Defensive Programming II - Loops

Atul Prakash

Break/continue in loops

- Loops can prematurely terminate if a break is encountered. In that case, control transfers to the statement after the loop.
- Loops can advance to the next iteration, skipping rest of the loop, on a continue statement.

Break/continue are convenient, not necessary

- They can be eliminated by using additional boolean variables

```
while (somecondition) {
   if (i < 10) break;
   ... do something with i;
}
```

```
boolean done = false;
while (somecondition && !done) {
   if (i < 10) done = true ;
    if (!done) {
        ... do something with i;
  }
}
```

For versus while

- A for-loop can always be implemented using a while loop

```
for (initializer; condition; advance) stmt;
```

is equivalent to

```
initializer;
while (condition) {stmt; advance;}
```

while -> for

- May be. Need to think about that one. Probably true. Example: LoopEquivalence.java

```
int max = 1000; // a test value for max.
int i = 0;
int sum = 0;
while (true) {
        sum = sum + i;
        if (sum > max) break;
        i++;
}
System.out.println("sum is " + sum);


// equivalent for loop
i = 0;
sum = 0;
for (;;i++) { // omitting the condition is treated as True
        sum = sum + i;
        if (sum > max) break;
}
System.out.println("sum is " + sum);
```

- Using while is cleaner code here.

Thinking about loops

What can we say about the values of sum, i, and k just before and just after the loop?

```
sum = 0;
for (i = 0; i < k; i++) {
        sum = sum + i;
}
```

Thinking about Loops

- Pre-condition: what is assumed to be true before we enter the loop
  - sum == 0 && i == 0
- Post-condition:
  - sum = sum of values from 0 to k-1
  - i >= k because that is the only way to exit the loop.

```
sum = 0;
for (i = 0; i < k; i++) {
        sum = sum + i;
}
```

Question

```
sum = 0;
for (i = 0; i < k; i++) {
        sum = sum + i;
}
```

- Can we conclude that i must be equal to k on exit? In other words, is the following a valid post-condition?
  - i == k

Answer

- No. If k is negative, then i would be 0 upon exiting the loop. i == k will not hold.
- But, if k is 0 or positive, then i cannot exceed k, since it is incremented by 1 every time. In that case, i == k will hold.
  - If pre-condition includes k >= 0, then, Post-condition can include i == k

Stating pre- and post-conditions

- Careful programmers use assert statements to state pre- and post-conditions. That way, if they are wrong, the code stops, rather than doing something stupid or dangerous.

Example

assert (k >= 0);

```
sum = 0;
for (i = 0; i < k; i++) {
        sum = sum + i;
}
```

assert (i == k);

Loop invariant

- loop invariants: a way to understand what a loop does
- Loop invariant: property that is true:
  - First time entering the loop
  - At the end of each round of loop (iteration)
- By implication, at the end of the loop

start (invariant holds); iterate (invariant holds) => invariant holds at exit as well

Example

```
sum = 0;
for (i = 0; i < k; i++) {
        sum = sum + i;
}
```

Here, on each iteration, sum goes up by i *and i is incremented. i++ is part of the iteration*

Loop invariant examples:

(1) i is greater than or equal to 0.

(2) k does not change during the loop

(3) sum contains the sum of values from 0 to i-1

Reasoning for i >= 0

- i >= 0 is true first time you enter the loop because i is initialized to 0 in for statement.
- i >=0 remains true after every iteration: executing the body of the loop and advancing i by 1 only increments i.
- Therefore, it is a loop invariant.

Note: integer overflows are not accounted for in the above reasoning. Some careful thinking should show that i cannot become negative even then in this code.

Stating Invariants

- Discovering and proving invariants can sometimes be hard (topic in EECS 203).
- But, for code safety, use asserts to capture what you believe to be key invariants in the code. For example, if you believe that i == k after the loop below, write it as an assert. If you are wrong, the program will fail gracefully.

```
sum = 0;
for (i = 0; i < k; i++) {
        sum = sum + i;
}
```

assert (i == k);

Invariants in Games

- Suppose you start out with 3n + 1 sticks, for some integer n, e.g., 28 sticks.
- You play a game with an opponent in which your opponent plays first. Each of you pick 1 or 2 sticks alternately. The player who picks the last stick loses.
- Can you come up with winning strategy?

Basic Idea

- Player A tries to leave 3n+1 sticks at all times. Initially, n = 9 in this example.
- If B picks 1, A picks 2. If B picks 2, A picks 1.
- # of sticks always remains of the form 3n +1.
- Eventually, n goes to 0, leaving 1 stick for B.
- Invariant after every pair of moves:
  - # of sticks = 3n+1, for some n.

Result

- We were able to show using invariants that a player can always win if he can force the number of sticks to 3n+1.
- Invariants can help you understand the result from a sequence of repeated actions - such as in games and loops.

Common Bugs in Loops: Off-by-1 error

- Getting the termination condition wrong.

```
sum = 0;
for (i = 0; i < k; i++) {
        sum = sum + i;
}
```

- What if i < k is changed to i == k?
- That changes what the program does.

Not exiting at correct point

- Suppose, we want to add numbers from i and up, and want to exit just before the sum exceeds 1000.

```
class LoopExitBug {
        public static void main(String[] args) {
                int max = 1000; // a test value for max.
                int i = 0;
                int sum = 0;
                while (true) {
                        sum = sum + i;
                        i = i + 1;
                        if (sum > max) break;
                }
                System.out.println("sum is " + sum);
        }

}
```

Can you see the bug in the logic of the code?

Analyzing the code

```
class LoopExitBug {
        public static void main(String[] args) {
                int max = 1000; // a test value for max.
                int i = 0;
                int sum = 0;
                while (true) {
                        sum = sum + i;
                        i = i + 1;
                        if (sum > max) break;
                }
                System.out.println("sum is " + sum);
        }

}
```

The loop exits only when sum > max.

Post-condition: (sum > 1000).

Violates the specs that sum should be < 1000.

Fixing the code

- You can change the exit check to
  - sum + i > max.
- But, that may not the best fix. The fix does not work if i is initialized to 10, and max is 5. We want sum to be 0 in that case.
- Better to move the modified check to before updating sum.

Summary

- One needs to think carefully when writing loops. Else, subtle bugs can arise. Good to state in comments or using assert:
  - pre-conditions
  - post-conditions
  - loop invariants