Objectives

- Collections
- Enumerated types

Collections

- Sometimes called containers
- Group multiple elements into a single unit
- Store, retrieve, manipulate, and communicate aggregate data
- Represent data items that form a natural group
  - Poker hand (a collection of cards)
  - Mail folder (a collection of messages)
  - Telephone directory (a mapping of names to phone numbers).
- Examples: HashMaps, Sets, Lists

Collections Framework

- Unified architecture for representing and manipulating collections
- More than arrays
  - More flexible, functionality, dynamic sizing
  - java.util

Collections Framework

- Interfaces
  - Abstract data types that represent collections
  - Collections can be manipulated independently of implementation
- Implementations
  - Concrete implementations of collection interfaces
  - Reusable data structures
- Algorithms
  - Methods that perform useful computations on collections, e.g., searching and sorting
  - Reusable functionality
  - Polymorphic: same method can be used on many different implementations of collection interface
**Implementation vs. Interface**

- Preferred Style:
  1. Choose an implementation
  2. Assign collection to variable of corresponding **interface** type

```
Interface variable = new Implementation();
```

- Also, methods should accept interfaces, not implementations

Why is this the preferred style?

**Example of the Way It Was**

- Before Java 1.5
- Doesn’t know what **type** of data is in the List

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
```

```
Integer x = (Integer) myIntList.get(0);
```

- Have to cast object we get out of list
- What if someone put in an object of wrong type previously?

**Generic Collection Interfaces**

- Added to 1.5
- Declaration of the Collection interface: **Type** parameter

```
public interface Collection<E> {
}
```

- When declare a Collection, specify **type** of object it contains

```
    Make sure put in, get out appropriate type
    Allows compiler to verify that object’s **type** is correct
    Reduces errors at runtime
```

- Example, a hand of cards:

```
List<Card> hand = new List<Card>();
```

**Comparable Interface**

- Also uses Generics

```
public interface Comparable<T> {

    int compareTo(T o)
}
```

The type it compares
Types Allowed with Generics

- Can only contain Objects, not primitive types
- Autoboxing and Autounboxing to the rescue!
  - Example: If collecting ints, use Integer

Comparing: Before & After Generics

- Before Generics
  ```java
  List myIntList = new LinkedList();
  myIntList.add(new Integer(0));
  Integer x = (Integer) myIntList.get(0);
  ```

- After Generics
  ```java
  List<Integer> myIntList = new LinkedList<Integer>();
  myIntList.add(new Integer(0));
  Integer x = myIntList.get(0);
  ```

  ✔ Improved readability and robustness

List Interface

- An ordered collection of elements
- Can contain duplicate elements
- Has control over where objects are stored in the list
  - `boolean add(<E> o)`
    - Boolean so that List can refuse some elements
    - e.g., refuse adding null elements
  - `<E> get(int index)`
    - Returns element at the position index
  - `int size()`
    - Returns the number of elements in the list
  - And more! (contains, remove, toArray, ...)

Differences from Python

- No shorthand
  - `list[pos]`

List Implementations

- `ArrayList`
  - Resizable array
  - Used most frequently
  - Fast
- `LinkedList`
  - Use if adding elements to beginning of list
  - Use if often delete from middle of list

cards.Deck.java
Sets

Set Interface
- No duplicate elements
  - Needs to determine if two elements are "logically" the same (equals method)
- Models mathematical set abstraction
  - boolean add(<E> o)
    - Add to set, only if not already present
  - int size()
    - Returns the number of elements in the list
- And more! (contains, remove, toArray, …)
  - Note: no get method -- get #3 from the set?

Select Set Implementations
- HashSet
  - Implements set using hash table
    - add, remove, and contains each execute in O(1) time
    - Used more frequently
    - Faster than TreeSet
    - No ordering
- TreeSet
  - Implements set using a tree
    - add, remove, and contains each execute in O (log n) time
    - Sorts

FindDuplicates Problem
- From the array of command-line arguments, identify the duplicates
  public static void main(String args[]) {
  }

FindDuplicates
  public static void main(String args[]) {
    Set<String> s = new HashSet<String>();
    for (String a : args) {
      if (!s.add(a)) {
        System.out.println("Duplicate detected: "+ a);
      }
    }
    System.out.println(s.size() + " distinct words detected: "+ s);
  }

Note how much code changes if s is a TreeSet