Unit 6. Control Structures: Iteration

Introduction to Computer Science and Computer Programming
Introducción a la Informática y la Programación (IIP)

Year 2017/2018

Departamento de Sistemas Informáticos y Computación

Contents

Contents

Motivation

- Problem solution with a program sometimes requires a repeated execution of a sequence of instructions
- Number of repetitions can be known *a priori* or not
- ***Examples***:
  - Sum all the values of a list
  - Calculate the sum of the first n natural numbers
  - Check if a word exists in a dictionary
  - Ask the user for a value in a range
  - Code a text changing all its letters
  - ...

Motivation

- Programming languages have repetition instructions: ***loops***
- Loops allow to repeat a sequence of instructions as many times as needed
- Basically, a block of instructions (***body***) is ***repeated while*** a ***condition*** (***guard***) is `true`
- ***Iteration*** or loop execution: any time the body is executed
- ***Loop***: set of instructions executed until an exit condition appears

Contents

Iteration

- A problem solution must accomplish the following conditions:
  - Its instructions must be described in a finite manner
  - Its execution must be finite
- Many simple problems require only simple sequences of instructions
- Other problems require some instructions to be repeated many times
- If repetition is needed, when solving the problem we must:
  - Detect instruction patterns that are repeated
  - Express the solution by using the detected patterns and repeat instructions

Iteration

- For example, multiply a number a by n using repeated sums (suposing n is not negative)

$$a*n == \underbrace{a + a + a + ... + a}_{n} \quad \text{if } n>0$$

$$a*n == 0 \quad \text{if } n==0$$

$$a*n == 0 + \underbrace{a + a + a + ... + a}_{n}$$

Iteration

- Solution uses an auxiliar var `acc`, which in each iteration takes the values:

`0*a`	Iteration 0
`1*a`	Iteration 1
`2*a`	Iteration 2
`...`	
`n*a`	Iteration n

- In each iteration, `acc` is updated according to

```
acc = acc + a;
```

- Repetition pattern: in the ith iteration, the value of `acc` is actually `i*a`

Iteration
Elements

In each loop there are three main basic structural components:

- ***Guard of the loop***: condition that while it is evaluated to `true` makes the body of the loop (sequence of instructions) to be executed

- ***Body of the loop***: sequence of instructions that allows to arrive step by step towards the solution of the problem

- ***Variable initialisation***: before the loop, the vars that are involved in its execution must have a correct value

Iteration

Elements

- Taking into account those elements, the usual structure of iteration is:

  variable initialisation
  while guard of the loop
  body of the loop

- In Java syntax, this is transcribed into:

```
variable initialisation;
while (guard of the loop) {
    body of the loop;
}
```

Iteration

Elements

Multiplication by sum example:

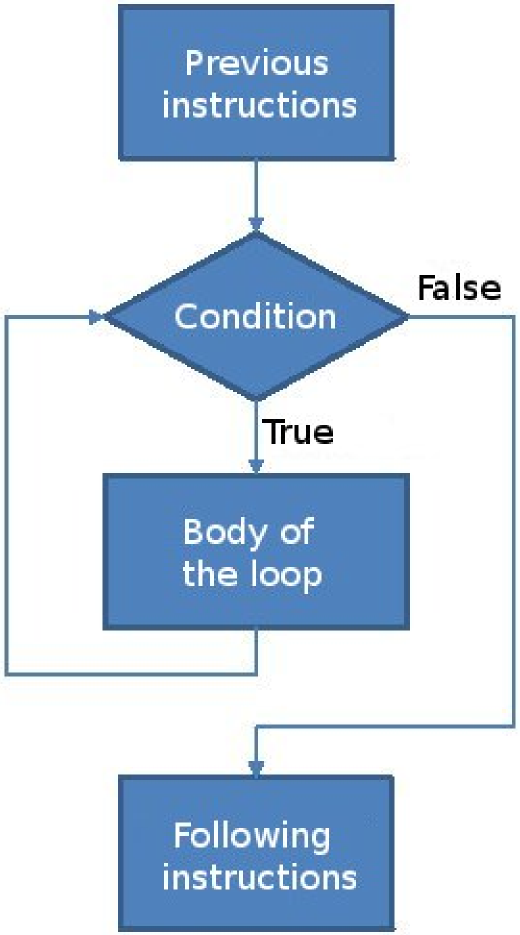
```
public static int mult(int a, int n) {
  int acc, numIt;

  // Variable initialisation
  acc = 0;
  numIt= 0;

  while (numIt<n) {  // Guard of the loop
    // Body of the loop
    acc = acc + a;
    numIt++;
  }

  return acc;
}
```

Notice the use of `numIt` to express the termination/continuation of the loop

Contents

Iterative instructions in Java: `while` loop

```
while (condition)
  instruction;

while (condition) {
  instruction_1;
  instruction_2;
  ...
  instruction_n;
}
```



Iterative instructions in Java: `while` loop

`while` loops are usually used for *a priori* unknown number of iterations

E.g.: read integer numbers until a 0 appears, show the sum of all

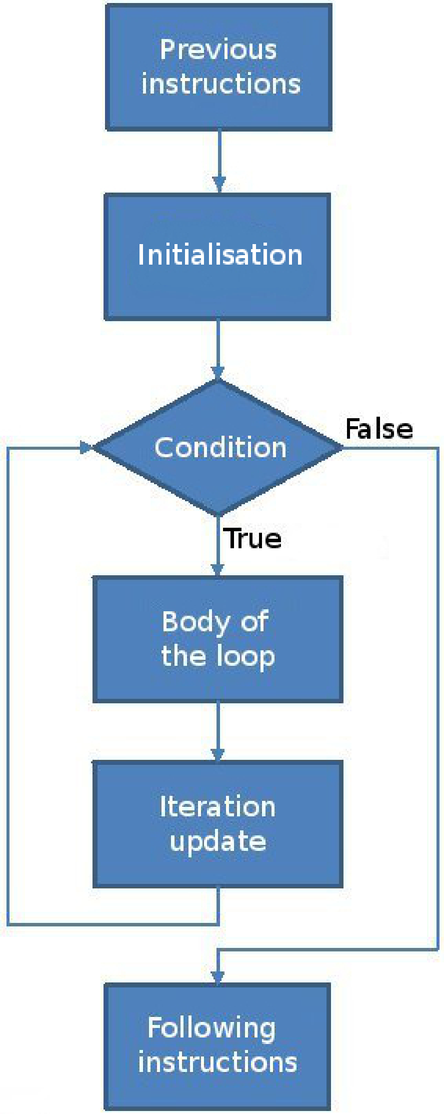
```
int i, s=0;

i=kbd.nextInt();
while (i!=0) {
  s+=i;
  i=kbd.nextInt();
}

System.out.println("Total sum: "+s);
```

Contents

Iterative instructions in Java: `for` loop

Execution

1. Execute the `initialisation` instructions
2. Evaluate `condition`
3. When it is `false`, go to step 6
4. Execute the instructions of the body
5. Execute `update` instructions, go to step 2
6. Continue with the instruction that follows the loop



Iterative instructions in Java: `for` loop

Syntax:

```
for ([initialisation]; [condition]; [update])
  instruction;

for ([initialisation]; [condition]; [update]) {
  instruction_1;
  ...
  instruction_n;
}
```

- Parts between brackets (`[]`) are optional
- ***Condition***: boolean expression (no condition is evaluated to `true`)
- ***Initialisation***: init of the loop; typically *counter=value* (`i=0`)
- ***Update***: progression of the loop; typically *counter=newVal* (`i=i+1`, `i++`)

`initialisation` and `update` could be lists of elements sepparated by commas (`,`) (e.g., `i=0,j=0,k=3` `i++,j+=2,k--`)

Iterative instructions in Java: `for` loop

The `for` loop:

```
for ([i1,i2,...,in]; [condition]; [u1,u2,...,un])
  instruction;
```

It is equivalent to the following `while` loop:

```
[i1;i2;...;in;]          // Initialization
while (condition) {
  instruction;
  [u1;u2;...;un;]        // Update
}
```

Iterative instructions in Java: `for` loop

`for` loops are usually used for a known number of iterations

E.g.: read ten real numbers and show the mean

```
double num, sum=0.0;
int i;

for (i=0;i<10;i++) {
  num=kbd.nextDouble();
  sum+=num;
}

System.out.println("Mean: "+sum/10);
```

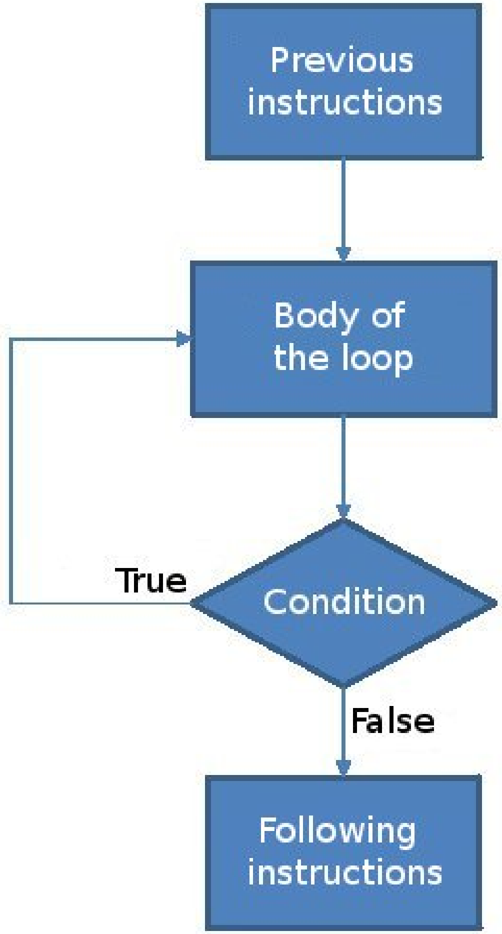
Contents

Iterative instructions in Java: `do - while` loop

Execution

1. Execute the instructions of the body of the loop until `condition` is `false`
2. The execution continues with the instruction that follows the loop

The instructions of the body of the loop get executed at least once



Iterative instructions in Java: `do - while` loop

Its general form is:

```
do
  instruction;
while (condition);
```

```
do {
  instruction_1;
  instruction_2;
  ...
  instruction_n;
} while (condition);
```

where `condition` is any boolean condition (guard of the loop) and the body of the loop is an instruction or a block of instructions

Iterative instructions in Java: `do - while` loop

- Differences between `while` and `do-while` loops:
  - Since in `do-while` the guard is evaluated after executing the body, at least the first iteration is executed in this loop
  - Syntax restriction: in `do-while`, a `;` must appear after `(condition)`

- Any problem expressed with a `do - while` loop can be expressed with a `while` loop and vice-versa, writing the auxiliary needed instructions

```
initial_instructions;
while (condition) {
  body_of_the_loop;
}
following_instructions;
```

```
initial_instructions;
if (condition)
  do {
    body_of_the_loop;
  } while (condition);
following_instructions;
```

Iterative instructions in Java: `do - while` loop

`do-while` loop is usually used for validating inputs

E.g.: read an age (must be non-negative)

```
int age;

do {
  System.out.print("Input your age: ");
  age=kbd.nextInt();
} while (age<0);
```

Contents

Nested loops

Loops can be inside other loops, which form ***nested loops***

In this case, the different guards must no interfere

E.g.: read ten valid ages and show the average age:

```
int i, age, sum=0;

for (i=0;i<10;i++) {
  do {
    System.out.print("Input your age: ");
    age=kbd.nextInt();
  } while (age<0);
  sum+=age;
}

System.out.println("Mean age: "+sum/10);
```

Nested loops

E.g.: generate sequences of four random real numbers between 0 and 1 until they sum more than 0.8

```
double x, sum=0.0;
int i;

while (sum<=0.8) {
  sum=0.0;
  for (i=0;i<4;i++) {
    x=Math.random();
    sum+=x;
  }
}
```

Nested loops

E.g.: show the multiplication table for numbers from 1 to 10

```
int i, j;

for (i=1;i<=10;i++) {

  System.out.println("Table for number "+i);
  System.out.println("-------------------");

  for (j=1;j<=10;j++)
    System.out.println(i+" X "+j+" = "+(i*j));

  System.out.println();

}
```

Contents

Theoretical aspects of iteration

Important details to take into account when a loop is implemented:

$\boxed{\text{A}}$ Define the ***loop invariant***

Define the repetitive structure from a property that is fulfilled at the end of each iteration, and that by the end of the loop guarantees the desired effect; this property is the loop invariant

Loop invariant is used to prove ***correctness***

$\boxed{\text{B}}$ Guarantee the ***termination***

The stop condition is the negation of the guard; the vars used in the guard are modified in the body in a way that will make the loop stop

Termination guarantees ***finiteness***

Theoretical aspects of iteration
The multiplication by sum example

```
public static int mult(int a, int n) {
  int acc, nI;

  // Variable initialisation
  acc = 0;
  nI = 0;

  while (nI<n) {  // Guard of the loop
    // Body of the loop
    acc = acc + a;
    nI++;
  }

  return acc;
}
```

- Invariant
  - The invariant is that in the ith iteration, acc has a value equal to a*nI
  - Thus, when nI equals to n, acc equals to a*n, which was the desired value
- Termination
  - When finishing the ith iteration, acc==a*nI
  - The loop must stop when nI equals to n
  - Stop condition: nI==n
  - Guard: nI!=n

Theoretical aspects of iteration
The GCD example

- Let consider the example of obtaining the Greatest Common Divisor (GCD) of two integer numbers A and B greater than 0
- Implementation of the Euclides algorithm (300 b.C.)
- GCD algorithm: let vars `a` and `b` take as initial values A and B, and then
  1. If `a` is equal to `b` stop, any of them is the GCD
  2. If `a>b`, make `a` equal to `a-b`,
     in other case make `b` equal to `b-a`
  3. Go to 1
- In Java code:

```
public static int gcd(int a, int b) {
  while (a!=b)
    if (a>b) a=a-b; else b=b-a;
  return a;
}
```

Theoretical aspects of iteration
The GCD example

$\boxed{\text{A}}$ The invariant property is given by:

- By the end of each iteration, GCD(A,B) = GCD(`a`,`b`), where A and B represent the initial values of `a` and `b`

- This property is a consequence of the mathematical property

$$GCD(a, b) = GCD(a - b, b) \qquad a > b > 0$$

Theoretical aspects of iteration
The GCD example

$\boxed{\text{B}}$ Termination: does always the execution stops? does it always arrive to the point where `a==b`?

Yes, because:

- `a` and `b` are always different from 0
- The value |`a-b`| becomes lower in each iteration and it is never lower than 0

Proving termination generally requires defining a function (boundary function)

Boundary function must accomplish:

- The more loop iterations, the lower its value becomes
- It is bounded