MALMÖ HÖGSKOLA

Programming Using C#,
Basic Course

Loops

Agenda:

- Increment and decrement operators
- Iterations, `for`, `while`, and `do-while` statements

Farid Naisan, University Lecturer, Malmö University, farid.naisan@mah.se

Iterations

- Loops are used a lot in programming. A sequence of statements that repeat themselves exactly or with minor modifications can be coded in form of loops.
- Iterations are applied whenever statements are to be carried out repeatedly:
  - for a certain number of times, or
  - while a certain control condition prevails.

C# iteration structures

- C# provides three structures for performing iterations:
  - `for`
  - `while`
  - `do-while`

- The `for` loop, used when the number of iterations are known in advance, ex
  - `For all students in a class`
    - `accumulate ages in years`

While and do-while statements

- The `while` loop (while one or more condition for looping prevail).
  - while input is valid
    - do the calculations
- The `do` loop
  - do
    - present the menu
    - continue until an item on the menu is selected by the user

Increment and decrement operators

- There are numerous times where a variable must be incremented or decremented.

```
count = count + 1;
count = count – 1;
```

- C# provide easier ways to increment and decrement a variable’s value.
- The operator used for these two purposes by using the unary operators ++ and -- respectively. Note: no black space between the ++ and – symbols.

Postfix or prefix

- The `++` or `--` unary operators can be applied in two forms, either in the postfix form as in:

```
count++;
count--;
```

  or the prefix form

```
++count;
--count;
```

- The difference is that in the postfix form, the value of the variable is used in the expression **before** the increment or decrement of the value of the variable takes place.
- In the prefix form, the variable is first incremented or decremented, and then the value is used.

Example

- Postfix:

```
int sum = 0, count = 0;
sum = 50 + count++; //Sum = 50, count = 1
```

- Prefix

```
int sum = 0, count = 0;
sum = 50 + ++count; //Sum = 51, count = 1
```

Prefix and postfix cont.

- Prefix and postfix forms of the decrement operator ‘--’ works exactly in the same way.
- When these operators occur as a single statement, there is not difference:

```
while (  (i < 100) && ( j > 0) )
{
    //statements
    i++;    //++i;  serves same purpose
    j--;    //--j; gives same effect
}
```

Flag

- A flag is a `bool` variable that monitors some condition in a program.
- The flag can be tested to see if the condition has changed.

```
bool done = false; //initialization

. . .

if (sum > 4000)

  done = true;
```

- The variable done can then be used for monitoring the flow of statements.

Flag – cont.

- A flag can also hold the final result of several bool expressions.

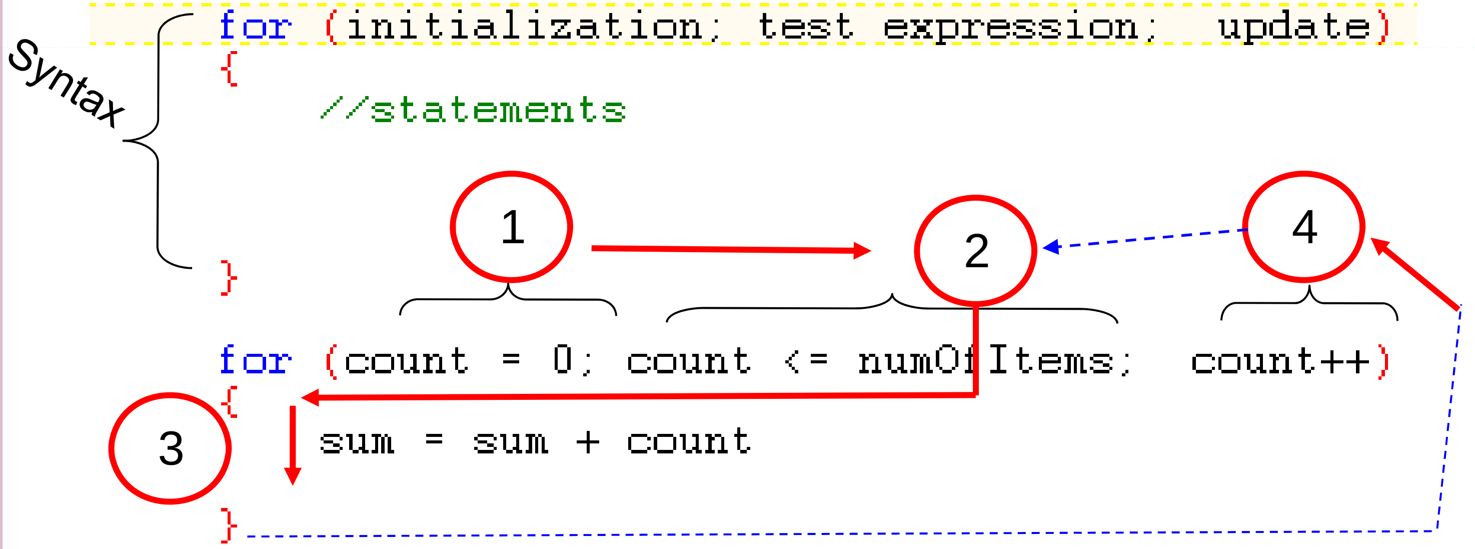
```
bool holiday = ( (day == “Sun”) ||
                 ( day == “Sat”) ) &&
                 (!isMyDutyTurn);

 if (holiday)  //= if (holiday == true)
    goOutAndHaveFun();
```

The for loop

- The for structure is designed for use in loops where the number of iterations is known.
- For loops have a special structure as shown in the example on next slide.
- For statements usually have a counter variable that is incremented or decremented after each iteration.
- The increment and decrement can any value, ex count += 5.

How a for-statement is executed

```
public double sumNumbers()
{
    int count = 0;
    int numOfIterms = 1000;
    double sum = 0.0;


    //A for loop to sum up numbers from 0 to 1000;

    for (initialization; test expression;  update)
    {
        //statements


    }

    for (count = 0; count <= numOfItems;  count++)
    {
        sum = sum + count

    }

}
```



Nested Loops

- As for the if statements, loops can be nested.
- If a loop is nested, the inner loop will execute all of its iterations for each time the outer loop executes once.
- The loop statements in this example will execute 100 times.

```
for(int i = 0; i < 10; i++)
{
    //outer loop statements
    for(int j = 0; j < 10; j++)
    {
        //inner loop statements

    }
    //outer loop statements
}
```

Exit a loop in advance, skip an iteration

- Loops can be exited ahead of the final iteration by using the keyword `break`.
- The statement break terminates in which the it is coded, i.e. the most immediate loop.
- An iteration can be skipped by using the keyword `continue`.
  - The rest of the code in the loop, after the `continue` statement to the end of the loop, will be skipped, and the execution continues with next iteration.

Continue and break example

this part will be skipped for row 5 and column 9

```
const int numOfRows = 15, numOfColumns = 4;

for (int row = 0; row < numOfRows; row++)
{
    if ((row == 5) || (row == 9))
        continue; //skip current row

    //Code...
    //...
    for (int col = 0; col < numOfColumns; col++)
    {
        //The continue statement affects only
        //the loop for (int col = ...., i.e the immediate loop
        if ((row == 1) && (col == 0))  //reserved for the boss
            continue; //skip current column,

        // other code
    }

    if (someBoolExpress) //some extraordinary condition
        break;  //Stop the iteration for (int row = ...
}

//Other code
```

The while loop

- The `while` loop is a control statement that is used when the number of iterations is not known.
- The code is executed repeatedly based on a given boolean condition.
- The while construct consists of a block of code and a condition.
- The curly braces must always be present.

```
while (condition)
{
  //block of one or
  //more statements
}
```

While loop is a pre-test loop

- The condition is tested prior to each iteration, inclusive the first one. (No semicolon after the `while` statement).

```
Boolean done = false;  //initial value

while (!done)
{
    ReadInput();                    //method
    Calculate();                    //method
    ShowResults();                  //method

    done = askToContinue();  //method with return
                            //value = true or false
}
```

Do-while loops

- The third loop construct that C# provides is the `do`-loop (`do-while` loop) a loop similar to `while`, but with the difference that the condition is tested after the first iteration.

```
do
{
  //statements
}while (condition);
```

do-loop example

- Notice the ‘;’ after the while statement.

```
Boolean done = false;  //initial value

do
{
    ReadInput();                //method
    Calculate();                //method
    ShowResults();              //method

    done = askToContinue();  //method with return
                            //value = true or false
}while (!done);
```

Nested loops

- `while` as well as `do-while` loops can be nested exactly as for `for`-loops.
- All the three loops can be nested in any combination. A `while` loop can nest, for example, a `for`-loop that in turn nests a `while` or another for-loop.
- The keywords `break` and `continue` can be applied in all the three iteration forms in the same way.

Infinite loops

- It is quite important that in a while and do loops the condition does not constantly hold the same value. This can easily happen as a common mistake.

```
int i = 0;
while (i < 1000)
{
   //statements

}
```

- This loop is infinite and is going to cause runtime problems, because the value of `i` does not change, and is always less than 1000, i.e. the condition is always `true`.
- Putting a i++ inside the block will remedy the problem in the above example.

Summary

- Three types of loops are provided by C#:
  - `for`
  - `while`
  - `do-while`
- Although any of these can be forced to work for all types of iterations, each of them is constructed for a certain purpose. Here are the rules of thumb:
  - Use a `for`-loop when the number of iterations are known.
  - Use a `while` loop when the condition should be checked from the beginning. The while loop executes 0 to many times.
  - Use a `do-while` loop instead of `while`-loop when the iteration must be done at least one time.