Topic Coverage for 9/28/2000

1. What is a framework and why bother?
2. Introduction to the Collections and Map Frameworks
3. Iterators and Hash Tables
4. Structure of the Collections Framework
5. Structure of the Map Framework

What is a Framework?

1. A set of interfaces and abstract classes which define a commonly used set of facilities and services.

2. A set of classes implementing the most used subset of facilities and services.

How is a Framework used?

1. The methods of the implementing classes are invoked by applications to utilize specific services.

2. The abstract classes are used as foundations and building blocks for extended and specialized implementations of the interfaces.

The Collections Framework

1. The basic requirement of any application program is storing, structuring and retrieving instances of objects.

2. The Collections framework provides modes of storing, structuring and retrieving of object instances . The Collections framework implements a family of containers for holding instances of objects.

3. Java provides one language construct, arrays, for implementation of storage, structuring and accessing of object instances (elements). Why do we need the Collections framework.

* An array can store only a single element type.
* An array is static. The number of instances to be stored must be specified in advance.
* An array has a fixed addressing pattern.
* Traversal must be procedurally programmed.

But

* Storage and retrieval from arrays is very efficient.
* Index based addressing is an appropriate abstraction for regular structures.

4. The basic concepts (abstractions) for storage and structuring of object instances include:

Bag (multi-sets) - A bag is an unordered collection of elements which may include duplicate elements. (Duplicate has a very specific meaning- the equals() method defined in the Comparator interface.)

Set - A set is an unordered collection of elements. Duplicate elements are not allowed.

List - A list is an ordered collection of elements.

Map - A map is an unordered collection of key-value pairs. The keys in a map must be unique.

5. Auxiliary Abstractions for Storage and Structuring

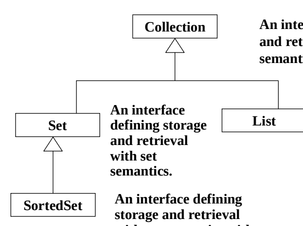
SortedSets - Sets where an ordering relation is imposed when elements are added to the set.

Sorted Maps - Maps where an ordering relation based on the keys is imposed when elements are added to the set.

6. Arrays and Vectors

Array - An array is a static collection where the elements are ordered by position in an index space. (An index space is a regular lattice of points in an N-dimensional space. We will return to this later.) Java also provides a class Arrays which implements services such as sorts on arrays.

Vector - The Java Vector class implements a hybrid abstraction with properties of arrays and lists. Vectors implement the List interface.



An interface defining storage and retrieval with bag semantics.

An interface defining storage and retrieval with set semantics.

The list interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Lists typically allow duplicate elements

An interface defining storage and retrieval with set semantics with the added specification that the iterator will traverse the set in ascending element order, sorted according to the *natural ordering* of its elements

public interface Collection

The root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate themwhere maximum generality is desired.

Bags or multisets (unordered collections that may contain duplicate elements) should implement this interface directly.

All general-purpose Collection implementation classes (which typically implement Collection indirectly throughone of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type Collection, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose Collection implementations in the JDK comply.

Method Summary for Interface Collection

```
boolean add(Object o)
Ensures that this collection contains the specified
element (optional operation).

boolean addAll(Collection c)
Adds all of the elements in the specified collection to
this collection (optional operation).

void clear()
Removes all of the elements from this collection
(optional operation).

boolean contains(Object o)
Returns true if this collection contains the specified
element.

boolean containsAll(Collection c)
Returns true if this collection contains all of the
elements in the specified collection.
```


Method Summary for Interface Collection - Continued

```
boolean equals(Object o)
Compares the specified object with this collection for
equality.

int hashCode()
Returns the hash code value for this collection.
boolean isEmpty()
Returns true if this collection contains no elements.

 Iterator iterator()
Returns an iterator over the elements in this
collection.

boolean remove(Object o)
Removes a single instance of the specified element
from this collection, if it is present (optional
operation).

boolean removeAll(Collection c)
Removes all this collection's elements that are also
contained in the specified collection (optional
operation).
```


Method Summary for Interface Collection - Continued

```
boolean retainAll(Collection c)
Retains only the elements in this collection that are
contained in the specified collection (optional
operation).

int size()
Returns the number of elements in this collection.

 Object[] toArray()
Returns an array containing all of the elements in this
collection. Object[] toArray(Object[] a)
Returns an array containing all of the elements in this
collection whose runtime type is that of the specified
array.
```


public interface List extends Collection

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The List interface places additional stipulations, beyond those specified in the Collection interface, on the contracts of the iterator, add, remove, equals, and hashCode methods. Declarations for other inherited methods are also included here for convenience.

The List interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the LinkedList class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

The List interface provides a special iterator, called a ListIterator, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the Iterator interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The List interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

The List interface provides two methods to efficiently insert and remove multiple elements at an arbitrary point in the list.

Method Summary for Interface List

```
void   add(int index, Object element)
Inserts the specified element at the specified position
in this list (optional operation).

boolean      add(Object o)
Appends the specified element to the end of this list
(optional operation).

boolean      addAll(Collection c)
Appends all of the elements in the specified collection
to the end of this list, in the order that they are
returned by the specified collection's iterator
(optional operation).

boolean      addAll(int index, Collection c)
Inserts all of the elements in the specified collection
into this list at the specified position (optional
operation).
```


Method Summary for Interface List - Continued

```
void   clear()
Removes all of the elements from this list (optional
operation).

boolean      contains(Object o)
Returns true if this list contains the specified
element.

boolean      containsAll(Collection c)
Returns true if this list contains all of the elements
of the specified collection.

boolean      equals(Object o)
Compares the specified object with this list for
equality.

Object get(int index)
Returns the element at the specified position in this
list.
```


Method Summary for Interface List - Continued

```
int    hashCode()
Returns the hash code value for this list.

int    indexOf(Object o)
Returns the index in this list of the first occurrence
of the specified element, or -1 if this list does not
contain this element.

boolean      isEmpty()
Returns true if this list contains no elements.

Iterator     iterator()
Returns an iterator over the elements in this list in
proper sequence.

int    lastIndexOf(Object o)
Returns the index in this list of the last occurrence of
the specified element, or -1 if this list does not
contain this element.
```


Method Summary for Interface List - Continued

```
ListIterator  listIterator()
Returns a list iterator of the elements in this list
(in proper sequence).

ListIterator  listIterator(int index)
Returns a list iterator of the elements in this list
(in proper sequence), starting at the specified
position in this list.

Object remove(int index)
Removes the element at the specified position in this
list (optional operation).

boolean      remove(Object o)
Removes the first occurrence in this list of the
specified element (optional operation).

boolean      removeAll(Collection c)
Removes from this list all the elements that are
contained in the specified collection (optional
operation).
```


Method Summary for Interface List - Continued

```
boolean      retainAll(Collection c)
Retains only the elements in this list that are
contained in the specified collection (optional
operation).

Object set(int index, Object element)
Replaces the element at the specified position in this
list with the specified element (optional operation).

int    size()
Returns the number of elements in this list.

List   subList(int fromIndex, int toIndex)
Returns a view of the portion of this list between the
specified fromIndex, inclusive, and toIndex, exclusive.
```


Method Summary for Interface List

```
Object[]     toArray()
Returns an array containing all of the elements in
this list in proper sequence.

Object[]     toArray(Object[] a)
Returns an array containing all of the elements in
this list in proper sequence; the runtime type of the
returned array is that of the specified array.
```

public interface Set extends Collection

A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.

The Set interface places additional stipulations, beyond those inherited from the Collection interface, on the contracts of all constructors and on the contracts of the add, equals and hashCode methods. Declarations for other inherited methods are also included here for convenience. (The specifications accompanying these declarations have been tailored to the Set interface, but they do not contain any additional stipulations.)

The additional stipulation on constructors is, not surprisingly, that all constructors must create a set that contains no duplicate elements (as defined above).

Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

Method Summary for Interface Set

```
boolean      add(Object o)
Adds the specified element to this set if it is not
already present (optional operation).

boolean      addAll(Collection c)
Adds all of the elements in the specified collection to
this set if they're not already present (optional
operation).

void   clear()
Removes all of the elements from this set (optional
operation).

boolean      contains(Object o)
Returns true if this set contains the specified element.

boolean      containsAll(Collection c)
Returns true if this set contains all of the elements of
the specified collection.
```


Method Summary for Interface Set - Continued

```
boolean      equals(Object o)
Compares the specified object with this set for
equality.

int    hashCode()
Returns the hash code value for this set.

boolean      isEmpty()
Returns true if this set contains no elements.

Iterator     iterator()
Returns an iterator over the elements in this set.

boolean      remove(Object o)
Removes the specified element from this set if it is
present (optional operation).

boolean      removeAll(Collection c)
Removes from this set all of its elements that are
contained in the specified collection (optional
operation).
```


Method Summary for Interface Set - Continued

```
boolean      retainAll(Collection c)
Retains only the elements in this set that are
contained in the specified collection (optional
operation).

int    size()
Returns the number of elements in this set (its
cardinality).

Object[]     toArray()
Returns an array containing all of the elements in
this set.

Object[]     toArray(Object[] a)
Returns an array containing all of the elements in
this set whose runtime type is that of the specified
array.
```


Method Comparisons - add

```
Collection

boolean add(Object o)
Ensures that this collection contains the specified
element (optional operation).

List

void   add(int index, Object element)
Inserts the specified element at the specified position
in this list (optional operation).

boolean      add(Object o)
Appends the specified element to the end of this list
(optional operation).

Set

boolean      add(Object o)
Adds the specified element to this set if it is not
already present (optional operation).
```


public interface Iterator

An iterator over a collection. Iterator takes the place of Enumeration in the Java collections framework. Iterators differ from enumerations in two ways:

Method Summary

```
boolean hasNext()
Returns true if the iteration has more elements.

Object next()
Returns the next element in the interation.

void remove()
Removes from the underlying collection the last
element returned by the iterator (optional
operation).
```

public interface ListIterator
extends Iterator

An iterator for lists that allows the programmer to traverse the list in either direction and modify the list during iteration.

Method Summary for Interface ListIterator

```
void   add(Object o)
Inserts the specified element into the list (optional
operation).

boolean      hasNext()
Returns true if this list iterator has more elements
when traversing the list in the forward direction.

boolean      hasPrevious()
Returns true if this list iterator has more elements
when traversing the list in the reverse direction.

Object next()
Returns the next element in the list.
```


Method Summary for Interface ListIterator- Continued

```
int    nextIndex()
Returns the index of the element that would be
returned by a subsequent call to next.

Object previous()
Returns the previous element in the list.

int    previousIndex()
Returns the index of the element that would be
returned by a subsequent call to previous.

void   remove()
Removes from the list the last element that was
returned by next or previous (optional operation).

void   set(Object o)
Replaces the last element returned by next or
previous with the specified element (optional
operation).
```


Method Comparison - iterator

Collection

```
Iterator iterator()
Returns an iterator over the elements in this
collection.
```

List

```
Iterator     iterator()
Returns an iterator over the elements in this list in
proper sequence.
ListIterator  listIterator()
Returns a list iterator of the elements in this list
(in proper sequence).
ListIterator  listIterator(int index)
Returns a list iterator of the elements in this list
(in proper sequence), starting at the specified
position in this list.
```

Set

```
Iterator     iterator()
Returns an iterator over the elements in this set.
```


What do we store and retrieve?

We have mostly illustrated storage and retrieval with simple entities such as ints or strings. Then the value of an object and its unique identifier are one and the same. But in practice what is stored and retrieved are objects such as personnel records, descriptions of regions in space, items for sale in a catalog, forms defining a sale from a catalog, dictionaries, etc. We typically use some subset of the values of an object as a *key*.

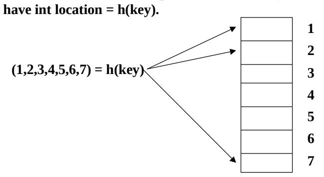
Key - A value or values (an object type) which can be used to locate the storage location of another object or collection of values.
1. The unique identifier of an object may be used as its key or a key may be derived from the unique identifier.
2. We speak of "searching for a key." to find the location of the object to which it refers.
3. But the index of a location in an array or vector is a location which can be computed.

Hash Tables

Imagine that instead of searching though a tree of values there existed a function h(key) which when executed returned the location of the desired object.

Location = h(key)

Let the locations of values be positions in an array or vector. Then we have int location = h(key).



Now let us look at typical keys and see how we might define a function h(key) which will take a value for a key of some type and return an integer which is a location in an array.

Definition - The domain of a function is the set of values the function takes as input and the range of a function is the set of values the function returns as outputs.

Let us look at last names as keys:
1. Assume no last name has more than 10 characters.
2. Assume each of the 26 letters of the English alphabet occur with equal probability in each of the possible 10 positions in a name.

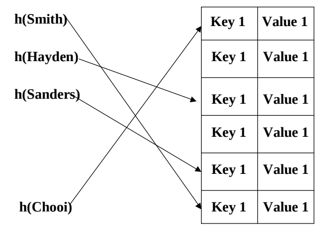
Number of names = 26*26* …= 2610 !!!**

Clearly we cannot have an array of size 2610 so we need to define h so that it has a range of the order of the number of names we want to store .**

Each alphabetic character has a numeric code representation. We can create a numeric representation of the name by concatenating the the numeric codes for each character. That still leaves us with 26**10 names but there are many possible transformations on numbers which cut down the range of a set of numbers.

1. Let N be the number of slots in the array or vector.
2. If we take the remainder of division of the number we get by concatenating the numeric representations of the characters in the name we will get a number in the range [0 to N]. h(key) = key%N => (0….N)

But this definition of h leads may generate the same value for several different names. There are many ways to deal with this non-uniqueness. We will show one solution where each entry in the array or vector is an instance of the object type List. If more than one key hashes to the same entry in the array we enter it into the list at that entry of the array. This solution will work well so long as the number of keys hashed to a given entry is much smaller than the size of the array.



What are we giving up to get this performance - Ordered storage - the ability to traverse in a meaning order and write reports.

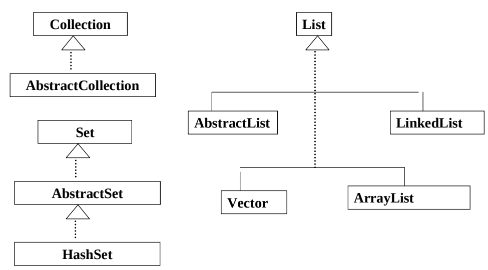
But there are methods which produce ordered storage in particular cases.

Performance of hash table methods

best	average	worst
O(1)	O(1)	O(1)


```
import java.util.*;
import java.io.*;
public class CountWords {
  static public void main(String[] args) {
    HashSet words = new HashSet();
    BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
    String delim = " \t\n.,:;?!-/()[]\"\'";
    String line;
    int count = 0;
    try {
      while ((line = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(line, delim);
        while (st.hasMoreTokens()) {
          count++;
          words.add(st.nextToken().toLowerCase());
        }
      }
    } catch (IOException e) {}
    System.out.println("Total number of words: " + count);
    System.out.println("Number of different words: " + words.size());
  }
}
```


Java.util Implementations of these Interfaces



public abstract class AbstractCollection extends Object implements Collection

This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.

To implement an unmodifiable collection, the programmer needs only to extend this class and provide implementations for the iterator and size methods. (The iterator returned by the iterator method must implement hasNext and next.)

To implement a modifiable collection, the programmer must additionally override this class's add method (which otherwise throws an UnsupportedOperationException), and the iterator returned by the iterator method must additionally implement its remove method.

The programmer should generally provide a void (no argument) and Collection constructor, as per the recommendation in the Collection interface specification.

The documentation for each non-abstract methods in this class describes its implementation in detail. Each of these methods may be overridden if the collection being implemented admits a more efficient implementation.

Method Summary for Abstract Collection

`boolean add(Object o)`
Ensures that this collection contains the specified element (optional operation).

`boolean addAll(Collection c)`
Adds all of the elements in the specified collection to this collection (optional operation).

`void clear()`
Removes all of the elements from this collection (optional operation).

`boolean contains(Object o)`
Returns true if this collection contains the specified element.

`boolean containsAll(Collection c)`
Returns true if this collection contains all of the elements in the specified collection.

Method Summary for Abstract Collection - Continued

boolean isEmpty()
Returns true if this collection contains no elements.
abstract Iterator iterator()
Returns an iterator over the elements contained in this collection.

boolean remove(Object o)
Removes a single instance of the specified element from this collection, if it is present (optional operation). boolean removeAll(Collection c)
Removes from this collection all of its elements that are contained in the specified collection (optional operation).
boolean retainAll(Collection c)
Retains only the elements in this collection that are contained in the specified collection (optional operation).
abstract int size()
Returns the number of elements in this collection.

Method Summary for Abstract Collection - Continued

Object[] toArray()
Returns an array containing all of the elements in this collection.

Object[] toArray(Object[] a)
Returns an array with a runtime type is that of the specified array and that contains all of the elements in this collection.

String toString()
Returns a string representation of this collection.

public abstract class AbstractSet
extends AbstractCollection
implements Set

This class provides a skeletal implementation of the Set interface to minimize the effort required to implement this interface.

The process of implementing a set by extending this class is identical to that of implementing a Collection by extending AbstractCollection, except that all of the methods and constructors in subclasses of this class must obey the additional constraints imposed by the Set interface (for instance, the add method must not permit addition of multiple intances of an object to a set).

Note that this class does not override any of the implementations from the AbstractCollection class. It merely adds implementations for equals and hashCode.

Method Summary for AbstractSet

boolean equals(Object o)
Compares the specified object with this set for equality.

int hashCode()
Returns the hash code value for this set.

public class HashSet
extends AbstractSet
implements Set, Cloneable, Serializable

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the intial capacity too high (or the load factor too low) if iteration performance is important.

Note that this implementation is not synchronized. If multiple threads access a set concurrently, and at least one of the threads modifies the set, it *must* be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the HashSet instance:

```
     Set s = Collections.synchronizedSet(new
HashSet(...));
```

The iterators returned by this class's iterator method are *fail-fast*: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the Iterator throws a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Frameworks - Collections Framework

Method Summary for HashSet

```
boolean      add(Object o)
Adds the specified element to this set if it is not
already present.
void   clear()
Removes all of the elements from this set.
Object clone()
Returns a shallow copy of this HashSet instance: the
elements themselves are not cloned.
boolean      contains(Object o)
Returns true if this set contains the specified element.
boolean      isEmpty()
Returns true if this set contains no elements.
Iterator     iterator()
Returns an iterator over the elements in this set.
boolean      remove(Object o)
Removes the given element from this set if it is present.
int    size()
Returns the number of elements in this set (its
cardinality).
```


Frameworks - Collections Framework

public abstract class AbstractList extends AbstractCollection implements List

This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array). For sequential access data (such as a linked list), AbstractSequentialList should be used in preference to this class.

To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the get(int index) and size() methods.

To implement a modifiable list, the programmer must additionally override the set(int index, Object element) method (which otherwise throws an UnsupportedOperationException. If the list is variable-size the programmer must additionally override the add(int index, Object element) and remove(int index) methods.

The programmer should generally provide a void (no argument) and collection constructor, as per the recommendation in the Collection interface specification.

Unlike the other abstract collection implementations, the programmer does not have to provide an iterator implementation; the iterator and list iterator are implemented by this class, on top the "random access" methods:
get(int index), set(int index, Object element), set(int index, Object element), add(int index, Object element) and remove(int index).

Frameworks - Collections Framework

Method Summary for AbstractList

```
void add(int index, Object element)
Inserts the specified element at the specified position in
this list (optional operation).

boolean add(Object o)
Appends the specified element to the end of this List
(optional operation).

boolean addAll(int index, Collection c)
Inserts all of the elements in the specified collection into
this list at the specified position (optional operation).

void clear()
Removes all of the elements from this collection (optional
operation).

boolean equals(Object o)
Compares the specified object with this list for equality.

abstract Object get(int index)
Returns the element at the specified position in this list.
```


Frameworks - Collections Framework

Method Summary for AbstractList

```
int hashCode()
Returns the hash code value for this list. int indexOf(Object
o)
Returns the index in this list of the first occurence of the
specified element, or -1 if the list does not contain this
element.

Iterator iterator()
Returns an iterator over the elements in this list in proper
sequence.

int lastIndexOf(Object o)
Returns the index in this list of the last occurence of the
specified element, or -1 if the list does not contain this
element.

ListIterator listIterator()
Returns an iterator of the elements in this list (in proper
sequence).
```


Frameworks - Collections Framework

Method Summary for AbstractList - Continued

```
ListIterator listIterator(int index)
Returns a list iterator of the elements in this list (in
proper sequence), starting at the specified position in the
list.

Object remove(int index)
Removes the element at the specified position in this list
(optional operation).

protected void removeRange(int fromIndex, int toIndex)
Removes from this list all of the elements whose index is
between fromIndex, inclusive, and toIndex, exclusive.

Object set(int index, Object element)
Replaces the element at the specified position in this list
with the specified element (optional operation).

List subList(int fromIndex, int toIndex)
Returns a view of the portion of this list between
fromIndex, inclusive, and toIndex, exclusive.
```


Frameworks - Collections Framework

```
java.util class Stack
java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractList
              |
              +--java.util.Vector
                    |
                    +--java.util.Stack
```

public class Stack extends Vector

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

boolean
empty()
Tests if this stack is empty.
Object
peek()
Looks at the object at the top of this stack without removing it from the stack.
Object
pop()
Removes the object at the top of this stack and returns that object as the value of this function.
Object
push(Object item)
Pushes an item onto the top of this stack.
int
search(Object o)
Returns the 1-based position where an object is on this stack.

Application of a Stack - Maze Traversal

Maze - Structure of partially connected cells with Start cell and End cell connected by at least one path. Each cell has k neighboring cells.

Maze Traversal -

1. Enter Start cell
2. Choose neighboring cell to enter.
3. Enter neighboring cell.
4. Continue to enter neighboring cells until a cell with no exits is encountered or until the End cell is encountered.
5. If a cell with no exits is encountered, go back one cell and take a different exit until either all exits have been taken and the End cell not reached or the End cell is reached.
6. Repeat 5 until either you have returned to the Start cell and exhausted all exits or you have reached the End cell.

Stack automates path reversal.