Lecture 08: Support for Abstract Data Types

Support for abstract data types (ADTs) is critical in contemporary programming. This lecture discusses the topic under the following subheadings:

- Introduction
- Design Comparisons
- Array-lists and Vectors
- Linked-lists
- Other ADTs
- Summary and Concluding Remarks

8.1 Introduction

An *abstract data type* (ADT) is a programming construct with a defined set of data items, and a set of possible operations on those data items. In contemporary programming, ADTs are implemented as *classes* and/or *packages*. Because of the nature of ADTs in contemporary programming, we typically look for their support in object-oriented programming languages (OOPLs).

Common ADTs as discussed in your course in Data Structures and Algorithms:
- Dynamic Lists
- Linked Lists
- Stacks
- Queues
- Binary Trees
- Binary Search Trees (BSTs)
- Heaps
- B-trees
- Hash tables
- Graphs

Common sort algorithms include the following:
- Straight Selection-sort
- Exchange Selection-sort
- Insertion-sort
- Bubble –sort
- Quick-sort
- Merge-sort
- Tree-sort (as in BST)
- Heap-sort

In your course in Data Structures and Algorithms, you would have learned how to construct, implement, and test these ADTs and algorithms in at least one programming language (but preferably multiple programming environments). In that course, you would have also learned how to analyze these algorithms for efficiency.

In this lecture, we shall look at implementation of these ADTs but from a much broader perspective. Here, our concern is not implementation details in any given language, but rather, how different programming languages provide support for these ADTs. As you look at a new programming language, this broadened focus should prepare you nicely for learning any language within a short timeframe (which by the way is an important objective of the course). The lecture gets you started but as usual, I will not be doing all the work; rather, you will be given important guidelines in your language exploration.

8.2 Design Comparisons

The obvious starting point is the construction of a class. How is this done in the new language that you are probing? Let's start with the familiar before launching into the unknown. Figure 8-1 shows the basic Java class anatomy, while figure 8-2 shows the C++ class anatomy. Take some time to familiarize yourself, or refresh your memory on class definition in both languages. Notice the similarities as well as the differences.

Figure 8-1: Anatomy of a Java Class

JavaClass ::=

```
public | private | protected class <ClassName>
[extends<ClassName>][implements<InterfaceName>[,…<InterfaceN>] ]
{
  // Data Item(s)
  …
  //Member Methods
  <Modifier> <ClassName> (<Parameter(s)>) // the constructor
  {
   // …
  }
  // … Additional methods
}
```

Method ::=

```
<Modifier><ReturnType><MethodName>(<Parameters>)
{
  … // Body of Method
}
```

Modifier ::=

```
[final] public | private | protected [static] [abstract]
```

Each keyword is important and therefore needs some clarification:

- The **public** keyword means that the method is available from anywhere in the program or from another class.
- The **private** keyword means that only instances of the class has access to this method.
- The **protected** keyword means that the method is protected within the class hierarchy only. It will be further clarified later in the course (lecture 6).
- The **static** keyword means that the method can be (and is of often) used without an instance of the class being created. In such case the class-name takes the place of the instance name.
- Keyword **abstract** means that the class or method is abstract. An abstract method has no statement (s). An abstract class is one for which instances cannot be created. It consists of at least one abstract method.
- The keyword **class** simply indicates to the Java compiler that a class is being defined.
- The **final** keyword means that the class or method cannot be inherited.

Figure 8-2: Syntax for Defining a C++ Class

```
<ClassDeclaration> :: =
class <ClassName>  [: <Modifier> <BaseClass1> [… , <Modifier> <BaseClass1>] ]
{
        [private:]
    <Private Members>       /* data items and function prototypes*/
        [public:]
    <Public Members>]       /* data items and function prototypes */
    [protected:]
    <Protected Members>]    /* data items and function prototypes */
};

/* Actual function definitions follow */
// …
```

```
<FunctionDefinition>::=
<ReturnType> <FunctionName> ([<Parameter>] [*…,<Parameter>*])
{
        <FunctionBody>
}
```

```
<FunctionPrototype>::=
<ReturnType> <FunctionName> ([<Parameter>] [*…,<Parameter>*])
```

```
<MemberFunctionSpecification> ::=
<ReturnType> <Class Name> :: <MemberFunctionName> ([<Parms>])
{
…
}
```

```
Modifier ::=
public | private | protected
```

One obvious difference you will notice between the C++ and the Java class definitions is that the syntax rules are different; this is expected. Another more far-reaching difference has to do with the principle of inheritance. From the definitions you can clearly see that Java supports inheritance from a single super-class, while C++ supports multiple inheritances from different super-classes.

Another not-so-obvious but very significant difference (but one you are no doubt familiar with) has to do with the specification of component methods of a class: Java prefers the term *method(s),* and insists that these component methods be part of the definitional structure of the class; in other words, they must be specified when the class is being defined. C++ prefers the term *member function(s).* Moreover, C++ provides a mechanism for declaring the member functions at class definition (via function prototyping), but not specifying the details of their internal code. The language gives the programmer the flexibility of specifying detail codes for these member functions at a subsequent time, and such detailed code may or may not be part of the file with the class definition. To tie the member functions back to their host class, the scope resolution operator (::) is used.

8.2 Design Comparison (continued)

Let us illustrate these observations in an example: Figure 8-3 shows the UML class diagram for a **LibraryPatron** class to represent patrons who may use the services of a library. Figure 8-4a shows a simple Java code for this and figure 8-4b shows the equivalent C++ code. Observe that there are various areas of similarity between the Java coding and the C++ coding. This should not surprise you, since as you are aware, Java was written mainly in C++. However, there are some differences as explained below.

Figure 8-3: The UML Diagram of the LibraryPatron Class

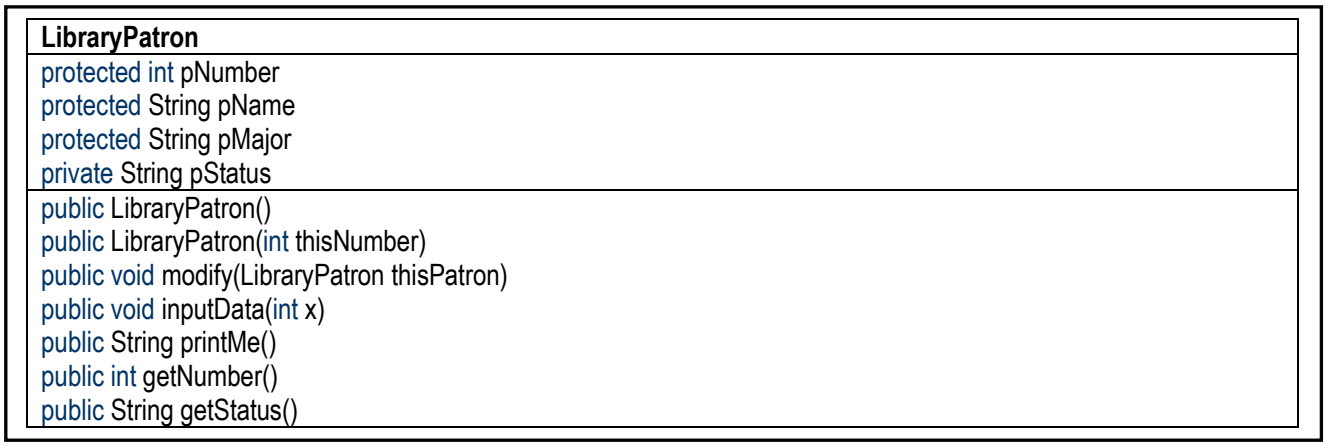


Figure 8-4a Java Code for the LibraryPatron Class

```
// LibraryPatron.java:  Allows for the definition of Library Patron objects.
//  ********************************************************************************
package adtDemo;
import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.

public class LibraryPatron
{
  // Define Data Items
  protected int pNumber; protected String pName, pMajor;
  private String pStatus;

   // Constructors
  public LibraryPatron() // Default constructor
  {
   pNumber = 0;  pName = "No Name"; pMajor = "No Major"; pStatus = "Excellent. Noting outstanding";
  } // End Constructor
  public LibraryPatron(int thisNumber) // Overloaded Constructor
  {
   pNumber = thisNumber; pName = "No Name";   pMajor = "No Major";  pStatus = "Excellent. Noting outstanding";
  } // End Overloaded Constructor

  // Important setters
  public void modify(LibraryPatron thisPatron) // Modification Method
  {
   pNumber = thisPatron.pNumber; pName = thisPatron.Name;  pMajor = thisPatron.pMajor;  pStatus = thisPatron.pStatus;
  } // End of Modification Method

   public void inputData(int x) //InputData Method
   {
     String PatronHeading = "Lambert Cox Library Patron Data Entry";
     String pNumberString = JOptionPane.showInputDialog(null, "Please Enter Patron Number of Patron #" + x +": ",  +
           PatronHeading, JOptionPane.QUESTION_MESSAGE);
     pNumber = Integer.parseInt(pNumberString);
     pName = JOptionPane.showInputDialog(null, "Please Enter Name of Patron #" + x +": ", PatronHeading,  +
              JOptionPane.QUESTION_MESSAGE);
     pMajor = JOptionPane.showInputDialog(null, "Please Enter Major of Patron #" + x +": ", PatronHeading,  +
              JOptionPane.QUESTION_MESSAGE);
     pStatus = JOptionPane.showInputDialog(null, "Please Enter Status of Patron #" + x +": ", PatronHeading,  +
              JOptionPane.QUESTION_MESSAGE);
   } // End of InputData Method

   // Important getters
   public String printMe() // The printMe method
   {  String printString = "Patron Number: " + PatronNumber + "\n" + "Name: " + Name + "\n" +  "Major: " + Major + "\n" + "Status: " + pStatus;
      return printString;
   } // End  printMe Method

   public int getNumber()  // getNumber Method
   {  return pNumber; } // End of getNumber Method

   public String getStatus()  // getStatus Method
   {  return pStatus; } // End of getStatusMethod

} // End of LibraryPatron
```

Figure 8-4b C++ Code for the LibraryPatron Class

```
// LibraryPatron.java:  Allows for the definition of Library Patron objects.
//  ********************************************************************************
#include <cstdlib>   #include <iostream>
#include<ctype.h>  #include<string.h>
using namespace std;

class LibraryPatron
{
  private: // Private data items
  string pStatus;
  protected: // Protected data items
  int pNumber;
  string pName, pMajor;

  // Public prototypes of member functions
  public:
  LibraryPatron();
  LibraryPatron(int thisNumber);
  void modify(LibraryPatron thisPatron);
  void inputData(int x);
  string printMe();
  int getPatronNumber();
  string getPatronStatus();
} // End of LibraryPatron Declaration

//… Specification of Member Functions — sometimes conveniently placed in another file
LibraryPatron :: LibraryPatron()  // Default constructor
{ pNumber = 0;  pName = "No Name"; pMajor = "No Major"; pStatus = "Excellent. Noting outstanding"; }
LibraryPatron :: LibraryPatron(int thisNumber ) // Overloaded constructor
{ pNumber = thisNumber;  pName = "No Name"; pMajor = "No Major"; pStatus = "Excellent. Noting outstanding"; }

  // Important setters
void LibraryPatron :: modify (LibraryPatron thisPatron)  // the modify member function
{
 pNumber = thisPatron.pNumber; pName = thisPatron.Name;  pMajor = thisPatron.pMajor;  pStatus = thisPatron.pStatus;
} // End of modify member function

void LibraryPatron :: inputData(int x)  // the inputData member function
{
    string PatronHeading = "Lambert Cox Library Patron Data Entry"; cout << patronHeading + “\n”;
    cout << "\nPlease Enter Patron Number for Patron #" + x +": ",  cin >> pNumber; getchar();
    cout << "\nPlease Enter Name for Patron #" + x +": ",  cin >> pName;
    cout << "\nPlease Enter Major for Patron #" + x +": ",  cin >> pMajor;
    cout << "\nPlease Enter Status for Patron #" + x +": ",  cin >> pStatus;
} // End of inputData member function

// Important getters
string LibraryPatron :: printMe() // the printMe member function
{ string printString = "Patron Number: " + PatronNumber + "\n" + "Name: " + Name + "\n" +  "Major: " + Major + "\n" + "Status: " + pStatus;
      return printString;
} // End of printeMe member function

int LibraryPatron :: getNumber() // the getNumber member function
{ return pNumber; } // End of getNumber member function

string LibraryPatron :: getStatus() // the getStatus member function
{ return pStatus; } // End of getStatus member function
```

8.2 Design Comparison (continued)

Figure 8-5 provides a summary of the main differences between C++ class implementation and Java class implementation. When learning a new language, it is a good idea to construct a comparison list like this, where on each programming principle of concern, you compare the implementation detail of the language you are learning with that of a language which you are very familiar with. This technique helps you to relate the new language to something that you are familiar with.

Figure 8-5: Comparison of Java Class Implementation with C++ Class Implementation

Java	C++
Each class must be defined in a separate file with the same name as the class-name	A program file may or may not contain zero or more class definitions. Moreover, the program-file may carry a different name from the class name.
A class is made up of data items and/or methods.	A class is made up of data items and/or member functions.
The methods are defined as part of the class definition.	The member functions may or may not be defined in the class declaration. The class declaration may contain member function prototypes instead of detailed function definitions (this is the preferred approach). The detailed function definitions typically follow the class declaration.
Multiple inheritance is not supported	Multiple inheritance is supported
Interfaces are supported	Interfaces are not supported.

Exercise 1: Try finding out how classes are supported in languages such as C#, Ada, Python, and Ruby. Set up your comparison grids for various areas of concern to you.

8.3 Array-lists and Vectors

Recall from your study of Data Structures and Algorithms, that array-lists and vectors are two forms of dynamic lists that are supported in many languages.

8.3.1 Array-Lists

An array-list is equivalent to a variable-length array that allows manipulation (insertion, modification, or deletion) at any point in the list; moreover, the list may shrink or grow as required. In Java, an array-list is best implemented by creating an instance of the **ArrayList** class. Recall that the Java **ArrayList** class is an ADT with instances of Java's **Object** class, and various methods for manipulating the instances. Figure 8-6a provides the UML diagram of the **ArrayList** class. Referring to the **LibraryPatron** class of figure 8-3, figure 8-6b illustrates how an array-list of **LibraryPatron** instances may be declared and created in Java. From here, you should be able to figure out (either by recollection or analysis of figure 8-6a) how to manipulate the list.

Figure 8-6a: UML Diagram for the Java ArrayList Class

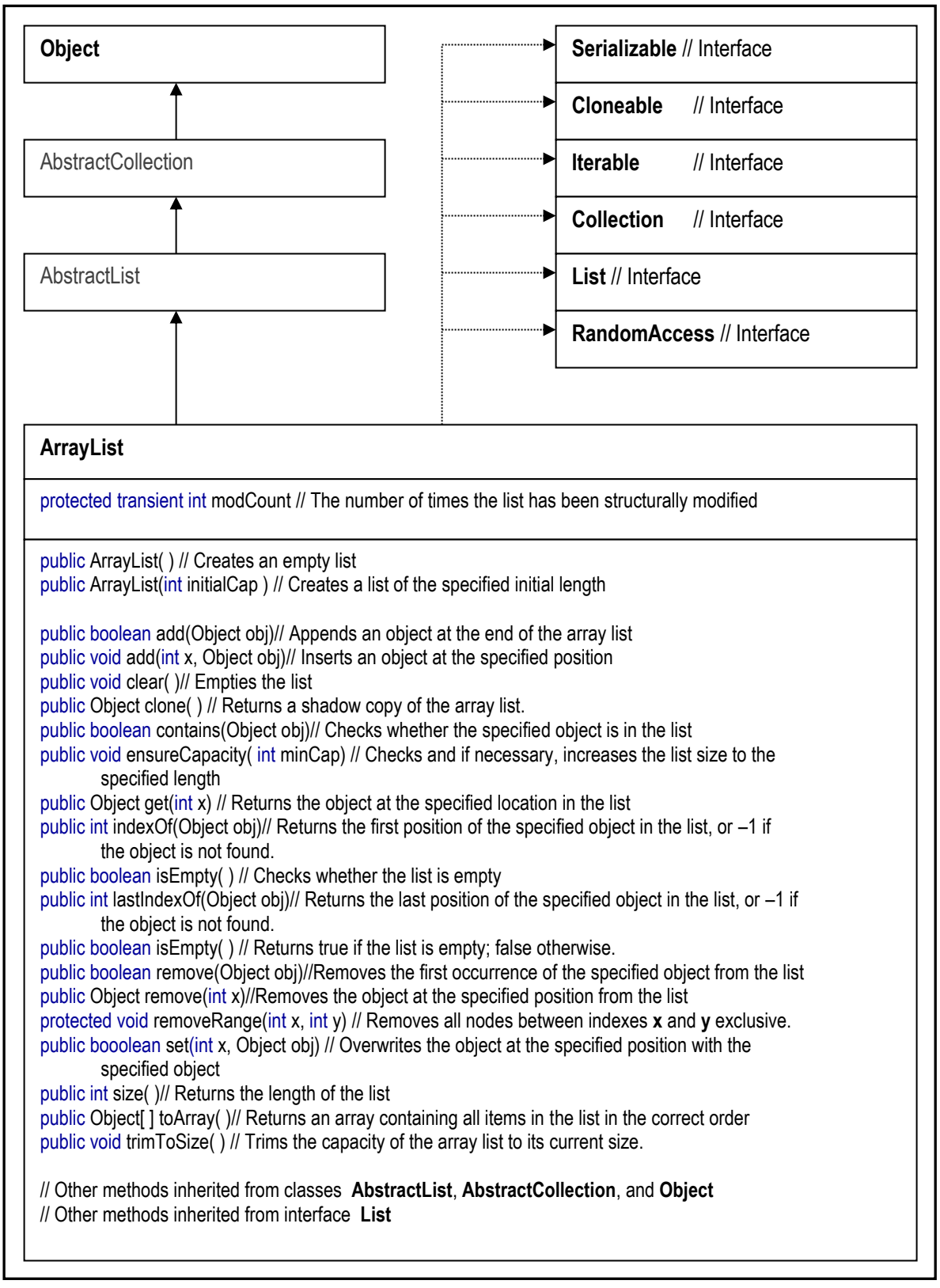


Figure 8-6b: Declaring and Creating a Java ArrayList of LibraryPatron Instances

```
package adtDemo;
import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.
import java.util.ArrayList;

public class PatronsArrayListMonitor
{
  // Global Data Items
  public static ArrayList PatronsList;
  // . . .
  public static void main(String[] args)
  {
    // . . .
    initializeList(); // . . .
    inputPatrons();
   // …
  } // End of main method

  // Initialize Method
  public static void initializeList()
  {
   PatronsList = new ArrayList(0); // Creates a default array-list of LibraryPatron objects
  // . . .
  } // End of initializeList method

 public static void inputPatrons()
 {
   int numberOfPatrons, x;
   LibraryPatron currentPatron;
   numberOfPatrons = Integer.parseInt(JOptionPane.showInputDialog(null, "Number of Patrons: ", HEADING, +
                    JOptionPane.QUESTION_MESSAGE));
   PatronsList.ensureCapacity(PatronsList.size() + numberOfPatrons); // Ensure correct size of list
   for (x =1; x <= numberOfPatrons; x++)
   {
    currentPatron = new LibraryPatron();
    currentPatron.inputData(x); // Prompt For and Accept LibraryPatron Data
    PatronsList.add(x-1, currentPatron);
   };   // End For
 } // End of inputPatrons Method

} // End of PatronsArrayListMonitor class
```

As mentioned in lecture 3, C++ handles array-lists in a much simpler but more elegant way: You can create a pointer to a base type, and manipulate the implied list as an array. Referring to the **LibraryPatron** class of figure 8-3, we can declare a pointer to **LibraryPatron**, and manipulate the resultant list as an array. Figure 8-7a provides the C++ syntax for creating such a list and allocating memory for it. And figure 8-7b illustrates how one could use this guideline to create and manage a dynamic list of **LibraryPatron** instances.

Figure 8-7a: C++ Syntax for Defining a Dynamic List, and Allocating Memory for it

```
DynamicListDefinition ::=
<BaseType>* <TargetList>; // You must specify the base-type and the name of the target-list
// …
<TargetList> = new <BaseType>"["<Length>"]"; // You must specify the amount of items
```

Figure 8-7b: Declaring and Creating Dynamic List of LibraryPatron Instances in C++

```
// ...
typedef LibraryPatron* PatronsList;
// ...
// Obtain list of patrons
int pLim = 0;
 PatronsList  pList = inputPatrons(pLim);
// ...
PatronsList inputPatrons(int &lSize)
{
  cout << "\n\nPlease enter the number of patrons required: ";
  cin >> lSize; getchar();
  PatronsList rList = new LibraryPatron[lSize]; // Allocate space for lSize LibraryPatron objects

  // Prompt for information on each student
  for (int x = 1; x <=lSize; x++)
  {       rList[x-1] = LibraryPatron(); // Instantiate the item
          rList[x-1].inputData(x);   // Obtain  information for the item
  }
  return rList;
} // End of inputPatrons Function
```

8.3.2 Vectors

Like an array-list, a vector is an ADT that maintains a dynamic list of objects belonging to a specified base type/class. Generally speaking, the vector provides more flexibility in its range of services provided to the programmer. Again using Java as a frame of reference, figure 8-8a shows the UML diagram for the Java **Vector** class. Following on, figure 8-8b illustrates how vector of **LibraryPatron** instances may be declared and created in Java. As you view figures 8-8a and 8-8b, notice the strong similarities between the **ArrayList** class and the **Vector** class, as well as their related applications.

Figure 8-8a: UML Diagram for the Java Vector Class

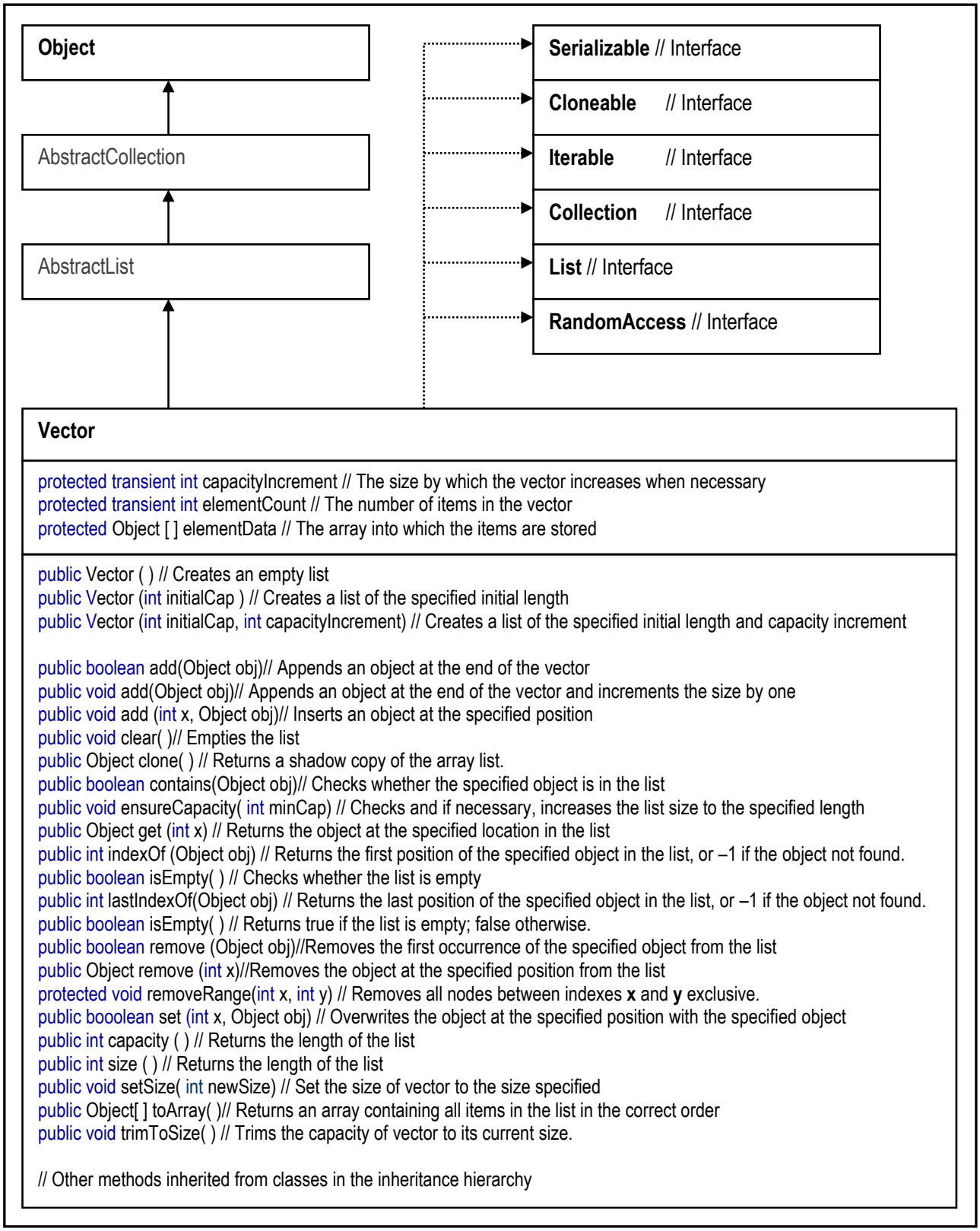


Figure 8-8b: Declaring and Creating a Java Vector of LibraryPatron Instances

```
package adtDemo;
import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.
import java.util.Vector;

public class PatronsVectorMonitor
{
  // Global Data Items
  public static Vector PatronsList;
  // . . .
  public static void main(String[] args)
  {
    // . . .
    initializeList(); // . . .
    inputPatrons();
   // …
  } // End of main method

  // Initialize Method
  public static void initializeList()
  {
   PatronsList = new Vector(0); // Creates a default array-list of LibraryPatron objects
  // . . .
  } // End of initializeList method

 public static void inputPatrons()
 {
   int numberOfPatrons, x;
   LibraryPatron currentPatron;
   numberOfPatrons = Integer.parseInt(JOptionPane.showInputDialog(null, "Number of Patrons: ", HEADING, +
                    JOptionPane.QUESTION_MESSAGE));
   PatronsList.setSize(PatronsList.size() + numberOfPatrons); // Ensure correct size of list
   // PatronsList.ensureCapacity(PatronsList.size() + numberOfPatrons); // This would also work as setSize(…)
   for (x =1; x <= numberOfPatrons; x++)
   {
    currentPatron = new LibraryPatron();
    currentPatron.inputData(x); // Prompt For and Accept LibraryPatron Data
    PatronsList.add(x-1, currentPatron);
   };   // End For
 } // End of inputPatrons Method

} // End of PatronsVectorMonitor class
```

In the interest of comparison, let us now examine how C++ handles vectors. Figure 8-9a provides the basic C++ syntax for declaring a vector. This is followed by figure 8-9b, which shows a list of important member functions in the **vector** class. No revisiting the **LibraryPatron** class of figure 8-3, we can declare a vector of **LibraryPatron** objects. Figure 8-9c illustrates how one could use this guideline to create and manage such a vector.

Figure 8-9a: C++ Syntax for Defining a Vector

VectorDefinition ::=
vector "<"<BaseType>">" <VectorName>;

Note:
1. The angular bracket that encloses the base type is required as prat of the syntax, hence the use of quotation marks.
2. The base type specified may be any valid primitive or advanced data type.

Figure 8-9b: Selected Member Functions of the vector Class

Member Function	Description
begin()	Returns the iterator to the beginning of the vector.
end()	Returns the iterator to the beginning of the vector.
size()	Returns an unsigned integer with the size (i.e. length) of the vector.
max_size()	Returns an unsigned integer with the maximum possible size (i.e. length) of the vector.
resize(. . .)	Resizes the vector to the number items indicated by the first argument; the second argument if specified, contains the default value for each element after the original size has been reached; if the second argument is not specified, additional elements after the original size is reached, are assigned the default value for the base type; returns void.
capacity()	Returns the storage space (in terms of number of elements) currently allocated for the vector; typically varies between **size**() and **max_size**().
empty()	Returns **true** or **false**, indicating whether the vector is empty or not
reserve(. . .)	Requests to ensure a minimum capacity for the vector; the argument specifies the minimum capacity; returns void.
shrink_to_fit()	Requests that the vector capacity be shrunk to be equal to its current size.
operator [ ]	Overloaded operator **[ ]** to access the particular element at the specified relative location in the vector; the argument is identical to an array subscript.
at(. . .)	Returns the particular element at the specified relative location in the vector; the argument is identical to an array subscript.
front()	Returns the element at the front of the vector, i.e., the first element.
back()	Returns the element at the back of the vector, i.e., the last element.
push_back(. . .)	Inserts an element at the back of the vector; the argument must be of the base type of the vector.
insert(. . .)	Inserts an element at the iterator position and returns the iterator position; the first argument states the iterator position; the second argument is the element to be inserted. Example: If **v** is a vector of integers, then two valid insertions are as follows: iterator it = v.insert(v.begin(), 100); it = v.insert(v.begin() +1, 300); Other overloaded forms of this member function exists.
pop_back()	Removes the last element from the vector; returns void.
erase(. . .)	Removes the element(s) indicated by the argument, which is specified in terms of iterator positions, e.g. v.**begin**(), v.**begin**() + 3, etc., where **v** represents a vector; returns void.
swap(. . .)	Exchanges the contents of the current vector with those of the vector supplied as the argument; returns void. Example: If vi and v2 are two vectors, swap them vie the statement v1.swap(v2);
clear()	Removes all elements from the vector; returns void.

Figure 8-9c: Declaring and Creating a C++ Vector of LibraryPatron Instances

```
// …
typedef vector <LibraryPatron> PatronsList;
// …

// Obtain list of patrons
int pLim = 0;
PatronsList  pList = inputPatrons(pLim);

// …

PatronsList inputPatrons(int &lSize)
{
  LibraryPatron currentPatron; PatronList rList;
  cout << "\n\nPlease enter the number of patrons required: ";
  cin >> lSize; getchar();

  // Prompt for information on each patron
  for (int x = 1; x <=lSize; x++)
  {       currentPatron = LibraryPatron(); // Instantiate the item
          currentPatron.inputData(x);   // Obtain information for the item
          rList.push_back(currentPatron);
  }
  return rList;
} // End of inputPatrons Function
```

8.4 Linked Lists

From the knowledge and skills acquired in your Data Structures and Algorithms course, recall that a linked-list is a data structure with the following properties:

- Each node belongs to a particular base type.
- Each node has a pointer to the next node in the list.
- It is often advantageous to specially label the first and last nodes in the list; we will denote them as **First** and **Last** respectively.

8.4.1 Linked List in Java

In Java, a programmer may construct a linked-list in a variety of ways as summarized below:

- **Build from Scratch:** Create an instance class if necessary to store the pertinent information, a node class, where each node contains the pertinent information, and a pointer to the next node in the list. Then create a linked-list class that manipulates instances of the node class, according to the various desired operations. Then create a driver (controller) class (the stack-class can be implemented as your driver class, but it is better to keep them separate).

- **Build from the ArrayList or Vector Class:** Create an instance class if necessary to store the pertinent information, a generic linked-list class that manipulates an instance of Java's generic **ArrayList** or **Vector** class to store instances of the pertinent information, according to the various linked-list operations desired. Then create a driver (controller) class (the linked-list class can be implemented as your driver class, but it is better to keep them separate).

- **Build from the LinkedList Class:** Create an instance class if necessary to store the pertinent information. Then create a driver (controller) class that implements the linked-list as an instance of the **LinkedList** class, and containing the pertinent information.

Revisiting the **LibraryPatron** class of earlier discussions (see figure 8-3), figure 8-10a shows how you may design from scratch, a linked-list of **LibraryPatron** instances. The figure omits the code snippet for the driver class; it is assumed that you understand that this would be required as well.

With your knowledge of data Structures and Algorithms, you should be able to use the guidelines above to create and employ a generic linked-list from via the **ArrayList** or **Vector** class; this is left as an exercise for you. You may also achieve a similar objective by making use of Java's generic **LinkedList** class. The UML diagram for the **LinkedList** class is shown in figure 8-10b; further exploration is left as an exercise for you.

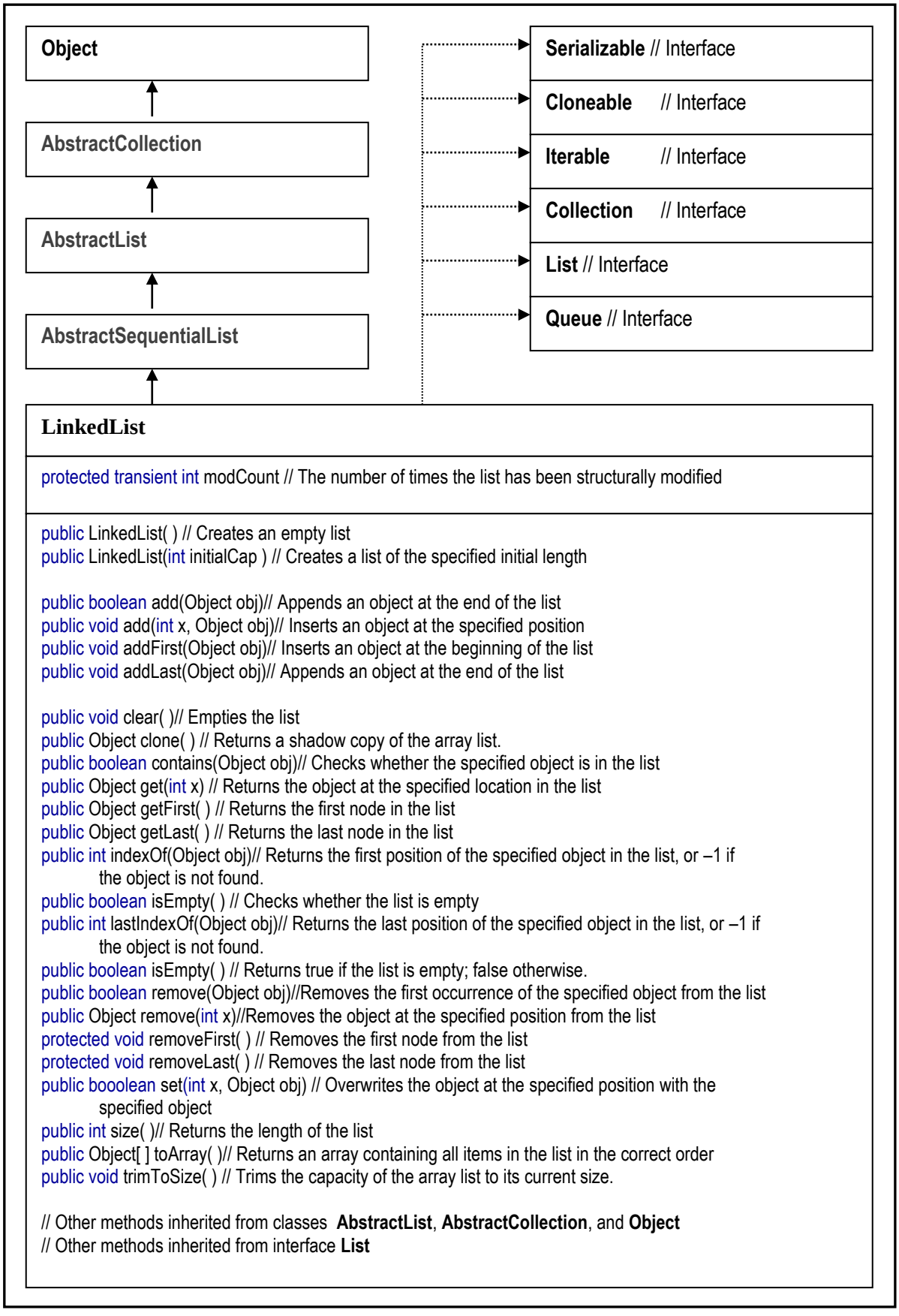
Figure 8-10a: Java Construction of a Linked-list for LibraryPatron Instances

```
// Code snippet for constructing a linked-list of LibraryPatron instances
package adtDemo;
import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.

public class LibraryPatron
{
        protected int pNumber; protected String pName, pMajor;
        private String pStatus;

        // Methods of the instance class would follow this point
}
```

```
// Code snippet for constructing a linked-list of LibraryPatron instances
package adtDemo;
import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.

public class PatronNode
{
        LibraryPatron nInfo;
        PatronNode nNext;

        // Methods of the node class would follow this point
};
```

```
// Code snippet for constructing a linked-list of LibraryPatron instances
package adtDemo;
import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.

public class PatronL
{
        PatronNode nFirst, nLast;
        int length;

        // Methods of the linked list class would follow this point
}
```

Figure 8-10b: Abbreviated UML Diagram for the LinkedList Class



8.4.2 Linked List in C++

Due to C++'s strong support of pointers, the language facilitates the creation of linked lists in a much more straightforward and elegant way than Java does. Let us again consider the **LibraryPatron** class of figure 8-3. To create a linked-list of **LibraryPatron** objects, we may use a structure to represent the node, and then construct the linked-list of nodes. Figure 8-11 illustrates C++ code snippets to represent this approach. As in the case of the previous subsection, the figure here omits the code snippet for the driver class; it is assumed that you understand that this would be required as well. Finally, notice that the C++ code requires fewer classes than the Java code. In the figure, two classes are used; the code could actually packed in a single program file, but in the interest of flexibility and clarity, two classes are recommended as shown.

Figure 8-11: C++ Construction of a Linked-list for LibraryPatron Instances

```
// Code snippet for constructing a linked-list of LibraryPatron instances
#include <cstdlib>   #include <iostream>
#include<ctype.h>  #include<string.h>
using namespace std;

class LibraryPatron
{
  private: // Private data items
  string pStatus;
  protected: // Protected data items
  int pNumber;
  string pName, pMajor;

  // Public prototypes of member functions would follow
  // . . .
} // End of LibraryPatron Declaration

// Specification of member functions would follow . . .
```

```
// Code snippet for constructing a linked-list of LibraryPatron instances
#include <cstdlib>   #include <iostream>
#include<ctype.h>  #include<string.h>
using namespace std;

struct PatronNode
{
  LibraryPatron nInfo;
  PatronNode* nNext;
};

public class PatronL
{
  protected:
  PatronNode nFirst, nLast;
  int length;

 // Member functions of the linked list class would follow this point
}
```

8.5 Other ADTs

Having looked at how classes, array-lists, vectors, and linked-lists are constructed in [two] different programming languages, the next logical step is to conduct a similar study for each of the ADTs mentioned in the introduction, and then expand the inquiry to other languages. Obviously, you should clearly see where this could be going: We could have one section per ADT, and this lecture would extend for several pages. While this exercise is very tempting and would no doubt be quite enlightening, in the interest of expediency, and on the basis of your previously acquired knowledge and skills in this and earlier Computer Science courses, you are encouraged to engage in this exploration on your own.

Exercise 2: Identify a language (an OOPL) that you would like to learn. Examine how the various ADTs mentioned in section 8.1 are supported in the language, and set up comparison grids comparing the new language with one that you are familiar with.

8.6 Summary and Concluding Remarks

Here are the salient points of this lecture:

- An ADT is a programming construct with a defined set of data items, and a set of possible operations on those data items.
- Common ADTs that appear in contemporary programming are dynamic lists, linked lists, stacks, queues, binary trees, binary search trees, heaps, B-trees, hash tables, and graphs.
- Common sort algorithms that are discussed in contemporary programming are straight-selection-sort, exchange-selection-sort, insertion-sort, bubble –sort, quick-sort, merge-sort, tree-sort (as in binary search tree), and heap-sort.
- The starting point for probing how a language supports ADTs is to look at how the language supports classes, dynamic lists, and various other ADTs.

The next lecture will build on the discussion here by focusing on support for OOP in contemporary programming languages.

8.7 Recommended Readings

[Pratt & Zelkowitz 2001] Pratt, Terrence W. and Marvin V. Zelkowits. 2001. *Programming Languages: Design and Implementation* 4th Edition. Upper Saddle River, NJ: Prentice Hall. See chapters 6 & 10.

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapter 11.

[Webber 2003] Webber, Adam B. 2003. *Modern Programming Languages: A Practical Introduction*. Wilsonville, Oregon: Franklin, Beedle & Associates. See chapters 12, 14, 15 & 16.