CMPE212 – Reminders

• Assignment 1 due Friday.
• Quiz 1 Next Week, in the lab on Thursday. Everything up to and including next Monday's lecture material. Topics will be listed later this week. Make sure to complete “Quiz 0” before writing Quiz 1.

Today

• Finish Style Guidelines for assignment 1. Last time: Using whitespace and commenting.
• Modular Coding at the Method Level.
• Some useful classes from java.lang that we should cover before getting into OOP.

More Style Hints

• Leaving commented out code in your program is really irritating to someone trying to read your program – it makes it look unfinished.

• If your program is not finished – go ahead and admit this in a comment above the method or class. Note what you think the problem is. Your TA is going to find out anyways if the code is broken – so you might as well just come clean and save him some effort!

Style Demo Programs

• See:
  – NoStyle.java
  – PoorStyle.java
  – GoodStyle.java

• They all are syntactically correct, will compile without error and all do the same thing.

Building Modular Code

• You should already know the advantages to modularity:
  – Easier to build.
  – Easier to test.
  – Easier to debug.
  – Easier to modify.
  – Easier to share.

• Building objects just takes modular design to the next level.
• But for now, let’s focus on functions (or methods).

Designing Functions

• Functions are written to avoid repeating code. So, make sure you only have one function to do the “one thing”.

• Functions should be short:
  – How short is “short”? One to ten lines?
  – If you can satisfy all the other rules and the code still explains itself, then the function is short enough.
Designing Functions, Cont.

- **Functions should only do one thing and do it well.**
  - Yah, but how can we define “one thing”?
  - Write the most abstract description of what the function does – does it still have the word “and” in it?
  - Look for loosely coupled sections within a function if it seems too large – these are natural dividing points. Sometimes the coder puts an empty line between sections as an unconscious admission of loose coupling.

- **Keep all code within the function at the same level of abstraction.**
- If you find yourself writing detailed code right beside high level code, you are not following this rule.
- Your program should be readable as a top-down narrative. The most abstract functions will be at the top of the program, leading to the least abstract functions further down. (BTW, I’m bad and I don’t always follow this rule.)

- **Use less than three parameters wherever possible – the best number is zero!**
- Try to use parameters for input only.
- Flag parameters can be ugly – they are saying that the function does at least two things.
- If needed, multiple parameters can be grouped into an object or list, provided they share a theme.
- For example:
  - `drawCircle(Point centrePoint, int radius)` is better than
  - `drawCircle(int centreX, int centreY, int radius)`

- **Check to see how readable the function name is when it has its parameter list – does it read like a sentence? (verb then noun?).**
- The function should not spawn any side effects.
  - Such as changing the contents of variables passed by reference.
  - Or spawning off another process that is at the same level of abstraction as the function itself.
- The function should only invoke functions that are at a lower level of abstraction than itself.
- A function should either do something or answer something, not both.

- **How to Write Good Functions**
  - It is not easy to do, especially if you are used to writing much larger functions.
  - Be prepared to write and then re-write your code several times. Each time, you will probably be breaking larger functions into smaller ones.
  - Each time you re-write (or refactor) the code it gets better (tidier, easier to read, and often shorter!).
  - Will a multiple function program be faster than a single function version that does the same thing?

- **What’s Next?**
  - Some Supporting Topics before we can start OOP:
    - Classes in java.lang, including the String class.
    - StringTokenizer class.
    - Method overloading.
    - Exceptions and catching them.
    - Arrays in memory.
    - Aliasing objects.
    - Passing objects by reference.
Some Useful Java Classes

- The classes defined in the java.lang package are automatically imported for you, since they are used quite often.
- They include:
  - The Wrapper classes
  - Math
  - Object
  - String
  - System
  - Thread
- (How do I know all this?...)

Aside – the Java API Documentation

- Application Programming Interface.
- This is the name of the huge collection of classes that support the operation of a Java program.
- The “API Docs” is the reference documentation that tells you how to use these classes.
- You can access the API docs on-line or download your own copy.
- Let’s have a look!

java.lang

- This is the only package (and all its interfaces, enums, classes, Exceptions, Errors and annotations) that is automatically imported into every Java program.
- If you need something that is in another package (and there are lots of them!), you will need to write a specific import statement for that class or for the entire package containing that class. Your IDE should be able to help you create the appropriate import statement – Eclipse does.

Aside - static Methods

- Many of these java.lang classes are utilitarian in nature – they contain many static methods:
  - static attributes and methods are loaded once into memory and not garbage collected until main is finished.
  - These methods will run faster the next time(s) they are invoked.
  - Generally, they are utility methods that do not depend on the values of a class’ attributes.

static Methods, Cont.

- static methods can be invoked without instantiation of the Object that owns them. Math.random(), for example.
- static methods and attributes are shared by all instances of a class – there is only one copy of these methods in memory.
- A static method can only invoke other static methods in its own class – you can’t have pieces of code disappearing from a static method in memory...
- This is all done for reasons of ease of use and efficiency.

Math Class

- As you would expect:
  - A collection of static constants and static mathematical methods.
  - You cannot instantiate the Math class, but why would you want to?
  - Let’s look over the API Docs.
Wrapper Classes

- Sometimes it is necessary for a primitive type value to be an Object, rather than just a primitive type.
  - Some data structures only store Objects.
  - Some Java methods only work on Objects.
- Wrapper classes also contain some useful constants and a few handy methods.

Wrapper Classes - Cont.

- Each primitive type has an associated wrapper class:
  - `char` Character
  - `int` Integer
  - `long` Long
  - `float` Float
  - `double` Double
- Each wrapper class `Object` can hold the value that would normally be contained in the primitive type variable, but now has a number of useful static methods.

Wrapper Classes - Cont.

- Integer number = new Integer(46); // "Wrapping"
- Integer num = new Integer("908");
- Integer.MAX_VALUE // gives maximum integer
- Integer.MIN_VALUE // gives minimum integer
- Integer.parseInt("453") // returns 453
- Integer.toString(653) // returns "653"
- number.equals(num) // returns false
- int aNumber = number.intValue(); // aNumber is 46

Wrapper Classes – Cont.

- The `Double` wrapper class has equivalent methods:
  - `Double.MAX_VALUE` // gives maximum double value
  - `Double.MIN_VALUE` // gives minimum double value
  - `Double.parseDouble("0.45E-3")` // returns 0.45E-3
- See the API Docs for other methods and constants dealing with NaN and -Infinity and Infinity

System Class

- We have used:
  - System.out.println()
  - System.out.print()
  - System.out.printf()
- Also:
  - System.err.println()
Other Useful System Class Methods

- `System.currentTimeMillis()`
  - Returns, as a long, the number of milliseconds elapsed since midnight Jan. 1, 1970.
- `System.exit(0)`
  - Immediate termination of your program.
- `System.getProperties()`
  - All kinds of system specific info - see the API.
- `System.getProperty(`string`)`
  - Displays single system property.
- `System.nanoTime()`
  - Time in nanoseconds

Strings, so Far

- String literals:
  "Press <enter> to continue."
- String variable declaration:
  `String testStuff;`
  or:
  `String testStuff = "A testing string.";`
- String concatenation ("addition"):
  `String testStuff = "Hello"; System.out.println(testStuff + " to me");`
  Would print the following to the console window:
  Hello to me!

Strings - Cont.

- Escape sequences in Strings:
  - These sequences can be used to put special characters into a String:
    \" a double quote
    \' a single quote
    \ a backslash
    \n a linefeed
    \r a carriage return
    \t a tab character
- For example, the code:
  `System.out.println("Hello
class! ");`
  prints the following to the screen:
  Hello
  class!

String Class - Cont.

- Since Strings are Objects they can have methods.
- String methods (67 of them!) include:
  `length()`
  `equals(OtherString)`
  `equalsIgnoreCase(OtherString)`
  `toLowerCase()`
  `toUpperCase()`
  `trim()`
  `charAt(Position)`
  `substring(Start)`
  `substring(Start, End)`

• String’s do not have any attributes.
  • See the API Docs for details on all the String class methods.
String Class - Cont.

- A few examples:

  ```java
  int i;
  boolean aBool;
  String testStuff = "A testing string."
  i = testStuff.length(); // i is 17
  aBool = testStuff.equals("a testing string."); // aBool is false
  aBool = testStuff.equalsIgnoreCase("A TESTING STRING."); // aBool is true
  ```

String Class - Cont.

- char aChar:

  ```java
  aChar = testStuff.charAt(2); // aChar is 't'
  i = testStuff.indexOf("test"); // i is 2
  ```

Aside - More about String's

- Is "Hello class" (a String literal) an Object?
  
  Yes, "Hello class".length() would return 12.

- Also, Strings are immutable – meaning that they cannot be altered, only re-assigned.
- There are no methods that can alter characters inside a string while leaving the rest alone.
- Arrays are mutable, in contrast – any element can be changed.

Aside – More Exercises

- Exercise 5 (Palindromes) is on strings.
- Exercise 6 will give you more practice with modular program design and get you thinking about Object Oriented design.