I. Data Structure

Difference between Big O and Big Omega notations

Big O: it measures the efficiency of an algorithm based on the time it takes for the algorithm to run as a function of the input size, describe the *worst* case running time for an algorithm.
Big Omega: describe the best case running time for a given algorithm.
Big O is used to represent the upper bound running time for an algorithm, which is also the "worst case". Big Omega is used to represent the lower bound, which is also the "best case" for that algorithm.

O(n) - which is also known as linear time. Linear time means that the time taken to run the algorithm increases in direct proportion to the number of input items.

What's the difference between DFS and BFS?

DFS (Depth First Search) and BFS (Breadth First Search) are search algorithms used for graphs and trees.
BFS(breadth first search): start at the root node, and then scan each node in the first level starting from the leftmost node, moving towards the right. Then you continue scanning the second level (starting from the left) and the third level, and so on until you've scanned all the nodes, or until you find the actual node that you were searching for.
When traversing one level, we need some way of knowing which nodes to traverse once we get to the next level. The way this is done is by storing the pointers to a level's child nodes while searching that level. The pointers are stored in FIFO (First-In-First-Out) queue. This, in turn, means that BFS uses a large amount of memory because we have to store the pointers.
DFS(depth first search): start at the root, and follow one of the branches of the tree as far as possible until either the node you are looking for is found or you hit a leaf node (a node with no children). If you hit a leaf node, then you continue the search at the nearest ancestor with unexplored children.
The big advantage of DFS is that it has much lower memory requirements than BFS, because it's not necessary to store all of the child pointers at each level.

What are the differences between a hash table and a BST?

Hash table: insert and retrieve elements in O(1). An unordered data structure – which means that it does not keep its elements in any particular order(you would need additional memory to sort the values).
Binary search tree: insert and retrieve elements in O(log(n)), which is already sorted, there will be no need to waste memory or processing time sorting records.

What's the difference between a stack and a heap?

They are both stored in the computer's **RAM (Random Access Memory)**.

In a *multi-threaded* application, *each thread* will have its *own stack*. But, all the different threads will *share the heap*. Which means that there has to be some coordination between the threads so that they don't try to access and manipulate the same piece(s) of memory in the heap at the same time.
An object can be stored on the stack. If you create an object inside a function *without using the "new" operator* then this will create and store the object on the stack, and not on the heap. Once a function call runs to completion, any **data on the stack** created specifically for that function call will automatically be deleted. Any **data on the heap** will remain there until it's manually deleted by the programmer (Java use garbage collection to automatically delete memory from the heap).
The **stack** is set to a *fixed size* (cannot grow). If there is not enough room on the stack to handle the memory being assigned to it, a stack overflow occurs. This often happens when a lot of nested functions are being called, or if there is an infinite recursive call. If the current size of the heap is too small to accommodate new memory, then *more memory can be added to the* **heap** by the operating system. The implementation for a stack and a heap really depends on the language, compiler, and run time. *The stack is much faster than the heap*. This is because of the way that memory is allocated on the stack. Allocating memory on the stack is as simple as moving the stack pointer up.
If the **stack** runs out of memory, then this is called a **stack overflow** (cause the program to crash). The **heap** could have the problem of **fragmentation**, which occurs when the available memory on the heap is being stored as noncontiguous/disconnected blocks (used blocks of memory are in between the unused memory blocks). Thus, allocating new memory may be impossible because of the fact that even though there is enough memory for the desired allocation, there may not be enough memory in one big block for the desired amount of memory.
The stack is small, you would want to use it when you know exactly how much memory you will need for your data, or if you know the size of your data is very small. It's better to use the heap when you know that you will need a lot of memory for your data, or you just are not sure how much memory you will need (e.g. dynamic array).

II. Recursion

What is recursion used for?

Any routine that calls itself is a **recursive routine**. It is best used for problems where a large task can be broken down into a repetitive "sub-task". Because a recursive routine calls itself to perform those sub-tasks, eventually the routine will come across a sub-task that it can handle without calling itself. This is known as a -**base case** *(the base case stops the recursion)*. In the base case, the routine does not call itself. But, when a routine does have to call itself in order to complete its sub-task, then that is known as the **recursive case**.
A **call stack** is a data structure used by the program to store information about the active subroutines in a program. The main reason for having a call stack is so that the program can keep track of where a subroutine should return control to once it finishes executing. A **stack**

frame is a *part of the call stack*, and a new stack frame is created every time a subroutine is called. The stack frame is used to store all of the variables for one invocation of a routine, it also stores the return address. A call stack is basically a stack of stack frames.
Recursive calls will be made continuously if there's no base case exists, and each time a recursive call is made a new stack frame is created. Every new stack frame created needs more memory, which then means that there is less memory on the call stack. The call stack has limited memory, which is usually determined at the start of the program – and when that limited memory is exceeded then the **stack is said to overflow**/program crashing.

Can recursive function be converted into iterative function?

Any problem that can be solved recursively can also be solved through the use of iteration. **Iteration** is the use of a looping construct in order to solve a problem, whereas **recursion** is the use of a function that calls itself to solve a problem. The reason that iteration can be used to solve any problem that recursion solves is because recursion uses the call stack behind the scenes, but a call stack can also be implemented explicitly and iteratively.

Differences between using recursion versus using iteration

Recursion is rarely the most efficient approach to solving a problem, and *iteration is almost always more efficient*. This is because there is usually *more overhead* associated with making recursive calls due to the fact that the call stack is so heavily used during recursion. Many computer programming languages will spend more time maintaining the call stack then they will actually performing the necessary calculations. Recursion uses more memory than iteration because of the extensive use of the call stack.

Differences between tail recursion and normal recursion

A function is **tail recursive** if the final result of the recursive call is also the final result of the function itself, and that is why it's called "tail" recursion, because the final function call (the tail-end call) actually holds the final result. Comparing that with normal recursion, the normal recursive call is certainly not in it's final state in the last function call, because all of the recursive calls leading up to the last function call must also return in order to actually come up with the final answer.
The **tail recursive** *does not actually need a new stack frame for each and every recursive* call. This is because the calculation is made within the function parameters/arguments, and the final function call actually contains the final result, and the *final result does not rely on the return value of each and every recursive call.* It is up to the *compiler/interpreter of the particular language* to determine whether or not the recursive calls in a tail recursive function actually use an extra stack frame for each recursive call to the function.
Tail recursion can be *as efficient as iteration* if the compiler uses what is known as tail recursion optimization (no risk of having the stack overflow). If that optimization is not used, then tail recursion is just as efficient as normal recursion.

Tail recursion optimization is *not actually supported by Java* because the Java standard does not require that tail recursion be supported, although there are some JVM implementations that do support it as an add-on. But, **tail recursion** itself is *supported in Java* because it is just a special case of normal recursion.

III. JAVA

How does System.out.println() work?

Dot operator: can only be used to call methods and variables
Out: either a method or a variable. Could not possibly be a method because of the fact that there are no parentheses -'()' – after it, must be a static variable since only static variables can be called with just the class name itself. 'out' is a *static member* variable belonging to the **System** class.
Plintln(): a method.
The more exact answer is this: inside the System class is the declaration of 'out' that looks like: 'public static final PrintStream out', and inside the Prinstream class is a declaration of 'println()' that has a method signature that looks like: 'public void println()'.

Is JVM (Java Virtual Machine) platform dependent or independent?

Every Java program is first compiled into an intermediate language called Ja**va bytecode**.
The **JVM** is used primarily for 2 things: the *first* is to translate the bytecode into the machine language for a particular computer, and the *second* thing is to actually execute the corresponding machine-language instructions as well.
The JVM and bytecode combined give Java its status as a "portable" language – this is because Java bytecode can be transferred from one machine to another.
The JVM must translate the bytecode into machine language, since the machine language depends on the operating system being used, it is clear that the JVM is platform (operating system) dependent.

Difference between method overloading and overriding?

Method overloading in Java occurs when two or more methods in the same class have the exact same name but different parameters. (a compile time phenomenon)
The *conditions for method **overloaded***(if one or both is true): the number of parameters is different for the methods; the parameter types are different.
NOT overload methods: just changing the return type of the method; changing just the name of the method parameters, but not changing the parameter types.
Overriding methods: If a derived class requires a different definition for an inherited method, then that method can be redefined in the derived class. An overridden method would have the *exact same method name, return type, number of parameters, and types of parameters as the*

method in the parent class, and the only difference would be the definition of the method. (a run time phenomenon that is the driving force behind polymorphism)

What’s the point of having a private constructor?

Defining a constructor with the private modifier says that only the native class (as in the class in which the private constructor is defined) is allowed to create an instance of the class, and no other caller is permitted to do so.
Two possible reasons why one would want to use a **private constructor** – the first is that you don’t want any objects of your class to be created at all, and the second is that you only want objects to be created **internally** – as in only created in your class. (A **singleton** is a design pattern that allows only one instance of your class to be created, and this can be accomplished by using a private constructor).
The other possible reason for using a private constructor is to prevent object construction entirely. This occurs when the class only contains static members. And when a class contains only static members, those members can be accessed using only the class name – no instance of the class needs to be created.

What’s the difference between an object and a class?

Class refers to the actual written piece of code, which is used to define the behavior of any given class. A class is a static piece of code that consists of attributes which don’t change during the execution of a program.
Object is an instance of a class, refers to an actual **instance** of a class. Every object must belong to a class. Objects are created and eventually destroyed – so they only live in the program for a limited time. While objects are ‘living’ their properties may also be changed significantly. Every object has a lifespan associated with it - it can not live forever. And, the properties of those objects can change as well while they ‘live’.
A class is a general concept; an object is a very specific embodiment of that class, with a limited lifespan.

How copy constructors work?

What does the ‘Final’ modifier mean?

When applied to a *method* definition the **final modifier** indicates that *the method may not be overridden* in a derived class.
When applied to a *class*, the **final modifier** indicates that *the class cannot be used as a base class* to derive any class from it. The final modifier, when applied to either a class or a method, turns off late binding, and thus prevents polymorphism.
When applied to an *instance variable*, the **final modifier** simply means that the *instance variable can’t be changed*.

What does the finally block do?

The **finally block**, if used, is placed after a *try block* and the *catch blocks* that follow it. The finally block contains code that will be run whether or not an exception is thrown in a try block. *Four potential scenarios*:

Try block runs to the end and no exception is thrown. The finally block will be executed after the try block.

An exception is thrown in the try block, which is then caught in one of the catch blocks. The finally block will execute right after the catch block.

An exception is thrown in the try block and no matching catch block can catch the exception. The exception object is thrown to the enclosing method, the finally block executes before the method ends;

The method returns wherever was invoked before the try block runs to completion, the finally block is executed before it returns to the invoking method. Finally block will take precedence over the return statement. It *will not be called* after return when *System.exit()* is called first, or the *JVM crashes*. If finally bloxk also has return, Anything that is returned in the finally block will actually override any exception or returned value that is inside the try/catch block(bad idea if finally block has return statement).

Access modifier (private, public, protected)

When an **access modifier** is *omitted* from a variable, it means that the variable *has default, or package access.* When a variable has package access, it means that any other class defined in the same package will have access to that variable by name. And, any class not defined in the same package will not be able to access it by name.
Any method or instance variable that is declared as **protected** can be accessed by name in its own class definition, in any class derived from it, and by any class that's in the same package.

What does it mean if a class is serializable?

If a class is **serializable**, it means that any object of that class can be converted into a sequence of bits so that it can be written to some storage medium (like a file), or even transmitted across a network.
(public class SomeClass implements java.io.Serializable)

How to implement multiple inheritance?

Multiple inheritance is the ability of a single class to inherit from multiple classes. *Java does not have* this capability(or will cause diamond problem). What a Java class does have is the ability to *implement multiple interfaces* – which is considered a reasonable substitute for multiple inheritance, but without the added complexity. You can also have a class that extends *one* class, while implementing an interface – or even multiple interfaces.

Two different methods of creating a thread in Java

Difference between an interface and an abstract class

An **abstract class** has one or more **abstract methods**, which only had a method heading, but no body (no implementation code inside curly braces like normal methods do).
The abstract class cannot give the abstract method specific code. Any classes that derive from the base class basically has 2 options: 1. The derived class must *provide a definition for the abstract method* OR 2. The derived class must be *declared abstract itself* (Any **non abstract class** is called a *concrete class*).
An **interface** differs from an abstract class because an interface is *not a class*. An interface is essentially *a type* that can be satisfied by any class that implements the interface. Any class that implements an interface must satisfy 2 conditions: 1. It must have the phase “implements Interface_Name” at the beginning of the class definition. 2. It must implement all of the method headings listed in the interface definition.
3 major difference between abstract class and interface:

1. **Abstract classes** are meant to be *inherited* from, and when one class inherits from another, it means that there is a strong relationship between those classes. An **interface** on the other hand, the relationship between the interface itself and the class implementing the interface is not necessarily strong. -> An abstract class would be more appropriate when there is a strong relationship between the abstract class and the classes that will derive from it. But with interfaces there need not be a strong relationship between the interface and the classes that implement the interface.
2. A class can only derive from one class, whether it’s abstract or not. However, a class can implement multiple interfaces – which could be considered as an alternative to for multiple inheritance. -> A Java class can inherit from only one abstract class, but can implement multiple interfaces.
3. An abstract class may provide some methods with definitions – so an abstract class can have non-abstract methods with actual implementation details. An abstract class can also have constructors and instance variables as well. An interface, however, can not provide any method definitions – it can only provide method headings. Any class that implements the interface is responsible for providing the method definition/implementation.

When to use abstract class and interface
Abstract class: plan on using inheritance since it provides a common base class implementation to derived classes; declare non-public members (all methods must be public in an interface); need to add methods in the future.
Interface: the API(application programming interface) will not change for a while; want to have something similar to multiple inheritance.

Dynamic binding

Dynamic binding basically means that the method implementation that is actually called is determined at *run-time*, and not at compile-time. And that's why it's called dynamic binding – because the method that will be run is chosen at run time. Dynamic binding is also known as late binding.(e.g. class C extends B, B extends A, implement someMethod(), if A x = new B(), will run method in B instead of A)

Can constructors be synchronized in Java?

Does Java pass by reference or by value?

Java **passes everything** (objects, arrays (which are objects in Java), primitive types (like ints and floats), etc.) **by value**, and not by reference.
The differences between pass by value and pass by reference: When passing an argument (or even multiple arguments) to a method, Java will *create a copy or copies of the values* inside the original variable(s) and pass that to the method as arguments. The key with pass by value is that the method will not receive the actual variable that is being passed – but just a copy of the value being stored inside the variable.
A variable of a class type stores an address of the object, and not the object itself. The object itself is stored at the address which the variable points to (a variable of a class type – or what is commonly referred to as an object of a class – really just stores a memory address that points to the real object).

Difference between a primitive type and a class type

Every variable in Java has a type, which essentially tells Java how that variable should be treated, and how much memory should be allocated for that variable.
Java has basic types for characters, different kinds of integers, and different kinds of floating point numbers (numbers with a decimal point), and also types for the values true and false – char, int, float, and bool. All of these basic types are known as **primitive types.**
Java classes are created to *solve object oriented problems*. Any object of a class has the type "class". Because an object of a class is more complex than "primitive" type, a variable naming an object is known to be a **class type**.
Every variable, whether it's of a primitive type or of a class type, is *implemented as a location* in computer memory. For a variable of a *primitive type*, the value of the variable is stored in the memory location assigned to the variable. However, a variable of a *class type* only stores the memory address of where the object is located – not the values inside the object. The object named by the variable is stored in some other location in memory and the variable contains only the memory address of where the object is stored. This memory address is called a reference to the object.
A value of a primitive type will always require the same amount of memory to store one value (value has a limited size since primitive type has boundaries). An object of a class type can be of

any size. But there is a limit to the size of an address. Because variables of a class type contain a reference (memory address), two different variables can hold the same reference, and in that situation both variables name the same object. Any change to the object named by one of these variables will produce a change to the object named by the other variable, since they both reference the same object. In Java, class types are also reference types. But, primitive types (like ints, booleans, etc.) are not reference types since primitive type variables do not hold references

Does Java have pointers？

Java does not have pointers. Because arrays and strings are provided as class types in Java, there is no need for pointers to those constructs. By not allowing pointers, Java provides effectively provides another level of abstraction to the programmer： Java has references, but not pointers.
Differences between references in Java and pointers in C++:
References store an address, which is the address in memory of the object. Pointers directly point to the object.
You cannot perform arithmetic operations on references.

Downcasting in java

There is a base class, and a class that derives from that base class either directly or indirectly. Then, anytime an object of that base class type is type cast into a derived class type, it is called a **downcast**. So, in downcasting, you are going down the inheritance diagram by taking an object of a base class (at the top), and then *trying* (downcasting does not always make sense depending on the code being written) to convert into the type of one of the derived classes (going down). There is no way that an object of the Parent class can ever be downcast to an object of it's child class (cause compile error).
Downcasing is allowed in Java and it is very useful when you need to compare one object to another. In Java, every descendant of a class X is also of type X.

What is the diamond problem?

Classes B and C are derived from class A, and class D derives from B and C (multiple inheritance). The **diamond problem** is that when instantiate an object of class D, any calls to method definitions in A will be ambiguous since it's not sure whether to call the version of the method derived from B or C. However, Java does not have multiple inheritance, thus *Java is not at risk of suffering the diamond problem*.

Can an interface extend another interface in Java?

An interface can extend another interface in Java. Any class that implements an interface *must* implement the method headings that are declared in that interface. And, if that interface extends from other interfaces, then the implementing class must also implement the methods

in the interfaces that are being extended or derived from.
Any time a class implements an interface, this means that when an object of that class is created, it will also have the interface type. An **interface is a type**, we can also *create a method with a parameter of an interface type*, and that parameter will accept as an argument any class that implements the interface.

Can an interface be instantiated in Java?

An interface cannot be instantiated in Java. However, because an interface is a type, you are allowed to write a method with a parameter of an interface type. And that method parameter will accept (as an argument) any class that implements the interface.

The difference between hashtable and array

Hashtables have a higher lookup overhead when compared to arrays. But the biggest difference is in *the memory* that each data structure would require. A **hash table** would only need to store the characters that actually exist in the input string. An **array** would need an element for every single possible value of a character because with an array you cannot skip indices (have to count for possible character that could be in the string, whether or not it is actually in the string). We simply don't know what is going to be in the string that's being passed in, with an array we have to be ready to accept all possible values.

The difference between equals() and ==

An **object** has both a *location in memory* and a *specific state* depending on the values that are inside the object.
The **"==" operator** compares the *objects' locations()* in memory. It is actually checking to see if the string objects (obj1 and obj2) refer to the exact same memory location. In other words, if both obj1 and obj2 are just different names for the same object then the "==" operator will return true when comparing the 2 objects.
The **equals() method** is defined in the Object class, from which every class is either a direct or indirect descendant. By default, the equals() method actually behaves the same as the "==" operator, it checks to see if both objects reference the same place in memory. But, the equals() method is actually meant to compare the contents of 2 objects, and not their location in memory. The equals() class is overridden to get the desired functionality whereby the object contents are compared instead of the object locations. This is the Java best practice for overriding the equals() method: you should compare the values inside the object to determine equality. The Java String class actually overrides the default equals() implementation in the Object class – and it overrides the method so that it checks only the values of the strings, not their locations in memory.

The number of trailing zeros

The number of trailing zeros means that we want to calculate how many zeros the factorial of

a number ends with – the number of terminating zeros.
For *any number*, it will have a trailing zero if it has a 10 as one of its factors.
In *factorial*, the number of 2′s as factors will always exceed the number of 5′s as factors. Thus, we just need to count the number of times 5 is a factor in the factorial to get the number of trailing zeros.(e.g. 29! Have 6 zeros since it has six 5, which are 25, 20, 15, 10, 5)

What is reflection in Java?

The **reflection** implies that the properties of whatever being examined are displayed or reflected to someone who wants to observe those properties. Similarly, Reflection in Java is the ability to examine and/or modify the properties or behavior of an object at run-time. It's important to note that reflection specifically applies to objects– so you need an object of a class to get information for that particular class.

Inner classes used in Java

Inner classes allow you to define one class inside another class, which is why they are called "inner" classes. *A class can have member classes*, just like how classes can have member variables and methods.

IV. Design Pattern

Singleton design pattern

The **singleton design pattern** is - applied to a given class - basically *limits* the class itself to having *just one instance* (a specific realization of any object)created.

V. Operating System

What's virtual memory?

Virtual memory doesn't physically exist on a memory chip. It is an optimization technique and is implemented by the operating system (Windows, Mac OS X, and Linux) in order to give an application program the impression that it has more memory than actually exists. E.g. an operating system needs 120 MB of memory in order to hold all the running programs, but there's currently only 50 MB of available physical memory stored on the RAM chips. The operating system will then set up 120 MB of virtual memory, and will use a program called the v**irtual memory manager (VMM)** to manage that 120 MB. The VMM will create a file (called **paging file**, which plays an important role in virtual memory) on the hard disk that is 70 MB in size to account for the extra memory that's needed. The O.S. will now proceed to address memory as if there were actually 120 MB of real memory stored on the RAM, even though there's really only 50 MB. So, to the O.S., it now appears as if the full 120 MB actually exists.

The **paging file** *combined* with the RAM accounts for all of the memory. Whenever the O.S. needs a 'block' of memory that's not in the real (RAM) memory, the VMM takes a block from the real memory that hasn't been used recently, writes it to the paging file, and then reads the block of memory that the O.S. needs from the paging file. The VMM then takes the block of memory from the paging file, and moves it into the real memory (in place of the old block). This process is called **swapping/paging,** and the *blocks of memory* that are swapped are called **pages**.
There are *two reasons* why one would want virtual memory: the first is to allow the use of programs that are too big to physically fit in memory. The other reason is to allow for multitasking – multiple programs running at once.
Virtual memory can *slow down performance*. If the size of virtual memory is quite large in comparison to the real memory, then more swapping to and from the hard disk will occur as a result. Accessing the hard disk is far slower than using system memory. Using too many programs at once in a system with an insufficient amount of RAM results in **constant disk swapping/thrashing**, which can really slow down a system's performance.