CS125 : Introduction to Computer Science

Lecture Notes #7
Loops

Lecture 7 : Loops

Repeating our code statements many times

Last time, we added the ability to have certain sections of code run *conditionally* – i.e. sometimes they run, and sometimes they don't, depending on whether certain conditions are true or not.

ex.: Read a student score, print out whether each has passed (better than 60) or failed (60 or worse).

```
Read score, call it examScore
If examScore is greater than 60,
   print out that the student has passed
Otherwise examScore must be <= 60.
   In this case, print out that
                 the student has failed.
```

- We solved problems like that above with the `if` statement and the `if-else` statement.
- What if we want to do this for many students, and not just one? How can we repeat that code over and over again?
- And once we start repeating that code, how can we stop? How can we tell the computer that there are no more student scores?

For example:

```
Read score, call it examScore

If examScore is greater than 60,
   print out that the student has passed,
   and then repeat all this code again.

Otherwise examScore must be <= 60. If it is
   also >=0, then it is a real exam score, In
   this case, print out that the student has
   failed, and then repeat all this code
   again.

Otherwise, examScore must be < 0. This is not
   a real exam score, so it must instead be a
   signal to stop reading in exam scores. So
   do NOT repeat all this code; instead,
   stop running any of this code, and move
   on to whatever statement comes next.
```

Loops

- Loops are designed to run a particular piece of code – called the *body of the loop* – many times in succession.
- The loop is controlled via a *condition*, just as the `if` statement was. The body of the loop will continue to be repeated over and over until the condition becomes false.
- There are different kinds of loops. All of them basically do the same thing, but for each one, the particular syntax of that kind of loop is more convenient for some situations and less convenient for other situations.

The `while` loop statement

The first loop statement we will look at is the `while` loop. The form is very similar to the `if` statement:

```
while (condition)
   statement;
```

We only proceed as long as condition is true.

- If condition is false to begin with – never run statement
- If condition is true to begin with – run statement once, check condition again
- Each time you re-check condition, if it is true, run statement one more time, and then check the condition again. If it is false, leave loop without running the loop statement again, and instead run the next line of code after the loop.

For example:

```
while (i >= 0)
   i--;
System.out.println("Done!");
```

The value of `i` will get smaller and smaller until finally it is less than 0, at which point the loop will stop. The print statement is not part of the loop, and is executed only once, after you leave the loop.

The `while` loop with a compound statement

Just as with the `if` statement, any kind of statement can go into the `statement` segment of a `while` loop, including a compound statement.

```
while (condition)
{
   statement1;
   statement2;
      .
      .
      .
   statementN;
}
```

- If the `condition` is `false`, the loop ends immediately and none of the statements is run.
- If the `condition` is `true`, each of the statements is run once, and *then*, the `condition` is evaluated again.
- Every time you re-evaluate the `condition` and it is still `true`, all the statements get executed again. Once you evaluate the `condition` and it is `false`, then you stop executing the statements, exit the `while`-loop statement, and move on to whatever statement is after the `while` loop.

A common error

- Whether the internal statement is a single statement or a compound statement, what happens if the condition is *always* true?
- Answer: you get an *infinite loop* (the loop repeats endlessly).
- This is a common run-time error. A compiler cannot find your infinite loops (it could perhaps find *some* of them...but never all), and so if the combination of your code and your data produces an infinite loop, you will not be able to tell that until run-time
- Symptom 1: your program starts printing the same thing over and over and over, or printing a lot of "junk" very rapidly for a long time.
- Symptom 2: your program appears to freeze and do nothing, because it is running the same internal calculations over and over and thus is not printing anything out to you. (Careful, though! The program might just be waiting for input! A good reason to use prompts...)

A counting example

We wish to print the integers from 1 through 10. This is the same as repeating the "print an integer" action 10 separate times. If some variable `i` held our integer, we would print it using the following statement:

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

So, let's put that in a loop:

```
while (condition)
   System.out.println(i);
```

However, we want to increase `i` after each print statement, so that needs to be in the loop too:

```
while (condition)
{
   System.out.println(i);
   i++;
}
```

And finally, we need to have `i` start out at 1, and the loop should stop after `i` has passed 10 in value.

```
int i = 1;
while (i <= 10)
{
   System.out.println(i);
   i++;
}
```

An input-based example

So, how can we write the program we discussed earlier? Remember, we want to read in grades one by one, printing for each grade whether it is a passing or failing grade, until a negative number is finally entered, at which point we stop.

```
int grade;
boolean notDone = true;
while (notDone)
{
   System.out.println("Enter grade:");
   grade = Keyboard.readInt();
   if (grade < 0)
      notDone = false;
   else if (grade > 60)
      System.out.println("Passed!");
   else  // grade >=0 and grade <= 60
      System.out.println("Failed!");
}
System.out.println("Done!");
```

You could also have used `grade` itself in the condition. Right now, we change the value of the `boolean` variable once `grade` is negative. Why not just directly check if `grade` is negative instead?

```
int grade = 0;
while (grade >= 0)
{
   System.out.println("Enter grade:");
   grade = Keyboard.readInt();
   if (grade > 60)
      System.out.println("Passed!");
   else if (grade >=0)  // we know grade <= 60
      System.out.println("Failed!");
   // else if grade < 0 we do nothing
}
System.out.println("Done!");
```

- If a negative grade is entered, nothing is printed, we go back to check the condition, and the condition is `false`, so we exit the loop and print `Done!`.
- It could be argued that while this is slightly faster, the previous version is slightly easier to read. Usually, in practice, both versions should be okay.

The `for` loop statement

There is another type of loop which separates some more of the loop control code from the body of the loop. This type of loop is called a `for`-loop.

```
for (statement1; condition; statement2)
   statement3;
```

Most commonly the above statements take the following roles:

```
for (initialization statement;
        condition to continue;
           ‘‘alteration statement’’)
   body-of-loop;
```

Loop equivalence

In a language, the `while` loop is really all you need. The `for` loop is simply a more convenient form for many loops, but anything you can express using a `for` loop, you can express using a `while` loop.

The loop:

```
for (statement1; condition; statement2)
   statement3;
```

is equivalent to the code:

```
statement1;
while (condition)
{
   statement3; <-- note body comes
   statement2;   before statement2
}
```

A counting example

Once again, we wish to print the integers from 1 through 10. Using the equivalence discussed on the previous slide, we can convert our earlier `while` loop that did this printing, into a `for` loop that does the same thing:

```
int i;
for (i = 1; i<=10; i++)
   System.out.println(i);
```

- As the loop begins, the first statement inside the parenthesis gets run. That is the only time that statement gets run.
- Next, we have a series of three steps:

  1. Check condition. Exit the loop if the condition is `false`. Otherwise, continue with step 2.
  2. Run the body of the loop. Move on to step 3.
  3. Run the last statement in the parenthesis, then move on to step 1.

  which runs over and over until step 1 finally causes the loop to exit.

Note that you can also declare the integer as part of the initialization statement:

```
for (int i = 1; i<=10; i++)
   System.out.println(i);
```

and just as before, that initialization statement only gets run once. In the previous example, the assignment was only run once, and now in this example, the declaration and assignment together are only run once.

This is slightly different from the previous example in that the scope of the variable `i` is now somewhat different. We will discuss that in just a moment.

The `do-while` loop

This is another loop form that, like the `for` loop, is sometimes more convenient to use than the `while` loop and is sometimes not more convenient. The `do-while` loop is generally used when you would have a `while` loop, but want to make sure the body of the loop gets run at least once. That is because the `do-while` loop will always run the body of the loop once before checking the condition.

```
do
   statement;
while (condition);
```

is equivalent to:

```
statement;
while (condition)
   statement;
```

Scope for loops

For the most part, the scope rules for loops are nothing new. If you have a one-line statement inside your loop, then you treat that as if it had curly braces around it – just as we mentioned with the `if`-statement. That is, the following two code segments are considered equivalent, for the purposes of scope:

```
while (grade < 60)          while (grade < 60) {
   int i = 6;                  int i = 6;
                               }
```

And, just as with `if`-statements, scope with compound statements within loops works just as we've described before. For example:

```
int i = 7;    <--- i declared outside the loop
while (i < 10)
{
   int x = 5; <--- x declared inside the loop
   i++;       <--- i usable inside the loop
   System.out.println(x);  <--- x usable inside the loop
}
System.out.println(i);   <---- i usable later, outside the loop
System.out.println(x);   <---- this line is wrong; x is out of scope!
```

A special situation that needs addressing

Because you can declare a variable inside the parenthesis of a `for` loop, we must decide exactly what the scope of that variable would be. And it turns out that the scope of the parenthesis of a `for` loop ends when the loop itself ends.

```
for (int i = 1; i<=10; i++)
   System.out.println(i);
System.out.println(i);   // <-- won't compile;
                         // the variable i is
                         // no longer
                         // accessible at this
                         // point in the code;
```

You could think of this as follows, if it will help you get a clearer idea of the scope involved:

```
{   // start a stand-alone block
   int i;
   for (i = 1; i <= 10; i++)
      System.out.println(i);
}   // stand-alone block ends; i is gone
System.out.println(i);  // and so this line
                        // won't compile
```

Example 4:

```
for (int i = 1; i<=10; i++) <---- i declared
{                            *inside* the loop
   int x = 9;                   <--- x declared
   System.out.println(x);       <---  and used
                                 inside the loop
}
System.out.println(x);   <-- ERROR!
System.out.println(i);   <-- ERROR!
                          Neither variable is
                           accessible here; x went out of scope
                           every time the compound statement ended,
                           and i went out of scope when the loop
                           ended (i.e. when the loop condition
                           was evaluated to false)
```

Example 5: multiple nestings

```
int y = 35;
while (y >= 0)
{
   int x = y;
   if (x % 2 == 0)
   {
      int z = x;
      System.out.print(z + " even");
      y = y - 2;  // this will indeed alter y
   }
   else
   {
      // z is NOT in scope here
      int w = x;
      System.out.print(w + " odd");
      y = y - 1; // this will indeed alter y
   }
   y = y - 1; // this will indeed alter y
   // neither z NOR w are in scope here
}
// neither z NOR w NOR x are in scope here
```

Every alteration to `y` is permanent, since `y` is in scope for every block. When we first enter the `while` loop, `y` is 35, but due to subtractions inside the loop, `y` will not remain at 35.

Control Flow Fact

Any control flow you might want through your program can be created using a combination of:

1. Sequential execution
2. Conditionals
3. Loops

That's all you need!! And so that's all we'll look at. There are no more "necessary" control flow constructs. The rest of what we will look at are constructs designed to make programming more organized and programs easier to write and maintain. But none of it is actually *necessary* in the sense of "needing to be built into the hardware in order for programs to work".