### Lecture 12

**Arrays**

<table>
<thead>
<tr>
<th></th>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

• **Purpose**

• **Syntax**
  - Lecture Questions: Slide 18, Slide 30

• **Multi-Dimensional Arrays**
  - Lecture Question: Slide 40

• **ArrayLists**
  - Lecture Questions: Slide 63, Slide 69

• **Generics**
Why Use Arrays? (1/2)

- So far, we’ve only studied 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 also cover lists, queues, stacks, and trees
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 Island
  - store all Islands in array for easy access (to film new seasons of Love Rhode Island!)

- Arrays are ordered - helpful when wanting to store or access instances in particular order, e.g., alphabetically
Your lovely TAs

• We want access to all 32 UTAs!
  o Abigail, Adwith, …, Wambui

• Could use instance variables:
  
  ```java
  public class CS15TAs {
      private TA _abigail, _adwith, …, _wambui;
  }
  ```

• Can’t access 32 instance variables very easily
  o what if we wanted to access CS15 TAs from 2019, 2018, …
Fibonacci Sequence (1/2)

- Pervasive in nature, along with golden ratio phi = 1.618, logarithmic spiral, etc. [http://jwilson.coe.uga.edu/emat6680/parveen/fib_nature.htm](http://jwilson.coe.uga.edu/emat6680/parveen/fib_nature.htm)

- 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 for this too:

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

• Gets tiresome and isn't flexible
  
  ○ try making sequence with forty numbers, one 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]`
      - the `index` of array is always of type `int`
  - neither base type nor class, but Java **construct**
    - use `new` to initialize an array (even though it’s not a class!)
    - special syntax, does not invoke constructor like for 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 or superclass
  - 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 an object reference, subarray, or base type. What real-world objects can be organized by arrays?
  o number of electoral votes by state
  o streets in Providence
  o *Strings* representing names or Banner ID’s of people in a course

● Elements ordered sequentially by numerical index
  o in math, use *subscript* notation, i.e., $A_0, A_1, A_2, \ldots A_{n-1}$
  o in Java, use *index* inside brackets, i.e., for an array of students: `students[0], students[1], \ldots students[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 representations](image)

Note: 1-D Arrays are called vectors, and 2-D or n-D arrays are called matrices in mathematics
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)

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

  Note: Could be modeled as a 2-D array (see slide 34)
Java’s Syntax for Arrays (1/5)

Declaration:

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

e.g., `private House[][] _houses;`

- `<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, superclass, or another array (nested arrays)
- 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 (2/5)

Initialization:

```java
new <type>[<size>];
```

- `<size>` must be integer value greater than 0; indices range from 0 to `<size>`-1
- We use `new` here, because arrays are a Java **construct**
- Only *array* is initialized, not *elements* of array; all references are set to a default of `null` for Objects, 0 for ints, false for booleans, etc.
  - we are still responsible for initializing elements, typically in a constructor
Initializing an Array

- Houses on a Neighborhood Street

House[] houses = new House[8];

- Sunlab Computers

Computer[] sunlab = new Computer[72];
Java’s Syntax for Arrays (3/5)

- 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;
  ...
  //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>]`

  o `index` must be integer between 0 and (array size-1)
  o result is variable stored at that index
  o if `<index>` > size, or < 0, `ArrayIndexOutOfBoundsException` gets thrown
  o also useful to check for uninitialized entries with
    `<ref> != null` — See slide 50

- Think of `student[i]` as the “name” of that particular student (like student$_i$) – simpler way to refer to each individual element in collection, better than having to use unique names

Note: some other languages allow an arbitrary value for the lower bound, but not Java!
Accessing Array Elements Example

- Houses on a Neighborhood Street
  ```java
  House[] houses = new House[8];
  //code initializing array elements elided
  House myHouse = houses[6];
  ```

- Sunlab Computers
  ```java
  CPU[] sunlab = new CPU[72];
  //code initializing array elements elided
  CPU myCPU = sunlab[42];
  ```
Lecture Question 1

Which of the following is the correct way to declare and initialize an array of Strings named suspects, of size 5?

A. String suspects = new array(5);
B. String[] suspects = new array(5);
C. String[] suspects = new String[5];
D. String[5] suspects = new String[];
An array element will work anywhere a variable or constant would

// initialize first element of array of objects implementing Colorables to be a Ball
myColorables[0] = new Ball();

// call a method on 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]);
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){ //no size declared!
    //code to compute sum of elements in the int array
  }
  ```

- Now we can do the following (somewhere else in the class that contains `sum()`):
  ```java
  int[] myNumbers = new int[5];
  //code elided - initializes myNumbers with values
  System.out.println(sum(myNumbers));
  ```

Note: there is no way to tell from this use of `sum` that `myNumbers` is an array - would need to see how `sum` and `myNumbers` were declared to know that!
Arrays as Parameters (2/3)

- How do we determine size of array?
  - arrays have `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 often used to traverse through all elements of array. Can use loop counter (`i` in this case) inside the body of loop, but should **never** reset it. Incrementing/decrementing counter is done by `for` loop itself!
Example: IslandContestants

• Design and implement a cartoon with IslandContestants

• When the “Find Love!” button is pressed, all IslandContestants should execute fallInLove() method
IslandContestant: Quick Look at Design

Things we need:

● App class
● PaneOrganizer class
● IslandContestant 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 IslandContestant in sequence?
package LoveRhodeIsland;

/* Standard setup for the Stage, Scene, etc. has been elided. Other standard code like making a Pane, adding the button, and implementing getRoot() have also been elided */

public class PaneOrganizer{
    private IslandContestant[] _contestants; // PaneOrganizer continued
    private Button _button;

    public PaneOrganizer(){
        _button = new Button("Find Love!");
        ...
        //Initialize array
        _contestants = new IslandContestant[5];

        //Fill the array with IslandContestants
        for(int i=0; i < _contestants.length; i++){
            _contestants[i] = new IslandContestant();
        }

        _button.setOnAction(new ClickHandler());
    }

    private class ClickHandler implements EventHandler<ActionEvent> {
        public void handle(ActionEvent event){
            //loop through array, telling each IslandContestant in turn to fall in love
            for (int i=0; i < _contestants.length; i++){
                _contestants[i].fallInLove();
            }
        }
    }
}

// end of PaneOrganizer class
Out-of-Bounds Problems

- Careful about bounds of loops that access arrays!
  - Java throws `ArrayIndexOutOfBoundsException` if index is negative since sequence starts at 0
  - throws `ArrayIndexOutOfBoundsException` if index is ≥ array size; remember that array goes from 0 to \( n-1 \)
  - exceptions typically lead to crashes
    - Java has `try/catch` keywords which can be used to try out code and “catch” and handle any exceptions due to bugs… used in CS16
ArrayIndexOutOfBoundsException

Example of a classic “off-by-one” error!

In Terminal:
Error:
java.lang.ArrayIndexOutOfBoundsException: 5
at (PaneOrganizer.java:14)

Note: The error tells you which index is throwing the error. Here, it is attempting to access the element at index=5, but our largest index of an array of size 5 is n-1 or, in this case, 4.
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), otherwise data at index of insertion will be erased and replaced.
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)
Lecture Question 2

Consider the sum function from slide 22:

```java
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 `ArrayIndexOutOfBoundsException`
D. None of the above
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
  - 2-D: `int [][] grid = new int [a][b];`
  - 3-D: `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 arrays of arrays of...

- Syntax in previous slide 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 in CS15; even in CS16 and beyond, it is unlikely you will need this (used predominately for scientific/engineering computation)
2-Dimensional Array Examples (1/2)

- **Pixel Array**
  - 2-D Array size: width by height
  - array indices: x, y
  - element type: RGB color
  - `Pixel[][]` MSFTLogo = new `Pixel[x][y]`; 

- **Connect Four**
  - 2-D Array size: 6 by 7
  - array indices: row, column
  - element type: checker
  - `Checker[][]` connect4 = new `Checker[6][7]`;
2-Dimensional Array Examples (2/2)

- The Sunlab
  - 2-D Array size: 10 by 8 (approx.)
  - array indices: row, column
  - element type: computer
  - `Computer[][] sunlab = new Computer[10][8];`
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 (1/3)

- 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 (2/3)

- Two visualizations of two-dimensional array (called ballArray) are equally valid. You can choose either for the organization of your array.

- Make sure there’s consistency in the way you index into your 2-D array throughout your program!
  - since the elements are not stored in a specific order, the way that we insert elements and initialize and index into our array determines the order
Ways to Think About Array Storage (3/3)

- The choice between row-major and column-major organization can sometimes be arbitrary
  - Connect 4, a large carton of eggs, etc.
- However, sometimes one will make more sense or simplify your program based on what you are trying to achieve
- Can Storage example
  - goal: use array to keep track of the number of each type of can
  - makes most sense to use column-major organization
    - each column would be a sub-array of cans of the same type
    - rows within each column are either null (empty) or hold a can
    - can count number of each type by checking to see how many entries are full (or not null) in each sub-array (column, here)
- For a table of entries use row major order, while for pixel (x, y) use column major order
Lecture Question 3

Here’s a grid of colored golf balls in column major order. What index is the light blue golf ball in?

A. ballArray[2][3]
B. ballArray[2][1]
C. ballArray[3][2]
D. ballArray[1][2]
Example: Size of 2-D Arrays

```java
public class ArraySize{
    //deciding which is row and which is column index is arbitrary, but must be consistent!!!
    private static final int NUM_ROWS = 10;
    private static final int NUM_COLS = 5;
    public ArraySize(){
        //String is just an arbitrary choice of type for this array!
        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));
    }

    //Method is general, here we use row major order, column of rows
    public int find2DArraySize(String[][] array){
        //number of entries in column, each is a row
        int numRows = array.length;
        //element/row 0 is an array
        int numCols = array[0].length;
        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)

- Cannot assign a scalar to an array

  int[] myArray = 5;

  o 5 is not an array
  o to initialize array elements, must loop over the array and assign values at each index. Here we assign 5 to each element:

  int[] myArray = new int[20]; //initializes array, not elements
  for (int i=0; i < myArray.length; i++){
    myArray[i] = 5;
  }
Common Array Errors - Watch Out! (2/2)

- Cannot assign arrays of different dimensions to each other

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

- Doing so will result in this 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 2-D 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 2-D array of CheckerSquares
  - CheckerSquare which has the ability to toggle its color

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

- Stores 2 colors and toggles between them; wrapper that contains a rectangle, doesn’t subclass from it

```java
public class CheckerSquare {
    //red toggles to white, black toggles to blue
    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 Rectangle getRect() {
        return _rect;
    }
}
```

//CheckerSquare class continued
```
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 Rectangle getRect() {
    return _rect;
}
```

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

- Let’s start with standard stuff
  - contains array of `CheckerSquares`; think of it as a checkerboard with a few methods
  - all `CheckerSquares` get added to `_root` in `PaneOrganizer` – so it needs accessor for the array

- Row-major or column-major order? It doesn’t necessarily matter in this example, as long as we are **consistent** throughout our program.
  - this example will be in **row-major order**, but it can easily be rewritten in column major
  - row-major order corresponds to **rows as first index** (y-coordinate) and **columns as second** (x-coordinate)

\[
\begin{array}{cccc}
  a & b & c & d \\
  e & f & g & h \\
  i & j & k & l \\
  m & n & o & p \\
\end{array}
\]

a is at (0,0), while n is at (3, 1), i.e., has index [3] [1]
Building **CheckerBoard (2/2)**

/* CheckerBoard is a thin wrapper around an array of CheckerSquares that in the constructor does a nested for loop to initialize the array, and provides an accessor for it. The event handler in handle() in the private inner class ClickHandler has access to the array so it can toggle all squares */

```java
public class CheckerBoard {
    private CheckerSquare[][] _checkerArray;
    public CheckerBoard(){
        _checkerArray = new CheckerSquare[Constants.NUM_SQRS][Constants.NUM_SQRS];
        for (int row=0; row< Constants.NUM_SQRS; row++){ //outer for loop through rows
            for (int col=0; col < Constants.NUM_SQRS; col++){ //nested inner for loop through columns
                CheckerSquare rect = null; // initialize rect so we can access it outside if statement
                if (((row + col) % 2) == 0){ // every other square should be red
                    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); //set graphical position
                _checkerArray[row][col] = 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 _checkerArray;
    }
}
```

This method returns our array of **CheckerSquares** so that they can be added to the Scene Graph in **PaneOrganizer**

setLocation positions rect based on its upper left corner
Updating **CheckerSquares**

- **ClickHandler** private inner-class sends message to **CheckerBoard** to toggle color of all its squares

```java
// in PaneOrganizer Class...
private class ClickHandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        for (int row=0; row < Constants.NUM_SQRS; row++) {
            for (int col=0; col < Constants.NUM_SQRS; col++) {
                // make sure value of array element isn't null (i.e., array initialized correctly)
                if (_checkerArray[row][col] != null) {
                    _checkerArray[row][col].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 to avoid crashing.
PaneOrganizer class

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

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

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

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

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

```java
private class ClickHandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        for (int row=0; row< Constants.NUM_SQRS; row++) {
            for (int col=0; col < Constants.NUM_SQRS; col++) {
                // make sure value of array element isn't null
                // (i.e., array initialized correctly)
                if (_checkerArray[row][col] != null){
                    _checkerArray[row][col].toggleColor();
                }
            }
        }
    }
}
```

```java
// End of private inner-class
// End of PaneOrganizer
```
public class PaneOrganizer {
    private CheckerBoard _board; private BorderPane _root;
    private CheckerSquare[][] _checkerArray;
    public PaneOrganizer(){
        _root = new BorderPane(); _board = new CheckerBoard();
        _checkerArray = _board.getRectangles();
        this.setUpBoardPane(); this.setUpButtonPane();
    }
    private void setUpButtonPane(){
        HBox buttonPane = new HBox();
        Button button = new Button("Change Color!");
        buttonPane.getChildren().add(button);
        buttonPane.setAlignment(Pos.CENTER);
        button.setOnAction(new ClickHandler());
    }
    public Pane getRoot(){
        return _root;
    }
    private void setUpBoardPane(){
        Pane boardPane = new Pane();
        for (int row=0; row<NUM_SQRS; row++){
            for (int col=0; col < NUM_SQRS; col++){
                boardPane.getChildren().add(_checkerArray[row][col].getRect());
            }
        }
    }
    private class ClickHandler implements EventHandler<ActionEvent> {
        @Override
        public void handle(ActionEvent event) {
            for (int row=0; row<NUM_SQRS; row++){
                for (int col=0; col < NUM_SQRS; col++){
                    if (_checkerArray[row][col] != null) {
                        _checkerArray[row][col].toggleColor();
                    }
                }
            }
        }
    }
}

public class Constants {
    public static final int NUM_SQRS = 8;
    public static final int SQR_SIZE = 30;
    public static final int APP_WIDTH = ...
    public static final int APP_HEIGHT = ...
    public static final int APP_WIDTH+25;
}

public class CheckerSquare {
    private Color _currentColor; private Color _otherColor; private Rectangle _rect;
    public CheckerSquare(Color primaryColor, Color secondaryColor) {
        _rect = new Rectangle();
        _rect.setRect(0);
        _rect.setRectWidth(100);
        _rect.setRectHeight(100);
        _rect.setRectForeColor(primaryColor);
        _otherColor = secondaryColor; _rect.setRectColor(_otherColor);
    }
    public void setLocation(int x, int y){
        _rect.setLocationX(x); _rect.setLocationY(y);
    }
    public void toggleColor(){
        _Color temp = _currentColor; _currentColor = _otherColor;
        _otherColor = temp; _rect.setRectColor(_currentColor);
    }
    public Rectangle getRect() {return _rect;}
}

public class CheckerBoard {
    private CheckerSquare[][] _checkerArray;
    public CheckerBoard(){
        _checkerArray = ...
        for (int row=0; row<NUM_SQRS; row++){
            for (int col=0; col < NUM_SQRS; col++){
                CheckerSquare rect = null;
                if (((row + col) % 2) == 0){
                    CheckerSquareColor(rect.RED, rect.WHITEM);
                }
                else {
                    CheckerSquareColor(rect.BLACK, rect.BLUE);
                    rect.setLocation(col*Constants.SQR_SIZE, row*Constants.SQR_SIZE);
                    _checkerArray[row][col] = rect;
                }
            }
        }
    }
    public CheckerSquare[][] getRectangles() {return _checkerArray;}
}
SciLi Tetris: Loops and Arrays Writ Large

• In 2000, Tech House constructed then the largest Tetris game on the Scili – the Woz flew out to play it!

• 5 months of work: 11 custom-built circuit boards, a 12-story data network, a Linux PC, a radio-frequency video game controller, and over 10,000 Christmas lights – see http://bastilleweb.techhouse.org/

• Video: https://www.youtube.com/watch?v=tkIRWoo9qrU&t=21s

• Article: http://news.bbc.co.uk/2/hi/science/nature/718009.stm
java.util.ArrayList (1/2)

- java.util.ArrayLists, 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 and mutable number of references
  - are Java classes, not Java constructs, so have methods
java.util.ArrayList (2/2)

- Why use them instead of arrays?
  - when number of elements to be held is unknown
  - storing more data in an array that’s too small leads to errors
  - 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 you expect both array and array list to hold same number of elements)
  - want faster operations
Objects

- **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?

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

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

- Generics allow **designer** to write collection class A to hold instances of another class B, without regard for what class B will be (can be any **Object** for **ArrayLists**). **User** of that class A then decides how to restrict/specialize type for that homogeneous collection.

- This is the constructor of the generic **ArrayList** (a collection class):

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

- Think of **ElementType** as a “type parameter” that is used as a placeholder that the user will substitute for with any non-primitive type (class, interface, array, …)
  
  - primitive types: **boolean**, **int**, **double** must be special-cased – Slide 67

- For example, we saw the use of generics to specialize implementation of **EventHandler** interface to handle a specific type of **Event**, e.g., **ActionEvent**

- Provides flexibility to have collection store any type while still having compiler help by doing type-checking

---

1As we will see in the Linked List lecture, the **EventHandler** interface has a generic that extends the **Event** superclass. This more specialized generic then forces users of the interface, i.e., our **EventHandler**s, to specify only subtypes of the **Event** superclass.
Generics! (2/2)

- With generics, `ArrayList` was implemented by the Java team to hold any `Object`, but once an instance of an `ArrayList` is created by a programmer, they must specify the type. Let’s create an `ArrayList` of `IslandContestants` for our Love Rhode Island theme!

```
ArrayList<IslandContestant> contestants = new ArrayList<IslandContestant>();
```

- We specify `IslandContestants` as the type that our `ArrayList`, `contestants`, can hold. Java will then replace `ElementType` with `IslandContestant` in `ArrayList` method parameters and return types.

- Can think of generics as a kind of parameter, just with different syntax (the `< >`) since only methods have parameters, not classes. In this case, `ElementType` acts as the formal parameter and `IslandContestant` is the argument.

- Generics, like classes and methods with parameters, provide generality in programming! (as does polymorphism in parameter passing)
java.util.ArrayList Methods (1/6)

//Note: only most important methods shown (ALL defined for you!)
//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
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- but is done
 * automagically
 */

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

public ElementType remove(int index)
//removes the ElementType at given index and returns it
public int size()
//returns number of elements stored in ArrayList

public boolean isEmpty()
//returns true if ArrayList contains zero elements; false otherwise
Lecture Question 4

Given an array of Cats, we would call .length to get how many elements were in the array. What would we call to get the number of elements in an ArrayList of Dogs?

A. .length
B. .length()
C. .size
D. .size()
java.util.ArrayList Methods (4/6)

- ArrayLists also have methods that access elements through search (as opposed to using an index)
  - these methods take parameter of type Object
  - but should never pass in anything besides ElementType
public int indexOf(ElementType elem)
//finds first occurrence of specified element, returns -1 if element not in ArrayList

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

public boolean remove(ElementType elem)
//remove first occurrence of specified element and returns true
//if ArrayList contains specified element
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 and return types:
  - `ElementType get(int index)`
  - `boolean add(ElementType element)`
  - `void add(int index, ElementType element)`
  - `int indexOf(ElementType elem) //search`
  - `ElementType remove (int index)`
  - `boolean remove (ElementType elem)`
  - `int size()`
  - `boolean isEmpty()`

*Weird edge case:* To make an `ArrayList` of primitive types, just specify `Boolean`, `Integer`, or `Float` in the generic brackets.

The `Boolean remove()` also has a weird edge case for Integers: you cannot use `remove(5)` to remove the first occurrence of 5, because it will treat it as the `ElementType` `remove`. This would remove whatever is at index 5. To remove an `Integer` element, use `remove(new Integer(<number>))`.
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

```java
ArrayList<IslandContestant> contestants = new ArrayList<IslandContestant>();
```

- Remember to use literal `<>` for specialized type!
Lecture Question 5
Which of the following uses an ArrayList correctly?

A. `ArrayList<Contestant> contestants = new ArrayList<Contestant>();
Contestant funnyContestant = new Contestant();
contestants.add(funnyContestant);

B. `ArrayList<ElementType> contestants = new ArrayList();
Contestant toxicContestant = contestants[0];`

C. `ArrayList<Contestants> contestants = new ArrayList<ElementType>();
Contestants fitContestant = contestants.first();`

D. `ArrayList<String> contestants = new ArrayList<Contestant>;
Contestant villainContestant = new Contestant();
contestants.add(villainContestant);`
Example: Couples (1/5)

```java
public class IslandCoupleRecord{
   /* IslandCoupleRecord 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. This type of enriching of basic collections is a very common pattern.
We’ll also restrict IslandCoupleRecord to have no more than twenty couples.
To declare ArrayList, specify type of object ArrayList stores.
Replace all occurrences of ElementType in method calls with Couple, a class modeling
couples, including where ElementType occurs in literal <> brackets. */
private ArrayList<Couple> _listOfCouples;

   public IslandCoupleRecord(){
      //ArrayList initialization- note literal <>
      _listOfCouples = new ArrayList<Couple>();
      for (int i=0; i<20; i++){
         //Add a Couple at end in each pass
         _listOfCouples.add(new Couple());
      //scenegraph code elided
   }
   }
//class definition continued on next slide
```

Note: Couple is a class that models a couple
Example: Couples (2/5)

/*Adds numCouples of new Couples at end if _listOfCouples has room, given only up to 20 items*/
public void addCouples(int numCouples){
    for (int i = 0; i < numCouples; i++) {
        if(_listOfCouples.size() < 20) {
            _listOfCouples.add(new Couple());
            //scenegraph code elided
        } else {
            System.out.println(“Couple list is full!”);
            break;
        }
    }
}

/*If _listOfCouples still has couples, remove and return first one, thereby removing it from the ArrayList, else return null*/
public Couple removeCouple(){
    if (!_listOfCouples.isEmpty()) {
        return _listOfCouples.remove(0);
    } else {
        System.out.println(“Couple list is empty!”);
        return null;
    }
}
} //End of Class
Example: Couples (3/5)

- `<Couple>` indicates use of Java generics
  - ensures only `Couple` instances can be stored and retrieved from this `ArrayList`
- In `IslandCoupleRecord`'s constructor, adding a new `Couple` works:
  
  ```java
  _listOfCouples.add(new Couple());
  ```
- However, adding another type to `ArrayList` of `Couple` will fail:
  
  ```java
  _listOfCouples.add(5);
  ```
  - “The method `add(Couple)` in the type `ArrayList<Couple>` is not applicable for the arguments (int)”
public class Will {
    private IslandCoupleRecord _record; // ArrayList of couples
    public Will(IslandCoupleRecord record) {
        _record = record;
    }
}

//Method allows Will to vote on Couples if the ArrayList isn't empty
public void voteOnCouples() { // vote on ALL couples!
    for (int i = 0; i < _record.size(); i++) {
        // store removed couple in a local variable since
        // we use it multiple times
        Couple currCouple = _record.removeCouple();
        if (currCouple == null) { // list was empty
            break; // exit loop
        } else {
            this.vote(currCouple);
        }
    }
}

// continuation of Will class
public void vote(Couple c) {
    // code elided
}
} // End of class
public class BrownIsland{
    private IslandCoupleRecord _record;//arrayList of couples
    private Will _will;

    public BrownIsland(){
        _record = new IslandCoupleRecord();
        _will = new Will(_record);
    }

    public void judgeCouples(){
        /* Can remove from ArrayList because it contains
        20 couples initially */
        _will.voteOnCouples();
    }
} // End of Class
for vs. for-each loop (1/4)

- 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 gives 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 (2/4)**

- *for* loops were extended to *for-each* (or *for-in*) loops, which iterate over the contents of a data structure rather than indices
  - <> 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

  - Can make up any arbitrary name for `<var>` field, just like when we declare a variable and choose its name.
**for vs. for-each loop (3/4)**

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

  ```java
  for (IslandContestant contestant: _contestants){
    //notice how don’t need to use index to get members from ArrayList
    contestant.fallInLove();
  }
  ```

- Great advantage of **for-each** loops is that they don’t raise **ArrayIndexOutOfBoundsException**! Why?
  - Java does the indexing for you!
**for vs. for-each loop (4/4)**

- Consider this `for` loop:

  ```java
  //Somewhere in LoveRhodeIsland class
  //note: _contestants is an ArrayList<IslandContestant>
  for (int i=0; i < _contestants.size(); i++){
      if (i % 2 == 0){ //if index ‘i’ is even
          //fallInLove() is defined in IslandContestant
          _contestants.get(i).fallInLove();
      }
  }
  ```

- Only want to call `fallInLove()` on elements at even indices, so `for-each` loop wouldn’t work
  - we don’t execute `fallInLove()` on every element in the `ArrayList`; we only care about elements at specific indices
public static void main(String[] args){
    ArrayList<IslandContestant> contestants = new ArrayList<IslandContestant>();
    contestants.add(new IslandContestant("Aalia"));
    contestants.add(new IslandContestant("Anna"));
    contestants.add(new IslandContestant("Gil"));
    contestants.add(new IslandContestant("Marina"));
    contestants.add(new IslandContestant("Will"));
    for (IslandContestant c : contestants){
        if(!c.getName().equals("Marina")){
            contestants.remove(c);
        }
    }
}

● When trying to modify an ArrayList while iterating through it with a for-each loop, you will get a ConcurrentModificationException
● Adding and removing cannot be done within a for-each loop because of the shifting of the elements in the list that Java does in response to an add or remove
● Note: this is important for DoodleJump! We’ll go over this issue in detail during the project help slides.
*Understanding Mainline (and optional params)*

- You’ve seen the mainline 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 one or no arguments get passed into mainline, compiler would raise `ArrayIndexOutOfBoundsException`!
  - Why? `argv`’s size is exactly equal to the number of parameters passed to mainline. We are accessing the first and second elements, and if those are out of bound, this would raise an error.
- You won’t need to use mainline parameters in CS15, but it’s a good thing to know!

```
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();
  }
}
```
Announcements

• Cartoon Late late handin tomorrow 3/5 at 11:59 EST
  o Remember to submit what you have to codePost!

• DoodleJump is out
  o Early: 3/15
  o On time: 3/17
  o Late: 3/19
  o Make sure to complete the code checkpoint and your mini assignment by section next week

• SRC extra credit assignment: details coming soon!
IT in the News

ft. Socially Responsible Computing!
What is AI, ML, and DL?

**Artificial Intelligence**
- describes the ability of machine to perform “intelligent” tasks – e.g., prediction, classification, learning, planning, or perception.
- Examples: autonomous vehicles, robots, text-generation/image-generation algorithms, image recognition

**Machine Learning**
- describes the ability of a machine to “learn” from data using mathematical and statistical methods.
- Examples: voice and facial recognition, recommendation algorithms

**Deep Learning**
- a subset of ML using complex networks based on a simplified model of the human brain. Can usually perform more complex tasks than simpler ML systems.
- Examples: autonomous vehicles, image classification, natural language processing (e.g., text generation, machine translation)
Uses of ML & DL

- Works based on **pattern recognition** → applications **everywhere**
  - Voice Recognition (e.g. Siri, Alexa)
  - Image Recognition/Classification (e.g., Microsoft Seeing AI)
    - Facial Recognition
  - Text generation/“writing” (e.g., GPT-3)
  - Weather Forecasting
  - Translation/Internationalization (e.g., Google Translate)
  - Content recommendations/“Personalization” (e.g., Netflix, YouTube, Google Search)
  - Autocomplete
  - Content moderation (recall FB)
  - Self-Driving/Autonomous Vehicles (e.g., Tesla)
  - Disease detection & diagnosis
  - and more, and more, and more, and more!

“[ML] uses past data and projects it forward”
– danah boyd ‘00, in visiting talk for CS1951V Hypertext Seminar

CS15 final project Evolution uses neural networks, a DL technique!
Positive Use Case: Improving Accessibility!

- **Sound-based User Interfaces (UIs)**
  - Blind/low-vision people often cannot use standard Graphical User Interfaces (GUIs), instead use Voice Recognition/Generation
    - Examples: Siri, Alexa, Dragon

- **Auto-captioning for video**
  - Deaf/Hard of Hearing people (and more!) rely on closed captions to understand video
    - Examples: Google Live Caption, Otter.ai (for Zoom and Google Meet!)

- **Live image recognition (in development)**
  - In day-to-day life, many blind/low-vision people cannot access print documents, visual cues in surroundings
    - Examples: Microsoft Seeing AI

Generated description: “Probably a child playing with a frisbee.”

Example of "Scene Recognition" functionality, Microsoft Seeing AI
Problematic Application: Surveillance (1/2)

- Facial recognition now widely used for authentication (opt-in process, e.g., Apple FaceID)
  - interesting ramification: makes it easier for law enforcement to unlock suspects’ phones without express consent

- Law enforcement has used Facial Recognition (FR) for decades with limited data (government databases: ~411M photos)
  - limited, biased data → false matches → false allegations/arrests

- Private companies allowed to use more data → make & sell FR for US law enforcement (legal loophole)
  - still, limited & unequal accuracy – e.g., 2018 scandal: Amazon AI system fails to identify members of Congressional Black Caucus

Source: Getty Images, via Vox
2019: new company, Clearview AI, uses images from social media, public websites for facial recognition – ~3B photos
  ○ 100% legal under US regulations, but against social media fair use/scraping policies
  ○ even if taken down online, images not removed from database

2021: Clearview AI illegal in Canada due to severe privacy infringement
  ○ but Clearview will not delete Canadians’ data

Clearview AI still used by 2,400 US law enforcement agencies

Source: Clearview AI, via NY Times (Jan. 18, 2020)
Different Applications, Overlapping Issues

“Technology is neither good nor bad, nor is it neutral.”

– Martin Kranzberg, technology historian

**Assistive/Accessible Technology**
- used by private individuals
- consent from primary users
- improper use causes little harm

**Clearview AI**
- used by law enforcement
- based on non-consensual data collection
- improper use jeopardizes privacy, safety, lives

- ML-based
- rely on large datasets, continuing data collection
- use image recognition
- pose privacy concerns

Andries van Dam © 2021 3/4/21