CSC 130 Fall 2017 | Sprint 3 Quiz Review Sheet

Loops

Loops allow sections of code to be executed multiple times. You have seen two types of loops: *while loops*, which are generally used to execute code until a situation occurs (and the exact number of iterations are not known in advance), and *for loops*, which are generally used to execute code some fixed number of times or by following a clear numerical pattern.

Take for example a game that a user plays multiple times. There are two basic ways to account for this situation:

- Ask the user how many times they want to play, then play that many times (*for loop*)
- Ask the user at the end of each game if they want to keep playing (*while loop*)

Here are example code snippets that allow for multiple games to be played. A quick note: the *equals* method checks if two *String* values are the same (consequently, *equals* returns a *boolean* value).

```
System.out.print("How many games? ");
int n = in.nextInt();

for (int i = 1; i <= n; i++)
{
  System.out.println("Starting Game #" + i);
  playGame();
}

System.out.println("THE END");
```

```
String playAgain = "y";
int i = 1;

while(playAgain.equals("y"))
{
  System.out.println("Starting Game #" + i);
  playGame();
  System.out.print("Play Again? (type y if so) ");
  playAgain = in.nextLine();
  i++;
}

System.out.println("THE END");
```

In boldface above, notice that a loop continues to iterate as long as some *boolean* value is true. The *boolean* value is checked before the loops runs in the first place, then checked again after the last line of code in the loop executes. Once that *boolean* value is *false*, the loop ends and any code outside (after) the loop begins to run in the usual way.

Pattern: Running Totals

One pattern is the “running total.” This is used when you need to accumulate an answer over several iterations. It involves creating an accumulator variable which is initialized outside the loop and updated inside the loop. In the code on the left, the variable `balance` is an accumulator. One common error is to forget to update the variable based on its previous value, as seen in the box on the right.

```
// Correct Code
System.out.print("How many months? ");
int months = in.nextInt();

System.out.println("Initial investment? ");
double invest = in.nextDouble();

double balance = invest;
for (int i = 1; i <= months; i++)
{
  double interest = balance * .02;
  balance = balance + interest;
}

System.out.println("End balance: "+balance);
```

```
// Incorrect Code
System.out.print("How many months? ");
int months = in.nextInt();

System.out.println("Initial investment? ");
double invest = in.nextDouble();

double balance = invest;
for (int i = 1; i <= months; i++)
{
  double interest = balance * .02;
  balance = invest + interest;      // DOES NOT ACCUMULATE
}

System.out.println("End balance: "+balance);
```

Pattern: Stop When...

Another loop pattern is what I call "stop when." It is sometimes the case that we can articulate when a loop should *stop*, even though in a while loop we write a boolean telling when to *continue*. There are two approaches to this. First, you can think of when a loop should stop and then write the opposite of that in the while loop. For example, "stop when x is 10" is the same as "go while x is not 10." An example of this is seen in the box on the left. Alternatively, you can use the boolean not operator `!` which is illustrated on the right.

```
// Stop when x is between 5 and 10 inclusive.
while (x < 5 || x > 10) // write the opposite
{
  ...
}
```

```
// Keep going unless x is between 5 and 10 inclusive
while (!(5 <= x && x <= 10))
{
  ...
}
```

Sometimes the conditions to stop a loop or keep a loop going are tricky to write and somewhat lengthy. A strategy you can use in these cases is to create a `boolean` variable that is initialized before the loop starts and then reset at the end of the loop. The loop then continues while the boolean is false. An example of this strategy is given in the box below.

```
// Stop when the length of the String s
// is between 5 and 10 inclusive.
boolean keepGoing = s.length() < 5 || s.length() > 10;
while (keepGoing)
{
  ...
  keepGoing = s.length() < 5 || s.length() > 10;
}
```

```
// Stop when the length of the String s
// is between 5 and 10 inclusive.
boolean stop = 5 <= x && x <= 10;
while (!stop)
{
  ...
  stop = 5 <= x && x <= 10;
}
```

Note how in both cases, the opposite of an *or* statement is an *and* statement (and vice versa).

E-S-T problems

An “e-s-t problem” is one in which you are computing an extreme. Consider asking the user to enter a series of numbers. You could then compute the biggest, smallest, the one with the most zeroes, etc… In these cases, the strategy is to keep a variable that has the best candidate SO FAR. Each time through the loop, this candidate will be replaced if a better candidate is found. One common question is how to initialize the “best candidate” variable. There are two ways to go about this. One is to initialize the variable with the FIRST candidate, and the other is to use a value that you know is the worst possible candidate (so the first candidate is sure to beat it). There is an example of each of these strategies below.

```
// Compute the longEST word

System.out.println("Enter a word or q to quit: ");
String word = in.nextLine();
String bestSoFar = word;   // First word is best so far

while(!word.equals("q"))
{
  if(word.length() > bestSoFar.length())
  {
    bestSoFar = word;      // Is new word better?
  }
  System.out.println("Enter a word or q to quit: ");
  word = in.nextLine();
}
```

```
// Compute the longEST word


String word = "";
String bestSoFar = "";     // Smallest word I know!

while(!word.equals("q"))
{
  if(word.length() > bestSoFar.length())
  {
    bestSoFar = word;      // Is new word better?
  }
  System.out.println("Enter a word or q to quit: ");
  word = in.nextLine();
}
```

Looping Through Strings Forward or Backward

Another common loop type involves processing a `String` one letter at a time. To do this, remember that in a `String`, each character is stored with an index, and these indexes start at 0. The indexes go up to *but do not include* the length of the `String`. A common way to access each letter in turn is with the loop on the right. Occasionally you will need to get the characters in a `String` in reverse order. The "guts" of the loop remain unchanged but the looping logic changes, as illustrated on the left

```
// Print one letter at a time

String input = in.nextLine();

for(int i = 0; i < input.length(); i++)   // Standard "for" loop
{
   String part = input.substring(i, i+1); // 1-letter substring
   System.out.println(part);
}
```

```
// Print one letter at a time in reverse order

String input = in.nextLine();

for(int i = input.length() -1; i > -1; i--)   // Backward loop
{
   String part = input.substring(i, i+1);     // Stays same
   System.out.println(part);                  // Stays same
}
```

Going backward Discussion

Let's look at that "backward" for loop in a little more detail.

```
for( int i = input.length() -1;      i > -1;                          i--)
     // Last index of any String     // This is the rule that says       // We're going backward, so
     // is the String length         // when to keep going, not when     // decrease i instead of
     // minus 1 because indexing     // to stop.  Since we are going     // increasing i
     // starts at 0.                 // backward, we keep going as
                                     // long as i is a valid index
                                     // number (0 or more).
```

Strings as Accumulators

Strings can also act as accumulator, or running total, variables (refer to the second page of the quiz review for numerical accumulators). One hint that this might be needed is if the return type of your calculation is a String. Like all accumulator variables, you should initialize the variable before the loop starts. Usually this means making a String of no letters (an empty String). You can then add to the String inside the loop, using the same code you used with numerical accumulators.

```
// This method tracks all the words a user enters
// except it does not include words with an 'e'

String noe = "";
String input = "";

while (!input.equals("q"))
{
  if (input.length() > 0 && !input.contains("e"))
  {
    noe = noe + "<" + input + ">"; // Add input to end of noe
  }
  System.out.println("Enter a word or q to quit: ");
  input = in.nextLine();
}

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