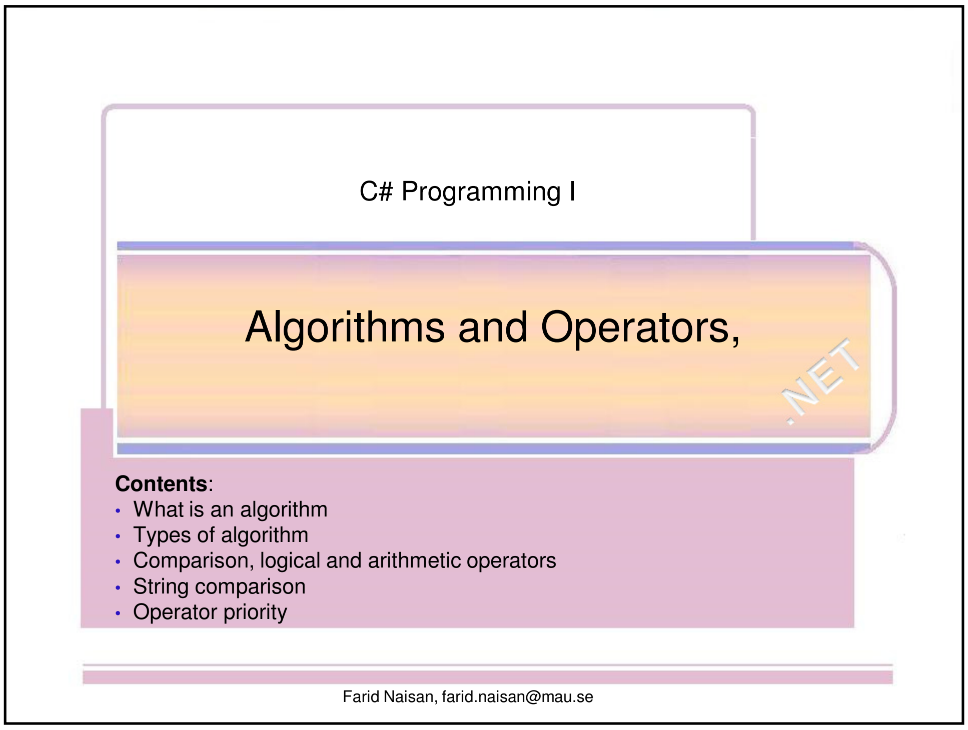
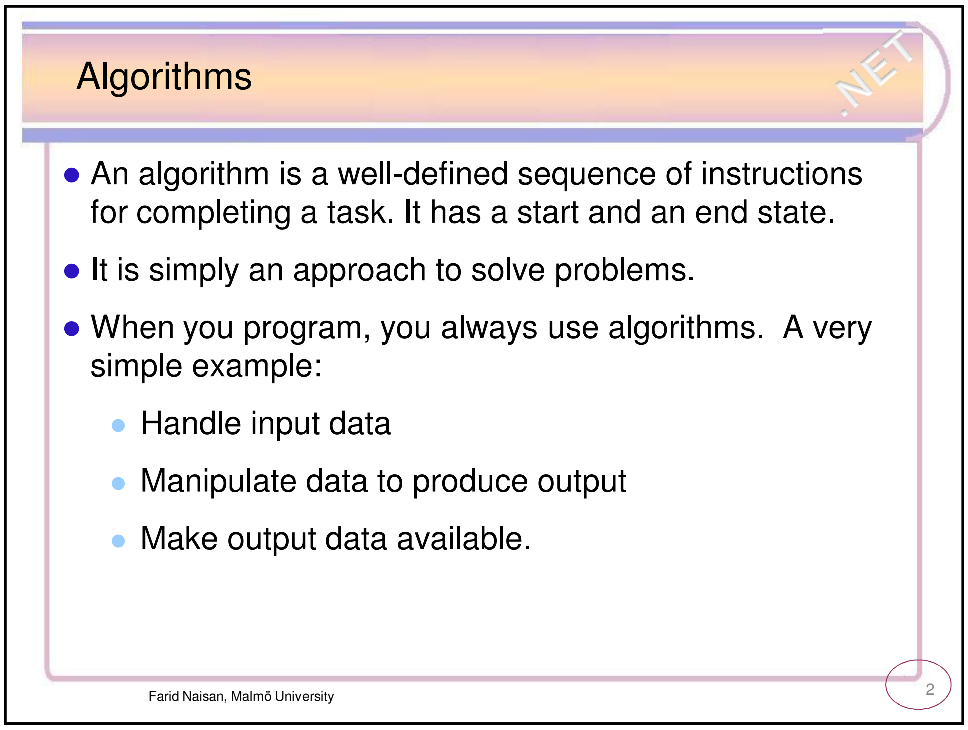


C# Programming I

Algorithms and Operators,

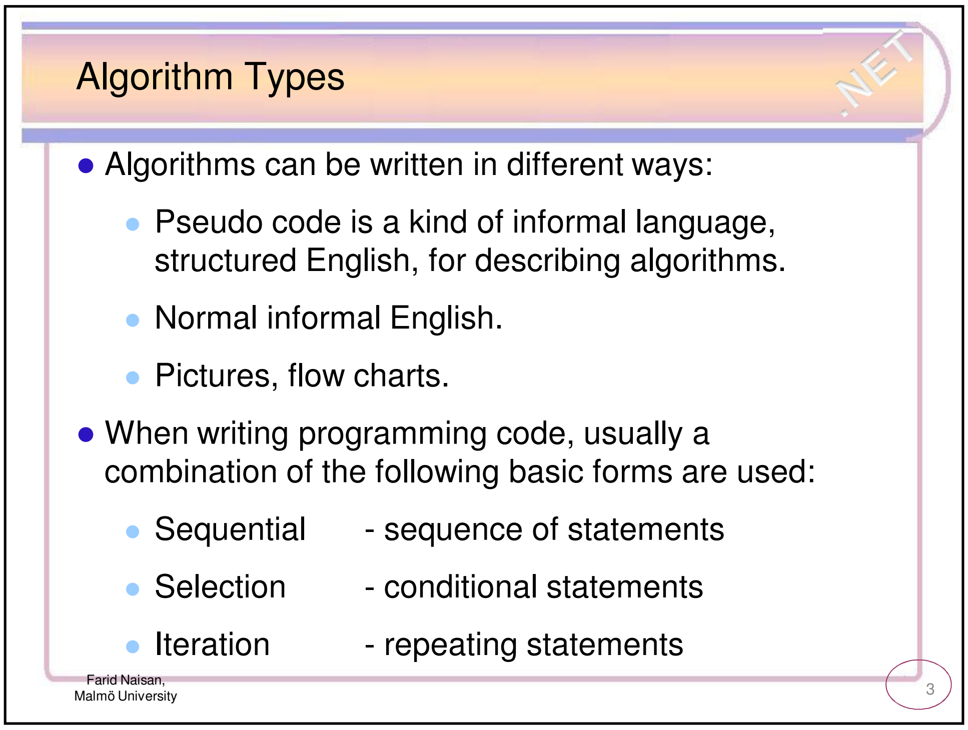
Contents:
- What is an algorithm
- Types of algorithm
- Comparison, logical and arithmetic operators
- String comparison
- Operator priority

Farid Naisan, farid.naisan@mau.se



Algorithms

- An algorithm is a well-defined sequence of instructions for completing a task. It has a start and an end state.
- It is simply an approach to solve problems.
- When you program, you always use algorithms. A very simple example:
  - Handle input data
  - Manipulate data to produce output
  - Make output data available.



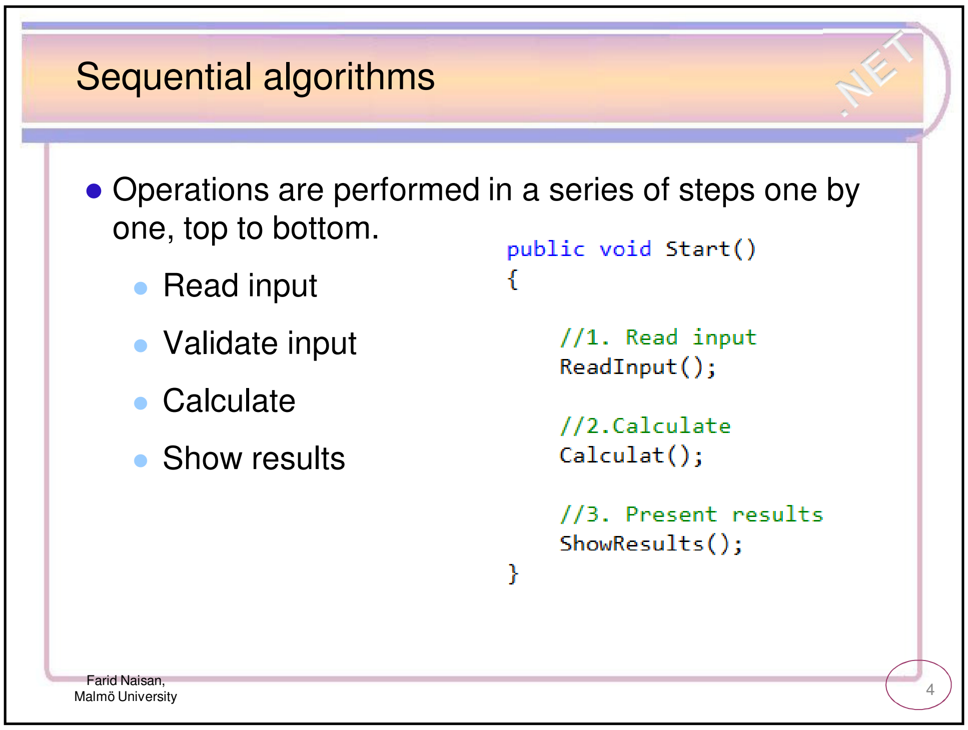
Algorithm Types

- Algorithms can be written in different ways:
  - Pseudo code is a kind of informal language, structured English, for describing algorithms.
  - Normal informal English.
  - Pictures, flow charts.
- When writing programming code, usually a combination of the following basic forms are used:

Sequential	- sequence of statements
Selection	- conditional statements
Iteration	- repeating statements

Farid Naisan,
Malmö University

3



Sequential algorithms

- Operations are performed in a series of steps one by one, top to bottom.
  - Read input
  - Validate input
  - Calculate
  - Show results

```
public void Start()
{

    //1. Read input
    ReadInput();

    //2.Calculate
    Calculat();

    //3. Present results
    ShowResults();
}
```

Farid Naisan,
Malmö University

4

Selection algorithm

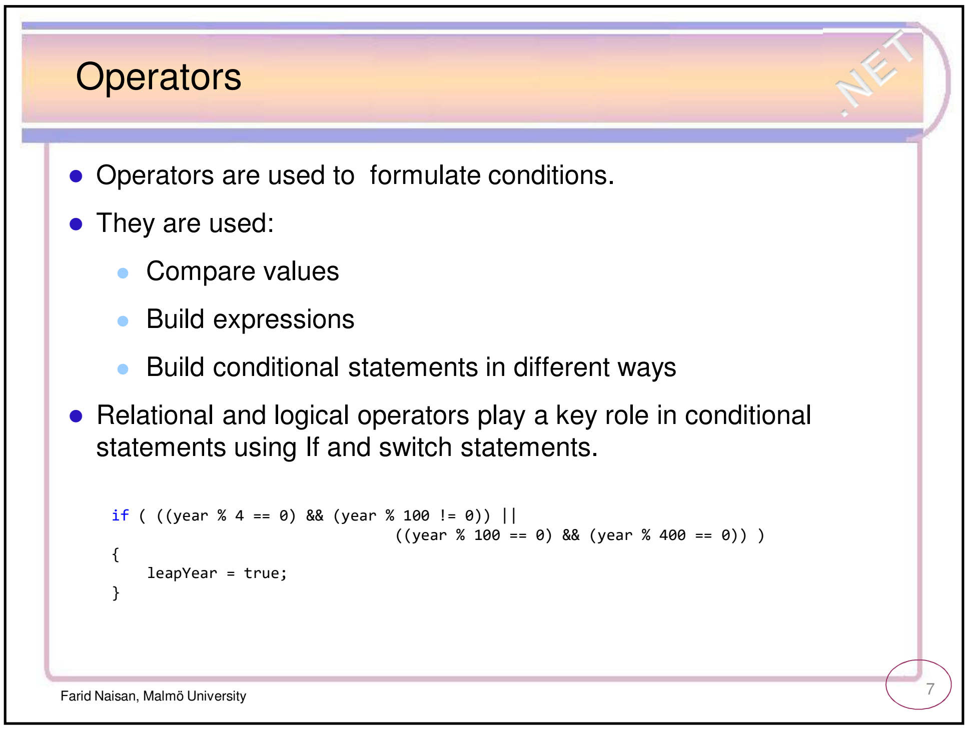
- All code executes sequentially. Selection algorithms (if-else) allow you to decide whether a section of code should be executed or not.
- C# has two types of statements for making decisions:
  - if
    - If-else
    - If-else if – else if…else
  - switch

```
//A simple example
private int Max (int a, int b)
{
    if (a > b)
       return a;
    else
       return b;
}
```

Iteration algorithms

- Loops are useful when certain steps need to be taken repeatedly.
- Iterations are performed either for a certain number of times, as long as some condition(s) prevail, or until some condition is met.
- Example: Write blank lines to the console window
  - The number of lines is input to the method
  - Count from 0 to number of lines -1:
    - Write a line with no character

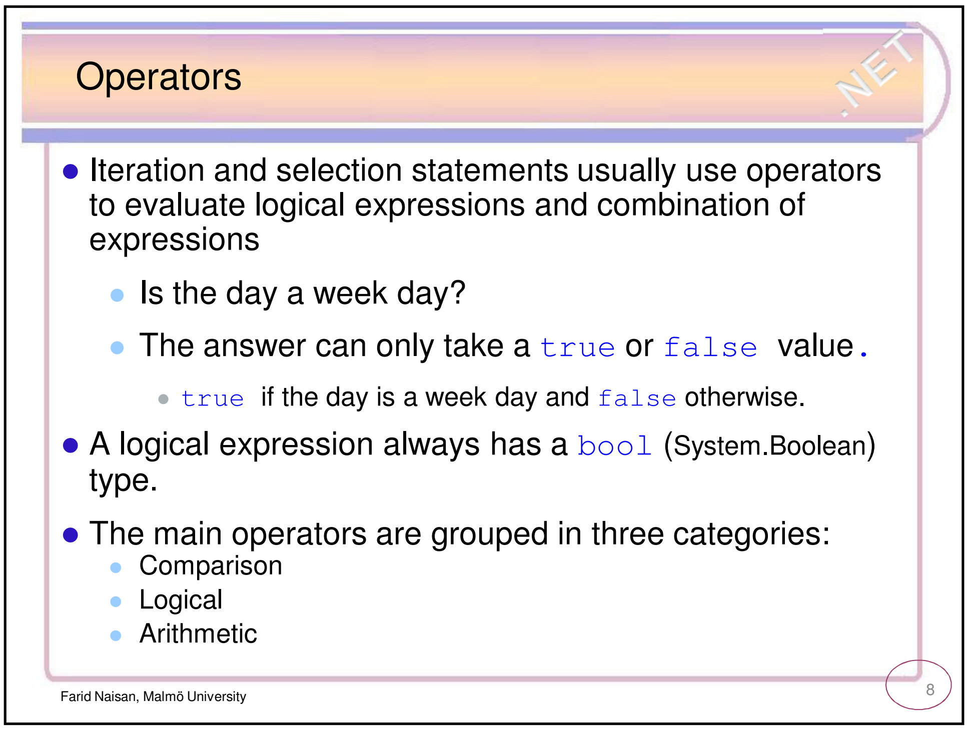
```
public void WriteBlankLines(int num)
{
    int counter;

    for ( counter = 0; counter < num; counter++)
    {
        Console.WriteLine();  //blank lines
    }
}
```



Operators

- Operators are used to formulate conditions.
- They are used:
  - Compare values
  - Build expressions
  - Build conditional statements in different ways
- Relational and logical operators play a key role in conditional statements using If and switch statements.

```
if ( ((year % 4 == 0) && (year % 100 != 0)) ||
                                ((year % 100 == 0) && (year % 400 == 0)) )
{
    leapYear = true;
}
```



Operators

- Iteration and selection statements usually use operators to evaluate logical expressions and combination of expressions
  - Is the day a week day?
  - The answer can only take a `true` or `false` value.
    - `true` if the day is a week day and `false` otherwise.
- A logical expression always has a `bool` (System.Boolean) type.
- The main operators are grouped in three categories:
  - Comparison
  - Logical
  - Arithmetic

Comparison operatorer

- Comparison operators are used to compare two values of same type.
- The result of the comparison of two Boolean expressions can only be `true` or `false`.

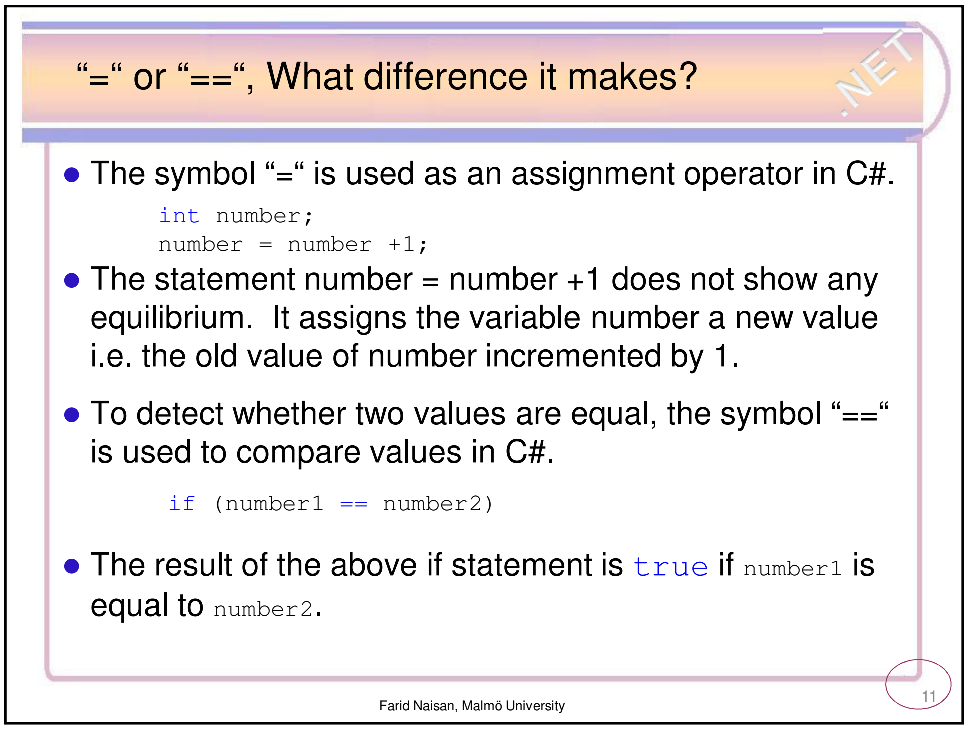
Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

- Notice that the double character operators do not have a blank space between the characters.

Comparison operators - example

- Assume: x = 5, y = 7
- The results of the operations are either `true` or `false`.

Expression	Description	**Result**
x > y	Is x greater than y?	`false`
x < y	Is x less than y?	`true`
x >= y	Is x greater than or equal to y?	`false`
x <= y	Is x less than or equal to y?	`true`
x == y	Is x equal to y?	`false`
x != y	Is x not equal to y?	`true`



"=" or "==", What difference it makes?

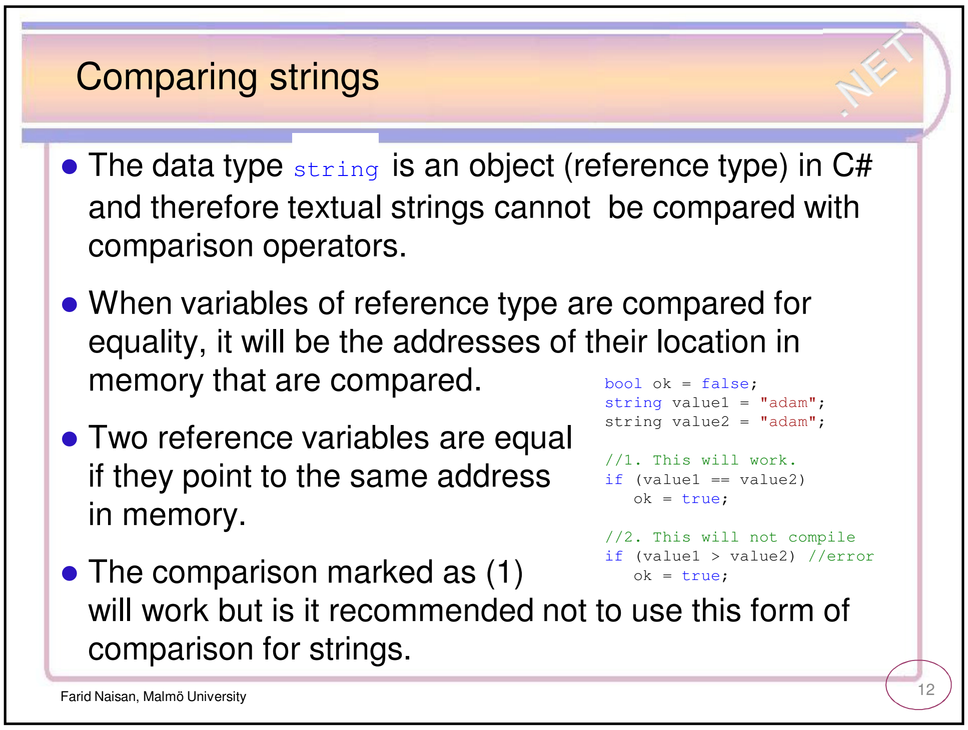
- The symbol "=" is used as an assignment operator in C#.

```
int number;
number = number +1;
```

- The statement number = number +1 does not show any equilibrium. It assigns the variable number a new value i.e. the old value of number incremented by 1.
- To detect whether two values are equal, the symbol "==" is used to compare values in C#.

```
if (number1 == number2)
```

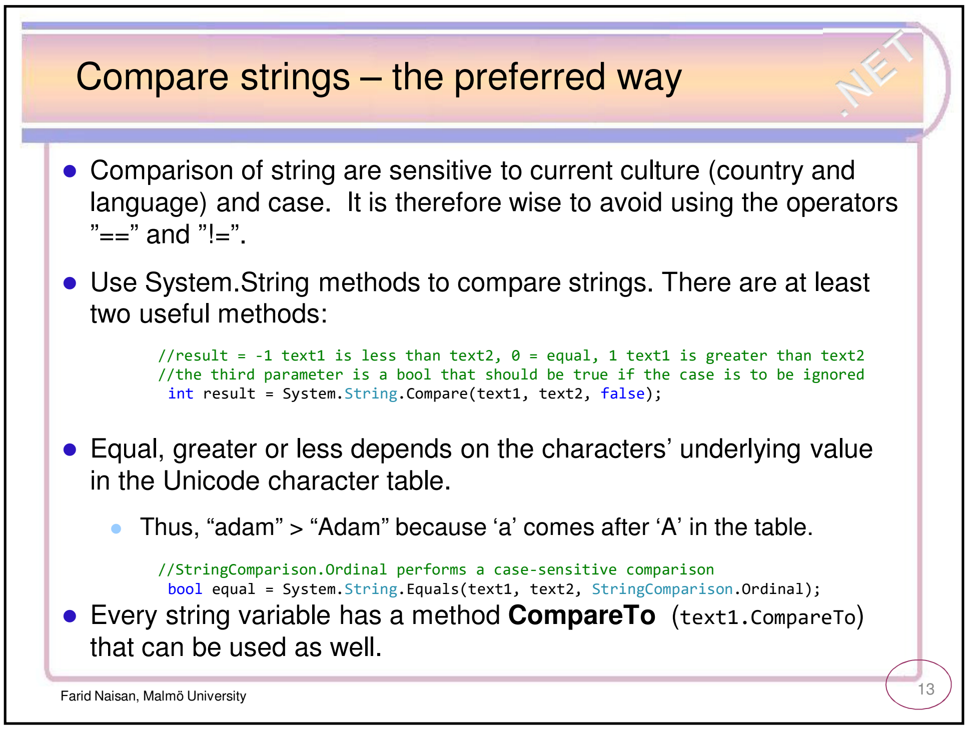
- The result of the above if statement is `true` if `number1` is equal to `number2`.



Comparing strings

- The data type `string` is an object (reference type) in C# and therefore textual strings cannot be compared with comparison operators.
- When variables of reference type are compared for equality, it will be the addresses of their location in memory that are compared.
- Two reference variables are equal if they point to the same address in memory.
- The comparison marked as (1) will work but is it recommended not to use this form of comparison for strings.

```
bool ok = false;
string value1 = "adam";
string value2 = "adam";

//1. This will work.
if (value1 == value2)
   ok = true;

//2. This will not compile
if (value1 > value2) //error
   ok = true;
```



Compare strings – the preferred way

.NET

- Comparison of string are sensitive to current culture (country and language) and case. It is therefore wise to avoid using the operators ”==” and ”!=”.
- Use System.String methods to compare strings. There are at least two useful methods:

```
//result = -1 text1 is less than text2, 0 = equal, 1 text1 is greater than text2
//the third parameter is a bool that should be true if the case is to be ignored
 int result = System.String.Compare(text1, text2, false);
```

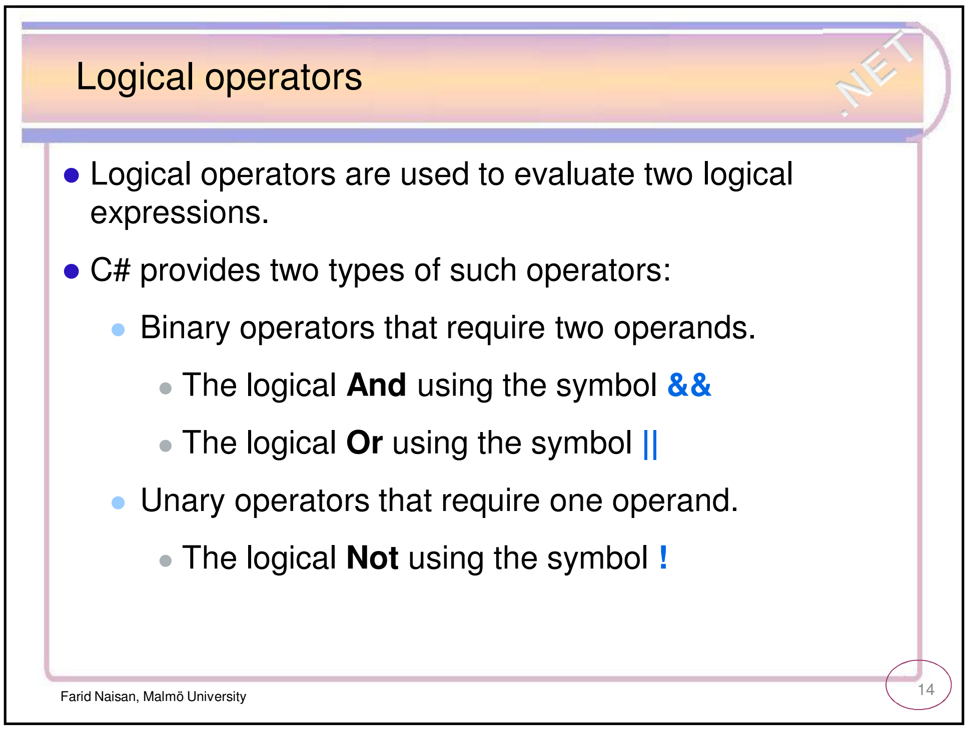
- Equal, greater or less depends on the characters’ underlying value in the Unicode character table.
  - Thus, “adam” > “Adam” because ‘a’ comes after ‘A’ in the table.

```
//StringComparison.Ordinal performs a case-sensitive comparison
 bool equal = System.String.Equals(text1, text2, StringComparison.Ordinal);
```

- Every string variable has a method **CompareTo** (`text1.CompareTo`) that can be used as well.

Farid Naisan, Malmö University

13



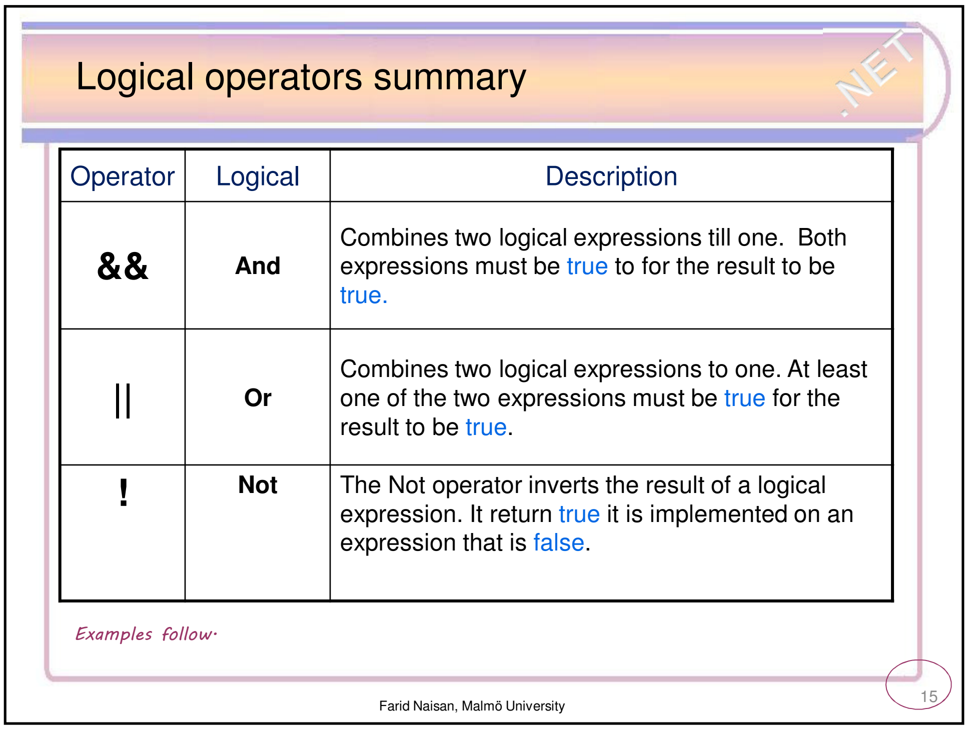
Logical operators

.NET

- Logical operators are used to evaluate two logical expressions.
- C# provides two types of such operators:
  - Binary operators that require two operands.
    - The logical **And** using the symbol **&&**
    - The logical **Or** using the symbol **||**
  - Unary operators that require one operand.
    - The logical **Not** using the symbol **!**

Farid Naisan, Malmö University

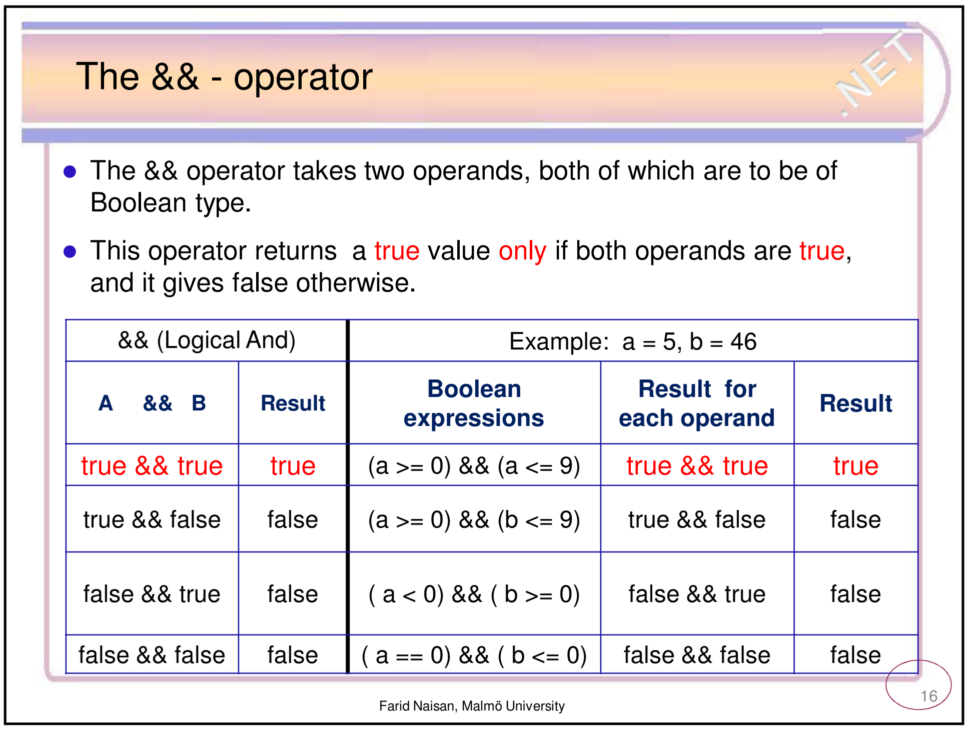
14



Logical operators summary

Operator	Logical	Description
&&	**And**	Combines two logical expressions till one. Both expressions must be true to for the result to be true.
||	**Or**	Combines two logical expressions to one. At least one of the two expressions must be true for the result to be true.
!	**Not**	The Not operator inverts the result of a logical expression. It return true it is implemented on an expression that is false.

Examples follow.



The && - operator

- The && operator takes two operands, both of which are to be of Boolean type.
- This operator returns a true value only if both operands are true, and it gives false otherwise.

&& (Logical And)		Example: a = 5, b = 46		
A && B	**Result**	**Boolean expressions**	**Result for each operand**	**Result**
true && true	true	(a >= 0) && (a <= 9)	true && true	true
true && false	false	(a >= 0) && (b <= 9)	true && false	false
false && true	false	(a < 0) && (b >= 0)	false && true	false
false && false	false	(a == 0) && (b <= 0)	false && false	false

||-operator

- The logical Or operator also takes two operands of Boolean type.
- It returns false only if both operands are false; it gives true otherwise.

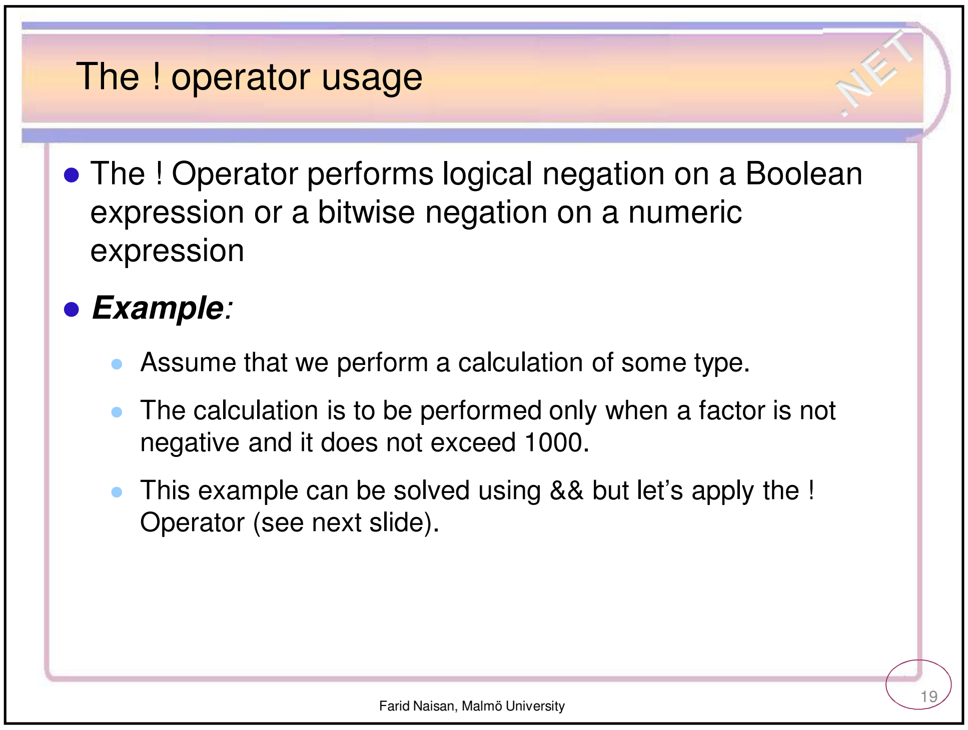
|| (Logical Or)		Example: a = 5, b = 46		
A || B	Resulta	Logical expressions	Result for every operand	Result
true || true	true	(a >= 0) || (a <= 9)	true || true	true
true || false	true	(a >= 0) || (b <= 9)	true || false	true
false || true	true	(a < 0) || (b >= 0)	false || true	true
false || false	**false**	(a == 0) || (b <= 0)	**false|| false**	**false**

The ! operator

- The ! (Not) operator takes one expression of Boolean type.
- It returns false if the expression is true and vice versa.

Logical expression A	**Result for !A**
true	**false**
false	**true**

Exempel: a = 5, b = 46		
Boolean expression	**Result for the operand**	**Result**
!(a >= 0)	!true	false
!(b <= 9)	!false	true

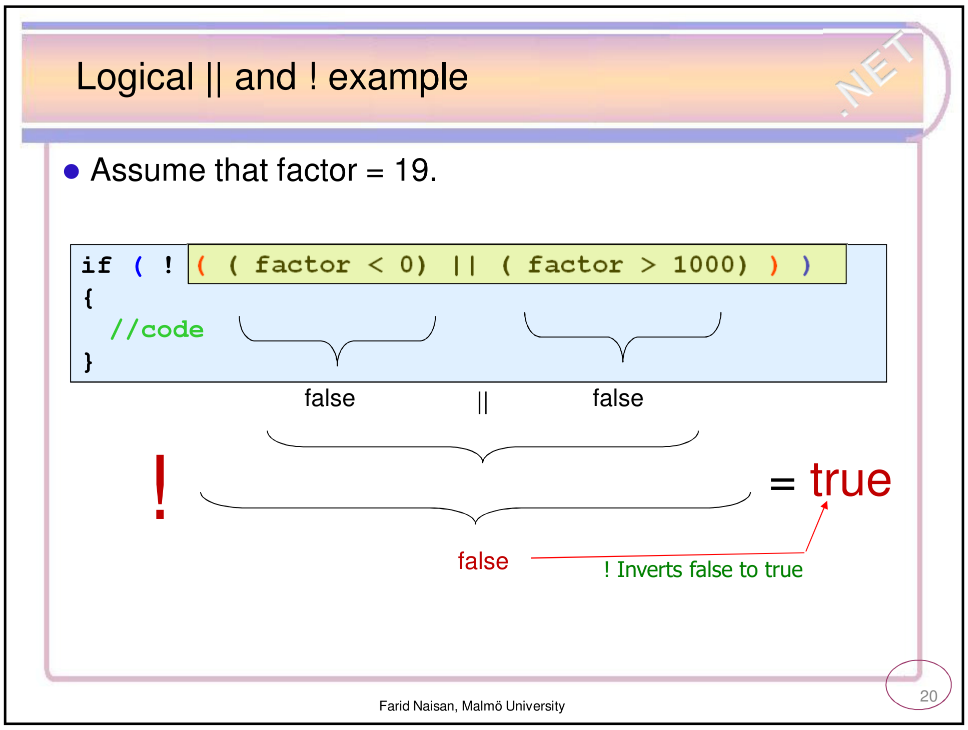


The ! operator usage

- The ! Operator performs logical negation on a Boolean expression or a bitwise negation on a numeric expression
- ***Example***:
  - Assume that we perform a calculation of some type.
  - The calculation is to be performed only when a factor is not negative and it does not exceed 1000.
  - This example can be solved using && but let's apply the ! Operator (see next slide).

Farid Naisan, Malmö University

19



Logical || and ! example

- Assume that factor = 19.

```
if ( ! ( ( factor < 0) || ( factor > 1000) ) )
{
   //code
}
```

Farid Naisan, Malmö University

20

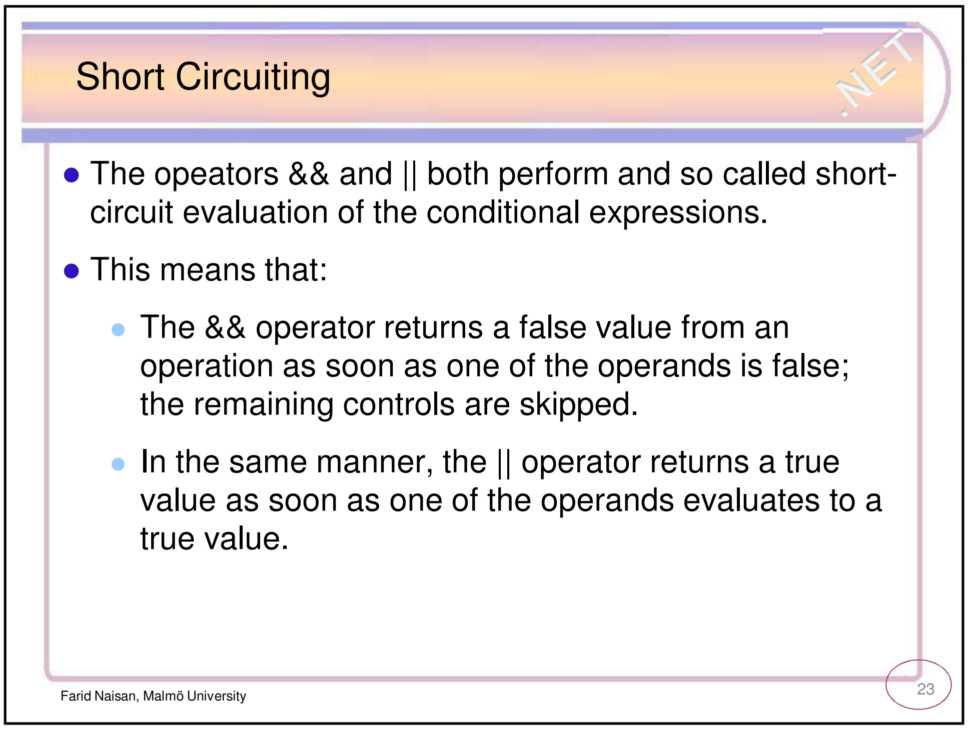
Arithmetic operators

Operator	Meaning	Example
+	Addition	`sum = price + tax;`
–	Subtraction	`cost = price – tax;`
*	Multiplication	`tax = cost * rate;`
/	Division	`salePrice = price / 2;`
%	Modulus	`remainder = sum % 5;`

The modulus operator (%)

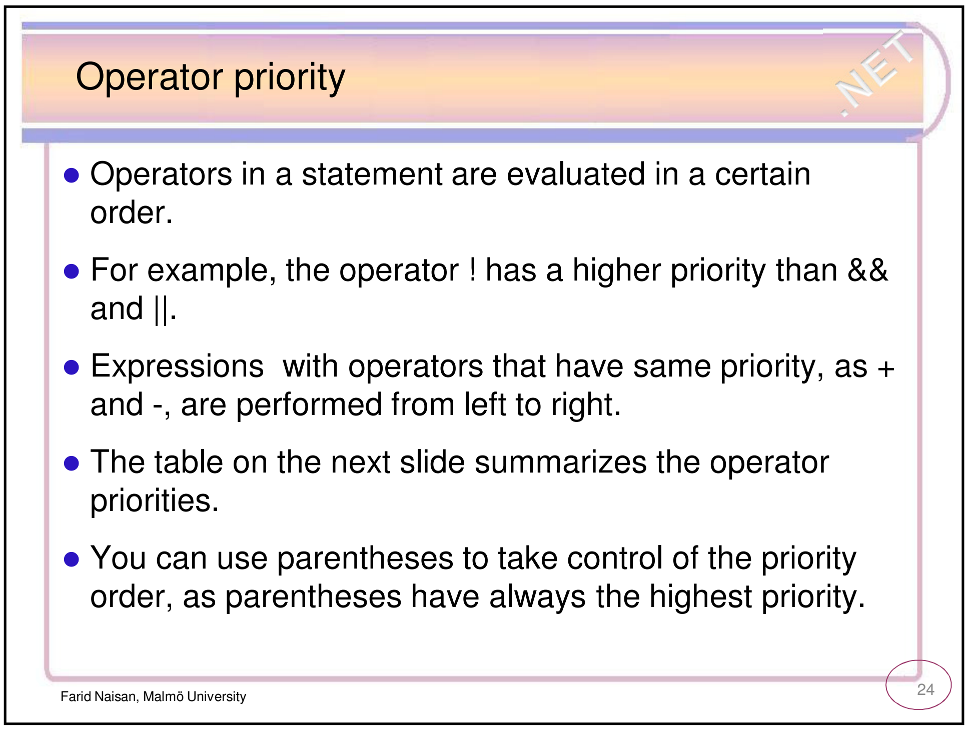
- The % operator returns the remainder after an integer division.
- In arithmetic, very fractional number has an integer whole-part and a fraction part consisting of an integer numerator and an integer denominator.
  - 16/5 = 3 1/5 – in this example, 3 is the whole part, 1 is the nominator and 5 is the denominator.
- The % returns the **nominator** which is also the remainder:

```
15 % 5 = 0 (15/5 = 3 0/5, remainder = 0)
16 % 5 = 1 (16/5 = 3 1/5, remainder = 1)
```



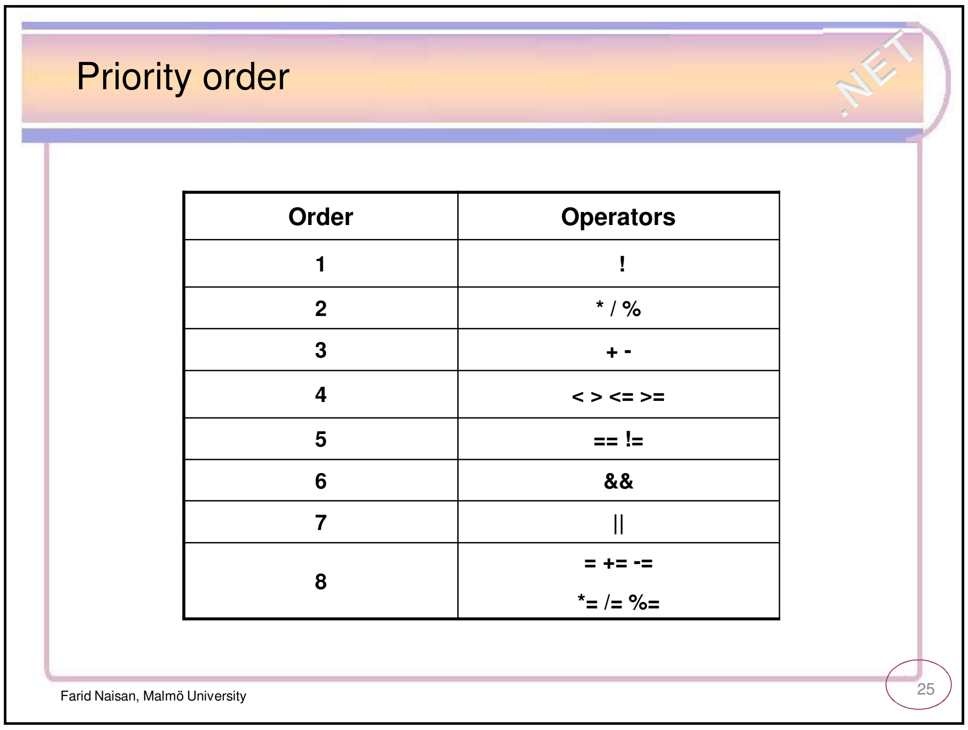
Short Circuiting

- The opeators && and || both perform and so called short-circuit evaluation of the conditional expressions.
- This means that:
  - The && operator returns a false value from an operation as soon as one of the operands is false; the remaining controls are skipped.
  - In the same manner, the || operator returns a true value as soon as one of the operands evaluates to a true value.



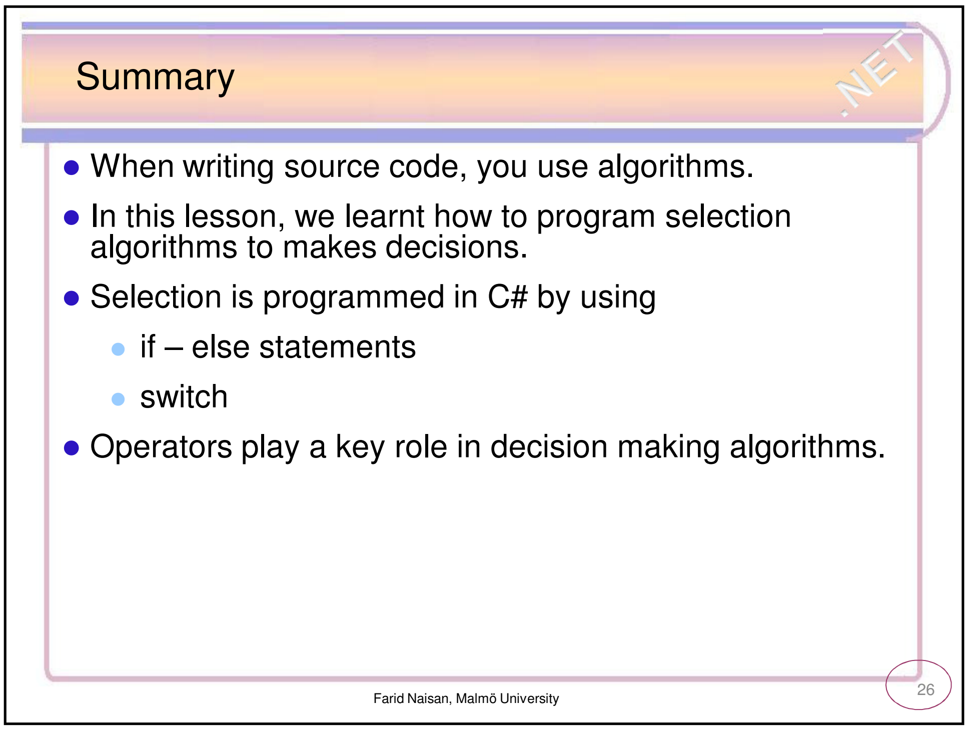
Operator priority

- Operators in a statement are evaluated in a certain order.
- For example, the operator ! has a higher priority than && and ||.
- Expressions with operators that have same priority, as + and -, are performed from left to right.
- The table on the next slide summarizes the operator priorities.
- You can use parentheses to take control of the priority order, as parentheses have always the highest priority.



Priority order

Order	Operators
1	!
2	* / %
3	+ -
4	< > <= >=
5	== !=
6	&&
7	||
8	= += -= *= /= %=



Summary

- When writing source code, you use algorithms.
- In this lesson, we learnt how to program selection algorithms to makes decisions.
- Selection is programmed in C# by using
  - if – else statements
  - switch
- Operators play a key role in decision making algorithms.

C# Programming - For Beginners

Decision Making using Selection Algorithms

.NET

Contents:
- **if** statements
- **switch** statements

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

Selection algorithms

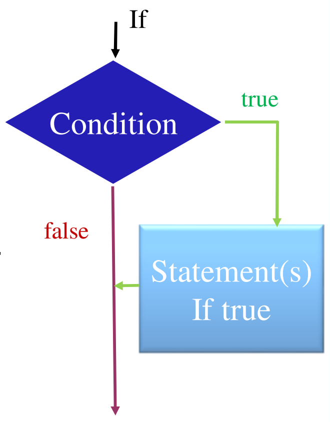
.NET

- All code executes sequentially. Selection algorithms (if-else) allow you to decide whether a section of code should be executed or not.
- This algorithm is used when making choices between two or more alternatives.
- Example: Check user input for yes/no
  - Let user give an answer
  - If the answer is 'Y' or 'y'
    - Product is a food item
  - Otherwise
    - Product is not a food item

```
private void ReadIfFoodItem()
{
    Console.Write("Food item (y/n): ");

    char response = char.Parse(Console.ReadLine());

    if ( (response == 'y') || (response == 'Y') )
        foodItem = true;
    else
        foodItem = false;
}
```

The Decision Structure

- The figure shows the flowchart for a typical conditional structure.
- Evaluate the condition and make a decision
  - *Is the divisor other than zero?*
  - *If yes (true), perform the division.*
  - *Otherwise (false), skip the division.*
- C# has two types of statements for making decisions:
  - `if`
    - `If-else`
    - `If-else if – else if…else`
  - `switch`



The if statements

- The simplest form of **if** statement is to execute a block of code only if a condition is true.

```
public bool ValidateToPositiveValue(double amount)
{
    bool valid = false; //initialization

    if (amount > 0.0)
    {
        valid = true;

        // more code here if necessary
    }
    return valid;
}
```

- The block of code following the if-statement and enclosed by the start brace '{' and the end brace '}' is executed only when the condition "amount > 0.0 " is true.

If statement syntax rules

- The condition must be enclosed inside a pair of normal parenthesis.
- If the block of code after the if-statement contains more than one statement, it should have a start and ending curly brace.
- No semicolon character should be placed after the condition expression.

Code block that executes only if the condition is evaluated to a true value.

```
if (condition)
{


}
```

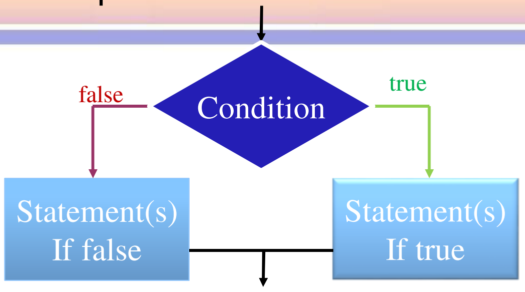
The if – else statements

- The **if** statement will execute or it ignores a group of statements.
- The **if…else** construct will execute one of two alternative groups of statements.

```
/*An if-else construction.
Either the if block or the
else block will be executed.*/
if (condition is true)
{
   statement(s) to
   be executed
}
else
{
   //other statement(s) to
   //be executed
}
```

```
/*The two code blocks here do not
represent an if-else structure. These
are 2 separate if structures independent
of each other.*/
if (condition is true)
{
   statement(s) to
   be executed
}

if (condition is false)
{
   //other statement(s) to
   //be executed
}
```

The if - else Example



```
public double Divide(double dividend, double divisor)
{
    double result = 0.0;

    if (divisor != 0.0)
    {
        result = dividend / divisor;
    }
    else
    {
        System.Windows.Forms.MessageBox.Show("The divisor cannot zero. It causes division by zero!");
    }

    return result;
}
```

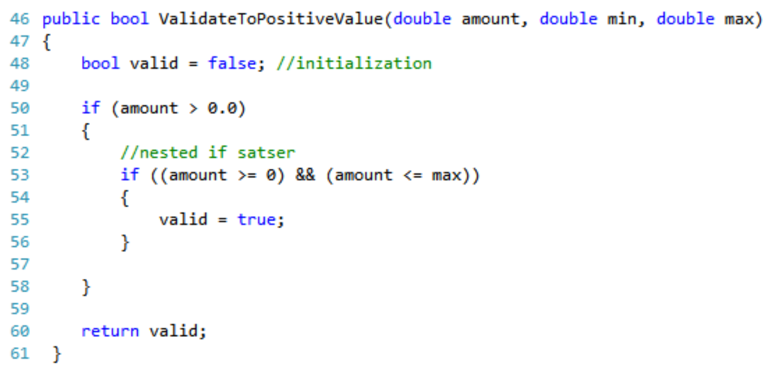
Else-If for testing multiple conditions

- If you want to check more than one condition at the same time, you can use several else if (2 words) blocks.

```
if (condition1 is true)
{
      //statement(s) to be executed
}
else if (condition2 is true)
{
      //other statement(s) to be executed
}
else if (condition3 is true)
{
      //other statement(s) to be executed
}
else //default case (optional)
{
      //if none of the above conditions are true
      //this block of statement(s)is to be executed
}
```

Nested If statements

- If-statements can be nested, i.e. can be contained inside other If-statements.
- There is no limitation on the number of nested blocks.



```
public bool ValidateToPositiveValue(double amount, double min, double max)
{
    bool valid = false; //initialization

    if (amount > 0.0)
    {
        //nested if satser
        if ((amount >= 0) && (amount <= max))
        {
            valid = true;
        }

    }

    return valid;
 }
```

Switch statements

- The switch statement is a decision-making construct that simplifies selecting among multiple values.
- While if-else can be used to test a single or multiple conditions, switch statements can be used when comparing one expression to multiple values.
- This form works well on the value of a variable when there is only a limited number of values that the variable can have.
    - A variable day intended to store days of the week can only have seven values.
- Switch choices are determined by the value of an expression called a **selector** (a variable or a value that is to be compared to a number of other values).

switch Syntax

- The general form of the switch block is shown by this simple example.
- The break-statement is important after each case statement. Otherwise, the execution will drop into the next case.
- The default case is implemented only when none of the other cases match the selector variable (day). The default case is optional.

```
switch (day)
{
    case "Monday":
        //code
        break;

    case "Tuesday":
    case "Thursday":
        //code;
        break;

    default:
        //code
        break;
}
```

switch

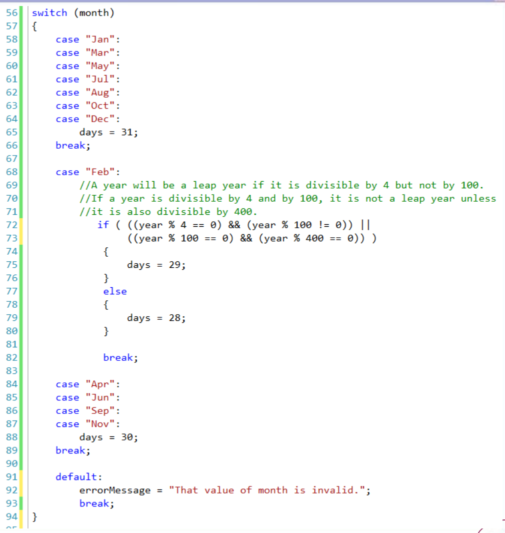
- The **switch** statement uses a selector that can be of the types short, int, long, char or a string as the variable `value` in the example below:

```
switch (value)     //to be compared
{
    case caseValue1: //constant value that is compared to value.
       // statements
       break;   //Necessary to break the case statement.

       case caseValue2:
    ...
}
```

- `caseValue1`, etc. must be a unique constant value of the same type as the selector, i.e. `value`.

Switch Example

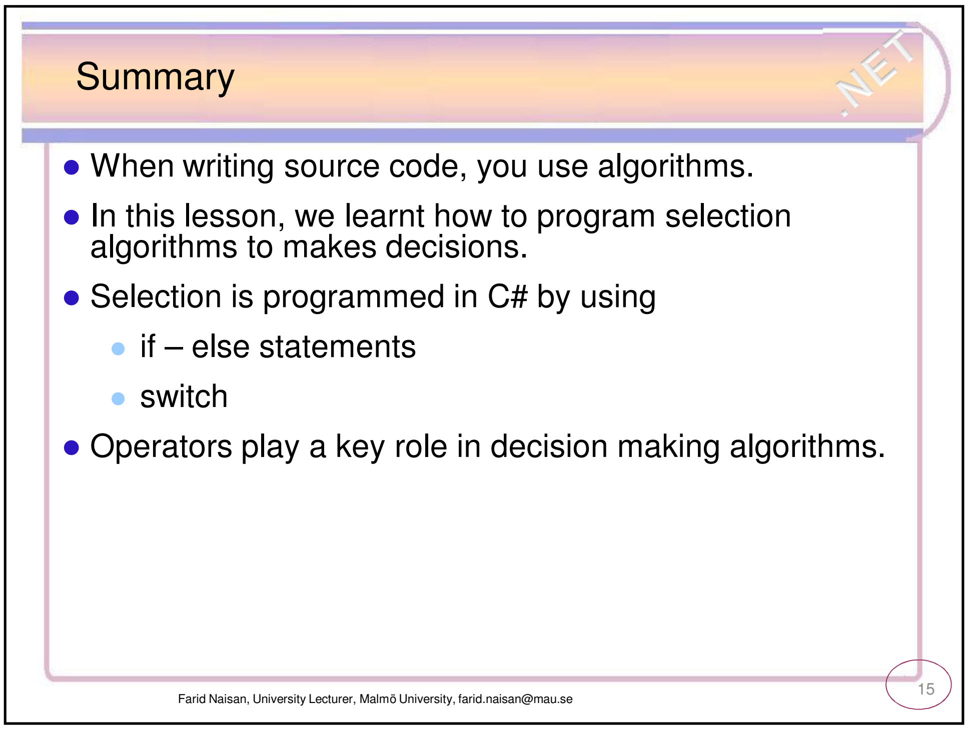
- The code on line 65 will execute if the value of month (string) matches any of the case values listed above the line.
- All code lines from number 69 to 82 will execute when **month** = “Feb”.



```
56  switch (month)
57  {
58      case "Jan":
59      case "Mar":
60      case "May":
61      case "Jul":
62      case "Aug":
63      case "Oct":
64      case "Dec":
65          days = 31;
66      break;
67
68      case "Feb":
69          //A year will be a leap year if it is divisible by 4 but not by 100.
70          //If a year is divisible by 4 and by 100, it is not a leap year unless
71          //it is also divisible by 400.
72             if ( ((year % 4 == 0) && (year % 100 != 0)) ||
73                  ((year % 100 == 0) && (year % 400 == 0)) )
74              {
75                  days = 29;
76              }
77              else
78              {
79                  days = 28;
80              }
81
82              break;
83
84      case "Apr":
85      case "Jun":
86      case "Sep":
87      case "Nov":
88          days = 30;
89      break;
90
91      default:
92          errorMessage = "That value of month is invalid.";
93          break;
94  }
```

Jump statements

- A switch statement can also be broken by using the keyword return.
- If you omit a jump statement, the execution will continue to the next case!
  - However, the C# compiler issues a compilation error.
- Never use a goto statement although it, unfortunately, is supported by C# and can be used as a jump statement.
- The goto statement makes your code difficult to read and debug. *It is strictly forbidden to use goto in this course!*



Summary

- When writing source code, you use algorithms.
- In this lesson, we learnt how to program selection algorithms to makes decisions.
- Selection is programmed in C# by using
  - if – else statements
  - switch
- Operators play a key role in decision making algorithms.

Programming Using C#,
Basic Course

Loops

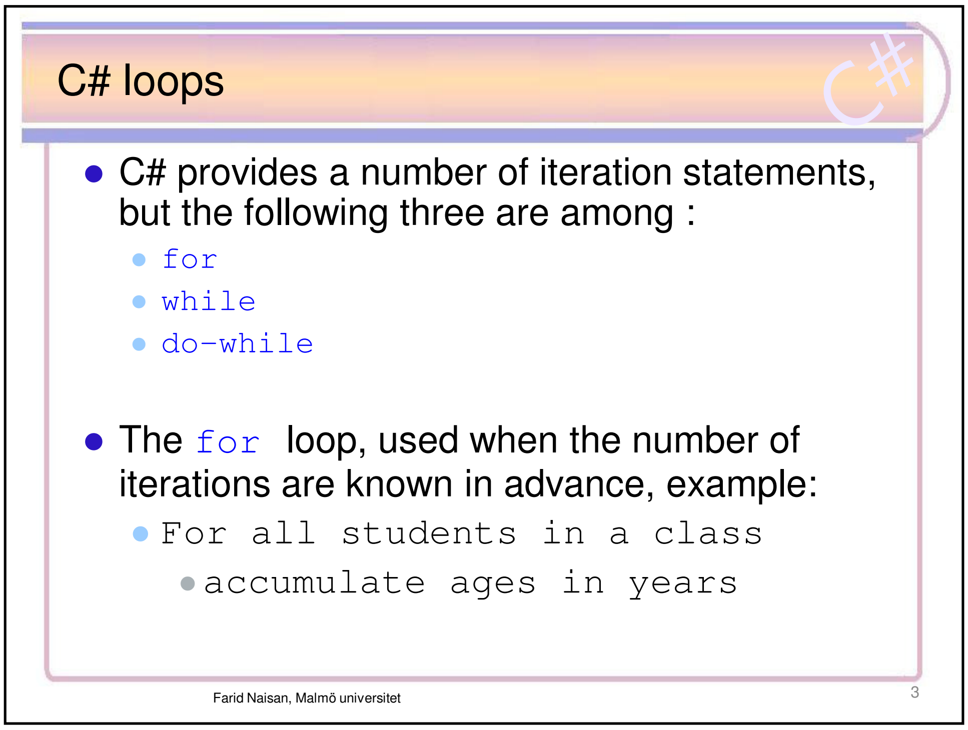
Agenda:

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

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

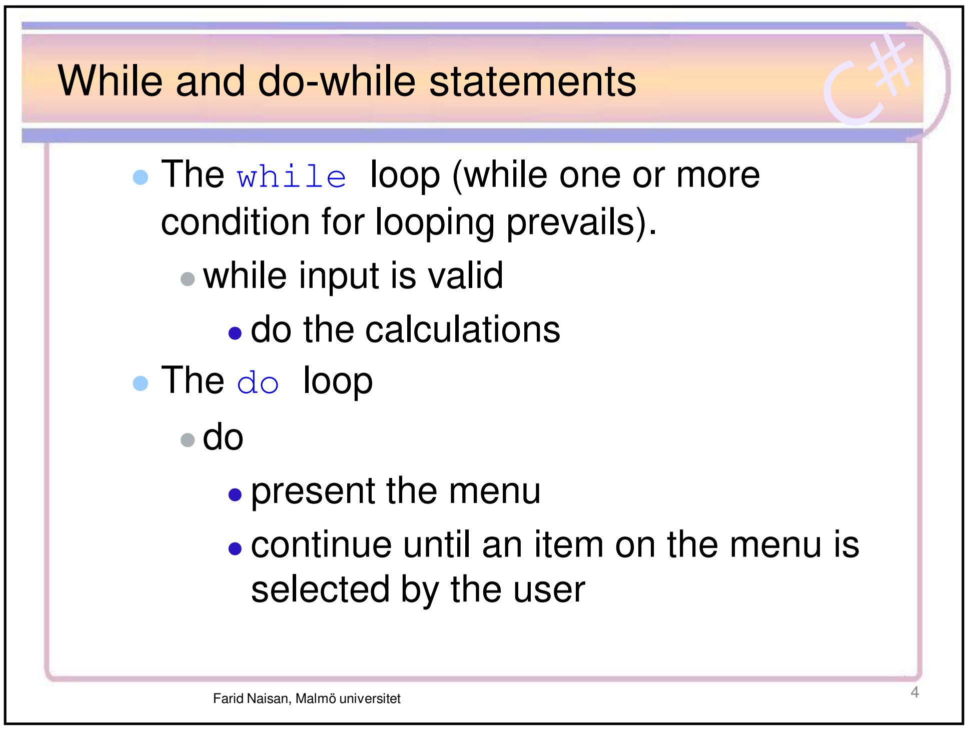
Iterations

- A sequence of statements that repeat themselves exactly or with minor modifications can be coded in form of loops.
- Iterations are used whenever statements are to be executed repeatedly either:
  - for a certain number of times, or
  - while a certain control condition prevails.
- By good programming, loops terminate subject to some criteria.



C# loops

- C# provides a number of iteration statements, but the following three are among :
  - `for`
  - `while`
  - `do-while`
- The `for` loop, used when the number of iterations are known in advance, example:
  - `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 prevails).
  - 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# provides easier ways to increment and decrement a variable's value.
  - There are two operators ++ and -- used for incrementing decrementing, respectively.

```
count++; //same as count = count+1;
count--; //same as count = count-1;
```

- Note: No black space between the ++ and -- symbols.

Postfix or prefix

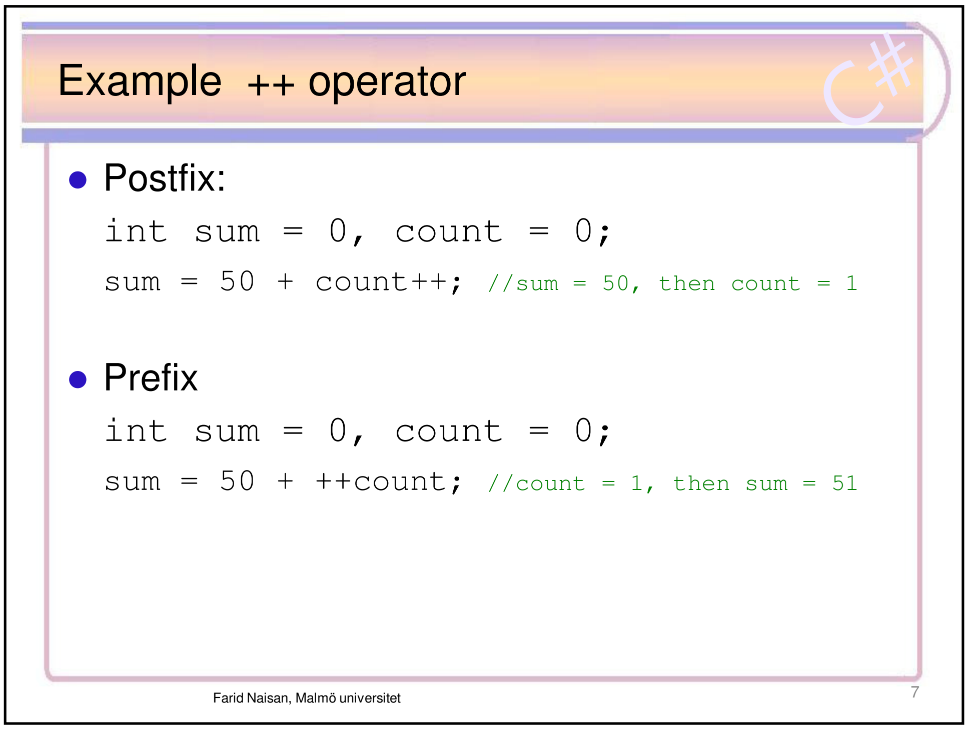
- The ++ or -- unary operators can be applied in two forms, either in the **pos**tfix form as in:

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

  or the **pre**fix 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.
- In the prefix form, the variable is first incremented or decremented, and then the value is used.



Example ++ operator

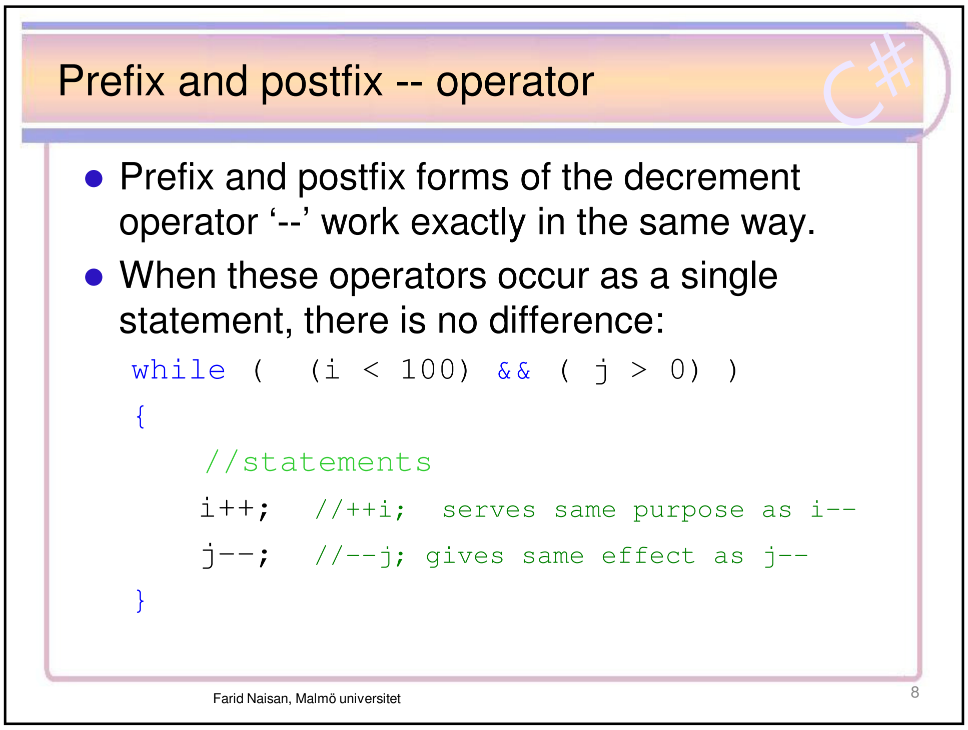
- Postfix:

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

- Prefix

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

Farid Naisan, Malmö universitet 7

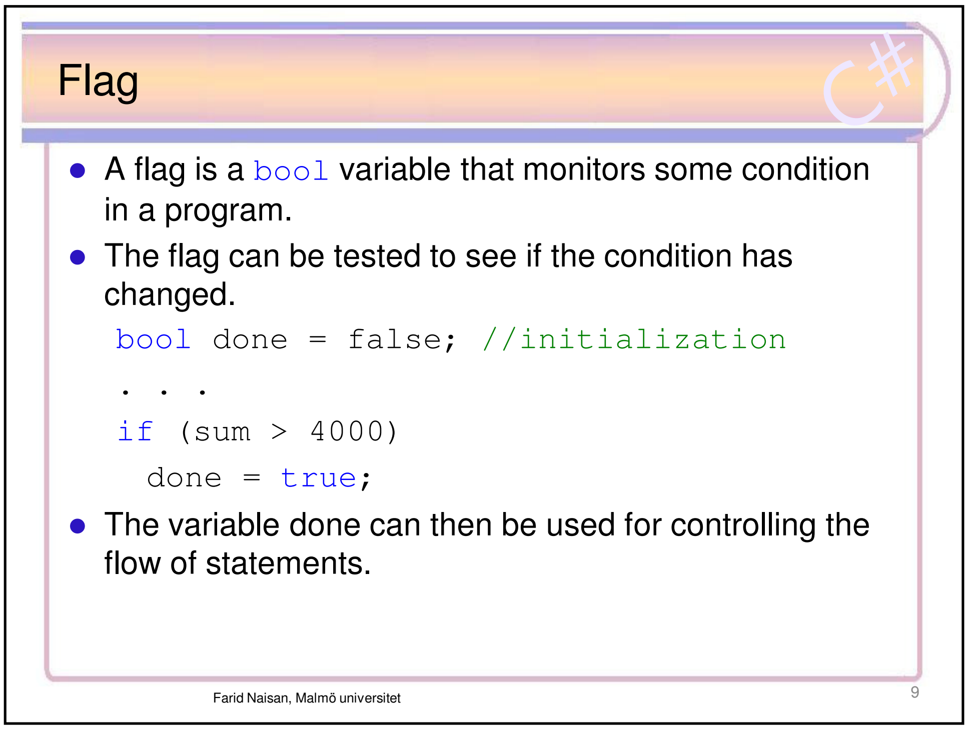


Prefix and postfix -- operator

- Prefix and postfix forms of the decrement operator '--' work exactly in the same way.
- When these operators occur as a single statement, there is no difference:

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

Farid Naisan, Malmö universitet 8

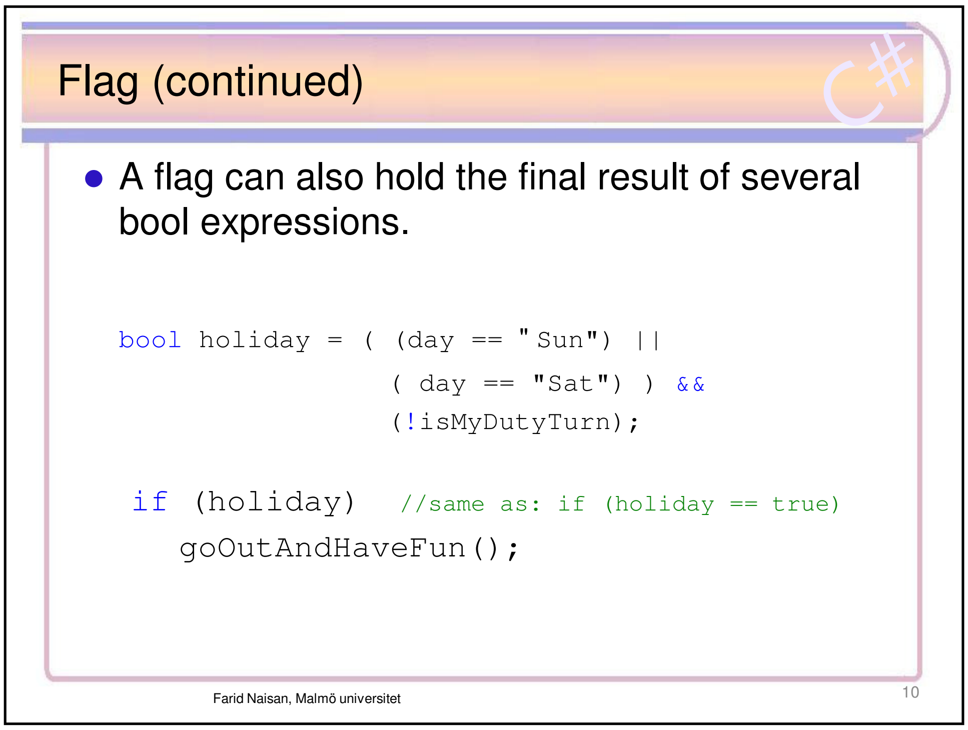


Flag

- 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 controlling the flow of statements.



Flag (continued)

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

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

 if (holiday)   //same as: if (holiday == true)
    goOutAndHaveFun();
```

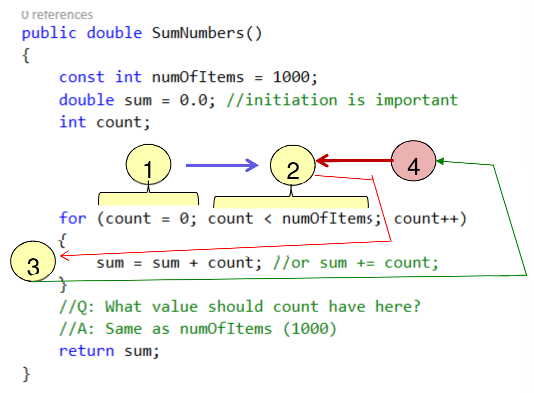
The for-loop

- 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 have any value, example: count++; count += 5.

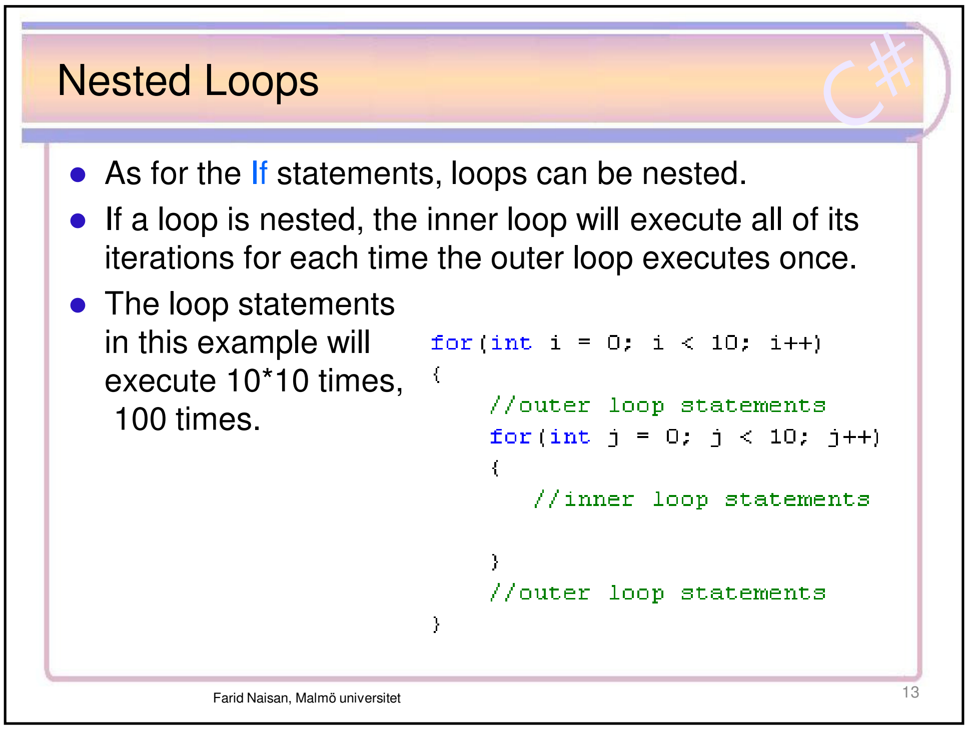
How a For-statement is executed

1. initialization of the counter – performed only once at the beginning
2. evaluation of the condition to continue with the next iteration or exit the loop
3. execution of the body of the loop – all code between { and } will be executed.
4. The value of the counter is incremented.

The condition is evaluated again: steps 2 to 4 is repeated after each iteration. The loop terminates when the condition is false (when count = numOfItems



```
0 references
public double SumNumbers()
{
    const int numOfItems = 1000;
    double sum = 0.0; //initiation is important
    int count;

    for (count = 0; count < numOfItems; count++)
    {
        sum = sum + count; //or sum += count;
    }
    //Q: What value should count have here?
    //A: Same as numOfItems (1000)
    return sum;
}
```

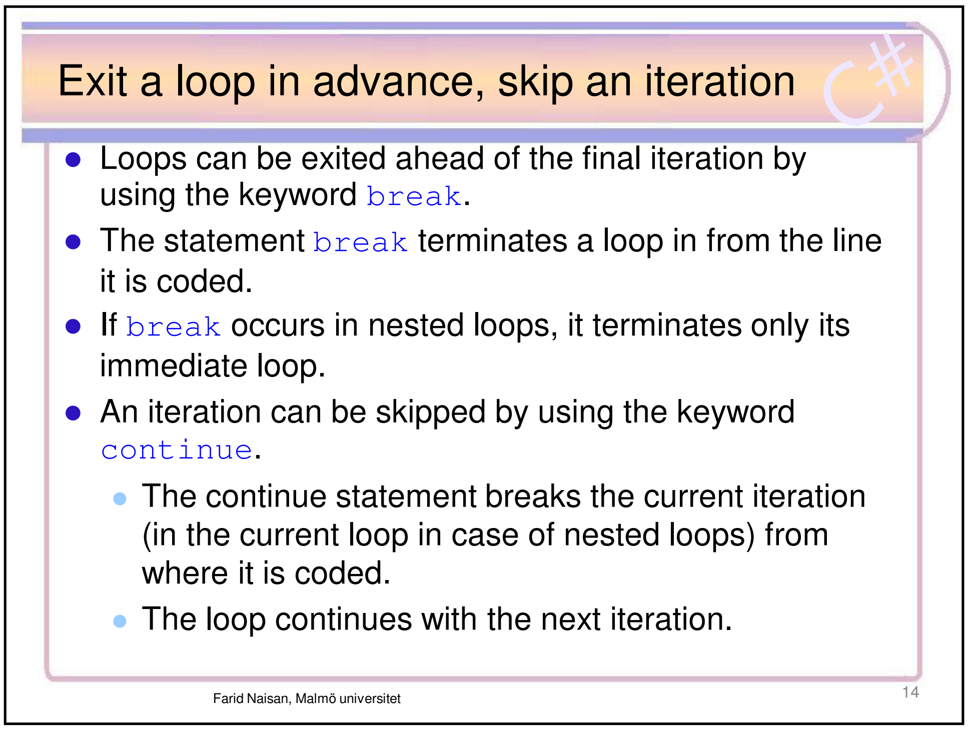


Nested Loops

- 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 10*10 times, 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
}
```

Farid Naisan, Malmö universitet 13

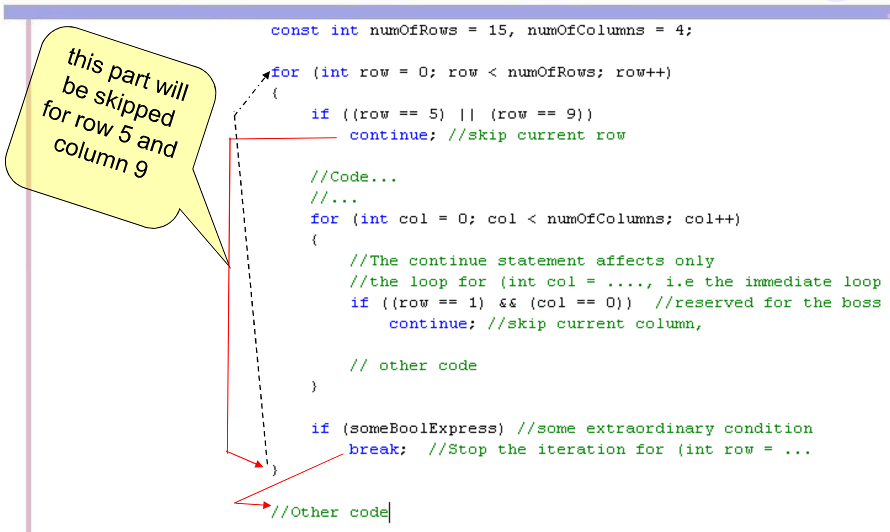


Exit a loop in advance, skip an iteration

- Loops can be exited ahead of the final iteration by using the keyword `break`.
- The statement `break` terminates a loop in from the line it is coded.
- If `break` occurs in nested loops, it terminates only its immediate loop.
- An iteration can be skipped by using the keyword `continue`.
  - The continue statement breaks the current iteration (in the current loop in case of nested loops) from where it is coded.
  - The loop continues with the next iteration.

Farid Naisan, Malmö universitet 14

Continue and break example

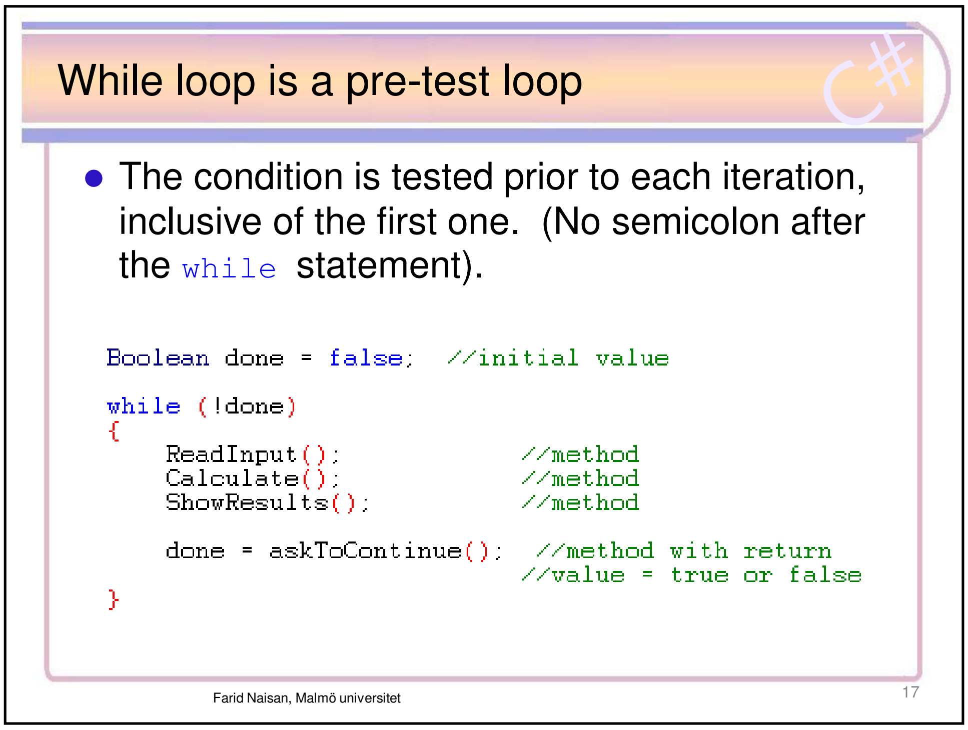


```
const int numOfRows = 15, numOfColumns = 4;

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 loops

- 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.
- The condition must be inside parenthesis.

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

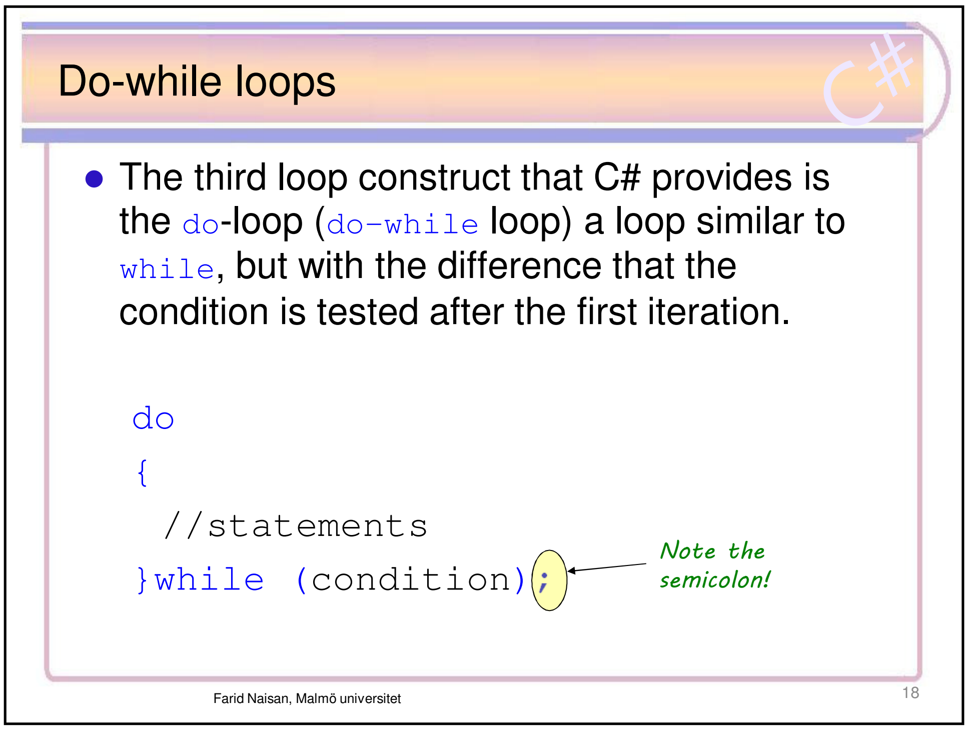


While loop is a pre-test loop

- The condition is tested prior to each iteration, inclusive of 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
}
```

Farid Naisan, Malmö universitet 17



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);
```

Note the semicolon!

Farid Naisan, Malmö universitet 18

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);
```

Farid Naisan, Malmö universitet 19

Nested loops

- `While` as well as `do-while` loops can be nested exactly as in `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.

Farid Naisan, Malmö universitet 20

Infinite loops

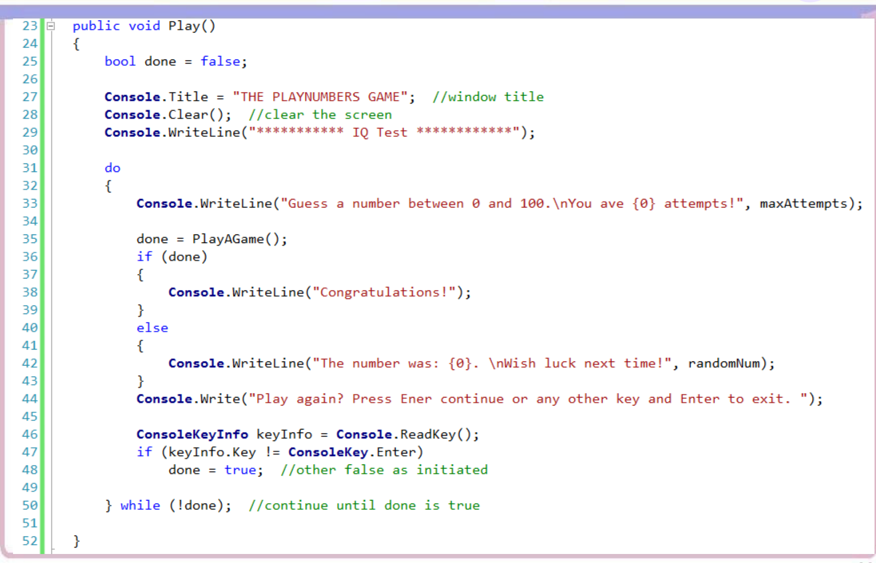
- It is quite important that in a while and do loop the condition does not constantly hold the same value. This can easily happen as a common mistake.

```
int i = 0 sum =0;
while (i < 1000)
{
    sum += i;
}
```

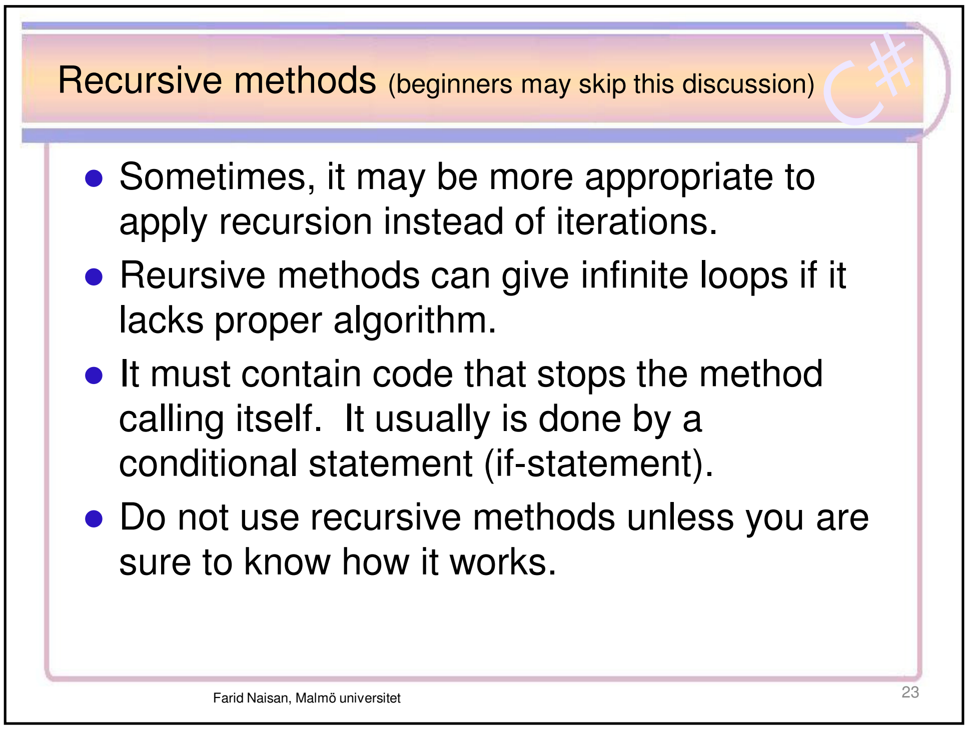
```
int i = 0 sum =0;
while (i < 1000)
{
    sum += i;
    i++;
}
```

- The left 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 code block (as in the right loop) will remedy the problem and the loop will terminate when i = 1000.

Another Loop Example

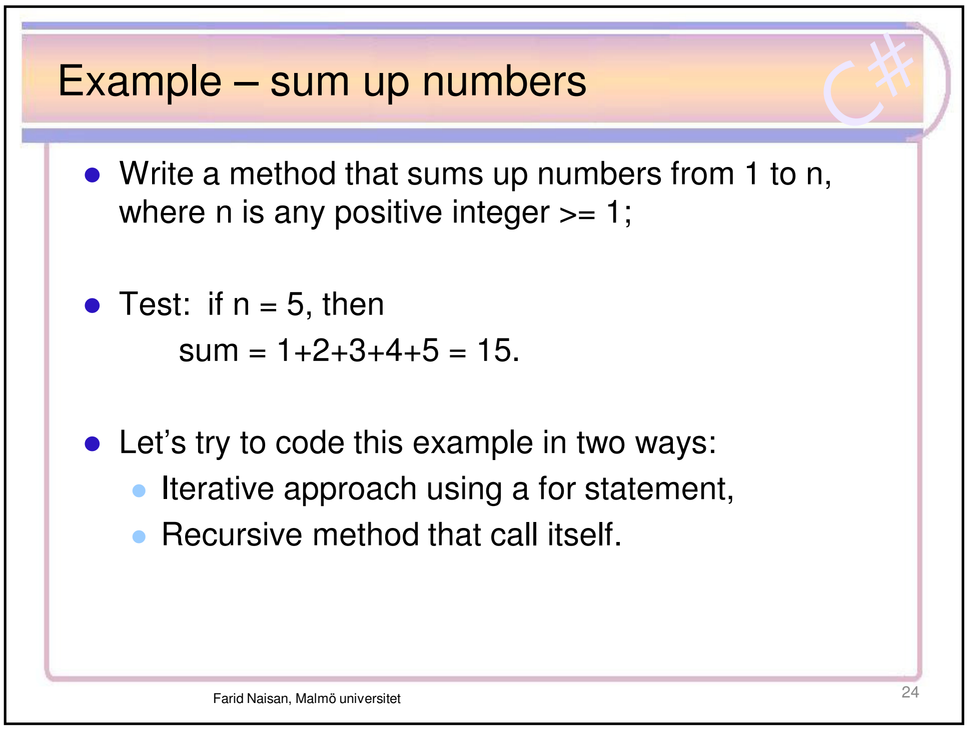


```
public void Play()
{
    bool done = false;

    Console.Title = "THE PLAYNUMBERS GAME";  //window title
    Console.Clear();  //clear the screen
    Console.WriteLine("*********** IQ Test ************");

    do
    {
        Console.WriteLine("Guess a number between 0 and 100.\nYou ave {0} attempts!", maxAttempts);

        done = PlayAGame();
        if (done)
        {
            Console.WriteLine("Congratulations!");
        }
        else
        {
            Console.WriteLine("The number was: {0}. \nWish luck next time!", randomNum);
        }
        Console.Write("Play again? Press Ener continue or any other key and Enter to exit. ");

        ConsoleKeyInfo keyInfo = Console.ReadKey();
        if (keyInfo.Key != ConsoleKey.Enter)
            done = true;  //other false as initiated

    } while (!done);  //continue until done is true

}
```



Recursive methods (beginners may skip this discussion)

- Sometimes, it may be more appropriate to apply recursion instead of iterations.
- Reursive methods can give infinite loops if it lacks proper algorithm.
- It must contain code that stops the method calling itself. It usually is done by a conditional statement (if-statement).
- Do not use recursive methods unless you are sure to know how it works.



Example – sum up numbers

- Write a method that sums up numbers from 1 to n, where n is any positive integer >= 1;
- Test: if n = 5, then

  sum = 1+2+3+4+5 = 15.
- Let's try to code this example in two ways:
  - Iterative approach using a for statement,
  - Recursive method that call itself.

Iterative solution

```
//Sum up numbers from 0 to "number" using a for-loop

public int AddNumbers(int number)
{
    int sum = 0;
    for (int count = 0; count < number; count++ )
    {
        sum += count;
    }
    return sum;
}
```

Here is what happens to sum when the loop is executed.

count	sum
0	0
1	0 + 1 = 1
2	1 + 2 = 3
3	3 + 3 = 6
4	6 + 4 = 10
5	10 + 5 = 15

Recursive methods

- In a recursive operation, the 1st call must wait until the 2nd call is evaluated so a return value is acquired, and the second call must wait for the return value of the 3rd call and so on.
- When the last call is done, then call results (return values) are added up (superimposed) from lowest values to its immediate upper values, as illustrated in the table on the next slide.

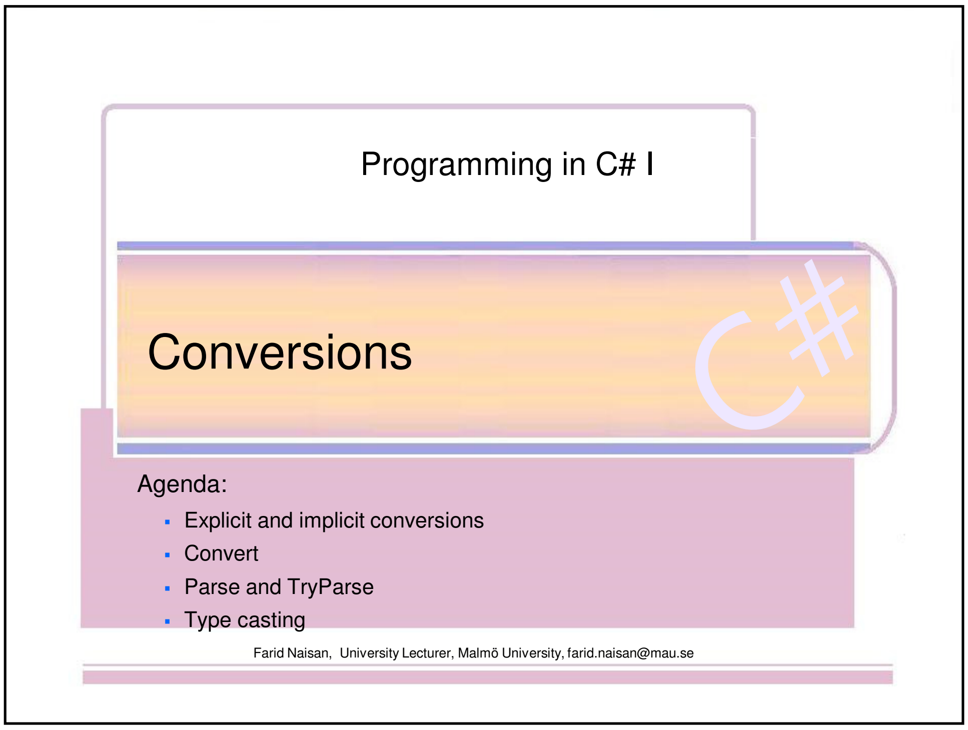
Recursive solution

```
81  //Method that adds from 1 to number by recursion
82  //For ex if number = 5, the sum = 1+2+3+4+5 = 15
83  public  int AddNumbers(int number)
84  {
85      if (number <= 1)   //base case (stop the recursiv calls)
86      {
87          return 1;
88      }
89      else
90      {
91          //recursive call - method calls itself
92          return number + AddNumbers(number - 1);
93      }
94  }
```

Code line getting executed	number -1	Return value	What happens
Line 92, number = 5	4	5 + AddNumbers(4)	5+10 = 15
Line 92, number = 4	3	4 + AddNumbers(3)	4+6 = 10 3
Line 92, number = 3	2	3 + AddNumbers(2)	3+3 = 6 2
Line 92, number = 2	1	2 + AddNumbers(1)	2+1 = 3
Line 87, number = 1	0	1	1 1

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 is 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.



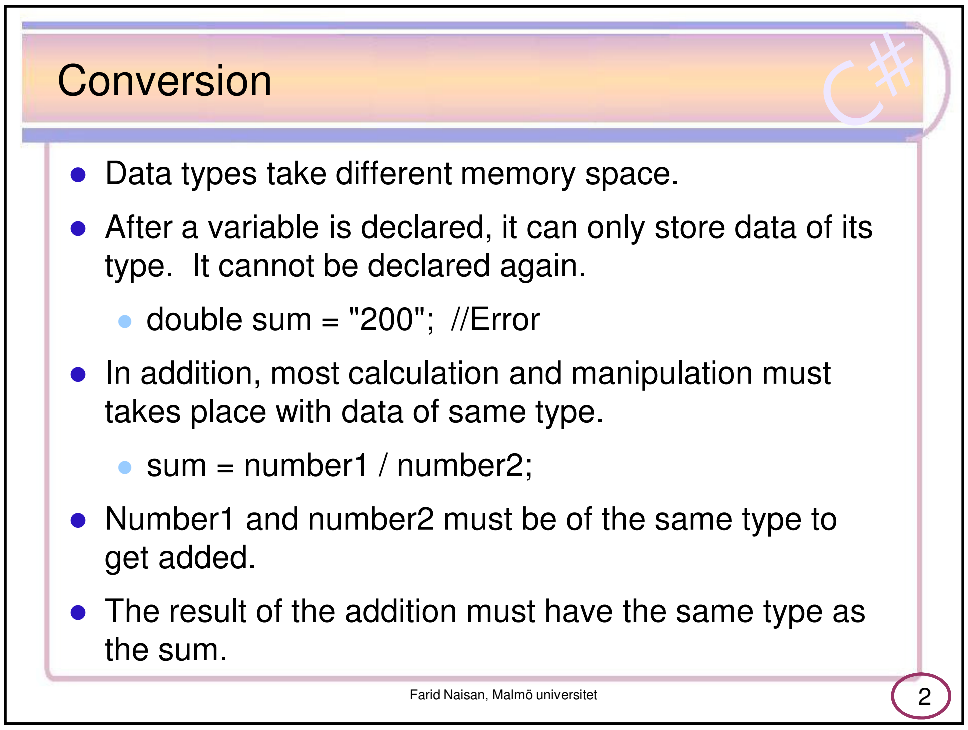
Programming in C# I

Conversions

Agenda:

- Explicit and implicit conversions
- Convert
- Parse and TryParse
- Type casting

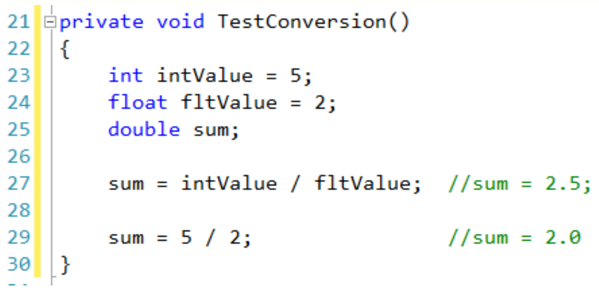
Farid Naisan, University Lecturer, Malmö University, farid.naisan@mau.se



Conversion

- Data types take different memory space.
- After a variable is declared, it can only store data of its type. It cannot be declared again.
  - double sum = "200"; //Error
- In addition, most calculation and manipulation must takes place with data of same type.
  - sum = number1 / number2;
- Number1 and number2 must be of the same type to get added.
- The result of the addition must have the same type as the sum.

Working with different types



```
private void TestConversion()
{
    int intValue = 5;
    float fltValue = 2;
    double sum;

    sum = intValue / fltValue;  //sum = 2.5;

    sum = 5 / 2;                //sum = 2.0
}
```

- Sum = 2.5 and 2.0, what happens.
- The first calculation is done using floating point division because of the **fltValue**.
- The second calculation is done as a integer division because both 5 and 2 are **integers**.

Implicit/Explicit conversions

- Implicit conversion occurs automatically by the compiler whenever required.

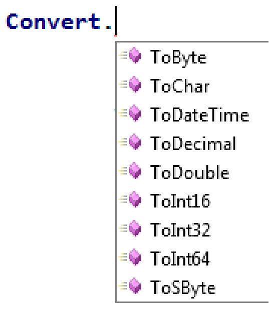
```
int count = 100;//no conversion 100 is an int
long number = 100;  //implicit conversion from int to long
long newCount = count; //implicit conversion from int to long
```

- Explicit conversion is when you mark your conversion in the code by type casting or otherwise.

```
//explicit conversion by type casting from long (newCount) to int (count)
count = (int)newCount;

double weight = 44; //implicit conversion from int to double

weight = (double)count * 10.0; //Explicit conversion by type casting

//Explicit conversion from double (0.0) to decimal (num)
decimal num = 0.0M;
```

Conversion methods – convertible types

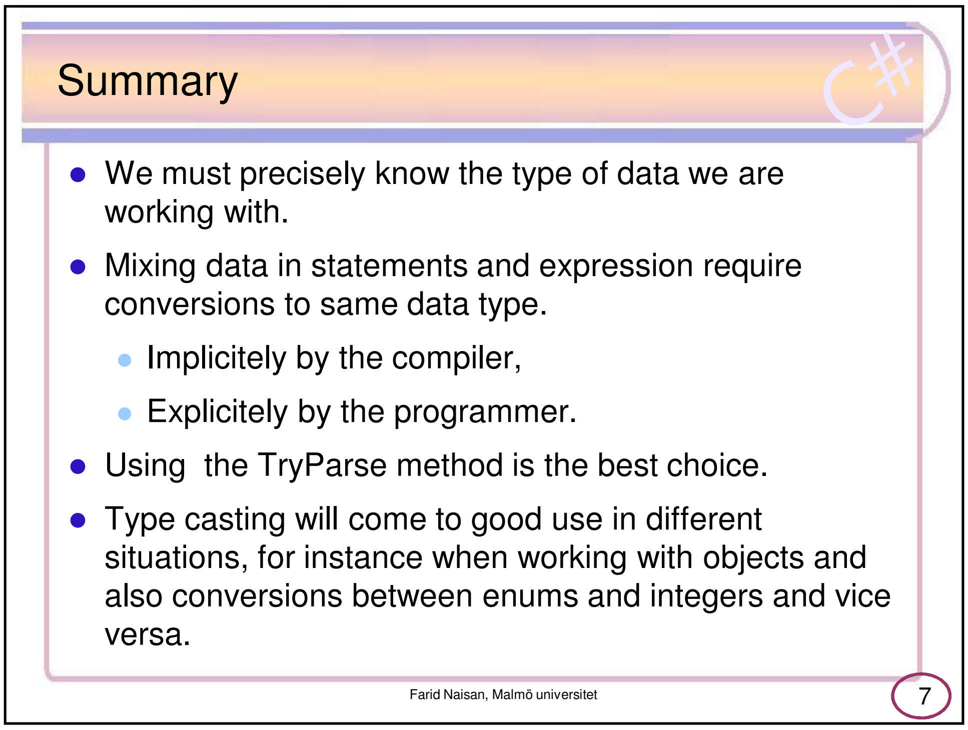
- **Implicit** conversion
  - double sum = 5; //from int to double
- **Explicit** conversion known as type casting
  - int sum = (int) (5/3.5); // = 1
- Using the **Convert** object.
  - Causes exception when failed.
- Using the **Parse** method;
  - Causes exception when failed.
- Using the **TryParse** method
  - Returns true if success and false otherwise



```
Convert.
```

Example

```
public void ConvertWithParse()
{
    Console.WriteLine("Give a valid decimal number.");
    string strNumber = Console.ReadLine();
    double number = Convert.ToDouble(strNumber);
    double number1 = double.Parse(strNumber);

    //More code
}
public void ConvertWithTryParse()
{
    Console.WriteLine("Give a valid decimal number.");
    double number = 0.0;   //double number
    string strNumber = Console.ReadLine();  //text
    bool goodNumber = double.TryParse(strNumber, out number);

    //More code
}
```



Summary

- We must precisely know the type of data we are working with.
- Mixing data in statements and expression require conversions to same data type.
  - Implicitely by the compiler,
  - Explicitely by the programmer.
- Using the TryParse method is the best choice.
- Type casting will come to good use in different situations, for instance when working with objects and also conversions between enums and integers and vice versa.