Generic Containers and Iterators in Java
Motivation

- containers are objects that store an arbitrary number of other objects
- these containers are manipulated by iterating over the contents
- virtually any non-trivial program will involve these two concepts
  - power of computers is in ability to quickly perform repetitive operations
Don't do everything from first principles

- if you find yourself writing code that manages the contents of an array or vector, performing inserts, deletes, etc, there's probably a container that already does what you're doing.
- arrays are relatively crude ways to store objects, only really useful for fixed sized groups of objects, without any properties like order or uniqueness
- using existing containers allows you to write faster, more correct code in less time
Containers

- **Collection**
  - a group of elements
  - often with additional constraints, like order or uniqueness
  - implements the `java.util.Collection` interface

- **Map**
  - a group of key-value pairs
  - also known as associative containers
  - implements the `java.util.Map` interface

- **Manage storage automatically**
Collections

- two dimensions, uniqueness of elements, and ordering of elements
  - ordered, non-unique: List
  - ordered, unique: *
  - unordered, non-unique: Multi-set, Bag
  - unordered, unique: Set
- the standard Java libraries do not include a multi-set or a unique-element list.
  - such collections do not conflict with the design however, one could write classes for these.
Collections

- the Collection interface defines all of the common operations you can perform on a group of elements

- all Collections support:
  - boolean contains( Object o )
  - Iterator iterator()
  - int size()  

- may also support:
  - boolean add( Object o )
  - boolean remove( Object o )
for any collection, you can define a “bigger than” method:

```java
public static boolean biggerThan( Collection lhs, Collection rhs ) {
    return lhs.size() > rhs.size();
}
```

as you can see, without iteration, we're pretty limited...
Iterators

- abstract the process of iteration
- advantageous because:
  - allows you to support many kinds of containers (even at run-time)
  - will often be more efficient than iterating over indices manually
  - exist as object separate from the container, so multiple iterations can be in progress at the same time
- replaces \texttt{Enumeration} from previous Java versions
iterator cont'd

- `java.util.Iterator` interface
  - `Object next()`: returns next element
  - `boolean hasNext()`: returns true if there are more elements
  - `void remove()`: if supported, removes the most recently accessed (via `next()`) element
- when created, the first call to `next()` will return the first object
Example

- generically define a “contains” method for collections

```java
// returns true if lhs contains all of the elements of rhs
public boolean contains( Collection lhs, Collection rhs ) {
    Iterator i = rhs.iterator();
    while ( i.hasNext() ) {
        if ( !lhs.contains( i.next() ) ) {
            return false;
        }
    }
    return true;
}
```
Ordered Collections

- if you care about the order that the elements are stored, use a List
- lists usually allow duplicate elements, so can be used in place of a multi-set
- refines add, to end of sequence
- refines remove, the first occurrence
- two lists are equal if they contain the same sequence of elements, compared using the elements' equals() method
  - thus you can compare different kinds of lists
ListIterator

- bidirectional, allow insertion and deletion
- **created by** `listIterator()` **method in List interface**
- `add(Object o)`: inserts `o` immediately before the next element
- `hasPrevious(), previous()`: analogous to `hasNext()` and `next()`, moving towards the front of the list
- `set(Object o)`: replaces the most recently returned element with `o`
List implementations

- **LinkedList**
  - good insert/delete performance
  - poor random access

- **ArrayList**
  - poor insert/delete (requires elements to shift)
  - good random access

- **Vector**
  - thread safe, but otherwise comparable to ArrayList
Unordered Collections

• if order is unimportant, use a Set
• Set also implies uniqueness of elements
  – a List can be used as a (less efficient) Set with duplicates in it
  – if you really need a proper multi-set, it would implement Collection
• refines `add` to refuse duplicates
Uniqueness and Equality

- to determine whether or not an element is already in the Set, the `equals()` method is used
- on the surface, this is straightforward, BUT...
- if the objects in the Set are mutable, the result of `equals()` must not change after they have been added to the set
- this can also work against you in the opposite direction
  - e.g. two Vectors are equal if they have the same state, i.e. for all i, `v1.get(i).equals(v2.get(i))`
  - as a consequence, you can't insert two empty Vectors into a Set!
Example

- can't insert v1 and v2 into s, even though they are different objects

```java
Set s = new HashSet();
Vector v1 = new Vector();
Vector v2 = new Vector();
s.insert( v1 );
s.insert( v2 ); // does nothing
v1.add( "something" );
if ( s.contains( v2 ) ) // false!
```
Example (cont'd)

• a solution, use a wrapper object that defines `equals` in terms of references:

```java
public class Wrapper {
    private Object wrapped;

    public Wrapper( Object o ) {
        wrapped = o;
    }

    public Object get() {
        return wrapped;
    }

    public boolean equals( Object o ) {
        if ( ! (o instanceof Wrapper) ) return false;
        return ( wrapped == ((Wrapper)o).wrapped );
    }
}
```

Set `s` = new HashSet();
Vector `v1` = new Vector();
Vector `v2` = new Vector();
s.insert( new Wrapper(v1) );
s.insert( new Wrapper(v2) );
v1.add( "something" );
if ( s.contains( new Wrapper(v2) ) ) // true!
```
Set implementations

- **HashSet**
  - `constant time` `add()`, `remove()`, `contains()`
  - iterator order unknown, may even change as contents change

- **TreeSet**, implements **OrderedSet**
  - elements are sorted (sequence not preserved though)
  - `O(logN)` `add()`, `remove()`, `contains()`
Comparator/Comparable

• you can define the order that elements are sorted in using two approaches:

• have the elements implement the Comparable interface
  
  - public int compareTo( Object rhs )
  
  - returns -1 if this < rhs, 0 if this is equal to rhs, and 1 if this > rhs
  
  - throws an exception if rhs is wrong type

• supply a Comparator object to the container
  
  - public int compare( Object lhs, Object rhs )
  
  - analogous semantics as compareTo()
  
  - Comparator is more flexible, since it can be chosen at run-time
Associative containers

- Map interface, not related to Collection
- defines key-value pairs
- a generalization of containers which can be accessed by index, keys can be arbitrary objects
- Collection values()
- Set keySet()
Map Example

Map m = new HashMap();
m.put("spot", new Dog("brown", "shaggy");
m.put("rover", new Dog("black", "short-haired");
System.out.println(m);

Dog d = m.get("rover");
System.out.println(d);

OUTPUT
{ spot=brown and shaggy dog, rover=black and short-haired dog }
black and short-haired dog
Map Implementations

- HashMap
- HashTable – old version of HashMap, thread safe
- WeakHashMap – values may be garbage collected if there are no external references to them
- TreeMap – slower for all operations $O(2\log N)$, but can provide sorted contents at no extra cost
Choosing a container

- identify the abstract properties you require:
  - ordered/unordered?
  - look-up by key?
  - duplicate elements allowed?
  - store sorted?
- this will pick the interface for you:
  - one of Collection, Set, List, Map, SortedMap, SortedSet
- pick an implementation, based on expected usage in the program
- if you get the interface right, you can easily change implementations if your performance needs turn out differently than expected (which they often do)