

# More AP Review

By Mr. Fahrenbacher

# Short Circuit Evaluation

- ♦ Boolean expressions are short circuited so that they evaluate faster and safer.

```
if( a || b) {  
    //code  
}
```

When a is true, b's value is never checked!

When a is false, b's value is always checked.

# Short Circuit Evaluation

```
if( x == 0 || 8/x > 3) {  
    S.o.p("Hi!");  
}
```

```
if(8/x > 3 || x == 0) {  
    S.o.p("Ho!");  
}
```

| x | output |
|---|--------|
| 2 | Hi!    |
| 4 |        |
| 0 | Hi!    |

| x | output |
|---|--------|
| 2 | Hi!    |
| 4 |        |
| 0 | CRASH  |

# Short Circuit Usage

- ♦ Short circuiting a boolean expression can prevent runtime errors!

```
Grid<Actor> gr = getGrid();  
if(gr != null && gr.isValid(loc)) {  
    System.out.println("loc is valid");  
}
```

When `gr` is null, the `&&` expression short circuits, preventing a runtime error (can't call a method on a null reference). Order is really important!

# Boolean Facts

- ♦ Demorgan's Law: !'s distribute, and flip everything!

$$!(a \ \&\& \ b) \ \rightarrow \ !a \ \|\ \ !b$$

$$!(x > 5 \ \|\ y == 4) \ \rightarrow \ x \leq 5 \ \&\& \ y \neq 4$$

$$!(x \geq 6) \ \rightarrow \ x < 6$$

$$!((a \ \&\& \ !b) \ \|\ (x > 3)) \ \rightarrow$$

$$!(a \ \&\& \ !b) \ \&\& \ !(x > 3) \ \rightarrow$$

$$(!a \ \|\ b) \ \&\& \ x \leq 3$$

# Object Comparison

- ◆ To compare int's, double's, and boolean's use: ==

```
int x = ...;  
if( x == 4) {  
}
```

- ◆ To compare Object's (Strings, etc), use either:
  - ◆ equals(Object) --> returns a boolean
  - ◆ compareTo(Object) --> returns an int
  - ◆ Check for null? Use obj == null

# Comparison Examples

```
String a = "hello";  
String b = "world";
```

```
if(a.equals(b)) {  
    //false  
}
```

```
int val = a.compareTo(b); //a negative value  
//similar to a - b (hello comes before world in the  
//dictionary, so a is smaller than b
```

# Free Response Hints

- ◆ When Writing a class...
  - ◆ Make sure you have `public class ClassName {`
  - ◆ All properties of a class are fields, and they should be private (`private int x;`)
  - ◆ All actions, commands, or questions that you would ask an object are methods (`public void doSomething(int param) {}`)
  - ◆ Creating objects of classes are through constructors (`public ClassName(int params) {}`)

# Class Diagram

```
public class ClassName {  
    private int field;  
  
    public ClassName(int constrParam) {  
    }  
  
    public void setterMethod(int x) {  
    }  
  
    public int getterMethod() {  
    }  
}
```

# Writing Constructors

- ♦ Set your class' instance fields to the parameter values
- ♦ Set your superclass' instance fields to the parameter values through a **super(...)** call.

```
public class A {  
    private int x;  
  
    public A(int theX) {  
        x = theX;  
    }  
}
```

```
public class B extends A {  
    private int y;  
  
    public B(int theX, int theY) {  
        super(theX); //must be first!  
        y = theY;  
    }  
}
```

# Writing Methods

- ♦ Examine what parameters your method has and decide what you could possible do with that parameter
  - ♦ You can do arithmetic operators with int's and double's
  - ♦ You can get or set values of an ArrayList or array
  - ♦ You could get parts of a String and compare it versus other String's
- ♦ If your parameter is of a type of another class, look over the methods in that class - those are methods that you can call off of your parameter!

```
int result = param.method();
```

# Working with Values

- ♦ Always pay close attention to what types of things you are given in an `ArrayList` or array. Make sure you determine what methods you can call off of objects of that type (and what parameters these methods need).

```
private ArrayList<Book> books;
```

I can get the price of each book  
if I want!

```
public class Book {  
    public double getPrice() {  
    }  
}
```

# Writing Methods

- ♦ Examine what class your method is in. Look over the methods in that class - those are methods that you can call off of yourself (this) or other objects of the same type!

```
int result = method();
```

- ♦ Sometimes you will be given methods that you will not use, but most of the time you are given a method for a reason! If you see a method, you should determine when and why you might want to use that method.

```
public class A {  
    public int method() { return 5; }  
  
    public void doSomething() {  
        int r = method(); //look mom, no dot!  
    }  
}
```

# Writing Methods

- ♦ Pay close attention if your class extends another class
  - ♦ You can call any of the public super class methods to help you
  - ♦ You can override a public super class method if you like
  - ♦ You **CANNOT** access the private variables in your superclass directly (use public getter methods)

```
public class A {  
    public int getYears() {  
        //code  
    }  
}
```

```
public class B extends A {  
    public int getMonths() {  
        return getYears() * 12;  
    }  
}
```

# Math Class

- ♦ The Math class has several useful functions! It is outlined in the quick reference guide. Some examples...
  - ♦ `int v = Math.abs(x); //absolute value`
  - ♦ `int min = Math.min(x, y); //smaller`
  - ♦ `int max = Math.max(x, y); //larger`
  - ♦ `int index = (int)(Math.random()*array.length); //random array index`
  - ♦ `int index2 = (int)(Math.random()*list.size()); //random list index`

# Recursion

- ♦ A way to do repetitions by having a method call itself.
- ♦ A recursive method must always have at least one bottom or base case.

```
public int recursiveSum(int[] array, int index) {  
    if(index < 0) //base case - when you stop  
        return 0;  
    else  
        return array[index] + recursiveSum(index-1);  
}
```

# Recursion

- ♦ A lot of recursion problems on the multiple choice can be solved quickly by working up from the base case.

```
public int foo(int x) {  
    if(x <= 0)  
        return 5;  
    return 6 + foo(x-1);  
}
```

What is foo(5)?

$$\text{foo}(0) = 5$$

$$\text{foo}(1) = 6 + \text{foo}(0) = 11$$

$$\text{foo}(2) = 6 + \text{foo}(1) = 17$$

$$\text{foo}(3) = 6 + \text{foo}(2) = 23$$

$$\text{foo}(4) = 6 + \text{foo}(3) = 29$$

$$\text{foo}(5) = 6 + \text{foo}(4) = 35$$

# Method Overloading

- ♦ You overload a method (in your class or from a superclass) by keeping the same name but having a different parameter list (different types or different number - variable names have no effect).
- ♦ Java determines which method you want to call by what data you pass into it.

```
public void foo() {  
}   
public void foo(int x) {  
}   
public void foo(double x) {  
}
```

# Overriding vs Overloading

- ◆ This method overrides the Object class' method

```
public boolean equals(Object o) {  
}
```

- ◆ This method overloads the Object class' method

```
public boolean equals(ClassName o) {  
}
```

# Selection Sort

- ♦ Works by searching for the maximum value of all the items in the array that have not already been placed in the right spot
- ♦ This maximum value is then swapped into the right spot in the list
- ♦ The number of comparisons is always the same  $(n * (n+1)/2)$ , so the running time is a very consistent, roughly  $n^2$
- ♦ The order of elements in the array has no effect on the running time!

# Selection Sort

```
public void selectionSort(int[] array)
{
    for(int stage = 0; stage < array.length-1; stage++)
    {
        int maxIndex = 0;
        for(int i = 1; i<array.length-stage; i++)
            if(array[i] > array[maxIndex])
                maxIndex = i;
        swap(array, maxIndex, array.length-stage-1);
    }
}
```

Demo

# Insertion Sort

- ♦ Insertion sort works by breaking the array into two parts: a sorted part on the left and an unsorted part on the right
- ♦ One by one, the next element from the unsorted part is inserted into the correct spot in the left through searching and shifting values
- ♦ The closer the array is to being sorted, the smaller the number of comparisons and shifts (only  $n$  required when completely sorted).
- ♦ The closer the array is to being out of order, the more comparisons and shifts are required (Same number as selection sort or roughly  $n^2$  when in reverse order)

# Insertion Sort

```
public void insertionSort(int[] array)
{
    for(int stage=0; stage < array.length - 1; stage++)
    {
        int i = stage + 1;
        int value = array[i];
        while(i > 0 && value < array[i-1])
        {
            array[i] = array[i-1];
            i--;
        }
        array[i] = value;
    }
}
```

DEMO

# MergeSort

- ♦ Works by breaking an array into two sub-arrays, recursively sorts both parts, then merges the two sorted sub-arrays together
- ♦ Recursion is successful because the base cases of having a sub-array of 1 element (do nothing) or 2 elements (swap if necessary) are easy to handle
- ♦ Requires an extra storage array to perform the merging procedure properly

# MergeSort (Cont)

- ♦ The merging procedure is accomplished by comparing the beginning values in each sub-array, picking the smaller value each time until all values are chosen.
- ♦ This is an easy task as both sub-arrays are already sorted - finding the next smallest number can always be found by looking at only the front spots of each sub-array
- ♦ The running time for mergesort is very fast because it is a successful divide and conquer strategy. Its running time is only slightly effected by the number of swaps required in the second base case.

# MergeSort

```
private static int[] storage;

public static void sort(int[] array) {
    storage = new int[array.length];
    sort(array, 0, array.length-1);
}

public static void sort(int[] array, int lo, int hi) {
    if(lo + 1 == hi) { //sub array with 2 items, might need to swap them!
        if(array[lo] > array[hi])
            swap(array, lo, hi);
    }
    else if (hi > lo) { //sub array with 3 or more items - divide and conquer!
        int firstLo = lo;
        int firstHi = (lo + hi)/2;
        int secondLo = firstHi + 1;
        int secondHi = hi;

        //split into two sub arrays, sort recursively, then merge together
        sort(array, firstLo, firstHi);
        sort(array, secondLo, secondHi);
        merge(array, firstLo, firstHi, secondLo, secondHi);
    }
}
```

# MergeSort (Cont)

```
public static void merge(int[] array, int lo1, int hi1, int lo2, int hi2) {
    int currentIndex = lo1;
    int i = lo1;
    int j = lo2;

    while(currentIndex <= hi2) {
        if(i > hi1) { //we only have a value from the 2nd array to consider
            storage[currentIndex] = array[j];
            j++;
        }
        else if(j > hi2) { //we only have a value from the 1st array to consider
            storage[currentIndex] = array[i];
            i++;
        }
        else { //we have to consider a value from both of the arrays
            if(array[i] < array[j]) {
                storage[currentIndex] = array[i];
                i++;
            }
            else {
                storage[currentIndex] = array[j];
                j++;
            }
        }
        currentIndex++;
    }

    //copy all the values from the storage array back
    for(int k = lo1; k <= hi1; k++)
        array[k] = storage[k];
}
```

DEMO