Arrays

C  S  C  I  0  1  5  0

[0] [1] [2] [3] [4] [5] [6] [7]
Topics

• Purpose
• Syntax
• Multi-Dimensional Arrays
• Array Lists
• Generics
Why Use Arrays? (1/2)

- Only been studying variables that hold references to single objects
- But what about holding lots of data? Many programs need to keep track of hundreds/thousands of data instances
- Want to hold arbitrary number of objects with single reference – represents a collection of elements
  - allows for simple communication to multiple elements
- Arrays are the simplest data structure or collection - we’ll cover lists, queues, stacks, trees, and hash tables
Why Use Arrays? (2/2)

- Arrays allow instances of particular type to be “packaged” together and accessed as group

- What if there are 10 instances of Barbaloot
  - store all Barbaloots in array for easy access (to tell them to eat Truffula fruit!)

- Arrays are ordered - helpful when wanting to store or access instances in particular order, e.g., alphabetically
Fibonacci Sequence (1/2)

- Pervasive in nature, along with golden ratio phi = 1.618, logarithmic spiral, etc.

- Tracking first 20 numbers in the Fibonacci Sequence:
  - sequence begins with 0 and 1; successive numbers determined by adding previous two numbers
  - 0, 1, 0+1=1, 1+1=2, 2+1=3, …
Fibonacci Sequence (2/2)

● Beginning of sequence:

```
0  1  1  2  3  5  8  13  21  34  55  89
```

● Could use instance variables...

```
public class FibSequence {
    private int _firstNum,_secondNum,... _twentiethNum;
}
```

● Gets tiresome and isn't flexible

○ try making sequence with forty numbers, thousand?

○ in algebra, there's subscript notation: $F_0, F_1, F_2, \ldots$
Arrays (1/4)

- Arrays store specified, constant number of data elements of the same type – our first **homogeneous** collection
  - each element must be same type or subclass of same type (polymorphism)
- Arrays are special in Java
  - special syntax to access array elements:
    - `_studentArray[index]`
  - neither base type nor class, but Java **construct**
    - use `new` to initialize an array (even though it’s not a class!)
Arrays (2/4)

- Arrays only hold elements of specified type
  - When declaring arrays, state type of object it stores:
    - base type
    - class
    - sub-arrays (for multi-dimensional arrays – soon)
    - or for max polymorphic flexibility, interface
  - Type can even be `java.lang.Object` to store any instance, but that isn’t useful: wouldn’t take advantage of compiler’s type-checking
Arrays (3/4)

- Every array element is object reference, subarray, or base type. What real-world objects can be organized by arrays?
  - number of electoral votes by state
  - streets in Manhattan
  - strings representing names or Banner ID's of people in a course

- Elements ordered sequentially by numerical index
  - in math, use subscript notation, i.e., $A_0, A_1, A_2, \ldots A_{n-1}$
  - In Java, use index inside brackets, i.e., array[0], array[1],...array[n-1]
Arrays (4/4)

- Arrays store objects in numbered slots
  - for array of size n, first index is always 0, last index is always n-1
- Common graphical representations of arrays:
Array Examples (1/2)

- Houses on a Neighborhood Street
  - array size: 8
  - array index: house number
  - element type: house

Note: arrays don’t need to be full (e.g., no house 0, 4, or 7)
Array Examples (2/2)

- Sunlab Computers
  - array size: 72
  - array index: node number
  - element type: computer

Note: Could be modeled as a 2D array (see slide 31)
Java’s Syntax for Arrays (1/6)

Declaration:

```java
<visibility>  <type>[][]  <array-name>;
```

- `<visibility>` denotes the accessibility, i.e. public, private, etc. – we’ve seen this before!
- `<type>` denotes data type array holds: can be class, base type, interface, or another array (nested arrays)
Java’s Syntax for Arrays (2/6)

Declaration and instantiation example:

```java
private Colorable[] _myColorables;
...
_myColorables = new Colorable[4];
```

- unlike some other programming languages, **size of array doesn’t get specified in declaration, but in initialization**
  - also no reserved word “array” - [] brackets suffice
Java’s Syntax for Arrays (3/6)

Initialization:

\(<\text{type}>[]\) <array-name> = new <\text{type}>[<\text{size}>];

- syntax for declaring arrays as local variables – can be instance variables too!
- \(<\text{size}>\) must be integer value greater than 0; indices range from 0 to \(<\text{size}>-1\)
- we use \text{new} here, but because arrays are built-in construct
- in Java, we have special syntax and \text{new} doesn't invoke constructor like it would for an instance of a class
- note: only array is initialized, not elements of array; all references are set to \text{null}, 0 for ints, false for booleans, etc.
Java’s Syntax for Arrays (4/6)

- Arrays can be local variables, so they can get declared and initialized in single statement - just like objects and base types:

  ```java
  Colorable[] otherColorables = new Colorable[5];
  ```

- Arrays can also be instance variables, which get declared and then initialized separately in constructor:

  ```java
  private Colorable[] _myColorables;
  ...
  ```

  ```java
  //in constructor of class that contains the array
  _myColorables = new Colorable[10];
  ```
Java’s Syntax for Arrays (5/6)

- Accessing individual elements:
  \[ \text{array-name}[\text{index}] \]
  - index must be integer between 0 and (array_size-1)
  - result is variable stored at that index
  - if \( \text{index} \geq \text{size}, \text{or} \ < 0 \),
    \( \text{ArrayIndexOutOfBoundsException} \) gets thrown
  - also useful to check for uninitialized entries with
    \( \text{ref} \neq \text{null} \) -- See slide 46

- Think of student\[i\] as the “name” of that particular student (like student\(_i\)) – avoids having to name each individual element in collection with unique name
Clicker Question 1

Which of the following is the correct way to declare and initialize an array of \texttt{ints} named \texttt{greenEggs}, of size 5?

A. \texttt{int greenEggs = new array(5);}
B. \texttt{int[] greenEggs = new array(5);}
C. \texttt{int[] greenEggs = new int[5];}
D. \texttt{int[5] greenEggs = new int[];}

Java’s Syntax for Arrays (6/6)

- Anywhere there’s a variable or constant, an array element will work. For example, in your PaneOrganizer *:

```java
// initialize first element of array Colorables to be Ball
myColorables[0] = new Ball();

// send a message to 3rd element
myColorables[2].setColor(javafx.scene.paint.Color.RED);

// assign fourth element to a local variable
Colorable myColorableVar = myColorables[3];

// pass 5th as a parameter
_myPaintShop.paintRandomColor(myColorables[4]);
```

*Note: You don’t have to use arrays in Cartoon, but if you start having too many objects, you may want to use one. This syntax will be most relevant for post-Cartoon assignments.
Arrays as Parameters (1/3)

- Can pass entire array as parameter by adding array brackets to type inside signature

```java
public int sum(int[] numbers){
    //code to compute sum of elements in the int array
}
```
Arrays as Parameters (2/3)

- How do we determine size of array?
  - arrays have their `length` as a public property (not a method)
  - use special “dot” syntax to determine `length`; here we inquire it, then store it for later

```java
int arrayLength = <array-name>.length;
```
Arrays as Parameters (3/3)

- How does `.length` work in actual code?

```java
public int sum (int[] numbers){ //sum all entries in array
    int total = 0;
    for (int i=0; i < numbers.length; i++){
        total += numbers[i];
    }
    return total;
}
```

Note: `for` loop to traverse through all elements of array
Example: Cats with Hats

Design and implement a cartoon with ten Cats.

When the “Wear Hat” button gets pressed, all Cats should execute `wearHat()` method.
Quick Look at Design

Things we need:
- App class
- PaneOrganizer class
- Cat class
- Private inner ClickHandler class for the button

But once we have all of that ...
- How do we make a button do something for all instances of Cat in sequence?
package DrSeuss;

/* standard setup for the Stage, Scene, etc. has been elided */

public class PaneOrganizer{

    private Cat[] _cats;
    private Button _button;

    public PaneOrganizer(){
        _button = new Button("Wear Hats!");

        //Initialize array
        _cats = new Cat[10];

        //Fill the array with Cats
        for(int i=0; i < _cats.length; i++){
            _cats[i] = new Cat();
        }

        _button.setOnAction(new ClickHandler());
    }

    // PaneOrganizer continued
    // Remember to use private inner classes for handlers!

    private class ClickHandler implements EventHandler<ActionEvent> {
        public void handle(ActionEvent event){
            //loop thru array, telling each cat in turn to wear hats
            for (int i=0; i < _cats.length; i++){
                _cats[i].wearHat();
            }
        }
    }
}
Out-of-Bounds Problems

● Careful about bounds of loops that access arrays!
  o Java throws `ArrayIndexOutOfBoundsException` if index is negative since sequence starts at 0
  o throws `ArrayIndexOutOfBoundsException` if index is $\geq$ array size; remember that array goes from 0 to $n-1$
  o Exceptions typically lead to crashes
    ▪ Java has `catch` keyword which can be used to “catch” and handle exceptions… used in CS16
    ▪ Brief intro for `try` coming at end of this semester
ArrayIndexOutOfBoundsException

- If you try to access an invalid index, you’ll receive “arrayIndexOutOfBoundsException” runtime error

```java
public class PaneOrganizer {

  private Cat _cats[];

  /* This is the constructor! */
  public PaneOrganizer() {
    _cats = new Cat[10]
    this.setupCats();
  }

  public void setupCats() {
    for(int i = 0; i<=10; i++) {
      _cats[i] = new Cat();
    }
  }

}
```

In Terminal:
Error: java.lang.ArrayIndexOutOfBoundsException: 10 at (PaneOrganizer.java:14)
Clicker Question 2

Consider the sum function from slide 22:

```
public int sum (int[] numbers){
    int total = 0;
    for (int i=0; i < numbers.length; i++){
        total += numbers[i];
    }
    return total;
}
```

What if the code read `i <= numbers.length`?

A. It would wrap around and add the value at index 0 again
B. It would reach the last element of the array
C. It would raise an `IndexOutOfBoundsException`
Multi-Dimensional Arrays (1/2)

- Modeling chess board:
  - not linear group of squares
  - more like grid of squares

- Can declare array to be 2 (or more) dimensions, by adding more brackets
  - one pair per dimension
  - 2D: int [][] grid = new int [a][b];
  - 3D: int [[][]][] cube = new int [x][y][z];
  // a, b, x, y, z are ints whose values are set elsewhere
Multi-Dimensional Arrays (2/2)

- Multi-dimensional arrays are array of arrays of...
- Syntax above is for rectangular, cuboid, etc. multi-dimensional arrays
  - since multi-dimensional arrays are just arrays of arrays, it is possible (using different syntax) to have **jagged arrays**, where each sub-array is of different length
  - thus can have “triangle” shaped array
  - don’t use this is CS15; even in CS16 and beyond, it is unlikely you will need this (predominately for scientific/engineering computation)
Two-Dimensional Array Examples

- **Pixel Array**
  - 2D Array size: pxl width by pxl height
  - array indices: x, y
  - element type: RGB color

- **Connect Four**
  - 2D Array size: 6 by 7
  - array indices: row, column
  - element type: checker

- **The Sunlab as a 2D array!**
  - 2D Array size: 10 by 8 (approx.)
  - array indices: row, column
  - element type: computer
Representing Multi-Dimensional arrays (1/2)

- Let’s say we want to represent this grid of numbers:
Representing Multi-Dimensional arrays (2/2)

- How do we want to represent this grid? There are two equally valid options:

Array of rows:

```
1  2  3
4  5  6
7  8  9
```

Array of columns:

```
1  2  3
4  5  6
7  8  9
```
Ways to Think About Array Storage (1/2)

- Multi-dimensional arrays in Java do **not** make a distinction between rows or columns
  - think about 1D array – it doesn’t really matter if we call it a “row” or a “column”
  - can think of arrays as ordered sequences of data stored in contiguous positions in memory - no intrinsic geometry/layout implied
Ways to Think About Array Storage (2/2)

- Two visualizations of two-dimensional array (called array) are equally valid
  - Column of Rows:
  
  - Row of Columns:

  column-major order, i.e., first index is column index (e.g., purple ball is at array[0][2] – column 0, row 2)

  row-major order, i.e., first index is row index (e.g., purple ball is at array[0][2] – row 0, column 2)

- Make sure there’s consistency in the way you index into your 2D array throughout your program!
Clicker Question 3

If we want to access the third row, sixth column slot on a chess board, with column-major order, which indices would we need to supply?

A. chessboard[3][6]
B. chessboard[6][3]
C. chessboard[2][5]
D. chessboard[5][2]
Example: Size of 2D Arrays

```java
public class ArraySize{
    //using rows and columns as indices is arbitrary
    private static final int NUM_ROWS = 10;
    private static final int NUM_COLS = 5;

    public ArraySize(){
        //String is just an arbitrary choice!
        String[][] myArray = new String[NUM_ROWS][NUM_COLS];
        System.out.println("Number of rows = "+NUM_ROWS);
        System.out.println("Number of columns = "+NUM_COLS);
        System.out.println("Size of array = "+this.find2DArraySize(myArray));
    }

    public int find2DArraySize(String[][] array){
        //row major order, column of rows
        int numRows = array.length; //number of entries in column vector, each a row
        int numCols = array[0].length; //element 0 is an array
        return (numRows * numCols);
    }
}
```

array.length gives size of first dimension (you decide whether you want row or column), and
array[0].length gives size of second dimension
Common Array Errors - Watch Out! (1/2)

● Assigning a scalar to an array

```java
int[] myArray = 5;
```

- 5 is not an array
- to initialize array elements must loop over the array and assign values at each index

```java
int[] myArray = new int[20]; //inits array, not elements
for (int i=0; i < myArray.length; i++){
    myArray[i] = 5;
}
```
Assigning arrays to scalars

```
int myInt = myArray;
```

Assigning arrays of different dimension to each other

```
int[] myIntArray = new int[23];
int[][] my2DIntArray = new int[2][34];
myIntArray = my2DIntArray;
```

Never assign arrays of different dimensions or you will become familiar with error:

“Incompatible types: Can’t convert int[] to int[][]”

Similar message for assigning arrays of mismatched type

Take note that Java will automatically resize an array when assigning a smaller array to a larger one
Let’s Make a Board … What Kind?

- Warm-up for Tetris…
- Let’s start with a specification:

  Write a Java program that draws sixty-four squares in a grid-like pattern of alternating colors, much like a checker board. The checker board should be eight squares long and eight squares wide. Additionally, the user should be able to click on a button and change the colors of the board from the traditional red and black to the new & bold colors white and blue.
Quick Look at the Design (1/2)

- Some things we’ve seen before:
  - *java program* – creates a `javafx.stage.Stage`
  - *buttons* – uses `javafx.scene.control.Button`
  - *red, black, white, blue* - `javafx.scene.paint.Color`

- New things:
  - *sixty-four squares* - we know about one square, `Shape.Rectangle`, but 64?
  - *checker board* - let’s make a 2D 8x8 array of squares
  - *row, column* - indices into array

- This sample program has crucial design hints for Tetris. Pay close attention!
Quick Look at Design (2/2)

- What classes will we write?
  - **PaneOrganizer** which creates graphical items and then adds to the Scene Graph
  - **CheckerBoard** which contains a 2D array of **CheckerSquares**
  - **CheckerSquare** which has the ability to toggle its color

- Let's build them bottom-up
Building **CheckerSquares** that Changes Colors

- Stores 2 colors and toggles between them

```java
public class CheckerSquare {
    private Color _currentColor;
    private Color _otherColor;
    private Rectangle _rect;

    public CheckerSquare(Color primaryColor, Color secondaryColor) {
        _rect = new Rectangle();
        _rect.setWidth(Constants.SQR_SIZE);
        _rect.setHeight(Constants.SQR_SIZE);

        _currentColor = primaryColor;
        _otherColor = secondaryColor;
        _rect.setFill(_currentColor);
    }

    public void setLocation(int x, int y) {
        _rect.setX(x);
        _rect.setY(y);
    }

    public void toggleColor() {
        Color temp = _currentColor;
        _currentColor = _otherColor;
        _otherColor = temp;
        _rect.setFill(_currentColor);
    }

    // for board pane to build scene graph
    public Node getNode() {
        return _rect;
    }
}
```

// End of Class
Building **Checkerboard** (1/2)

- Let’s start with standard stuff
  - contains array of `CheckerSquares`
  - all `CheckerSquares` get added to `_root` in `PaneOrganizer`—so it needs accessor for the array

- Row-major or column-major order? For pixel locations or for square (`CheckerSquares`) location, natural to think of `(x, y)` order
  - column-major order corresponds to columns as first index (x coordinate) and rows (y coordinate) as second
Building Checkerboard (2/2)

public class CheckerBoard {
    private CheckerSquare[][] _rects;
    public CheckerBoard(){
        _rects = new CheckerSquare[Constants.NUM_SQRS][Constants.NUM_SQRS];
        for (int col=0; col< Constants.NUM_SQRS; col++){
            //outer for loop through columns
            for (int row=0; row < Constants.NUM_SQRS; row++){
                //nested inner for loop through rows
                CheckerSquare rect; // every other square should be red
                if (((row + col) % 2) == 0){
                    rect = new CheckerSquare(Color.RED, Color.WHITE);
                }else{
                    rect = new CheckerSquare(Color.BLACK, Color.BLUE);
                }
                rect.setLocation(col*Constants.SQR_SIZE, row*Constants.SQR_SIZE);
                _rects[col][row] = rect; //add it to array, now that we have positioned it
            }
            //end of nested for loop
        }
        //end of outer for loop
    }
    public CheckerSquare[][] getRectangles(){
        return _rects;
    }
}

This method returns our array of CheckerSquares so that they can be added to the Scene Graph in P.O.

Andries van Dam © 2016 10/18/16
Updating **CheckerSquares**

- ClickHandler private inner-class sends message to `CheckerBoard` to change color of squares

```java
// in PaneOrganizer Class...
private class ClickHandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent arg0) {
        for (int col=0; col<Constants.NUM_SQRS; col++){
            for (int row=0; row <Constants.NUM_SQRS; row++){
                //local variable points to selected rect
                CheckerSquare rect = _rects[col][row];
                //make sure value of array element isn't null (i.e., array initialized correctly)
                if(rect != null){
                    rect.toggleColor();
                }
            }
        }
    }
}
```

What would happen if we didn't check for `null`? We might get a **NullPointerException**! In this code that couldn't happen, but in general, it's a useful technique.
PaneOrganizer class

```java
public class PaneOrganizer {
    private CheckerBoard _board;
    private BorderPane _root;
    private CheckerSquare[][] _rects;

    public PaneOrganizer()
    {
        _root = new BorderPane();
        _board = new CheckerBoard();
        _rects = _board.getRectangles(); // gets array of
        // graphical squares from the board
        this.setUpBoardPane();
        this.setUpButtonPane();
    }

    private void setUpBoardPane()
    {
        Pane boardPane = new Pane();
        _root.setTop(boardPane);
        for (int col=0; col< Constants.NUM_SQRS; col++) {
            // outer for loop through columns; inner for loop through rows
            for (int row=0; row < Constants.NUM_SQRS; row++) {
                // local variable points to selected rect
                CheckerSquare rect = _rects[col][row];
                // make sure value of array element isn't null
                // (i.e., array initialized correctly)
                if (rect != null) {
                    rect.toggleColor();
                }
            }
        }
    }

    private void setUpButtonPane()
    {
        HBox buttonPane = new HBox();
        _root.setBottom(buttonPane);
        Button button = new Button("Change Color!");
        button.setOnAction(new ClickHandler());
        buttonPane.getChildren().add(button);
        buttonPane.setStyle("-fx-background-color: gray;");
        buttonPane.setAlignment(Pos.CENTER);
    }

    public Pane getRoot()
    {
        return _root;
    }
}
```

private void setUpBoardPane()
{
    Pane boardPane = new Pane();
    _root.setTop(boardPane);
    for (int col=0; col< Constants.NUM_SQRS; col++) {
        // outer for loop through columns; inner for loop through rows
        for (int row=0; row < Constants.NUM_SQRS; row++) {
            boardPane.getChildren().add(_rects[col][row].getNode());
        }
    }
}

private class ClickHandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent arg0) {
        for (int col=0; col< Constants.NUM_SQRS; col++) {
            for (int row=0; row < Constants.NUM_SQRS; row++) {
                // local variable points to selected rect
                CheckerSquare rect = _rects[col][row];
                // make sure value of array element isn't null
                // (i.e., array initialized correctly)
                if (rect != null) {
                    rect.toggleColor();
                }
            }
        }
    }
}
```
java.util.ArrayList (1/2)

- `java.util.ArrayList`s, like arrays, hold references to many objects of same data type
- Another kind of collection, also using an index, but much easier management of making changes to array at runtime
- As name implies, it has properties of both arrays and lists (covered later)
- Differences with arrays:
  - don’t need to be initialized with size - can hold an arbitrary number of references
  - are Java classes, so have methods
java.util.ArrayList (2/2)

- Why use them instead of arrays?
  - when number of elements to be held is unknown
  - making array too small leads to bugs or crashes
  - making array too large is inefficient, takes up more memory than necessary
  - handles update dynamics (shifting elements in memory) for you

- Why use arrays instead of array lists?
  - want something simple
  - want to use less memory (when expect both array and array list to hold same number of elements)
  - want faster operations
Adding and Deleting in Arrays (1/2)

- When adding at particular index, all other elements falling in and after that index must get shifted right by programmer (their indices are incremented by 1)
Adding and **Deleting** in Arrays (2/2)

- When **deleting** from particular index, all other elements falling in and after that index must get **shifted left** by programmer to fill the newly opened space (index decremented by 1)
What can ArrayLists hold? (1/2)

- ArrayLists can hold any object!
- Every class implicitly extends Object
  - every object “is an” Object
  - methods of Object you can usefully redefine (i.e., override):
    - boolean equals (Object obj): checks for equality
    - void finalize(): used in garbage collection
    - String toString(): returns object’s “state” as string, could be used to print all instance variables’ values
What can **ArrayLists** hold? (2/2)

- **Upside**: **ArrayLists** store things as **Object**-maximum polymorphic flexibility
  - since **everything** is an **Object**, **ArrayLists** can hold instances of any and every class
  - easy adding/removing **anything**

- **Downside**: **ArrayLists** only store **Objects**:
  - only methods available are trivial ones of **Object** itself: `equals()`, `toString()`, and `finalize()`
  - want homogenous collection to store only objects of particular type AND have the compiler do type-checking for that type to enforce homogeneity
Generics! (1/2)

- Generics allow us to write collection class A to hold instances of another class B, without regard for what that class B is.
- This is the constructor of the generic `ArrayList`:

  ```java
  public ArrayList<ElementType>()
  ```

- Already seen use of generics to specialize implementation of `EventHandler` interface
  - Replace code inside `<>` with a subclass of `Event`, like `ActionEvent`
Generics! (2/2)

- Can use “generics” to implement collection class without knowing specific type of object that collection wants to store
  - example: Java’s `ArrayList`’s class file defines array list of `ElementType`, but when `ArrayLists` get declared, type must get specified (e.g., an `ArrayList` of `Truffulas` for the Lorax)
  - Java replaces `ElementType` with `Truffulas` in return types and parameters of any `ArrayList` method
  - Still keep the literal `< >` brackets wherever generics get used!

- Generics allow for flexibility to use any type while still having compiler type checking
java.util.ArrayList Methods (1/6)

//Note: only most important methods shown; see JavaDocs for
//full class
//Note: literal use of < and >

public ArrayList<ElementType>()
//one of the many constructors for ArrayList class – specialize
//it by providing ElementType, just as Array has the type it
//stores. Note: < and > are literal – think of them as “of type”

public ElementType get(int index)
//returns the object of type ElementType at that index
java.util.ArrayList Methods (2/6)

public void add(int index, ElementType element)
/* inserts the specified element at the specified position in this ArrayList; just as with arrays, causes indices of elements “to the right” to be incremented */

public boolean add(ElementType element)
//inserts specified element at end of ArrayList

public ElementType remove(int index)
//removes the ElementType at given index
java.util.ArrayList Methods (3/6)

public int size()
//returns number of elements stored in ArrayList

public boolean isEmpty()
//returns true if the ArrayList contains zero elements; false
//otherwise
java.util.ArrayList Methods (4/6)

- ArrayLists also have methods which access elements through search (as opposed to using an index)
  - these methods take parameter of type `Object`
  - But should never pass in (or get back) anything besides `ElementType`
    - using polymorphism here **not for generality** but with **generics mechanism** in order to get compile-time type checking
java.util.ArrayList Methods (5/6)

public int indexOf(Object elem)
//finds first occurrence of specified element

public boolean contains(Object elem)
//return true if ArrayList contains specified element

public boolean remove(Object elem)
//remove first occurrence of specified element
java.util.ArrayList Methods (6/6)

- Some other ArrayList notes...
  - can add object in particular slot or append to end
  - can retrieve object stored at particular index and perform operations on it
  - can use for loop to access all objects in ArrayList
  - shifting elements for adding/deleting from ArrayList is done automagically by Java!
    - Beware that indices past an insertion/deletion will increment/decrement respectively
Summary of ArrayLists (1/2)

- More flexible than arrays for insertion/deletion
  - dynamically shifting elements and adjusting size in response to insert/delete is all done automagically

- Useful methods:
  - get(int index), add(ElementType element)
  - add(int index, ElementType element)
  - indexOf(ElementType elem) //search
  - remove (int index), size(), isEmpty()
Summary of ArrayLists (2/2)

- Can hold heterogeneous collection of any kind of `Object`; want homogeneous collections...

- **Specialize** the ArrayList type by adding “generic” specification to a declaration or instantiation - thereby specifying two classes in one statement: the collection and the type of object it will hold and return

```
ArrayList<Truffula> truffulas = new ArrayList<Truffula>();
```

Now `truffulas` will only hold instances of type `Truffula`

- Remember to use literal `<>` for specialized type!
Clicker Question 4

Which of the following uses an `ArrayList` correctly?

A. `ArrayList<String> seussQuotes = new ArrayList<String>();
   String x = seussQuotes.get(1);`

B. `ArrayList<ElementType> seussQuotes = new ArrayList;
   String x = seussQuotes[0];`

C. `ArrayList<Strings> seussQuotes = new ArrayList<ElementType>();
   String x = seussQuotes.first();`

D. `ArrayList<String> seussQuotes = new ArrayList<String>
   String x = seussQuotes.get(0);`
Example (1/6) Lorax vs. Once-ler

```java
public class TruffulaForest{
    /* TruffulaForest is a “wrapper” for an ArrayList that augments the
       functionality of an ArrayList with, for example, the code that adds and
       removes items from the scenegraph. Thus, it provides controlled access to the
       contained ArrayList.
       To declare ArrayList, must specify type of object ArrayList stores.
       Replace all occurrences of ElementType with Truffula, including where
       ElementType occurs in literal <> brackets.*/
    private ArrayList<Truffula> _truffulas;

    public TruffulaForest(){
        //ArrayList Initialization- note literal <>
        _truffulas = new ArrayList<Truffula>();
        for (int i=0; i<500; i++){
            //Add a Truffula at end in each pass
            _truffulas.add(new Truffula());
            //scenegraph code elided
        }
    }
}
```

//class definition continued on next slide
Example (2/6) Lorax vs. Once-ler

//Adds a new Truffula at the end
public void plantTruffula(Truffula tree){
    _truffulas.add(tree);
    //scenegraph code elided
}

//If there are still truffulas, chop down and remove the first one
public boolean chopDownTruffula(){
    if (!_truffulas.isEmpty()){
        _truffulas.remove(0);
        //scenegraph code elided
        return true;
    }
    return false;
}
} //End of Class
Example (3/6) Lorax vs Once-ler

- `<Truffula>` indicates use of Java generics
  - Ensures only Truffula instances can be stored and retrieved from this ArrayList
- In TruffulaForest’s constructor, adding a new Truffula works:
  
  ```java
  _truffulas.add(new Truffula());
  ```
- However, adding another type to ArrayList of Truffulas will fail:
  
  ```java
  _truffulas.add(5)
  ```
  
  - “The method add(Truffula) in the type ArrayList<Truffula> is not applicable for the arguments (int)”
public class Lorax {
    private TruffulaForest _truffulaForest;
    private Truffula _lastTree;

    public Lorax(TruffulaForest trees){
        /* Lorax does not contain the TruffulaForest;
            * rather it is associated with this peer object */
        _truffulaForest = trees;
    }

    // Method to add numTrees Truffulas at the end of the current _truffulaForest
    public void plantTrees(int numTrees){
        System.out.println(“I am the Lorax. I speak for the trees.”);
        for(int i = 0; i < numTrees; i++) {
            _truffulaForest.plantTruffula(new Truffula());
        }
    }
}
Example (5/6) Lorax vs. Once-ler

```java
public class Onceler {
    private TruffulaForest _truffulaForest;
    private ArrayList<Thneed> _thneeds;

    public Onceler(TruffulaForest trees) {
        _truffulaForest = trees; //association
        _thneeds = new ArrayList<Thneed>(); //containment
    }

    public void makeThneeds(int numThneeds) {
        System.out.println("A thneed is what you need!");
        for(int i = 0; i < numThneeds; i++) {
            if(_truffulaForest.chopDownTruffula()) {
                _thneeds.add(new Thneed());
            } else {
                break;
            }
        }
    }
}
```

Andries van Dam © 2016 10/18/16
public class TruffulaLand {
    private TruffulaForest _truffulaForest;
    private Lorax _lorax;
    private Onceler _onceler;

    public TruffulaLand(){
        _truffulaForest = new TruffulaForest();
        _lorax = new Lorax(_truffulaForest);
        _onceler = new Onceler(_truffulaForest);
    }
    public void helpBarbaloots(){
        _lorax.plantTrees(10);
    }
    public void hurtBarbaloots(){
        _onceler.manufactureThneeds(10);
    }
} // End of Class

Example (6/6) Lorax vs. Once-ler

UNLESS someone like you cares a whole awful lot,
nothing is going to get better.
It's not.
—The Lorax
for vs. for-each loop (1/4)

- Consider this for loop:

```java
//Somewhere in the WhoVille class
//note: _whos is an ArrayList<Who>
for (int i=0; i<_whos.size(); i++){
    if (i % 2 == 0){ //if index ‘i’ is even
        //sing() is defined in Who
        _whos.get(i).sing();
    }
}
```

- Only want to call `sing()` on elements at even indices, but for-each loop wouldn’t work
  - we don’t execute `sing()` on every element in the `ArrayList`; we only care about elements at specific indices
for vs. for-each loop (2/4)

- Intended to simplify most common form of iteration, when loop body gets applied to each member of collection

- How do for-each loop and for loops differ?
  - for loop give access to index where item is stored
  - for-each loops don’t have direct access to index, but can easily access item (see next example)
**for vs. for-each loop (3/4)**

- **for** loops were extended in Java 5 to improve iteration
  - commonly called **for-each** or **for-in** loop
  - <> here **NOT** literal, i.e., not for generics

```
for (<type> <var>: <structure>){
    <loop body>
}
```

- **<type>**: class of objects stored in the **<structure>**
- **<var>**: name of current element—holds each successive element in turn
- **<structure>**: data structure (array or other collection) to iterate through
for-each vs. for loop (4/4)

- If every element needs to be iterated and loop body doesn’t need element index, for-each loops suffice:

  //Instead of only even-numbered whos, now everyone must sing!
  for (Who currWho: _whos){
      //notice how don’t need to use index to get who from ArrayList
      currWho.sing();
  }

- Great advantage of for-each loops is that they don’t raise ArrayIndexOutOfBoundsExceptions! Why?
  - Java does the indexing for you!
Understanding Mainline (and optional params)

package Demos.Mainline;
//Dummy class to understand Mainline
public class App{
    public App(){
        //constructor elided
    }

    //Standard mainline function
    public static void main(String[] argv){
        System.out.println(argv[0]);
        System.out.println(argv[1]);
        new App();
    }
}

● You’ve seen the main line before, but let’s talk about its parameters

● If we type this in a terminal:
  java Demos.Mainline.App Hello CS15

● Output says:
  Hello
  CS15

● If two or more arguments get passed into main line, compiler would raise
  ArrayIndexOutOfBoundsException!

● Why? Array’s argv’s size is exactly equal to the number of parameters passed to main line

● You won’t need to use main line parameters in CS15, but it’s a good thing to know!
Announcements

• Cartoon deadlines
  o check-ins (10/13-10/15)
  o early handin (10/18) Tuesday 11:59PM
  o on-time handin Thursday (10/20) 10:00PM
  o late handin Friday (10/21) 10:00PM
• 10% deduction if code doesn’t compile
• Don’t copy the lecture demo!
• The sign up for hours will begin 5 minutes before hours start. When hours officially start, this list will be randomized
  o this is to discourage the long pre-line before hours start
• We highly recommend attending lab 5 in person to get eclipse set up with the TAs present, as there can be issues
  o you are expected to use eclipse to write code from here on out