Collections API

- The package `java.util` is often called the "Collections API"

- Extremely useful classes that you must understand to be a competent Java programmer

- Divided into three main types of collections: *lists*, *sets*, and *maps*
Collection Definitions

- **list** – an ordered collection of elements that can have duplicates

- **set** – an unordered collection of elements with no duplicates

- **map** – an association between keys and values. Maps each key to a single value.
Collections API organizes its classes into a hierarchy of interfaces
Abstract Class Hierarchy

- Abstract collection classes provide skeletal implementations of the interfaces.

- To implement your own collection, just subclass one of the abstract classes and implement the few abstract methods.

- No need to implement the entire interface from scratch.
Concrete Collection Classes

- The non-abstract (concrete) classes in the collections API provide good implementations of the interfaces.
- Use these unless you really need to implement your own.
- The most widely used are `ArrayList`, `LinkedList`, `HashSet`, and `HashMap`.
Lists

- **list** – an ordered collection of elements that can have duplicates

- The **List** interface specifies the behavior of a list collection.

- **API** provides two implementations of a list: **ArrayList** and **LinkedList**
Collections Quiz 1

1. What type of collection would I use to store the players on the Ghana football team? set
2. The schedule of the teams opponents for the Ghana football team? list
3. T/F. All classes in the Collections API implement Collection? false, maps do not
4. T/F. If a class does not implement Comparable the elements cannot be sorted with Collections.sort false, w/ Comparator
Collections Quiz 2

What's wrong with the equals method below?

class Person {
    private int age;
    private String name;

    public boolean equals(Object o) {
        if (!(o instanceof Person)) return false;
        Person p = (Person)o;
        return age == p.age && name.equals(p.name);
    }
}

name.equals(p.name)
The Equals Method: Pop Quiz

What will the following code print out?

```java
Double a1 = new Double(95.0);
Double a2 = new Double(95.0);

ArrayList grades = new ArrayList();
grades.add(a1);
System.out.println(grades.contains(a2));
```

`a1 != a2` but `true` because `a1.equals(a2)`
Two Forms of Equality

- **Reference Equality**
  - objects are the same exact object
  - test with `==` operator

- **Logical Equality**
  - objects are logically equivalent
  - test with the `equals` method
  - the object determines what it means for other objects to be logically equivalent to it
Equals Method

- Every class inherits an equals method from the class `Object`

- The `Object equals` method just tests for reference equality (==)

- For nearly all the classes you write, you should override the `.equals` method to provide a more useful test of logical equality
4 Steps to `.equals` Heaven

1. Use the `instanceof` keyword operator to check that the argument is the correct type. (This returns false if the argument is null.)

2. If it is not of the correct type, return false; otherwise cast the argument to that type.

3. Check equality of each primitive data field of the argument using `==`.

4. Check equality of each object data field of the argument using `.equals`. 
public class ColorPoint {
    private int x, y;
    private Color c;

    public boolean equals(Object o) {
        if (!(o instanceof ColorPoint)) {
            return false;
        }
        ColorPoint cp = (ColorPoint) o;
        return x == cp.x && y == cp.y &&
               c.equals(cp.c);
    }
}
Comparing Objects

- Sometimes we want to test not only equality, but whether one object is greater than or less than another object.

- If a class implements the `Comparable` interface, it can be compared in this way.

- Or if there is a `Comparator` which compares instances of that class.
The Comparable interface has only a single method `compareTo`

- If the object is greater than the argument, then `compareTo` returns a positive integer
- If the object is less than the argument, then `compareTo` returns a negative integer
- If the argument is equal, `compareTo` returns 0
The Comparator interface only has a single method `compareTo`

- If the first argument is greater than the second, `compareTo` returns a positive integer
- If the first argument is less than the second, `compareTo` returns a negative integer
- If the arguments are equal, `compareTo` returns 0
public class Person implements Comparable {
    private int age;
    private String name;

    // sorts people first by age, then by name
    public int compareTo(Object o) {
        Person p = (Person)o;
        if (age != p.age) return age - p.age;
        else return name.compareTo(p.name);
    }
}
public class PersonComparator

    implements Comparator {

    // sorts people first by age, then by name
    public int compareTo(Object o1, Object o2) {
        Person p1 = (Person)o1;
        Person p2 = (Person)o2;
        if (p1.age != p2.age) return p1.age - p2.age;
        else return p1.name.compareTo(p2.name);
    }
}
If an class implements Comparable or has a Comparator, Lists of that class can be sorted with Collections.sort

The class Collections provides a lot of useful static methods that operate on and return collections.

Look there before writing your own.