Tutorial 3

Programming II

Exercise 1

Implement a class String stack to replace class uk.ac.ic.doc.su2.utils.StringStack that was used in the Tutorial 1.

- StringStack(): Void // Constructor. Builds an empty stack
- push(String s): Void // Adds s to the top of the stack
- pop(): String // If the stack is not empty, removes and returns the string on the top of the stack.
- isEmpty():boolean // Returns true if the stack is empty, otherwise returns false

Use class java.utils.Vector (See appendix for Java Documentation for the class):

- *Use the following methods:*
  - *void add(int index, Object element)*
  - *Object remove(int index)*
  - *int size()*
- *Make sure you perform type castings where necessary.*

Exercise 2

Part 1

Write a class Dictionary as a subclass of class Book given below. Dictionary should provide a constructor Dictionary(String ISDN, String language1, String language2) that constructs the Dictionary and states what languages it can translate to and from (e.g. a Spanish-English dictionary) and a public method void setDefinitions(int def) that sets the number of definitions that the dictionary has. Dictionary should also reimplement printAllDetails() to print all the object's details.

```
package bookStore;

public class Book {
	protected int pages = 0;
	protected String title = "";
	protected String ISDN;
	protected String publisher = "";
	protected int year = 0;

	public Book(String ISDN) {
		this.ISDN = ISDN;
	}

	public void setTitle(String title) {
		this.title= title;
	}

	public void setPublicationDetails(String publisher, int year) {
```

```
            this.publisher = publisher;
            this.year = year;
      }

      public void setPages(int p) {
            pages = p;
      }

      public int getPages() {
            return pages;
      }

      public void printAllDetails() {
            System.out.print(title + ", " + publisher + "(" + year + "). ");
            System.out.println("ISDN:" + ISDN + " pages: " + pages);
      }

      public void printBasicDetails() {
            if (title.equals(""))
                  System.out.println("ISDN:" + ISDN + " Title: " + title);
            else
                  System.out.println("ISDN:" + ISDN);
      }
}
```

Part 2

Write a class Bookshelf that can store books. Use class java.utils.Vector for your class. Bookshelf should provide the following methods:

- int size();
  - Returns the number of books on the bookshelf.
- void addBookOnLeftSide(Book b);
  - Adds book on the left side of the shelf. Shifts all books one position to the right.
- void addBookOnRightSide(Book b);
  - Adds book on the right side of the shelf. Shifts all books to the left.
- void addBook(int i, Book b);
  - Adds book in position i counting from the left. Hence, addBook(n, b) with n<= 1 is equivalent to addBookOnLeftSide(b) and addBook(n, b) with n>=size() + 1 is equivalent to addBookOnRightSide(b).
- Book remove(int i)
  - Removes book from position i, shifts all books with positions greater than i to the left (i.e. one position less).
- void printLR();
  - Prints all details of books from left to right.
- void printRL();
  - Prints all details of books from right to left.

You will probably find the following method of class Vector useful:
void insertElementAt(Object obj, int index)

Part 3

Write a program that creates 6 books (2 dictionaries and 4 non-dictionaries), adds them to a bookshelf, then randomly picks books out and adds them back onto the left of the bookshelf, and finally prints the contents of the bookshelf from left to right.

Part 4

Write a method that will take a bookshelf as a parameter and reorder it so that all dictionaries are on the right.

Part 5

Write a class SalesItem that could be used by a bookstore to handle its stock system. The class should support at least the following methods:

- SalesItem(Book book);
- void setPrice(int pricePerPage); //sets the price of the book according to its length.
- in setPrice(); // returns the price of the book (not the price per page)
- void newCopiesArrived(int n) {
- void soldCopy()
- int getAvailableCopies() {
- void print()

Exercise 3

Part 1

Write a class Rectangle with the following methods:

- void Rectangle(int height, int width);
- void setHeight(int newHeight);
- void setWidth(int newWidth);
- int getHeight();
- int getWidth();
- int getArea();

Part 2

Write a class Square that is a subclass of Rectangle

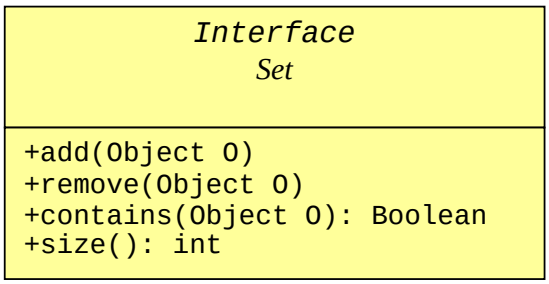
Part 3

- Why does the IS A relation not hold between Square and Rectangle?
- Given a sequence of method calls on an object r of apparent type Rectangle that would produce unexpected results if the actual type of r is Square.

Exercise 4

Part 1

Write an interface for sets as described below:



Part 2

Assume you have two classes that implement the interface Set: MemoryEfficientSet and SpeedEfficientSet. The first has been optimised to use as least memory as possible when storing objects at the cost of slower speed in adding and removing objects from the set. The second has been optimised for fast addition and removal at the cost of memory consumption.
Write a method getNames(int n) that inputs from the user n names and returns a set containing them. If n >10, then the method should return a MemoryEfficientSet, otherwise it should return a SpaceEfficientSet.

Exercise 5

Part 1

Write an abstract class called AbstractStack, that implements the following methods.

- Stack(List l): Void // Constructor. Builds an empty stack
- push(Object O): Void // Adds s to the top of the stack
- pop(): Object // If the stack is not empty, removes and returns the object on the top of the stack.
- isEmpty():boolean // Returns true if the stack is empty, otherwise returns false

List is an interface that provides the following methods

- *void add(int index, Object element)*
- *Object remove(int index)*
- *int size()*

Part 2

Write a concrete class VectorStack, that is a subclass of AbstractStack. VectorStack should use an object of class java.utils.Vector (that implements interface List) to store objects.

java.util

Class Vector

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

All Implemented Interfaces:

Cloneable, Collection, List, RandomAccess, Serializable

Direct Known Subclasses:

Stack

---

public class **Vector**
extends AbstractList
implements List, RandomAccess, Cloneable, Serializable

The `Vector` class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a `Vector` can grow or shrink as needed to accommodate adding and removing items after the `Vector` has been created.

Each vector tries to optimize storage management by maintaining a `capacity` and a `capacityIncrement`. The `capacity` is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of `capacityIncrement`. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation.

As of the Java 2 platform v1.2, this class has been retrofitted to implement List, so that it becomes a part of Java's collection framework. Unlike the new collection implementations, Vector is synchronized.

The Iterators returned by Vector's iterator and listIterator methods are *fail-fast*: if the Vector is structurally modified at any time after the Iterator is created, in any way except through the Iterator's own remove or add methods, the Iterator will throw 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. The Enumerations returned by Vector's elements method are *not* fail-fast.

Since:

JDK1.0

See Also:

`Collection`, `List`, `ArrayList`, `LinkedList`, Serialized Form

Field Summary

protected int	**capacityIncrement** The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.
protected int	**elementCount** The number of valid components in this `Vector` object.
protected Object[]	**elementData** The array buffer into which the components of the vector are stored.

Fields inherited from class java.util.AbstractList

`modCount`

Constructor Summary

`Vector()`
Constructs an empty vector so that its internal data array has size `10` and its standard capacity increment is zero.

`Vector(Collection c)`
Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.

`Vector(int initialCapacity)`
Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.

`Vector(int initialCapacity, int capacityIncrement)`
Constructs an empty vector with the specified initial capacity and capacity increment.

Method Summary

void	**`add(int index, Object element)`** Inserts the specified element at the specified position in this Vector.
boolean	**`add(Object o)`** Appends the specified element to the end of this Vector.
boolean	**`addAll(Collection c)`** Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
boolean	**`addAll(int index, Collection c)`** Inserts all of the elements in in the specified Collection into this Vector at the specified position.
void	**`addElement(Object obj)`** Adds the specified component to the end of this vector, increasing its size by one.
int	**`capacity()`** Returns the current capacity of this vector.
void	**`clear()`** Removes all of the elements from this Vector.
Object	**`clone()`** Returns a clone of this vector.

boolean	**contains**(Object elem) Tests if the specified object is a component in this vector.
boolean	**containsAll**(Collection c) Returns true if this Vector contains all of the elements in the specified Collection.
void	**copyInto**(Object[] anArray) Copies the components of this vector into the specified array.
Object	**elementAt**(int index) Returns the component at the specified index.
Enumeration	**elements**() Returns an enumeration of the components of this vector.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
boolean	**equals**(Object o) Compares the specified Object with this Vector for equality.
Object	**firstElement**() Returns the first component (the item at index 0) of this vector.
Object	**get**(int index) Returns the element at the specified position in this Vector.
int	**hashCode**() Returns the hash code value for this Vector.
int	**indexOf**(Object elem) Searches for the first occurence of the given argument, testing for equality using the equals method.
int	**indexOf**(Object elem, int index) Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
void	**insertElementAt**(Object obj, int index) Inserts the specified object as a component in this vector at the specified index.
boolean	**isEmpty**() Tests if this vector has no components.
Object	**lastElement**() Returns the last component of the vector.
int	**lastIndexOf**(Object elem) Returns the index of the last occurrence of the specified object in this vector.
int	**lastIndexOf**(Object elem, int index) Searches backwards for the specified object, starting from the specified index, and returns an index to it.
Object	**remove**(int index) Removes the element at the specified position in this Vector.
boolean	**remove**(Object o) Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
boolean	**removeAll**(Collection c) Removes from this Vector all of its elements that are contained in the specified

	Collection.
void	**removeAllElements**() Removes all components from this vector and sets its size to zero.
boolean	**removeElement**(Object obj) Removes the first (lowest-indexed) occurrence of the argument from this vector.
void	**removeElementAt**(int index) Deletes the component at the specified index.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
boolean	**retainAll**(Collection c) Retains only the elements in this Vector that are contained in the specified Collection.
Object	**set**(int index, Object element) Replaces the element at the specified position in this Vector with the specified element.
void	**setElementAt**(Object obj, int index) Sets the component at the specified index of this vector to be the specified object.
void	**setSize**(int newSize) Sets the size of this vector.
int	**size**() Returns the number of components in this vector.
List	**subList**(int fromIndex, int toIndex) Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
Object[]	**toArray**() Returns an array containing all of the elements in this Vector in the correct order.
Object[]	**toArray**(Object[] a) Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
String	**toString**() Returns a string representation of this Vector, containing the String representation of each element.
void	**trimToSize**() Trims the capacity of this vector to be the vector's current size.

Methods inherited from class java.util.AbstractList
iterator, listIterator, listIterator

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.List
iterator, listIterator, listIterator

Field Detail

elementData

```
protected Object[] elementData
```

The array buffer into which the components of the vector are stored. The capacity of the vector is the length of this array buffer, and is at least large enough to contain all the vector's elements.

Any array elements following the last element in the Vector are null.

---

elementCount

```
protected int elementCount
```

The number of valid components in this `Vector` object. Components `elementData[0]` through `elementData[elementCount-1]` are the actual items.

---

capacityIncrement

```
protected int capacityIncrement
```

The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity. If the capacity increment is less than or equal to zero, the capacity of the vector is doubled each time it needs to grow.

Constructor Detail

Vector

```
public Vector(int initialCapacity,
              int capacityIncrement)
```

Constructs an empty vector with the specified initial capacity and capacity increment.

Parameters:

`initialCapacity` - the initial capacity of the vector.

`capacityIncrement` - the amount by which the capacity is increased when the vector overflows.

Throws:

`IllegalArgumentException` - if the specified initial capacity is negative

Vector

```
public Vector(int initialCapacity)
```

Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.

Parameters:

`initialCapacity` - the initial capacity of the vector.

Throws:

`IllegalArgumentException` - if the specified initial capacity is negative

Vector

```
public Vector()
```

Constructs an empty vector so that its internal data array has size `10` and its standard capacity increment is zero.

Vector

```
public Vector(Collection c)
```

Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Parameters:
`c` - the collection whose elements are to be placed into this vector.
Throws:
`NullPointerException` - if the specified collection is null.
Since:
1.2

Method Detail

copyInto

```
public void copyInto(Object[] anArray)
```

Copies the components of this vector into the specified array. The item at index `k` in this vector is copied into component `k` of `anArray`. The array must be big enough to hold all the objects in this vector, else an `IndexOutOfBoundsException` is thrown.

Parameters:
`anArray` - the array into which the components get copied.
Throws:
`NullPointerException` - if the given array is null.

---

trimToSize

```
public void trimToSize()
```

Trims the capacity of this vector to be the vector's current size. If the capacity of this vector is larger than its current size, then the capacity is changed to equal the size by replacing its internal data array, kept in the field `elementData`, with a smaller one. An application can use this operation to minimize the storage of a vector.

---

ensureCapacity

```
public void ensureCapacity(int minCapacity)
```

Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.

If the current capacity of this vector is less than `minCapacity`, then its capacity is increased by replacing its internal data array, kept in the field `elementData`, with a larger one. The size of the new data array will be the old size plus `capacityIncrement`, unless the value of `capacityIncrement` is less than or equal to zero, in which case the new capacity will be twice the old capacity; but if this new size is still smaller than `minCapacity`, then the new capacity will be `minCapacity`.

Parameters:

`minCapacity` - the desired minimum capacity.

---

setSize

```
public void setSize(int newSize)
```

Sets the size of this vector. If the new size is greater than the current size, new `null` items are added to the end of the vector. If the new size is less than the current size, all components at index `newSize` and greater are discarded.

Parameters:

`newSize` - the new size of this vector.

Throws:

`ArrayIndexOutOfBoundsException` - if new size is negative.

---

capacity

```
public int capacity()
```

Returns the current capacity of this vector.

Returns:

the current capacity (the length of its internal data array, kept in the field `elementData` of this vector).

---

size

```
public int size()
```

Returns the number of components in this vector.

Specified by:

`size` in interface `List`

Specified by:

`size` in class `AbstractCollection`

Returns:

the number of components in this vector.

---

isEmpty

```
public boolean isEmpty()
```

Tests if this vector has no components.

Specified by:
`isEmpty` in interface `List`
Overrides:
`isEmpty` in class `AbstractCollection`
Returns:
`true` if and only if this vector has no components, that is, its size is zero; `false` otherwise.

---

elements

```
public Enumeration elements()
```

Returns an enumeration of the components of this vector. The returned `Enumeration` object will generate all items in this vector. The first item generated is the item at index `0`, then the item at index `1`, and so on.

Returns:
an enumeration of the components of this vector.
See Also:
`Enumeration`, `Iterator`

---

contains

```
public boolean contains(Object elem)
```

Tests if the specified object is a component in this vector.

Specified by:
`contains` in interface `List`
Overrides:
`contains` in class `AbstractCollection`
Parameters:
`elem` - an object.
Returns:
`true` if and only if the specified object is the same as a component in this vector, as determined by the `equals` method; `false` otherwise.

---

indexOf

```
public int indexOf(Object elem)
```

Searches for the first occurence of the given argument, testing for equality using the `equals` method.

Specified by:
`indexOf` in interface `List`
Overrides:
`indexOf` in class `AbstractList`
Parameters:
`elem` - an object.
Returns:
the index of the first occurrence of the argument in this vector, that is, the smallest value `k`

such that `elem.equals(elementData[k])` is `true`; returns `-1` if the object is not found.

See Also:

`Object.equals(Object)`

---

indexOf

```
public int indexOf(Object elem,
                   int index)
```

Searches for the first occurence of the given argument, beginning the search at `index`, and testing for equality using the `equals` method.

Parameters:

`elem` - an object.
`index` - the non-negative index to start searching from.

Returns:

the index of the first occurrence of the object argument in this vector at position `index` or later in the vector, that is, the smallest value `k` such that `elem.equals(elementData[k]) && (k >= index)` is `true`; returns `-1` if the object is not found. (Returns `-1` if `index` >= the current size of this `Vector`.)

Throws:

`IndexOutOfBoundsException` - if `index` is negative.

See Also:

`Object.equals(Object)`

---

lastIndexOf

```
public int lastIndexOf(Object elem)
```

Returns the index of the last occurrence of the specified object in this vector.

Specified by:

`lastIndexOf` in interface `List`

Overrides:

`lastIndexOf` in class `AbstractList`

Parameters:

`elem` - the desired component.

Returns:

the index of the last occurrence of the specified object in this vector, that is, the largest value `k` such that `elem.equals(elementData[k])` is `true`; returns `-1` if the object is not found.

---

lastIndexOf

```
public int lastIndexOf(Object elem,
                       int index)
```

Searches backwards for the specified object, starting from the specified index, and returns an index to it.

Parameters:

`elem` - the desired component.
`index` - the index to start searching from.

Returns:

the index of the last occurrence of the specified object in this vector at position less than or equal to `index` in the vector, that is, the largest value `k` such that `elem.equals(elementData[k]) && (k <= index)` is `true`; `-1` if the object is not found. (Returns `-1` if `index` is negative.)

Throws:

`IndexOutOfBoundsException` - if `index` is greater than or equal to the current size of this vector.

---

elementAt

```
public Object elementAt(int index)
```

Returns the component at the specified index.

This method is identical in functionality to the get method (which is part of the List interface).

Parameters:

`index` - an index into this vector.

Returns:

the component at the specified index.

Throws:

`ArrayIndexOutOfBoundsException` - if the `index` is negative or not less than the current size of this `Vector` object. given.

See Also:

`get(int)`, `List`

---

firstElement

```
public Object firstElement()
```

Returns the first component (the item at index `0`) of this vector.

Returns:

the first component of this vector.

Throws:

`NoSuchElementException` - if this vector has no components.

---

lastElement

```
public Object lastElement()
```

Returns the last component of the vector.

Returns:

the last component of the vector, i.e., the component at index `size() - 1`.

Throws:

`NoSuchElementException` - if this vector is empty.

---

setElementAt

```
public void setElementAt(Object obj,
                         int index)
```

Sets the component at the specified `index` of this vector to be the specified object. The previous component at that position is discarded.

The index must be a value greater than or equal to `0` and less than the current size of the vector.

This method is identical in functionality to the set method (which is part of the List interface). Note that the set method reverses the order of the parameters, to more closely match array usage. Note also that the set method returns the old value that was stored at the specified position.

Parameters:

`obj` - what the component is to be set to.
`index` - the specified index.

Throws:

`ArrayIndexOutOfBoundsException` - if the index was invalid.

See Also:

`size()`, `List`, `set(int, java.lang.Object)`

---

removeElementAt

```
public void removeElementAt(int index)
```

Deletes the component at the specified index. Each component in this vector with an index greater or equal to the specified `index` is shifted downward to have an index one smaller than the value it had previously. The size of this vector is decreased by `1`.

The index must be a value greater than or equal to `0` and less than the current size of the vector.

This method is identical in functionality to the remove method (which is part of the List interface). Note that the remove method returns the old value that was stored at the specified position.

Parameters:

`index` - the index of the object to remove.

Throws:

`ArrayIndexOutOfBoundsException` - if the index was invalid.

See Also:

`size()`, `remove(int)`, `List`

---

insertElementAt

```
public void insertElementAt(Object obj,
                            int index)
```

Inserts the specified object as a component in this vector at the specified `index`. Each component in this vector with an index greater or equal to the specified `index` is shifted upward to have an index one greater than the value it had previously.

The index must be a value greater than or equal to `0` and less than or equal to the current size of the vector. (If the index is equal to the current size of the vector, the new element is appended to the Vector.)

This method is identical in functionality to the add(Object, int) method (which is part of the List interface). Note that the add method reverses the order of the parameters, to more closely match array usage.

Parameters:

`obj` - the component to insert.

`index` - where to insert the new component.

Throws:

`ArrayIndexOutOfBoundsException` - if the index was invalid.

See Also:

`size()`, `add(int, Object)`, `List`

---

addElement

```
public void addElement(Object obj)
```

Adds the specified component to the end of this vector, increasing its size by one. The capacity of this vector is increased if its size becomes greater than its capacity.

This method is identical in functionality to the add(Object) method (which is part of the List interface).

Parameters:

`obj` - the component to be added.

See Also:

`add(Object)`, `List`

---

removeElement

```
public boolean removeElement(Object obj)
```

Removes the first (lowest-indexed) occurrence of the argument from this vector. If the object is found in this vector, each component in the vector with an index greater or equal to the object's index is shifted downward to have an index one smaller than the value it had previously.

This method is identical in functionality to the remove(Object) method (which is part of the List interface).

Parameters:

`obj` - the component to be removed.

Returns:

`true` if the argument was a component of this vector; `false` otherwise.

See Also:

`List.remove(Object)`, `List`

---

removeAllElements

```
public void removeAllElements()
```

Removes all components from this vector and sets its size to zero.

This method is identical in functionality to the clear method (which is part of the List interface).

See Also:

`clear()`, `List`

---

clone

`public Object clone()`

Returns a clone of this vector. The copy will contain a reference to a clone of the internal data array, not a reference to the original internal data array of this `Vector` object.

Overrides:
`clone` in class `Object`
Returns:
a clone of this vector.
See Also:
`Cloneable`

---

toArray

`public Object[] toArray()`

Returns an array containing all of the elements in this Vector in the correct order.

Specified by:
`toArray` in interface `List`
Overrides:
`toArray` in class `AbstractCollection`
Returns:
an array containing all of the elements in this list in proper sequence.
Since:
1.2
See Also:
`Arrays.asList(Object[])`

---

toArray

`public Object[] toArray(Object[] a)`

Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array. If the Vector fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this Vector.

If the Vector fits in the specified array with room to spare (i.e., the array has more elements than the Vector), the element in the array immediately following the end of the Vector is set to null. This is useful in determining the length of the Vector *only* if the caller knows that the Vector does not contain any null elements.

Specified by:
`toArray` in interface `List`
Overrides:
`toArray` in class `AbstractCollection`
Parameters:
a - the array into which the elements of the Vector are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose.
Returns:

an array containing the elements of the Vector.

Throws:

`ArrayStoreException` - the runtime type of a is not a supertype of the runtime type of every element in this Vector.

`NullPointerException` - if the given array is null.

Since:

1.2

---

get

```
public Object get(int index)
```

Returns the element at the specified position in this Vector.

Specified by:

`get` in interface `List`

Specified by:

`get` in class `AbstractList`

Parameters:

`index` - index of element to return.

Returns:

object at the specified index

Throws:

`ArrayIndexOutOfBoundsException` - index is out of range (index < 0 || index >= size()).

Since:

1.2

---

set

```
public Object set(int index,
                  Object element)
```

Replaces the element at the specified position in this Vector with the specified element.

Specified by:

`set` in interface `List`

Overrides:

`set` in class `AbstractList`

Parameters:

`index` - index of element to replace.

`element` - element to be stored at the specified position.

Returns:

the element previously at the specified position.

Throws:

`ArrayIndexOutOfBoundsException` - index out of range (index < 0 || index >= size()).

Since:

1.2

---

add

```
public boolean add(Object o)
```

Appends the specified element to the end of this Vector.

Specified by:
`add` in interface `List`
Overrides:
`add` in class `AbstractList`
Parameters:
`o` - element to be appended to this Vector.
Returns:
true (as per the general contract of Collection.add).
Since:
1.2

---

remove

```
public boolean remove(Object o)
```

Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged. More formally, removes the element with the lowest index i such that `(o==null ? get(i)==null : o.equals(get(i)))` (if such an element exists).

Specified by:
`remove` in interface `List`
Overrides:
`remove` in class `AbstractCollection`
Parameters:
`o` - element to be removed from this Vector, if present.
Returns:
true if the Vector contained the specified element.
Since:
1.2

---

add

```
public void add(int index,
                Object element)
```

Inserts the specified element at the specified position in this Vector. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

Specified by:
`add` in interface `List`
Overrides:
`add` in class `AbstractList`
Parameters:
`index` - index at which the specified element is to be inserted.
`element` - element to be inserted.
Throws:
`ArrayIndexOutOfBoundsException` - index is out of range (index < 0 || index > size()).
Since:
1.2

---

remove

```
public Object remove(int index)
```

Removes the element at the specified position in this Vector. shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the Vector.

Specified by:
remove in interface List
Overrides:
remove in class AbstractList
Parameters:
index - the index of the element to removed.
Returns:
element that was removed
Throws:
ArrayIndexOutOfBoundsException - index out of range (index < 0 || index >= size()).
Since:
1.2

---

clear

```
public void clear()
```

Removes all of the elements from this Vector. The Vector will be empty after this call returns (unless it throws an exception).

Specified by:
clear in interface List
Overrides:
clear in class AbstractList
Since:
1.2

---

containsAll

```
public boolean containsAll(Collection c)
```

Returns true if this Vector contains all of the elements in the specified Collection.

Specified by:
containsAll in interface List
Overrides:
containsAll in class AbstractCollection
Parameters:
c - a collection whose elements will be tested for containment in this Vector
Returns:
true if this Vector contains all of the elements in the specified collection.
Throws:
NullPointerException - if the specified collection is null.
See Also:
contains(Object)

---

addAll

`public boolean addAll(Collection c)`

Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator. The behavior of this operation is undefined if the specified Collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified Collection is this Vector, and this Vector is nonempty.)

Specified by:
`addAll` in interface `List`
Overrides:
`addAll` in class `AbstractCollection`
Parameters:
`c` - elements to be inserted into this Vector.
Returns:
`true` if this Vector changed as a result of the call.
Throws:
`ArrayIndexOutOfBoundsException` - index out of range (index < 0 || index > size()).
`NullPointerException` - if the specified collection is null.
Since:
1.2
See Also:
`List.add(Object)`

---

removeAll

`public boolean removeAll(Collection c)`

Removes from this Vector all of its elements that are contained in the specified Collection.

Specified by:
`removeAll` in interface `List`
Overrides:
`removeAll` in class `AbstractCollection`
Parameters:
`c` - a collection of elements to be removed from the Vector
Returns:
true if this Vector changed as a result of the call.
Throws:
`NullPointerException` - if the specified collection is null.
Since:
1.2
See Also:
`List.remove(Object)`, `List.contains(Object)`

---

retainAll

`public boolean retainAll(Collection c)`

Retains only the elements in this Vector that are contained in the specified Collection. In other words, removes from this Vector all of its elements that are not contained in the specified

Collection.

Specified by:
`retainAll` in interface `List`
Overrides:
`retainAll` in class `AbstractCollection`
Parameters:
`c` - a collection of elements to be retained in this Vector (all other elements are removed)
Returns:
true if this Vector changed as a result of the call.
Throws:
`NullPointerException` - if the specified collection is null.
Since:
1.2
See Also:
`List.remove(Object)`, `List.contains(Object)`

---

addAll

```
public boolean addAll(int index,
                      Collection c)
```

Inserts all of the elements in in the specified Collection into this Vector at the specified position. Shifts the element currently at that position (if any) and any subsequent elements to the right (increases their indices). The new elements will appear in the Vector in the order that they are returned by the specified Collection's iterator.

Specified by:
`addAll` in interface `List`
Overrides:
`addAll` in class `AbstractList`
Parameters:
`index` - index at which to insert first element from the specified collection.
`c` - elements to be inserted into this Vector.
Returns:
`true` if this Vector changed as a result of the call.
Throws:
`ArrayIndexOutOfBoundsException` - index out of range (index < 0 || index > size()).
`NullPointerException` - if the specified collection is null.
Since:
1.2

---

equals

```
public boolean equals(Object o)
```

Compares the specified Object with this Vector for equality. Returns true if and only if the specified Object is also a List, both Lists have the same size, and all corresponding pairs of elements in the two Lists are *equal*. (Two elements `e1` and `e2` are *equal* if `(e1==null ? e2==null : e1.equals(e2))`.) In other words, two Lists are defined to be equal if they contain the same elements in the same order.

Specified by:

`equals` in interface `List`

Overrides:

`equals` in class `AbstractList`

Parameters:

o - the Object to be compared for equality with this Vector.

Returns:

true if the specified Object is equal to this Vector

---

hashCode

```
public int hashCode()
```

Returns the hash code value for this Vector.

Specified by:

`hashCode` in interface `List`

Overrides:

`hashCode` in class `AbstractList`

Returns:

the hash code value for this list.

See Also:

`Object.hashCode()`, `Object.equals(Object)`, `List.equals(Object)`

---

toString

```
public String toString()
```

Returns a string representation of this Vector, containing the String representation of each element.

Overrides:

`toString` in class `AbstractCollection`

Returns:

a string representation of this collection.

---

subList

```
public List subList(int fromIndex,
                    int toIndex)
```

Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive. (If fromIndex and ToIndex are equal, the returned List is empty.) The returned List is backed by this List, so changes in the returned List are reflected in this List, and vice-versa. The returned List supports all of the optional List operations supported by this List.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a List can be used as a range operation by operating on a subList view instead of a whole List. For example, the following idiom removes a range of elements from a List:

```
list.subList(from, to).clear();
```

Similar idioms may be constructed for indexOf and lastIndexOf, and all of the algorithms in the Collections class can be applied to a subList.

The semantics of the List returned by this method become undefined if the backing list (i.e., this List) is *structurally modified* in any way other than via the returned List. (Structural modifications are those that change the size of the List, or otherwise perturb it in such a fashion that iterations in progress may yield incorrect results.)

Specified by:

`subList` in interface `List`

Overrides:

`subList` in class `AbstractList`

Parameters:

`fromIndex` - low endpoint (inclusive) of the subList.
`toIndex` - high endpoint (exclusive) of the subList.

Returns:

a view of the specified range within this List.

Throws:

`IndexOutOfBoundsException` - endpoint index value out of range `(fromIndex < 0 || toIndex > size)`
`IllegalArgumentException` - endpoint indices out of order `(fromIndex > toIndex)`

---

removeRange

```
protected void removeRange(int fromIndex,
                           int toIndex)
```

Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the ArrayList by (toIndex - fromIndex) elements. (If toIndex==fromIndex, this operation has no effect.)

Overrides:

`removeRange` in class `AbstractList`

Parameters:

`fromIndex` - index of first element to be removed.
`toIndex` - index after last element to be removed.
