Week 1 - Wednesday

COMP 2000

Last time

- What did we talk about last time?
- Syllabus
- Started refresh on Java
  - Primitive types
  - Basic operations
  - Shortcuts

Questions?

Java Basics

Reference types

- There are an unlimited number of reference types, including:
  - **`Object`**
  - **`String`**
  - **`Scanner`**
  - All arrays
  - Any type that begins with an uppercase letter
  - Any type that isn't one of the 8 primitive types
- Reference types do not use operators (except for **=** and **==** and **+** [for **`String`** concatenation])
- Instead, we interact with reference types with methods

Strings

- The **`String`** type is immutable in Java
  - You can never change a **`String`**, but you can create a new **`String`**
  - The second line creates a new **`String`**:

```
String stuff = "Break it down ";
stuff += "until the break of dawn";
```

- This approach can be very inefficient:

```
String values = "";
for(int i = 0; i < 1000000; i++)
  values += i;
```

- When a lot of concatenation is expected, use **`StringBuilder`**

Case sensitivity

- **Java** is a case-sensitive language
- **Class** is not the same as **class**
- `System.out.println("Word!");` prints correctly
- `system.Out.Println("Word!");` does not compile

Whitespace

- Java generally ignores whitespace (tabs, newlines, and spaces)

```
System.out.println("Hello, world!");
```

- is the same as:

```
        System.out.

println(                    "Hello, world!");
```

- You should use whitespace effectively to make your code readable

Comments

- There are three kinds of comments
- Single line comments use **//**

```
System.out.println("Hi!"); // this is a comment
```

- Multi-line comments start with a **/*** and end with a ***/**

```
System.out.println("Hi!"); /* this is
                              a multi-line
                              comment */
```

Documentation comments

- The third kind of comment is a **documentation comment**
- These comments look like multi-line comments but have an extra asterisk at the beginning: **/****
- Documentation comments have special syntax inside of them that allows the programmer to make notes about the program that the compiler can recognize and used to generate documentation

```
/**
 * This method peels carrots.
 * @param carrot the carrot to peel
 * @return the peeled carrot
 * @author Barry Wittman
 */
public Carrot peelCarrot(Carrot carrot) {
  // Do stuff
}
```

Control Structures

Control structures

- Java control structures come in two categories
- Selection (making a choice):
  - **`if`** statements
  - **`switch`** statements
- Repetition (loops):
  - **`while`** loops
  - **`for`** loops
  - Enhanced **`for`** loops
  - **`do-while`** loops

if statement

```
if( condition ) {
    statement1a;
    statement2a;
    …
}
else {
    statement1b;
    statement2b;
    …
}
```

- The **condition** is any statement that evaluates to a **boolean**
- If the **condition** is **true**, the code inside the body will execute
- If it's **false**, the code inside the **else** body will execute
- Braces for both the **if** and the **else** are optional if there is only a single statement
- The **else** part is optional as well

switch statement

```
switch( data ) {
case value1:
     statement1;
     break;
case value2:
     statement2;
     …
case valuen:
     statementn;
default:
     statementdefault;
}
```

- The data that you are performing your switch on must be either an **int**, a **char**, a **String**, or an **enum**
- The value for each case must be a literal
- Execution will jump to the **case** that matches
- If no **case** matches, it will go to **default**
- If there is no **default**, it will skip the whole **switch** block
- Execution will continue until it hits a **break**

Loops

- Allow us to repeatedly execute code
- Care must be taken to run exactly the right number of times
  - Not too many
  - Not too few
  - Not an infinite number
  - Not zero (unless that's what should happen)
- Loops come in three flavors:
  - `while` loops
  - `for` loops
  - `do-while` loops

while loops

- Used when you don't know how many times a loop will run
- Runs as long as the condition is `true`
- Syntax:

```
while( condition ) {
  // Statements
  // Braces not needed for single statement
}
```

for loops

- Used when you do know how many times a loop will run
- Still runs as long as the condition is `true`
- Syntax:

```
for(initialize; condition; increment) {
  // Statements
  // Braces not needed for single statement
}
```

Enhanced `for` loops

- Used to iterate over the contents of an array (or other collection of data)
- Similar to **`for`** loops in Python
- The **`type`** must match the elements of the array (or other collection)
- Syntax:

```
for(type value : array) {
  // Statements
  // Braces not needed for single statement
}
```

Enhanced for loop example

- Method to find largest value in an array

```
public double findLargest(double[] numbers) {
  double largest = numbers[0];
  for(double number : numbers) {
     if(number > largest)
          largest = number;
  }
  return largest;
}
```

Enhanced `for` loop rules

- It's common that you want to iterate over an entire list
- Enhanced **`for`** loops are great for that but not as flexible as other loops
- You have to loop over everything (unless you use **`break`**), and you can't look at the previous or next elements
- You can **never** change the values in the list with an enhanced **`for`** loop

```
int[] array = new int[100];
for(int value : array)
  value = 25; // Does nothing!
```

- It can only read the values

do-while loops

- Used infrequently, mostly for input
- Useful when you need to guarantee that the loop will run at least once
- Runs as long as the condition is `true`
- Syntax:

```
do {
  // Statements
  // Braces not needed for single statement
} while( condition );
```

Loop examples

- Write a **`for`** loop to reverse the contents of an array
- Write a **`while`** loop to reverse the contents of an array
- Now turn it into a **`do-while`** loop, just for the hell of it

break and continue

- The keyword **break** is necessary syntax to stop executing a switch statement
- However, it can also be used to leave any of the four kinds of loops

```
int i = 7;
while(true) {
  if(i == 14)
     break;
  System.out.println(i);
  ++i;
}
```

- In any loop, instead of using **break**, you could use **continue**, which jumps to the end of the loop instead of exiting it
- Most style guides discourage the use of **break** and **continue**
- If you use them in my classes, you will lose style points

Arrays

Definition of an array

- An array is a **homogeneous**, **static** data structure
- **Homogeneous** means that everything in the array is the same type: **`int`**, **`double`**, **`String`**, etc.
- **Static** (in this case) means that the size of the array is fixed when you create it
- Unlike Python lists, you cannot push, pop, or resize an array

Declaration of an array

- To declare an array of a specified **`type`** with a given **`name`**:

```
type[] name;
```

- Example with a list of type **`int`**:

```
int[] list;
```

- Just like any variable declaration, but with **`[]`**

Instantiation of an array

- When you declare an array, you are only creating a variable that can hold an array
- To use it, you have to create an array, supplying a specific size:

```
double[] list;
list = new double[100];
```

- This code creates an array of 100 **double** values

Accessing elements of an array

- You can access an element of an array by **indexing** into it, using square brackets and a number

```
list[9] = 138.7;
System.out.println(list[9]);
```

- Once you have indexed into an array, that variable behaves exactly like any other variable of that type
- You can read values from it and store values into it
- **Indexing starts at 0 and stops at 1 less than the length**

Length of an array

- When you instantiate an array, you specify the length
- You can use its **`length`** member to find out

```
double[] list = new double[42];
int size = list.length;
System.out.println("List has " + size +
" elements"); //prints 42
```

- The indexes of an array and its length are always **`int`** values, no matter what the elements inside the array are

Two dimensional array

- To declare a two dimensional array, we just use two sets of square brackets (`[][]`):

```
int [][] table;
```

- Doing so creates a variable that can hold a 2D array of **`int`**s
- As before, we still need to instantiate the array to have a specific size:

```
table = new int[5][10];
```

Static Methods

Static Methods

- Static methods allow you to break your program into individual pieces that can be called by each other repeatedly
- Advantages:
  - More modular programming
    - Break a program into separate tasks
    - Each task could be assigned to a different programmer
  - Code reusability
    - Use code over and over
    - Even from other programs (like **`Math.sqrt()`**)
    - Less code (and error) duplication
  - Improved readability
    - Each method can do a few, clear tasks
    - Well named method are self-documenting

Return type and parameters

- A method takes in 0 or more parameters and returns 0 or 1 values
- A method that doesn't return a value is declared as a **`void`** method
- Definition syntax:

```
public static type name( type arg1, type arg2, … ) {
      //statements
      //braces are always required!
}
```

Calling syntax

- Proper syntax for calling a static method gives first the name of the class that the method is in, a dot, the name of the method, then the arguments

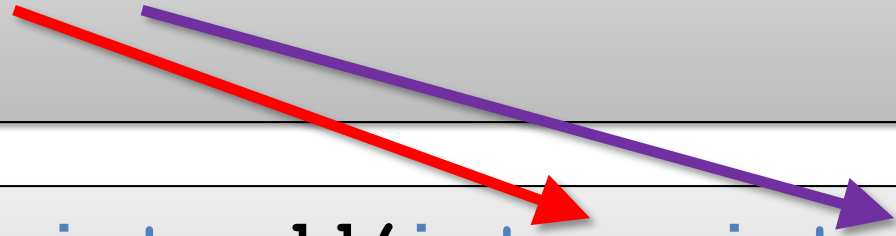
```
Class.name(arg1, arg2, arg3);
```

- If the method is in the same class as the code calling it, you can leave off the `Class.` part
- If it is a value returning method, you can store that value into a variable of the right type

Binding example

```
int a = 10;
int x = 3;
int y = add( 5, a );   //y contains 15 now
```

```
public static int add(int x, int y){
     int z = x + y; //5 + 10
     return z;
}
```



- No connection between the two different **x**'s and **y**'s

Binding

- When a method is called, the arguments passed into the method are copied into the parameters
- The names for the values inside the method can be different from the names outside of the method
- Methods **cannot** change the values of the arguments on the outside for primitive types
- Methods **can** change the values inside of arrays and sometimes inside of object types
  - But they can't change which array or object the reference is pointing to

Method practice

- Write a method with the following signature that converts a **`String`** representation of an integer into an **`int`** value
- **`public static int parseInt(String value)`**

Upcoming

Next time…

- First graded lab tomorrow
- On Friday, we'll talk about:
  - Classes
  - Objects
  - Enums
  - Packages

Reminders

- Review Chapters 3 – 8 (except for 7)
- Office hours end at 4 p.m. instead of 5 p.m. today for Faculty Assembly