Pillars of OOP

* Abstraction
  + Hiding the implementation of our code
    - When we uses interfaces and abstract classes
    - We know what the code is able to do and how to call its methods but we do not know (or really care) how it works/ is implemented
* Polymorphism
  + The ability for one object to take on multiple forms
    - Many labels for a single object
      * Animal a = new Cat();
    - Static Polymorphism/ Compile time polymorphism
      * Overloading – same method name different parameters
    - Dynamic Polymorphism/ Runtime polymorphism
      * Overriding – same method signature different implementation in the child class
* Inheritance
  + Creates an is-a relationship between two classes
  + Allows a child class to inherit/ take on the methods and variables in the parent class
    - It allows to share our code
    - Avoid writing the same functionality multiple times
    - DRY (Do Not Repeat yourself)
      * If you find yourself writing a method almost identical to something in another class than you might want to reexamine your class structure.
* Encapsulation
  + Allows us to protect our code and variables
  + Done using access modifiers keywords
    - Prevents variables from getting values that they should not
    - Prevents helper methods from being called
    - Forces developers to use the public methods of a class to interact with the object

High Level Java Features

* WORA (Write Once Run Anywhere)
  + Any Java program can run on any JVM regardless of the operating system or hardware
* 99% OOP language
  + Primitives
  + Lambdas (Functional programming)
* Strongly and statically typed
  + Strong your types cannot be coerced into other types
    - You cannot do “hello” == 8;
  + Static you must declare the types of your variables
    - int x =100;
* Automatic memory management
  + You do not manage pointers
  + Garbage collection – objects in memory that are no longer referenced/available are able to be deleted
* Compiled language
  + Write source code in a something.java file that is human readable
  + Compiles into a something.class file which is the Java Byte code that a JVM can execute

JDK vs JRE vs JVM

* JDK (Java Development Kit) – needed to write Java applications. The JDK contains the compiler.
* JRE (Java Runtime Environment) – needed to run Java applications. What most users of Java have.
* JVM (Java Virtual Machine) – Think of virtual hardware. It reads in byte code and can perform memory operations and register operations its virtual ram and registers.

Access Modifiers in Java

* Public – accessible anywhere
* Protected – the package plus inheriting children in other packages
* (default) – the package
* Private – Just the class

Scopes in Java

* Static/ class – variable belongs to the class itself
* Instance/ object – the variable is attached to an instance of the class
* Method – the variable passed in as a parameter to the method
* Local – variables declared within a method

Constructors

* Special methods that create an instance of a class
* They must be named the same as the class
* They do not return type
* You can overload constructors
* First line must be a call to super()
* You get a default no args constructor if no constructors are declared
* Cannot override constructors
  + They are inherited
  + It does not make sense for the child to dictate how its predecessor was built
  + Constructors must be named the same as the class so parent and child constructors have different signatures

Overloading vs Overriding

* Overloading
  + Same method name different parameters
* Overriding
  + Same method signature different implementation in the child class

Abstract Classes vs Interfaces

* Abstract classes
  + A class that you cannot directly instantiate
  + Otherwise they have all the properties of a regular class
  + They can have instance variables
  + You can only inherit one
  + Because you cannot directly instantiate, new Thing(), them you can safely make their methods abstract
    - OPTIONAL
    - You do not have to have abstract methods in an abstract class
* Interfaces
  + You can only have public static final variables
  + No instance variables
  + No constructors
  + Everything public
  + You cannot instantiate them
  + All their methods are abstract by default
    - Technically in Java 8 you can use the default keyword to create concrete methods in an interface
  + You can implement as many interfaces as you want
  + Special types of interfaces
    - Marker interfaces (no methods just provide a type to a class)
    - Functional interfaces (used to create custom lambdas)

Java Collections Framework

* JCF is a conglomeration of interfaces and classes whose main purpose is to store and organize other objects
* The top interface is Iterable
* Main interfaces to know
  + Collection Interface (Most collections in the framework implement)
  + List
    - Maintain the order of insertion
    - Can have duplicates
      * Implementation Classes: ArrayList and LinkedList
  + Set
    - Cannot have duplicates
    - Does not maintain the order of insertion
    - Does not mean they are not ordered
      * Implementation Classes: HashSet and TreeSet
  + Queue
    - FIFO (First In First Out)
      * Implemented by Dequeue
  + Map
    - Stores objects as key value pairs
    - DOES NOT IMPLEMENT ITERABLE OR COLLECTION
    - You cannot iterate over a map
    - Implementing collection
      * Implementations : HashMap Hashtable
* Collections
  + This is a utility with a lot of static methods for dealing with collections
    - Sort
    - Reverse
    - getIndexOf
    - duplicates

<Generics>

* Generics use the <> diamond brackets
* They allow you to specify a type for an interface or a class
* Very helpful in collections where we can restrict the type of object that goes into our collection
  + 90% of the time I use generics

Exceptions and Errors

* Exceptions occur when the code executes along the unhappy path of execution. (Something goes wrong)
* Developers are expected to write code to deal with exceptions
* Two main types of exceptions
  + Checked or compile time exceptions
    - Must be handled in order for your code to compile
  + Unchecked or runtime exceptions
    - Are not required to be handled for your code to compile
    - Usually a good idea to handle them anyway
* Errors
  + Catastrophic failures that developers should not handle
    - OutOfMemory, StackOverflow etc
* Two options for handling exceptions
  + Put the failable code in try-catch block
  + Use throws keyword in the method signature to pass the exception to whoever called the method

Design Patterns

* Agreed upon software design solutions to solve recurring problems in programming.
  + Factory design pattern
    - Problem: You know what you need to do however you do not know what implantation class you need
    - Solution: Have a factory method that returns you an implementation class that fulfills an interface you need.
  + Singleton design pattern
    - Problem: I need only one instance of this class
    - Solution: Private constructor with a public getInstance method that returns the same object everytime

Strings

* Strings ARE NOT PRIMITIVES
* Strings are objects
* Immutable
* You can create String using literal syntax String s = “something”;
* Strings created in such a way use the string pool
  + String pool is a special location in memory reserved just for strings
  + Strings of the same value are the same object
* StringBuilders
  + Mutable version of String
* StringBuffers
  + Threadsafe version of StringBuilders

Object Class

* The Root class of all classes in Java
* Every object is of type Object
* Object Methods to know
  + Equals method to override to say whether this object is “equal ” to another object
  + toString – string representation of our object
  + Hashcode – The hash number of our object

Comparable

* Interface that a class can implement
* Gives the comapreTo method
  + Return -1 for smaller
  + 1 for larger
  + 0 for same
* Referred to as the natural ordering for a class

Wrapper Classes

* Object version of primitives
* Wrapper => Primitive (Unboxing video)
* Primitive => Wrapper (Autoboxing)

Threads

* Threads are a path of execution through the program
* Java supports multithreading
* Threading can increase application efficiency
* Can also cause concurrency problems
* How to make a thread
  + 1. Create a Runnable lambda with a piece of code
  + 2. Create a thread and pass in the lambda to the constructor
  + 3. Thread.start() to have the thread begin executing

Lambdas

* Lambda is part of functional programming
* Storing executable code as an object
* We can pass around lambda just like any other objects
* We can pass lambdas as parameters to other methods