Computer Science II

Professor Tom Ellman
Lecture 9
Today’s Topics

• Data encapsulation.
• Visibility.
• Inheritance.
Data Encapsulation

• An instance of a class holds some data in instance variables.
• The class provides methods for manipulating the data.
• The data may only be read or modified using the prescribed methods.
• These rules maintains the integrity of the data.
FlexibleArray Class

• Similar to a Java array.
• All entries are initialized to zero.
• Array will automatically grow to accommodate new entries.
• Array index may be any integer.
Behavior v. Implementation

• Behavior:
  – A **FlexibleArray** looks like an infinite array.
  – Indexes range from $-\infty$ to $\infty$.
  – All entries are initially zero.

• Implementation:
  – Let low…high-1 be the range of indexes into which values have been stored by the user.
  – Use a Java array to hold all entries in the range low…high-1.
  – All entries outside this range must be zero.
Behavior

<table>
<thead>
<tr>
<th></th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>low</td>
<td>0</td>
</tr>
<tr>
<td>low+1</td>
<td>1</td>
</tr>
<tr>
<td>low+2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>high-1</td>
<td>high-low-1</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Implementation
package flexiblearray;

public class FlexibleArray {

    private static final int DEFAULT_INITIAL_SIZE = 10;

    protected int low, high;

    protected int[] contents;

    // ... Omitted ...

}
public FlexibleArray() {
    this(0, DEFAULT_INITIAL_SIZE);
}
public FlexibleArray(int low, int high) {
    this.low = low;
    this.high = high;
    int size = high - low;
    contents = new int[size];
    for (int i = 0; i < size; i++) {
        contents[i] = 0;
    }
}

These constructors create an integer array of length initHigh-
initLow, with all its entries set to zero. They also initialize the
range [low...high-1] of explicitly stored values to [initLow
... initHigh-1].
What is this?

• A Java keyword.

• May be used in body of any method.

• Refers to object on which method was invoked.
public FlexibleArray(int low, int high) {
    this.low = low;
    // Omitted ...
}
public void store(int value, int location) {
    if (location >= high) {
        reSize(low, location + 1);
    } else if (location < low) {
        reSize(location, high);
    }
    contents[location - low] = value;
}

Method for explicitly storing values. If the location lies within the explicitly stored range, then the value is simply assigned to the appropriate array element. If the location lies outside the explicitly stored range, the method calls resize with parameters indicating the new range of values to be stored explicitly, and then assigns the value to the appropriate array element.
public int retrieve(int location) {
    if (location < low || location >= high) {
        return 0;
    } else {
        return contents[location - low];
    }
}

Method for retrieving. If the location lies within the explicitly stored range, then the value is found in the appropriate array element. If the location lies outside the explicitly stored range, its value must be zero.
protected void reSize(int newLow, int newHigh) {
    int[] newContents = new int[newHigh - newLow];
    for (int i = newLow; i < low; i++) {
        newContents[i - newLow] = 0;
    }
    for (int i = low; i < high; i++) {
        newContents[i - newLow] = contents[i - low];
    }
    for (int i = high; i < newHigh; i++) {
        newContents[i - newLow] = 0;
    }
    low = newLow;
    high = newHigh;
    contents = newContents;
}

Method for expanding the range of explicitly stored values. First create a new array of the appropriate size: **newHigh-newLow**. Then put zeros into the range **newLow..low-1**. Then copy previously stored values into the range **low..high-1**. Then put zeros into the range **high..newHigh-1**. Finally, update the instance variables **low**, **high** and **contents**.
Public Interface to FlexibleArray

DEFAULT_INITIAL_SIZE

low  reSize(…)

high  contents

store(…)

retrieve(…)

Visibility in **FlexibleArray**

- **Public:** Can be accessed by the methods of any class.
  - `FlexibleArray()`
  - `FlexibleArray(int initLow, int initHigh)`
  - `store(int value, int location)`
  - `retrieve(int location)`

- **Protected:** Can only be accessed by the methods of the `FlexibleArray` class, or classes in the same package as the `FlexibleArray` class, or of classes that extend the `FlexibleArray` class.
  - `protected int low, high;`
  - `protected int[] contents;`
  - `protected void resize(int newLow, int newHigh)`

- **Private:** Can only be accessed by the methods of the `FlexibleArray` class.
  - `private static final int DEFAULT_INITIAL_SIZE;`
import flex.*;

public class VisibilityDemo {

    public static void main(String[] args) {

        // Legal:
        FlexibleArray flexibleArray = new FlexibleArray();
        flexibleArray.store(25, 5);
        System.out.println(flexibleArray.retrieve(5));

        // Illegal:
        System.out.println(FlexibleArray.DEFAULT_INITIAL_SIZE);
        flexibleArray.contents[5] = 100;
        System.out.println(flexibleArray.high);
        System.out.println(flexibleArray.low);
    }
}
FlexibleArrayDeclare Class

• Extends the FlexibleArray class.

• Allows user to declare his/her intentions regarding the range of indexes to be used.
These two **FlexibleArrayDeclare** constructors simply invoke the corresponding **FlexibleArray** constructors. We could omit the definition of **FlexibleArrayDeclare()**, but not the definition of **FlexibleArrayDeclare(int, int)**.
public void declare(int newLow, int newHigh) {
    newLow = Math.min(newLow, low);
    newHigh = Math.max(newHigh, high);
    reSize(newLow, newHigh);
}

Method to allow the user to state his/her intentions regarding the range in indexes to be used. It just calls the `reSize` method, inherited from the `FlexibleArray` class, with the new range of indexes.
Public Interface to FlexibleArray

**Declaration**

- **store(...)**
- **retrieve(...)**
- **declare(...)**

**Attributes**

- **DEFAULT INITIAL SIZE**
- **low**
- **reSize(...)**
- **high**
- **contents**
Visibility in
FlexibleArrayDeclare

- **Public**: Can be accessed by the methods of any class.
  - FlexibleArrayDeclare()
  - FlexibleArrayDeclare(int initLow, int initHigh)
  - public void declare(int newLow, int newHigh)

- **Protected**: Can only be accessed by the methods of the **FlexibleArrayDeclare** class, or of classes in the same package as the FlexibleArrayDeclare class, or of classes that extend the FlexibleArrayDeclare class. (Nothing protected.)

- **Private**: Can only be accessed by the methods of the FlexibleArrayDeclare class. (Nothing private.)
FlexibleArraySum Class

• Extends the FlexibleArray class.

• Includes a method that returns the sum of all elements in the array.
1st FlexibleArraySum Implementation

• Define an instance variable `total` to hold the sum of all the stored values.
• Extend the `store(...)` method to update the `total` variable.
• Define a `sum(...)` method to return the value of the `total` variable.
package flexiblearray;

public class FlexibleArraySum extends FlexibleArray {

    private int total;

    public FlexibleArraySum() {
        super();
        total = 0;
    }

    public FlexibleArraySum(int low, int high) {
        super(low, high);
        total = 0;
    }

    // . . . Omitted . . .
}

Instance variable to keep track of the running total.

Constructors initialize the running total after invoking the corresponding **FlexibleArray** constructor.
public void store(int newValue, int location) {

    int oldValue = retrieve(location);
    super.store(newValue, location);

    total = total - oldValue + newValue;
}

public int sum() {
    return total;
}

Invoke the store method of the FlexibleArray class.

Update the running total.

Method to return the running total.
Public Interface to FlexibleArraySum

DEFAULT_INITIAL_SIZE

low reSize(…)

high contents total

store(…)

retrieve(…)

sum()
Visibility in `FlexibleArraySum`

- **Public**: Can be accessed by the methods of any class.
  - public `FlexibleArraySum()`
  - public `FlexibleArraySum(int initLow, int initHigh)`
  - public `void store(int newValue, int location)`
  - public `int sum()`

- **Protected**: Can only be accessed by the methods of the `FlexibleArraySum` class or methods in the same package as the `FlexibleArraySum` class, or classes that extend the `FlexibleArraySum` class. (Nothing protected.)

- **Private**: Can only be accessed by the methods of the `FlexibleArraySum` class.
  - private `int total;`
2nd **FlexibleArraySum** Implementation

- Don’t keep track of the running total.
- No need to extend the `store(...)` method to update the running total.
- Define the `sum(...)` method to compute and return the sum of all the stored elements.
package flexiblearray;

public class FlexibleArraySumAlt extends FlexibleArray {

    Since we will compute the sum when it is needed, we don’t need to define an instance variable to store the sum.

    public FlexibleArraySumAlt() {
        super();
    }

    public FlexibleArraySumAlt(int initLow, int initHigh) {
        super(initLow, initHigh);
    }

    // . . . Omitted . . .
}
Method to compute the current sum of all the stored element, \textit{on the fly}.
Public Interface to FlexibleArraySumAlt

- \texttt{DEFAULT\_INITIAL\_SIZE}
- \texttt{store(\ldots)}
- \texttt{retrieve(\ldots)}
- \texttt{sum()}
- \texttt{low reSize(\ldots)}
- \texttt{high contents}
Visibility in FlexibleArraySumAlt

- **Public**: Can be accessed by the methods of any class.
  - public FlexibleArraySumAlt()
  - public FlexibleArraySumAlt(int initLow, int initHigh)
  - public int sum()

- **Protected**: Can only be accessed by the methods of the FlexibleArraySumAlt class or methods in the same package as the FlexibleArraySumAlt class, or classes that extend the FlexibleArraySumAlt class. (Nothing protected.)

- **Private**: Can only be accessed by the methods of the FlexibleArraySumAlt class. (Nothing private.)
Punchline!

• The public interfaces to `FlexibleArraySum` and `FlexibleArraySumAlt` are the same.
• Suppose an application works correctly using `FlexibleArraySum`.
• Suppose we replace `FlexibleArraySum` with `FlexibleArraySumAlt`, (but keep the class name the same).
• The application will continue to work correctly.
## Summary of Visibility Rules

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>&lt;none&gt;</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>protected</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>public</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Scenario #1

- Macrosoft, Inc. sells a **FlexibleArraySum** class program to Luckheed, Inc. and Boing, Inc. who are collaborating on the F-222 fighter.

- Should Macrosoft make use of the Java visibility mechanism to deny Luckheed and Boing access to the **FlexibleArraySum** implementation?
Scenario #2

- Larry, Curly and Moe meet in a tree house each day after school to program together. Moe is implementing a `FlexibleArraySum` class to share with Larry and Curly to use in their new video game “Immortal Combat”.

- Should Moe make use off the Java visibility mechanism to deny Larry and Curly access to the `FlexibleArraySum` implementation?
Scenario #3

• Harry Horatio Hacker is implementing a `FlexibleArraySum` class which he himself will use in his new virus program “RobinHood”, which will deduct one penny from each bank transaction and donate it to UNICEF.

• Should Harry make use off the Java visibility mechanism to deny himself access to the `FlexibleArraySum` implementation?
Who should use the Java visibility mechanism?

- Macrosoft? ........................................Yes!
- Moe? .............................................Yes!
- Harry Horatio Hacker? ............Yes!
Uses of the Java Visibility Mechanism

- To protect the implementation from adversaries. (Macrosoft versus Luckheed and Boing.)
- To protect the implementation from friendly but, shall we say “inept”, users. (Moe versus Larry and Curly.)
- To protect an ace hacker’s implementation from his/her most dangerous enemy –him/herself. (Harry versus himself.)
Benefits of Controlling Visibility

• Correctness: By forcing users of the class to work through the public access functions, consistency of the data will be maintained.

• Maintainability: When the class needs to be modified, e.g., to add more features, the modifications can be made in a way that preserves functionality of programs currently using the class.