AP Computer Science A Exam Review

What you ABSOLUTELY MUST KNOW above everything else

(according to the Barron's test prep book)

The Basics

- Every AP exam question uses at least one of these:
  - Types and Identifiers
  - Operators
  - Control structures

Identifiers

- Identifiers – name for variable, parameter, constant, user-defined method/class, etc.
  - Convention says identifiers for variables and methods will be lowercase (with uppercase letters to separate multiple words)
  - E.g. `getName, findSurfaceArea, preTaxTotal`
  - Class names will be capitalized
  - E.g. `Student, Car, BankAccount`

Built-in/Primitive Types

- **int**
  - an integer, e.g. 5, -77, 9001
- **boolean**
  - a boolean, true or false
- **double**
  - a double precision floating-point number, 2.718, -3456.78, 1.4e5

Storage of numbers

- int types use 32 bits, largest integer is 2^{31} - 1
- Floating-point numbers use mantissa and exponent: sign*mantissa*2
- When floating point numbers are converted to binary, most cannot be represented exactly, leading to **round-off error**

Miscellaneous Note #1

- Integer.MIN_VALUE and Integer.MAX_VALUE
  - Represent the absolute lowest and highest values that can be stored in an integer
  - If you're trying to find the minimum or maximum value of an array or ArrayList, initialize your variable to these

Number representation

- Decimal – base 10 – regular numbers
- Binary – base 2 – zeros and ones
- Hexadecimal – base 16 – 0-F
- Octal – base 8 – 0-7
- $. \quad . 25$
  - 11001
  - 19_h
  - 31

Final Variables

- final variable is a quantity whose value will not change
- E.g. final int CLASS_SIZE = 30

Arithmetic Operators

Operator	Meaning	Example
+	Addition	3 + x
-	Subtraction	p – q
*	Multiplication	6 * i
/	Division	10 / 4 //returns 2
%	Mod (remainder)	11 % 8 //returns 3

Arithmetic Operators Notes

- Integer division truncates the answer (cuts off the decimal)
- Use type casting to control how to divide.
- Which do not evaluate to 0.75?
  - 3.0 / 4
  - 3 / 4.0
  - (int) 3.0 / 4
  - (double) 3 / 4
  - (double) (3 / 4)

Relational Operators

Operator	Meaning	Example
==	Equal to	if (x == 100)
!=	Not equal to	if (age != 21)
>	Greater than	if (salary > 30000)
<	Less than	if (grade < 65)
>=	Greater than or equal to	if (age >= 16)
<=	Less than or equal to	if (height <= 6)

Relational Operators Notes

- Should only be used for primitive types (i.e. int, double, boolean)
- Be careful for comparing floating-point numbers due to round-off error

Logical Operators

Operator	Meaning	Example
!	NOT	if (!found)
&&	AND	if (x < 3 && y > 4)
||	OR	if (age < 2 || height < 4)

Logical Operators Example

- (x && y) || !(x && y)
  A. Always true
  B. Always false
  C. true only when x is true and y is true
  D. true only when x and y have the same value
  E. true only when x and y have different values

Another example

- Which is equivalent to:
  !(a < b) && !(a > b)
  A. true
  B. false
  C. a == b
  D. a != b
  E. !(a < b) && (a > b)

Assignment Operators

Operator	Example	Meaning
=	x = 2	Simple assignment
+=	x += 4	x = x + 4
-=	y -= 6	y = y – 6
*=	p *= 5	p = p * 5
/=	n /= 10	n = n / 10
%=	n %= 10	n = n % 10
++	k++	k = k + 1
--	i--	i = i - 1

Operator Precedence

- (1) !, ++
- (2) *, /, %
- (3) +, -
- (4) <, >, <=, >=
- (5) ==, !=
- (6) &&
- (7) ||
- (8) =, +=, -=, *=, /=, %=

Control Structures

- if
- if…else
- if…else if
- while loop
- for loop
- for-each loop

if Statement

if (*boolean expression*)

{

statements

}

//statements will be executed if *boolean expression* is true

if example

- 3 bonus questions → must get all correct for 5 points added to **`grade`**
- **`bonus1`**, **`bonus2`**, and **`bonus3`** are `boolean` variables that indicate whether they are correct
- Write an if statement for this example
  - if(bonus1 && bonus2 && bonus3)
    grade += 5;

if…else Statement

```
if (boolean expression)
{
   statements
   //will be executed if boolean expression is true
}
else
{
   statements
   //will be executed if boolean expression is false
}
```

if…else if

```
if (grade.equals(“A”))
    System.out.println(“Excellent”);
else if(grade.equals(“B”))
    System.out.println(“Good”);
else if(grade.equals(“C”))
    System.out.println(“Poor”);
else
    System.out.println(“Invalid”);
```

Other if Notes

- if/if…else if statements do not always need an else at the end
- An if statement inside of an if statement is a nested if statement:
  if (*boolean expr1*)
  if (*boolean expr2*)
  statement;
- Can also be written as:
  if (*boolean expr1 && boolean expr2*)
  statement;

Rewrite using only if…else

- if(value < 0)
  return “Not in range”;
  else if(value > 100)
  return “Not in range”;
  else
  return “In range”;
- if(value < 0 || value > 100)
  return “Not in range”;
  else
  return “In range”;

While vs. For loops

While loops

- ```
  int i = 0;
  while (i < 100)
  {
      //repeated code
      i++;
  }
  ```

For loops

- ```
  for(int i = 0; i<100; i++)
  {
      //repeated code
  }
  ```

While vs. For-each loops

While loop

- int[] locationCells
  = new int[100];
  int i = 0;
  while (i < 100)
  {
  System.out.println
  (locationCells[i]);
  i++;
  }

For-each loop

- int[] locationCells
  = new int[100];
  for(int cell : locationCells)
  {
  System.out.println(cell);
  }

For loop vs. for-each loop

For loop

- ```
  for(int i = 0; i<100; i++)
  {
        System.out.println
            (locationCells[i]);
  }
  ```

For-each loop

- ```
  for(int cell : locationCells)
  {
     System.out.println(cell);
  }
  ```

For loop vs. for-each loop

For loop

- Has an index → useful for setting data that depends on the index
- Initialization, boolean test, and iteration expression are all in one line

For-each loop

- Easier to write when simply accessing data from an array
- Not much better than a while loop if not accessing array data

While loop example

- int value = 15;
  while (value < 28) {
  System.out.println(value);
  value++;
  }
- What is the first number printed?
  - 15
- What is the last number printed?
  - 27

Another example

```
int a = 24;
int b = 30;
while (b != 0) {
    int r = a % b;
    a = b;
    b = r;
}
System.out.println(a);
```

A. 0
B. 6
C. 12
D. 24
E. 30

Yet another example

```
int k = 0;
while (k < 10) {
    System.out.print
            ((k % 3) + “  ”);
    if ((k % 3) == 0)
        k = k + 2;
    else
        k++;
}
```

A. 0 2 1 0 2

B. 0 2 0 2 0 2

C. 0 2 1 0 2 1 0

D. 0 2 0 2 0 2 0

E. 0 1 2 1 2 1 2

For loop example

- String str = “abcdef”;
  for (int r = 0; r < str.length()-1; r++)
  System.out.print(str.substring(r, r+2));
- What is printed?
  A. abcdef
  B. aabbccddeeff
  C. abbccddeef
  D. abcbcdcdedef
  E. Nothing, IndexOutOfBoundsException thrown

Another example

```
public void numberCheck(int maxNum) {
   int typeA = 0;
   int typeB = 0;
   int typeC = 0;
   for (int k = 1; k<=maxNum; k++) {
      if (k%2 == 0 && k%5==0)
         typeA++;
      if (k%2 == 0)
         typeB++;
      if (k%5 == 0)
         typeC++;
   }
   System.out.println(typeA + " " + typeB
                                  + " " + typeC);
}
```

What is printed by numberCheck(50)?

A. 5 20 5
B. 5 20 10
C. 5 25 5
D. 5 25 10
E. 30 25 10

Yet another example

- for (int outer = 0; outer < n; outer++)
  for(int inner = 0; inner <= outer; inner++)
  System.out.print(outer + “ ”);
- If n has a value of 4, what is printed?

A. 0 1 2 3

B. 0 0 1 0 1 2

C. 0 1 2 2 3 3 3

D. 0 1 1 2 2 2 3 3 3 3

E. 0 0 1 0 1 2 0 1 2 3

Objects, Classes, and Inheritance

- You may have to write your own class. You'll definitely need to interpret at least one class that's given. Very common, esp. on FRQ.
  - Methods
  - Subclasses
  - Abstract classes
  - Interfaces

Method Headers

- With the exception of constructors, all method headers should have these 4 things:
  - **Access modifier**: public, private
  - **Return type**: void, int, double, boolean, SomeType, int[], double[], Pokemon[], etc.
  - **Method name**: e.g. withdraw
  - **Parameter list**: e.g. String pass, double amt
  - (Some methods may also be static)
- public void withdraw(String pass, double amt)

Types of Methods

- **Constructors** → create an object of the class
  - public BankAccount()
- **Accessor** → gets data but doesn't change data
  - public double getBalance()
- **Mutator** → changes instance variable(s)
  - public void deposit(String pswd, double amt)
- **Static methods** → class methods, deals with class variables
  - public static int getEmployeeCount()

Static methods in Driver class

- **Methods in the driver class** (the class that contains your main() method) are **usually all static** because there are not instances of that class.
- public static void main(String[] args)

Method Overloading

- **Two or more methods with the same name but different parameter lists**
  - public int product(int n) {return n*n;}
  - public double product(double x) {return x*x;}
  - public double product(int x, int y){return x*y;}

- (return type is irrelevant for determining overloading)

Inheritance

- **Inheritance** is where a **subclass** is created from an existing **superclass**.
- The subclass copies or inherits variables and methods from the superclass
- Subclasses usually contain more than their superclass
- Subclasses can be superclasses for other subclasses

Class hierarchy - Which is true?

A. Superclass should contain the data and functionality that are common to all subclasses that inherit from the superclass
B. Superclass should be the largest, most complex class from which all other subclasses are derived
C. Superclass should contain the data and functionality that are only required for the most complex class
D. Superclass should have public data in order to provide access for the entire class hierarchy
E. Superclass should contain the most specific details of the class hierarchy

Implementing Subclasses

- Subclasses copy everything except what?
- public class Superclass {
  //superclass variables and methods
  }
- public class Subclass extends Superclass {
  //copies everything from Superclass
  //**EXCEPT constructors**
  }

Inheriting Instance Methods/Variables

- Subclasses **cannot directly access private variables** if they are inherited from a superclass
- Subclasses must use the public accessor and mutator methods
- (Subclasses can directly access if variables are protected but protected is not in the AP Java subset.)

Method Overriding and super

- If a method has the same name and parameter list in both the superclass and subclass, the **subclass method overrides the superclass method**
- To invoke the method from the superclass, use the keyword **super**
- E.g. if the superclass has a computeGrade() method, use super.computeGrade()
- If you are invoking the constructor use super() or super(*parameters*)

Rules for Subclasses

- Can add new private instance variables
- Can add new public, private, or static methods
- Can override inherited methods
- May not redefine a public method as private
- May not override static methods of superclass
- Should define its own constructors
- Cannot directly access the private members of its superclass, must use accessors or mutators

Declaring Subclass Objects

- Superclass variables **can reference both superclass objects and subclass objects**
- Which of these is not valid:
  - Student c = new Student();
  - Student g = new GradStudent();
  - Student u = new UnderGrad();
  - UnderGrad x = new Student();
  - UnderGrad y = new UnderGrad();

Polymorphism

- **Method overridden** in at least one subclass is polymorphic
- What are method calls are determined by?
  - the type of the actual object
  - the type of object reference
- Selection of correct method occurs **during the run** of the program (dynamic binding)

Type Compatibility

- **Only polymorphic if method is overridden**
- E.g. if GradStudent has a getID method but Student does not, this will lead to a compile-time error:
  Student c = new GradStudent();
  int x = c.getID(); //compile-error
- You can cast it as a subclass object to fix the error:
  int x = ((GradStudent) c).getID();

Abstract Class

- Superclass that represents an abstract concept
- Should never be instantiated
- May contain abstract methods
  - When no good default code for superclass
- Every subclass will override abstract methods
- If class contains abstract methods, it must be declared an abstract class

Notes about abstract classes

- Can have both abstract and non-abstract methods
- Abstract classes/methods are declared with keyword abstract
- Possible to have abstract class without abstract methods
- Abstract classes may or may not have constructors
- Cannot create abstract object instances
- Polymorphism still works

Interfaces

- Collection of related methods whose headers are provided without implementations
- Classes that implement interfaces can define any number of methods
- Contracts to implement all of them; if cannot implement all, must be declared an abstract class
- Interface keyword for interfaces; implements keyword for class that implement them
- Class can extend a superclass and implement an interface at the same time:
  public class Bee extends Insect implements FlyObject

Interface vs. Abstract Class

- Use abstract class for object that is application-specific, but incomplete without subclasses
- Consider interface when methods are suitable for your program but also equally applicable in a variety of programs
- Interface cannot provide implementations for any of its methods; abstract class can
- Interface cannot have instance variables; abstract class can
- Both can declare constants
- Both cannot create an instance of itself

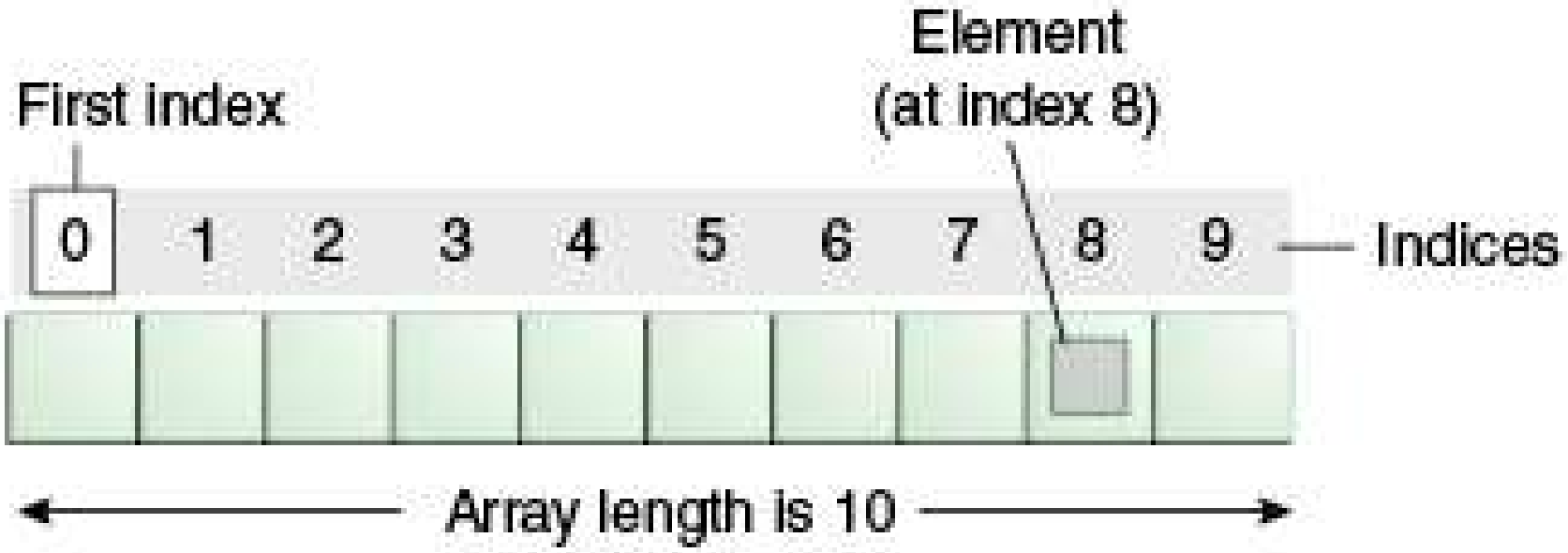
Miscellaneous Note #2

- List → basically the same as ArrayList
  - List is an interface
  - ArrayList implements List

Lists and Arrays

- Manipulate a list. Search, delete and insert an item. Very common on the AP exam.
  - One-dimensional arrays
  - ArrayLists
  - Two-dimensional arrays

1-D Arrays

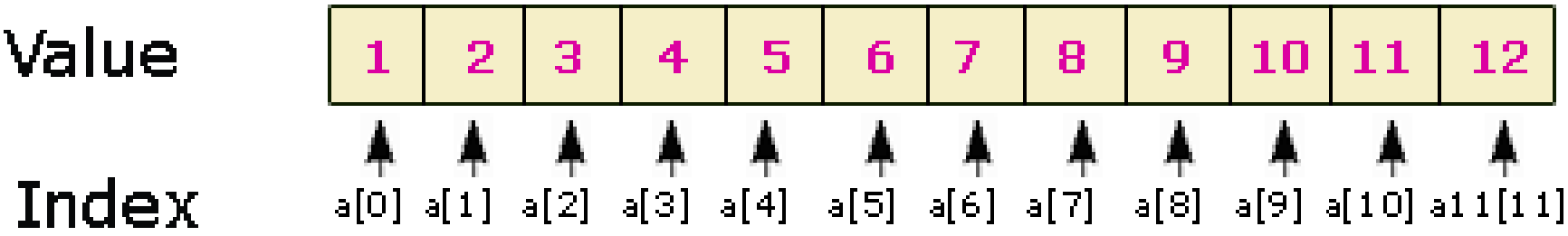

Element
(at index 8)
First index
0
1
2
3
4
5
6
7
8
9
Indices
Array length is 10

1-D Array Initialization

- Which of these are valid ways to assign a reference to an array?
  - double[] data = new double[25];
  - double data[] = new double[25];
  - double[] data;
    data = new double[25];
- All are three are valid!

One Dimensional array

Initialization `int a[] = new int [12];`




```
System.out.print(a[5]);
```

Output: 6

1-D initializing values

- Small arrays can have values initialized with either of the following ways:
  - int[] coins = new int[4];
    coins[0] = 1;
    coins[1] = 5;
    coins[2] = 10;
    coins[3] = 25;
  - int[] coins = {1, 5, 10, 25};

Array Length

- length is a public instance variable of arrays:
  String[] names = new String[25];
  names.length; //returns 25
- Array indices go from 0 to names.length-1 (i.e. 0 to 24)
- length is not a method for arrays; length is a method for Strings

Traversing an Array

- Use for-each loop when you need to access (only access) every element in an array without replacing or removing elements
- Use for loop for all other cases

What to do with arrays

- You need to be able to read and write code that accomplishes each of the following:
  - Counting elements
  - Printing elements
  - Summing elements
  - Swapping elements
  - Finding the minimum or maximum
  - Inserting elements
  - Deleting elements

Counting & Printing

- Counting:

```
int total = 0;
for(int i = 0; i<arr.length; i++) {
    total++;
}
```

- Printing:

```
for(int i = 0; i<arr.length; i++) {
    System.out.println(arr[i]);
}
```

Summing Values

- The method calcTotal is intended to return the sum of all values in vals.
- private int[] vals;
  public int calcTotal() {
  int total = 0;
  /* missing code */
  return total;
  }
- What code should replace /* missing code */ in order for calcTotal to work correctly?

Summing Values

- ```
  private int[] vals;
  public int calcTotal() {
      int total = 0;
      for(int pos = 0; pos < vals.length; pos++) {
          total += vals[pos];
      }
      return total;
  }
  ```


  ```

Summing Values

```
private int[] vals;
public int calcTotal() {
    int total = 0;
    int pos = 0;
    while (pos < vals.length) {
        total += vals[pos];
        pos++;
    }
    return total;
}
```

Swapping values

- int[] arr = new int[10];
- How to swap arr[0] and arr[5]?

A.
```
arr[0] = 5;
arr[5] = 0;
```

B.
```
arr[0] = arr[5];
arr[5] = arr[0];
```

C.
```
int k = arr[5];
arr[0] = arr[5];
arr[5] = k
```

D.
```
int k = arr[0];
arr[0] = arr[5];
arr[5] = k;
```

E.
```
int k = arr[5];
arr[5] = arr[0];
arr[0] = arr[5];
```

Min and Max

- ```
  int min = arr[0];
  for(int j = 0; j<arr.length; j++){
      if (arr[j]<min)
          min = arr[j];
  }
  ```
- ```
  int max = arr[0];
  for(int j = 0; j<arr.length; j++){
      if (arr[j]>max)
          max = arr[j];
  }
  ```

Arrays vs. ArrayList

Arrays

```
String[] arr = new String[10];
…
//insert Strings into array
…
for(int i=0; i<arr.length; i++)
{
   System.out.println(arr[i]);
}
```

ArrayList

```
ArrayList<String> arrList =
new ArrayList<String>();
…
//insert Strings into ArrayList
…
for(int i=0; i<arr.size(); i++)
{
   System.out.println
                  (arrList.get(i));
}
```

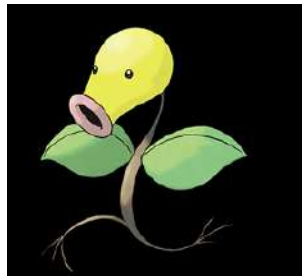
Arrays vs. ArrayList

Arrays

```
String[] arr = new String[10];
…
//insert Strings into array
…
for(String x : arr)
{
   System.out.println(x);
}
```

ArrayList

```
ArrayList<String> arrList =
new ArrayList<String>();
…
//insert Strings into ArrayList
…
for(String x : arrList)
{
 System.out.println(x);
}
```



Arrays vs. ArrayList

Arrays

Fixed length, set when it is created

Must keep track of last slot if array is not full

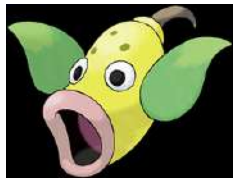
Must write code to shift elements if you want to insert or delete

ArrayList

Shrinks and grows as needed

Last slot is always arrList.size()-1

Insert with just arrList.add(object)
Delete with just arrList.remove(objectIndex) or arrList.remove(object)



Insert and Delete

- If asked to insert or delete for arrays, you'll likely need to create a new array
- More likely asked about ArrayLists
  - ArrayLists can change length more easily

ArrayList Question

- ArrayList<String> items =
  new ArrayList<String>();

```
items.add("A");
items.add("B");
items.add("C");
items.add(0, "D");
items.remove(3);
items.add(0, "E");
System.out.println(items);
```

A. [A, B, C, E]
B. [A, B, D, E]
C. [E, D, A, B]
D. [E, D, A, C]
E. [E, D, C, B]

Another ArrayList Question

```
public void replace(ArrayList<String> nameList,
                                String name, String newValue) {
    for (int j = 0; j<nameList.size(); j++) {
        if( /* expression */ )
            nameList.set(j, newValue);
    }
}
```

What should be used to replace **/*expression*/** so that the **replace** method will replace all occurrences of **name** in **nameList** with **newValue**?

- nameList.get(j).equals(name)

removeAll

- Write the removeAll method that will remove all instances of String str from ArrayList arrList and return the number of items removed

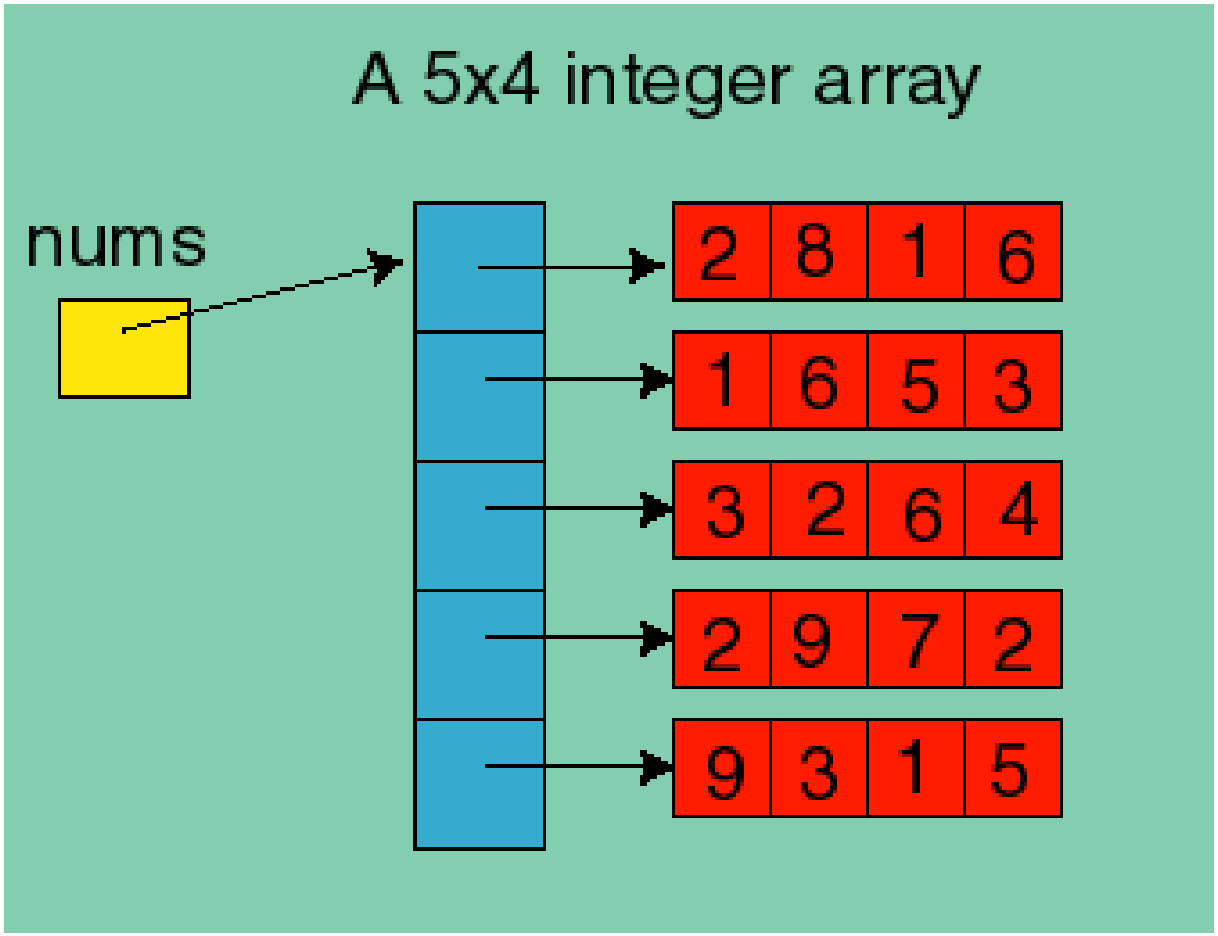
- public int removeAll(ArrayList<String> arrList,
  String str) {
  /* complete this method */
  }

removeAll

- public int removeAll(ArrayList<String> arrList,
  String str) {
  int numRemoved = 0;
  for (int i = arrList.size()-1; i>=0; i--) {
  if(str.equals(arrList.get(i)) {
  numRemoved += 1;
  arrList.remove(i);
  }
  }
  }

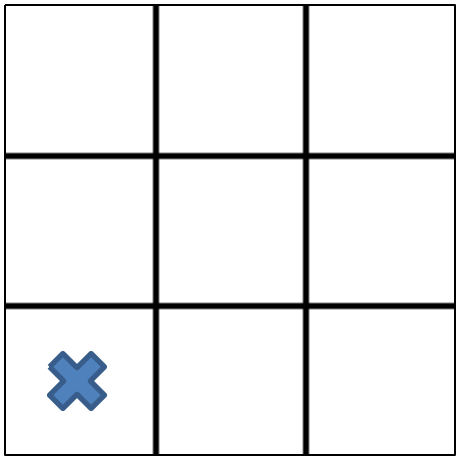
2-D Arrays

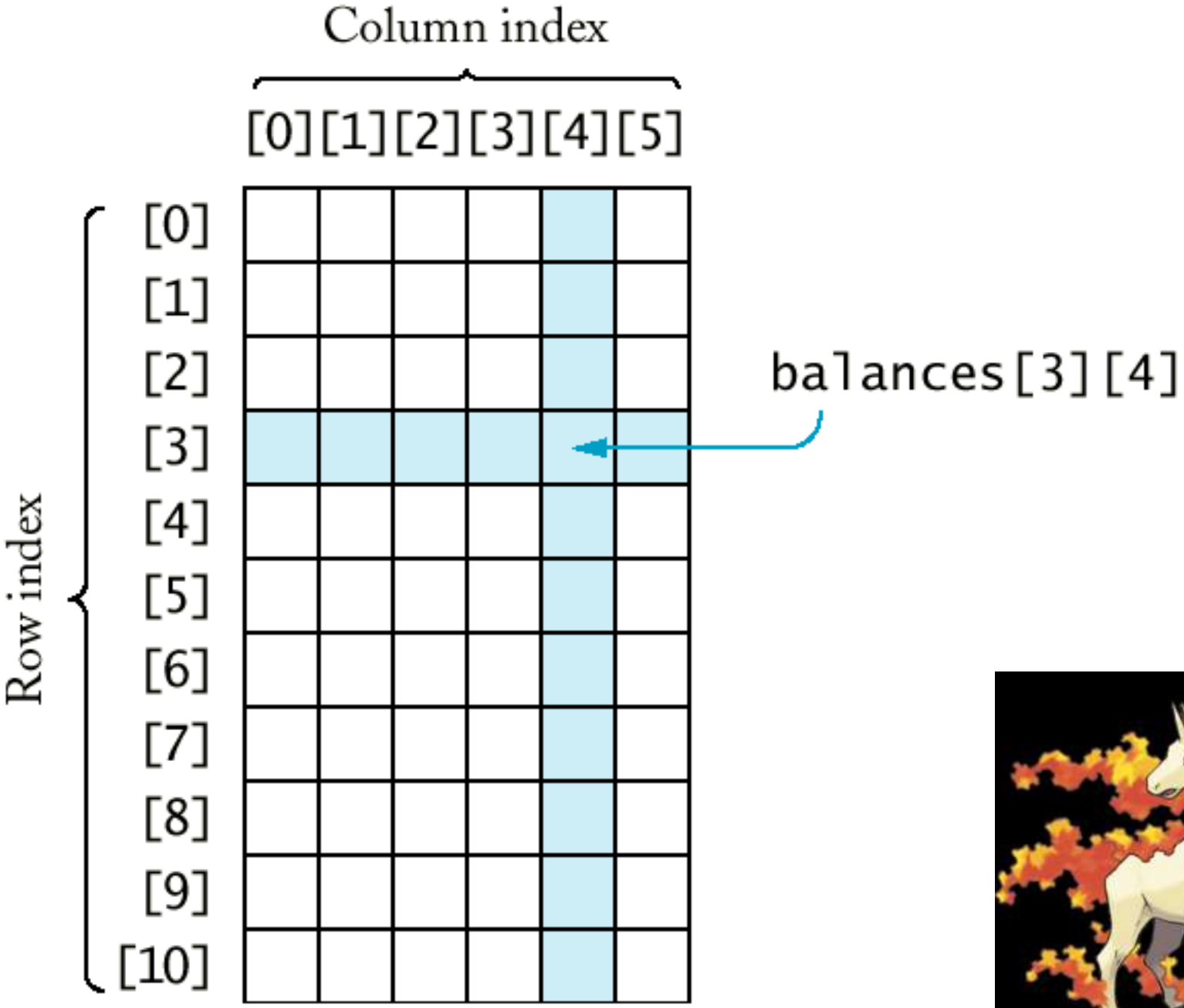
- `int[][] nums = new int[5][4];`



2-D Array as a table

```
int numRows = 3;
int numCols = 3;
String[][] table =
           new String[numRows][numCols];
table[2][0] = “x”;
```




Column index
[0][1][2][3][4][5]
Row index
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
balances[3][4]

2-D Array Question

- public void alter(int c) {
  for (int i = 0; i<mat.length; i++)
  for (int j = c+1; j<mat[0].length; j++)
  mat[i][j-1] = mat[i][j];
  }
- mat is a 3x4 matrix with values:

  1 3 5 7
  2 4 6 8
  3 5 7 9
- alter(1) will change mat to what?
- 1 5 7 7
  2 6 8 8
  3 7 9 9

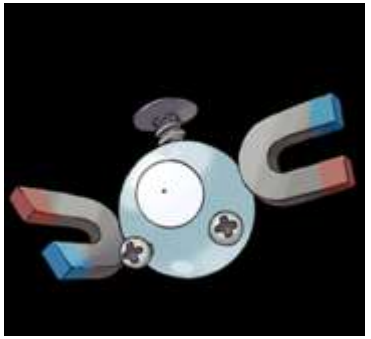
Sorting and Searching

- Know these algorithms; at least one or two questions on AP exam
  - Selection Sort
  - Insertion Sort
  - Merge Sort
  - Binary Search

Selection Sort Algorithm (ascending)

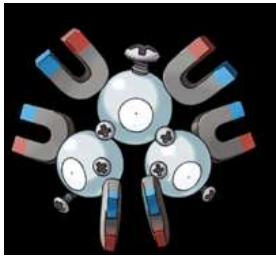
"Search and swap" algorithm:

1. Find smallest element (of remaining elements).
2. Swap smallest element with current element (starting at index 0).
3. Finished if at the end of the array. Otherwise, repeat 1 and 2 for the next index.



Selection Sort Example(ascending)

- 70 75 89 61 37
  - Smallest is 37
  - Swap with index 0
- 37 75 89 61 70
  - Smallest is 61
  - Swap with index 1
- 37 61 89 75 70
  - Smallest is 70
  - Swap with index 2
- 37 61 70 75 89
  - Smallest is 75
  - Swap with index 3
    - Swap with itself
- 37 61 70 75 89
  - Don't need to do last element because there's only one left
- 37 61 70 75 89



Selection Sort Question

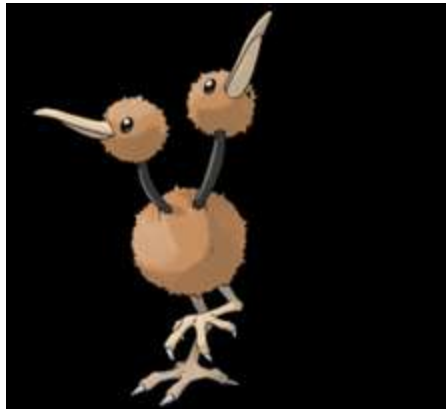
- In an array of Integer contains the following elements, what would the array look like after the third pass of selectionSort, sorting from high to low?

89 42 -3 13 109 70 2

A. 109 89 70 13 42 -3 2
B. 109 89 70 42 13 2 -3
C. 109 89 70 -3 2 13 42
D. 89 42 13 -3 109 70 2
E. 109 89 42 -3 13 70 2

Selection Sort Notes

- For an array of n elements, the array is sorted after n-1 passes
- After the kth pass, the first k elements are in their final sorted position
- Inefficient for large n



Insertion Sort Algorithm (ascending)

- Check element (store in temp variable)
- If larger than the previous element, leave it
- If smaller than the previous element, shift previous larger elements down until you reach a smaller element (or beginning of array). Insert element.

Insertion Sort Algorithm (ascending)

- 64 54 18 87 35
  - 54 less than 64
  - Shift down and insert 54
- 54 64 18 87 35
  - 18 less than 64
  - 18 less than 54
  - Shift down and insert 18
- 18 54 64 87 35
  - 87 greater than 64
  - Go to next element
- 18 54 64 87 35
  - 35 less than 87
  - 35 less than 64
  - 35 less than 54
  - 35 greater than 18
  - Shift down and insert 35
- 18 35 54 64 87

Insertion Sort Question

- When sorted biggest to smallest with insertionSort, which list will need the greatest number of changes in position?

A. 5,1,2,3,4,7,6,9

B. 9,5,1,4,3,2,1,0

C. 9,4,6,2,1,5,1,3

D. 9,6,9,5,6,7,2,0

E. 3,2,1,0,9,6,5,4

Insertion Sort Question

- A worst case situation for insertion sort would be which of the following?
  - A list in correct sorted order
  - A list sorted in reverse order
  - A list in random order

Insertion Sort Notes

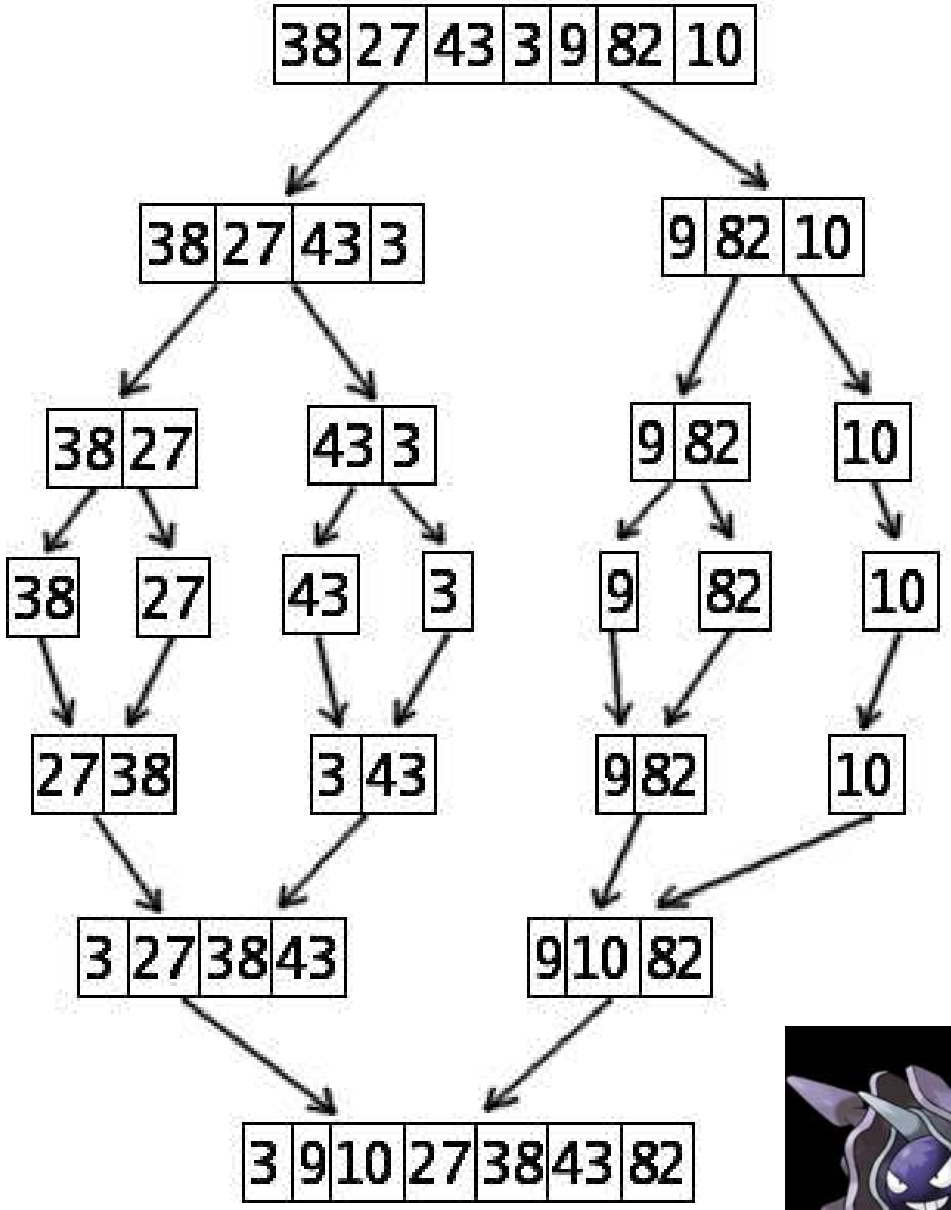
- For an array of n elements, the array is sorted after n-1 passes
- After the kth pass, a[0], a[1],…, a[k] are sorted with respect to each other but not necessarily in their final sorted positions
- Worst case occurs if array is initially sorted in reverse order
- Best case occurs if array is already sorted in increasing order
- Inefficient for large n

Merge Sort Algorithm

- The idea behind merge sort is **divide and conquer**
  1. Divide data into 2 equal parts
  2. Recursively sort both halves
  3. Merge the results

Merge Sort Example

1. Divide data into 2 equal parts
2. Recursively sort both halves
3. Merge the results



Merge Sort Example

- 8 45 87 34 28 45 2 32 25 78
- 8 45 87 34 28 | 45 2 32 25 78
- 8 45 87 | 34 28 | 45 2 32 | 25 78
- 8 45 | 87 | 34 | 28 | 45 2 | 32 | 25 | 78
- 8 | 45 | 87 | 34 | 28 | 45 | 2 | 32 | 25 | 78
- 8 45 | 87 | 28 34 | 2 45 | 32 | 25 78
- 8 45 87 | 28 34 | 2 32 45 | 25 78
- 8 28 34 45 87 | 2 25 32 45 78
- 2 8 25 28 32 34 45 45 78 87

Merge Sort Notes

- Main disadvantage of Merge sort is use of a temporary array→problem if space is a factor
- Merge sort is not affected by the initial ordering of the elements→best and worst case take the same amount of time

Merge Sort Question

- Which of the following is a valid reason why mergesort is a better sorting algorithm than insertion sort for sorting long lists?
  - Mergesort requires less code than insertion sort
  - Mergesort requires less storage space than insertion sort
  - Mergesort runs faster than insertion sort

Binary Search

- Check middle element
  - Is this what we're looking for? If so, we're done.
  - Does what we're looking for come before or after?
- Throw away half we don't need
- Repeat with half we do need

- What does this look like in Java?

Recursive

- Binary search is recursive
- Uses indices to know where to search in the array
- Calculate an index midway between the two indices
- Determine which of the two subarrays to search
- Recursive call to search subarray

Binary Search Question

Which of the following is **not true** of a binary search of an array?

A. The method involves looking at each item in the array, starting at the beginning, until either the value being searched for is found or it can be determined that the value is not in the array.
B. In order to use the binary search, the array must be sorted first.
C. The method is referred to as “divide and conquer.”
D. An array of 15 elements requires at most 4 comparisons.
E. Using a binary search is usually faster than a sequential search.

How many executions?

- Check how many halves you threw away + 1
- Or check how many times you checked a middle element

Binary Search Question

- A binary search is to be performed on an array with 600 elements. In the worst case, which of the following best approximates the number of iterations of the algorithm?

A. 6
B. 10
C. 100
D. 300
E. 600

Binary Search Question

- Consider a binary search algorithm to search an ordered list of numbers. Which of the following choices is closest to the maximum number of times that such an algorithm will execute its main comparison loop when searching a list of 1 million numbers?

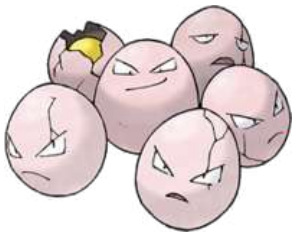
A. 6
B. 20
C. 100
D. 120
E. 1000

The GridWorld Case Study

- Bugs and Critters are 25% of the exam
  - Bug Class
  - BoxBug Class
  - Critter Class
  - ChameleonCritter Class

Bug Methods

- public Bug()
  - Default constructor. Creates red Bug.
- public Bug(Color bugColor)
  - Constructor. Create Bug with said color.
    - Note: both constructors use inherited setColor method
- public void act()
  - Bug moves if possible; otherwise turns.
    - Note: only overridden method of Actor class
- public void turn()
  - Turns Bug 45^{o} to right without changing location (i.e. add Location.HALF_RIGHT to current direction)



Bug Methods (cont.)

- public boolean canMove()
  - Get bug's grid, if null, return false (null if another actor removed the bug).
  - Get adjacent location in front of bug.
  - If location is invalid, return false.
  - Get actor in neighboring location.
  - If actor is flower or null, return true. Otherwise, return false.

Bug Methods (cont.)

- public void move()
  - Get bug's grid, if null, exit method.
  - Get bug's current location
  - Get adjacent location in front of bug
  - If location is valid, bug moves there. Otherwise, removes itself from grid.
  - Creates a flower that is same color as bug
  - Place flower in bug's previous location

BoxBug Variables

- sideLength
  - number of steps in a side of its square
- steps
  - keeps track of where the BoxBug is in creating a side

BoxBug Methods

- public BoxBug(int length)
  - Constructor. Set sidelength to said length and initializes steps to 0
- public void act()
  - overridden method performs one step each call
  - checks to see if steps<sidelength and canMove
  - if true, move; else, turn twice

Critter Class

- No explicit constructor; uses default constructor
- Extends Actor with following behavior:
- public void act()
  - Get a list of neighboring actors
  - Process actors
  - Get list of possible locations to move to
  - Select location from list
  - Move to location

Critter Methods

- public void act()
- public ArrayList<Actor> getActors()
  - returns list of adjacent neighboring actors
- public void processActors(ArrayList<Actors> actors)
  - Processes actors by iterating through list. Critter “eats” all actors that are not rocks or other Critters (actors remove themselves from the grid).
- public ArrayList<Location> getMoveLocations()
  - returns list of valid, adjacent, empty, neighboring locations, which are possible for the next move. getEmptyAdjacentLocations is used with current location

Critter Methods (cont.)

- public Location selectMoveLocation
  (ArrayList<Location> locs)
  - Assign n as length of list
  - If n is 0 (no valid locations), return current location
  - Get random int from 0 to n-1
  - return location at index r in locs ArrayList
- public void makeMove(Location loc)
  - moves Critter to said location with moveTo(loc)

ChameleonCritter

- public void processActors
  (ArrayList<Actor> actors)
  - Randomly selects adjacent neighbor, changes to neighbor's color. Does nothing if no neighbors
- public void makeMove(Location loc)
  - Before moving, turns to face new location
  - sets the direction to getLocation().getDirectionToward(loc)