Section 1 - Baby GUIs

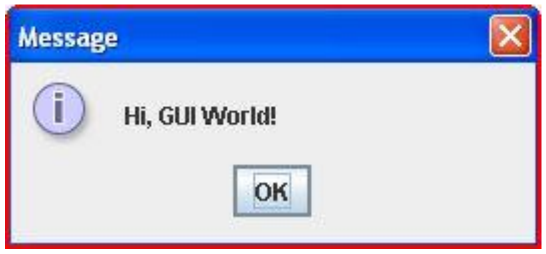
4B.1.1 GUIs

GUI stands for ***Graphical User Interface***. Some years ago, this used to be a big deal because all the programs at one time worked in ***console windows*** similar to the output of your first program, ***Hello World!***. Now, however, it is the norm to create applications that run using ***windows, menus, buttons***, etc. These are ***GUI applications*** and you've probably been using them for as long as you can remember.

Well, today, you are going to write your own.

4B.1.2 GUI World!

The program we write next will cause the following window to appear on your screen:



Hi, GUI World!

As you can see, this is a little more like it. Now it looks like we are doing something that feels like a real Windows, Mac or Android application. Admittedly, it doesn't do much, but you can already see the beginning of hope. There is an X box to close the window. There is a message. There is an **OK** button that we can press if we want to feel useful.

4B.1.3 The Source

It must be difficult to write a program that draws a window on the screen, right? Wrong. It's pretty easy. Compare this source program to the ***Hello World!*** console application:

```
import javax.swing.*;

public class HelloGui
{
   public static void main(String[] args)
   {
      JOptionPane.showMessageDialog(null, "Hi, GUI World!");
   }
}
```

As you can see, there is only one line, and this replaces the one line of the console-based application. Instead of **System.out.println()** we are using **JOptionPane.showMessageDialog()**. Always use a null in the first position inside the parentheses, and enter the string you want displayed in the second position.

The Oscar Speech, GUI Style

For a second example, we take the Oscar acceptance speech. Because we are using the GUI style output, **JOptionPane.showMessageDialog()**, the user will click **OK** after each sentence. That works well here since it gives one the impression that there really is a little person talking to you:

```
import javax.swing.*;

public class Experiment_2
{
   public static void main(String[] args)
   {
      // declare some string references (variables)
      String thxMom, thxAgent, thxFox;
      String outputString;

      // for fun, create the String object directly in the declaration
      String acceptance_speech = new String("I'd like to thank ");

      // create the rest of the  string objects to use
      // in your speech.
      thxMom = new String("my Mother, Ethel, and wife Kitty.");
      thxFox = new String("everyone at Fox and FBC.");
      thxAgent = new String("my agent and everyone at Paradigm.");

      // stand up at the podium and get settled ...
      // for this use String Literals directly in the
      // output statements.
      JOptionPane.showMessageDialog(null,
         "I didn't really expect to win ...");
      JOptionPane.showMessageDialog(null,
         "I don't even have a speech prepared!");
      JOptionPane.showMessageDialog(null, "Anyway ...");

      // now finally start to thank people.
      outputString = "First of all " + acceptance_speech + thxFox;
      JOptionPane.showMessageDialog(null, outputString);

      outputString = "Next, " + acceptance_speech + thxAgent;
      JOptionPane.showMessageDialog(null, outputString);

      outputString = "But mostly, " + acceptance_speech + thxMom;
      JOptionPane.showMessageDialog(null, outputString);
   }
}
```

4B.1.4 GUI App vs. Console App

Why do we sometimes write an application that runs in the console window (called a ***console app***) like the *Hello World!* program, and other times write a ***GUI application***? Normally, it isn't even an issue: we usually write ***GUI apps*** for any application that we plan on distributing for commercial use. The world is ***GUI***. However, there are times when ***console apps*** are desirable. For example:

- For fast design and execution of a short program that processes data, we might want to use a console app since it is generally easier to write.
- If we are not going to distribute the program, and simply want answers, we might use a console app.
- If we want to display a lot of output to the screen, it is easier to write a console app.
- If we are working in a console environment (like a Unix Shell or Command Window), we would use a console app.

Because of the rapidity with which we can write ***console apps,*** we will use them almost exclusively in this course. But we will at least introduce ***GUIs***, to give you a flavor of what it takes to create them.

Section 2 - GUI User Input

4B.2.1 showInputDialog()

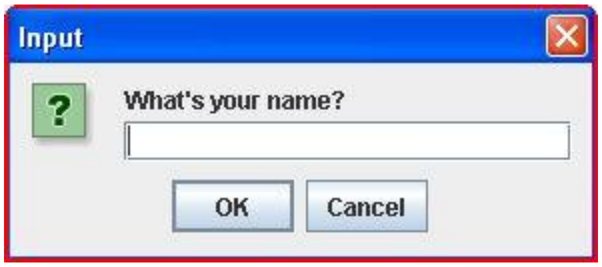
The easiest way to ask questions meant for user response in a Java GUI program is through the method **showInputDialog()** in the **JOptionPane** class**.** It works very much like the **JOptionPane** method **showMessageDialog()**, with a twist. **showMessageDialog()** just says something, while, **showInputDialog()** not only says something (actually *asks* is a better description), but it also returns an answer. In some computer languages, methods that return things to the program are called ***functions***. We've seen the **Scanner** function **nextLine()**, and its relatives, and now **showInputDialog()** works the same way.

Here is how it works:

```
// Declare a String reference to hold the user's answer
String strUserAnswer;

// ask the question and get the answer
strUserAnswer = JOptionPane.showInputDialog("What's your name? ");
```

It's that simple. When we run that code, it will result in the following pop-up:



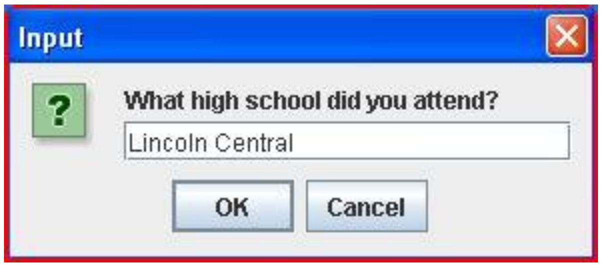
As you see, the question that we placed in the method call appears on the dialog window. Also, we have a place to enter our answer and click either **OK** or **Cancel**. If we click **OK**, the **String** we enter is ***returned*** to the program and placed into **strUserAnswer**. If we click **Cancel**, a **null** (nothing) value is placed into **strUserAnswer**, which effectively means it remains undefined.

Let's see how this gets used in a simple program.

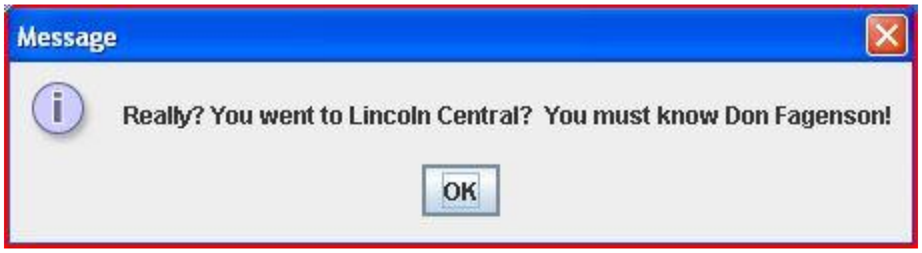
Section 3 - A GUI Input Example

4B.3.1 Your High School

Let's now retool our High School program as a baby GUI:



After the user answers the question and clicks **OK**, it displays something like this:



Really? You went to Lincoln Central? You must know Don Fagenson!

The program uses only what we have learned up to this point. Copy and run it in your own Java IDE:

```
import javax.swing.*;

public class Experiment_2
{
   public static void main(String[] args)
   {
      // declare some string variables
      String strUserAnswer;
      String strQuestion;
      String strFinalStatement;

      // Prepare a question for the user
      strQuestion = new String("What high school did you attend?");

      // Show the user the question and get a response
      strUserAnswer = JOptionPane.showInputDialog(strQuestion);

      // build up a final statement using the user's input
      strFinalStatement = "Really? You went to " + strUserAnswer + "? ";
      strFinalStatement = strFinalStatement + " You must know Don Fagenson!";

      // show the final output
      JOptionPane.showMessageDialog(null, strFinalStatement);
   }
}
```

Why "Baby"?

I called this a "baby" GUI. It's not a true GUI, because it works more like a console app than a graphical interface. A real GUI needs to have a window which *stays open* and which has various areas positioned within it: an input area, some labels, a result area, etc. We do not want windows disappearing and reappearing every time we take an action in a GUI, and that's what these baby GUIs have been doing. Such behavior is really nothing more than a console app masquerading as a GUI.

Note:

Do not submit a JOptionPane GUI as a homework assignment. I don't accept them. I am presenting them to you for your own personal use and as an intro to real GUIs.

I will present a couple optional C-modules in this course which describe how to build true GUIs. You will be allowed to substitute a GUI app version of one of your homework assignments in place of the console version.

Section 4 - Classes, Objects and Methods

4B.4.1 OOP

Java is a programming language that uses what we call ***Object-Oriented Programming,*** or ***OOP,*** as a basis for its design and implementation. What ***OOP*** means and does is the subject of this course, and I cannot define it for you this early. However there are three terms we will use frequently:

1. Class
2. Object
3. Method

I won't try to define these yet, although you've seen the terms ***class*** and ***method*** used already. Let's explore this one last time before we end for the week.

4B.4.2 Invoking Methods

If we compare our ***console app*** and our ***GUI app***, we can see some similarity between the two statements **System.out.println(...)** and **JOptionPane.showMessageDialog(...)**.

- Each ends in parentheses, with the string we want to display inside the parentheses.
- Each starts with some words separated by periods or dots.
- Each causes something to be displayed.

Both of these statements are called ***method invocations***, or ***method calls***. The last word after the last dot is the ***method name***. We talk about the **println()** ***method*** or the **showMessageDialog()** ***method.***

The first word(s) before the period(s) are what we call ***class*** or ***object*** names. You can't tell the difference between a ***class*** and an ***object*** in the above examples by just looking at them -- sometimes we have a ***class***, other times we have an ***object***. You will learn which is which. But in the current example, **JOptionPane** is a class. while**System.out** is an object. So when we call a ***method*** we *usually* use either a ***class*** or an ***object*** to make the call.

```
   JOptionPane.showMessageDialog(...)
```

uses the **JOptionPane** ***class*** to call the **showMessageDialog()** ***method***.

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

uses the **System.out** ***object*** to call the **println()** ***method***.

Terminology

When we use a particular ***class*** to call a ***method*** in this manner, i.e, **classname.methodname(),** we say we are ***dereferencing the method using the class.*** We also say we are ***using the class to dereference the method***. So above, we used the **JOptionPane** class to ***dereference*** the **showMessageDialog()** ***method***.

An alternative way to use the term is to say that we are ***dereferencing the class*** to get at the ***method***. Many authors use this version, but I prefer the verbiage of paragraph above.

4B.4.3 Defining Methods vs. Invoking Methods

The methods **println()** and **showMessageDialog()** are defined for us by Java. We only need to ***call***, or ***invoke***, them as we have done in the single statements above. However, we will sometimes ***define*** our own methods (and also ***invoke*** them). We actually saw an example in which we ***defined*** a method of our own: the **main()** method. Recall that we spoke of the ***method header*** and the ***method body*** and gave the general form of the **main()** method:

```
   public static void main(String[] args)
   {
      ;
      ;
      ;
      ;
   }
```

This seems to be a very different use of the word ***method*** than the simple ***method invocations*** we just discussed. That's because in this case we are not invoking **main()** but ***defining*** it. When you ***define*** a method it usually takes several lines and statements. When you ***invoke*** a method, you are basically just typing the method's name and expecting the method to be magically executed.

Except for the **main()** ***method***, we won't be defining our own methods for several weeks. Therefore, when I speak about a ***method***, I'll usually be talking about a ***method call,*** like the **println()** or **showMessageDialog()** above.

4B.4.4 Defining vs. Using Classes

The concepts of ***defining methods*** and ***invoking methods*** can be carried over to ***classes***, as well. In the first two projects we spoke of ***defining*** the **main** ***class***. In one case it was called **Foothill**, the other was called **HelloGui**. But directly above, I spoke about classes in a different way, explaining that they are used to ***invoke methods***. For example, I said that the **JOptionPane** ***class*** was used to call the **showMessageDialog()** ***method***. How do we understand the two seemingly disparate uses of the term ***class***?

This again comes down to whether or not we are ***defining*** the class or ***using*** the class. We will ***use*** classes (and objects) to ***call methods*** using the dot notation -- **System.out.println()**. When class names appear in this context we say we are ***using*** the class (or object). This will happen a lot in the first few weeks.

When we spoke of the **Foothill** ***class*** or **HelloGui** class, we were ***defining***, not ***using***, the ***classes***.

Except for the **main** ***class*** in every program that holds all of our program statements, we are not going to be ***defining*** classes much for now. So every program will have one main class that we ***define*** (**Foothill** or **HelloGui**, for example) and, in it we may ***use*** many different ***classes*** to help us ***invoke methods***. These classes will have names like S**ystem, JOptionPane, Scanner, Math, Double** and **String.**

Section 5 - Formatting Numbers

4B.5.1 Number Format

Java will often display the result of a computation incorrectly, as in:

```
 10.2 + 35.9 = 46.099999999999994
```

You and I both know that the right answer is **46.1**, but computers store numbers in odd ways, making such errors common. We can fix (or at least mask) this problem by limiting the number of fractional places to the right of the decimal point that we show the user. This is done with the **NumberFormat** class, part of the Java packages **java.text.*** and the **java.util.***.

It's quite simple. First you establish a **NumberFormat** ***object***, then you tell that object how many fraction digits you want it to display (at most):

```
   NumberFormat tidy = NumberFormat.getInstance(Locale.US);
   tidy.setMaximumFractionDigits(2);
```

Now that we have the **NumberFormat** object tidy, we can use it to display any numeric variable. Here is the complete **adder()** example, making use of the **NumberFormat** class to print the answer neatly.

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

public class Sample
{
   public static void main (String[] args)
   {
      double x = 10.2, y = 35.9, z;

      z = adder(x,y);

      NumberFormat tidy = NumberFormat.getInstance(Locale.US);
      tidy.setMaximumFractionDigits(2);

      System.out.println( x + " + " + y + " = " + tidy.format(z));
   }

   // method adder ------------------
   static double adder(double x, double y)
   {
      double temp;

      temp = x+y;
      return temp;
   }
}

/* ----------- output --------
10.2 + 35.9 = 46.1
--------------------------- */
```

Notice that we need to import two new packages, **java.text.*** and **java.util.***. Also, notice that, once **tidy** (or whatever you choose to call the ***object***) is defined, you can use it throughout the program whenever you want to convert a ***numeric*** type to a **String** type. Finally, you can change **Locale.US** to other locales (like **Locale.GERMANY, Locale.UK**, etc.) to display the number in the format appropriate to that region.

4B.5.2 Useful NumberFormat Methods

You saw that we invoked the **setMaximumFractionDigits()** ***method*** and gave it an argument of **2**. That obviously established an upper limit to the number of digits to be displayed. But, in fact, fewer than two, actually only one, fractional digit was displayed. That's because the other digit was a zero (0) and Java decided there was no point in displaying it. What if we wanted it? Easy: **setMinimumFractionDigits()**.

```
   NumberFormat tidy = NumberFormat.getInstance(Locale.US);
   tidy.setMaximumFractionDigits(2);
   tidy.setMinimumFractionDigits(2);
```

Now the output will be **46.10**.

This suggests there are other **NumberFormat** ***methods*** at our disposal. There are several. Two of the others you might experiment with are:

- setMinimumIntegerDigits
- setMaximumIntegerDigits

You'll see as you experiment, that the numbers show up with commas in the right places, as in **12,345.678**.

What about money?

4B.5.3 Currency

If you want to display numbers as monetary amounts, you make one small change to the above. Instead of invoking the ***method*** **getInstance()** to create a **NumberFormat** object use the method **getCurrencyInstance()**, as follows:

```
   double x = 10.2, y = 35.9, z;

   z = adder(x,y);

   NumberFormat bucks
      = NumberFormat.getCurrencyInstance(Locale.US);
   System.out.println( bucks.format(x) + " + "
      + bucks.format(y) + " = " + bucks.format(z));
```

The output looks like this:

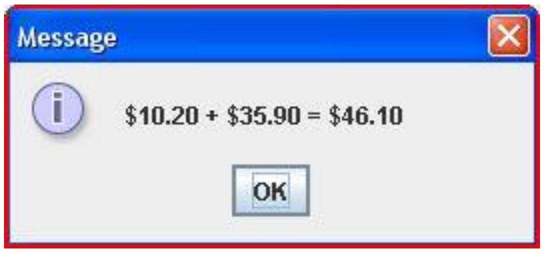
```
$10.20 + $35.90 = $46.10
```

4B.5.4 Use NumberFormat Anywhere

As I said, you can use this trick for anything, not just console output. You can use it in **JOptionPane** method

```
   JOptionPane.showMessageDialog( null,
       bucks.format(x) + " + " + bucks.format(y)
       + " = " + bucks.format(z));
```

to yield:



$10.20 + $35.90 = $46.10

or even without any immediate output:

```
   String s = bucks.format(x) + " plus " + bucks.format(y);
```

We've learned a lot in the last two sections, and can use these things to improve our mortgage calculator, yet again. However, we'll table that for another week, and instead turn to something a little different. *Passwords*.

Section 6 - A Nice Example

4B.6.1 Choosing a Password: The Elements

So how hard is it, anyway, to write a program that gets a password selection from the user? It's easy, but not short. Let's look at it in steps. Assume that the user enters the password candidate into a **string** variable, **password**. This is done in an input filter loop that keeps getting a new candidate **password** until the user enters a legal password **string**. We assume that there is a larger while loop that we don't see in these code fragments. The while loop I am speaking of is the loop that continues to get another password candidate from the user until it passes all of our tests for validity.

1. **The password has to be a certain length.**

   For this we use the **length()** method of the **string** class. It tells us the length of the **string** that is used to call it
2. `length = password.length();`

By storing the ***return value*** of the **length()** method in an **int** variable, **length**, we can reuse this number without re-invoking the **length()** method later in the code. This is a common practice whenever we use the ***return value*** of a function several times in a program.

```
    // test for reasonable length
    if (length < 6 || length > 15)
    {
       System.out.println("Password must be between 6 and 15 characters.");
       continue;
    }
```

In the above code fragment, we are assuming that we are inside a while loop that keeps going until the password is validated. The use of **continue** tell us we are going to start the ***while loop*** again, getting a new password candidate from the user.

3. The user may want to quit.

   If the user types a 'q' or 'Q' then they want to quit without typing a password.

```
4.     // test for quit first
5.     if ( length == 1
6.        && ( password.charAt(0) == 'q'  || password.charAt(0) == 'Q') )
7.     {
8.        System.out.println("No password defined.");
9.        break;  // from loop
10.    }
```

This tests the condition that the user typed one letter and that was a 'q' (or 'Q'). Make sure you understand the logic here. If the test passes, then we know the user wants out, so we **break** from the input filter ***while loop***.

11. **We examine each letter one-at-a-time in a for loop.**

    We have to look at every **char** in this **string**. That involves a ***for loop***. In each ***pass*** through the ***for loop*** we examine a character and see if it is a letter (between 'a' and 'z', or between 'A' and 'Z') or a numeral (between '0' and '9'). Note that single quotes are needed here since we are comparing each letter to the **ASCII** codes of the characters or numerals. Here is the loop that performs this test.

```
12.     validated = true; // assume innocent entering loop
13.     // allow only letters and numbers
14.     for (int k = 0; k < length; k++)
15.     {
16.        letter = password.charAt(k);    // store in local variable so we can reuse
17.        if (letter >= 'a' && letter <= 'z')
18.           continue; // the for loop, we have a good lower case letter
19.        else if (letter >= 'A' && letter <= 'Z')
20.           continue; // the for loop, we have a good upper case letter
21.        else if (letter >= '0' && letter <= '9')
22.           continue; // the for loop, we have a good numeral
```

```
      else
      {
         // if we fell through, this letter is not one of the three legal types
         System.out.println("Use only A-Z, a-z or 0-9, please.");
         validated = false;
         break; // from the for loop, leaving validated as false
      }
   }
```

An important note here is the meaning of the **break** and the **continue**. This ***for loop*** is inside a larger ***while loop*** (which we don't see, because I have taken this out-of-context). So when we encounter a **break** inside a loop, inside a *larger* loop, which loop are we breaking? The answer is the same in all languages: the innermost loop. So the **break** and **continue** apply to this ***for loop***, not the larger ***while loop***. We see here that we are examining each character in the candidate password. As a common practice, we read the current **char** into a local variable, **letter**, and test **letter**. This is in place of ***re-invoking*** **password.charAt(k)** each time we need the letter. The method above is more efficient.

4B.6.2 The Entire Password Program

Here is the whole listing.

Note: We are testing the three criteria in a slightly different order than presented above: first ***quit***, then ***length*** and finally, ***characters***.

You should really try it and study it. This is a great sample for all the concepts we have been learning.

```
import java.util.Scanner;

public class Sample
{
   public static void main (String[] args)
   {
      Scanner input = new Scanner(System.in);
      String password;
```

```
boolean validated;
char letter;
int length;

// Get password candidate from console, until it passes our tests
validated = false;
while (!validated)
{
   // get the password candidate from user
   System.out.print("Enter a password, please ('q' to quit): ");
   password = input.nextLine();

   length = password.length();

   // test for quit first
   if ( length == 1
      && ( password.charAt(0) == 'q'  || password.charAt(0) == 'Q') )
   {
      System.out.println("No password defined.");
      break;  // from loop
   }

   // test for reasonable length
   if (length < 6 || length > 15)
   {
      System.out.println(
            "Password must be between 6 and 15 characters.");
      continue;
   }

   validated = true;   // assume innocent entering loop
   // allow only letters and numbers
   for (int k = 0; k < length; k++)
   {
      letter = password.charAt(k);   // store in letter for reuse
      if (letter >= 'a' && letter <= 'z')
         continue; //  we have a good lower case letter
      else if (letter >= 'A' && letter <= 'Z')
         continue; //  we have a good upper case letter
      else if (letter >= '0' && letter <= '9')
         continue; //  we have a good numeral
      else
      {
         // this letter is not one of the three legal types
         System.out.println("Use only A-Z, a-z or 0-9, please.");
         validated = false;
         break;   // from the for loop leaving validated as false
      }
   }

   // if the above loop yielded an error, we try again
   if (!validated)
      continue;

   // if here, it is the proper length and contains legal chars
   // but is the first character a letter?
   letter = password.charAt(0);
```

```
         if (letter >= '0' && letter <= '9')
         {
            System.out.println(
                  "First character must be a letter (non-numeric).");
            validated = false;
            continue;
         }
         else
         {
            // they passed the final test
            System.out.println(
                  "Your password " + password + " has been accepted.");
            break;   // this is not really needed but is safe
         }
      }

      System.out.println("\nGood bye.\n");
   }
}

/* ----------- Output ----------------

Enter a password, please ('q' to quit): hi mom
Use only A-Z, a-z or 0-9, please.
Enter a password, please ('q' to quit): sdfkj(*&
Use only A-Z, a-z or 0-9, please.
Enter a password, please ('q' to quit): 8thhing
First character must be a letter (non-numeric).
Enter a password, please ('q' to quit): sdf
Password must be between 6 and 15 characters.
Enter a password, please ('q' to quit): adfa sdfjasdlfkjasdlfkjasdf
Password must be between 6 and 15 characters.
Enter a password, please ('q' to quit): sdf sdf
Use only A-Z, a-z or 0-9, please.
Enter a password, please ('q' to quit): asdfASDF
Your password asdfASDF has been accepted.

Good bye.

------------------------------------- */
```

4B.6.4 A Shortcut - the Character Class

As you will see, besides the ***primitive*** data types **int, double, char, float,** etc., we will discover more complex data types. These are built in to the Java language and are actually called ***classes***. One such pre-defined ***class*** is the **Character *class***. While this is similar to the **char** data type, it is not the same. One thing that the **Character *class*** offers that no primitive data type could hope to have are ***class methods***. These ***methods*** do cool things.

Start Using Character Methods in your Homework

These methods can be used to do very useful things like converting a character to UPPER CASE or testing to see if a character is a letter. Follow the link below and experiment with some of these methods.

One such method is the **isLetter()** method which tells whether or not the **char** that you are looking at is a letter (as compared with a digit or other symbol). Another method is the **isDigit()** method, which reveals whether or not a **char** value is a numeral '0' through '9'. Yet another Character method is **isLetterOrDigit()**. There are many others (**isLowerCase(), isUpperCase(), toUpperCase(), toLowerCase(),** etc.). Check out this page for a complete list of Character methods:

http://docs.oracle.com/javase/7/docs/api/java/lang/Character.html

Note - some of these methods also apply directly to **Strings** and can be called on an entire string rather than just an individual letter. See the **String** class for a complete list.

We can use these methods to shorten the above program. For example, we can use the following test instead of the lengthy one it replaces:

```
letter = password.charAt(k);
if ( Character.isLetterOrDigit(letter) )
   continue;
else
{
   // ...
}
```

This is a common example of certain built-in ***class methods***. We first need to know which ***class*** the ***method*** belongs to, in this case the **Character *class***. Then we use the class name followed by a dot (.) followed by the method name, as in **Character.isLetter()**. This is called ***dereferencing*** the **isLetter()** method through the class name **Character** (or by many authors, ***dereferencing*** the class name **Character** to get at the method **isLetter()**). The thing that goes into the parentheses is the **char** variable we want to test. (That's right, even though we are using the **Character *class*** to ***dereference*** the ***method***, we are still testing a primitive **char** variable.) The method invocation returns a **boolean true** or **false**, depending on the char and the method.

We shorten a later test in the same program:

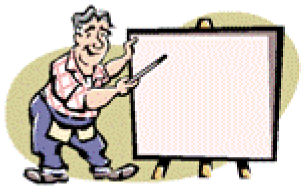
```
if ( !Character.isLetter(letter) )
{
   System.out.println("First character ...");
```

We can also save work in other programs by first converting a character to upper case before testing it (using **Character.toUpperCase()**). This way, if we want to know if the user gave us a **'D',** but will accept either that or a **'d'**, we don't have to test against both. We convert it to uppercase first and then test only for **'D'**. Here's an example that improves the above program:

```
// test for quit first
if ( length == 1
   && ( Character.toUpperCase( password.charAt(0) ) == 'Q') )
{
   System.out.println("No password defined.");
   break;  // from loop
}
```

That will just about do it for this lesson. Good work -- you learned a lot. In the next lesson, you are going to finally learn how to write your own methods, rather than rely on pre-written Java methods.

Prevent Point Loss



- **Pull common statements out of multiple if/else blocks.** If you have the same statement inside every one of your if/else blocks in a particular if/else statement, then it should be removed from each block and placed before or after the entire if/else statement. For instance if you have x = x+1; inside the if block and all the else blocks, then you are going to add 1 to x no matter what ... so you may as well add it before or after the block and remove the individual statements inside the blocks. (1 - 2 point penalty)
- **Check for errors before you compute.** If the user can supply an incorrect input to your program, test for this before, not after, you use the input to compute. If it is incorrect, you won't do the computation. In other words, test the input first, and don't do the computation if the test shows invalid input. (2 - 3 point penalty)
- **Be efficient.** If you see that you are repeating the same code, or your code looks unnaturally long, you are probably overlooking one of the tools at your disposal. Most problems can be solved using simple logic that is not excessively long. (1-3 point penalty)
- **Indent *IF* statements and *LOOPS*.** Look at examples in the lessons or text - there are many. These are expensive errors. (2-4 point penalty)

- **Use a for loop, not a while loop, when counting.** Don't use a while loop if you are going to repeat the body a fixed number of times. If you have a loop counter, use a for loop. (1 point penalty)

- **Do not adjust a loop counter inside the body of a for loop.** Managing a loop counter inside a for loop is an error. You should adjust it only in the for statement. (1 - 2 point penalty)