COMP 110

Prasun Dewan[1]

13. Loops

Conditionals allow us to create branches in the program execution path by allowing us to determine whether or not a statement is executed, that is, whether a statement is executed once or not at all. We will study here more sophisticated branching mechanisms, called loops, that allow us to execute a statement an arbitrary number of times – zero, one, or even infinity. Like conditionals, loops correspond to actions we perform in ordinary life such as "taking five steps forward" or "driving north on a particular road until we see a restaurant." However, it will take far more practice to master them. In fact, one chapter is not enough to understand the various ways in which they are used; therefore, in this chapter we will identify the basics of loop usage leaving other chapters to cover the more advanced looping techniques. We will look at two kinds of loops, counter-controlled and event-controlled, and show how they can be used to compute an important class of problems in which a series of values is folded into a single value. It is easy to make errors while writing loops; we will identify common errors programmers make in using loops.

Variable Execution

To understand why we might execute a statement more than once, consider a simple method that prints "hello" n times, where the value n is an argument to the method.

Before looking at the computer solution to this problem, let us try to find an algorithm we humans might use. To print "hello" n times, we might start counting from 0 using our fingers as a counter. Before we increment the counter, we would check if we have already counted to n. If we have, we stop. If we have not, we increment the counter, utter "hello", and repeat this process, as shown in Figure 1.

The arrow indicates that we want to repeat the if statement. Java does not allow such an arrow to be attached to an if statement (though some languages, termed "visual languages", do) but it provides equivalent constructs to specify repetition.

While Loops

The *while statement* is perhaps the most intuitive. It looks, syntactically, much like the if statement (without an else). Recall that an if statement has the syntax:

```
if (<boolean expression>)
      <statement>
```

```
public static void printHello(int n) {

        int counter = 0;
        if (counter < n) {
                counter = counter + 1;
                System.out.println ("hello");
        }

}
```

Figure 1.

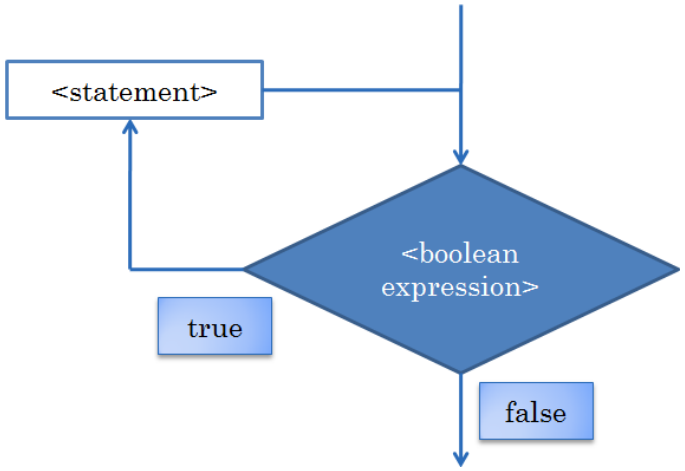

Figure 2. The while loop

A while statement has the syntax:

```
while (<boolean expression>)
      <statement>;
```

As in the case of an if conditional, <statement> can be any statement including a compound statement. Moreover, like an if conditional, a while loop executes <statement> in case <boolean expression> is **true**, and skips it otherwise. Unlike the former, however, after executing <statement>, it checks <boolean expression> again and repeats the process until <boolean expression> is **false**. <statement> is called the while *body* and <boolean expression> the *condition* of the loop. Each execution of the loop body is called an *iteration* through the loop.

The following code illustrates the use of a while loop.

```
public static void greetRepeatedly(int n) {
      int counter = 0;
      while (counter < n) {
            counter = counter + 1;
            System.out.println ("hello");
      }
}
```

It implements the algorithm above for printing "hello" n times.

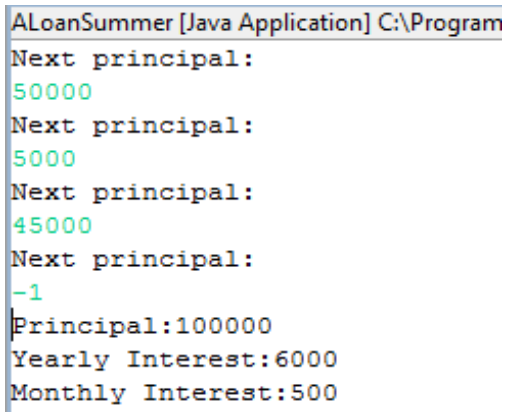

Figure 3. Sentinel-controlled summation

Developing and Testing a Loop

Let us try a more challenging problem, a variation of the loan problem of the previous chapters. Instead of summing a pair of loans, as we did before, we wish to sum up a list of loans, where the list can contain an arbitrary number of elements. The user specifies a loan by entering its principal value, and the end of the list by entering a negative integer. When list has been completely entered, the program prints the total principal, yearly interest, and monthly interest, as in Figure 3.

A value such as –1 that marks the end of a list is called the *sentinel* of the list.

Like the previous problem, this one also requires the program to execute a variable number of steps. If we can identify a pattern that is repeated in these steps, we can write a loop that executes this pattern in its body.

In the previous problem, the pattern was simple, consisting of a single statement that was repeated multiple times. This problem requires more work to identify the pattern. Try doing this problem on your own. To some of you the correct solution will be obvious.

For those of you who had difficulty writing the loop, it might help to first see how we might solve the problem for a fixed number of values.

```
Loan loan1 = readLoan();
Loan loan2 = readLoan();
Loan loan3 = readLoan();
Loan loan4 = readLoan();
Loan sumLoan = ALoan.add(
      loan1, ALoan.add(loan2, ALoan.add(loan3, loan4)))
print(sumLoan);
```

Recall that the method `add` of class `ALoan` returns a `Loan` object that is the sum of the two `Loan` objects passed to it as parameters. Here we assume methods `readLoan` and `print` for reading the next `Loan` and printing a `Loan`, respectively. We saw how such methods could be written in the previous chapter. Therefore, we will focus on the summing problem.

Unfortunately, our solution for 4 loans does not generalize to a variable number, N, of loans. Consider the statement to sum the loans. We cannot simply write:

```
print (ALoan.add(loan1, ALoan.add(loan2,  …, ALoan.add(loanN-1, loanN))));
```

because we do not know N when we write this expression. The exact number depends on how many values the user input when the program runs. But we have to commit to this expression earlier, when the program is written. Loops can execute a statement multiple times, but not part of an expression.[2]

Instead of writing a single statement that does multiple additions, we can write multiple statements, each of which does a single addition:

```
Loan sumLoan = ALoan.add(loan1, loan2);
sumLoan = ALoan.add(sumLoan, loan3);
sumLoan = ALoan.add(sumLoan, loan4);
```

Thus, we keep a running total of the loans in variable, `sumLoan`, and add each subsequent input loan to the running total. This solution can be generalized to N values:

```
Loan sumLoan = ALoan.add(loan1, loan2);
sumLoan = ALoan.add(sumLoan, loan3);
…
sumLoan = ALoan.add(sumLoan, loanN);
```

It requires the computer to execute a variable number of statements, but we have seen that loop can be used for this purpose.

However, to write such a loop, we would have to create N Loan variables to store the list of Loan values entered by the user. When we study arrays and vectors, we will see that we can, in effect, create an arbitrary number of variables. However, for this problem, we do not really need to read each input value into a different variable. That would be necessary only if we needed to access all of these values simultaneously. Once we have added a variable to the running total, we no longer need its value, and can thus use it to read the next value. The following alternative implementation of the fixed-size problem illustrates this algorithm:

```
1. Loan loan1 = readLoan();
2. Loan loan2 = readLoan();
3. Loan sumLoan = ALoan.add(loan1, loan2);
4. loan1 = readLoan();
5. sumLoan = ALoan.add(sumLoan, loan1);
6. loan1 = readLoan();
7. sumLoan = ALoan.add(sumLoan, loan1);
8. print(sumLoan);
```

[2] We shall see that we can use an alternative concept for repeating a computation, called recursion, that will allow us to repeat an expression calculation multiple times.

We can see a pattern developing after statement 3 – statements 6 and 7 are repetitions of statements 4 and 5 respectively. We can actually write a more elegant solution with fewer variables in which the pattern develops earlier.

```
1. Loan sumLoan = readLoan();
2. Loan nextLoan = readLoan();
3. sumLoan = ALoan.add(sumLoan, nextLoan);
4. nextLoan = readLoan();
5. sumLoan = ALoan.add(sumLoan, nextLoan);
6. nextLoan = readLoan();
7. sumLoan = ALoan.add(sumLoan, nextLoan);
8. print(sumLoan);
```

Now the pattern develops after line 2. Thus, all statements after it can be executed as part of a loop that terminates when we read a negative sentinel value:

```
1. Loan sumLoan = readLoan();
2. Loan nextLoan = readLoan();
2'. while (nextLoan.getPrincipal() >= 0) {
3.    sumLoan = ALoan.add(sumLoan, nextLoan);
4.    nextLoan = readLoan();
5. }
6. print(sumLoan);
```

Lines 1, 2, 3, and 4 are the same as before. However, by adding the loop, we make sure that after executing line 4, the program goes back and executes line 3, which is essentially line 5 of the original solution. Since line 4 is the same as line 6 of the original solution, we seem to have captured the desired pattern in this loop.

Whenever we write a loop we must check the boundary conditions to see what it does with them. We expect the user to add at least two loans, but our problem specification allows the user to enter an arbitrary number of loans.

Let us see, first, what happens if the user enters no number at all? The program will block forever, waiting for input, which seems acceptable.

What if the user enters a single negative number to end the interaction with the program? The program will block for ever, waiting for the second value, which does not seem acceptable.

We can fix this problem by initializing `sumLoan` to a loan of 0 dollars, rather than the first loan read.

```
Loan sumLoan = new ALoan(0);
Loan nextLoan = readLoan();
while (nextLoan.getPrincipal() >= 0) {
      sumLoan = ALoan.add(sumLoan, nextLoan);
      nextLoan = readLoan();
}
return sumLoan;
```

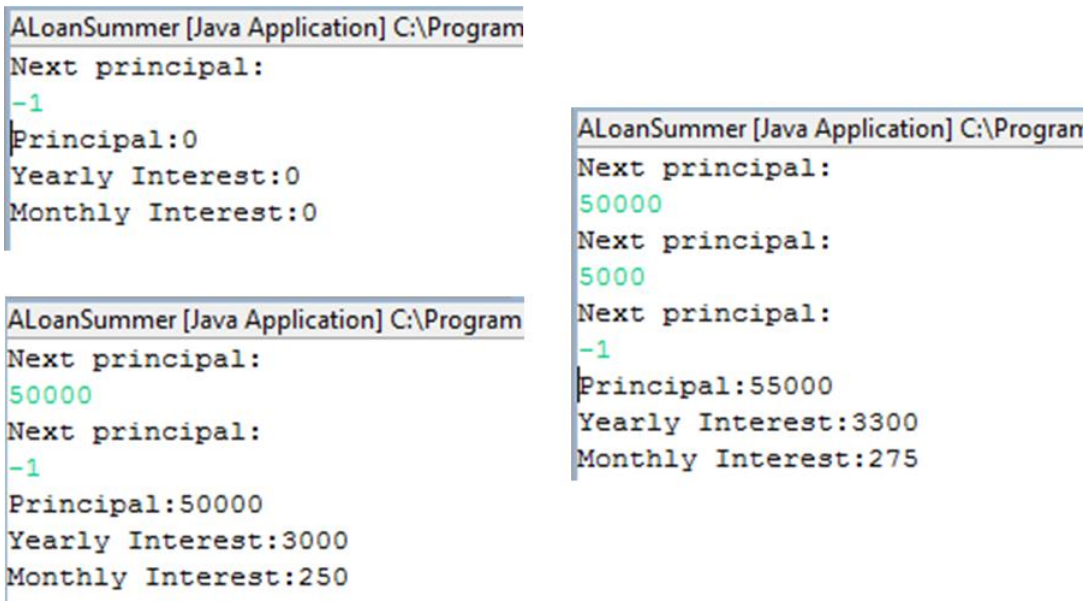

Figure 4. Testing the loop: (top-left) sentinel is the first entry; (bottom-left) a single valid entry; (right) multiple valid entries

This looks right. To test whether this is indeed the case, we would try it for the boundary conditions, lists of less than two elements, and also some lists of two and more elements (Figure 4).

The program works on our test data. There are techniques to formally prove its correctness, but we will not study them in this book, relying on the kind of process we went through to convince ourselves that it works.

Sentinel-Controlled Folding

What we have seen here is general technique for solving a whole class of problems. For instance, multiplying N numbers entered by the user has a very similar solution:

```
int product = 1;
int nextNum = Console.readInt();
while (nextNum >= 0) {
   product = product*num;
   nextNum = Console.readInt();
}
print(product);
```

Here we assume, as we do in Mathematics, that the product of an empty list of numbers is 1. As we can see, there is a one-to-one correspondence between the statements of the two programs.

In general, the technique used in these two programs works for problems with the following properties:

- We are given a binary operation, f, with the signature:

  T, T → T

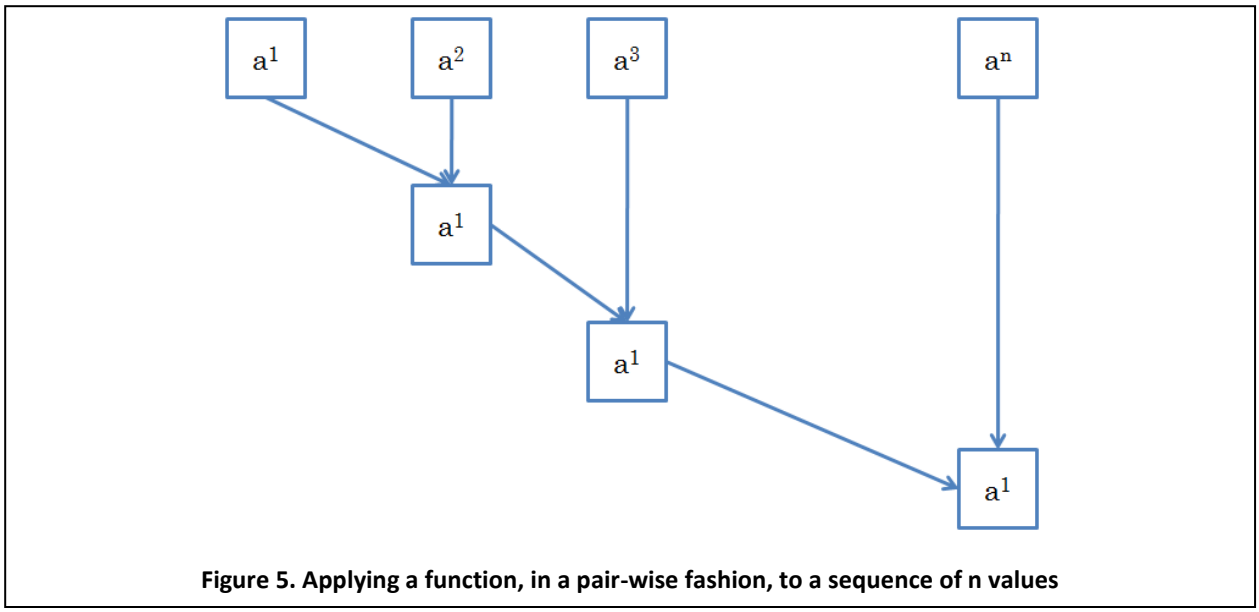

Figure 5. Applying a function, in a pair-wise fashion, to a sequence of n values

- For instance, in the summation problem, the binary operation was + and in the product example, it was *. The operation may be specified using infix or function syntax. In our general solution, will assume the method invocation syntax, though the pattern will also apply to infix syntax.
- We wish to apply f, pair-wise, to a sequence of n values, as shown in Figure 5
- The function has an identity, I. That is, there exists a value I such that:

  $f(a^j, I) == a^j$

  for any item a^j in the list. In the summation problem, I = new Loan(0), and in the product example, I = 1.
- If the list is empty, the result is I.
- If the list has one element, the result is that element.
- The end of the list is indicated by a sentinel whose type is the same as the type of the arguments and result of f.

A general solution to such problems is:

```
T result = I;
T nextValue = getNextValue()
while (!isSentinel(nextValue)) {
   result = f(result, nextValue);
   nextValue = getNextValue(..);
}
```

where `getNextValue` is a function that gets the next value in the list and `isSentinel` is a `boolean` function or expression that checks if its argument is a sentinel.

We will call a binary function, f, with the signature, T,T -> T, a *folding* function, and problems that apply it cumulatively to a list as folding problems, since we essentially fold the result of the scanned list into a

single result. Folding problems do not require arrays or vectors to store the list elements, as we have seen here. Of course, not all list-processing problems are folding problems. For instance, as you will see in later courses, if we wanted to sort the list, we would need to keep all elements in memory simultaneously.

Cumulative Assignment

The assignment pattern used above to fold values:

```
result = f(result, value);
```

where the new value assigned to the LHS side is the result of performing some operation on its current value, appears in many problems. To save us the trouble of typing the LHS twice,[3] for predefined infix operator, Java supports a short hand for making such assignments. If ○ is such an operator, it supports the cumulative assignment statement of the form:

```
<variable>  <operator>=  <expression>
```

which is equivalent to:

```
<variable>  = <variable> <operator> <expression>
```

Thus:

```
sum += nextValue
```

is equivalent to:

```
sum = sum + nextValue
```

and

```
product *= nextValue
```

is equivalent to:

```
product = product*nextValue
```

Counter-Controlled Folding

To get some more practice with while loops, let us consider a variation of the product problem, shown in Figure 6, in which the user inputs the number of elements to be input instead of entering a sentinel.

This problem combines elements of the problems of printing “hello” n times and summing a sentinel-based list. As in the first problem, the number of steps is known before we execute the loop, and as in the second problem, the loop involves summing a list of values:

[3] and in fact do a more efficient compilation

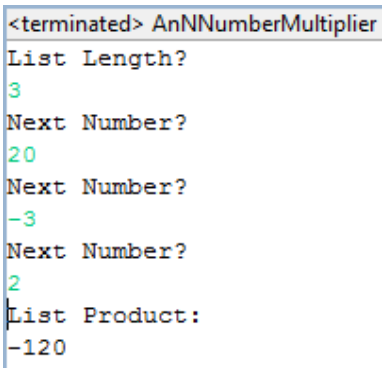

Figure 6. Counter-controlled Folding

```
int product = 1;
int n = readNumElements();
int counter = 0;
while (counter < n) {
      int nextNum = readNum();
      product *= nextNum;
      counter += 1;
}
```

Therefore, as in Figure 2, we maintain a counter, which determines how many times the loop is executed, and as in Figure 3, we fold the input values into a list product.

Off –by-One Errors

Yet another variation of this problem is to write a loop that finds the product of the first n positive integers:

```
1*2*3* … n
```

which is referred to as the factorial of n. We no longer need to read user input, since the program can create a counter-based loop to generate the numbers. The same counter also determines when the loop is done. Thus, we may write:

```
int product = 1;
int counter = 0;
while (counter < n) {
            product *= counter;
            counter += 1;
}
System.out.println (product);
```

However, this solution will always give the answer 0, since it computes the product:

```
1*0*1*2* .. n-1
```

We could start the counter from 1:

```
int product = 1;
int counter = 1;
while (counter < n) {
            product *= counter;
            counter += 1;
}
System.out.println (product);
```

However, this solution also does not work, since it omits the n in the multiplication, computing:

```
1*1*2* .. n-1
```

We can fix this problem by changing the termination condition of the loop:

```
int product = 1;
int counter = 1;
while (counter <= n) {
            product *= counter;
            counter += 1;
}
System.out.println (product);
```

We seem to have finally got our solution. The “off-by-one” errors we made in our two attempts above, where the execution includes or omits an extra iteration, are common. Therefore, it is very important to make the extra effort of checking that the loops we write do not have them.

Sometimes off-by-one errors are made because of two possible interpretations of a counter. In general, a counter keeps track of how many loop iterations have occurred. It can do so by storing, at loop entry:

- the number of next iteration.
- the number of the previous iteration.

Off-by-one errors occur when we one interpretation in one part of our code and the other in another part. For instance, in the second erroneous solution, we initialized the counter to 1, which means that (when the loop is entered) it is intended to store the number of the next iteration. However, our loop condition assumes it is stores the number of the previous iteration, terminating the loop if the counter has the value n. Similarly, the first erroneous solution combines both interpretations.

Since it is easy to make such mistakes, we should make the exact role of the counter explicit in the program. For instance, in this example, based on whether it stores the value of the previous or next iteration, we could call it previousMultiplier or nextMultiplier. Consider the first erroneous solution with the change in the counter name:

```
int product = 1
int previousMultiplier = 0;
while (previousMultiplier < n) {
            product *= previousMultiplier
            previousMultiplier += 1;
}
System.out.println (product);
```

Now the error is more obvious, because in each loop iteration we are multiplying the previous value of the multiplier. To correct it, we should increment the counter before multiplying it:

```
int product = 1
int previousMultiplier = 0;
while (previousMultiplier < n) {
      previousMultiplier += 1;
      product *= previousMultiplier;
}
System.out.println (product);
```

Similarly, if we rewrite the second solution with the change in the counter name:

```
int product = 1
int nextMultiplier = 1;
while (nextMultiplier < n) {
      product *= nextMultiplier;
      nextMultiplier += 1;
}
System.out.println (product);
```

it is easier to see that the termination condition does not include n in the multiplication. This approach makes us give more thought to the name of the counter, but makes the program clearer and easy to debug.

Most of you are probably more comfortable with the second solution, in which the counter keeps track of the number of the next iteration. Therefore, it is best to use this approach, incrementing the counter at the end rather than the beginning of the loop.

Avoiding Infinite Loops

Consider the following loop to find the product of the first n numbers:

```
int product = 1
int nextMultiplier = 1;
```

```
while (nextMultiplier < n) {
      product *= nextMultiplier;
}
System.out.println (product);
```

The loop executes its body an infinite number of times, that is, it does not terminate. The problem here is that in the loop body we did not change any variable on which the loop condition depends.

Here is another kind of an infinite loop:

```
int product = 1
int nextMultiplier = 1;
while (nextMultiplier < n) {
      product *= nextMultiplier;
      nextMultiplier -= 1;
}
System.out.println (product);
```

The problem here is that even though the counter is being updated, it will never get a value that makes the loop condition false. In other words, the loop condition does not converge to a false value.

Thus, we must be careful that:

1) In the loop body the variables affecting the loop condition are updated.
2) These variables will take values that will make the expression false after some finite number of steps.

We know we are in an infinite loop if the Java program freezes, that is, it does not respond to any commands, lets us enter data, or select a menu. If this happens, you can kill the program by typing CTRL-C in the console window.

Decrementing the Counter

So far, all of our counters have been incremented during each iteration of the loop. It is also possible to write loops that decrement the counter. Consider again the problem of finding the sum of the first n numbers. We essentially did a forward multiplication, multiplying increasing numbers to the product:

```
1 * 2 * 3 * … * n
```

Instead, we could have also done a backwards product, multiplying decreasing numbers to the product:

```
n * .. * 3 * 2 *1
```

The following loop implements a backwards product, starting a counter at n and decrementing it by 1 it until it reaches 0.

```
int product = 1;
while (n > 0) {
      product *= n;
      n -= 1;
}
```

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

Notice that we are using the variable `n` as also the counter, since we do not need its value at the end of the loop.

Increment/Decrement Assignment

Incrementing and decrementing of a variable occurs so often in assignment statements that Java provides short hands for entering such statements. It provides the increment statement of the form:

```
<variable>++
```

which is equivalent to:

```
<variable> = <variable> + 1
```

It also provides a decrement statement:

```
<variable> --
```

which is equivalent to:

```
<variable> = <variable> - 1
```

Counter-controlled vs. Event-controlled Loops

We can classify the loops we have seen here into two categories: *counter-controlled* and *event-controlled*. In both kinds of loops, the number of times the loop body is executed is variable, that is, determined when the program is executed and not when it is written. In the case of a *counter-controlled* loop, the number of loop iterations is known before the loop is executed, while in case of an *event-controlled* loop, it is determined while the loop is executing.

Counter-controlled loops have the following properties:

1) They initialize a counter to some value.
2) They change the counter by some step (1, 2) at the beginning or end of the loop body. As mentioned before, it is best to change the counter at the end of the loop body.
3) They terminate execution when the counter goes up/goes down to some *limit* that was computed before the loop was started.

Event-controlled loops, on the other hand, test for one or more events that occurs while the loop is executing (such as input of a sentinel value) and make the loop condition false.

Guarding Against Limits of the Wrong Sign

We were implicitly assuming in all of our counter-controlled loops that the upper limit, `n` was not negative. This may not be guaranteed, especially if the value is input by the user. Therefore, we must take this into consideration when we write loops. Consider again the loop that does a forward sum of the first `n` numbers:

```
int product = 1
int nextMultiplicand = 1;
while (nextMultiplicand <= n) {
            product *= nextMultiplicand;
            nextMultiplicand += 1;
}
System.out.println (product);
```

Instead of this solution, we might have written:

```
int product = 1
int nextMultiplicand = 1;
while (nextMultiplicand != n) {
            product *= nextMultiplicand;
            nextMultiplicand += 1;
}
System.out.println (product);
```

These two solutions are the same as long as `n` is not negative. If n is negative, however, the test:

```
nextMultiplicand <= n
```

will fail in the first iteration while the test:

```
nextMultiplicand != n
```

will always succeed, resulting in an infinite loop. In general, using the != operator to compare a counter with its limit is a signal that we might not have taken into account limits that have an unexpected sign.

Nested Loops

Consider an extension of the problem of finding the product of the first `n` numbers, that is, finding the factorial of `n`. Assume that the value of `n` is input by the user. Suppose also that we are interested in finding the factorial of a list of numbers that is terminated by a negative sentinel, as shown in Figure 7.

In the above interaction, we need a loop to process the list of `n`'s entered by the user. In addition, we need the loop is we have been using to multiply the first `n` numbers. The multiplying loop must be nested in the loop that reads the list, as shown below:

```
<terminated> AFactorialList
3
factorial = 6
2
factorial = 2
-1
```

Figure 7. Factorials of a list of input numbers

```
int n = Console.readInt();
while (n >= 0) {
      int product = 1;
      int nextMultiplicand = 1;
      while (nextMultiplicand <= n){
            product *= nextMultiplicand;
            nextMultiplicand++;
      }
      System.out.println("factorial = " + product);
      n = Console.readInt();
}
```

The outer loop is much like the loop we saw in sentinel-controlled folding. It repeatedly executes its body until a sentinel is found. In each iteration, the body of the outer loop prints the value of factorial computed by the inner loop, and sets `n` to the next input integer. The rest of the body of the outer loop is identical to the forward product solution we saw in Figure ?. It initializes the values of the product and counter variables and executes the loop for multiplying the first n values.

It is instructive to write a nested loop that uses the backward product solution show here:

```
int n = Console.readInt();
while (n >= 0) {
      int product = 1;
      while (n > 0){
            product *= n;
            n--;
      }
      System.out.println("factorial = " + product);
      n = Console.readInt();
}
```

Here, instead of grafting Figure 7 into the outer loop, we have grafted Figure 16. Though economical in how many variables it uses, this solution is potentially more dangerous, since both loops change the value of `n`., potentially interfering with each other. It actually works, since changes made to n by the outer loop do not depend on its previous value. As a result, the variable can be safely modified by the inner loop without interfering with the assumptions made by the outer loop. However, it is difficult to convince ourselves that this solution indeed works.

In general, nested loops can be hard to read. Moreover, they are error-prone since two counters or sentinel variables may be created, and it is easy to give them the same names, which may cause them to interfere with each other.

Moreover, because the inner loop has two levels of indentation, the statements in it often wrap around the screen, making them hard to read.

It is best to write a separate method for each loop, which makes the solution clearer, as shown below:

```
public static void listFactorial() {
      int newVal = Console.readInt();
      while (newVal >= 0) {
            System.out.println("factorial = " + factorial(n));
            newVal = Console.readInt();
      }
}
public static int factorial (int n)  {
      int product = 1;
      while (n > 0) {
            product *=  n;
            n--;
      }
      return product;
}
```

Here we do not need to worry about the two changes to `n` interfering with each other since each method gets its own private `n`. Separating each loop in a separate method also increases reusability since the method can be called from multiple code fragments. For instance, the `factorial` function can be used in this problem and the simpler problem of computing the factorial for a single `n`. In contrast, an inner loop can be used only in a single outer loop.

break

Our event-controlled loops had an annoying feature: we had to write two statements to receive the events that control the loops, once at the beginning of the loop and once inside. For instance, in `listFactorial`, we repeated the statement:

```
newVal = Console.readInt();
```

that received the input value. But the whole point of loops is to input a statement once and have the loop execute the statement the required number of times!

Unfortunately, we cannot prevent such repetition using just the while loop. The reason is that the event must be received before the while condition is first tested, and must also be received inside the loop, for subsequent iterations.

Fortunately, Java provides a special mechanism, the break statement, which allows us to solve this problem. This statement can be enclosed inside a loop. When it is executed, it causes termination of the loop, regardless of the loop condition. The following code shows how it can be used to avoid repeating the input statement:

```
public static void breakingListFactorial() {
      while (true) {
            int newVal = Console.readInt();
            if (newVal < 0) break;
            System.out.println("factorial = " + factorial(n));
      }
}
```

Here, two conditions are tested to check if the loop should be terminated, (1) the while condition:

```
true
```

and the breaking condition:

```
n < 0
```

The while condition is always true, and thus never causes loop termination. The real terminating condition, then, is the breaking condition. This condition, unlike the while condition, does not have to be tested at the beginning of a loop iteration. It can be tested anywhere in the loop. In this example, it is checked after the user value is read, thereby allowing the same input statement to be used to receive the first and subsequent input values. To write such concise programs, many programmers prefer to make the real loop termination condition as a break condition rather than a while condition.

Animating Shuttle

As a more interesting example of loops, consider again the shuttle location object we created earlier. Suppose we create a variation of it, `AnAnimatingShuttleLocation`, that provides an additional method, `animateFromOrigin` (Figure 8), that animates the path the shuttle took from the origin to its current position. As we don't know actually know this exact path, we will make the simplifying assumption that the shuttle first went vertically up to its current Y position (Figure 9), and then horizontally went to its current X position (Figure 10).

Before we try to code this object, let us first define what exactly it means for a method to animate the shuttle from one location to another one. It could imply that the method:

1) moves the shuttle a distance, D, in a straight line towards the destination.
2) checks if the shuttle has reached its destination. If yes, the method terminates; otherwise it repeats the above two steps.

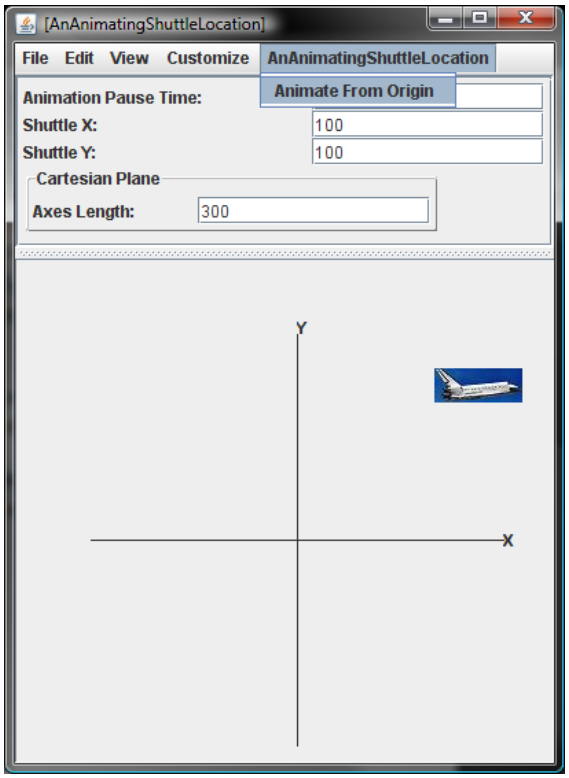

Figure 8. Instance of `AnAnimatingShuttleLocation`

However, this algorithm does not work as the computer will execute the steps so fast that the user will not see the intermediate positions of the shuttle – the shuttle will seem to reach its destination instantly. Thus, we need the method to pause for some time after step 1.

In summary, the method:

1) moves the shuttle a distance, D, in a straight line towards the destination.
2) pauses for some time T to make the shuttle stays at its current location.
3) checks if the shuttle has reached its destination. If yes, the method terminates; otherwise it repeats the above three steps.

We will assume that D is given by a named constant and time T is specified by the user-defined property, `AnimationPauseTime`.

```
int animationPauseTime;
public int getAnimationPauseTime() {
      return animationPauseTime;
}

public void setAnimationPauseTime(int newVal) {
      animationPauseTime = newVal;
}
```

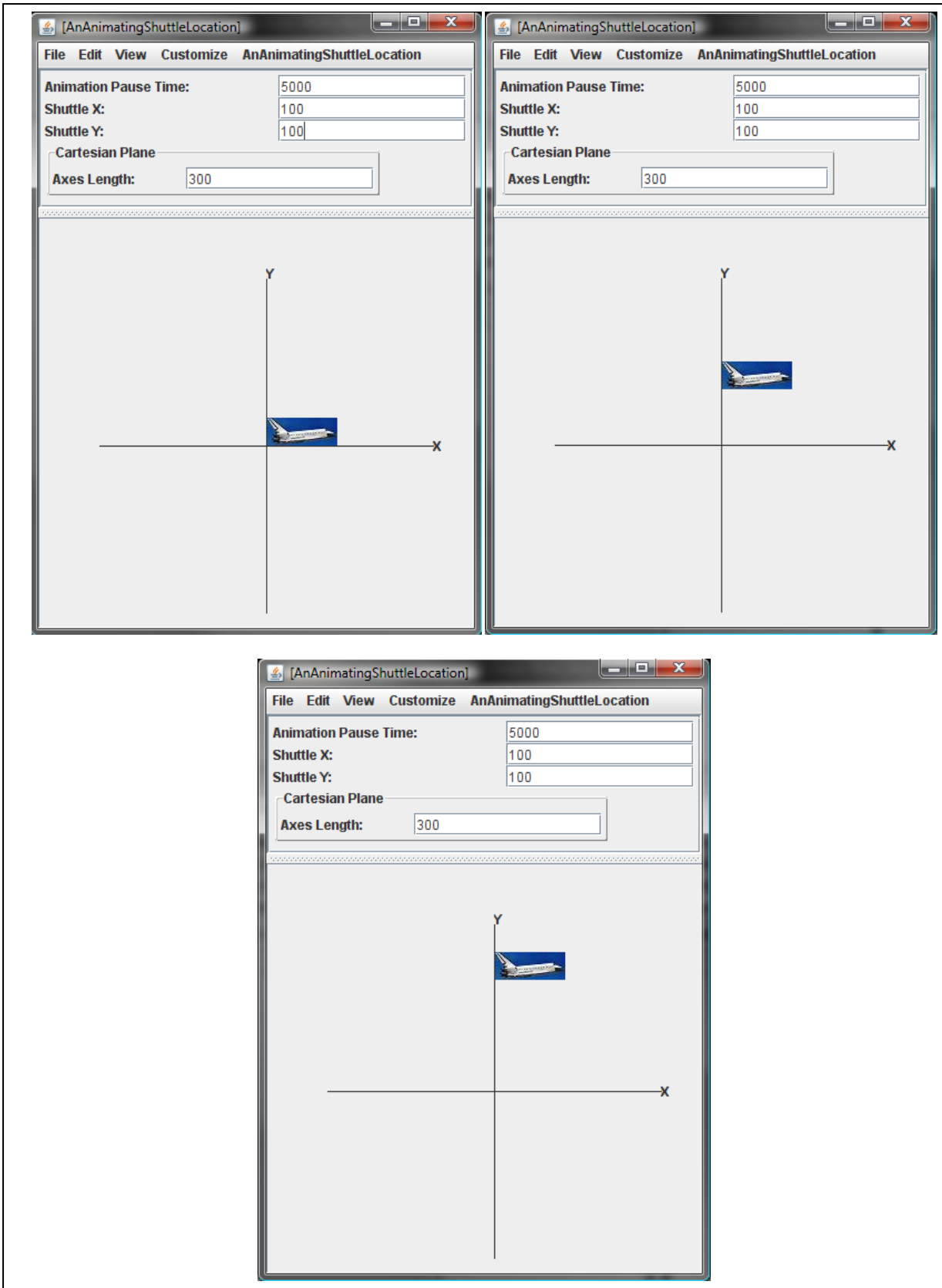

Figure 9. Animating the vertical shuttle motion

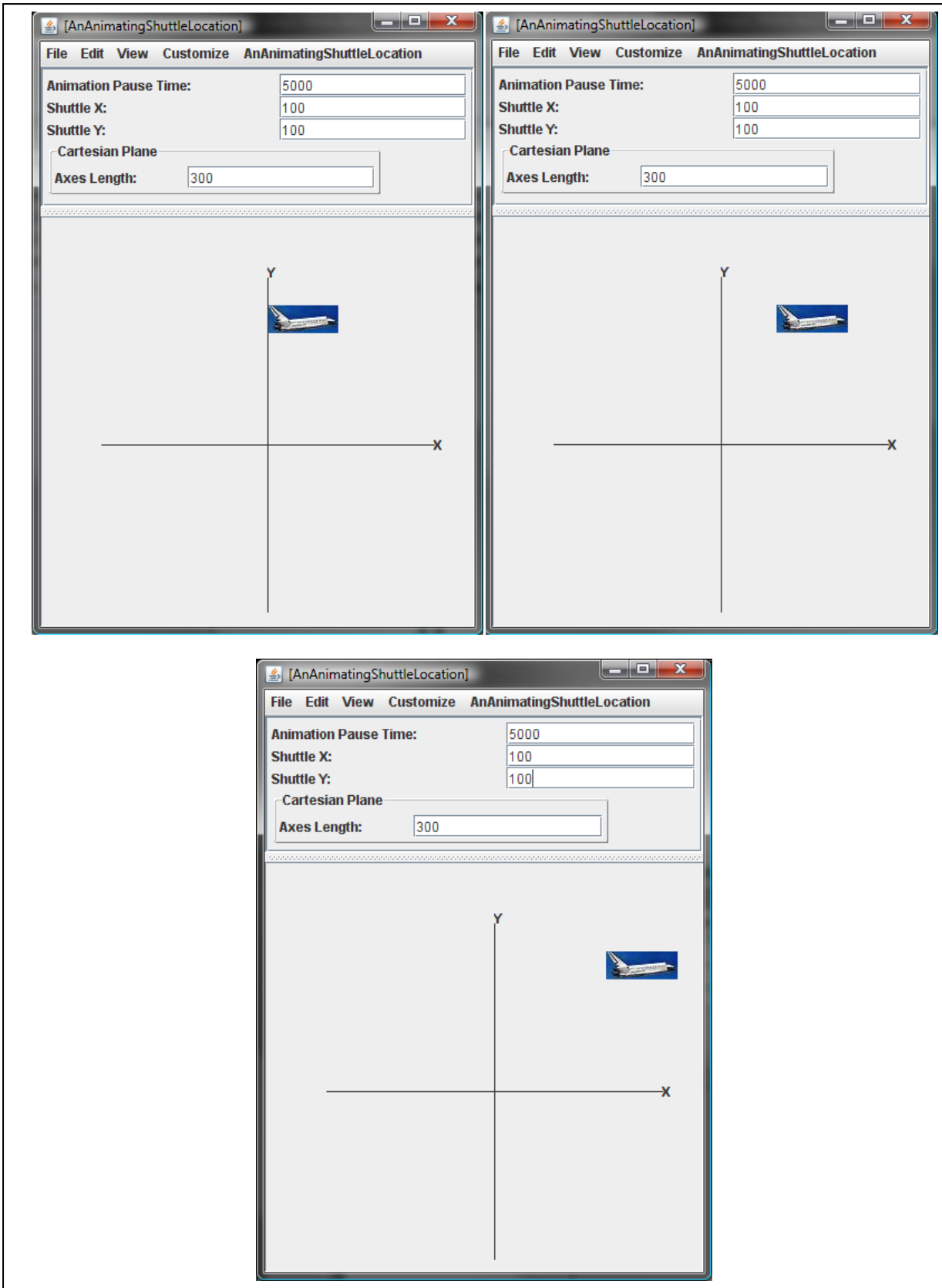

Figure 10. Animating the horizontal shuttle motion

In the remainder of this discussion, we will assume that time is specified in milliseconds. Thus, in Figure 8, the `AnimationPauseTime` specifies a pause time of 5000 milliseconds or 5 seconds. The large pause time gave us enough time to take a screen dump of each intermediate position of the shuttle. For the animation to appear smooth, the pause time should be about 30 milliseconds.

The repetition of the three steps shows the role loops play in solving this problem. However, what we have learnt so far is not sufficient to completely code it. How do we make the program pause, that is, do nothing for some period of time, `pauseTime`? The following is one way to do so:

```
void sleep(int pauseTime) {
      int numberOfAssignments = pauseTime; //ASSIGNMENT_TIME
      for (int i = 0; i < numberOfAssignments; i++) {
            int dummy = 0; // nonsense assignment
}
```

We could repeatedly make an unnecessary assignment in a loop until we have made enough to take time `pauseTime`. This "solution" has two problems. First, we would have to change our code each time we execute it on a computer of different power. Second, and more important, we unnecessarily use the computer executing the loop. It is for this reason that such a loop is called *busy waiting*.

What we really need is an operation that asks the operating system to suspend or put to sleep our program for `pauseTime` so that it can execute some other applications during this time. Java provides such an operation, `Thread.sleep(pauseTime)`, and the following recoding of our `sleep` method shows its use:

```
void sleep(int pauseTime) {
      try {
            Thread.sleep(pauseTime);
      } catch (Exception e) {
            // program may be forcibly interrupted while sleeping
            e.printStackTrace();
      }
};
```

A sleeping method may be woken up not only when its regular alarm goes off, but also because of some unexpected condition such as the user terminating the program. To signal an abnormal waking up, `Thread.sleep(pauseTime)` throws an `InterruptedException`. We have therefore enclosed the call to it in a try-catch block.

We can now code our animation algorithm:

```
public synchronized void animateFromOrigin() {
      int curX = 0;
      int curY = 0;
      setLabelX(windowX(curX));
      setLabelY(windowY(curY));

      while (curY < getShuttleY()) {
```

```
            // loop make sure we don't go past final Y position
            sleep(getAnimationPauseTime());
            curY += ANIMATION_STEP;
            setLabelY(windowY(curY));
        }

        // move to destination Y position
        setLabelY(windowY(getShuttleY()));

        while (curX < getShuttleX()) {
            sleep(getAnimationPauseTime());
            curX += ANIMATION_STEP;
            setLabelX(windowX(curX));
        }

        setLabelX(windowX(getShuttleX()));
    }
```

We have a separate loop for moving in the X and Y direction. Each loop keeps track of the next X/Y position. In each iteration, this position is used to change the X/Y coordinate of the shuttle, and the position is incremented it by the distance ANIMATION_STEP. If the next X/Y position exceeds the final X/Y position, the loop exits, and thus does not change the shuttle coordinate. Therefore, after the loop, the method moves the shuttle to the final position.

Notice the keyword, **`synchronized`**, in the header of `animateFromOrigin`:

```
public synchronized void animateFromOrigin()
```

ObjectEditor requires that every animating method have this keyword in the header. If you do not put it, the animation will not work.

Concurrency and Synchronization*

The reason for making an animating method synchronized is subtle. It has to do with sequencing of actions taken by ObjectEditor and the methods it calls on our behalf.

When we ask ObjectEditor to execute a non-synchronized method, the ObjectEditor waits for the method to finish execution, just as any calling method waits for a called method to finish execution. During this period the user-interface is frozen – the display is not updated, and we cannot use any of the ObjectEditor menus. We have not noticed this so far because our methods returned quickly. The method `animateFromOrigin` is different because of the time-consuming loops it executes. Figure 11 shows what happens if it was not declared to be synchronized.

While `animateFromOrigin` is executing, the menu is frozen and the display is not updated. When the method finishes executing, the display is updated, at which point the shuttle is back at its original position. Thus, the effect of executing the method is a long pause as ObjectEditor suspends its activities for the time it takes to complete the “animation," that is, the time it takes to complete all the sleeps in

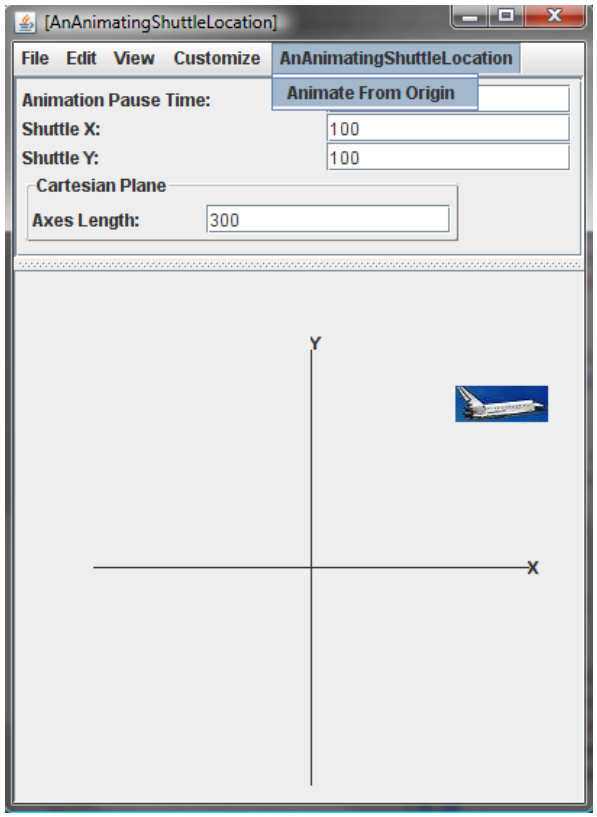

Figure 11. Instance of `AnAnimatingShuttleLocation` when keyword `synchronized` is omitted from the `animateFromOrigin` header

the method. The keyword synchronized in a method declaration tells ObjectEditor to create a new *thread* or activity for executing the method. It is this new thread that waits for the method to execute while the original ObjectEditor thread is free to update the display and process user-commands. Thus, when we make `animateFromOrigin` synchronized, we can make a new call to `animageFromOrigin` while the previous call to it is executing, as shown in Figure 12. As before, ObjectEditor will start a new thread to execute this call. This is displayed in Figure 12 (bottom), which shows Thread-3 and Thread-4 executing the two calls to `animateFromOrigin`. As we see, each thread is associated with its own stack of calls, with a calling method waiting for a called method to finish. The call at the top of both stacks is `animateFromOrigin`, the ones below are calls to ObjectEditor code, about which we don't have to worry. Figure 12 (bottom) shows that the two calls are executing at different locations of `animateFromOrigin`. The next statement to be executed by Thread 3 is the sleep call in the first loop, while the next statement to be executed by Thread 4 is the very first statement of the method.

As the thread numberings imply, these are not the only threads in the system. One of the other two threads is the ObjectEditor thread that processes user commands and updates the display. The others thread(s) are system threads that do "garbage collection," that is, gets rid of object we no longer need and other book-keeping/clean-up activities.

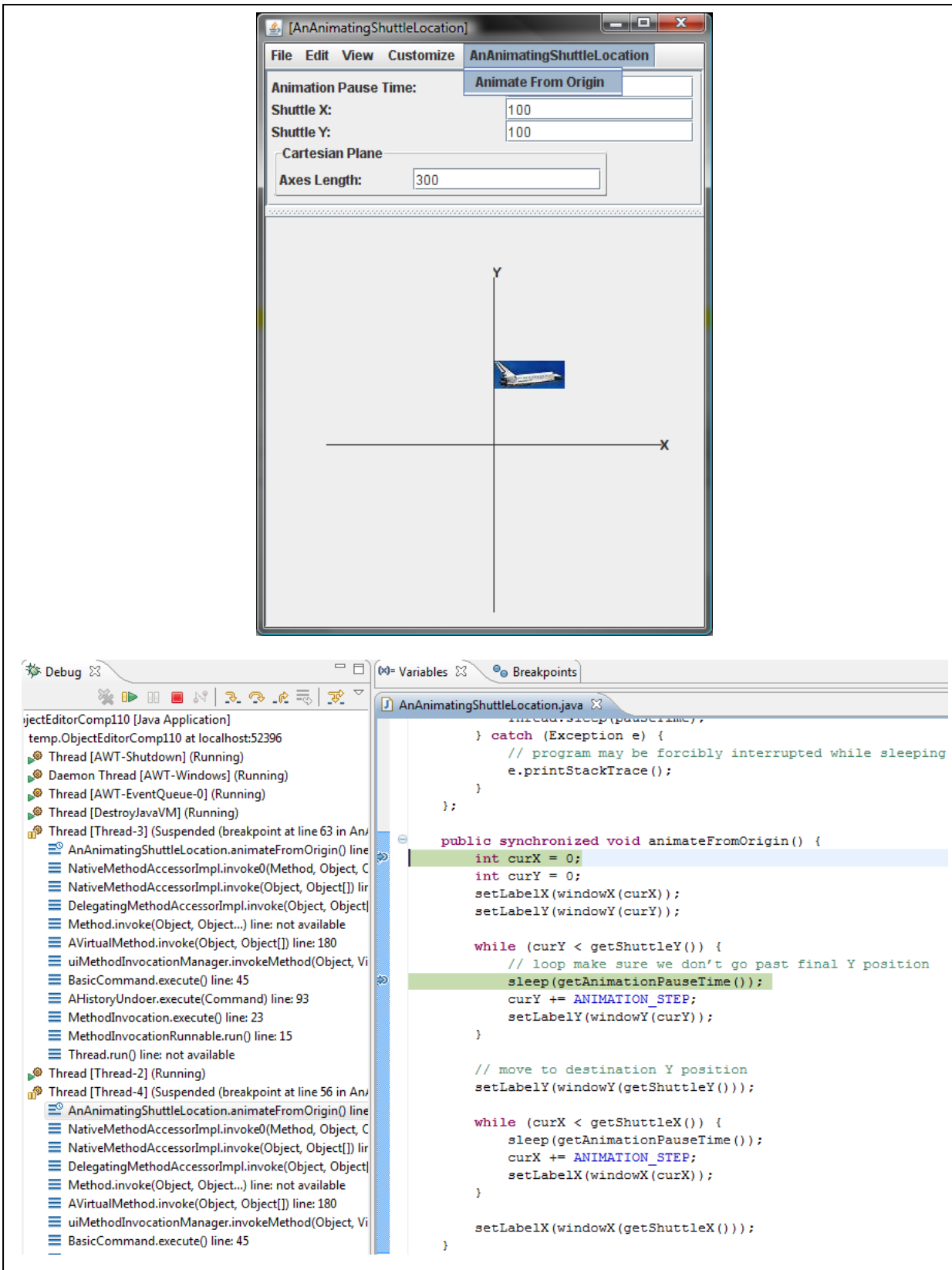

Figure 12. Making a call to `animateFromOrigin` before the previous one has finished

When two calls to a method are active concurrently, they can step on each other's toes, leaving the instance variables they share in an inconsistent state. Figure 12 shows the problem that can occur. If Thread 4 is allowed to proceed, it can reset the shuttle location, thereby interfering with the animation of the first method! As it turns out, this problem will not actually occur because `animateFromOrigin` is declared to be **`synchronized`**. This keyword tells Java that this method should be executed serially, that is, only one thread should execute it at any one time. Thus, Thread-4 will wait at the first statement of `animatreFromOrigin` until Thread-3 has finished executing the method.

Thus, the keyword **`synchronized`** in the header of a method tells:

- ObjectEditor that a new thread should be created for executing the method.
- Java that only one thread should execute the method at one time.

Thus, the keyword both increases and reduces the concurrency in the system. It increases concurrency by allowing method to be executed concurrently with the `paint` method in ObjectEditor, which displays objects in edit windows. It reduces concurrency by making sure that the method is not executed concurrently by two threads.

Even if we were not using ObjectEditor, we would need to make an animating method synchronized. The reason, as mentioned above, is that user-interface and animation operations must interleave execution, which requires multiple threads. If we were not using ObjectEditor, we would ourselves create a special thread for executing an animating method. We would still need to make the method synchronized to ensure that concurrent executions of the animating method by different threads are serialized so that they don't interfere with each other.

As it turns out, this keyword cannot be used in a method declared in an interface. Thus, in the interface of the class, we must declare the header of `animateFromOrigin` as:

```
public void animateFromOrigin()
```

even though, in the implementation of the interface, we declare it as:

```
public synchronized void animateFromOrigin()
```

Normally, Java requires matching of all components of corresponding method headers in interfaces and the classes implementing it, but not in this case, probably to give the implementers of an interface the flexibility of deciding if they want to allow for concurrency and pay the cost of synchronization.

Incremental Display Update

As mentioned above, normally ObjectEditor updates the display only at the execution of each method. In the case of an animating method, ObjectEditor should update the display after each animation step. On the other hand, it does not know when an animation step has completed. Therefore, after each animation step, the method should explicitly tell ObjectEditor that the state displayed has changed. In other words, the object containing the animation method should behave as an observable that allows

ObjectEditor and other observers to be registered and informs them whenever the animated state changes. As we saw earlier, there are multiple ways in which observable/observer can be defined, which could be classified into application-specific and application-independent approaches. As ObjectEditor is an observable of arbitrary objects, we need to use an application-independent approach. As the state in which it is interested consists of JavaBeans properties, it makes sense to use the standard approach for supporting property observers or listeners.

ObjectEditor implements the standard `java.beans.PropertyChangeListener` interface:

```
public interface java.beans.PropertyChangeListener {
      public void propertyChange(PropertyChangeEvent arg)
}
```

If the class of a displayed object defines the standard method defined by JavaBeans for registering property listeners:

```
public void addPropertyChangeListener(PropertyChangeListener l)
```

ObjectEditor calls the method to register itself as an observer. It is the responsibility of the method to store references to ObjectEditor and other listeners:

```
PropertyChangeListenerHistory observers =
      new APropertyChangeListenerHistory();
public void addPropertyChangeListener(PropertyChangeListener l) {
      observers.addElement(l);
}
```

Here `APropertyChangeListenerHistory` is simply a history of objects of type `PropertyChangeListener`, providing the standard history operations: `addElement`, `size`, and `elementAt`.

A method that changes a property should notify all of the stored observers about the changed property. For example, the `setX` method in (a non-immutable version of) `ACartesianPoint` can notify all of its observers of the change to the X property:

```
public void setX(int newVal) {
      int oldVal = x;
      x = newVal;
      notifyAllListeners(new PropertyChangeEvent(
            this, "x", oldVal, newVal);
}
```

where `notifyAllListeners` is defined as follows:

```
public void notifyAllListeners(PropertyChangeEvent e) {
       for (int index = 0; index < observers.size(); index++) {
            observers.elementAt(i).propertyChange(e);
}
```

`notifyAllListeners` calls the `propertyChange` method in each observer, passing it the change event. The implementation of this method in ObjectEditor updates the display:

```
public class ObjectEditor implements java.beans.PropertyChangeListener
{
      public void propertyChange(PropertyChangeEvent arg) {
      // update display of property arg.getPropertyName()
      // to show arg.getNewValue().
            …
      }
}
```

Let us use these ideas to complete the implementation of `AnAnimatingShuttleHistory`. Consider how we can animates the X location – animation of the Y location is similar.

The X location is changed by the statement:

```
setLabelX(windowX(curX)); // need to update display
```

in the first loop. As the statement does not directly change the shuttle location, we need to look at the implementation of `setLabelX`:

```
void setLabelX(int x) {
      Point oldLocation = shuttleLabel.getLocation();
      Point newLocation = new ACartesianPoint(x, oldLocation.getY());
      shuttleLabel.setLocation(newLocation);
}
```

Again, the method does not directly change the shuttle location. Therefore, we need to look at the implementation of `setLocation` in class `ALabel`:

```
public void setLocation(Point newVal) {
      location = newVal;
}
```

This is the method in which the shuttle location changes. Therefore, we need to make its class, `ALabel`, an observable in the fashion described above. We can add the `addPropertyChangeListener` and `notifyAllListeners` implementations given above without any changes to the class, as these are standard and do not depend on the property being changed. Now we can call `notifyAllListeners` from `setLocation`:

```
public void setLocation(Point newVal) {
      Point oldVal = location;
      location = newVal;
      notifyAllListeners(new PropertyChangeEvent(
            this, "Location", oldVal, newVal);
}
```

Here, `setLocation` assigns the instance variable, `location`, a new instance of `ACartesianPoint`. Suppose that `Point` was not immutable, that is, it provided setter methods for the X and Y properties. In this case, if `setLocation` simply changed the X and Y coordinate of the existing location, we would need to change the `setX` and `setY` methods of `ACartesianPoint` class in the manner described above.

Summarizing Animation

In general, an animating method performs one or more animation steps, where an animation step changes one or more animating (graphical) properties such as size, location, icon of one or more graphical objects and then pauses execution for some time. Execution can be paused using busy waiting or a sleep call provided by the operating system. Busy waiting has the problem that it is platform-specific and does not allow other activity to proceed while the animation is paused. Therefore using the sleep call is preferable.

After each animation step, all displays of the animation must be updated. The observable-observer concept can be used to ensure these updates are made. This means that we must ensure that for each graphical property changed by the animation, the class of the property allows observers to be registered and the setter of the property informs the observers about the update. ObjectEditor requires the JaveBeans observer-observer approach based around the PropertyChangeListener interface.

An animating method should be executed in a separate thread as otherwise the user-interface thread will wait for it to finish execution before performing any screen update. This means that it is possible to start multiple executions of the method concurrently. We should use the keyword **`synchronized`** in the declaration of the method to ensure that it is executed serially by the thread, that is, to ensure that if a thread is in the middle of executing the method, other threads wait for it to finish.

The keyword **`synchronized`** also tells ObjectEditor to start a new thread to execute the method.

Thus, an animating method should be declared as **`synchronized`**.

In general, a method that performs the animation steps and a method that changes the value of some animating property may be in different classes such as `AnAnimatingShuttleLocation` and `ALabel`.

Summary

- A loop executes its body a variable number of times, which is determined by its condition.
- A particularly useful application of a loop is to fold a series of values into a cumulative result.
- A loop can be classified into a counter-controlled or an event-controlled loop depending on whether the number of iterations executed by it is determined before it is executed.
- If you are having trouble writing a loop for a problem that has a variable size, it is sometimes useful to solve a fixed-size version of the problem, find a repeated pattern of statements, and make this pattern the loop body.
- It is important to check the correctness of the loop for boundary conditions, avoiding common off-by-one errors.

- It is also important to avoid infinite loops by making sure that the condition for continuing the loop converges to a false value.
- In a counter-controlled loop, we must also guard against receiving a counter limit of the unexpected sign.
- Instead of nesting a loop directly within another loop, it is best is to define a separate method for the loop, which makes the program less error-prone and more reusable.
- The break statement allows the loop termination condition to be tested in the middle of a loop iteration, thereby allowing us to write a single statement for receiving a loop event.

Exercises

1) What is the output of the following loops, when n is (i) 4 and (ii) –4.

   a.

```
while (n >= 0)  {
      n = n -1;
      System.out.println (n);
}
```

   b.

```
while (n != 0)  {
      n = n -1;
      System.out.println (n);
}
```

2) Write a program that prints all positive odd numbers less than some limit n.

3) All of the solutions to the problem of computing n factorial, performed the multiplication:

```
1*1*2*… n
```

   Write a solution to avoid the extra multiplication by 1. Be sure to check the boundary conditions.

4) What is the output of the following program fragments assuming the user enters the four lines:

```
hello
hello
goodbye
goodbye
```

   a.

```
String inputLine = Console.readString();
while (inputLine.equals("hello"))
      System.out.println ( " ca va");
```

   b.

```
String inputLine = Console.readString().
while (inputLine.equals("hello")){
      System.out.println (inputLine + "/ca va");
inputLine = Console.readString();
}
```

   c.

```
while (Console.readString().equals("hello"))
      System.out.println ( "ca va");
```

d.

```
while (Console.readString().equals("hello"))
      System.out.println ("ca va");
```

e.

```
while (Console.readString().equals("goodbye"))
      System.out.println ("Au revoir");
```

f.

```
while (Console.readString().equals("hello"))
      System.out.println ("Ca va");
while (Console.readString().equals("goodbye"))
      System.out.println ("Au revoir");
```

g.

```
while (Console.readString().equals("hello"))
      ; // null statement (do nothing)
while (Console.readString().equals("goodbye"))
      System.out.println ("Au revoir");
```

h.

```
String inputLine = Console.readString().
while (inputLine.equals("hello"))
      inputLine = Console.readString();
while (inputLine.equals("goodbye"))  {
      System.out.println ("Au revoir");
      inputLine = Console.readString();
}
```

5) Write a program that allows a user to enter a list of strings

$s_1, s_2, \ldots, s_n$

and outputs a single string

$s_1 + s_2 \ldots + s_n,$

Each element of the string list is entered on a separate line and the end of the list is entered by a special string, "end". Thus, if the user enters

```
hello
world
end
```

the program should output:

```
helloworld
```

To check that two strings, s_1 and s_2, have the same sequence of characters, you should call the equals method

```
s1.equals(s2)
```

rather than check for equality

```
s1== s2
```

6) Extend your solution to problem 4 so that it processes a list of lists terminated by the special string, “quit”. For example, if the user enters:

```
hello
world
end
goodbye
world
end
quit
```

the program should output:

```
helloworld
goodbyeworld
```