- **Why Java**
    - Platform Independence. Java works on any OS that supports JVM
    - JVM: Interprets compiled Java bytecode for the computer
    - Garbage collection: don't need to worry about memory management
    - Object Oriented: almost everything is an object and helps us program in the way we see the world.
    - Open Source: Freeeeee
    - Libraries: A substantial amount of built in libraries so we don't have to "reinvent" the wheel
    - Large Job Market: Get PAAAAAID
- **OOP: 4 pillars**
    - Encapsulation
        - A mechanism of wrapping the data (variables) and the code acting on the data (methods) together as a single unit. Also, variables of a class will be hidden from other classes and can only be accessed through methods of that class. (Data Hiding)
    - Polymorphism
        - The ability of an object to take many forms. Method overloading and method overriding.
            - Method Overloading: When you have multiple methods with the same name but different parameters. Java will determine which one is the best fit.
            - Method Overriding: When a class redefines a method that was given to it via an interface or a parent class.
    - Abstraction
        - Hiding the implementation (give us a general idea of what something will do without going into details).
    - Inheritance
        - The ability to derive a new class from an existing class
- **Naming Conventions**
    - Variables, methods - camelCase
    - Classes - PascalCase
    - Constants - UPPER_CASE_WITH_UNDERSCORES
    - Packages - reverse.domain.name
- **Constructors**
    - First line of a constructor must be either super(); or this();. If it is neither of those, then it is inferred to be super();
    - Must be the named the same as the class and must not have a return
    - 2 types of constructors:
        - Default (no parameters)
        - Parameterized (has parameters)
- **Scopes of a variable**

    - Static/class : Doesn't need to be instantiated (closest thing java has to a global scope
    - Instance: scoped to a particular instance of that class
    - Method: denotes parameters passed into the method a (int params) params are in the method scope
    - Block: exists within anyset of {}
- **Control Statements**
    - If-else
    - While
    - Do-While : executes at least once
    - For
    - Switch(case):
        - Case:
            - Can't be boolean
            - Can't be float
            - Can't be double
- **Short Circuit Operators**
    - && and ||. Checks the first condition, if the second condition will have no effect on the outcome then it will just continue on without checking the second condition.
- **Arrays**
    - A fixed container of data
    - Always indexed from 0
    - Can store anything
    - Brackets can go either after the type declaration or after the name String[] arr or String arr[]
    - Double arrays have 2 brackets: string doubleArr[][]
- **Var Args**
    - Allow you to pass in 0 to infinite number of arguements to a method
    - Public void myMethod(int ... params)
    - Var Args must be the last item in the list of parameters
    - The var Args will be an array
- **For each loop (enhanced for loop)**
    - for(int elem : elements) sysout(elem);
- **Packages: folder structure**
    - A group of similar types of classes, interfaces and sub-packages
    - 2 types: built-in and user-defined
- **Access Modifiers**
    - Public : access from anywhere
    - Protected: Class, package, and children
    - Private: Class
    - Default (no access modifier) class and package
- **Strings**

    - Strings are Immutable
    - Strings are an Object
    - **StringPool**
        - This is where Strings are supposed to be stored
        - Java will try to reuse strings. If there are two “hello” strings there will be only 1 “hello” int the StringPool
        - *Things to Note regarding the String Pool:*
            - When using “new” to make a string, it will be on the heap
            - All String.methods will add a string to the heap
- **StringBuilder and StringBuffer**
    - Ways to make mutable String
    - String Bulider is not threadsafe
    - StringBuffer is threadsafe
- **Exception Handling**
    - Try-catch-finally
        - Try needs either a catch or a finally or both
            - Finally will always be run
        - Catch order matters. Can have multiple catches for a single try
        - Try with resources Try(blahblah){} will automatically close the item in the try();
- **Exception types**
    - **Checked**
        - CompileTime Exceptions: have to be checked
        - Require us to handle before code will compile
        - Must be surrounded by a try-catch block or requires the method to have a throws
        - All IO Exceptions
    - **Unchecked**
        - RunTime Exceptions
        - Do not require any catches
        - Can be avoided with logic
    - **Error**
        - Implements the throwable class
        - Usually a fatal event that will kill your program
        - StackOverflow
- **Garbage Collection**
    - Automagic memory management
    - Always calls finalize() before it deletes the object;
    - Can be requested by using System.gc()
- **Object class**
    - Objects are the father class that all other classes extend from
- **Wrapper Classes**
    - Object representation of a primitive

    - Autoboxing: automatically converting the primitive into the wrapper
    - AutoUnboxing: automatically converting the wrapper into the primitive
- **Reflection:**
    - view and modify classes at runTime
    - Java advises not using reflection
- **Overriding/Overloading**
    - **Overloading:**
        - Having multiple methods with a single name but with different parameters
    - **Overriding:**
        - Changing a method that was received by an interface or a parent class
- **Interface vs Abstract Class**
    - **Interfaces:**
        - All implicitly abstract methods. We don't define any methods as being abstract. They cannot have a method body unless declared default or static
        - Implicitly public interfaces
        - Implemented using the implements keyword
        - A class can implement 65k interfaces
        - Cannot be instantiated
        - Use when no concrete methods
    - **Abstract Classes:**
        - Can have abstract methods and concrete methods. Have to specify abstract on the abstract methods.
        - Have to specify access modifiers
        - Extended using the extends keyword
        - Classes can only extend 1 class. Doesn't matter if the class is abstract or not
        - Cannot be instantiated
        - Use where some concrete methods
- **Generics**
    - Allows us to construct a class in a way where we can ignore type to some degree but allows flexibility for handling Multiple types
    - All Containers are generic
    - For class end with <E>
- **Comparing objects**
    - **Comparable<T>**
        - Interface with the method .compareTo(T other){}
        - Implemented on the object itself
        - For natural ordering
        - Can only be one
    - **Comparator<T>**
        - Functional Interface with method .compare(T one, T two){}
        - Implemented on a separate object

        - Unnatural oder (we can pick order)
        - Can be an infinite amount
- **Final**
    - Final variables are constant and can't be modified
    - Final classes cannot be extended
    - Final methods cannot be overriden
    - Final Objects can have their variables changed, but the Object cannot be reassigned
- **Git**
- **Serialization**
    - Conversion between object and bytecode
    - To make a class serializable it must implement serializable
    - ObjectInputStream and ObjectOutputStream are used to serialization
        - Require an InputStream for input and an OutputStream for output
- **BufferedReader and Writer**
    - Reads lines at a time
    - Requires a Reader/Writer during instantiation
- **Collection Interface (a group of data)**
    - Inherits from Iterable
        - Iterable comes with an Iterator
    - **List**
        - Ordered which means we can access by index
        - No restriction on duplicates
            - ArrayList
                - Underlying storage is an array
                - Very fast access
                - Inefficient when growing
            - LinkedList
                - Stored into a doubly linked list
                - Efficient for adding and inserting
    - **Set**
        - No guaranteed order, thus no access via index
        - No duplicates
            - Hashset
                - Faster
                - Stores data of an object based off the Hash value
                    - WARNING: if an object is updated while in the HashSet then it's new hash won't match the hash that's stored in the set
            - TreeSet
                - Creates a binary tree
                - Tree will rebalance itself when its depth is 2 more on one 1 side

        - Pretty fast at reading
    - **Queue**
        - Ordered
        - Can only push to back
        - Can only pop from front
        - Usually FIFO
            - Exception: PriorityQueues allow contents to have a priority which may push them to the front
        - Types of Queues:
            - LinkedList
            - PriorityQueue
    - **Deque**
        - Can read and write from either side
- **Marker Interfaces**
    - An empty interface
        - Important marker interfaces
            - Serializable
            - Cloneable
            - Remote
            - Threadsafe
- **Functional Interface**
    - An interface with only 1 abstract method
    - *Built in Java interfaces are*
        - *Function<T,R> : takes in one parameter T and return type R*
            - *IntFunction, LongFunction, DoubleFunction*
            - *ToIntFunction, ToLongFunction, ToDoubleFunction*
            - *DoubletoIntFunction, etc*
        - *BiFunctions*
        - *Supplies<R> supplies something but takes nothing*
        - *Consumers<T> takes in a parameter T and returns nothing*
        - *Predicate<T> takes a value and returns a boolean*
        - *Operators receive and return the same kind of value*
            - *Replace all is an example*
- **Lamba**
    - Anonymous function
        - Methods belong to a class
        - Functions don't belong to anything
- **Lambda notation**
    - () -> {}
- **Streams**
    - Stream<E> is a generic stream
    - There are primitive streams
    - .boxed() converts a primitive stream to a generic stream

    - .filter() removes unwanted items using a Predicate
    - .map() converts the contents of the stream
    - .reduce() shrinks the stream to one time
    - .foreach() uses a consumer on each item in the stream
    - Intermediate operations return streams
    - Terminal operations return an object
- **Ternary:**
    - 3 point operator (condition ? true : false)
- **What is a thread**
    - A thread of execution in a program that will allow to run things concurrently
- **Why use a thread**
    - Allows concurrent execution within a program
    - Can decrease time needed for a program to execute
- **Extending Thread vs Implementing Runnable**
    - Extending thread is not the preferred way because of overhead and built in that could cause issues if unwisely overridden
    - Implementing Runnable is the preferred way since runnable is a functional interface that Thread's constructor can take. It's only method is run()
- **isAlive()**
    - Checks to see if the thread exists in a state between new and terminated
- **join()**
    - Will halt the current thread until the thread that join is called on terminates
- **States**
    - New: when we first create a thread
    - Runnable: when the thread is running
    - TimedWating: when the thread is waiting a predefined amount of time
    - Waiting: when the thread is waiting indefinitely (will need to woken up)
    - Blocked: when the thread is waiting for a resource that is being used by another thread
    - Terminated: when the thread is dead
- **Synchronized**
    - Used to allow only 1 thread to access at a time
        - For methods just include synchronized in between the access modifier and the return declaration
        - Can have synchronized statements
            - synchronize(obj) {}
            - Require the obj that will be modified to be specified in the ()
- **Deadlock**
    - When two threads are trying to access things each other posses and thus cannot proceed
- **producer/consumer problem**
    - A producer produces and a consumer consumes. Extremely inefficient and may cause error.