CMPE212 – Reminders

- You can now work on Exercise 4 on File I/O – this will not be covered in class.
- Assignment 2 is posted.
- Quiz 1 This Week. Described in last Thursday’s lecture. Quiz topic coverage ends on the material covered today.

Today

- Last time – Looking at useful classes in java.lang. We were in the middle of talking about the String class.
- StringBuilder class.
- Tokenizing Strings.
- Method Overloading.
- Exceptions. (if we have time)

Today

- 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
```

Today

- Is "Hello class" a String literal? Yup, "Hello class!".length() would return 12.

Today

- Also, String’s 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.

Today

- Also in java.lang: a mutable character buffer that works mostly like a String.
- (Also, StringBuffer, if you need a thread-safe version that is less efficient than StringBuilder.)
- Works by putting characters in a buffer. The buffer usually has extra room, and can grow once it is filled up.
- Three constructors:
  - Takes a String object or literal.
  - Takes an int – which will become the size of the buffer – the “capacity”.
  - An empty constructor.

Prof. Alan McLeod
StringBuilder, Cont.
• If you don't specify a buffer size, then the buffer will grow (in chunks of 16 characters) as needed. Use .capacity() to get the buffer size.
• Use .length() to get the number of occupied positions in the buffer.
• .append() can be used to append a String, a char, or another StringBuilder object to the end of the string on which it is invoked.
• .insert() does an insert and takes a character position as its first argument. Insert a char, a String, or another StringBuilder object.
• .setCharAt() changes a character at a given position.

StringBuilder, Cont.
• .delete() deletes a substring, .deleteCharAt() deletes a single character.
• More methods are listed in the API docs.
• Also, [] does not work (as it would for an array), and .equals() will not work.
• To compare for equality, use .toString() to extract a String and then use .equals() on the String objects instead.
• .compareTo() does work properly if you need to sort a collection of StringBuilder objects.
• See the demo program: StringBuilderDemo.java.

Aside – More Exercises
• Exercise 5 (Palindromes) is on strings.

Other java.lang Classes
• Object is the base class for all objects in Java. We'll need to learn about object hierarchies (Inheritance) for this to make more sense.
• Thread is a base class used to create threads in multi-threaded programs. More about this topic near the end of the course.

StringTokenizer Class
• This useful class is in the "java.util" package, so you need to have an import java.util.*; or a
  import java.util.StringTokenizer; statement at the top of your program.
• This class provides an easy way of parsing strings up into pieces, called "tokens".
• Tokens are separated by "delimiters", that you can specify, or you can accept a list of default delimiters.

StringTokenizer Class - Cont.
• The constructor method for this class is overloaded.
• So, when you create an Object of type StringTokenizer, you have three options:
  new StringTokenizer(String s)
  new StringTokenizer(String s, String delims)
  new StringTokenizer(String s, String delims, boolean returnTokens)
StringTokenizer Class - Cont.

- s is the String you want to "tokenize".
- delim is a list of delimiters, by default it is:
  
  " 
  
  or space, tab, line feed, carriage return.
  
  You can specify your own list of delimiters if you provide a different String for the second parameter.

StringTokenizer Class - Cont.

- If you supply a true for the final parameter, then delimiters will also be provided as tokens.
- The default is false - delimiters are not provided as tokens.

StringTokenizer Class - Cont.

- Here is some example code:

```java
String aString = "This is a String - Wow!";
StringTokenizer st = new StringTokenizer(aString);
System.out.println("The String has " + st.countTokens() + " tokens.");
System.out.println("The tokens are:");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
} // end while
```

Screen output:

```
The String has 6 tokens.
The tokens are:
This
is
a
String
-
Wow!
```

StringTokenizer Class - Cont.

- Note that the StringTokenizer object is emptied out as tokens are removed from it.
- You will need to re-create the object in order to tokenize it again.

Scanner Class Tokenizer

- The Scanner class has a tokenizer built into it.
- Scanner uses a regular expression or "regex" instead of the (easier to understand, but less powerful!) delimiter list.
- The default regex is "\p{javaWhitespace}+" which means "any number of whitespace characters".
- A whitespace character is a space, a tab, a linefeed, formfeed or a carriage return.
- " \t\n\f\r" in other words.
Tokenizing Demo

- See SystemPropertiesDemo.java
- Includes some old-fashioned string parsing code that uses String class methods only.

Method Overloading

- A method can have the same name in many different classes (println(), for example).
- "Overloading" is when a method name is used more than once in method declarations within the same class, (also like println()…)
- The rule is that no two methods with the same name within a class can have the same number and/or types of parameters in the method declarations. (The "NOT" rule.)

Method Overloading - Cont.

- Why bother? – Convenience!
  - Java does not have default arguments.
  - Allows the user to call a method without requiring him to supply values for all the parameters.
  - One method name can be used with many different types and combinations of parameters.
  - Allows the programmer to keep an old method definition in the class for “backwards compatibility”.

Method Overloading - Cont.

- How does it work?
  - Java looks through all methods until the parameter types match with the list of arguments supplied by the user. If none match, Java tries to cast types in order to get a match. (Only “widening” casting like int to double, however.)

Method Overloading - Cont.

- Final notes on overloading:
  - You can have as many overloaded method definitions as you want, as long as they are differentiated by the type and/or number of the parameters listed in the definition.
  - Do not change the return type – that is tacky!
  - See the getInt() and getDouble() overloaded methods in the Exercise 1 IOHelper class for example.

Exceptions

- How can a method indicate that it is unable to return what it is supposed to return?
- How can a method deliver details about the error condition?
- How can you prevent the instantiation of an Object?

The limitation of only returning a single “thing” means that you either designate error values for the “thing” or you have some other way to return the indication of an error.
Exceptions - Cont.

- The designers of Java followed conventions used by many other OOP languages - they allowed for another way to get something out of a method. However, an exception is thrown, not returned.
- Exceptions are Objects (big surprise!).
- When an error condition is encountered, a method can throw an instance of a pre-defined exception Object.
- A method can throw several exceptions, one for each possible kind of error condition.

Exceptions - Cont., Propagation

- If a method throws an exception, then that method is immediately halted and there is no need for any return value, even if the method is non-void.
- The invoking method then receives the exception - if it does not catch it, then it goes to the next invoking method - all the way to main, if necessary. Called “cascading”.
- Finally, if main does not catch the exception, your program crashes and a message is sent to the console window.

Exceptions - Cont., Message Handler

- How does an Exception Object contain information about the error condition?
  - The type of the Object:
    - IOException
    - NumberFormatException
    - FileNotFoundException
    - ArrayIndexOutOfBoundsException
    - …
  - So, the method should throw a relevant exception object.
- Turns out that exceptions can also carry a String message.

Exceptions – Cont.

- An exception is just a:
  - Bearer of "Bad Tidings"
  - The exception itself cannot do anything about the problem.
  - But it can act to stop your program!

Exceptions – Cont., Catching

- This is done with a “try/catch” block (see the next slide for the syntax).
- The compiler will force you to use try/catch blocks when you invoke methods that throw exceptions.
- (Eclipse can help build a try/catch:
  - Select the code to be surrounded.
  - Right click and choose “Surround with”, then “Try/catch block.
  - The wizard will automatically choose all the possible exceptions.)

Exceptions – Cont.

- Syntax of a “try-catch block”:

  ```java
  try {
    // block of statements that might
    // generate an exception
  }
  catch (exception_type identifier) {
    // block of statements
  }
  finally {
    // block of statements
  }
  ```
Exceptions – Cont.

- You must have at least one “catch block” after the “try block” (otherwise the try block would be useless!)
- You can have many catch blocks, one for each exception you are trying to catch.
- The code in the “finally” block, if you have one, is always executed, whether an exception is thrown, caught, or not.

Checked vs Unchecked Exceptions

- Checked exceptions must either be caught in a method or thrown from that method (using the throws clause in the method header). Examples: IOException, FileNotFoundException, ClassNotFoundException
- You should not catch an un-checked exception or an Error. Examples: OutOfMemoryError, StackOverflowError, VirtualMachineError

Checked vs Unchecked Exceptions, Cont.

- The compiler will ensure that checked exceptions are handled properly.
- Unchecked exceptions occur only at runtime and the compiler does not care about whether you try to catch them or not. But you should not catch them in any case!
- Unchecked exceptions are all sub-classes of java.lang.Error

Try With Resources

- This new syntax is very useful with Java 7 & newer versions’ improved file I/O syntax.
- More on this topic in exercise 4.

Try With Resources – Cont.

- Syntax of a “try-with-resources block”:

```java
try (instantiation; instantiation; ..) {
  // other statements that might // generate an exception
  // block of statements
} catch (exception_type identifier) {
  // block of statements
} catch (exception_type identifier) {
  // block of statements
  ...}
finally {
  // block of statements
}
```

- The instantiation(s) inside the set of ( ) immediately after the try keyword are declared resources that must be local to the try/catch block.
- Note that there is no ; at the end of the list and that there does not have to be any catch blocks.
- These resources must all implement the AutoCloseable interface, which means that the try block can close the resource when it is finished.
- Resources will be closed whether or not an exception is thrown because their scope is forced to be in the try block only.
- As a result, all use of the resource must also take place in the try block.