**Useful Java things you don’t have to build:**

* constants in java: int final CONSTANT, or for global, public static final int CONSTANT
* static fields/variables in a class apply to all instances
  + static methods associate it with the type not with an instance - this is efficient in memory because it only stores it once
* overloading
* Scanner(system.in) and BufferedReader(FileReader f); Stringbuilder (StringBuffer is threadsafe but much slower); File; PrintStream
* Iterator; ListIterator lets you go backwards
* specific exceptions (e.g. throw new IllegalArgumentException(“Welp”);
* either “throws FileNotFoundException” to ignore it or use try catch finally

Efficient I/O code example:

BufferedReader f = new BufferedReader(new FileReader("dualpal.in"));

PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter("dualpal.out")));

StringTokenizer st = new StringTokenizer(f.readLine());

int N = Integer.parseInt(st.nextToken());

int S = Integer.parseInt(st.nextToken());

f.close();

out.close();

System.exit(0);

* Every object by default has toString, equals, getClass, hashCode; rewrite any of these if specification needed; you should write compareTo if relevant, the class should have “implements Comparable<ClassName>”
  + In theory you should rewrite hashCode iff you rewrite equals but no one really cares that much?
* For objects, consider if “o instanceof Point” or something like that
* For each loops (use an iterator for time efficiency)
* Boxing and Unboxing
* HashSet, HashMap, TreeSet, TreeMap
  + You can initialize a HashSet with a List
  + you can iterate through sets (not maps) if you have an iterator
* Inheritance: extends
  + | Class | Package | Subclass | World  
    ————————————+———————+—————————+——————————+———————  
    public      |  y    |    y    |    y     |   y  
    ————————————+———————+—————————+——————————+———————  
    protected   |  y    |    y    |    y     |   n  
    ————————————+———————+—————————+——————————+———————  
    no modifier |  y    |    y    |    n     |   n  
    ————————————+———————+—————————+——————————+———————  
    private     |  y    |    n    |    n     |   n
  + method overriding (differs from overloading). @Override is good practice
  + abstract classes
* Interfaces - can have multiple
* Stack has its own impementation, queue is built off of list, tuple, priority queue (uses a heap), set, sorted set (uses a set and a linked list), maps/dictionaries
* You can use interfaces as object types (for interface Fillable, you can call Fillable[] fillables = new Fillable[4] for an array of 4 fillable things)
* Generics:
  + function example: public static <E> void printList(ArrayList<E> list)
    - public static <E extends Comparable<E>> Tuple2<E,E> getLargestAndSmallest(E[] arr)
  + class example: class Tuple2<T1,T2>
  + Wildcard type: public static <E> void foo(List <E> list){} /\* == \*/ public static void foo(List<?> list){}
* Invariance, Covariance and Contravariance
* Nested Classes (static nested work outside the class but can use static stuff in class
  + public class Foo{
  + private static final int SECRET\_TO\_LIFE = 42;
  + public static class Bar{
  + public int getSecret(){
  + return SECRET\_TO\_LIFE;
  + }
  + }
  + }
* If not static, inner class defined with respect to instance
* Local class:
  + tupleList.sort(new Comparator<Tuple2<A, B>>(){
  + //public int compare(...)
  + //class body
  + })
* Callbacks
  + public class ListHelper1 {
  + public interface MapFunction<A, B> { //callback in that you can change this interface how you want to chance everything else
  + public B apply(A input); //this is a special type of functional interface since we only want one function so in Java 8 they changed this
  + }
  + public static <A, B> List<B> map(List<A> list, MapFunction<A, B> func){
  + List<B> mappedList = new ArrayList<>();
  + for(A elem : list){
  + mappedList.add(func.apply(elem));
  + }
  + return mappedList;
  + }
  + public static void main(String[] args){
  + List<String> words = new ArrayList<>();
  + words.add("hello");
  + words.add("goodbye");
  + words.add("hi");
  + List<Integer> lengths = map(words, new MapFunction<String, Integer>(){
  + public Integer apply(String str){
  + return str.length();
  + }
  + });
  + for(int length : length){
  + System.out.println(length);
  + }
  + }
  + }
* Lambda expressions
  + import java.util.function.Function;
  + public class ListHelper1 {
  + public static <A, B> List<B> map(List<A> list, Function<A, B> func){ //thanks to the import
  + List<B> mappedList = new ArrayList<>();
  + for(A elem : list){
  + mappedList.add(func.apply(elem));
  + }
  + return mappedList;
  + }
  + public static void main(String[] args){
  + List<String> words = new ArrayList<>();
  + words.add("hello");
  + words.add("goodbye");
  + words.add("hi");
  + List<Integer> lengths = map(words, str -> str.length()); //this is called a Lambda Expression
  + for(int length : length){
  + System.out.println(length);
  + }
  + }
  + }