Submit to Catalyst Journal!

Catalyst Journal is a science-inspired literary journal that aims to bridge the gap between the sciences and the humanities.

Submit
- Prose
- Poetry
- Drawing
- Painting
- Photography
- Etc.

We are looking for anything about or inspired by the scientific world.

Email submissions to catalyst.brown@gmail.com by October 21!

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...
  ```java
  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
  o each element must be same type or subclass of same type (polymorphism)
- Arrays are special in Java
  o special syntax to access array elements:
    ```java
    _studentArray[index]
    ```
  o 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
  o 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
  o 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?
  o number of electoral votes by state
  o streets in Manhattan
  o strings representing names or Banner IDs 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., `array[0], array[1]...array[n-1]`

Arrays (4/4)
- Arrays store objects in numbered slots
  o for array of size n, first index is always 0, last index is always n-1
- Common graphical representations of arrays:
  ```
  [0][1][2][3][4][5]   [0][1][2][3][4]
  [1][2][3][4][5][6][7][8][9]
  [10][11][12][13][14]
  ```

Array Examples (1/2)
- Houses on a Neighborhood Street
  o array size: 8
  o array index: house number
  o element type: house
  ```
  0  1  2  3  4  5  6  7
  ```
- 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.
- `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:

```java
<type>[] <array-name> = new <type>[<size>];
```

- syntax for declaring arrays as local variables - can be instance variables too!
- `<size>` must be integer value greater than 0; indices range from 0 to `<size>-1`
- we use `new` here, but because arrays are built-in construct
- in Java, we have special syntax and `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 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; ...
  //in constructor of class that contains the array
  _myColorables = new Colorable[18];
  ```

Java’s Syntax for Arrays (5/6)

- Accessing individual elements:
  ```java
  <array-name>[<index>]
  ```
  - index must be integer between 0 and `<array_size>-1`
  - result is variable stored at that index
  - if `<index>`  ≥ size, or < 0, `ArrayIndexOutOfBoundsException` gets thrown
  - also useful to check for uninitialized entries with `cref != null` — See slide 46

- Think of `student[i]` as the "name" of that particular student (like student) — 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 ints named greenEggs, of size 5?

A. int greenEggs = new array(5);
B. int[] greenEggs = new array(5);
C. int[] greenEggs = new int[5];
D. 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:
  // 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]);

Arrays as Parameters (1/3)
- Can pass entire array as parameter by adding array brackets to type inside signature
  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
    int arrayLength = <array-name>.length;

Arrays as Parameters (3/3)
- How does .length work in actual code?
  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;
  }

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?

Cats with Hats - PaneOrganizer

```java
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());
    }
    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!
  - Java throws `ArrayIndexOutOfBoundsException` if index is negative since sequence starts at 0
  - You'll receive an "arrayIndexOutOfBoundsException" runtime error
  - 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

ArrayIndexOutOfBoundsExceptions

```java
public class PaneOrganizer {
    private Cat[] _cats;
    /* You use the constructor */
    public PaneOrganizer() {
        _cats = new Cat[10];
        this.unexpected();
    }
    public void unexpected() {
        for(int i = 0; i < _cats.length; i++) {
            _cats[i] = new Cat();
        }
    }
}
```

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:

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

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
```

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

```
[1] [2] [3]                  [1] [2] [3]
[7] [8] [9]                  [7] [8] [9]
```

- 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]

Common Array Errors - Watch Out! (1/2)
- Assigning a scalar to an array
  ```java
type 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];
  ```

Common Array Errors - Watch Out! (2/2)
- Assigning arrays to scalars
- Assigning arrays of different dimension to each other
- 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:

Quick Look at the Design (1/2)
- Some things we've seen before:
  - java program - creates javafx.stage.Stage
  - buttons - uses javafx.scene.control.Button
  - red, black, white, blue - javafx.scene.paint.Colors
- New things:
  - sixty-four squares - we know about one square Shape.Rectangle, but 64?
  - checker board - lets 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
  - private Color _primaryColor;
  - private Color _secondaryColor;
  - private Rectangle _rect;

- public void toggleColor();
  - _primaryColor = _secondaryColor;
  - _secondaryColor = _primaryColor;

- public Rectangle getRect();
  - _rect = new Rectangle(_primaryColor, _secondaryColor);

- public void actionEvent(ActionEvent event) {
  - _rect.setFill(_primaryColor);
  - _rect.setStroke(_secondaryColor);
}

---

**Building Checkerboard (1/2)**

- Let's start with standard stuff
  - 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

---

**Building Checkerboard (2/2)**

- public class CheckerBoard {
  - private CheckerSquare[][] _rects;
  - public CheckerBoard();
  - _rects[x][y] = new CheckerSquare();
  - for (int x = 0; x < NUM_SQRS; x++) {
    - for (int y = 0; y < NUM_SQRS; y++) {
      - _rects[x][y] = new CheckerSquare();
    }
  }

- private void toggleColor(int x, int y) {
  - _rects[x][y].toggleColor();
}

- public void actionEvent(ActionEvent event) {
  - _rects[event.getX()][event.getY()].actionEvent(event);
}

---

**Updating CheckerSquares**

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

  ```java
  public void handle(ActionEvent event) {
    CheckerSquare square = (CheckerSquare) event.getSource();
    square.toggleColor();
  }
  ```

- What would happen if we didn't check for null? What might happen if we didn't check for the board's size before accessing its contents?

---

**PaneOrganizer class**

- public PaneOrganizer() {
  - for (int x = 0; x < NUM_SQRS; x++) {
    - for (int y = 0; y < NUM_SQRS; y++) {
      - _rects[x][y] = new CheckerSquare();
    }
  }

- private Color _primaryColor;
  - public void setPrimaryColor(Color color) { _primaryColor = color; }

- private Color _secondaryColor;
  - public void setSecondaryColor(Color color) { _secondaryColor = color; }

- private void toggleColor() {
  - _primaryColor = _secondaryColor;
  - _secondaryColor = _primaryColor;
}

- public void actionEvent(ActionEvent event) {
  - _rects[event.getX()][event.getY()].toggleColor();
  - PaneOrganizer actionEvent(event);
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 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 lists 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 homogeneous 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 class file defines array list of ElementType, but when ArrayList get declared, type must get specified (e.g., an ArrayList of Truffula for the Lorax).
  - Java replaces ElementType with Truffula 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

//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

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 automatically 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 automatically
- 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>();
  - Remember to use literal <> for specialized type!

Example (1/6) Lorax vs. Once-ler

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(){
    //SceneGraph code...
    for (int i=0; i<500; i++)
      _truffulas.add(new Truffula());
  }
} //class definition continued on next slide

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<String> seussQuotes = new ArrayList<ElementType>();
   String x = seussQuotes.first();
D. ArrayList<String> seussQuotes = new ArrayList<String>
   String x = seussQuotes.get(0);...
Example (2/6) Lorax vs Once-ler

```java
// Abb a new Truffula at the end
class OnceLer {
    public void plantTrees(TruffulaTree truffulaTree) {
        _truffulas.add(truffulaTree);
    }
    // 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)
  ```
  o "The method add(Truffula) in the type ArrayList<Truffula> is not applicable for the arguments (int)"

Example (4/6) Lorax vs Once-ler

```java
public class OnceLer {
    private Truffula _currentTruffula;
    private ArrayList<Truffula> _truffulas;
    public TruffulaIfTruffulaOrtrees() {
        /* Some code does not contain the TruffulaForest; */
        _currentTruffula = new Truffula();
    }
    // Method to add new Truffulas at the end of the current TruffulaOrtrees
    public void addNew(int numTrees) {
        System.out.print("A thneed is what you need!");
        _truffulas.add(new Truffula());
    }
}
```

Example (5/6) Lorax vs Once-ler

```java
for vs. for-each loop (1/4)

- Consider this for loop:
  ```java
  for (int i = 0; i < whos.size(); i++) {
      if (i % 2 == 0) {// if index 'i' is even
          // sing() is defined in whos
          whos.get(i).sing();
      }
  }
  ```
- Only want to call sing() on elements at even indices, but for-each loop wouldn't work
  o 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

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

  - types: 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 vs. for-each loop (4/4)

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

  ```java
  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)

```java
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

  ```java
  if (two or more arguments get passed into main line, compiler would raise
  ArrayIndexOutOfBoundsException! Why?
  ```

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

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

Announcements

- Cartoon deadlines
  - check-ins (10/13-10/15)
  - early handin (10/18) Tuesday 11:59PM
  - on-time handin (10/20) 10:00PM
  - 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
  - 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
  - you are expected to use eclipse to write code from here on out