Section 1 - Introduction and Resources

4A.1.1 Overview

This week we have two big topics to cover.

Loops

We complete the first phase of the course by covering the control structure known as ***looping***. This capability enables our programs to do a repetitive task a certain number of times or as long as the user wishes. It confers great power to our code.

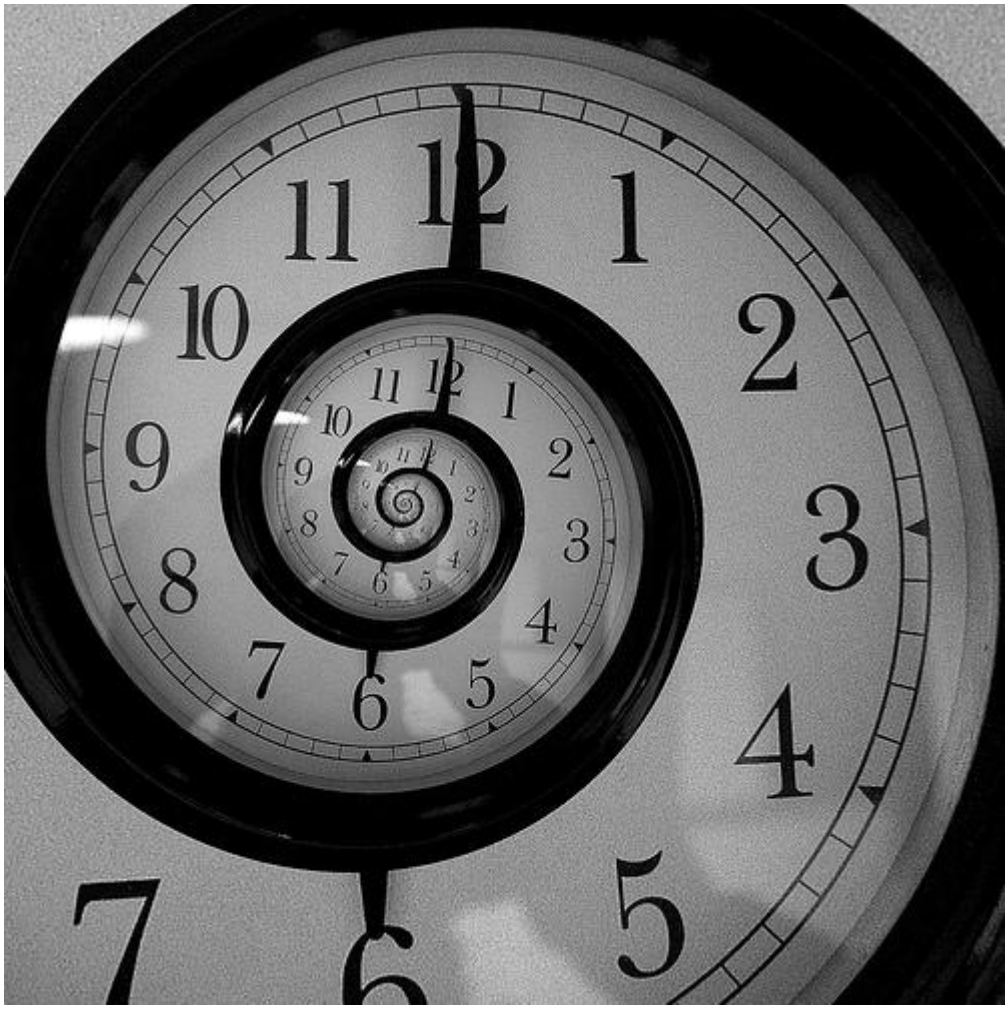
Introduction to GUI Programming

We will have a peek at Graphical User Interface, or ***GUI***, programming. For those of you interested, you will have a chance to write GUI code.

4A.1.2 Assigned Reading

As usual, read the week's modules and paste every code sample into your IDE so you can test and expand upon it. Do this *before looking at the assignment*. If you want more source material after reading the modules, look up any new terms in your textbook index, and see what it has to say about these things.

Section 2 - Loops



4A.2.1 The Loop Concept

A ***loop*** does a statement (or block of statements) zero or many times, depending on the values controlling it. The part of the statement that manages the number of iterations of the loop is called the ***loop control***. The statement, or block of statements, that is repetitively executed is called the ***loop body***. The general form is

```
<loop control>
   <loop body> ;
```

If the ***loop body*** contains more than one statement, you must enclose all the statements in ***braces*** like so:

```
<loop control>
{
   <body statement>;
   <body statement>;
   <body statement>;
   <body statement>;
   <body statement>;
}
```

It is very important to indent the ***loop body***. The compiler doesn't care, but we humans do. It makes it easy to see which and how many statements of the loop body are controlled by the ***loop control***. You will lose points if you forget to indent a loop body.

4A.2.2 Simple For Loops

If we wanted to count to **10**, printing each number as we went, a ***for-loop*** would do the job:

```
int k;

for (k = 1; k <= 10; k = k + 1)
   System.out.print( "k = " + k);
```

The explanation of this loop is what you would expect. The top line is called the ***loop control.***

The loop control says

- how to begin the ***loop,*** or the ***init*** (the first part, **k=1**, says start the loop by setting k to 1),
- how far to ***loop,*** or the ***test*** (the second part says keep looping as long as **k<=10**, and if it ever becomes false, end the loop), and
- what to do each time we execute the ***loop body,*** or the ***iteration***. (**k = k+1** says to add 1 to k at the end of each loop pass).

When the loop ends, the program executes the next statement after the loop body.

Guess, then confirm by running for yourself, what the output of the above loop looks like.

This ***for loop*** is good if you know, beforehand, how many times something will be repeated. If the number of times that something is to be repeated is determined based on user input or other factors known only at run time, then a ***while loop*** is preferred.

Assignment #2 is a great opportunity to make for loops simplify our work. For example, if I had said, "add all the odd numbers from 15 to 1001," then you should be able to do it simply with a loop rather than having one of those really long statements like ***15 + 17 + 19 + ... (sigh) ... + 999 + 1001***. Give it a shot on your own. Then compare with my solution:

```
import java.util.Scanner;

public class Foothill
{
   public static void main(String[] args)
   {
      // make these symbolic constants (finals) so if we wish to change them
      // we can do so quickly up here rather than muck around in the code
      final int LOW_VAL = 15;
      final int HIGH_VAL = 1001;

      int sum;    // holds the answer
      int k;      // loop counter

      sum = 0;    // start it off at 0 - we will add to it
      // loop, adding the ODD numbers
      for ( k = LOW_VAL; k <= HIGH_VAL; k = k + 2 )
      {
         sum = sum + k;
      }

      System.out.println("The sum of the odd numbers between " + LOW_VAL
         + " and " + HIGH_VAL + ", inclusive, is " + sum + "\n");
   }
}
/* ------------------- Sample Run --------------------

Run the program to get the answer!

---------------------- End Sample Run ---------------- */
```

4A.2.3 While Loops - A Closer Look

Simpler than for loops, ***while loops*** are used when you don't have so much bookkeeping to do. The for loop had three control sections: ***init***, ***test***, ***iteration***. The while loop only has one, the ***test***. The idea is that you test the ***condition*** in the while statement. If it is **true**, you do the loop *one more time*, and test again after each loop pass. As long as the condition is **true**, you keep on looping. If it ever becomes **false**, you end the loop and move on to the statement following the ***loop body***.

In the following excerpt from a program we ask the user for a value. If they give us a number less than 20 we slap their wrists and ask again. When they finally follow instructions, we leave them alone and end the input loop:

```
   // get the withdrawal amount from user
   System.out.print("Amount of withdrawal ($20 minimum): ");
   withdrawal = input.nextDouble();

   // trap user until they follow directions
   while (withdrawal < 20)
   {
      System.out.print("$20 Minimum.  \nPlease try again: ");
      withdrawal = input.nextDouble();
   }

/* -------------------  PORTION OF OUTPUT:

Amount of withdrawal ($20 minimum): 18
$20 Minimum.
Please try again: 25

New balance:975.03

... etc.

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

This example was obviously not a complete program but merely a code fragment. It demonstrates a ***while loop***, and also gives a reasonable ***input filter*** on which you can base future assignments. An input filter is a chunk of code that traps the user until he follows directions. But be careful: We don't always want to trap the user into giving us a value with such a strict loop. Often we want an escape so the user can quit gracefully after he sees what kind of "crazy" number we are asking him to enter. This normally entails a slightly different logic.

Here is another example that shows a complete program. This is a guessing game. The program asks the user to keep guessing until he gets it right. This is an ideal situation for while loop - we don't know how many times the loop will go. If we did, we would be using a for loop. The spec is :

Write a program that produces some int using a calculation, and do so in a manner that will make it difficult for even the programmer to know (or guess) what int will result from this calculation. Then, enter a loop that asks the user to guess the number. If the user guesses too low, tell him "too low", and if he guesses high, tell him "too high", and then allow him to guess again. When the user guesses correctly, congratulate him and end the loop.

See if you can write this program first. Then, look at the answer. I expect you to run this program and make it work. If you can't, then ask me. Otherwise, be prepared to tell me what the ***secret value*** is when you have a question about the next assignment!

```
import java.util.Scanner;

public class Foothill
{
   public static void main(String[] args)
   {
      // declare an object that can be used for console input
      Scanner inputStream = new Scanner(System.in);

      // secret value - obfuscate so the programmer doesn't even know it
      final int secretValue = (25 * 9801/35 - 200) / ( 25 + 34/12 );

      // non constant variables
      int guess;
      boolean correct; // loop control

      // welcome message
      System.out.println("Try to guess the secret value ... ");

      // now prepare to loop for the game
      correct = false;  // this gets us into the loop

      // loop as long as we are NOT correct
      while ( !correct )
      {
         System.out.print("Your guess: ");
         guess = inputStream.nextInt();
         if (guess == secretValue)
         {
            // they got it.  wish them well and set loop control variable
            System.out.println("Congratulations!!");
            correct = true;  // this will end the loop next time we test
         }

         else if (guess < secretValue)
            System.out.println("Too low.  Guess higher");
         else
            System.out.println("Too high.  Guess lower");
      }
   }
}
/* ------------------- Sample Run --------------------

Try to guess the secret value ...
Your guess: 15
Too low.  Guess higher
Your guess: 9000
Too high.  Guess lower

// THE RUN CONTINUES UNTIL THE USER EVENTUALLY GUESSES CORRECTLY WHEN ...

Congratulations!!

---------------------- End Sample Run ---------------- */
```

The above examples will give you plenty of practice so you can write simple loops in real programs. Try a few easy examples of your own.

Section 3 - A Closer Look at *For Loops*

Let's dig a little deeper into the first type of loop, the ***for loop***.

4A.3.1 *For Loops* - A Closer Look

You rent a house at $1700 per month. You can expect the rent to increase, on average, 5% per year. After five years, how much have you spent on rent? The first year is easy: $1700 * 12. But for the second year we have to multiply 1700 by 1.05 to get the new rent, and the years after that require even more computation. With loops this is very easy.

We use a variable called **rent** and multiply it by 1.05 each year to get the new year's rent. We add as we go along. Here is the loop.

```
rent = 1700;   // our initial rent
total = 0;     // keeps track of how much we paid "so far"

// loop for five years
for (year = 1; year <= 5; year = year + 1)
{
   costThisYear = rent * 12;
   total = total + costThisYear;

   // now increase the rent in prep for the next year
   rent = rent * 1.05;
}
```

Notice that there is *no semicolon* after the ***loop control.*** If there had been it would be an error:

```
// loop for five years
for (year = 1; year <= 5; year = year + 1);
{
   costThisYear = rent * 12;
   total = total + costThisYear;
   // etc ,,,
```

Look up. You are looking at the *number one error* among beginning Java programmers. The statement does compile and execute but the compiler thinks that the programmer doesn't want any statements in the ***loop body.*** The semicolon at the end of the loop control line is premature and tells the compiler that there is nothing in the ***loop body.*** Don't put a semicolon after the loop control unless you intentionally want an empty body (which may well be!).

Before going on, let's look at a smart version of the above loop in a full program. Here we use shortcuts like ***=** and **++** as well as using the alternate method of defining constants called the **final** statement. Study this for full comprehension, then you can read on:

```
public class Foothill
{
   public static void main (String[] args)
   {
      double rent, total, costThisYear;
      int year;
      final double INITIAL_RENT = 1700;

      rent = INITIAL_RENT;         // our initial rent
      total = 0;                   // keeps track of how much we paid "so far"

      // loop for five years
      for (year = 1; year <= 5; year++)
      {
         costThisYear = rent * 12;
         total += costThisYear;

         // now increase the rent in prep for the next year
         rent *= 1.05;
      }
      System.out.println("\nYou will have paid $" + total + " after five
years\n");
   }
}

/* ------------------ SAMPLE RUN -------------------------
 *
You will have paid $112722.8775 after five years

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

That's a lot of rent to pay in five years: $112,723!

The ***loop control*** is broken down into three sections, each separated by a semicolon:

```
   for ( <initialization> ; <test> ; <iteration> )
```

The first section is an ***initialization***, and the statement that you place here is executed only once prior to entering the ***loop***. Virtually any Java statement can go into this section. In fact, you can include several statements here, all intended to be initializations. If you *do* want to have multiple initialization statements in the first section, use commas -- not semicolons -- to separate them:

```
   for (total = 0, year = 1, rent = INITIAL_RENT ; year <= 5; year++)
```

The second section is a ***test*** performed at the beginning of each loop iteration (or loop ***pass***). If the test results in a **true** value, then the loop body is executed one more time. If it is **false**, the loop terminates and control passes to the statement following the ***loop body***.

The third section is the ***iteration*** and contains any statement (or statements, again separated commas if more than one) that you want to execute at the *end* of each ***loop pass***. In the above loop we wanted **year** to be iterated, so we had an appropriate assignment statement, **year = year + 1**, in the first code example, and the shorter, but equivalent, **year++**, in the second.

The ***body*** of the loop is the single statement following the control. If more than one statement is to be executed at each pass of the loop, they are placed

in braces:

```
   for (k=0, sum=0;  k < 5; k=k+1)
   {
      // statement 1
      // statement 2
      // statement 3
   }
   // the next statement after the loop.
```

Nothing in a ***for loop*** is done automatically.

Notice how much we can put into the loop control if we wish. The following does everything the longer version, above, does, but it has more of the bookkeeping built-in to the loop control:

```
public class Foothill
{
   public static void main (String[] args)
   {
      double rent, total;
      int year;
      final double INIT_RNT = 1700;

      // loop for five years
      for (total = 0, year = 1, rent = INIT_RNT;   year <= 5;   year++, rent *= 1.05)
         total += (rent * 12);
      System.out.println("\nYou will have paid $" + total + " after five
years\n");
   }
}
```

4A.3.2 Variations on the For Loop

If we wish, we may have a termination condition that has nothing to do with the number of loop passes:

```
import java.util.Scanner;

public class Sample
{
   public static void main (String[] args)
   {
      Scanner input = new Scanner(System.in);
      double withdrawal, balance;

      for (balance = 1000;   balance > 0;  )
      {
         System.out.print("Amount of withdrawal: ");

         // get the withdrawal and subtract from balance
         withdrawal = input.nextDouble();
         balance = balance - withdrawal;

         System.out.println("New balance:" +  balance);
      }

      // loop ends when we are broke
      System.out.println(
         "Sorry, you're out of money.  ATM Card confiscated.");
   }
}

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

Amount of withdrawal: 250
New balance:750.0
Amount of withdrawal: 500
New balance:250.0
Amount of withdrawal: 332
New balance:-82.0
Sorry, you're out of money.  ATM Card confiscated.

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

We see that the third section of the ***loop*** can be empty. Everything we need to do at the end of each ***loop*** is already done. We could even move the last statement of the ***loop body*** into the ***loop control:***

```
      for (balance = 1000;   balance > 0;
         System.out.println("New balance:" +  balance))
      {
         System.out.print("Amount of withdrawal: ");

         // get the withdrawal and subtract from balance
         withdrawal = input.nextDouble();
         balance = balance - withdrawal;
      }
```

Because we now have a long ***loop control,*** I broke it up onto two lines indenting the continuation line.

Normally, if a statement is sufficiently close (semantically) to the others in the block, it should stay in the ***block***. If it is conceptually separate and seems to be more of a bookkeeping statement than a data manipulation statement, it should go in the ***control***.

Finally, let's consider the general math problem, a^p, that is ***a raised to the p.*** Can you scribble on a paper the for loop needed to do this without looking below? Assume that ***a*** and **p** are each **int *variables*** and have positive values in them. How would you have the computer figure out the answer, a^p, and store it into the variable **result**? When you are ready, look at one solution:

```
result = 1;
for ( k = 0; k < p; k = k + 1 )
   result = result * a;
```

In this loop I counted to ***p*** by having k go from **0** to ***p*-1:**

```
for (k = 0; k < p; k = k + 1 )
```

This is the usual way to count in Java. This is because ***arrays*** (when we get to them) begin from **0**, not **1**, so this type of counting is more useful.

Just for fun let's see how terse we can make this code. Remember, I said that a short hand way to write **k = k + 1** is **k++**? You can also abbreviate **k = k - 1** to **k--**. Add this notation to another short-cut and we get the ***loop***:

```
result = 1;
for ( ; p > 0; p--)
   result = result * a;
```

This ***loop*** can be condensed to be even more compact and unreadable. Can you do it?

Section 4 - A Closer Look at *While Loops*

Let's dig a little deeper into the second type of loop, the ***while loop***.

4A.4.1 While Loops - A Closer Look

The only portion of the loop control is the ***test***, and this is done at the beginning of every pass, including the first. Since we have already seen a simple while loop, let's look at one that is a little more interesting.

Suppose you borrowed **$500,000** to buy a home five years ago. You were given a 6% fixed rate loan for 30 years. If you do the math, or ask the bank, you will find that your monthly payment is **$2998**. (This can vary depending on how often the interest "compounds," but the payment will be very close to this number, regardless.) You make your payments and *five years pass*. You look at your statement and see that the remaining principal on your loan is **$465,254**, which means that after five years, you still owe a lot of money!

There are still **25** years to go on your loan -- that's **300** *monthly payments* remaining. One day while you are writing the monthly mortgage check, you notice that you could pay an *additional* **$250** without feeling the pain, and you get to thinking, *"If I did this every month, how much would I save over the life of the loan?"* Even more interesting, *"How much sooner would I pay off the loan?"* Being a computer programmer, you know it would only take you half an hour to write the program to solve this little problem, and best of all, you get to use your favorite loop: ***the while loop***.

The reason that this might naturally lend itself to a ***while*** loop, as opposed to a ***for*** loop, is that you don't really know how many times the loop will cycle. If you did, you would have your answer. After all, you are trying to figure out how many payments you need to make to get the balance down to **$0**. How do you think about the problem?

Well, we start the principal off at the current amount, **$465,254**. Each month you have to do two things. First, you have to add the interest that the bank (or more likely the hedge-fund that owns the securitization into which your loan has been swallowed!) accrues each month. This is easy. You take the monthly interest, **6% / 12** or **.5%**, and multiply the principal by **1.005**. This makes the principal a little bit larger at the end of the month and that's exactly how mortgages work. Next you subtract your monthly payment, which is **$2998 + $250**, that is, your original monthly payment, plus the **$250** you are thinking about adding each month. That is, you subtract **$3248**.

You do this in a loop, over and over. Each loop pass you *multiply the remaining principal by 1.005*, and then you *subtract $3248*.

How long do you do it? That's what you want to know! You can't say *"do it for 270 months"* or *"do it for 194 months"* because the very thing you are trying to discover is how many months it takes to get the balance to 0. Instead, you do it as long as *"the principal is greater than zero."*

One of the things you'll need to do, along the way, is show how many months have passed. You can keep track of that using a **numMonths** variable that starts at **0** and increases by **1** each *loop pass*. Another thing you'll have to do when you are done is figure out how much money you have saved. Once you have calculated **numMonths**, this will be easy.

I'm going to give you a chance to figure this one out on your own before you see my solution. An important thing to keep in mind is the following: You may want to change the *interest rate* or the *amount extra* you pay each month, so it is best to not put constants like **6%** or **2998** directly into your calculations, but instead, make these values symbolic variables at the top of the program. That way, you can rerun the program with your own, personal figures and find out how much you can save under different scenarios.

For the fun of it, and to make the output a little more interesting, I decided to print out the principal at the end of each year, i.e., every 12 months. You should try to do that, as well, when you write your program. Here's mine (don't look until you've got yours ready.):

```
public class Foothill
{
   public static void main (String[] args)
   {
      double moRate, yrRate, origPayment, payment, principal, amountSaved,
extraAmount;
      int numMonths, monthsSaved;

      // annual amounts assuming $500k for 30 years, but after 5 years
      principal = 465254;     // remaining principal at the moment
      yrRate = .06;          // 6%
      origPayment = 2998;    // fixed monthly pmt bank ordered you to pay
      extraAmount = 250;

      // convert % to monthly values:
      moRate = yrRate/12;

      payment = origPayment + extraAmount;
      numMonths = 0;
      while (principal > 0)
      {
         // start by adding interest charge to principal:
         principal *= (1 + moRate);

         //  now make payment
         principal -= payment;
         numMonths++;

         // every 12 months we'll print out a balance
         if (numMonths%12 == 0)
            System.out.println("Balance after " + numMonths
               + " months =  $" + principal);
      }

      System.out.println("You paid off the loan in " + numMonths  + " months.");
      monthsSaved = 300 - numMonths;  // 25 years less the actual time
      amountSaved = 300 * origPayment - numMonths * payment;
      System.out.println("You paid it off " + monthsSaved + " months early.");
      System.out.println("You saved $" + amountSaved);
   }
}
```

```
/* ------------------ SAMPLE RUN -------------------------

Balance after 12 months =  $453883.94209402637
Balance after 24 months =  $441812.60389563965
Balance after 36 months =  $428996.73197090015
Balance after 48 months =  $415390.40510870697
Balance after 60 months =  $400944.8697781406
Balance after 72 months =  $385608.3654371737
Balance after 84 months =  $369325.93906680547
Balance after 96 months =  $352039.24826606817
Balance after 108 months =  $333686.35220236314
Balance after 120 months =  $314201.48966807226
Balance after 132 months =  $293514.8434481856
Balance after 144 months =  $271552.29015464155
Balance after 156 months =  $248235.1346309943
Balance after 168 months =  $223479.82797574427
Balance after 180 months =  $197197.66817396376
Balance after 192 months =  $169294.48226453632
Balance after 204 months =  $139670.28890416707
Balance after 216 months =  $108218.94011907952
Balance after 228 months =  $74827.7409607406
Balance after 240 months =  $39377.04570278368
Balance after 252 months =  $1739.8291322408268
You paid off the loan in 253 months.
You paid it off 47 months early.
You saved $77656.0

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

There are some things to note about this example.

1. You could have used a ***for*** loop. There is an initialization and iteration (can you find that?) that can be neatly folded into a ***loop control***. This is an example of a "toss-up." Some programmers would use a ***for***, others would use a ***while***.
2. Check out the way in which I used the ***% operator*** to print partial balances only in the whole years rather than every month.
3. There are other ways to do this. For one thing, you can come up with a single formula if you are a mathematical type. However, this example demonstrates that even if you aren't too good at math, you can devise this simple solution.
4. You might say, *"Yeah - but who's got an extra $250 each month?"* You might be surprised. Some people buy lunch six days a week at about $10 a day, when they could eat a home-packed lunch that costs $2 on average. That's $192 right there. Maybe you have a gourmet coffee every day costing $4. There is $120 you could recapture. Do you get HBO? Do you ***need*** HBO? What big purchases you put off for a few months or years when you can do the purchase and pay the extra $250? These are just thoughts.
5. Of course there are some tax issues, but, even if you are in the highest **50%** (+/-) *fed + state* bracket for all **25** years, you would still save almost ***$40,000 after taxes***. Also, even if you sell the house in five years, if you use the same strategy on your next home, then it is just as good -- if not better. The figures on the two homes, combined, will add up to savings that meet or beat the ones above.

4A.4.2 Do While Loops

The ***body*** of the ***while loop*** may not get executed even once, due to the test being **false** right off the bat. That's often desired, but not always. Here (again for some of you) is a little input filter that uses the ordinary while loop:

```
   // get the withdrawal amount from user
   System.out.print("Amount of withdrawal ($20 minimum): ");
   withdrawal = input.nextDouble();

   // trap user until they follow directions
   while (withdrawal < 20)
   {
      System.out.print("$20 Minimum.  \nPlease try again: ");
      withdrawal = input.nextDouble();
   }

/* --------------------

Amount of withdrawal ($20 minimum): 18
$20 Minimum.
Please try again: 25

New balance:975.03

... etc.

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

The problem, above, is that we have the same ***input*** statement duplicated, and it's always bad to duplicate code if we can help it. We avoid having two separate ***input*** statements by using a variation that performs the test *at the end* of the loop: the ***do/while*** statement:

```
   System.out.print("Amount of withdrawal?");
   // filter input until at least $20
   do
   {
      System.out.print("($20 Minimum) ... ");
      withdrawal = input.nextDouble();
   }
   while (withdrawal < 20);

/* ---------------------------

Amount of withdrawal?($20 Minimum) ... 4
($20 Minimum) ... 0
($20 Minimum) ... 11
($20 Minimum) ... 23

New balance:957.0
... etc.

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

As with the ***for loop***, the body is a single statement. If you only have one statement to do, you don't need the braces. If you have more, you need them.

Avoid point loss.

Always indent the body of a ***for***, ***do-while*** or ***while*** loop.

Section 5 - Exiting Loops

4A.5.1 Exit Using Break

So far, we have seen that a ***loop*** is a collection of statements that gets executed over and over until a certain **boolean** condition in the ***loop control*** becomes **false**. Sometimes it is convenient, however, to have an alternate way to leave the ***loop***. We may, for instance, find ourselves suddenly compelled to exit a ***loop*** while we are still deep in the middle of the ***loop body***.

Let's start by looking at alternative ways to exit from ***for, while*** or ***do/while loops.***

The **break** statement can be used to exit from a loop from anywhere inside its body.

As an example, say we are in a large ***loop body***, and we ask the user for a response. If the user types a **'q',** they want out immediately.

```
String response;

while( balance > 0 )
{
   // lots of statements, then...

   response = input.next();

   // if response is "quit" user wants out!
   if ( response.charAt(0) == 'q' || response.charAt(0) == 'Q' )
      break;
   // lots of other statements
}
```

Notice that the loop has some overall condition that is easy to understand in the while loop control: **balance > 0**. Meanwhile, during the processing of the loop we get input from the user that may indicate his wish to end the loop (by typing of "Q" or "quit", etc.). The point is, that we don't want to do the remainder of that particular loop pass if **response** is **"quit"**, nor do we want to do any other ***loop passes***. Therefore we test for the quit response and if we find it, we issue a break statement that gets us out of the loop instantly.

Do you remember that **response.charAt(position)** is an individual **char** from the **string**? Here we are testing the first **char** in the **string -** that is, the first letter of the word the user typed. Why do we use 0 for position instead of 1?

4A.5.2 Break is a Valid Structured Programming Statement

There are some programming authors or teachers who have the misguided view that there is something wrong with **break** statements. This is absolutely untrue and you need to be ready to answer their view (which is regurgitated from their instructors, without careful consideration). So, I'm going to help you win this argument, once and for all. (This will be good at your next programming cocktail party!).

How would they solve the problem of exiting the above loop, mid loop body, if they did not use the **break** statement?

There are really only two ways:

1. Place the remainder of the loop body (i.e, the statements after the **input.next()** method call) in a large ***if body***, controlled by

   ```
   if ( response.charAt(0) == 'q' || response.charAt(0) == 'Q' )
   ```

2. Incorporate a test for **response.charAt(0) == 'q'** somewhere in the ***while control***, as in:

   ```
   while ( (balance > 0)
      && (response.charAt(0) != 'q' )
      && (response.charAt(0) != 'Q' )
   ```

Workaround 1 is ugly because it would require indenting the entire remainder of the loop body controlled by the ***if***, making it look in some way subordinate to the upper part of the loop body:

```
while ( <condition> )
{
   <statement>;
   <statement>;
   <statement>;
   if ( response != "quit" )
   {
      <statement>;
      <statement>;
      <statement>;
   }
}
```

As you can see, this gives an incorrect impression about the flow of the loop since the statements below are of equal importance and scope as the ones above.

Workaround 2 is problematic because the condition

```
(response.charAt(0) != 'q' ) && (response.charAt(0) != 'Q' )
```

may be unrelated to the principal condition for the loop, namely the balance being positive, and thus create a long condition which is unnecessarily hard to understand. Of equal importance,

you would have to force an unnatural value into **response** before the loop begins to get it past this test (like **response = "OK"**).

These are important issues, not just for motivating the **break** statement next, but for your general education. You need to learn how to evaluate choices in Java, since there are always many ways to do the same thing. By letting you see the thought process behind our need for **break**, even though we can do without it, you will come to make the correct choice in other situations.

In some programs, you might even see this:

```
while (true)   /* forever */
{
   <statement>;
   <statement>;
   <statement>;
   <statement>;
   if ( <condition> )
      break;
   <statement>;
   <statement>;
   <statement>;
   <statement>;
}
```

In this code, the programmer could not find a way to test near the beginning or end of the loop, so the exit condition had to appear in the middle. Therefore, while it looks like an infinite loop there is an escape in the form of a **break**.

4A.5.3 Skipping the Current Loop Pass with Continue

There are times when you want to stay in the loop, but skip past the remainder of the statements in the body for just the current pass. This is done with a **continue** statement:

```
while ( <loop condition> )
{
   <statement>;
   <statement>;
   <statement>;
   <statement>;
   if ( <skip condition> )
      continue;
   <statement>;  // if <skip condition> is true
   <statement>;  // all these lines will be
   <statement>;  // skipped for this one
   <statement>;  // pass
}
```

As soon as the **continue** is reached, control will pass back to the top, and the **<loop condition>** will be evaluated for possible termination.

Caution

break and **continue** are not the correct way to deal with mutually exclusive decision choices (like different age brackets or distinct animals). The proper way to handle mutual exclusivity is consecutive ***else if*** statements, not loops.