Introduction to Programming - Concepts and Tools

Lecture 2

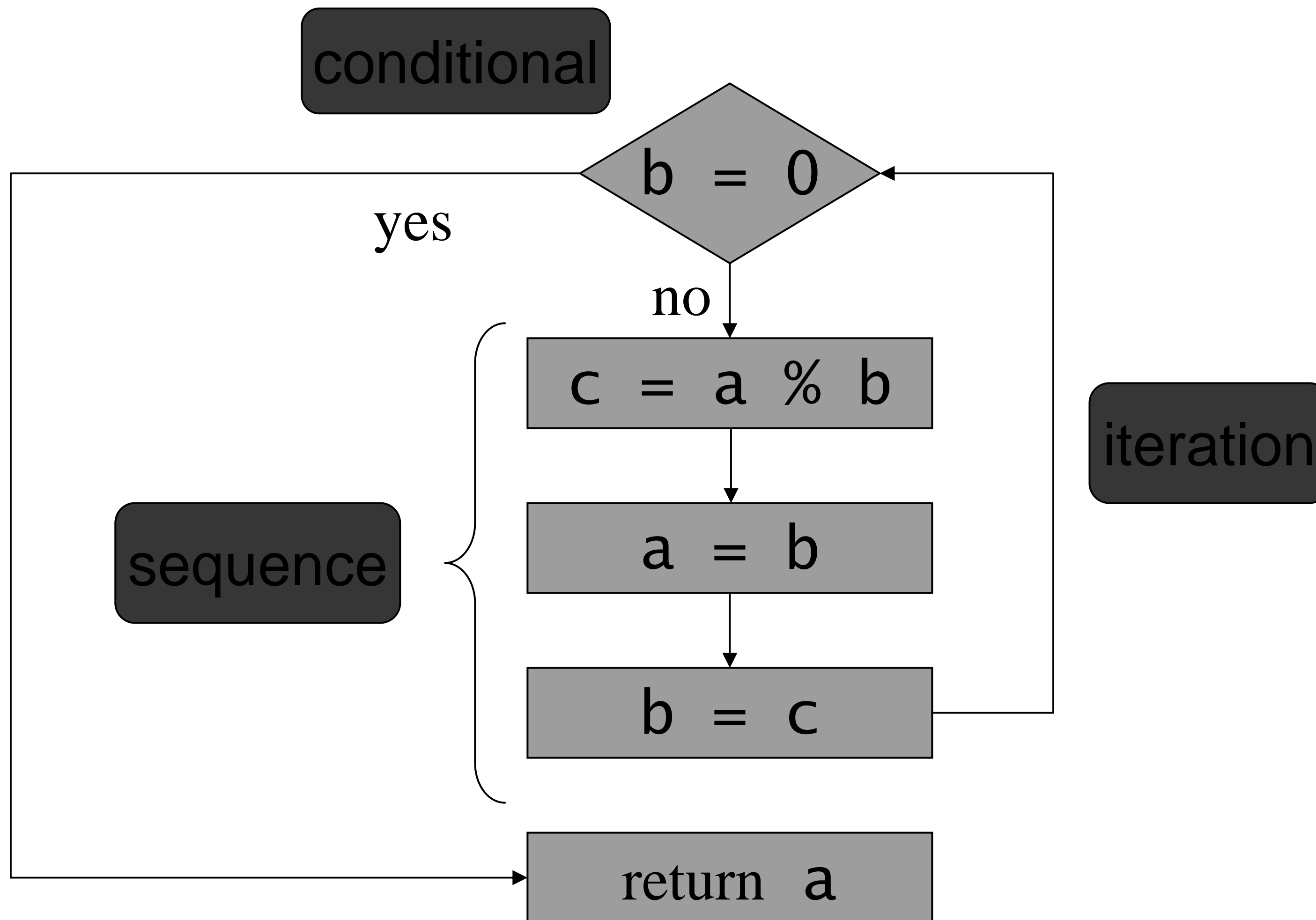
Control Structures

Jens Chr. Godskesen

Overview

- Control structures
  - Statements
  - Branching (`if` and `if-else`)
  - Loops (`for`-loops and `while`-loops)
  - Termination of loops
  - Escaping loops (`break` and `continue`)

Sequence, Condition, and Iteration



Sequence, Condition, and Iteration

In Java we would write:

```
while (b != 0) {
    c = a % b;
    a = b;
    b = c;
}
System.out.println("The gcd is: ” + a);
```

Statements

The things that can be put together in **sequence** are **statements,** i.e.:

- simple statements (variable declaration, assignment, and method call)
- blocks
- conditionals (`if`, `if-else`, and `switch` statements)
- loops (`while` and `for` statements)

Variable Declaration Statements

The simplest **variable declaration statement** is a comma separated list of variables (of the same type)

```
int i;
int a, b, c;
```

Variables may be **initialized** when declared

```
int a, b = 0, c;
```

Expression Statements

We have seen **assignment** expression statements e.g.

```
c = a % b;
```

and **method call** expression statements e.g.

```
System.out.println("The gcd is: ” + a);
```

In general an expression statement is on the form

```
expr;
```

for some expression `expr`. E.g. `i++;` is an expression statement whereas `a % b;` is not.

Block Statements

A seqence of statements may be grouped inside a **block**, started by a left ({) and terminated by a right brace (}). E.g.

```
{
    c = a % b;
    a = b;
    b = c;
}
```

Blocks

```
while (b != 0)
    c = a % b;
    a = b;
    b = c;
```

```
while (b != 0) {
    c = a % b;
    a = b;
    b = c;
 }
```

The while loops above are *inequivalent*

Blocks

```
while (b != 0)
    c = a % b;
    a = b;
    b = c;
```

```
while (b != 0) {
    c = a % b;
    a = b;
    b = c;
}
```

The while loops above are *inequivalent*, but the two below are similar

```
while (b != 0)
    c = a % b;
    a = b;
    b = c;
```

```
while (b != 0)
    c = a % b;
a = b;
b = c;
```

Nested Blocks

Blocks may be **nested** as in

```
int i = 0;
{
    int j = 0;
    while (j < 10) {
        System.out.println("i+j is " + (i+j));
        j++; i++;
    }
}
```

The **local variable** `j` is declared and initialized when entering the *inner block* but it doesn't exists in memory when the block is exited.

Empty Statements

A semicolon all by itself is an **empty statement** doing nothing

What is the outcome of running:

```
int i = 0;
{
   int j = 0;
    while (j < 10) {
      System.out.println("i+j is " + (i+j));
      ; //j++; i++;
   }
}
```

Boolean Expressions

A **boolean expression** is an expression evaluating to *true* or *false*.

- Literals: `true` and `false`
- *Equality operators*: `a == b` and `b != 0`
- *Relational operators*:
  - `a < b, 1 > a, 0 <= b, b >= 10`
- *Logical operators*:
  - negation: `!(b == 0)`
  - or: `(a < b) || (b > a)`
  - and: `(a <= b) && (b > a)`

Precedence and Associativity

The precedence between the operators are

`||` < `&&` < equational < relational < `!`

`B || C && D` equals `B || (C && D)`

`! B == a < b` means `(!B) == (a < b)`

The binary operators associates from left to right:

`a != b == B` means `(a != b) == B`

Short-Curciut Evaluation

`&&` and `||` uses **short-curciut evaluation**, i.e. *expressions are evaluated from left to right in so far further evaluation is needed.*

In

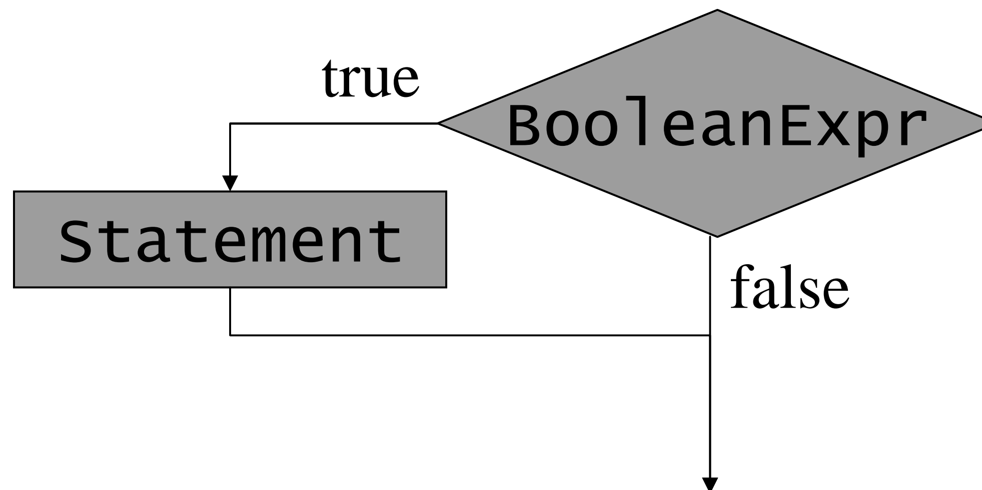
```
boolean B =(i != 0) && (j/i > 1);
```

`(j/i > 1)` is evaluated only in case `(i != 0)` is true.

`if` statements

The simplest *conditional* statement is the `if`-statement

```
if (BooleanExpr)
    Statement
```



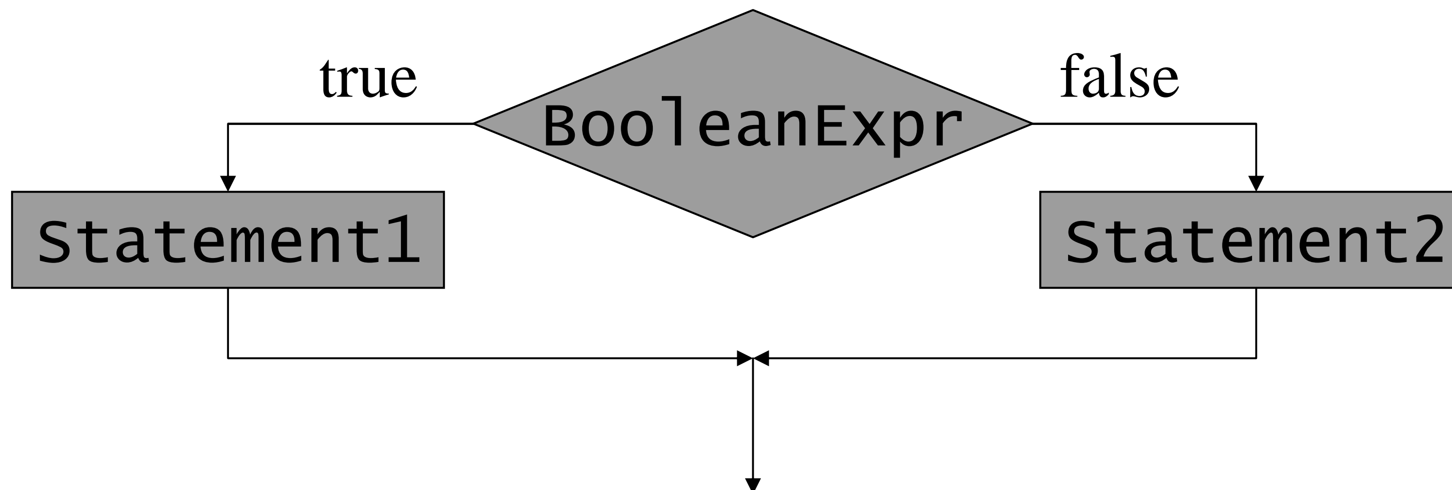
Example

```
if (a > b) {
  int tmp = a;
  a = b;
  b = tmp;
}
System.out.print(”The sorted order of a and b is: ”);
System.out.println(a + ”, ” + b);
```

`if-else` statements

The `if-else`-statement has the form

```
if (BooleanExpr)
   Statement1
else Statement2
```



Example

```
System.out.print(”The sorted order of a, b, and c is: ”);
if (a >= b) {
   if (b >= c)
      System.out.println(a + ", " + b + ", " + c);
}
else { // if (a < b)
  if (a >= c) {
    if (c >= b)
      System.out.println(a + ", " + c + ", " + b);
  }
  else {
     .....
  }
}
```

Example

Since `if-else` statements are genuine statements in this case we may skip the block structure getting:

```
System.out.print(”The sorted order of a, b, and c is: ”);
if (a >= b)
   if (b >= c)
      System.out.println(a + ", " + b + ", " + c);
else
  if (a >= c)
    if (c >= b)
      System.out.println(a + ", " + c + ", " + b);
  else .....
```

`if-else` statements

The true branch of the `if` statements contains only an `if` statement (and no `else` statement) so it's more elegant to combine the two boolean expressions with `&&`.

```
System.out.print("The sorted order of a, b, and c is: ");
if (a >= b && b >= c)
  System.out.println(a + ", " + b + ", " + c);
else if (a >= c && c >= b)
       System.out.println(a + ", " + c + ", " + b);
     else if (b >= a && a >= c)
            System.out.println(b + ", " + a + ", " + c);
          else if (b >= c && c >= a)
                 System.out.println(b + ", " + c + ", " + a);
               else if (c >= a && a >= b)
                      System.out.println(c + ", " + a + ", " + b);
                    else System.out.println(c + ", " + b + ", " + a);
```

`if-else` statements

But this code is more *efficient* (at most 3 comparisons needed)

```
System.out.print(”The sorted order of a, b, and c is: ”);
if (a > b)
  if (a > c)
    if (b > c) // a > b > c
      System.out.println(a + ", " + b + ", " + c);
    else // a > c >= b
      System.out.println(a + ", " + c + ", " + b);
  else // c >= a > b
    System.out.println(c + ", " + a + ", " + b);
else // b >= a
  if (a > c) // b >= a > c
    System.out.println(b + ", " + a + ", " + c);
  else // b >= a && c >= a
    if (b > c) // b > c >= a
      System.out.println(b + ", " + c + ", " + a);
    else // c >= b >= a
      System.out.println(c + ", " + b + ", " + a);
```

The Dangling `else`

An `else` is always matched with the nears preceeding `if` that doesn't have an `else`, hence the indentation below is misleading

```
if (BooleanExpr1)
  if (BooleanExpr2)
     Statement1
else Statement2
```

it should have been

```
if (BooleanExpr1)
  if (BooleanExpr2)
     Statement1
  else Statement2
```

`switch` Statements

Suppose we want to print the day of the week assuming `dayOfWeek` holds values from `1` to `7`.

```
if (dayOfWeek == 1) nameOfDay = "Sunday";
if (dayOfWeek == 2) nameOfDay = "Monday";
if (dayOfWeek == 3) nameOfDay = "Tuesday";
if (dayOfWeek == 4) nameOfDay = "Weednsday";
if (dayOfWeek == 5) nameOfDay = "Thursday";
if (dayOfWeek == 6) nameOfDay = "Friday";
if (dayOfWeek == 7) nameOfDay = "Saturday";
System.out.println(”It is ” + nameOfDay);
```

Too many comparisons!

`switch` Statements

The following code is better

```
if (dayOfWeek == 1) nameOfDay = "Sunday";
else if (dayOfWeek == 2) nameOfDay = "Monday";
else if (dayOfWeek == 3) nameOfDay = "Tuesday";
else if (dayOfWeek == 4) nameOfDay = "Weednsday";
else if (dayOfWeek == 5) nameOfDay = "Thursday";
else if (dayOfWeek == 6) nameOfDay = "Friday";
else if (dayOfWeek == 7) nameOfDay = "Saturday";
System.out.println(”It is ” + nameOfDay);
```

But even better (more structured) to use a `switch`

`switch` Statements

```
switch (dayOfWeek) {
   case 1: nameOfDay = "Sunday"; break;
   case 2: nameOfDay = "Monday"; break;
   case 3: nameOfDay = "Tuesday"; break;
   case 4: nameOfDay = "Weednsday"; break;
   case 5: nameOfDay = "Thursday"; break;
   case 6: nameOfDay = "Friday"; break;
   case 7: nameOfDay = "Saturday";
}
System.out.println("It is " + nameOfDay);
```

A `switch` requires that the *condition* evaluates to an *integer type* and that the constants after the `case` labels are unique.

A `break` causes execution to continue immediately after the `switch` statement.

`switch` Statements

```
switch (dayOfWeek) {
  case 7:
  case 1: System.out.println("It is weekend"); break;
  case 2:
  case 3:
  case 4:
  case 5:
  case 6: System.out.println("It is a week day"); break;
  default: System.out.println("Not a day number " + dayOfWeek);
}
```

If there is no `break` statement the execution “falls through”.
There can only be one `default` label, it can occur anywhere.

Iterations

Iteration is to execute certain parts of the code several times as e.g. in

```
a = Console.in.readInt();
b = Console.in.readInt();
while (b != 0) {
        c = a % b;
        a = b;
        b = c;
   }
System.out.println("The gcd is: ” + a);
```

How to implement the algorithm above without iterations?

`for` statements

The `for` loop is most often used to perform a definite number of iterations, say

```
int i;
double square_root;
for (i = 1; i <= 10; i++) {
    square_root = Math.sqrt(i);
    System.out.println("the square root of " + i +
                       " is " + square_root);
}
```

The *initialization* `i = 1` is performed only once. The *condition* `i <= 10` is evaluated before each interation, and the *incremention* `i++` is executed ending an interation.

`for` statements

A `for` statement may contain a *local variable* in its initialization

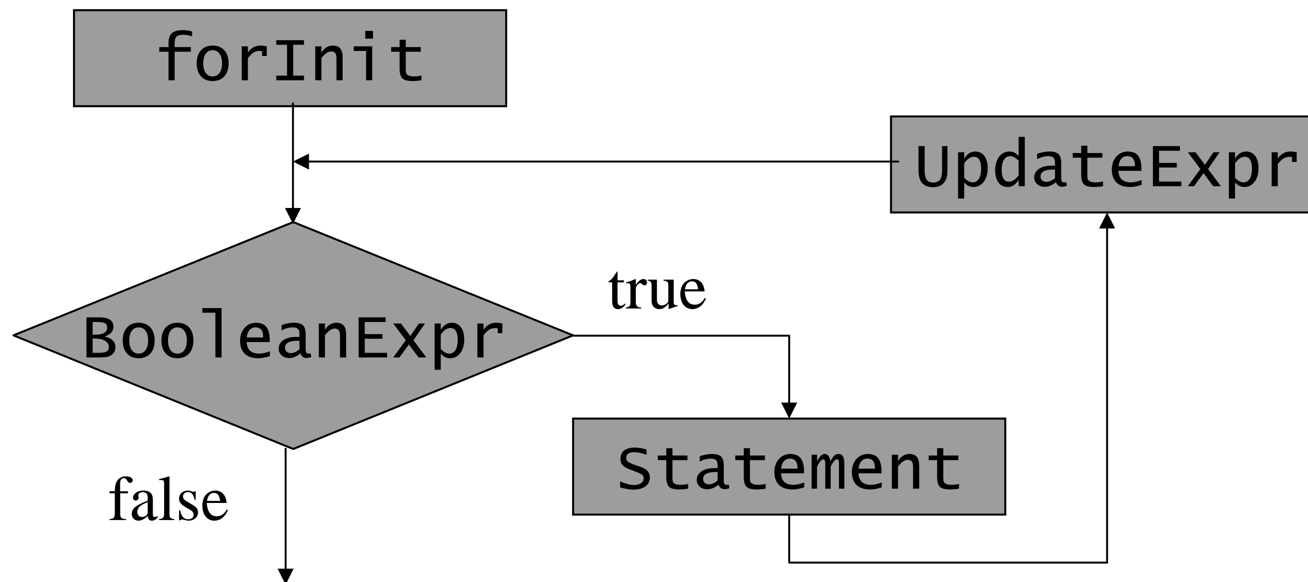
```
for (char c = 1; c <= 100; c++)
    System.out.println("the code for " + c +
                       " is " + (int)c);
```

that is not visible outside the for loop, it is said to have *local scope*.

`for` statements

The general form of a `for` statement is

```
for(forInit; BooleanExpr; UpdateExpr)
    Statement
```



`while` statements

```
i = 1; // initialization of the loop
while (i <= 10) {
  square_root = Math.sqrt(i);
  System.out.println("the square root of " + i +
                     " is " + square_root);
  i++; // prepare for next iteration
}
```

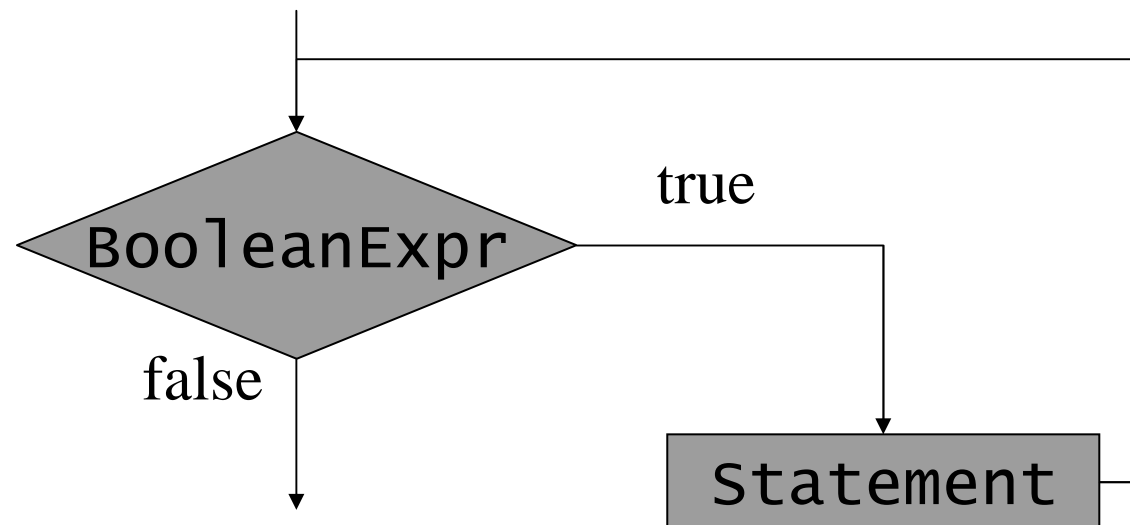
is equivalent to

```
for (i = 1; i <= 10; i++) {
    square_root = Math.sqrt(i);
    System.out.println("the square root of " + i +
                       " is " + square_root);
}
```

`while` statements

The `while` statement has the following general form

```
while (BooleanExpr)
   Statement
```



`while` statements

```
while (B)
  S
```

is equivalent to

```
if (B) {
   S
   while (B)
     S
 }
```

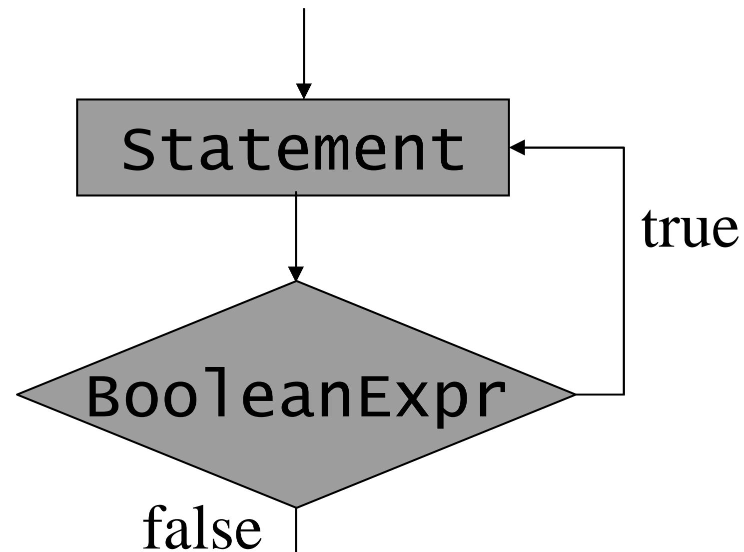
Note that the body of a `while` loop may not be executed at all.

`do-while` statements

The `do-while` statement has the following form

```
do  Statement
while (BooleanExpr);
```

`Statement` is always executed at least once.



Example

In [JbD] you may find code similar to this (p.74)

```
double number, runningTotal = 0;
int count = 0;
System.out.println("Type some numbers, ” +
                   "the last one being 0");
number = Console.in.readDouble();
while (number != 0) {
   runningTotal = runningTotal + number;
   count = count + 1;
   number = Console.in.readDouble();
}
System.out.print("The average of the ” + count);
System.out.print(" numbers is ” + runningTotal/count);
```

for calculating the average of a seqence of numbers. Let’s redo it using a `do-while` loop.

Example

```
double number, runningTotal = 0;
int count = 0;
System.out.println("Type some numbers, “ +
                   "the last one being 0");

do {
  number = Console.in.readDouble();
  runningTotal = runningTotal + number;
  count++;
}
while (number != 0);
System.out.print("The average of the " + (count-1));
System.out.print(" numbers is " + runningTotal/(count-1));
```

Notice that `count` is subtracted by one! What happens if only a `0` is typed?

Termination, sentinels

On the previous slide we used a **sentinel** to terminate a loop

```
System.out.println("Type some numbers, “ +
                   "the last one being 0");
do {
  number = Console.in.readDouble();
  …
}
while (number != 0);
```

If the user don’t input 0 the loop will not terminate.

Termination, no counter incrementation

```
i = 1;
while (i <= 10) {
   square_root = Math.sqrt(i);
   System.out.println("the square root of " + i +
                      " is " + square_root);
   //i++; // prepare for next iteration
 }
```

If `i` is not incremented the loop will not terminate.

It's the programmers responsability to make sure the program terminates

Termination, missing check

```
// print multiples of 13 between 1 and 100
i = 13;
while (i != 100) {
    System.out.println(i);
    i += 13;
  }
```

Testing for **<=** or **>=** is usually better than testing for equality or inequality.

Termination cannot be guaranteed

```
System.out.println("Type a natural number: ");

int n = Console.in.readInt();

while (n > 1) {
   Console.in.readLine(); // allows to step through
   if ( n%2 == 0) n = n/2;
    else n = 3*n + 1;
   System.out.println("The current value of n is: " + n );
}

System.out.println("The program terminated");
```

It's unknown if the code terminates for any value of `n` [Collatz].

Mandatory Assignment

The first mandatory assignment has been posed. Before doing the assignment

– Make groups!

Before handing in the solution:

– Make code public at your homepage and write link on the cover sheet of the pile of answer handed in
– Only hand in code you've tried to compile

The End

Continue in room 4A54 or 4A56