CS100J March 15, 2005
For Loops
Reading: Secs 7.5-7.6

Quote for the Day:
Perhaps the most valuable result of all education is the ability to make yourself do the thing you have to do, when it ought to be done, whether you like it or not; it is the first lesson that ought to be learned; and however early a man's training begins, it is probably the last lesson that he learns thoroughly.

Thomas Henry Huxley
1825 - 1895
Technical Education, 1877

Find the quotient q and remainder y when x is divided by y

(We did this last lecture)

```
// pre: x ≥ 0  and  y > 0
int r= x;
int q= 0;
// invariant:  x = q*y + r   and   0 ≤ r
while (r >= y) {
    q= q + 1;
    r= r – y;
}
// post:  x = q*y + r   and   0 ≤ r < y
```

Notes:
1. the postcondition defines the properties of quotient q and remainder r.

2. The invariant was found from the postcondition by seeing that all by r < y was easily truthified by setting r to x and q to 0 —but relation r < y then need not be true (so it was deleted).

3. We didn't make progress with r= r–1; because we can't then fix the invariant.

4. The following relation helps us see that the invariant is kept true:

q*y + r = (q+1)*y + (r–y)

Linear algorithm to compute a**b

(We did this last lecture)

```
// pre: b ≥ 0
double z= 1;
int k= 0;
// invariant:  z = a**k
while (k != b) {
    z= z * a;
    k= k+1;
}
// post:  z = a**b
```

1. The loop process the range of integers 0..b–1. For each one, it multiplies z by a.

2. Execution of the loop requires b iterations. This can be a huge number. To compute 2**32767 requires 32767 iterations!

On the next slide, we show a different algorithm that requires much fewer iterations.

"logarithmic" algorithm to compute a**b

(We did this last lecture)

```
// pre: b ≥ 0
double z= 1;
double x= a;
int y= b;
// invariant:  z*x**y = a**b
//                and  y ≥ 0
while (y != 0) {
   if  (y % 2 == 0)
          {x= x*x; y= y/2;}
   else {z= z*x; y= y–1;}
}
// post:  z = a**b
```

1. In the case y is even, we see that the invariant is kept true because of this formula:
x**y = (x*x)**(y/2)

2. In the case y is odd, we see that the invariant is kept true because of this formula:
z*x**y = z*x*x**(y–1)

3. The loop processes the bits of the binary representation of y. Each bit is processed at most twice: Example: calculate a**7

7 is 111 in binary: 111 → 110 → 11 → 10 → 1 → 0

32767 = 2**15 – 1 in binary is 111111111111111 (15 1's)

So, calculating 1.000001**32767 needs only 30 iterations of the loop! The algorithm on the previous slide requires 32767 iterations.

The for-loop

```
// precondition: n >= 0
int k= 1;
x= 0;
// inv: x = sum of 1..(k-1)
while (k != n) {
    x= x + k;
    k= k+1;
}
// postcondition: x = sum of 1..(n-1)
```

k is a **loop counter**. Initialized before the loop and changed only in the last statement of the repetend of the while-loop

The for-loop: an abbreviation of a while loop with a loop counter

```
// pre: n >= 0
int k= 1;
x= 0;
// inv: x = sum of 1..(k-1)
while (k != n) {
    x= x + k;
    k= k+1;
}
// post: x = sum of 1..(n-1)
```

```
// pre: n >= 0
x= 0;
// inv: x = sum of 1..(k-1)
for (int k = 1;  k != n;  k= k+1) {
    x= x + k;
}
// post: x = sum of 1..n-1
```

k is a **loop counter**. Initialized before loop; changed only in last statement of the repetend of the while-loop

for-loop collects, within the parentheses following **for**, everything to do with initialization of loop counter, stopping, and making progress.

Syntax and semantics of the for loop

```
for ( initialization ; condition ; progress ) {
      repetend
}
```

meaning of for-loop:

```
initialization
while ( condition ) {
      repetend
      progress
}
```

```
x= 0;
// inv: x = sum of 1..(k-1)
for (int k = 1;  k != 20;  k= k+1) {
    x= x + k;
}
// x = sum of 1..19
```

k is a **loop counter**. Initialized before the loop and not changed in the repetend of the for-loop.

Find the quotient q and remainder y when x is divided by y

```
// pre: x ≥ 0  and  y > 0
int r= x;
int q= 0;
// invariant:  x = q*y + r  and   0 ≤ r
while (r >= y) {
    q= q + 1;
    r= r – y;
}
// post:  x = q*y + r   and   0 ≤ r < y
```

```
// pre: x ≥ 0  and  y > 0
int q= 0;
// invariant:  x = q*y + r   and   0 ≤ r
for (int r= x;   r >= y;   r= r – y) {
    q= q + 1;
}
// post:  x = q*y + r   and   0 ≤ r < y
```

r is a **loop counter**. Initialized before the loop and changed only in the last statement of the repetend of the while-loop

Scope of the counter of a for-loop

```
x= 0;
int k= 1;
// inv: x = sum of 1..(k-1)
while (k != 20) {
    x= x + k;
    k= k + 1;
}
// x = sum of 1..19
System.out.println(k);
```

```
x= 0;
// inv: x = sum of 1..(k-1)
for ( int k = 1;  k != 20;  k= k+1) {
    x= x + k;
}
// x = sum of 1..19
System.out.println(k);
```

Illegal: Scope of k is only the loop.

A for-loop schema to process a range of integers

```
// Process m..n
// inv: m..k-1 has been processed
        and m <= k <= n+1
for (int k= m;  k <= n;  k= k+1) {
        Process k;
}
// Post: m..n has been processed
```

Use the schema to count number of ‘e’s in a string

```
// Process m..n
// inv: m..k-1 processed, m <= k <= n+1
for (int k= m; k <= n; k= k+1) {
        Process k;
}  // Post: m..n processed
```

```
// Store in x the no. of ‘e’s in s[0..s.length()-1]
// inv: m..k-1 processed, m <= k <= n+1
        x = no.of ‘e’s in s[0..k-1]
for (int k= m; k <= n; k= k+1) {
        Process k;
} // Post: m..n processed
```

Example:
s = “sequioa”, x = 1
s = “defense”, x = 2

m:
n:

Which loop condition do you like better? And why?

```
// Process 0..n
k= 1;
// inv: 0..k-1 processed, 0 <= k <= n+1
while (k <= n) {
    Process k;
    k= k+1;
}
// post: 0..n processed
```

```
// Process 0..n
k= 1;
// inv: 0..k-1 processed, 0 <= k <= n+1
while (k != n+1) {
    Process k;
    k= k+1;
}
// post: 0..n processed
```