while and for loops

Looping in time

Friday, November 2, 2007

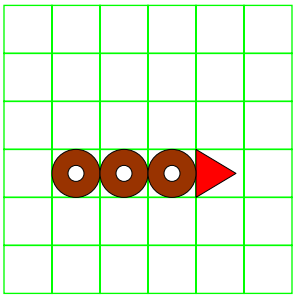
CS111 Computer Programming

Department of Computer Science
Wellesley College

bagelForward(n)

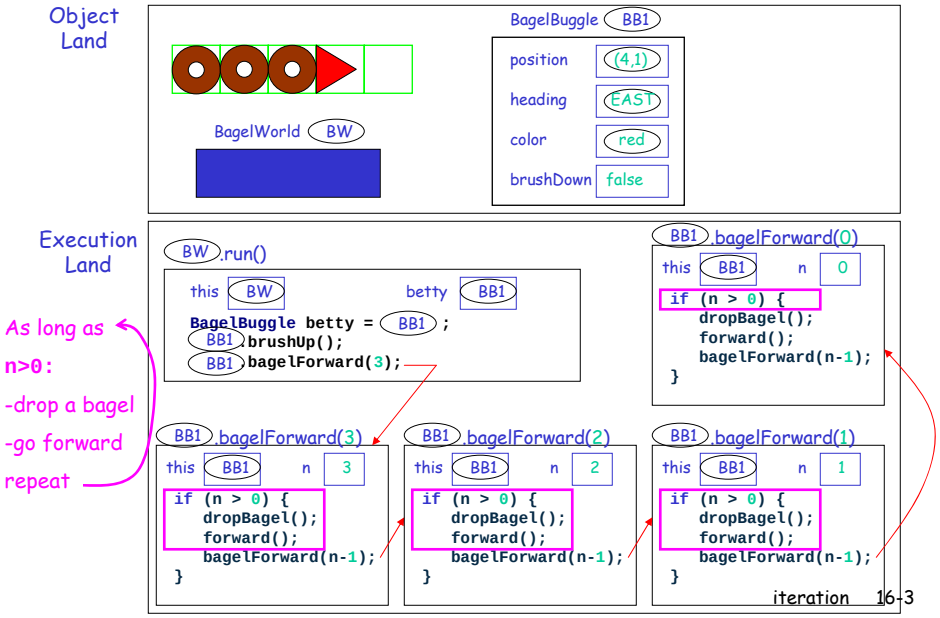
Write a method that teaches a buggle to leave behind a trail of bagels of a given length. Assume brushUp().

```
public void bagelForward(int n) {

     if (n > 0) {
          dropBagel();
          forward();
          bagelForward(n-1);
     }
}
```



bagelForward(3);

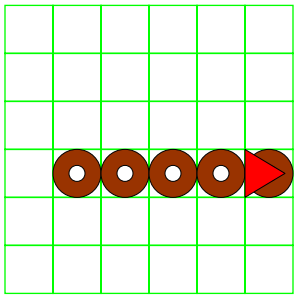
JEM illustrating `bagelForward(3)`



`bagelsToWall()`

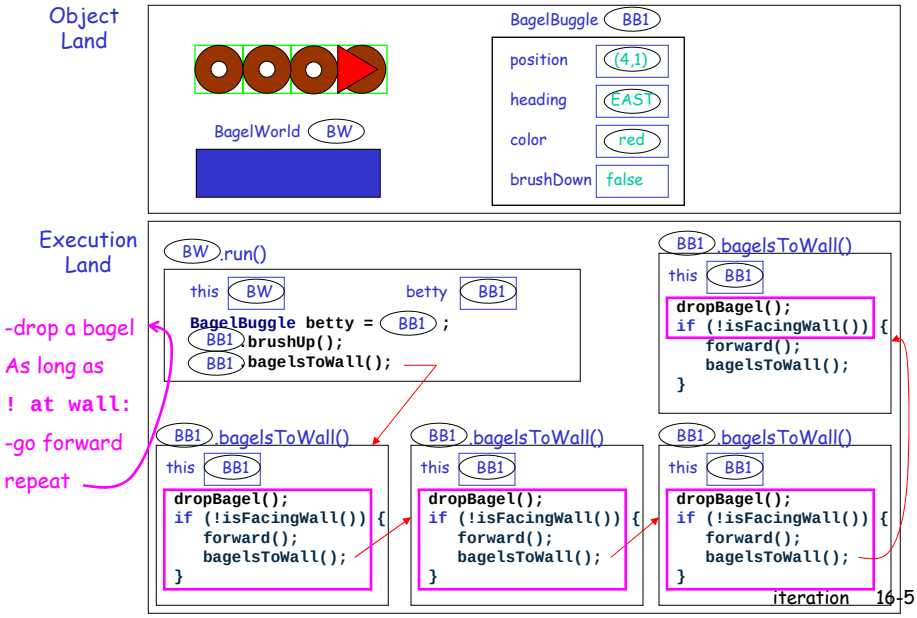
Write a method that teaches a buggle to drop a line of bagels from the buggle's current position all the way up to the wall. Assume brushUp().

```
public void bagelsToWall() {

     dropBagel();
     if (!isFacingWall()) {
          forward();
          bagelsToWall();
     }
}
```



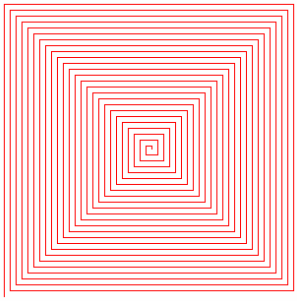
`bagelsToWall();`

JEM illustrating `bagelsToWall()`



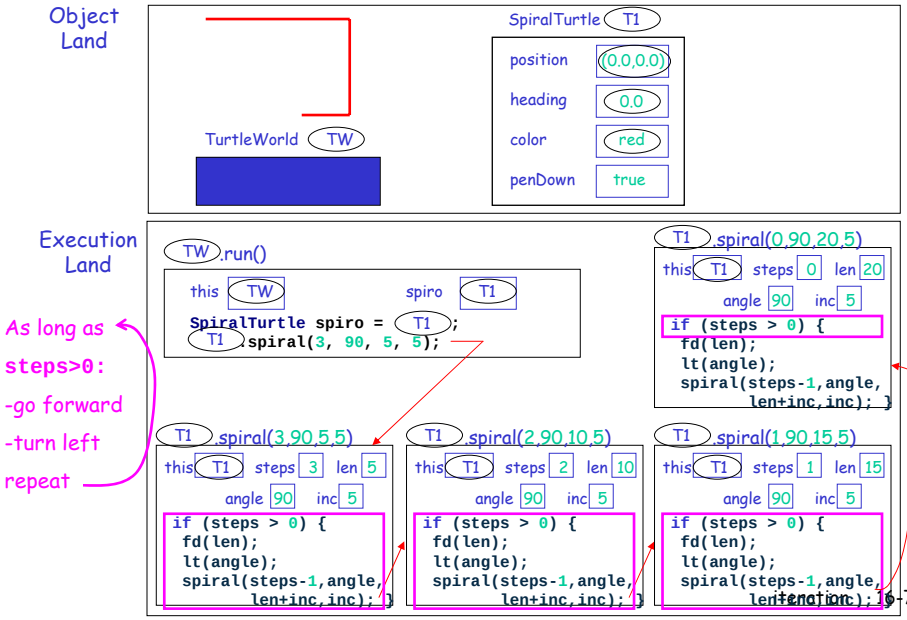
`spiral(steps,angle,len,inc)`

```
public void spiral(int steps,
                   double angle,
                   double len,
                   double inc) {

     if (steps > 0) {
          fd(len);
          lt(angle);
          spiral(steps-1, angle,
                 len_inc, inc);
     }
}
```



`spiral(100,90,0,3);`

JEM illustrating `spiral(3,90,5,5)`



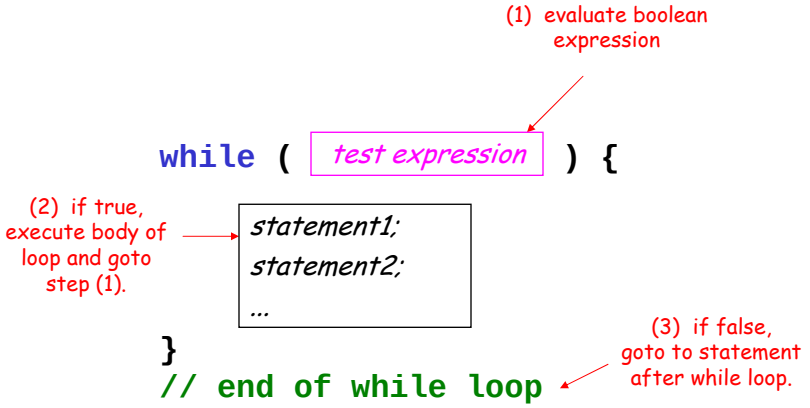
iteration 16-7

What do `bagelForward()`, `bagelsToWall()`, and `spiral()` have in common?

- They are tail recursive!
- Recall: A method is **tail recursive** if it has one recursive subproblem and no pending operations to be performed after the recursive invocation.
- Effectively, tail recursive methods describe an iterative process. A process is **iterative** if it can be expressed as a sequence of steps that is repeated until some stopping condition is reached.
- Iteration is such a common aspect of problem solving, that many computer programming languages provide special constructs for describing iterative processes.

iteration 16-8

While construct*



* Reads: while boolean expression is true, execute body of loop.

Variable declaration and assignment

```
int sum;
String s;
Buggle becky;
```

variable declarations

```
sum = 0;
s = "non ministari";
becky = new Buggle();
sum = 5;
sum = sum + 5;
sum = sum + sum + sum;
s = s + " sed ministrare";
```

variable assignment

```
int n = 17;
String name = "ada";
Buggle bernice = becky;
```

variable declarations and assignments

parameter assignment

```
public double fahrenheitToCelsius(double temp) {
    temp = temp - 32.0;
    temp = 5.0 * temp / 9.0;
    return temp;
}
```

How to exit from a while loop?

```
public void countDown (int n) {
     while (n > 0) {
          System.out.println(n);
          n = n - 1;
     }
     System.out.println("blastoff!");
}
```

countDown(5) ⟶

5
4
3
2
1
blastoff!

Houston, we have a problem

```
public int countDown (int n) {
     while (n > 0) {
          System.out.println(n);
     }
     System.out.println("blastoff!");
}
```

countDown(5) ⟶

5
5
5
5
5
5
5
5
5
5
5

Factorial

```
public static int fact (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fact (n - 1);
    }
}
```

to calculate n!, we invoke `fact(n)`

```
public static int factTail (int num, int ans) {
    if (num == 0) {
        return ans;
    } else {
        return factTail(num - 1, num * ans);
    }
}
```

to calculate n!, we invoke `factIter(n)`

```
public static int factIter(int n) {
  return factTail(n,1);
}
```


Iteration table for `factTail()`

```
public static int factTail (int num, int ans) {
     if (num == 0) {
          return ans;
     } else {
          return factTail(num-1, num * ans);
     }
}
```

Iteration table shows the values of variables for each invocation

Invocation	num	ans
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

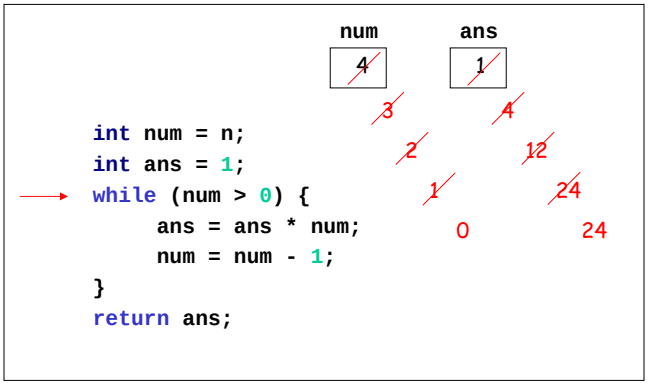
Factorial with a `while` loop

```
public static int factWhile (int n) {
     // declaration of local state variables
     int num = n;
     int ans = 1;

     while (num > 0) {
          ans = ans * num;    // calculate product
          num = num - 1;      // and decrement num
     }
     return ans;
}
```

Note: order of assignments matters!

Execution Land

factWhile(4)



Iteration table for `factWhile()`

```
public static int factWhile (int n) {
     int num = n;
     int ans = 1;
     while (num > 0) {
          ans = ans * num;
          num = num - 1;
     }
     return ans;
}
```

Exactly the same table as `factTail()`

Loop #	num	ans
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

`factTail()` and `factWhile()`

```
public static int factIter(int n) {
  return factTail(n, 1);
}

public static int factTail (int num, int ans) {
  if (num == 0) {
    return ans;
  } else {
    return factTail(num-1, num*ans);
  }
}
```

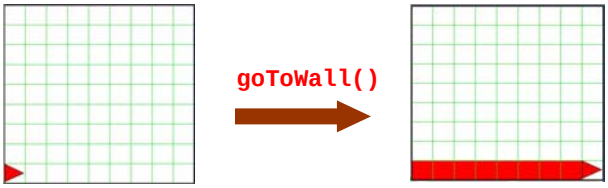
Initialize variables

When done, return ans

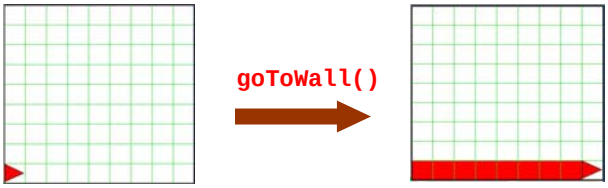
```
public static int factWhile (int n) {
  int num = n;
  int ans = 1;
  while (num > 0) {
    ans = ans * num;
    num = num - 1;
  }
  return ans;
}
```

While not done, update variables

Let's try that again



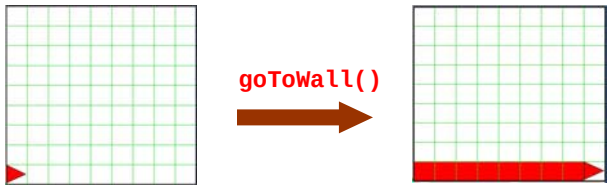
Tail recursive solution



Tail recursion

```
public void goToWallRec() {
     if (!isFacingWall()) {
          forward();
          goToWallRec();
     }
}
```

The corresponding looping solution



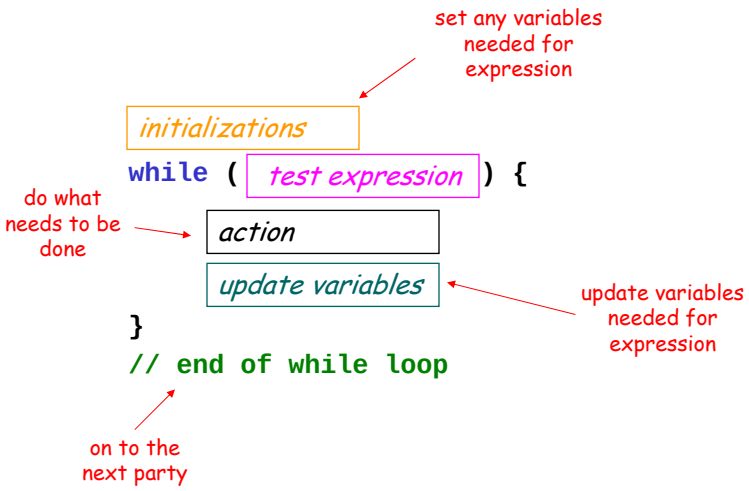
Tail recursion

```
public void goToWallRec() {
     if (!isFacingWall()) {
          forward();
          goToWallRec();
     }
}
```

Iterative

```
public void goToWallWhile() {
     while (!isFacingWall()) {
          forward();
     }
}
```


Typical `while` loop



Factorial with a `while` loop

```
public static int factWhile (int n) {
     // declaration of local state variable
     int num = n;
     int ans = 1;
     while (num > 0) {
          ans = ans * num;   // calculate product
          num = num - 1;     // and decrement num
     }
     return ans;
}
```


`factFor()`

```
public static int factWhile (int n) {
     int num = n;
     int ans = 1;
     while ( num > 0 ) {
          ans = ans * num;
          num = num - 1;
     }
     return ans;
}
```

Iteration table for factFor(4)

Loop #	num	ans
1	4	1
2	3	4
3	2	12
4	1	24
5	0	24

```
public static int factFor (int n) {
     int ans = 1;
     for ( int num = n ; num > 0; num = num - 1 ) {
          ans = ans * num;
     }
     return ans;
}
```

for loop syntax

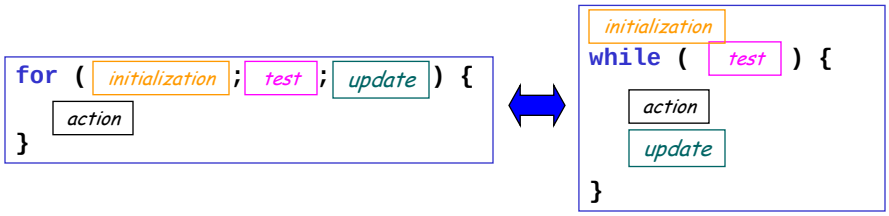
a.k.a.
continue condition

```
for ( initialization of loop counter ; test expression ; update loop counter ) {

    action

}
// end of for loop
```

As long as the continue condition holds, execute body of for loop

As soon as boolean_expr is false drop down to here

Conversion from one loop to another



Variations on a theme*

```
public static int factForUp (int n) {
     int ans = 1;
     for (int i = 1;  i <= n;  i = i + 1) {
          ans = ans * i;
     }
     return ans;
}
```

Iteration table for factForUp(4)

Loop #	i	ans
1	1	1
2	2	2
3	3	6
4	4	24
5	5	24

*For those who prefer counting up.

Remember me

```
public void bagelForward(int n) {

     if (n > 0) {
          dropBagel();
          forward();
          bagelForward(n-1);
     }
}
```

What would loop versions look like?

```
public void bagelForward(int n) {
     if (n > 0) {
          dropBagel();
          forward();
          bagelForward(n-1);
     }
}
```

```
public void bagelForwardWhile(int n) {
     while (n > 0) {
          dropBagel();
          forward();
          n = n-1;
     }
}
```

```
public void bagelForwardFor(int n) {
   for (         ; n > 0 ; n=n-1)  {
        dropBagel();
        forward();
   }
}
```


Or, if you prefer

```
public void bagelForwardFor(int n) {
     for (         ; n > 0 ; n=n-1)  {
          dropBagel();
          forward();
     }
}
```

```
public void bagelForwardFor(int n) {
     for (int i = n; i > 0; i = i-1)  {
          dropBagel();
          forward();
     }
}
```

As an aside…

Incrementing and decrementing are common operations.

The following statements are equivalent:

```
h = h + 1;
h += 1;
h++;
++h;
```

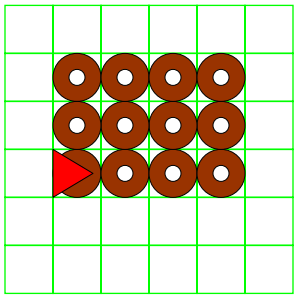
The following statements are equivalent:

```
h = h - 1;
h -= 1;
h--;
--h;
```


bagelRectangle(int width, int height)

Write a method that enables a buggle to draw a rectangle of bagels of width, w, and height, h, and return to the starting position. Assume brushUp().

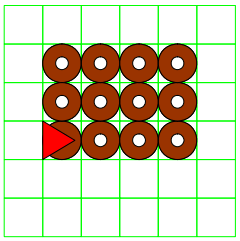
```
bagelRectangle(4,3);
```



Recycling code

```
public void bagelRectangle(int width, int height) {
     for (int h = 1; h <= height; h++)  {
          bagelForwardFor(width);
          backward(width);

          // position buggle for next line
          left();
          forward();
          right();
     }

     // return buggle to starting location
     left();
     backward(height);
     right();
}

public void bagelForwardFor(int n) {
     for (int i = n; i > 0; i--)  {
          dropBagel();
          forward();
     }
}
```

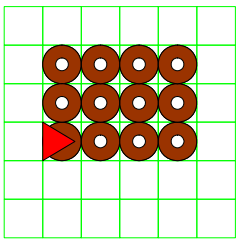
bagelRectangle(4,3);



Nested loops

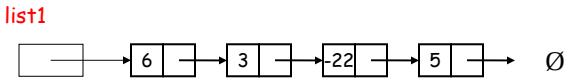
```
public void bagelRectangle(int width, int height) {
     for (int h = 1; h <= height; h++)  {

          // bagelForwardFor(width);
          for (int i = width; i > 0; i--)  {
               dropBagel();
               forward();
          }
          backward(width);

          // position buggle for next line
          left();
          forward();
          right();
     }
     // return buggle to starting location
     left();
     backward(height);
     right();
}
```

bagelRectangle(4,3);



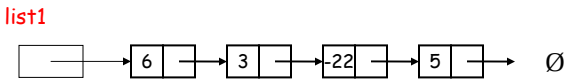
Iteration with recursive data structures

```
// Returns the integer sum of all elements in L
public static int sumList (IntList L) {
     int result = 0;
     if (!IntList.isEmpty(L))  {
          result = IntList.head(L) + sumList(IntList.tail(L));
     }
     return result;
}
```



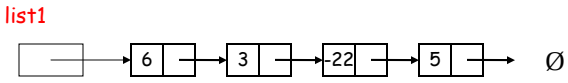
Tail recursive version of sumList

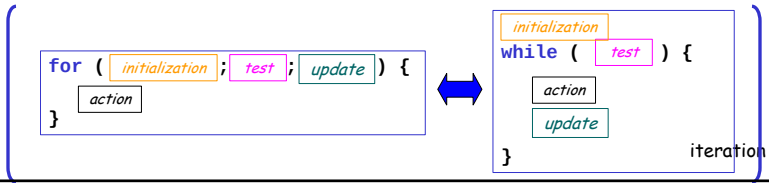
```
public static int sumListIter (IntList L) {
  return sumListTail(L, 0);
}

// Returns the integer sum of all elements in L
public static int sumListTail (IntList L, int result) {
  if (IntList.isEmpty(L)) {
    return result;
  } else {
    return sumListTail(IntList.tail(L),IntList.head(L)+result);
  }
}
```



sumListWhile()

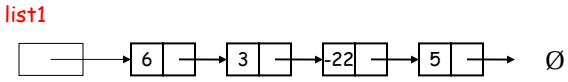
```
// Returns the integer sum of all elements in L
public static int sumListWhile (IntList L) {
     int result = 0;
     while (!IntList.isEmpty(L))  {
          result = IntList.head(L) + result;  // action
          L = IntList.tail(L);                // update
     }
     return result;
}
```





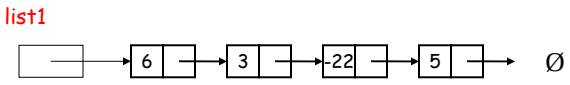
sumListFor()

```
// Returns the integer sum of all element in L
public static int sumListFor (IntList L) {
  int result = 0;
  for (          ; !IntList.isEmpty(L); L=IntList.tail(L)) {
     result = IntList.head(L) + result;
  }
  return result;
}
```



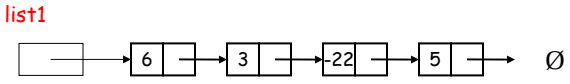
Any statement can go in the first part of the for loop

```
// Returns the integer sum of all element in L
public static int sumListFor (IntList L) {
  int result;
  for (result=0; !IntList.isEmpty(L); L=IntList.tail(L)) {
     result = IntList.head(L) + result;
  }
  return result;
}
```



Variable scope

```
// Returns the integer sum of all element in L
public static int sumListFor (IntList L) {

  for (int result=0; !IntList.isEmpty(L); L=IntList.tail(L)) {
     result = IntList.head(L) + result;
  }
  return result;
}
```



There is a problem! The variable result only exists within the for loop. We cannot refer to the variable after the for loop, i.e., in the return statement.