Chapter 10

Dynamic Data Structures

- Vectors
- Linked Data Structures

Overview

- This chapter is about data structures that are *dynamic*:
  They can grow and shrink while your program is running
- Vectors are similar to arrays but are more flexible.
- Linked lists are a dynamic data structure commonly used in many programming languages.

Vectors

"Well, I'll eat it," said Alice, "and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door; so either way I'll get into the garden…"

Lewis Carroll, *Alice's Adventures in Wonderland*

VECTORS

Think of them as arrays that can get larger or smaller when a program is running.

Arrays Versus Vectors

Arrays

Bad:

- Size is fixed when declared
- Inefficient storage: can use a partially full array, but space has been allocated for the full size
- If one more value needs to be added past the maximum size the array needs to be redeclared

Good:

- More efficient (faster) execution
- Allows primitive type elements

Vectors

Good :

- Size is not fixed
- Better storage efficiency: a partially full vector may be allocated just the space it needs
- If one more value needs to be added past the maximum size the vector size increases automatically

Bad:

- Less efficient (slower) execution
- Elements must be class types (primitive types not allowed)

Using Vectors

- Vectors are not automatically part of Java
  - » they are in the `util` library
  - » you must import `java.util.*`
- Create a vector with an initial size of 20 elements:
  `Vector v = new Vector(20);`

Initial Capacity and Efficiency: a Classic Engineering Tradeoff

- *Engineering* involves making difficult tradeoffs
  - » *"There's no such thing as a free lunch."*
    - – an American saying
  - » Usually, if you gain something you lose something somewhere else
- Choosing the initial size of a vector is an example of a tradeoff
  - » making it too large wastes allocated memory space
  - » making it too small slows execution
    - – it takes time to resize vectors dynamically
- Solution?
  - » optimize one at the expense of the other
  - » or make good compromises
    - – choose a size that is not too big and not too small

Vector Syntax

- The idea is the same as for arrays, but the syntax is different
- As with arrays, the index must be in the range 0 to size-of-the-vector

<u>Array:</u> `a` is a `String` array

```
a[i] = "Hi, Mom!");

String temp = a[i];
```

<u>Vector:</u> `v` is a vector

```
v.setElementAt("Hi, Mom!", i);

String temp =
  (String)v.elementAt(i);
```

Instead of the index in brackets and = for assignment, use vector method `setElementAt` with two arguments, the value and the index

Use vector method `elementAt(int index)` to retrieve the value of an element

Note: the cast to `String` is required because the base type of vector elements is `Object`

Vector Methods

- The vector class includes many useful methods:
  - » constructors
  - » array-like methods, e.g. `setElementAt` & `elementAt`
  - » methods to add elements
  - » methods to remove elements
  - » search methods
  - » methods to work with the vector's size and capacity, e.g. to find its size and check if it is empty
  - » a `clone` method to copy a vector
- See the text for more information

A little Detail about **`setElementAt`**

"The devil's in the details."

– an American engineering saying

- Vectors put values in successive indexes
  - » `addElement` is used to put initial values in a vector
  - » new values can be added only at the next higher index
- You cannot use `setElemetAt` to put a value at any index
  - » `setElemetAt` can be used to assign the value of an indexed variable only if it has been previously assigned a value with `addElement`

Base Type of Vectors

- The base type of an array is specified when the array is declared
  - » all elements of arrays must be of the same type
- The base type of a vector is `Object`
  - » elements of a vector can be of any class type
  - » in fact, *elements of a vector can be of different class types!*
  - » to store primitive types in a vector they must be converted to a corresponding wrapper class

Good Programming Practice

Although vectors allow elements in the same vector to be of different class types, it is best not to have a mix of classes in the same vector -

– it is best to have all elements in a vector be the same class type.

Detail: One Consequence of the Base Type of Vectors Being **Object**

- The following code looks very reasonable but will produce an error saying that the class `Object` does not have a method named `length`:

```
Vector v = new Vector()
String greeting = "Hi, Mom!";
v.addElement(greeting);
System.out.println("Length is " +
   (v.elementAt(0)).length());
```

- `String`, of course, does have a `length` method, but Java sees the type of `v.elementAt(0)` as `Object`, not `String`
- Solution? Cast `v.elementAt(0)` to `String`:

```
System.out.println
   ("Length is " + (String)(v.elementAt(0)).length());
```


One More Detail: Size Versus Capacity

- Be sure to understand the difference between *capacity* and *size* of a vector:
  - » *capacity* is the declared size of the vector
    - – the current maximum number of elements
  - » *size* is the actual number of elements being used
    - – the number of elements that contain valid values, not garbage
    - – remember that vectors add values only in successive indexes
- Loops that read vector elements should be limited by the value of `size`, not `capacity`, to avoid reading garbage values

Programming Tip: Increasing Storage Efficiency of Vectors

- A vector automatically increases its size if elements beyond its current capacity are added
- But a vector does not automatically decrease its size if elements are deleted
- The method `trimSize` shrinks the capacity of a vector to its current size so there is no extra, wasted space
  - » the allocated space is reduced to whatever is currently being used
- To use storage more efficiently, use `trimSize` when a vector will not need its extra capacity later

And Another Detail: Correcting the Return Type of **`clone`**

- The method `clone` is used to make a copy of a vector but its return type is `Object`, not `Vector`
  - » of course you want it to be `Vector`, not `Object`
- So, what do you do?
  - » Cast it to `Vector`

```
Vector v = new Vector(10);
Vector otherV;
otherV = vector;
Vector otherV = (Vector)v.clone();
```

This just makes `otherV` another *name* for the vector `v` (there is only one copy of the vector and it now has two names)

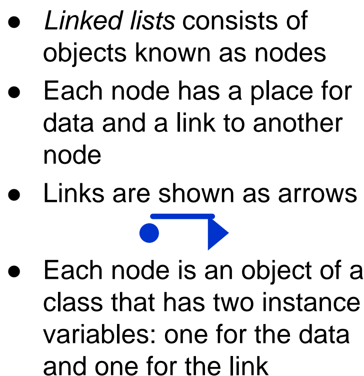
This creates a *second* copy of `v` with a different name, `otherV`

And Yet Another Detail: Protecting Private Variables

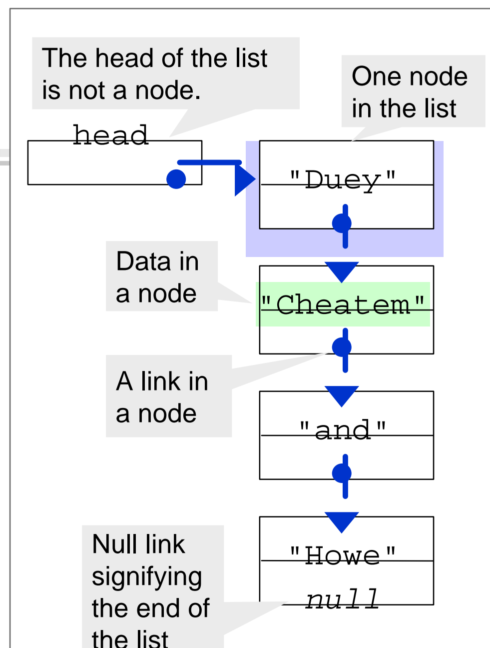
- Just as with arrays, be careful not to return addresses of private vector variables, otherwise calling methods can access them directly
  - » "Information Hiding" is compromised
- To protect against it, return a copy of the vector
  - » use `clone` as described in the previous slide
- But that's not all:
  - » if the elements of the vector are class (and not primitive) types, they may not have been written to pass a copy
  - » they may pass the address
  - » so additional work may be required to fix the accessor methods

Linked Lists

- *Linked lists* consists of objects known as nodes
- Each node has a place for data and a link to another node
- Links are shown as arrows

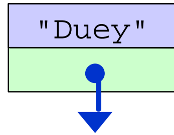


- Each node is an object of a class that has two instance variables: one for the data and one for the link



ListNode Class:
Instance Variables and Constructor

```
public class ListNode
{
    private String data;
    private ListNode link;

    public ListNode(String newData, ListNode linkValue)
    {
        data = newData;
        link = linkValue;
    }
```



Two parameters for the constructor:

- data value for the new node
- Link value for the new node

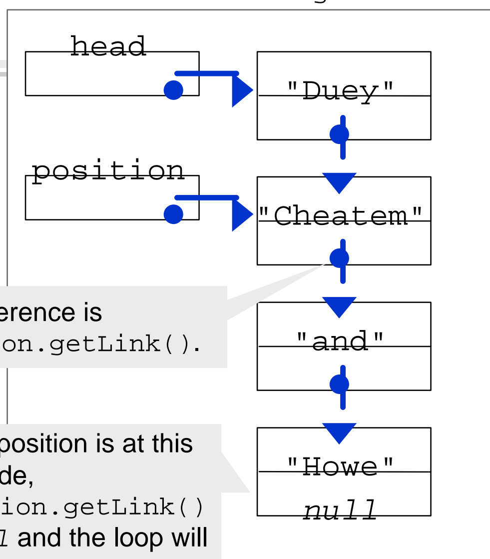
Stepping through a List

Excerpt from `showList` in `StringLinkedList`

Start at beginning of list

```
ListNode position;
position = head;
while (position != null)
{
    ...
    position =
      position.getLink();
}
```

Moves to next node in the list.



This reference is `position.getLink()`.

When position is at this last node, `position.getLink()` is *`null`* and the loop will terminate.

Adding a Node

To add a node at the beginning of the list:

```
public void addANodeToStart(String addData)
{
   head = new ListNode(addData, head);
}
```

- The new node will point to the old start of the list, which is what `head` pointed to.
- The value of `head` is changed to point to the new node, which is now the first node in the list.

Deleting a Node

To delete a node from the beginning of the list:

```
public void deleteHeadNode()
{
   if (head != null)
   {
      head = head.getLink();
   }
   else
        // prints an error message and exits
...
```

- Doesn't try to delete from an empty list.
- Removes first element and sets head to point to the node that was second but is now first.

Node Inner Classes

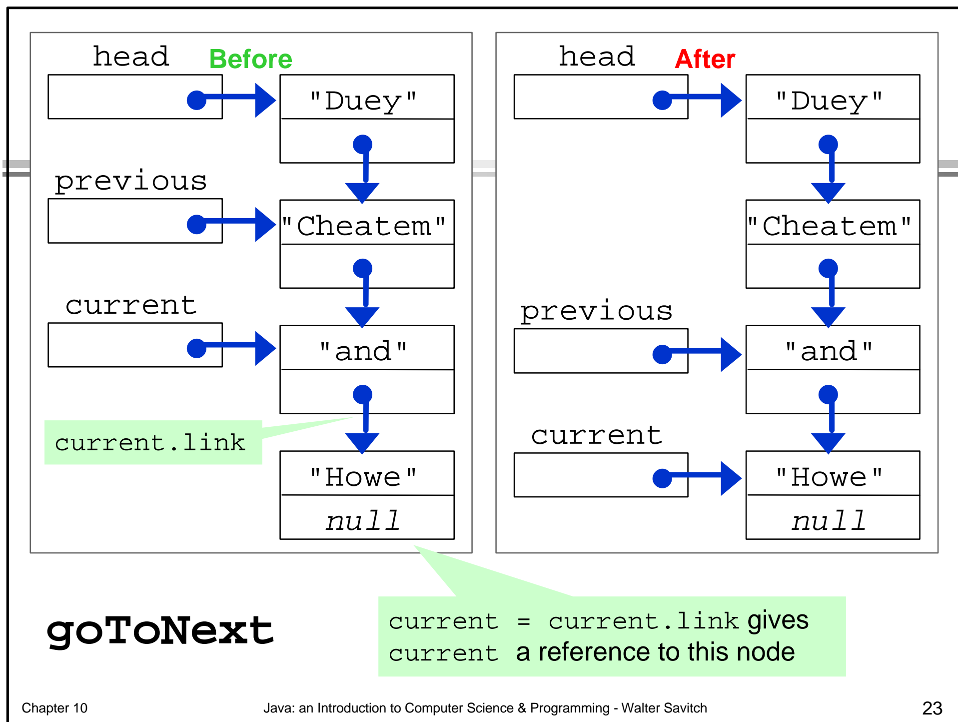
```
public class StringLinkedList
{
   private ListNode head;
      <methods for StringLinkedList inserted here>

   private class ListNode
   {
      <Define ListNode instance variables and methods here>
   }
}
```

- Using an inner class makes `StringLinkedList` self-contained because it doesn't depend on a separate file
- Making the inner class private makes it safer from the point of view of information hiding

Iterators

- An object that allows a program to step through a collection of objects and do some action on each one is called an *iterator*.
- For arrays, an index variable can be used as an iterator, with the action of going to the next thing in the list being something like:
  `index++;`
- In a linked list, a reference to the node can be used as an iterator
- `StringLinkedListSelfContained` has an instance variable called `current` that is used keep track of where the iteration is.
- The goToNext method moves to the next node in the list by using the statement:

```
current = current.next;
```



Other Methods in the Linked List with Iterator

- `getDataAtCurrent`—returns the data part of the node that the iterator (`current`) is at
- `moreToIterate`–returns a `boolean` value that will be true if the iterator is not at the end of the list
- `resetIteration`–moves the iterator to the beginning of the list

- Can write methods to add and delete nodes at the iterator instead of only at the head of the list.
  » Following slides show diagrams illustrating the add and delete methods.

head Before
"Duey"
current
"Cheatem"
"Howe"
null
newNode
"and"
null

head After
"Duey"
current
"Cheatem"
"Howe"
null
newNode
"and"

Adding a Node
Step 1

Create the node with reference `newNode`
Add data to the node

```
newNode.link = current.link
```


head Before
"Duey"
current
"Cheatem"
"Howe"
null
newNode
"and"

head After
"Duey"
current
"Cheatem"
"Howe"
null
newNode
"and"

Adding a Node
Step 2

```
current.link = newNode.link
```

The node has been added to the list although it might appear out of place in this diagram.

Adding a Node

- After creating the node, the two statements used to add the node to the list are:

```
newNode.link = current.link;
current.link = newNode;
```

- What would happen if these two steps were done in reverse order?

previous
Before
"Duey"
current
"Cheatem"
This node will be removed from the list.
"and"
"Howe"
null

previous
After
"Duey"
current
"Cheatem"
"and"
newNode
"Howe"
null

Deleting a Node
Step 1

```
previous.link = current.link
```

What should be done next?

previous

Before

"Duey"

current

"Cheatem"

"and"

newNode

"Howe"

null

previous

After

"Duey"

This node is not accessible from the head of the list.

"Cheatem"

current

"and"

newNode

"Howe"

null

Deleting a Node

Step 2

`current = current.link`

The node has been deleted from the list although it is still shown in this picture.

A Doubly Linked List

- A doubly linked list allows the program to move backward as well as forward in the list.
- The beginning of the node class for a doubly-linked list would look something like this:

```
private class ListNode
{
    private Object data
    private ListNode next;
    private ListNode previous;
```

Declaring the data reference as class `Object` allows any kind of data to be stored in the list.

null

null

Summary

- Vectors can be thought of as arrays that can grow in length as needed during run time.
- The base type of all vectors is `Object`.
- Thus, vector elements can be of any class type, but not primitive types.
- A linked list is a data structure consisting of objects known as nodes, such that each node can contain data, and each node has a reference to the next node in the list.
- You can make a linked list self-contained by making the node class an inner class of the linked list class.
- You can use an iterator to step through the elements of a collection.