**Arrays**

<table>
<thead>
<tr>
<th>C</th>
<th>S</th>
<th>C</th>
<th>I</th>
<th>0</th>
<th>1</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
</tr>
</tbody>
</table>

**Topics**
- Purpose
- Syntax
- Multi-Dimensional Arrays
- Array Lists
- Generics

**Why Use Arrays? (1/2)**
- We've been studying used variables that hold references only to single objects
- But what if we want to hold lots of data? Many programs need to keep track of hundreds (or hundreds of thousands) of data pieces
- Want to hold arbitrary number of objects with single reference
  - Represents *collection* of *elements*
  - Can send messages to multiple elements much more easily
- 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 you to "package" together instances of a particular type that you need to access as a group
- For example, say we have 10 instances of *Jock*
  - You can store all your *Jocks* in an array so they’re easy to access when you want to tell all 10 *Jocks* to throw a basketball!
- Arrays are *ordered*, so they’re especially helpful when you want to store or access instances in a particular order, e.g., alphabetically
Fibonacci Sequence (1/2)  
- Occurs pervasively in nature, along with golden ratio $\phi = 1.618$, logarithmic spiral, etc.
- Suppose we want to keep track of the first 20 numbers in the Fibonacci Sequence
  - Sequence begins with 0 and 1; successive numbers determined by adding previous two numbers
  - Third: $0+1=1$; fourth: $1+1=2$; etc.

Fibonacci Sequence (2/2)  
- Beginning of sequence looks like this:
  
<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- We could do it with instance variables...
  ```java
  public class FibSequence {
    private int _firstNum, _secondNum, _twentiethNum;
  }
  ```
- This gets tiresome and is not very flexible
  - Try making a sequence with forty numbers, let alone a thousand
  - In algebra, we'd use subscript notation: $F_0, F_1, F_2, \ldots$

Arrays (1/4)  
- Arrays store a 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
  - Provide special syntax to access array elements:
    ```java
    _studentArray[index]
    ```
  - Neither a base type nor a class, but a Java construct
  - Cannot subclass
  - Use `new` to initialize an array (even though it’s not a class!)

Arrays (2/4)  
- Arrays hold only elements of specified type
  - When declaring an array, state type of object that array stores (base type, class, or for max polymorphic flexibility, interface)
  - You can only put references to instances of a class, base types, and even sub-arrays into the array
  - Note: you could declare array to be of `java.lang.Object` type to store any instance, but not useful: wouldn’t take advantage of compiler’s type-checking...
Arrays (3/4)

● Every array element is an object reference or base type
  What real-world objects could we organize with arrays?
  ○ Floors of buildings
  ○ Streets in Manhattan
  ○ Strings representing names or Banner id’s of people in a course
● Elements are ordered sequentially by numerical index
  ○ In mathematics, use subscript notation, i.e., \( A_0, A_1, A_2, \ldots A_{n-1} \)
  ○ With Java, represent 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

● 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

● The Sunlab Computers
  ○ Array size: 72
  ○ Array index: node number
  ○ Element type: computer

Note: This could also be modeled as a 2D array (see slide 27)
Java's Syntax for Arrays (1/5)

Declaration: `<visibility> <type>[] <array-name>;`
- `<visibility>` denotes the accessibility, i.e. public, private, etc. – we’ve seen this before!
- `<type>` denotes data type that the array is to hold in each slot: can be class, base type, interface, or even another array (for nested arrays)
- Colorable[] myColorables; declares a local variable: a one-dimensional array that holds references to instances of classes that implement `Colorable` interface
  - Note: unlike some other programming languages, size of array not specified in declaration, but in initialization;
  - Also no reserved word “array” - [] brackets suffice

Java's Syntax for Arrays (2/5)

Initialization:
`<type>[] <array-name> = new <type>[[<size>];`
- This syntax is for declaring arrays as local variables – can be instance variables too!
- `<size>` must be an integer value greater than 0; indices will range from 0 to `<size>-1`
- We use `new` here, but because arrays are a built-in type in Java, we have special syntax and `new` does not invoke a constructor, as it would for an instance of a class
- Note only array is initialized, not elements of array; all references are set to null, 0 for ints, false for booleans, etc.

Java's Syntax for Arrays (3/5)

- Arrays can be local variables, so we can declare and initialize in one statement, just like we can with objects and base types:
  Colorable[] otherColorables = new Colorable[5];
- Alternatively, arrays can be instance variables, in which case we initialize array in constructor after declaring it:
  private Colorable[] _myColorables;
  ...
  //in constructor of class that contains the array
  _myColorables = new Colorable[10];

Java's Syntax for Arrays (4/5)

- Accessing individual elements:
  `<array-name>[<index>]`
  - index must be an integer between 0 and (array_size-1)
  - result is the variable stored at that index
  - if `<index>` ≥ size, or < 0, an `ArrayIndexOutOfBoundsException` is thrown
  - Additionally, useful to check for uninitialized entries with `ref != null` – See slide 40
- Think of jock[i] as the “name” of that particular Jock, just as you would think jock as a name – avoids having to name each individual element in the collection with a unique name
Java’s Syntax for Arrays (5/5)

- Anywhere that you can use a variable or a constant, you can also use an array element. For example, in your *PaneOrganizer*:
  
  ```
  //initialize first element of an array of Colorables to be a Ball
  myColorables[0] = new Ball();
  //send a message to 3rd element of the array, after having checked it is not null
  myColorables[2].setColor(javafx.scene.paint.Color.RED);
  //assign fourth element of array to a local variable
  Colorable myColorableVar = myColorables[3];
  //pass 5th element of the array 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 upcoming assignments.*

Arrays as Parameters

- Can pass in an entire array as parameter by adding array brackets to type of formal parameter

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

Arrays as Parameters – length 1/2

- How do we determine size of array?
  - Could pass in a second parameter with each array parameter to specify its size, but we can avoid having to book-keep that property ourselves:
  - Arrays have their `length` as a public property (not a method)
  - Use special “dot” syntax to find out length; here we inquire it, then store it for future use

  ```java
  int arrayLength = <array-name>.length;
  ```

Arrays as Parameters – length 2/2

- How might you use `length` in an actual piece of code?
  Note common use of `for` to loop thru all elements of an array

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

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

  ```java
  
  “off-by-one” error - go through loop once too often
  ```
Example: Shooting Hoops

Design and implement a cartoon with ten BballPlayers.

When the "Score" button is pressed, all BballPlayers should execute their score method.

Quick Look at Design

Things we need:
- An App class
- A PaneOrganizer class
- A BballPlayer class
- A 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 BballPlayers in sequence?

Create ‘DropButton’ Button (2/2)

```java
package BasketballGame;

public class PaneOrganizer{
    private BballPlayer _team[];
    private Button _button;
    public PaneOrganizer(){
        _team = new BballPlayer[10];
        //Initialize array
        for(int i=0; i<_team.length; i++){
            _team[i] = new BballPlayer();
        }
        _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 player in turn to score
            for(int i=0; i<_team.length; i++){
                _team[i].score();
            }
        }
    }
}
```

Out-of-Bounds Problems

- Careful about bounds of loops that access arrays!
  - Java will throw ArrayIndexOutOfBoundsException if index is negative since sequence starts at 0
  - Also will throw ArrayIndexOutOfBoundsException if index is ≥ array size; remember that array goes from 0 to n-1
- Exceptions typically lead to crashes
  - Java has a catch keyword which can be used to "catch" and handle exceptions... used in CS16
  - Brief intro to how to try experimental code and catch errors at tail end of CS15
Multi-Dimensional Arrays (1/2)

- Say we wanted to model a chess board
  - Not a linear group of squares
  - More like a grid of squares
- Can declare an 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 array is an array of arrays of...
- The syntax above is for rectangular, cuboid, etc. multi-dimensional arrays
  - Since multi-dimensional arrays are just arrays of arrays of arrays, it is possible (using different syntax) to have jagged arrays, where each sub-array is of a different length
  - Thus can have a “triangle” shaped array
  - Don’t use this is CS15; even in CS16 and beyond, it is unlikely you will use this, but it is used 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:

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

Array of rows

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

Array of columns

Ways to Think About Array Storage

- Multi-dimensional arrays in Java do not make a distinction between rows or columns
  - Think about a 1D array – it doesn’t really matter if we call it a "row" or a "column"
  - You may think of an array as an ordered sequence of data stored in contiguous positions in memory - no intrinsic geometry/layout implied

Ways to Think About Array Storage

- Following two visualizations of a two-dimensional array (called array) are equally valid

```
<table>
<thead>
<tr>
<th>Column of Rows:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
</tr>
<tr>
<td>4 5 6</td>
</tr>
<tr>
<td>7 8 9</td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>Row of Columns:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
</tr>
<tr>
<td>4 5 6</td>
</tr>
<tr>
<td>7 8 9</td>
</tr>
</tbody>
</table>
```

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

- Just make sure you are consistent in the way you index into your 2D array throughout your program!

Example: Size of 2D Arrays

```java
public class ArraySize{
  //using row and column 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 you must loop over the array and assign values at each index
  ```java
  int[] myArray = new int[20]; //init array, not elements
  for (int i=0; i<myArray.length; i++) {
    myArray[i] = 5;
  }
  ```

Common Array Errors - Watch Out! (2/2)

- Assigning an array to a scalar
  ```java
  int myInt = myArray;
  ```
- Assigning arrays of different dimension to each other
  ```java
  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 the error:
  "Incompatible type for =. Can’t convert int[] to int[][]"
- Similar message for assigning arrays of the mismatched type

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 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
- Things that are new
  - 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 adds graphical items 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 Square 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);
  }

  public Node getNode() {
    return _rect;
  }
}
```

Building CheckerBoard (1/2)
- Let's start with standard stuff
  - contains array of CheckerSquares
  - All CheckerSquares will be added to the _root in the PaneOrganizer - so we must provide an 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)

```java
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)) { //every other square should be red
          rect = new CheckerSquare(Color.RED, Color.WHITE);
        } else { //every other square should be black
          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
      }
    }
    //for board pane to build scene graph
    public Node getNode() {
      return _rects;
    }
}
```
Updating Squares

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

```java
public void handle(ActionEvent arg0) {
    //local variable points to selected rect
    if(rect != null){
        //make sure value of array element isn't null (i.e., array initialized correctly)
        rect.toggleColor();
    }
}
```

What would happen if we didn't check for `null`? We might get a `NullPointerException`!

java.util.ArrayList (1/2)

- `java.util.ArrayList`, 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 the array at runtime
- As its 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 a particular index, all other elements falling in and after that index must be **shifted right** by programmer (their indices are incremented by 1)

Adding and Deleting in Arrays 2/2)

- When **deleting** from a particular index, all other elements falling in and after that index must be **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
    - String toString(): returns object’s “state” as string, could be used to print all instance variables’ values
    - void finalize(): used in garbage collection

What can ArrayLists hold? (2/2)

- **Upside:** **ArrayLists** store things as **Object**, so maximum polymorphic flexibility
  - since everything is an **Object**, **ArrayLists** can hold instances of any and every class
  - adding/removing **anything** from an **ArrayList** is easy
- **Downside:** **ArrayLists only** store **Objects**
  - only methods we can use on retrieved objects are those few trivial ones of **Object** itself: equals, toString, and finalize
  - want a homogenous collection to store only objects of a particular type AND have the compiler do type-checking for that type to enforce homogeneity
Generics! (1/2)

- Generics allow us to write a 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>()
  ```

- We already saw the use of generics in Graphics I, do you remember?
  - We used generics to specialize implementers of `EventHandler` interface.
  - You replace code inside `<>` with a subclass of `Event`, like `ActionEvent`.

Generics! (2/2)

- Programmers use “generics” to implement a collection class without knowing what specific type of object the user (another programmer) of that collection will want to store.
  - example: Java’s `ArrayList`’s class file defines an array list of `ElementType` left unspecified, but when users of the `ArrayList` actually declare and instantiate one, the type must be fully specified (e.g., an `ArrayList` of `Jocks` to serve Troy), much the way `arrays` (and all other variables) need types.
  - Java replaces `ElementType` with `Jocks` in return types and parameters of any `ArrayList` method you want to use.
  - but you must keep the literal `< >` brackets wherever they are used to indicate use of generics.

- Generics allow for generality of using any type while still having compiler do type checking.
- Think of the “generic” as a specialization of the type `ArrayList`.

java.util.ArrayList Methods (1/6)

```java
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"
```

```java
public ElementType get(int index)
//returns the object of type ElementType at that index
```

java.util.ArrayList Methods (2/6)

```java
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 */
```

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

```java
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 by searching (as opposed to using an index)
  - these methods take a parameter of type Object (superclass from which all Java classes inherit)
  - But you should never pass in (or get back) anything except an ElementType - using polymorphism here not for generality but with generics mechanism 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 at particular slot or at end
  - can retrieve an object stored at a particular index and perform operations on it
  - can use for loop to access all objects in ArrayList
  - shifting elements for adding/deleting from an ArrayList is done automagically by Java!
    - but beware, the 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 done automatically
- full class with useful methods, eg.,
  - 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 a heterogeneous collection of any kind of Object; want homogeneous collections...
- So specialize the ArrayList type by adding a "generic" specification to a declaration or instantiation, thereby specifying two classes in one statement: the collection and the type of object it is to hold and return
  - implementer of ArrayList collection declared constructor with <ElementType> as placeholder
    ```java
    public ArrayList<ElementType> ();
    ```
  - user of ArrayList "fills in" ElementType
    ```java
    private ArrayList<Jock> _jocks;
    _jocks = new ArrayList<Jock>();
    ```
- Be sure to use literal < > for specialized type!

Example (1/5)
```
public class JockCollection{
    /* To declare ArrayList, must specify type of object ArrayList stores.
    Replace all occurrences of ElementType with Jock, including where
    ElementType occurs in literal <> brackets. Could extend ArrayList, but it
    goes against good design. */
    private ArrayList<Jock> _jocks;

    public JockCollection(){
        //ArrayList Initialization - note literal <>
        _jocks = new ArrayList<Jock>();
        for (int i=0; i<3; i++){
            //Add a Jock at each pass
            _jocks.add(new Jock());
        }
    }
    //class definition continued on next slide
}
```

Example (2/5)
```
//Adds a new Jock at the end
public void addJock (Jock jock){
    _jocks.add(jock);
}

//If the specified Jock is in the collection, remove and banish him
public void banishJock (Jock jock){
    if (_jocks.contains(jock)){
        _jocks.remove(jock);
    }
}
} //End of Class
```
Example (3/5)

- `<Jock>` indicates use of Java generics
  - now, only `Jock` instances can be stored and retrieved from this `ArrayList`
- In `JockCollection`'s constructor, adding a new `Jock` works:
  - `_jocks.add(new Jock());`
- However, adding another type to the `ArrayList` of `Jocks` won’t work:
  - `_jocks.add(new String("Yeah!"));` // Won’t work… parameters need to be of type `Jock`
- Exception thrown!
  - “The method `add(Jock)` in the type `ArrayList<Jock>` is not applicable for the arguments (String)”

Example (4/5)

```java
public class Troy {
    private JockCollection _jocks;
    private Jock _lazyJock;

    public Troy(JockCollection jocks, Jock lazyJock){
        _jocks = jocks;
        _lazyJock = lazyJock;
    }
    // Method to act like Team Captain
    public void runPractice(Jock jock){
        jock.obey(this); // method def elided
        this.shootBaskets(); // method def elided
    }
}
```

Example (5/5)

```java
public class HighSchoolMusical{
    private JockCollection _jocks;
    private Troy _tbolt;
    public HighSchoolMusical(){
        _jocks = new JockCollection();
        Jock zeke = new Jock("zeke");
        _jocks.add(zeke);
        _tbolt = new Troy(_jocks, zeke);
    }
    // End of Constructor
    public void haveBoringDay(){
        _tbolt.doHairCommercial();
        _tbolt.hangWithGamonte();
    }
    // End of Class
}
```

Enhanced for Loop (1/2)

- Remember for loops from last lecture?
- Basic for loop was extended in Java 5 to make iteration over arrays and other collections easier
  - commonly called for-each or for-in loop
  - <> here NOT literal, i.e., not for generics

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

- `<type>`: class of objects stores in the `<structure>`
- `<var>`: name of the loop counter – you don’t advance it explicitly yourself but you can use it to access elements
- `<structure>`: data structure (array or other collection) you’d like to iterate over
Enhanced for Loop (2/2)

- Intended to simplify most common form of iteration, when, as name suggests, loop body is applied to each and every member of the collection
- But, how do `for-each` loop and `for` loop differ?
  - In a `for` loop, have access to index at which an item is stored in an array or collection (and hence item itself)
  - In a `for-each` loop, we don’t have direct access to the index, but can easily access item (see next example)

for-each vs. for loop (2/2)

- However, if we want to operate on every element in the `ArrayList` and the loop body does not require element indices, we may use a `for-each` loop:

  ```java
  //Instead of only even-numbered students, now everyone must apologize!
  for (EastHighStudent student : _eastHighStudents){
    //notice how don’t need to use index to get student from ArrayList
    student.apologize();
  }
  ```

- A great advantage of `for-each` loops is that they don’t give you `ArrayIndexOutOfBoundsExceptions`! Why?
  - Java does the indexing for you. And Java is never wrong!

for-each vs. for loop (1/2)

- Consider this `for` loop:

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

- In the loop above, we only want to call `apologize()` on elements at even indices. A `for-each` loop would not be appropriate in this case. Why?
  - Because we don’t execute `apologize()` on every element in the `ArrayList`; we only care about elements at specific indices.

Understanding Mainline (and optional params)

- If we type this in a terminal:
  ```bash
demo Demos.Mainline/App Hello CS15
  ```
- We get the output:
  ```bash
  Hello
  Hello
  ```
- In this example, if we passed two or more arguments to the mainline, we’d get an `ArrayIndexOutOfBoundsException`!
- Why? Because array `argv`’s size is exactly equal to the number of parameters passed to the mainline.
- You won’t need to use mainline parameters in CS15, but we wanted you to know they exist. You will probably use them in CS32!
Announcements

• Cartoon deadlines
  o early handin Thursday 11:59PM
  o on-time handin Friday 10:00PM
  o late handin Sunday 11:59PM

• 10% deduction if code doesn't compile

• Don't copy the Alien demo!