July 1, 2010

Introduction

- I am James. (You are free to forget this information.)
- Syllabus
- Homework format

Program Design and Data Structures

- Program Design: a systematic way of organizing the flow of a program's execution.
- Data Structure: a systematic way of organizing and accessing a record of data

Java Variables

- variable: a location in a computer memory which can be changed during the course of a program's execution
- field: a variable that is associated with an object
- static field: a variable that is associated with a class
- instance variable: a variable that appears in a block of code

Variables can be primitive (meaning that they are the eight basic types available to all programs) or they can be Objects. Objects are high level blocks of memory with methods and field associations.

There are 9 primitive types, and each begin with a lowercase variable. In Java code, the convention is that primitive variables begin with a lowercase letter and Objects begin with an uppercase letter.

- void: Means “no type”. This can only be applied to methods.
- boolean: Use for True/False expressions. Size: 1 bit
- byte: Integers from -128 to 127. Size: 8 bits
- char: ASCII character data. Size: 16 bits
- short: Integers -32,768 to 32,767. Size: 16 bits
- int: Integers from -2,147,483,648 to 2,147,483,647. Size: 32 bits
- float: Decimal Numbers, low precision. Size: 32 bits
- double: Decimal Numbers, high precision. Size: 64 bits
- long: Integers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Size: 64 bits

Typically, you pick the primitive variable you need based on the precision that you need and the memory limitations of your project. In this course, we will be using “void” for typeless methods, “boolean” for boolean flags, “char” for character data, “int” for integer data, and “double” for decimal data. “byte”, “short”, and “long” will not be used.

All instance variables in Java have a “scope” in which they exist and do not exist. Think of scope like a lifespan. Variables are born, they live, and they die. Variables only exist within the scope in which they were defined. Scope is defined by blocks. Blocks are defined by ”{ }” symbols in your program.

```
import java.io.*;


class Example {
	public static void main(String[] args) {
		int count;
		{
			int students = 19;
		}
		    count = students; // Error! Cannot find symbol 'variable students'! Can you spot the problem?
	}
}
```

Java Input

As of Java 1.5, the Scanner class has made taking input easier on the developer. To import that Scanner class into your program, you need the following line of code at the top:

```
import java.util.*;
```

To create a new scanner, you need the following line of code:

```
Scanner scan = new Scanner(...Some Source...)
```

The Scanner class requires that you initialize the scanner with an input source. There are several types of input, but we will concentrate on these three:

Input From String

To treat a string as an input source, simple create a new scanner an use the variable name of the string as the source.

```
String myInput = "Our baseball team as 9 players.";

Scanner scan = new Scanner(myInput);
```

Input From File

To read from a file, create a new scanner and make the source “new File(filename)”:

Note: The “File” object requires that you also import the “java.io.*” library.

```
Scanner scan = new Scanner(new File("records.txt"));
```

Input From Keyboard

To read from the Keyboard, create a new scanner like this:

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

Now that we've initialized the scanner, we can read some input. Reading input is the same no matter what source type you use. Reading a file is no different than reading from the keyboard. Keep this in mind if you are ever asked to change your program from keyboard input to file input.

The following are the most common scanner methods:

- scan.next() - Reads one token up to the first whitespace character and returns a string.
- scan.nextLine() - Reads one full line of input up to the next newline character and returns a string.
- scan.nextInt() - Reads one integer, returns an integer.
- scan.nextDouble() - Reads one decimal number, returns a double.

Each of these methods has a boolean counterpart which returns TRUE or FALSE if a token actual exist.

- scan.hasNext() - Checks if anything is still in the input stream.
- scan.hasNextLine() - Checks if anything is still in the input stream.
- scan.hasNextInt() - Checks if an integers is next the input stream.
- scan.hasNextDouble() - Checks if a decimal is next in the input stream.

Your first homework assignment will be to implement a scanner, ask the user for a phrase, perform a little bit of string processing and display a new phrase on the screen.

```
import java.util.*;


class Name {

    public static void main(String[] args) {

        Scanner scan = new Scanner(System.in);

        System.out.print("What is your name? ");

        String name = scan.nextLine();

        System.out.println("Hello "+name);

    }

}
```

If/Else Statements

Programs work sequentially. The next statement in a program usually follows the current statement. There are ways to control the flow of a program to avoid some lines being executed.

“If”-statements allow us to branch a program in one direction if some condition is met and to a second direction if a different condition is met.

“If”-statements can execute a block of statements (defined by ”{ }”), or they can execute a single statement, which means no curly braces are needed. If you are ever unclear about whether you need to use ”{ }” curly braces or not, always opt for using ”{ }” curly braces.

```
System.out.println("How many people? ");

if (players < 9)

        System.out.println("Not enough for baseball or soccer.");

else if (players < 11)

        System.out.println("Enough for baseball, but not soccer.");

else

        System.out.println("Enough for baseball and soccer.");
```

Loops

Loops repeat a statement or block (notice how this is similar to “If”-statements) until a condition is met.

There are the two basic loop types:

- Precondition loops test for a condition at the *beginning* of every loop, which means a loop can execute zero times. There are two precondition loops in Java: “for” and “while”.
- Postcondition loops test for a condition at the *end* of every loop, which means a loop will always execute one time at least. There is one postcondition loop in Java: “do… while”.

“for” and “while” loops have different syntax, but essentially do the same thing.

```
// Example of "for" loop counting to 10


for (int i = 1; i <= 10; i++) {

        System.out.println("#"+i);

}


// Example of "while" loop counting to 10


int i = 1;

while (i <= 10) {

        System.out.println("#"+i);

        i++;

}


// Example of "do...while" loop counting to 10


int i = 1;

do {

        System.out.println("#"+i);

        i++;
} while (i <= 10);
```

Loops have two control statements:

- break: Immediately end the loop and jump to the first statement after the end of the loop
- continue: Start the loop over and …
    - do…while: do not retest the condition
    - while: retest the condition
    - for: perform the increment step and retest the condition

Java Arrays

An array is a reference to a continuous block of memory. This block of memory represents 1 or more variables of the same data type. The three common task associated with any array are 1) reading from an array, 2) writing to an array, and 3) finding the length of an array. One of the major downsides to an array is that once you define the length of an array, that length can never change.

An array is like a spreadsheet on your computer. A spreadsheet is a listing of cells that can be modified. An array is a listing of cells that can be modified. The first cell in an array starts at location 0, the second cell is at location 1, the third cell is at location 2, all the way to the last cell at location (length-1).

There are two ways to declare an array.

Declared with data:

```
int[] jenny = {8, 6, 7, 5, 3, 0, 9};
```

Declared with size:

```
int[] number = new int[7];
```

We can print each value of an array individually using “variable\[number\]” syntax.

```
System.out.println(jenny[0]); // Prints "8"

System.out.println(jenny[1]); // Prints "6"
```

We can print the length of the array and use it in calculations:

```
System.out.println(jenny.length);
```

We can assign values to each member of the array via the keyboard:

```
for (int i = 0; i < jenny.length; i++)

    jenny[i] = console.nextInt();
```

We can print all of the values in an array using a simple “for” loop which starts at 0:

```
for (int i = 0; i < jenny.length; i++)

    System.out.println(jenny[i]);
```

New values can be placed in array positions using the following syntax:

```
jenny[0] = 4; // Now the array reads {4, 6, 7, 5, 3, 0, 9}
```

We can reference the first item in the list at position 0. We can reference the last item in the list at position

```
jenny.length-1
```

.

Sometimes we need to find the largest value in a list. In the case of array “jenny”, the largest value is 9 at position 6, but how do we know that with an array that we've never seen before? We need an algorithm. (This is on page 540 in the textbook.)

```
int maxIndex = 0;

for (int index = 1; index < jenny.length; index++) {

    if (jenny[maxIndex] < jenny[index]) {

        maxIndex = index;
    }
}


int largestValue = jenny[maxIndex];
```

Bounds Checking

Every array has two ends (or bounds as they are more often called). The lower bound is always at position 0. The high bound is always at position (length-1). If an array position is ever called that is less than 0 or greater than (length-1), the ArrayIndexOutOfBoundsException trigger will stop your program. This is known as “bounds checking”.

In the programming language C, there is no “bounds checking”. If an array existed in memory that was beside other variables in memory, using a simple call to “array[-1]” allowed a program to access memory using an array that had nothing to do with that array. It is possible to change and manipulate portions of a program using array calls to values that do not exist in an array, which a major security flaw in the C language. Computer crackers have learned to exploit these flaws and trick C programs into executing their own code using “buffer overruns”. Buffer overruns were a major concern to the developers who made Java, and the ArrayIndexOutOfBoundsException prevent crackers from taking advantage of flaws in software.

When the ArrayIndexOutOfBoundsException triggers, it aborts the program and displays what's called the “Stack Trace” onto the screen.

```
import java.util.*;


class stack_trace_error {
    public static void main(String[] args) {
        int[] array = {10,20,30,40,50};


        System.out.println(array[0]); // 10
        System.out.println(array[1]); // 20
        System.out.println(array[2]); // 30
        System.out.println(array[3]); // 40
        System.out.println(array[4]); // 50
        System.out.println(array[5]); // ArrayOutOfBoundsException
    }
}
```

Later on in the semester, we will look at “try/catch” blocks, which are tools built into Java that will allow you to

Java Methods

A method is a set of procedures that can be executed as needed. The purpose of a method is to create reusable code for common task.

There are many task in mathematics that have very common, generic purposes. Finding the average of a list of numbers is always the same, no matter what the subject matter of the numbers may be. The algorithm for finding the average of a list of numbers is ideal to place in its own method. By defining a block of code in its own method, we can call the method as needed in a single line rather than rewrite the algorithm. If we need to update that method, we only need to change the code in one place as long as the inputs and outputs do not change.

There are three parts to a method:

- return type: The type of value returned by the method
- signature: The name and parameters to the method
- body: The actual code which is executed when the method is called

It should be noted that the signature of a method is made up of the method name and the method parameters. Because of this, when a method is called, Java looks for a name of matching name and matching parameters. It is possible for two methods to coexist in the same class with the same name as long as they have different input parameters.

To compute the average of a list of numbers, here are the steps you need to take:

1. Add all of the values in the list
2. Count all of the values in the list
3. The average is computed by dividing the sum of the values by the number of values.

Here is a method to compute the average of a list of numbers.

```
double average(int[] numbers) {
    double sum = 0;
    for (int i = 0; i < numbers.length; i++)
        sum = sum + numbers[i];

    return sum / (double) numbers.length;
}
```

In this example, “average” has…

- A return type of “double”, meaning that this method will compute some decimal number.
- One parameter “int[] numbers”, which is the list of numbers for which we need to compute the average.
- A block of code which finishes in a finite amount of time

Pass-By-Reference VS Pass-By-Value

Java's approach to variable passing is different to that of traditional languages. When a primitive is passed to a method, Java passes the variable by value. When an array or object is passed, Java passes the variable by reference.

- Pass-by-value means that the variables that are given to a method are copied directly to new values found in the method. The instance variables in a Java Method all have their own scope. They all are destroyed by the end of the method. If they change, those changes are lost.
- Pass-by-reference means that the memory locations of each variable are passed to a method. This means that if a variable changes, it changes outside that method as well.

```
int a = 4;
int b = 3;

swap(a, b);

// Java and C++, pass-by-value
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

// C++, pass-by-reference
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

The first of these two methods does nothing. Once the values are swapped, those changes are lost at the end of the method. The second method (written in C++), the values are passed by reference, meaning that the values really do change when this method. This isn't possible in Java.