Here are two loop statements:

break: exit from the loop

continue: exit from the loop body and jump to the loop condition

IMPORTANT: Most of the times break and continue are used in code, it is to compensate for a poorly designed loop. You are NOT allowed to use either break or continue in a loop in this course.

BAD EXAMPLE:

for (int count = 0; true; count = count + 1) {

System.out.println("Hello");

if (count == 4)

break;

}

Why is this bad? The purpose of the for loop is to put all the loop control information at the top. Someone reading this code will first assume the loop runs forever.

BAD EXAMPLE 2: (Avoiding break by doing something exactly the same, but even worse!)

for (int count = 0; count < 10; count = count + 1) {

System.out.println("Hello");

if (count == 4)

count = 10;

}

This is the same as break because it changes the count to the stopping value. It is even worse than break because it changes the value of the loop control variable inside the loop!

Someone reading the for loop will both assume that the loop runs for 10 iterations and that count increases by 1 each time, but both assumptions are wrong.

Verifying the logic of our code

While testing will catch small mistakes, it will not always help us tell if we have the correct logic for our code.

Verifying the logic of code is quite hard to do, and the following will give an example.

It will take a while to develop the skill, but it is worth working at it. Strong logic skills are very beneficial for computer programmers and computer scientists.

First Steps:

1) Decide on the desired goal (also called the postcondition) for the method.

2) Decide on a reasonable precondition for the method.

3) The key part: verify that the given precondition is really all that must be true before the method is run for it to achieve the desired goal.

- if the precondition is not sufficient, then there is a logic flaw in your code.

We will use the isEnglishPalindrome method as our example.

Goal: The method returns true if (and only if) the English letters of s read the same forwards as backwards, ignoring the case.

Desired pre-condition: The input String s is not null.

(We can quickly see that the pre-condition is necessary because if s is null, the expression s.length() will produce a NullPointerException.

The key question is whether this is all that must be true for our method to work. If it is, we have a method that will work in all possible cases, as long as s is not null.)

Weakest precondition argument:

Given a goal, we will calculate the minimum that must be true before executing the code such that the goal is true after we execute the code.

We will then compare this "weakest precondition" with the desired precondition we listed above. If they match (or if the desired precondition is stronger than the weakest precondition), then the method is logically correct.

For example, given the statement:

x = x \* 10;

and the post-condition/goal "x > 100" that we want after the statement is executed, what is the weakest precondition?

For assignment statements, we can find the weakest precondition by substituting the right hand side of the assignment for all occurrences of the left hand side in the post-condition.

In other words, for "x > 100" after the statement, and since the statement assigns x \* 10 to x, we need "x \* 10 > 100" before the statement.

"x > 100" becomes "x \* 10 > 100", and simplifying gives "x > 10"

Consider a sequence of statements:

x = x + y;

x = x + 1;

and the post-condition/goal of "x > 0". What is the weakest precondition?

We work backwards. The weakest precondition of the last statement becomes the goal of the first statement.

The weakest precondition of x = x + 1; is "x + 1 > 0" or "x > -1".

Using "x > -1" as the goal for the statement, the weakest precondition for x = x + y; is "x + y > -1".

Another example: if statements

if (x < 0)

x = x + 10;

else

x = x - 10;

with the post-condition "x = 5".

Since we do not know which branch of the if will be taken, we make the if post-condition the post-condition for each branch, and we determine the weakest precondition for each branch.

For x = x - 10, this is an assignment statement so we substitute "x - 10" for "x" in the post-condition, and the weakest pre-condition for this statement is "x - 10 = 5" or "x = 15".

For x = x + 10, this is an assignment statement so we substitute "x + 10" for "x" in the post-condition, and the weakest pre-condition for this statement is "x + 10 = 5" or "x = -5".

Finally, we add in the if condition, and we get the followign weakest precondition for the entire if:

"If x < 0, x = -5. Otherwise x = 15."

We can reason that x = -5 is always less than 0, so we can simplify the weakest precondition to:

"x = -5 or x = 15"

Loop Correctness

Finding the weakest precondition for a loop is trickier. For this we need:

1) The goal of the loop

2) A subgoal for each iteration of the loop. The subgoal should have the following properties:

2a) When the loop condition becomes false, the subgoal logically implies the goal of the loop.

2b) The subgoal is true after each loop iteration.

2c) The subgoal is true before starting the first loop iteration.

The subgoal is also called the "loop invariant".

3) Show that the loop will eventually stop.

- If this is not obvious, you should redesign your loop. (As an aside, it is ALWAYS possible to design a loop so convoluted it is impossible to figure out if it will eventually halt.)

- It is obvious for isEnglishPalindrome because for each branch of the if statement, either front increases or back decreases or both. Eventually front will equal or pass back.

Here is the code:

public static boolean isEnglishPalindrome(String s) {

int front = 0, back = s.length() - 1;

// after each iteration, the letters that appear before front read the

// same as the letters after back, but in reverse order

while (front < back) {

if (!isLetter(s.charAt(front)) // front is not a letter

front = front + 1; // so skip it

}

else if (!isLetter(s.charAt(back)) // back is not a letter

back = back - 1; // so skip it

}

else if (Character.toLowerCase(s.charAt(front)) != // front and back not are the same letter

Character.toLowerCase(s.charAt(back))) {

return false;

}

else { // otherwise front and back are the same letter

front = front + 1; // so keep going

back = back - 1;

}

}

return true;

}

Step (1): Determine the goal of the loop. In this case, the loop is almost the entire method so the goal of the loop will be very close to the goal of the method.

The loop terminates right before the return true; statement.

The goal is "The English letters of s read the same forwards as backwards, ignoring the case."

Step (2a): Find a good subgoal for the loop. The subgoal, when the loop condition is false, must give the goal for the loop. So the subgoal statement must be close to the loop goal statement, but also include "front" and "back".

The subgoal is "After each iteration, the English letters in s from index 0 up to index front-1 read the same, ignoring case, as the English letters in s from index length-1 down to back + 1"

Check: What happens when the loop condition is false? (i.e. front >= back) Does the subgoal become the loop goal?

The loop goal states that the letters of s must read the same forwards as backwards.

From the subgoal, we know that s(0 up to front-1) reads same as s(length-1 down to back+1)

Suppose the loop stops when front = back. The letters of s are broken into s(0 up to front-1), s(front), s(front+1 up to length-1). By the subgoal, these read the same as s(length-1 down to back+1), s(back), s(back-1 down to 0). So s reads same forwards as backwards!

(Do you see the logic? I applied the subgoal to each "half" of the string and used the fact that front = back.)

Suppose the loop stops when front = back + 1. The letters of s are broken into s(0 up to front-1), s(front up to length-1). By the subgoal, these read the same as s(length-1 down to back+1), s(back=front-1 down to 0). So s reads the same forwards as backwards

Step (2b): We must verify that the loop subgoal is true after each loop iteration.

We will use the weakest precondtion. Why? Assume the loop subgoal is the goal for the back loop iteration. The weakest preconditon for the back iteration becomes the goal of the second to back iteration.

The weakest precondition of the second to back iteration becomes the goal of the third to back iteration, and so on.

If the subgoal is true after each iteration, then the subgoal should also be the weakest precondition of the loop body.

To be continued: