Midterm Review

Topics

- Java Basics and Structure
- Variables
- Branching
- Loops
- Methods
- Arrays

Midterm Review
Kage Weiss
Lab 001 TA, SPR 17

Java Basics and Structure

```
Class{
    class variables;
    methods{
        method variables;
        statements;
    }
}
```

Java Basics and Structure

Code blocks (classes, methods) are defined by braces { }

Classes: <publicity> class <name> {/* code here*/}

Methods: <publicity> <return type> <name>
(<parameter1 type> <parameter1 name>,<param2>…)
{/* code here*/}

Variables only exist in the code block they're defined in:

block{ var x; }

X exists X does not exist anymore

Variables

Variables are used to store information of declared data types.

<data type> <variable name>; (e.g. int x;)

<data type> <variable name> = <value>; (e.g. int x = 5;)

Variables are assigned data via the assignment operator =

e.g. int x = 5;

Converting between the types can be done with type casting

float y = (float) x; (y now equals 5.0)

Variables must be declared before usage.

Variables are either primitive data types or object data types

Variables

Primitive data types:

- boolean - true or false - 1 bit
- char - Unicode Characters (2 bytes)
- byte - One byte (8 bits)
- short - Short integer (2 bytes)
- int - Regular integer (4 bytes) (−2 31 ⟹ (231 − 1))
- long - Large integer (8 bytes)
- float - Floating point (4 bytes)
- double - Floating point (8 bytes)

Relational Operators

A == B // Is A equal to B?
A != B // Is A not equal to B?
A < B // Is A less than B?
A > B // Is A greater than B?
A <= B // Is A less than or equal to B?
A >= B // Is A greater than or equal to B?

All of these operations return Booleans, true or false depending on the values of A and B

Assignment Operators

```
x = y; //x now has the value from y
x -= y; // same as: x = x - y;
x += y; // same as: x = x + y;
x *= y; // same as: x = x * y;
x /= y; // same as: x = x / y;
x %= y; // same as: x = x % y; (for integers x, y)
q &&= p; // same as: q = q && p; (for booleans q, p)
```

Increment and decrement:

```
x++; // same as: x = x + 1;
x--; // same as: x = x - 1;
```

Control Structures

Branching

- Branching allows for code with choices and options, and is often used hand-in-hand with variables to clean and optimize code.
  - Switch/case/default
  - if/then, if/else if/else

Loops

- Loops allow you to repeat a set of statements until certain conditions are met.

Branching

The main options are if statements and switch statements:

```
switch(test) {
        case <option>:
                //code here
        case <option2>:
                //code here
                break;
        default:
                //code here
}
```

```
if(test) {
        //code here
} else if(test2) {
        //code here
} else {
        //code here
}
```

If Statements

If statements are a simple code block inside a test, if the test fails it won't enter the block. If you place an else (or an else if) it will be evaluated when and only when the test fails.

```
if(x > 0) {
        x /= 2;
} else if(x < 0) {
        x *= -1;
} else {
        x = 9;
}
```

Remember the order of tests matters, if its passes the first it will not check any more.

Switch Statements

Switch statements are similar to sets of if/else if/else if/else if/… tests. Based on the result of one test it will chose an option (case)

```
//String x;
switch(x) {
        case “Op1”:
                //code here
        case “Op2”:
                //code here
                break;
        default:
                //code here
}
```

If you don’t break after a case, the program will “fall through” into the next code block inside the switch until it hits a break or a return statement.

Branching

Remember all tests must be binary (return a Boolean), and don't forget your Boolean operators.

A && B // AND : Are both A and B true ?

A || B // OR: Are either A or B true?

!A // NOT : Negate value of A (opposite)

This can be used to string tests together inside one big test:

```
//int x, y, z;
if( x < 0 || y > 0 || ((( z == x ) && ( z == y)) ) { /*Code here*/ }
```

Just remember: if it passes an OR or fails an AND before it completes all the tests, it wont bother testing them all.

Loops

The main options are for loops and while loops:

```
for(initial value; test; increment) {
        //code here
}
```

```
while(test) {
        //code here
}
```

Loops will continue until they fail their test, or until the program either “break”s or “return”s. A break will kick the program out of a loop, a return will leave the current method with the value included.

For Loops

For loops generally operate on an index: declaring it, testing it, completing the code block, and then changing the index

```
for(int i = 0; i < 9; i++) {
        System.out.println(i);
}
```

For loops give you an index (i) to use inside the loop.

Enhanced for loops give us a variable value instead of an index, if you know how to use them.

While Loops

While loops just keep testing. (So make sure it'll fail the test at some point, or break/return.)

```
//int x, y;
while(x != y) {
        x += ((y - x)/2);
        if(x > y) {
                break;
        }
}
```

While loops also have a special version. Want to run your block before you test? Use a do { } while(test);

Loops

Remember that loops will continue until their tests fail, so be sure the test has to fail at some point, or if you want it to loop infinitely, then be sure it'll break or return. Some infinite loops are shown here.

```
for(int i = 100; true; i--) {
        //code here, with an index
}
```

```
while(true) {
        //code here
}
```

```
for( ; ; ) {
        //code here
        //a for loop without an index?!?
}
```

Methods

Methods are small (normally) blocks of code that are useful ore repeated so instead of typing out the full code or rewriting with a bunch of different variables, you use method calls. Methods are defined using the form:

<publicity> <return type> <name>(<param1 type> <param1 name>,
<param2>…) {
/* code here*/
}

You call a method with the form:

<name>(<params>);

Methods

If an object has its own methods, you can call the methods for that object by saying <object>.<method>(<params>);
For example, if we have a String x, and we want to see if it contains the letter “y”, we would say:
boolean containsY = x.contains(“y”);
Where “x” is our object (String), “contains” is our method, and “y” is the String parameter we are passing in. Contains returns a Boolean, so we store the value of “x.contains(“y”)” in our variable “containsY”.
If we write our own method:
public void methodName() { /*code here*/ }
We call it by saying:
methodName();

Methods

If there are two versions of a method that have different parameters, we call this overloading. Both exist, and java will determine which to call based on the parameters you give it in the method call.

A method must return the type it declares as its return type.

Methods for Strings

- String y = x.replace(“<String1>”, “<String2>”);
  - Replaces all instances of String1 in x with String2
- x.contains(“<String>”)
  - Returns true if “<String>” exists in
- x.indexOf (“<String>”)
  - Returns the index of the first appearance of “<String>”
- x.charAt (int i)
  - Returns the character at the index “i”
- x.toUpperCase ()
  - Returns the String x in all caps
- x.substring(int beginIndex, int endIndex)
  - Returns the String from beginIndex to endIndex in x

Arrays

An array is a variable that is an indexed collection of data.
To declare an array, we use the format:
<type>[] <name>;
When we initialize it we must give it a size, the size for an array cannot change. W initialize an array as such:
<type>[] <name> = new <type>[<size>];

For example if I wanted an array, book, of 30 Strings, I would say:
String[] book = new String[30];
We can also initialize an array with set values:
String[] book = { “Arrays”, “are”, “fun”, “and”, “quite”, “useful”, “.”};

Arrays

To reference a value in an array, we say <name>[<index>]
These values work just like any other variable:
book[0] = “Words”;

Arrays don’t come pre-filled with nice values, arrays of numbers contain all 0’s, arrays of reference types contains all null.

Remember, arrays indices start at 0, so an array of 30 Strings would have a String at book[0] and at book[29], but not at book[30]. You can always find the length of an array (number of values it hold) by saying <name>.length

Arrays

You can also overwrite arrays, because as variables they are references to the information, not the information itself:

String [] arr1 = { " Hello", " World" };

String [] arr2 = { " Goodbye", " World" };

arr1 = arr2 ;

arr1 and arr2 both now point to the data { " Goodbye", " World" }

{ "Hello", "World" } gets thrown away because arr1 no longer references it.

Combinations

Combinations of variables, arrays, methods, loops, and classes allow us to accomplish anything possible in java, though if you limit yourself to just these then some things might be very difficult. When writing any program start simple with what you're comfortable doing, and build from there both on your own knowledge and on your program.