Midterm II Review

Memory Diagrams
Look over the slides from the last lab.

Conditionals
Will look at the if statement after we look at boolean values, relational expressions and logical expressions.

Boolean Values
A boolean value can be true or false.
boolean test1 = false;
boolean test2 = true;

Relational Expressions and Operators
A boolean value is the result of relational expressions including operators like ==, <=, >=, <, >, and !=.

Examples:
int a = 5;
int b = -13;
(a < 5)    // This is false!
(b >= -15) // This is true!

Logical Expressions and Operators

Relational Expressions can be combined with Logical Operators like not (!), and (&&) and or (||). They also result in boolean values.

and (&&) – Both values must be true for the result to be true

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp;&amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

or (||) – At least one of the two values must be true for the result to be true

| A    | B    | A || B |
|------|------|--------|
| False| False|   False|
| False| True |    True|
| True | False|    True|
| True | True |    True|
Not (!) – Reverses the value – true becomes false, false becomes true

<table>
<thead>
<tr>
<th>A</th>
<th>!A</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

Conditionals

Conditional statements test relational and logical expressions and based on the result (true or false), will execute one set of code or another.

```
int a = 4;
int b = -17;
if ((a < 5) && (b > -15)) {
    // Code to execute when true! Only executes when a is strictly less than 5 and b is strictly greater than -15.
}
```

Conditionals do ‘short-circuiting’ in Java. They only evaluate expressions until they know the final value.

```
int a = 5;
int b = -13;
if ((a < 5) && (b > -15)) {
    // Code to execute when true! Only executes when a is strictly less than 5 and b is strictly greater than -15.
} else {
    // Code to run if the logical expression is false
}
```
Since Relational and Logical expressions, do not complicate things with a conditional if you are trying to return a Boolean value based on the test. So if you are returning true or false from a method depending on whether an integer variable (for example, a) is even (true) or odd (false), you write:

```
return (a % 2 == 0);
```

**NOT**

```
if (a % 2 == 0) {
    return true;
} else {
    return false;
}
```

Or worse:

```
boolean returnValue;
if (a % 2 == 0) {
    returnValue = true;
} else {
    returnValue = false;
}
return returnValue;
```

The above two examples suggest a very poor understanding of Java semantics and efficiency.

**While Loops**

While loops will execute as long as a condition is true (or while it is false if we add negation (!)).

```
int a = 0;
while (a < 10) {
    System.out.println(a + "", a*a);
    a++; // Important: we have to change something that affects the conditional test! Otherwise we have a non-terminating (infinite) loop
}
```

Here is an example where negation is used to do a loop until a condition is false. We have a method called isEven() that will return true if some integer is even. We do not have an isOdd test. So we can test for odd numbers by saying !isEven().

```
Random r = new Random();
b = r.nextInt(100);
while (!isEven(b)) {
    // Do stuff
    b = r.nextInt(100);
}```
Arrays
Arrays hold a collection of multiple items of the same type. Elements of an array are indexed with an integer from 0 to 1 less than the size of the array.

So for an array with, for example, 10 elements, indices will run from 0 through 9.

Declarations can look like:

```java
int myArray[] = new int[10]; // Reserve space for 10 integers and initialize myArray with
// the memory reference for the reserved space.
```

You can also create and initialize the elements of an array at the same time

```java
int myArray[] = {10, 20, 30, 40, 50, 15, 25, 35, 45, 55};
```

Notice in each case, there is no element count in the brackets by the variable name! (myArray). The count of elements is determined in the ‘new’ clause or by Java counting the number of values given between brackets {}.

Arrays can have an almost unlimited amount of ‘dimensions (only limited by memory size). Multiple dimensions each need a separate index. Indices are kept in separate sets of brackets []. Here is a 2 dimensional array

```java
// This array has 10*20 = 200 integer elements which means it needs 200*4 or 800 bytes of space.
int myBigArray[][] = new int[10][20];
```

The next section shows a convenient way to traverse arrays (single and multi-dimensional).

For Loops
For loops repeat code a fixed number of times. Useful when you know exactly how many times you want code executed.

The for clause has 3 sections:

```java
for (<init>; <test>; <end-of-loop-code>) {...}
```

- `<init>` - Usually used to initialize a control variable (variable used in the `<test>`)  
- `<test>` - Condition test to determine if the loop should execute again.  
- `<end-of-loop-code>` - Some code to run at the end of the loop before the next `<test>`

Any of the 3 sections (or even all of them) can be left blank.

However, note that this:

```java
for (;;) {...}
```

Results in an infinite (non-terminating) loop.
Note: A for loop can be used to implement the equivalent of any while loop and vice-versa.

Remember this loop?

```java
int a = 0;
while (a < 10) {
    System.out.println(a + "", " + a*a);
    a++;
    // Important: we have to change something that affects the conditional test! Otherwise
    // we have a non-terminating (infinite) loop
}
```

Here is the same with for:

```java
int a;
for (a = 0; a < 10; a++) {
    System.out.println(a + "", " + a*a);
}
```

The above loop assumes you may need the variable ‘a’ after the loop is done. If you do not need it, it can be declared directly inside the for clause:

```java
for (int a = 0; a < 10; a++) {
    System.out.println(a + "", " + a*a);
}
```

Here is the other loop we saw above:

```java
Random r = new Random();
b = r.nextInt(100);
while (!isEven(b)) {
    // Do stuff
    b = r.nextInt(100);
}
```

Which could be written:

```java
Random r = new Random();
for (b = r.nextInt(100); !isEven(b); b = r.nextInt(100)) {
    // Do stuff
}
```

Break, Return, Continue

Loops can be terminated early, if desired.

- You may use `return` directly inside of a loop.
- You can use `break`, to exit the loop early and continue with code after the loop.
- You can use `continue`, to end the current iteration of the loop and test to see if the loop should run again.

Obviously, these should be used inside an ‘if’ that tests for a condition that suggests an early exit!
Random r = new Random();
b = r.nextInt(100);
while (!isEven(b)) {
    if (b % 7 == 0) {
        break; // leave early if the random number is divisible by 7
    }
    // Do stuff
    b = r.nextInt(100);
}

Passing Arrays as Arguments, Returning Arrays from Methods

Arrays (single or multi dimensional) can be passed as arguments to methods. In this case, the reference is passed and any code in the method that assigns values to elements of the array do affect the caller’s copy of the array! This is a big difference from passing simple values like ints and doubles.

public static void swapElements(int[] array, int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

int myArray[] = {10, 20, 30, 40, 50, 15, 25, 35, 45, 55};
swapElements(myArray, 2, 5);

You can return an array from a method.

public static int[] randArray(int range, int count) {
    int[] ret = new int[count];
    random r = new Random();
    for (int i = 0; i < count; i++) {
        ret[i] = r.nextInt(range);
    }
    return ret;
}
Console Input

Read from the console with a ‘Scanner’ object:

```java
import java.util.Scanner; // Note capital ‘S’!

Scanner console = new Scanner(System.in);
```

Scanner can be used for other types of input. For now, we can set it up to read from the keyboard by giving an argument of ‘System.in’.

```java
Scanner console = new Scanner(System.in);
```

The Scanner contains many methods for reading input and testing for input types.

```java
int data = console.nextInt(); // Read an integer from console
double floatValue = console.nextDouble(); // Read a double from console
String name = console.next(); // Read a word (ending with ‘white space’: space, tab, or newline)
String wholeline = console.nextLine(); // Read a complete line up to a newline character
```

Passing a Scanner

Scanners can be passed to methods. They can be initialized in one method and passed to another to be used for input. You do not have to create a new Scanner in every routine.

```java
public static int fetchData(Scanner console) {
    int returnValue = console.nextInt();
    return returnValue * 2;
}
```

```java
public static void main (String[] args) {
    Scanner input = new Scanner(System.in);

    int data = fetchData(input);
    ...
}
```