales. Big O references how **complex** an algorithm is.

Big O is represented using something like $O(n)$. The O simply denoted we're talking about big O and you can ignore it (at least for the purpose of the interview). n is the thing the complexity is in relation to; for programming interview questions this is almost always the size of a collection. The complexity will increase or decrease in accordance with the size of the data store.

Below is a list of the Big O complexities in order of how well they scale relative to the dataset.

$O(1)$ /Constant Complexity: Constant. This means irrelevant of the size of the data set the algorithm will always take a constant time.

1 item takes 1 second, 10 items takes 1 second, 100 items takes 1 second.

It always takes the same amount of time.

$O(\log n)$ /Logarithmic Complexity: Not as good as constant, but still pretty good. The time taken increases with the size of the data set, but not proportionately so. This means the algorithm takes longer per item on smaller datasets relative to larger ones.

1 item takes 1 second, 10 items takes 2 seconds, 100 items takes 3 seconds.

If your dataset has 10 items, each item causes 0.2 seconds latency. If your dataset has 100, it only takes 0.03 seconds extra per item. This makes log n algorithms very scalable.

$O(n)$ /Linear Complexity: The larger the data set, the time taken grows proportionately.

1 item takes 1 second, 10 items takes 10 seconds, 100 items takes 100 seconds.

$O(n \log n)$: A nice combination of the previous two. Normally there's 2 parts to the sort, the first loop is $O(n)$, the second is $O(\log n)$, combining to form $O(n \log n)$.

1 item takes 2 seconds, 10 items takes 12 seconds, 100 items takes 103 seconds.

$O(n^2)$ /Quadratic Complexity: Things are getting extra slow.

1 item takes 1 second, 10 items takes 100, 100 items takes 10000.

$O(2^n)$: Exponential Growth! The algorithm takes twice as long for every new element added.

1 item takes 1 second, 10 items takes 1024 seconds, 100 items takes 1267650600228229401496703205376 seconds.

It is important to notice that the above is not ordered by the best to worst complexity. There is no "best" algorithm, as it completely hinges on the size of the dataset and the task at hand. It is also important to remember the code maintenance cost; a more complex algorithm may result in an incredibly quick sort, but if the code has become unmaintainable and difficult to debug is that the right thing to do? It depends on your requirements.

There is also a variation in complexity within the above complexities. Imagine an algorithm which loops through a list exactly two times. This would be $O(2n)$ complexity, as it's going through your lists length (n) twice!

Why does this matter?

Simply put: *an algorithm that works on a small dataset is not guaranteed to work well on a large dataset.* Just because something is lightning fast on your machine doesn't mean that it's going to work when you scale up to a serious dataset. You need to understand exactly what your algorithm is doing, and what its big O complexity is, in order to choose the right solution.

There are three things we care about with algorithms: **best case**, **worst case** and **expected case**. In reality we only actually care about the latter two, as we're a bunch of pessimists. If you ask an algorithm to sort a pre-sorted list it's probably going to do it much faster than a completely jumbled list. Instead we want to know the worst case (the absolutely maximum amount of steps the algorithm could take) and the expected case (the likely or average number of steps the algorithm could take). Just to add to the fun, these can and often are different.

Examples

Hopefully you're with me so far, but let's dive into some example algorithms for sorting and searching. The important thing is to be able to explain what complexity an algorithm is. Interviewers love to get candidates to design algorithms and then ask what the complexity of it is.

O(1)

Irrelevant of the size, it will always return at constant speed. The javadoc for Queue states that it is “*constant time for the retrieval methods (peek, element, and size)*”. It’s pretty clear why this is the case. For peek, we are always returning the first element which we always have a reference to; it doesn’t matter how many elements follow it. The size of the list is updated upon element addition/removal, and referencing this number is just one operation to access no matter what the size of the list is.

O(log n)

The classic example is a Binary search. You’re a massive geek so you’ve obviously alphabetised your movie collection. To find your copy of “Back To The Future”, you first go to the middle of the list. You discover the middle film is “Meet The Fockers”, so you then head to the movie in between the start and this film. You discover this is “Children of Men”. You repeat this again and you’ve found “Back to the Future”. There’s a great interactive demo of binary search available online at [Armstrong State University](http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html)³.

Although adding more elements will increase the amount of time it takes to search, it doesn’t do so proportionally. Therefore it is O(log n).

O(n)

As discussed in the collections chapter, LinkedLists are not so good (relatively speaking) when it comes to retrieval. It actually has a complexity of O(n) for the **worst case**: to find an element T, which is the last element in the list, it is necessary to navigate the entire list of n elements. As the number of elements increases so does the access time in proportion.

O(n log n)

The best example of O(n log n) is a **merge sort**. This is a divide and conquer algorithm. Imagine you have a list of integers. We divide the list in two again and again until we are left with with a number of lists with 1 item in: each of these lists is therefore sorted. We then merge each list with it’s neighbour (comparing the first elements of each every time). We repeat this with the new composite list until we have our sorted result. To explain why this is O(n log n) is a bit more complex. In the above example of 8 numbers, we have 3 levels of sorting:

- 4 list sorts when the list sizes are 2
- 2 list sorts when the list sizes are 4
- 1 list sort when the list size is 8

³<http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html>

Now consider if I were to double the number of elements to 16: this would only require one more level of sorting. Hopefully you recognise this is a $\log n$ scale.

However, on each level of sorting a total of n operations takes place (look at the red boxes in the diagram above). This results in $(n * \log n)$ operations, e.g. $O(n \log n)$.

$O(n^2)$

The Bubble Sort algorithm is everyone's first algorithm in school, and interestingly it is quadratic complexity. If you need a reminder; we go through the list and compare each element with the one next to it, swapping the elements if they are out of order. At the end of the first iteration, we then start again from the beginning, with the caveat that we now know the last element is correct.

Imagine writing the code for this; it's two loops of n iterations.

```
1 public int[] sort(int[] toSort){
2     for (int i = 0; i < toSort.length - 1; i++) {
3         boolean swapped = false;
4         for (int j = 0; j < toSort.length - 1 - i; j++) {
5             if(toSort[j] > toSort[j+1]){
6                 swapped = true;
7                 int swap = toSort[j+1];
8                 toSort[j + 1] = toSort[j];
9                 toSort[j] = swap;
10            }
11        }
12        if(!swapped)
13            break;
14    }
15    return toSort;
16 }
```

This is also a good example of best vs worst case. If the list to sort is already sorted, then it will only take one iteration (e.g. n) to sort. However, in the worst case we have to go through the list n times and each time looping another n items (less how many loops we have done before) which is slow.

You may notice that it's technically less than n^2 as the second loop decreases each time. This gets ignored because as the size of the data set increases this impact of this becomes more and more marginal and tends towards quadratic.

$O(2^n)$

Exponential growth! Any algorithm where adding another element dramatically increases the processing time. Take for example trying to find combinations; if I have a list of 150 people and

I would like to find every combination of groupings; everyone by themselves, all of the groups of 2 people, all of the groups of 3 people etc. Using a simple program which takes each person and loops through the combinations, if I add one extra person then it's going to increase the processing time exponentially. Every new element will double processing time.

In reality $O(2^n)$ algorithms are not scalable.

How to figure out Big O in an interview

This is not an exhaustive list of Big O. Much as you can $O(n^2)$, you can also have $O(n^3)$ (imagine bubble sort but with an extra loop). What the list on this page should allow you to do is have a stab in the dark at figuring out what the big O of an algorithm is. If someone is asking you this during an interview they probably want to see how you try and figure it out. Break down the loops and processing.

- Does it have to go through the entire list? There will be an n in there somewhere.
- Does the algorithm's processing time increase at a slower rate than the size of the data set? Then there's probably a $\log n$ in there.
- Are there multiple loops? You're probably looking at n^2 or n^3 .
- Is access time constant irrelevant of the size of the dataset? $O(1)$

Sample question

I have an array of the numbers 1 to 100 in a random number. One of the numbers is missing. Write an algorithm to figure out what the number is and what position is missing.

There are many variations of this question all of which are very popular. To calculate the missing number we can add up all the numbers we do have, and subtract this from the expected answer of the sum of all numbers between 1 and 100. To do this we have to iterate the list once. Whilst doing this we can also note which spot has the gap.

```
1 public class BlankFinder{
2
3 public void findTheBlank(int[] theNumbers) {
4     int sumOfAllNumbers = 0;
5     int sumOfNumbersPresent = 0;
6     int blankSpace = 0;
7
8     for (int i = 0; i < theNumbers.length; i++) {
9         sumOfAllNumbers += i + 1;
10        sumOfNumbersPresent += theNumbers[i];
```

```
11         if (theNumbers[i] == 0)
12             blankSpace = i;
13     }
14
15     System.out.println("Missing number = " + (sumOfAllNumbers -
    sumOfNumbersPresent) + " at location " + blankSpace + " of the array"); }
```

```
1     public static void main(String[] args) {
2         new BlankFinder().findTheBlank(new int[]{7,6,0,1,3,2,4});
3     }
4     //Missing number = 5 at location 2 of the array
5 }
```

*Caveat: you can also calculate sumOfAllNumbers using $(\text{theNumbers.length}+1) * (\text{theNumbers.length}) / 2.0$. I would never remember that in an interview though.*

What is the big O of your algo?

Our algorithm iterates through our list once, so it's $O(n)$.

Conclusion

You're amazing. Well done for reading this book all the way through. I'm very proud of it, but more specifically I'm proud of how I've been able to help people to get amazing new jobs. Whenever I get emails or messages through from people saying thanks because I've helped them to get a better job it gives me the biggest buzz.

If everything goes well and you've benefited from this, then please do let me know. Or even if you have extra questions, want someone to look over your CV, or just want to say hi. I try and reply to every message I get. I'm also available for one on one coaching and I'd love to help you, one on one, through your interview process. Drop me an email to discuss further.

e: hello@corejavainterviewquestions.com

t: @sambahk

Go out and ace that interview.

Sam