Structured Data in Java
The Collections Framework

Motivation

- Arrays can store a collection of data for us
  - But the size is fixed once the array is created
  - We had to manage where data was stored in the array
  - Not many built-in functions
  - Not very efficient for some applications

Java Collections Framework

- A Collection is a container that groups similar elements into a single entity
  - Examples would include a list of students, set of playing cards, group of name to phone numbers pairs

- The Collections Framework in Java offers a unified approach to store, retrieve and manipulate a group of data
  - This framework has a more intuitive approach when compared to complex frameworks in other languages (e.g. C++)

Benefits

- Provides many standard data structures, thus allowing the developer to concentrate more on functionality rather than the lower level details
- Size will grow or shrink automatically as needed
- Faster execution: several options available for choice of implementation. The framework improves the quality and performance of Java applications
- Aids in inter-operability between APIs since the collections framework is coded on the principles of “Code to Interface”
- Learning Curve: Easy to learn and start using Collections due to the “Code to Interface” principles

Benefits Summary

- Collections are one of the best-designed parts of Java, because:
  - They are *elegant:* they combine maximum power with maximum simplicity
  - They are *uniform:* when you know how to use one, you almost know how to use them all
  - They allow maximum *flexibility*: you can easily convert from one to another

Definitions

- Java defines a *collection* as “an object that represents a group of objects”

- Java defines a *collections framework* as “a unified architecture for representing and manipulating collections, allowing them to be manipulated independent of the details of their representation.”

Definitions

- An *interface* in the Java programming language is an abstract type that is used to specify a set of methods that classes must implement
  - It is *abstract* in the sense that it only defines method headers, but no method implementations

- A *generic type* is a class or interface that is parameterized over types
  - This allows the type of data that the container will hold to be specified later when we instantiate, or create, an actual container object

Why Interfaces & Generics

- Why interfaces:
  - If all the containers support the same methods (or interface) then you can later change the underlying container without having to change your code that uses that container

- Why generics:
  - Want to be able to store any type of information in a container, but we need to know what it being stored to insure type compatibility

Collections

- **collection**: an object that stores data; a.k.a. *data structure*
  - The objects stored are called *elements*
  - Some collections maintain an ordering; some allow duplicates
  - Typical operations: *add*, *remove*, *clear*, *contains* (search), *size*
  - Examples found in the Java class libraries:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`,
  - All collections are in the `java.util` package

```
import java.util.*;
```

Type Parameters (Generics)

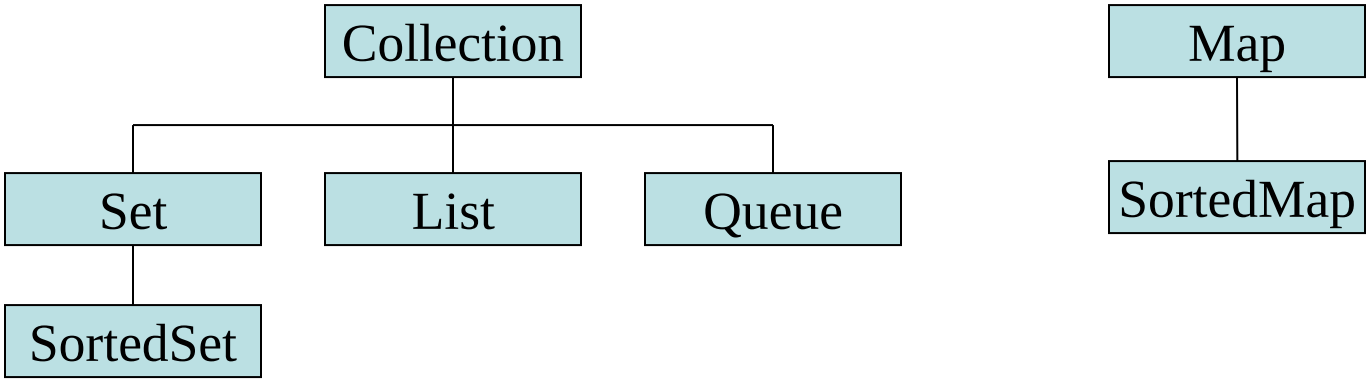
- When constructing a collection, you must specify the type of elements it will contain between < and >
  - This is called a *type parameter* or a *generic* class
  - Allows the same class to store elements of different types

Collection<**Type**> **name** = new **Collection**<**Type**>();

```
ArrayList<String> words = new ArrayList<String>();
words.add("Interface");
words.add("Generic");
```

Core Collections Framework

- The Collection framework forms a hierarchy:



The Collection Interface

- The Collection interface specifies (among other operations):
  - `boolean add(E o)`
  - `boolean contains(Object o)`
  - `boolean remove(Object o)`
  - `boolean isEmpty()`
  - `int size()`
  - `Iterator<E> iterator()`

- You should learn *all* the methods of the Collection interface--all are important

The Set Interface

- A Set is an *unordered* collection of elements that does not allow duplicates
  - Models the mathematical set abstraction

```
interface Set<E> extends Collection, Iterable
```

- Does not add any more methods to the Collection interface, but only redefines their behavior to prevent duplicates

The List Interface

- A List is an *ordered* sequence of elements

```
interface List<E> extends Collection, Iterable
```

- Some important List-specific methods are:
  - `void add(int index, E element)`
  - `E remove(int index)`
  - `E set(int index, E element)`
  - `E get(int index)`
  - `int indexOf(Object o)`
  - `int lastIndexOf(Object o)`
  - `ListIterator<E> listIterator()`
    - A ListIterator is like an Iterator, but has, in addition, hasPrevious and previous methods

The Map Interface

- A Map is a data structure for associating keys and values

`Interface Map<K,V>`

- The two most important methods are:
  - `V put(K key, V value)` // adds a key-value pair to the map
  - `V get(Object key)` // given a key, looks up the associated value
- Other useful methods are:
  - `Set<K> keySet()`
    - Returns a set view of the keys contained in this map
  - `Collection<V> values()`
    - Returns a collection view of the values contained in this map

- Next we need to discuss implementing these interfaces
  - That will be our next lecture