CITS1001 week 5
Repetition

Arran Stewart

March 27, 2018

Outline

- topics:
  - repetition
  - for loops
  - while loops
- Reading: Chapter 4 (section 4.10 to 4.16) of Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling

Review

- We often want to repeat some actions over and over
- e.g. “do this action for each student in the university”
  - typically we’ll handle this with a “for-each” loop
- e.g. “do this action seventeen times”
  - This is typically done using a `for` loop
- e.g. “do this action until this condition is true”
  - The third paradigm is done using a while loop

for loops

Pseudocode for `for` loop

General form of the for loop

```
for(initialization; condition; post-body action) {
statements to be repeated
}
```

for loop structure

The *header* of the loop is the information in the round brackets

```
for (<initialization>; <boolean-expression>; <post-body update>)
{
<statement-1>
<statement-2>
…
<statement-n>
}
```

The *body* of the loop is the collection of statements in the curly brackets

Header – the initialization part

- The initialization part consists of any Java statement
- It is performed *once* only, when execution first reaches the for loop
- It is normally used to initialize a counter variable
    - (also known as “the index variable”)

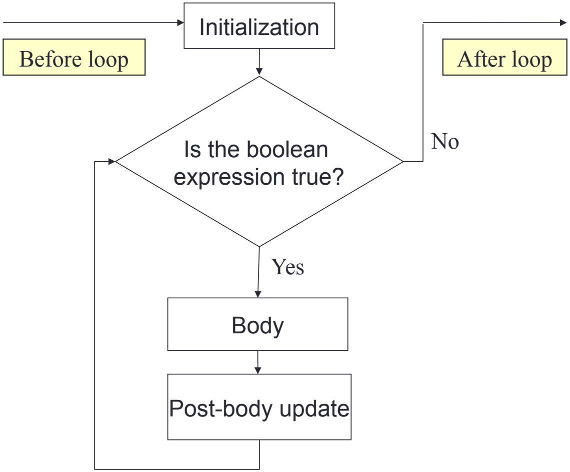
Header – the boolean-expression part

- The boolean expression controls whether or not the body of the loop is executed
- The expression is evaluated immediately after initialization, and at the start of every subsequent iteration
- If its value is `true`, then the statements in the body of the loop are executed;
  if its value is `false`, then the loop has finished and the statements in the body are NOT executed
    - When the loop finishes, execution continues at the first statement after the `for` loop

Header – the post-body update

- The post-body update is a Java statement that is executed once each time the body of the for loop is executed
- It is executed immediately after the *last* statement of the body has been executed
- It is usually used to *update* the counter variable

for loop flowchart



The for loop idiom

- for loops are quite general, but one common use is when we want to do something a specific number of times.

The for loop idiom

- for loops are quite general, but one common use is when we want to do something a specific number of times.
- A typical idiom for doing that is code like the following:

```
for (int i=0; i<5; i=i+1) {
  System.out.println(i);
}
```

The for loop idiom

- for loops are quite general, but one common use is when we want to do something a specific number of times.
- A typical idiom for doing that is code like the following:

```
for (int i=0; i<5; i=i+1) {
  System.out.println(i);
}
```

- Output:

```
0
1
2
3
4
```

The for loop idiom – steps

How did this work?

Initialization creates i and sets it to 0

First iteration
$$\left\{\begin{array}{l}\text{Check if i } < 5\text{, yes}\\ \text{Print out i – causes 0 to appear on terminal window}\\ \text{Update i from 0 to 1}\end{array}\right.$$

Second iteration
$$\left\{\begin{array}{l}\text{Check if i } < 5\text{, yes}\\ \text{Print out i – causes 1 to appear on terminal window}\\ \text{Update i from 1 to 2}\end{array}\right.$$

$$\left\{\begin{array}{l}\text{Check if i } < 5\text{, yes}\\ \text{Print out i – causes 2 to appear on terminal window}\\ \text{Update i from 2 to 3}\end{array}\right.$$

$$\left\{\begin{array}{l}\text{Check if i } < 5\text{, yes}\\ \text{Print out i – causes 3 to appear on terminal window}\\ \text{Update i from 3 to 4}\end{array}\right.$$

$$\left\{\begin{array}{l}\text{Check if i } < 5\text{, yes}\\ \text{Print out i – causes 4 to appear on terminal window}\\ \text{Update i from 4 to 5}\end{array}\right.$$

Check if i $<$ 5, NO

The increment operator

- Something you may see in existing Java code
- The post-body update often consists of just:

  ```
  i = i+1;
  ```

- Because it's so often used, there is a short-hand notation for this operation –
  - The statement `i=i+1` may be replaced simply by `i++` (often pronounced "increment `i`")

    ```
    for (int i=0; i<5; i++) {
      System.out.println(i);
    }
    ```

NB: Use either `i=i+1` or `i++`, but don't try and mix the two

Braces

- If the body consists of only one statement, then you can leave out the braces . . .
- However, it is better style to always include them
- Serious security bugs have been caused by programmers omitting them

Braces (2)

```
for (int i=0; i<5; i++) {
  System.out.println(i);
}
```

is the same as

```
for (int i=0; i<5; i++)
  System.out.println(i);
```

Writing for loops

- What output do we expect to get from the following code?

```
for (int i=0; i<5; i++);
{
  System.out.println(i*i);
}
```

Writing for loops

- What output do we expect to get from the following code?

```
for (int i=0; i<5; i++);
{
  System.out.println(i*i);
}
```

- Perhaps . . . ?

```
0
1
4
9
16
```

Writing for loops

- What output do we expect to get from the following code?

```
for (int i=0; i<5; i++);
{
  System.out.println(i*i);
}
```

- Perhaps . . . ?

```
0
1
4
9
16
```

- In fact, the output is just

```
100
```

For loop issues

- A common mistake when writing `for` loops is accidentally including a surplus semicolon.

  (this mistake can be very hard to track down)
- The problem with the previous code was a problem with the loop body
- We might have *thought* the body was

  ```
  System.out.println(i*i);
  ```

  but in fact it was

  ```
  ;
  ```

A common mistake

```
int i;
for (i=0; i<10; i++) ;
{
   System.out.println(i*i);
}
```

```
int i;
for (i=0; i<10; i++)
{
   System.out.println(i*i);
}
```

The first loop has an *empty body* (just a single semicolon!), while the second shows the "desired" body

Another use for `for` loops – making tables

- Another common use of for loops is to produce tables
- Suppose you are asked to produce a temperature conversion table listing the Fahrenheit equivalents of Celsius temperatures from 0–100°C, going up in increments of 5°C

Celsius	Farenheit
0	32
5	41
10	50
15	59
20	68
...	

Making tables (cont'd)

A for loop is the solution:

```
int celsius;
int fahrenheit;
for (celsius=0; celsius <= 100; celsius = celsius + 5) {
  fahrenheit = 32 + celsius*9/5;
  System.out.print(celsius);
  System.out.print(" ");
  System.out.println(fahrenheit);
}
```

- NB: we use System.out.print() instead of System.out.println() when we want to print something without starting a new line afterward

A numerical example – approximating π

- A formula for the value of π is:

$$\pi = 4 \times (\tfrac{1}{1} - \tfrac{1}{3} + \tfrac{1}{5} - \tfrac{1}{7} + \tfrac{1}{9} - \ldots)$$

- Suppose we wish to approximate π using this formula; here are two approaches we might take
  - Approximate π using a given number of terms from the above formula (say, the first 10 terms)
  - Approximate π to within a given accuracy (e.g., to within 0.01)
- The first way is best done using a `for` loop

Approximating π using a given number of terms

Formula:

$$\pi = 4 \times (\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ...)$$

Code:

```
public double pi(int n) {
  double approx_pi = 0;
  double mult      = 4;                    // mult. each term
                                           //   by +4 or -4
  for (int i=0; i<n; i++) {
    approx_pi = approx_pi + mult/(2*i+1);
    mult = -mult;                          // "flip" multiplier
  }
  return approx_pi;
}
```

Approximating π – variable values at top of loop

i	mult	2*i+1	approx
0	4.0	1	0.00
1	-4.0	3	4.00
2	4.0	5	2.67
3	-4.0	7	3.47
4	4.0	9	2.90
5	-4.0	11	3.34
6	4.0	13	2.98

Loop stops when `i` reaches the requested value

for loops with bigger steps

```
// Print multiples of 3 up to 40
for(int num = 3; num < 40; num = num + 3) {
    System.out.println(num);
}
```

Output:

```
3
6
9
12
15
18
21
24
27
30
33
36
39
```

Review of for loops

Use them when:

- the number of repetitions is known in advance
- an index variable is required
- there is a regular step-size

But note:

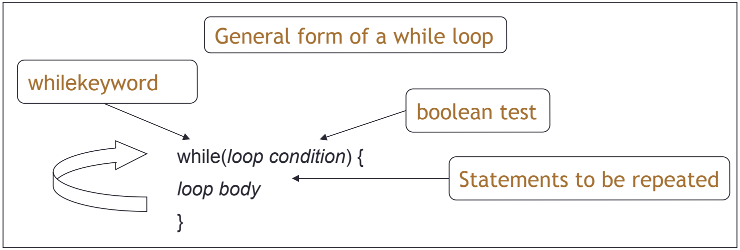
- "For-each" loops have less scope for error than for loops
- So: use a for-each loop unless you need access to indices or step-size

while loops

Use of `while` loops

- The repetition pattern embodied in `while` loops is: "do this action until this condition is true"
    - We don't know in advance how many iterations there will be
- In Java, this is done with a while loop
- We use a boolean condition to decide whether or not to keep going

while loop pseudocode

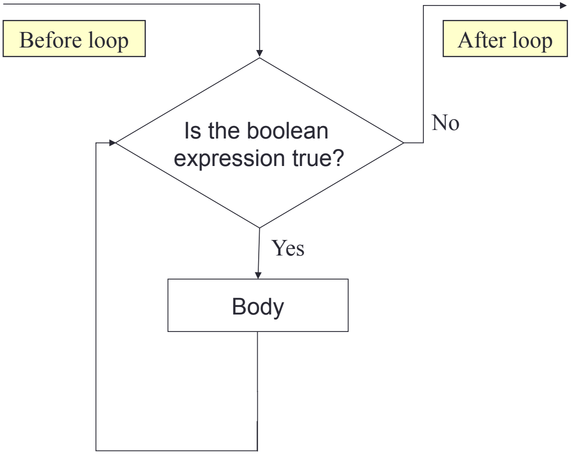


```
while(loop condition) {
loop body
}
```

Pseudo-code expression of the actions of a while loop

while we wish to continue, do the things in the loop body

while loop flowchart



Looking for your keys

```
while( the keys are missing ) {
   look in the next place
}
```

or equivalently:

```
while( not( the keys have been found )) {
   look in the next place
}
```

Looking for your keys – Java code

```
boolean stillSearching = true;
Location place = firstPlace;
while( stillSearching) {
    if( /* the keys are in place... */ ) {
      stillSearching = false;
    } else {
      place = next(place);
}
```

π to within a given accuracy

Another strategy for approximating π:

- approximate it to within a given accuracy (say, to within 0.001)

π to within a given accuracy (cont'd)

```
public double pi(double accuracy) {
  double approx = 0;
  double mult   = 4;
  double denom  = 1;
  while (Math.abs(mult/denom) > accuracy) { // use Math class
    approx = approx + mult/denom;
    mult   = -mult;
    denom  = denom + 2;
  }
  return approx;
```

Approximating π – values of variables at top of loop

Step	denom	mult/denom	approx
0	1.0	4.00	0.00
1	3.0	-1.33	4.00
2	5.0	0.80	2.67
3	7.0	-0.57	3.47
4	9.0	0.44	2.90
5	11.0	-0.36	3.34
6	13.0	0.31	2.98

Loop stops when the next term is smaller than `accuracy`

Features to note

- We have effectively declared an index variable
  - (In this case, `denom`)
- The index variable must be incremented explicitly
  - It's not updated automatically in the header, as with the `for` loop
- The condition must be expressed correctly
- We must know that the loop will end

Exercise – searching a collection with `while`

- Recall the “book journal” class from previous lectures
- Let us write another taken on a search method: one which searches for the first title containing a search string, and returns the index of that item (or -1 if no such item is found)

Searching a collection (cont'd)

- pseudocode:

```
index = 0
stillSearching = true

while stillSearching && index < bookTitles.size():
  bookTitle = bookTitles.get(index)
  if bookTitle contains searchString:
    stillSearching = false // stop searching.
  else:
    index++

if stillSearching:
  return -1 // We didn't find it.
else:
  return index // Return where it was found.
```

Searching a collection (cont'd)

- code: see http://teaching.csse.uwa.edu.au/units/CITS1001/lectures/wk05-books-journal-search.html

Questions

- The loop's condition repeaedly asks the bookTitles collection how many
  titles it is storing.

Questions

- The loop's condition repearedly asks the bookTitles collection how many
  titles it is storing.
    - Does the value returned by `size()` vary from one check to the next?

Questions

- The loop's condition repearedly asks the bookTitles collection how many
  titles it is storing.
  - Does the value returned by `size()` vary from one check to the next?
  - If not – rewrite the method so the number of titles is stored *once* in a variable, before the loop starts. Then use that variable, rather than calling `size()`.

Questions

- The loop's condition repeaedly asks the bookTitles collection how many
  titles it is storing.
    - Does the value returned by `size()` vary from one check to the next?
    - If not – rewrite the method so the number of titles is stored *once* in a variable, before the loop starts. Then use that variable, rather than calling `size()`.
- Can `findFirst` be implemented using the "search and return" pattern we've seen previously? How do the two implementations compare?

Questions (cont'd)

- Does the code in `findFirst` work if the collection is empty?

for-each versus while

- What are some of the advantages and drawbacks of using a “for-each” loop, as opposed to a `while` loop?

for-each versus while (cont'd)

- for-each:
    - Easier to write
    - Safer: it is guaranteed to stop
- while:
    - Easy to stop processing part-way through a collection
    - Doesn't even have to be used with a collection
    - Take care: could be an infinite loop
    - Handy when you don't know how many times a loop will be repeated

Exercises

- Write a while loop that prints out multiples of 5 between 10 and 95
- Write a while loop to add up the values 1 to 10 and print the sum once the loop has finished