**Why Java?**

* Java is OOP => not 100% OOP, since Java has primitives
* Automatic Memory Management
* Platform Independent => "Write once, run anywhere"
* Open source
* Rich API => no need to reinvent the wheel
* Supported by Oracle => Languages like Ruby and Python just don't have the same support
* Simple (no pointers, no operator overloading, etc)

**What is necessary to write/execute a Java program?**

* Java Virtual Machine (JVM)
  + Abstract machine that provides the runtime environment in which Java code may be executed
* Java Runtime Environment (JRE)
  + Provides class and runtime libraries (such as java.lang, java.util, java.math, etc), the JVM, and other components necessary to run Java applications
  + The JVM is actually running the program, but it relies on the class files and supporting files provided by the JRE
  + The JRE is a superset of the JVM
* Java Development Kit (JDK)
  + contains the the tools necessary to develop Java programs, such as the compiler and the Java application launcher (which opens the JRE, loads the class, and invokes the main method)
  + The JDK is a superset of the JRE

**How do I run my Java program from the command line? Suppose we have just created MyFirstClass.java**

1. In a terminal, navigate to the directory where MyFirstClass.java was created
2. [Revature@Domain]$ javac MyFirstClass.java
   * This command compiles the .java file into a .class file (bytecode)
3. [Revature@Domain]$ java MyFirstClass
   * This command loads and executes the MyFirstClass.class file
   * Note: We do not include the .class extension when using the java command

**What is the difference between a Class and an Object?**

* A Class can be thought of as a blueprint that define state (fields, variables) and behavior (methods)
* An Object is an instance of the Class definition

**What are the primitive types in Java?**

1. boolean => size not specified per documentation
2. byte => 8-bit
3. short => 16-bit
4. char => 16-bit
5. int => 32-bit
6. long => 64-bit
7. float => 32-bit, floating point
8. double => 64-bit, floating point

**Where are Java Objects stored in memory?**

* Objects are stored in the heap
* Everything else is stored in the stack

**What all can a Java class conatin? (Class Members)**

* Constructors
* Behavior (methods)
* State (variables/fields)
* Blocks
* Inner Classes/Interfaces

**What are the scopes of a variable in Java?**

* Instance/Object
* Class/Static
* Method
* Block

**Is Java Pass-by-Value or Pass-by-Reference?**

* *STRICTLY* Pass-by-Value

**What are the Four Pillars of Object Oriented Programming?**

* An easy way to remember them is with the acronym *A PIE*
* Abstraction
  + Abstraction is a programming principle in which we centralize common characteristics and generalize behavior into conceptual classes
  + In Java, we achieve Abstraction through extending an Abstract Class or implementing an Interface
* Polymorphism
  + By definition, Polymorphism means "taking on many forms"
  + In Java, Polymorphism is achieved through
    - The subclassing of conceptual classes
    - Method Overloading
    - Method Overriding
    - Covariant Return types
* Inheritance
  + Inheritance is the act of subclassing or extending a parent class which in turn gives the child class all of the state and behavior of the parent class
* Encapsulation
  + Encapsulation is the process of hiding and preventing change of a class' data members
    - In Java, Encapsulation is achieved through declaring data members as private, with public getters and setters

**What is the difference between Method Overloading and Method Overriding?**

* Method Overloading is when there are two or more methods in a class with the same method name, but different method signatures
* Method Overriding is when a method in a child class has the same method signature as a method in the parent class, but with different implementation

**What is Covariant Return Types?**

* Covariant return types are an extension of Method Overriding
* A Covariant Return Type occurs when an Overridden Method in a child class' return type is a subclass of the parent method's return type
* class A {
* public Object print() {
* // Do stuff
* }
* }
* class B extends A {
* public String print() {
* // This is an example of a Covariant Return Type
* // print() is an overridden method, and String is a subclass of Object
* }

}

**Does Java support multiple inheritance? Does Java support multi-level inheritance?**

* **NO**, Java **DOES NOT** support multiple inheritance for classes.
  + However, there is a workaround as a class can implement multiple interfaces
* Yes, Java does support multi-level inheritance

**What is the difference between Abstract Classes and Interfaces?**

* Abstract Class
  + Cannot be instantiated
  + Instance variables do not have to be public, static, or final
  + Can have both abstract and concrete methods
  + Does not support multiple inheritance
* Interface
  + Cannot be instantiated
  + Variables are implicitly public, static, final
  + Methods are implicitly public, abstract
  + Since Java 8, interfaces can provide method implementations if the method is marked static or default
  + Supports multiple inheritance

**What is a method?**

* A method is a named block of code which is called by name and invoked at any point in a program by utilizing a method's name

**What is a constructor?**

* They return an instance of a class, **DOES NOT HAVE A RETURN TYPE**
* Must be named the same as the Class name
* If no constructor is provided explicitly, the JVM provides a No-Argument constructor by default
* Can be overloaded
* First line in any constructor is either this() or super()

**What does static mean?**

* Static implies that the variable or method belongs to the class, not individual instances

**Can the main method be overloaded?**

* Yes, but overloading the main method doesn't quite accomplish anything. Java sees it as just a static method called main
* **"THE"** main method is static and void, and accepts a String array as it's parameter

class Demo {

// Overloaded main method

// Is not the starting point of the program

public static void main(int a) {

System.out.println("You provided the int '" + a + "' as an argument");

}

// THE actual main method

// Is the starting point of the application. Can invoke the overloaded main as such

public static void main(String[] args) {

// Overloaded method

main(13); // You provided the int '13' as an argument

}

}

**Can the main method be overridden?**

* No. Static methods are never overridden, they are hidden
  + Method Hiding replaces the parent method in the calls defined in the child class

**What is the Order of Initialization in a Java class?**

1. Static Variables
2. Static Initializers
3. Instance Variables
4. Instance Initializers
5. Constructors

class Application {

static {

System.out.println("Inside the static initializer");

}

{

System.out.println("Inside the instance initializer");

}

public Application() {

System.out.println("Inside the constructor");

}

public static void main(String[] args) {

new Application();

// Will print out, in this order:

// Inside the static initializer

// Inside the instance initializer

// Inside the constructor

}

}

**What are the Access Modifiers in Java?**

* public => accessible anywhere
* protected => accessible in this package, and this class' subclass
* package-private => accessible only in this package
* private => accessible only in this class

**What is the difference between final, finally, and finalize?**

* final is a non-access modifier applied to
  + variables to ensure that their values cannot be reassigned
  + methods to ensure that they cannot be overridden
  + classes to ensure that the class cannot be extended
* finally is a keyword that is used at the end of a try/catch block
  + Code in the finally block will **ALWAYS** execute, unless there is a call to System.exit() or some catastrophic system error
* finalize is a method in the java.lang.Object class which is called by the Garbage Collector at most once
  + Overridden by your POJO with the intent to provide clean up logic, which will be called whenever the object is destroyed

**What is Garbage Collection?**

* Garbage Collection is the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program
* We can *request* Garbage Collection to be run through
  + System.gc()
  + Runtime.getRuntime().gc()
  + System.runFinalize()

**What is the String Pool?**

* Within the heap, the String Pool is a location where all String values are stored
  + This is important because String's take up a lot of memory. Being able to reuse the same value throughout your application is advantageous

**What are Wrapper Classes?**

* Wrapper classes are classes that let you treat primitives as Objects
  + *autoboxing* is the process of converting a primitive to its Wrapper class
  + *unboxing* is the process of converting a Wrapper class to its primitive

**What is an array?**

* An array is a group of sequentially stored elements of the same type
* With an array, the size is fixed
* Arrays are declared with square brackets after the type of the array

int[] myInts = new int[]{1, 2, 3, 4};

String languages[] = {"Java", "JavaScript", "SQL"};

**What is varargs?**

* Variable Arguments, allows us to set an argument to a method whose size is determined at runtime
* Treated as an array under the hood, Java will create an array to fit the arguments which you provide
* **MUST** be the last parameter
* Can omit the vararg value and Java creates an array of size 0

**What is an Error?**

* An Error is something that went so horribly wrong with your application that you should not attempt to recover from
  + ExceptionInInitializerError
  + OutOfMemoryError
  + StackOverflowError

**What is an Unchecked Exception?**

* An Unchecked Exception that your application tends to be unexpected but are not necessarily fatal
  + Any class that extends RuntimeException
  + Unchecked Exceptions do not require to be handled or declared

**What is a Checked Exception?**

* A Checked Exception is something in your application that tends to be more anticipated and "might be reasonably recovered from"
  + **MUST** be handled or declared
    - *handled* means that the risky code is placed inside a try/catch block
    - *declared* means that the type of exception to be thrown must be declared in the method signature with the throws keyword

**What is the difference between String, StringBuffer, and StringBuilder?**

* String is immutable and slowest of the three
* StringBuffer is mutable and thread-safe
* StringBuilder is mutable and the fastest of the three

**What is StringTokenizer?**

* StringTokenizer is an object which parses a String and splits it based on a delimiter

**What is the difference between throw, throws, and Throwable?**

* throw is a keyword which explicitly throws an exception in a method implementation
* throws is a keyword which declares an exception that could be thrown in the method signature
* Throwable is the base object of the Exception hierarchy

**What is the Reflection API?**

* Reflection allows us to inspect and/or modify runtime attributes of classes, interfaces, fields, and methods

**What is File I/O?**

* File I/O refers to the nature of how data is accessed, either by reading it from a resource (input) or writing it to a resource (output)
* In Java, File I/O objects fall into one of two categories
  + Streams are for reading or writing bytes
  + Reader/Writer are for reading or writing characters
* Some Common File I/O classes are
  + FileInputStream - reads raw bytes from a file
  + FileOutputStream - writes raw bytes to a file
  + FileReader - reads characters from a file
  + FileWriter - writes characters to a file
  + BufferedReader - reads a file line by line, needs an instance of a FileReader with a path to the resource to be read in the constructor
  + BufferedWriter - writes to a file line by line, needs an instance of a FileWriter with a path to the destination file in the constructor
* **ALWAYS** close your resources!
* Character files are read line by line, either until a carriage return (\r) or a newline (\n)

**What is Serialization?**

* Serialization is the process of writing the state of an object to a byte stream
* In order for an object to be serialized, it must implement the Serializable interface
* To serialize an Object, you need a FileOutputStream instance inside the constructor of an ObjectOutputStream, passing in the file path of where you want the Object to be serialized
  + Call the ObjectOutputStream.writeObject(yourBean) method
* To deserialize an Object, you need a FileInputStream instance inside the constructor of an ObjectInputStream, passing in the file path of where the serialized object is
  + Call the ObjectInputStream.readObject() method, casting it to a bean of your type

**What is Generics?**

* Generics enforce compile time safety by allowing you to use parameterized types
* Before Java 5, you had to write something like this and hope other developers understood to only put Strings inside
  + List names = new ArrayList();
* Since then, you can use the diamond operator and allow the compiler to prevent you from causing this problem
  + List<String> names = new ArrayList<>();
* Naming Convention for Generics
  + Technically, Type Parameters can be named anything you want. The convention is to use single, uppercase letters to make it obvious that they are not real class names
  + E => Element
  + K => Map Key
  + V => Map Value
  + N => Number
  + T => Generic data type
  + S, U, V, and so on => For multiple generic data types

**What is the Collections Framework?**

* Collections Framework in Java is a set of classes and interfaces that implement commonly used data structures
* A collection is a single object which acts as a container for other objects

**List<E> - Interface**

\* ordered => preserves the order in which inserted

\* duplicate entries allowed

\* elements accessed by index

**ArrayList<E> implements List<E>**

\* resizable array

\* traversal is fast (constant time)

\* insertion/removal is slow (linear time, since there is a risk of having to resize the underlying array)

**LinkedList<E> implements List<E>, Queue<E>**

\* implements both List and Queue => has all methods in both interfaces

\* insertion/removal is fast (no risk to resize)

\* traversal is slow for an arbitrary index

**Vector<E> implements List<E>**

\* Old school, essentially a thread-safe implementation of an ```ArrayList```

**Stack<E> implements List<E>**

\* Old school, should use an ```ArrayDeque``` for the Stack data structure

**Set<E> - Interface**

\* NOT index driven

\* Unique elements allowed only

\* DOES NOT preserve the order in which they were inserted

**HashSet<E> implements Set<E>**

\* Backed by a HashMap<K, V>

\* Guarantees no ordering when iterating

\* Allows one null value

\* Insertion/Traversal are fast

\* Tradeoff is that it does not maintain order in which you insert elements

**TreeSet<E> implements Set<E>**

\* Backed by a Sorted Tree

\* Main benefit is that they maintain sorted order

\* Tradeoff is that insertion/removal are slow, because the elements must maintain sorted order

\* Cannot contain any null values, since null cannot be compared to any object

**Queue<E> - Interface**

\* Used when elements should be added and removed in a specific order

\* Unless specified, elements are ordered FIFO

**ArrayDeque<E> implements Queue<E>**

\* Pronounced as 'deck'

\* Implementation of a pure double-ended queue

\* Stores elements in a resizable array

\* can be implemented as either a Queue or a Stack

\* If Queue, we use offer/peek/poll

\* If Stack, we use push/poll/peek

**ArrayDeque<E> Methods**

| **Operation** | **Throws Exception** | **Returns null** |
| --- | --- | --- |
| Insert | boolean add(E e) | boolean offer(E e) |
| Remove | E remove() | E poll() |
| Examine | E element() | E peek() |
| void push(E e) is for a Stack (LIFO), which adds elements to the front of the queue |  |  |

**Map<K, V> Interface**

\* Used to identify a value by a key, each element in a map is a Key-Value pair

\* NOTE: Map does not implement the Collection interface, however it is considered to be part of the Collections framework

**HashMap<K,V> implements Map<K,V>**

\* Stores elements in key-value pairs

\* Insertion/Retrieval of elemeny by key is fast

\* Tradeoff is that it does not maintain the order of insertion

\* Permits one null key and null values

**TreeMap<K,V> implements Map<K,V>**

\* Keys are stored in a Sorted Tree structure

\* Main benefit is that keys are always in a sorted order

\* Insertion/Retrieval are slow

\* Cannot contain null keys as null cannot be compared for sorting

**Hashtable<K,V> implements Map<K,V>**

\* Old school, thread-safe implementation of a HashMap

\* Does not allow null keys or null values

**What is java.lang.Comparable?**

* Comparable is an interface which defines a class' natural ordering
* Class must implement Comparable if it is to be sorted by the compareTo() method

public interface Comparable<T> {

public int compareTo(T o);

}

* Returns an int... what does it mean?
  + Zero, if the two objects are equal
  + Negative, if this object is smaller than that
  + Positive, if this object is greater than that

**What is java.util.Comparator?**

* Comparator is an interface which allows you to sort classes by more than one variable
* Class that is to be sorted by compare() does not have to implement Comparator

public interface Comparator<T> {

public int compare(T firstObject, T secondObject);

}

**What's so special about java.lang.Object?**

* Every class, directly or indirectly, inherits from Object, which means that all classes inherit these 11 methods
  1. Object clone()
  2. boolean equals(Object o)
  3. void finalize()
  4. Class<?> getClass()
  5. int hashCode()
  6. void notify()
  7. void notifyAll()
  8. String toString()
  9. void wait()
  10. void wait(long timeout)
  11. void wait(long timeout, int nanos)
* toString()
  1. Automatically called if you print an Object
  2. Usually overridden to provide human-readable output
  3. Otherwise, you will print fill.qualified.ClassName@memoryAddress
* equals(Object o)
  1. What's the difference between == and .equals()?
  2. Unless you explicitly override it, you are only checking if references point to the same object
  3. The method implementation in java.lang.Object of the .equals() method is
* public boolean equals(Object o) {
* return (this == o);

}

* hashCode()
  1. You are expected to override hashCode() if you override equals()
  2. A hash code is a number that puts instances of a class into a finite number of categories
     + The result of hashCode() should not change in a program
     + if .equals() returns true, the hash codes should be equal
     + if .equals() returns false, the hash codes do not have to be distinct. However, doing so will help the performance of hash tables

**What is a process?**

* First, we must talk about *programs*. A program is a blob of binary data consisting of a series of instructions for the CPU organized into a package and stored on the hard drive
* A *process* is an allocation of space in working memory (RAM) which serves as a copy of those program instructions and resources
  + So technically, when we say we are running a program, what we really mean is that we are running a copy of it, a process
* Processes consist of many threads
* When a Hava program starts up, one thread begins running immediately. The "main" thread
  + is the thread from which other threads spawn
  + must be the last thread to finish execution. When it stops, your program terminates

**What is a thread?**

* A thread is a subset of a process that is also an independent sequence of execution, but threads of the main process run in the same memory space, managed independently by a scheduler

**What is Multithreading?**

* Extends the idea of multitasking into applications where you can subdivide operations in a single application into individual, parallel threads
* Each thread can have its own task
* OS divides processing time not just with applications, but between threads

**What is the lifecycle of a Thread?**

* At any given time, a Thread can be in only one of these states
  1. New: newly created thread that has not started executing
  2. Runnable: either running or ready for execution but waiting for its resource allocation
  3. Blocked: waiting to acquire a monitor lock to enter or re-enter a synchronized block/method
  4. Waiting: waiting for some other thread to perform an action without any time limit
  5. Timed\_Waiting: waiting for some other thread to perform a specific action for a specified time period
  6. Terminated: has completed its execution

**What is a Thread Priority?**

* Priorities signify which order threads are to be run
  + MIN\_PRIORITY = 1
  + NORM\_PRIORITY = 5, default
  + MAX\_PRIORITY = 10

**How do we create a Thread?**

* We have 2 options
  1. Create a class that implements Runnable
     + implement the run() method
     + pass an instance of your class to a Thread constructor
     + call the start() method
* public class MyRunnable implements Runnable {
* @Override
* public void run() {
* System.out.println("Inside the MyRunnable class");
* }

}

* 1. Create a class that extends Thread
     + implement the run() method
     + instantiate your class
     + call the start() method

public class MyThread extends Thread {

@Override

public void run() {

System.out.println("Inside the MyThread class");

}

}

public class ThreadDemo {

public static void main(String[] args) {

Thread myRunnable = new Thread(new MyRunnable());

Thread myThread = new MyThread();

myRunnable.start();

myThread.start();

}

}

**What is Deadlock?**

* Deadlock describes a situation where 2 or more threads are blocked trying to access the same resource, waiting for the other

**What is the purpose of the synchronized keyword?**

* In a multithreaded environment, a race condition occurs when 2 or more threads attempt to access the same resource
* Using the synchronized keyword on a piece of logic enforces that only one thread can access the resource at any given time

**What is Test Driven Development? (TDD)**

1. Write a unit test
2. Run the test => test will fail
3. Fix the test by writing application code
4. Retest until the test passes
5. Repeat

**What is jUnit?**

* jUnit is a Java API for unit level testing
* Assertions verify that the state of the application meets what is expected

**What is a Unit Test?**

* Unit testing is the testing of individual software components, modules and methods

**What is Maven?**

* Maven is a dependency manager and build automation tool

**Where are Maven dependencies stored?**

* Locally => in the .m2/repository folder (can be changed in $MAVEN\_HOME/conf/settings.xml)
* Globally => mvnrepository.com
* Project => workspace

**What is a Maven Goal?**

* A goal represents specific tasks which contribute to the building or managing of projects
  + clean => deletes all compile code (anything in the /target folder)
  + compile => compiles source code
  + package => performs all tests and packages to WAR/JAR
  + install => installs WAR/JAR to local repository
  + deploy => copies final WAR/JAR to remote repository (a server, usually)

**What is the Maven Build Lifecycle?**

1. Validate => project is correct and all necessary information is available
2. Compile => compiles project source code
3. Test => tests all compiled code
4. Package => packages all compiled code to WAR/JAR file
5. Integration => performs all integration tests on WAR/JAR
6. Verify => runs checks on the results of integration tests
7. Install => installs WAR/JAR to local repository
8. Deploy => copies final WAR/JAR to the remote repository

**What is a POM?**

* Project Object Model is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration detauls used by Maven to build the project; the configuration includes project dependencies and plugins

**What is a Design Pattern?**

* Design patterns are established general solutions to commonly occuring software development problems
  + Creational
  + Structural
  + Behavioral

**What is a Singleton?**

* Singleton is a design pattern which creates an object in memory only once in an application and can be shared accross multiple classes
* To make a class into a Singleton,
  1. private static variable of the class' type
  2. private constructor
  3. public static getInstance() method, which will either instantiate the object or return the instance in memory

public class Singleton {

// Private static variable of the class' type

private static Singleton instance;

// Private Constructor

private Singleton() {}

// Public static getInstance method

public static Singleton getInstance() {

if (instance == null)

instance = new Singleton();

return instance;

}

}

**What is a Factory?**

* Factory is a design pattern which creates objects in which the precise type may not be known until runtime
* To make a factory
  1. Abstract data type
  2. Classes that inherit the abstract data type (the objects whose instantiation details may not be known until runtime)
  3. Static method that returns a concrete instance, whose return type is the abstract data type in (1.)

// Abstract Data Type

public interface Dessert {}

// Classes that inherit the abstract data type

public class Cake implements Dessert {}

public class Cookie implements Dessert {}

public class IceCream implements Dessert {}

// Good practice to throw an exception if the desired concrete instance is not found

public class DessertNotFoundException extends RuntimeException {}

// Factory class that returns the concrete instance

public class DessertFactory {

public static Dessert getDessert(String dessertType) {

switch(dessertType) {

case "cake":

return new Cake();

case "cookie":

return new Cookie();

case "ice cream":

return new IceCream();

default:

throw new DessertNotFoundException(dessertType + " not found!");

}

}

public class Demo {

public static void main(String[] args) {

Dessert d1 = DessertFactory.getDessert("cake");

Dessert d2 = DessertFactory.getDessert("cookie");

Dessert d3 = DessertFactory.getDessert("ice cream");

Dessert d4 = DessertFactory.getDessert("candy"); // Throws DessertNotFoundException

}

}

}

**What is log4j?**

* Reliable, fast, and flexible logging framework supported by Apache

**Why do we need logging?**

* Logging records events that occur during software execution
* As users execute programs on the client side, the system accumulates log entries for the support team
* In general, it allows for developers to access information about applications in which we do not have direct access

**What are the log levels?**

1. ALL => all levels
2. DEBUG => designates fine-grained informational events that are most useful to debug an application
3. INFO => informational messages that highlight the progress of the application at the coarse grained level
4. WARN => designates potentially harmful situations
5. ERROR => designates error events that might still allow the application to continue running
6. FATAL => severe error events that presumably lead the application to abort
7. OFF => highest possible level, intended to turn off logging

**How do logging levels work?**

* A log request of level *x* in a logger with level *y* is enabled with *x >= y*
* For the standard levels, we have that ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF