Today's Agenda

> To give a practical introduction to data structures

> To look specifically at Lists, Sets, and Maps

> To talk briefly about Generics in Java

> To talk about interfaces in Java

Data Structures

Data structures are objects of data stored in memory in a particular way along with accompanying methods to access them in various ways.

Different data structures are appropriate for different tasks, and identifying the appropriate data structure can speed up your code immensely.

Some of the general questions you would ask to determine an appropriate data structure include:

Does the data need to be sorted?

Will the data generally accessed in order?

Or will I need "random access" to the data?

Will I need to multiple copies of the same item?

Can I look up my data with a unique key?

How will I most often traverse my data?

Interfaces

An interface is set of methods that provides a "contract" for any classes that implement it to fulfill.

It provides no actual code, but simply guarantees that the implementing code must include those methods.

A class can implement more than one interface at a time.

```
public interface FlyingBehavior {
     public void flyAlgorithm();
}
public interface ChirpingBehavior {
     public void chirpAlgorithm();
}

public class BaldEagle implements FlyingBehavior, ChirpingBehavior {
     public void flyAlgorithm() {
          swoopThenDive();
     }
     public void chirpingAlogrithm() {
          println("ca-caaaw!");
     }
     private void swoopThenDive() {
          //do swooping and diving stuff...
     }
}
```

Collections

Java includes a number of common data structures within the Collections Framework

All Java data structures implement either the "collection" interface or the "map" interface.

The ***collection*** interface requires all implementing classes to include a few basic methods universal to any collection of elements, including

the ability to add and remove an object to the data structure
the ability to get the number of elements stored in the data structure
the ability to iterate through the elements in order

Additionally, a number of basic algorithms can be run on these collections: sorting, searching, shuffling, etc (see the javadocs for *java.util.collections*)

Lists

The most common collection is a List.

A List is a sub-interface of Collection which requires some extra functionality, such as:

the ability to get or set an object at a particular index
the ability to grab a slice of the list

There are lots of different types of Lists which have specialized functionality, but the most common are ArrayList and LinkedList.

An ArrayList uses a native "primitive" array which makes it easy to grab things instantly at any position, but is slower to add things to it (unless you are added to the end of it).

A LinkedList provides links between each element and its neighbors. It is slower to grab something in the middle of the list (as you have to traverse the List until you find it), but much faster to manipulate. If you add something to it you simply update the links, rather than shifting all of the elements.

(draw on board)

Lists

```
public class ListTest {
    public ListTest(){
     List<Integer> ints = new ArrayList<Integer>();

     for (int i = 10; i >= 0; i--) {
          ints.add(i);
     }

     Collections.sort(ints);

     for (Integer i : ints) {
          System.out.println(i);
     }
}
```

Maps

A Map is a super fast collection which using a hashing function to quickly store and retrieve elements. Also known as a hash table or a dictionary.

The trade-off is that the elements are not ordered in a meaningful way, and that iterating through the collection may not be as fast as iterating through a List.

Also, you need to keep track of a "key" that maps to your object. This key must be unique.

(draw on board)

There are two main flavors, a HashMap and a TreeMap. The TreeMap guarantees that the keys are always sorted.

Maps

```
public class MapTest {
    public MapTest() {
     Map<String, Integer> grades = new HashMap<String, Integer>();

     grades.put("James", 93);
     grades.put("Javier", 99);
     grades.put("Miles", 95);

     Integer gradeForJavier = grades.get("Javier");
    }
}
```

Sets

A Set is an iterable collection with no positional indexing that contains unique elements. It is backed by a Map.

```
public class SetTest {
    public SetTest() {
     Set<String> people = new HashSet<String>();

     people.add("Mike");
     people.add("Gates");
     people.add("Xarene");

     people.add("Mike"); //warning! returns false because Mike is already in this Set!
     people.add(99); //error! This is Set of Strings!

     boolean gatesPresent = people.contains("Gates"); //returns true
     boolean gustavoPresent = people.contains("Gustavo"); //returns false
    }
}
```

Collections

It is easy to transform one collection into another.

example: Set to List

```
for (String person : people) { System.out.println(person) };
//not guarenteed to be in order!

Collections.sort(people); //error! can't sort a HashSet!

List<String> peopleList = new ArrayList<String>(people);
//now we have a List of people which we can sort

Collections.sort(peopleList);
for (String person : peopleList) { System.out.println(person) };
//okay this is in order!
```

Equality

see code example (SetTest.java)