
# C# programming basics

# BASIC PRINCIPLES OF C #, CLR

# Training program

- **Block 1**. C# programming fundamentals
- **Block 2**. Windows application development
- **Block 3**. Service-oriented and web application development
- **Block 4**. Application architecture and design patterns
- **Block 5**. Certification

# Block 1 content

1. Basic principles of C#, CLR
2. Object oriented fundamentals
3. Exception handling
4. Advanced programming (Delegates, events, lambdas. Generics. Collections)
5. Assembly management and application debug
6. Multithreading and asynchronous processing
7. Data access
8. Unsafe code and pointers. .NET Framework security

# Module contents

- Basic principles of C#, CLR
  - C# & CLR basics
  - Data types
  - Operators
  - Array, Structure, Enum
  - System.Console

# Lecture contents

- Basic principles of C #, CLR. Operators
  - Operators
  - Program control statements
  - Use principles of structure with branch. Determining the appropriate iteration processing method

# Lecture contents

- Basic principles of C #, CLR. Operators
  - Operators
  - Program control statements
  - Use principles of structure with branch. Determining the appropriate iteration processing method

# Operators

- **Operator -** a sign or symbol that specifies the type of calculation to perform within an expression. There are mathematical, comparison, logical, and reference operators

- In C#, an operator is a program element that is applied to one or more operands in an expression or statement

.

# Expression

- **Expression -** any combination of operators, constants, literal values, functions, and names of fields (columns), controls, and properties that evaluates to a single value

- An expression is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, method, or namespace
  - Expressions can consist of a literal value, a method invocation, an operator and its operands, or a simple name. Simple names can be the name of a variable, type member, method parameter, namespace or type

# Statements

- The actions that a program takes are expressed in statements The order in which statements are executed in a program is called the **flow of control** or **flow of execution**

- A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block

- A statement block is enclosed in {} brackets and can contain nested blocks

# Labeled statements

- Labels are used to transfer program control directly to the specified statement

```
identifier :  statement
case constant-expression :  statement
default :  statement
```

- The scope of a label is the entire function in which it is declared

# Operators description (1/2)

- Operators that take one operand are referred to as unary operators
- Operators that take two operands are referred to as binary operators
- One operator, the conditional operator (?:), takes three operands and is the sole ternary operator in C#

# Operators description (2/2)

- An operand can be a valid expression that is composed of any length of code, and it can comprise any number of sub expressions
- There are main categories of operators: **arithmetic, bitwise, relational, and logical** operators
- There are several other operators that handle specialized situations, such as **array indexing**, **member access**, and the **lambda operator**

# The assignment operator

- The assignment operator works like in other computer languages

```
var_name = expression;
```

- The type of `var_name` must be compatible with the type of expression

# Arithmetic operators. Overview (1/2)

- **Addition** uses statement like this: `int n = 1 + 2;`
- **Subtraction** uses statement like this `int m = 10 - 9;`
- **Multiplication** uses the asterisk character ('*') like this:

```
int i = 46* 2;
```

- **Division** uses the forward slash character ('/') like this:

```
int j = 18/ 6;
```

- The **modulus** operator ('%') gets the remainder of a division like this: `int e = 11 % 2;`

# Arithmetic operators. Overview (2/2)

- You can use any of the built-in numeric data type types (double, float, int, short, etc.) for doing math
- You can combine multiple operations into one, as well as use a previously created variable in a statement:

```
int i = 3*j - 4;
```

- Order of operations matters and are like in mathematics, with multiplication and division happening before addition and subtraction, in a left to right order

# Arithmetic operators

| Operator | Meaning |
|---|---|
| + | Addition (also unary plus) |
| – | Subtraction (also unary minus) |
| * | Multiplication |
| / | Division |
| % | Modulus (remainder operator, it can be applied to both integer and floating-point types ) |
| ++ | Increment |
| –– | Decrement |

# The modulus operator

**Code example #1**

```
1. // int
2. Console.WriteLine(7 % 3);
3. // int
4. Console.WriteLine(-7 % 3);
5. // double
6. Console.WriteLine(7.0 % 3.2);
7. // decimal
8. Console.WriteLine(7.0m % 3.2m);
9. // double
10.Console.WriteLine(-7.2 % 3.0);
```

**The output from the program**

```
1
1
-1
0,6
0,6
-1,2
```

# Increment and decrement operators (1/2)

## Code example #2

```
1.  int i=0, j=0; i = i + 1;
2.  Console.WriteLine("Addition i = "+i);
3.  i = 0; j = 0; i = i++; //postfix
4.  Console.WriteLine("Incrementing (postfix, i = i++) i = " + i);
5.  i=0; j=0; i = ++i; //prefix
6.  Console.WriteLine("Incrementing (prefix, i = ++i) i = " + i);
7.  i=0; j=0; j = i + 1;
8.  Console.WriteLine("Addition j = i+1 ="+j+", i = "+i);
9.  i=0; j=0; j = i++;
10. Console.WriteLine("Incrementing (postfix) j = i++ =" + j + ", i =
    " + i);
11. i=0; j=0; j = ++i;
12. Console.WriteLine("Incrementing (prefix) j = ++i =" + j + ", i =
    " + i);
```

# Increment and decrement operators (2/2)

## Program output

```
Addition i = 1
Incrementing (postfix, i = i++) i = 0
Incrementing (prefix, i = ++i) i = 1
Addition j = i+1 =1, i = 0
Incrementing (postfix) j = i++ =0, i = 1
Incrementing (prefix) j = ++i =1, i = 1
```

# Relational, type and equality operators

| Expression | Description | Type |
|---|---|---|
| `x < y` | Less than | Relational |
| `x > y` | Greater than | Relational |
| `x <= y` | Less than or equal | Relational |
| `x >= y` | Greater than or equal | Relational |
| `x is T` | Return true if x is a T, false otherwise | Type |
| `X as T` | Return x typed as T, or null if x is not a T | Type |

| Expression | Description | Type |
|---|---|---|
| `x == y` | Equal | Equality |
| `x != y` | Not equal | Equality |

# Logical, conditional, and null operators (1/2)

| Category | Expression | Description |
|---|---|---|
| Logical AND | `x & y` | Integer bitwise AND, Boolean logical AND |
| Logical XOR | `x ^ y` | Integer bitwise XOR, boolean logical XOR |
| Logical OR | `x \| y` | Integer bitwise OR, boolean logical OR |
| Conditional AND | `x && y` | Evaluates y only if x is true |
| Conditional OR | `x \|\| y` | Evaluates y only if x is false |
| Null coalescing | `x ?? y` | Evaluates to y if x is null, to x otherwise |
| Conditional | `x ?: y : z` | Evaluates to y if x is true, z if x is false |
| Logical negation | `! x` | |

# Logical, conditional, and null operators (2/2)

| **Operands** | **\|\|** | **&&** | **^** | **&** | **\|** |
|---|---|---|---|---|---|
| true,true | true | true | false | true | true |
| true,false | true | false | true | false | true |
| false,true | true | false | true | false | true |
| false,false | false | false | false | false | false |

| **!** | |
|---|---|
| true | false |
| false | true |

# The ? operator

- The conditional operator (?:) returns one of two values depending on the value of a Boolean expression. Following is the syntax for the conditional operator

```
condition ? first_expression : second_expression;
```

- The condition must evaluate to true or false. If condition is true, first_expression is evaluated and becomes the result. If condition is false, second_expression is evaluated and becomes the result. Only one of the two expressions is evaluated

# Null-coalescing

- The ?? operator is called the null-coalescing operator
- It returns the left-hand operand if the operand is not null; otherwise it returns the right hand operand

# Compound assignment operators

- simplify the coding of certain assignment statements
- used for many of the binary operators

```
var_name op = expression;
```

- supports these operators:

```
+=, -=, *=, /=, %=, &=, |=, !=, <<=, >>=
```

# Bitwise operators

- act directly upon the bits of their operands
- are important to the systems-level programming tasks
- are `&`, `|`, `^`, and `~`
- can be used in compound assignments

# ~ Operator (bitwise negation)

- The `~` operator performs a bitwise complement operation on its operand, which has the effect of reversing each bit
- Bitwise complement operators are predefined for **`int`**, **`uint`**, **`long`**, and **`ulong`**

# Bitwise Operators. Code example

**Code example #3**

```
1. for (int i = 0; i <= n; i++)
2. {
3.     result = i & 0xFFFE;
4.     Console.Write("{0}\t", result);
5. }
```

# Shift operators

- The **left-shift operator (<<)** shifts its first operand left by the number of bits specified by its second operand. The type of the second operand must be an int or a type that has a predefined implicit numeric conversion to int
- The **right-shift operator (>>)** shifts its first operand right by the number of bits specified by its second operand

# Shift Operators. Code example

**1 << n returns 2n**

**Code example #4**

```
1. System.Diagnostics.Stopwatch tm = new System.Diagnostics.Stopwatch();
2. tm.Start();
3. byte pow = 1 << 5;
4. tm.Stop();
5. Console.WriteLine("Left-shift time = "+tm.Elapsed.TotalMilliseconds);
6. // first time
7. tm.Reset();
8. tm.Start();
9. byte mathPow = (byte)Math.Pow(2, 5);
10.tm.Stop();
11.Console.WriteLine("Math time = "+tm.Elapsed.TotalMilliseconds);
12.// second time (for first probe is best)
```

# Operators and precedence (1/2)

- Precedence refers to the order in which operations should be evaluated
- Subexpressions with higher operator precedence are evaluated first
- Operators with left associativity are evaluated from left to right. When an operator has right associativity, its expression is evaluated from right to left

# Operators and precedence (2/2)

| Symbol1 | Type of Operation | Associativity |
|---|---|---|
| [ ] ( ) . –> postfix ++ and postfix –– | Expression | Left to right |
| prefix ++ and prefix –– sizeof & * + – ~ ! | Unary | Right to left |
| typecasts | Unary | Right to left |
| * / % | Multiplicative | Left to right |
| + – | Additive | Left to right |
| << >> | Bitwise shift | Left to right |
| < > <= >= | Relational | Left to right |
| == != | Equality | Left to right |
| & | Bitwise-AND | Left to right |
| ^ | Bitwise-exclusive-OR | Left to right |
| \| | Bitwise-inclusive-OR | Left to right |
| && | Logical-AND | Left to right |
| \|\| | Logical-OR | Left to right |
| ? : | Conditional-expression | Right to left |
| = *= /= %= += –= <<= >>=&= ^= \|= | Simple and compound assignment2 | Right to left |
| , | Sequential evaluation | Left to right |

# () Operator

- Parentheses used to specify the order of operations in an expression
- Parentheses perform the following tasks:
  - Specify casts, or type conversions

```
double y = 11.1;
int i;
i = (int)y;
```

  - Invoke methods or delegates

```
MyMethod();
```

# Lecture contents

- Basic principles of C #, CLR. Operators
  - Operators
  - Program control statements
  - Use principles of structure with branch. Determining the appropriate iteration processing method

# Decision constructs

- There are two simple constructs to alter the flow of your program:
- The if-else statement
- The switch statement

# The if-else statement

- The simple form of the if statement is

```
if(condition) statement;
else statement;
```

where the targets of the if and else are single statements. The **else** once is optional

- The general form of the if using blocks of statements is

```
if(condition)
{
statement sequence
}
else
{
statement sequence
}
```

# The if-else-if ladder

- The if-else-if Ladder for the block is the costruct like this:

```
if(condition)
{
statement sequence
}
else if(condition)
{
statement sequence
}
else if(condition)
{
statement sequence
}
```

# The if-else statement. Code example (1/3)

**Code example #5**

```
1.          int i=4;
2.              if (6 < 4 * 3)
3.              Console.WriteLine ("6 < 4 * 3 - true"); // true
4.                  if (6 < 4 * 3)
5.                  {
6.                      Console.WriteLine ("6 < 4 * 3 - true");
7.                      Console.WriteLine("...");
8.                  }

9.              if (6 < 4 * 3)
10.                 if (6 > i * 3)
11.                     Console.WriteLine();
12.                 else
13.                     Console.WriteLine ("executes");
14.             // equivalent
```

# The if-else statement. Code example (2/3)

**Code example #5**

```
1. if (6 < 4 * 3)
2. {
3. if (6 > i * 3)
4. Console.WriteLine();
5. Else
6. Console.WriteLine
   ("executes");
7. }
```

```
8. if (6 < i * 3)
9. {
10.if (6 > i * 3)
   Console.WriteLine();
11.}
12.else
   Console.WriteLine("does not
   execute");
13.}
```

# The if-else statement. Code example (3/3)

**Code example #5**

Program output

```
6 < 4 * 3 - true
6 < 4 * 3 - true
...
executes
executes
```

# The switch statement (1/4)

- allows to handle program flow based on a predefined set of choices
- C# demands that each case (including default) that contains executable statements have a terminating **`break`** to avoid fall-through
- The **`switch`** expression must be of an integer type, such as **`char`**, **`byte`**, **`short`**, or **`int`**, of an enumeration type, or of type **`string`**
- The **`default`** sequence is executed if no **`case`** constant matches the expression. The **`default`** is optional

# The switch statement (2/4)

- Each case label specifies a constant value
- The switch statement transfers control to the switch section whose case label matches the value of the switch expression
- If no case label contains a matching value, control is transferred to the default section, if there is one
- If there is no default section, no action is taken and control is transferred outside the switch statement

# The switch statement (3/4)

- A switch statement can include any number of switch sections, and each section can have one or more case. However, no two case labels may contain the same constant value.
- Execution of the statement list in the selected switch section begins with the first statement and proceeds through the statement list, typically until a **`jump`** statement, such as a **`break`**, **`goto case`**, **`return`**, or **`throw`**, is reached
  - At that point, control is transferred outside the switch statement or to another case label

# The switch statement (4/4)

```
switch(expression) {
case constant1:
statement sequence
break;
case constant2:
statement sequence
break;
…
default:
statement sequence
break;
}
```

# The switch statement. Code example

**Code example #6**

```
1. a = int.Parse(Console.ReadLine());
2. switch (a)
3. {
4.     case 1:
5.     Console.WriteLine("1");
6.     break;
7.     case 2:
8.     Console.WriteLine("2");
9.     break;
10.    default:
11.    Console.WriteLine("Exit");
12.    break;
13.}
```

# Iteration statements (1/2)

- All programming languages provide ways to repeat blocks of code (create loops) until a terminating condition has been met
- Iteration statements
  - cause embedded statements to be executed a number of times, subject to the loop-termination criteria
  - are executed in order, except when a **jump** statement is encountered

# Iteration statements (2/2)

- `do`
- `for`
- `foreach`
- `in`
- `while`

# The for loop (1/2)

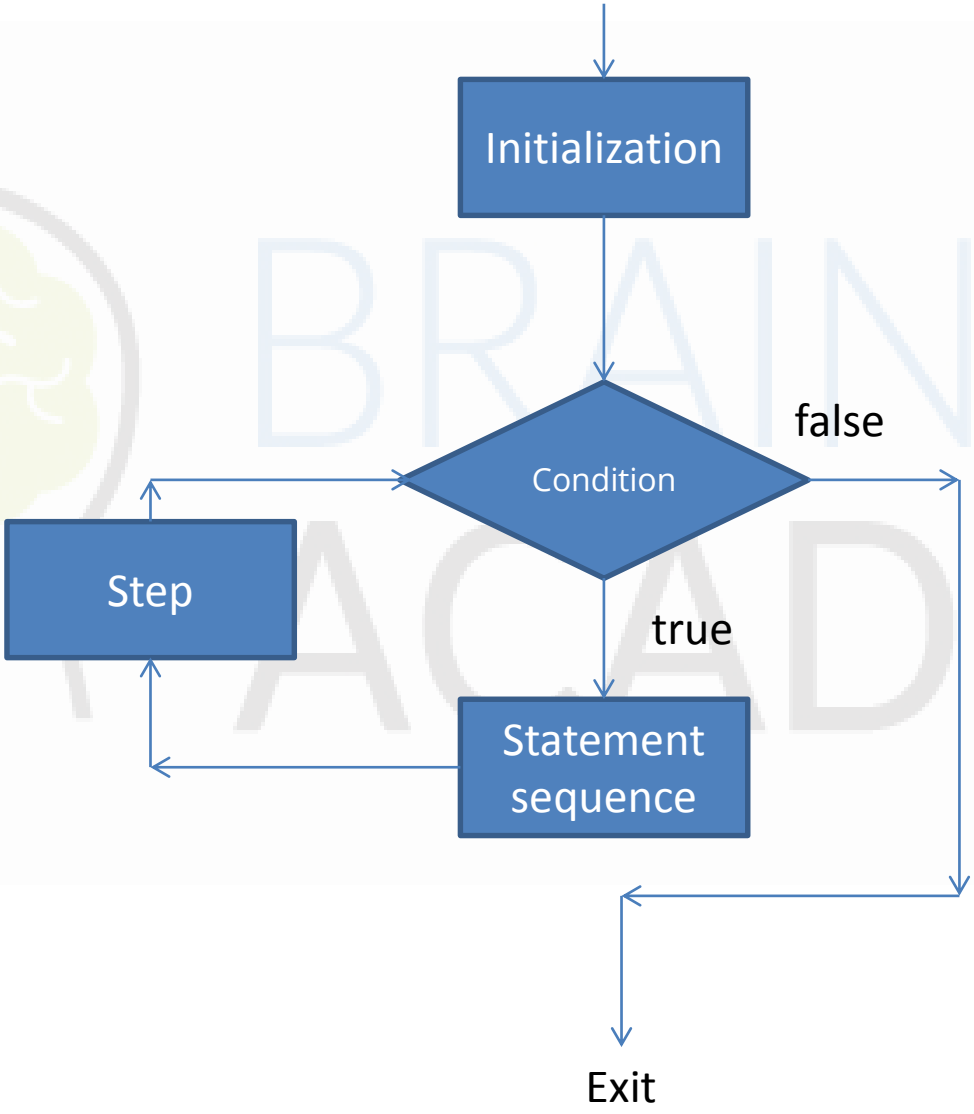
- The general form of the for loop for repeating a single statement is

```
for(initialization; condition; iteration)
statement;
```

- For repeating a block, the general form is

```
for(initialization; condition; iteration)
{
statement sequence
}
```

# The for loop (2/2)



# The for Loop. Code example (1/3)

**Code example #7**

```
1.  int res = 1, lim = 10;
2.  for (int j = 2; j <= lim; )
3.  {
4.      //multiplication
5.      res *= j;
6.      Console.WriteLine("j = "+j+" res = res*j = "+res);
7.      j++;
8.  }
9.  Console.WriteLine("End res = "+res +" . Press any key");
10. Console.ReadKey();
```

# The for Loop. Code example (2/3)

**Code example #8**

```
1.  char opt;
2.  for (; ; )
3.  {
4.  do
5.  {
6.      Console.WriteLine("Help on:");
7.      Console.WriteLine("  1. operator");
8.      Console.WriteLine("  2. statement");
9.      Console.WriteLine("  3. label");
10.     Console.WriteLine("  4. expression");
11.     Console.Write("Choose number (q to quit): ");
```

# The for loop. Code example (3/3)

```
12.     do
13.     {
14.     opt = (char)Console.Read();
15.     } while (opt == '\n' | opt == '\r');
16. } while (opt < '1' | opt > '4' & opt != 'q');

17. if (opt == 'q') break;

18. Console.WriteLine("\n");
19. Console.WriteLine("Come back to menu press any key");
20. Console.ReadKey();
21. }
```

# The while and do-while looping construct

- are useful should you wish to execute a block of statements until some terminating condition has been reached (a specified expression evaluates to false)

- unlike the while statement, a do-while loop is **executed one time** before the conditional expression is evaluated (are guaranteed to execute the block of code at least once)

# The while looping construct (1/2)

- General form:

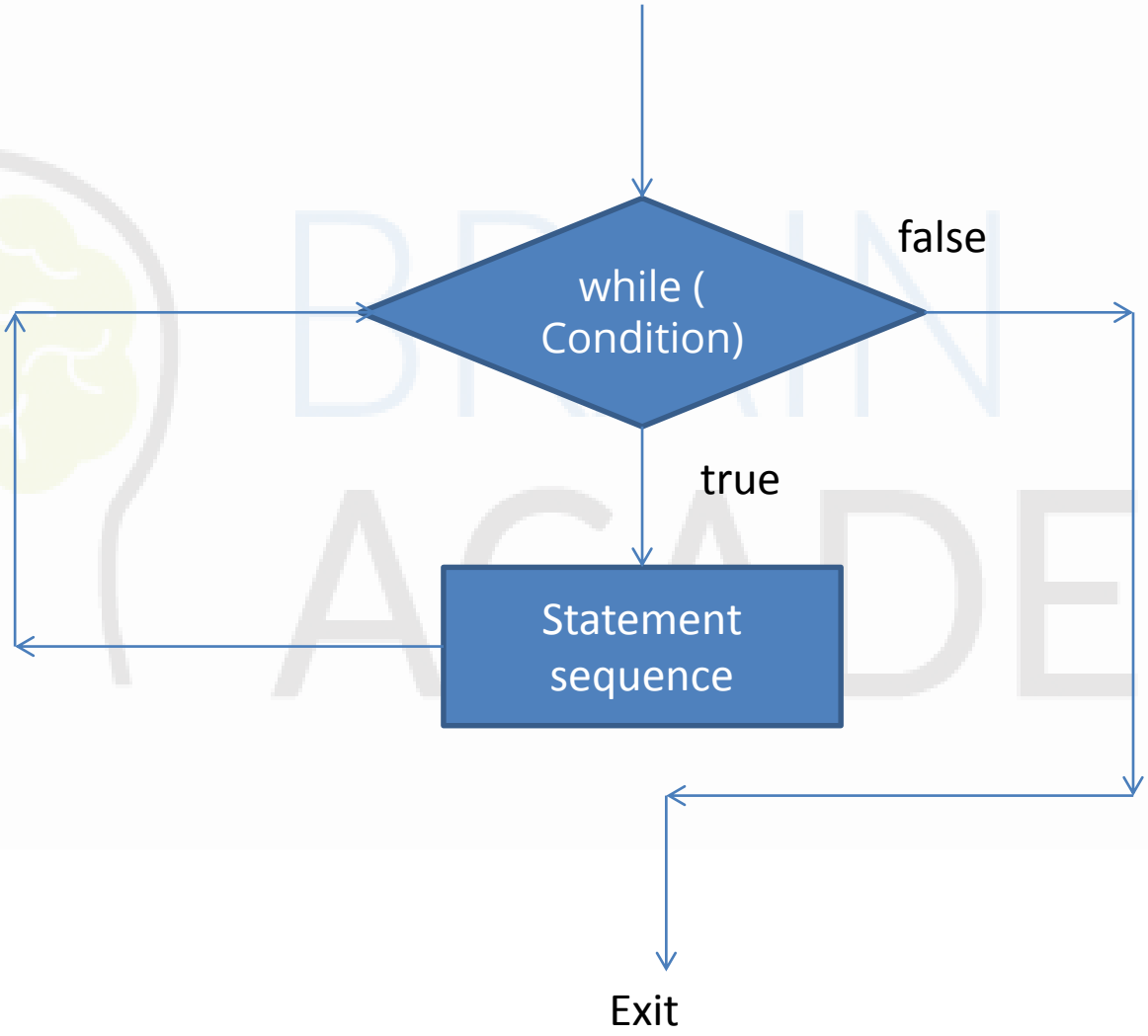
  `while(condition) statement;`

  where statement can be a single statement or a block of statements

- It checks the conditional expression at the top of the loop

- A while loop can be terminated when a **`break`**, **`goto`**, **`return`**, or **`throw`** statement transfers control outside the loop. To pass control to the next iteration without exiting the loop, use the **`continue`** statement

# The while looping construct (2/2)



# The while loop. Code example (1/4)

**Code example #9**

```
1.  int i = 0;
2.  while (i < 8)
3.  {
4.      i++;
5.      if (i == 4)
6.      {
7.              i++;
8.              continue;
9.      }
10. }
11. Console.WriteLine("Done. i = "+i + ".  Press any key");
```

# The while loop. Code example (2/4)

**Code example #10**

```
1.  int i = 0;
2.  while (i < 8)
3.  {
4.      i++;
5.      Console.WriteLine(" i = " + i);
6.      if (i == 4)
7.              break;
8.  }
9.  i++;
10. Console.WriteLine("Done. i = " + i + ".  Press any key");
```

# The while loop. Code example (3/4)

**Code example #11**

```
1.  int i = 0;
2.  while (i < 8)
3.  {int i = 0;
4.                  while (i < 8)
5.                  {
6.                      Console.WriteLine(" i = " + i);
7.                      i++;
8.                      if (i == 4)
9.                          goto fin;
10.                 }
```

# The while loop. Code example (4/4)

**Code example #11**

```
11. fin:
12.     Console.WriteLine("Done. i = " + i + ".  Press any
    key");
13.     i++;
14.     Console.WriteLine(" i = " + i);
15.     if (i == 4)
16.             break;
17. }
18. i++;
19. Console.WriteLine("Done. i = " + i + ".  Press any key");
```

# The do-while loop construct

- General form:

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

- The braces are not necessary when only one statement is present
- At any point in the do-while block, you can break out of the loop using the **`break`** statement. You can step directly to the **`while`** expression evaluation statement by using the **`continue`** statement

# The do-while loop. Code example

**Code example #12**

```
1. int m;
2. m = -5;
3. do
4. {
5.     if (m > 0) break;
6.     Console.Write(m + "  ");
7.     m++;
8. } while (m <= 10);

9. Console.WriteLine("Done m = "+m);
```

# The foreach, in statements

- The **foreach** statement repeats a group of embedded statements for each element in an array or an object collection

# The jump statements (1/2)

- Branching is performed using jump statements, cause an immediate transfer of the program control:
  - **`break`**
  - **`continue`**
  - **`goto`**
  - **`return`**
  - **`throw`**

# Some jump statements (2/2)

- The **`break`** statement terminates the closest enclosing loop or switch statement in which it appears. Control is passed to the statement that follows the terminated statement, if any
- The **`continue`** statement passes control to the next iteration of the enclosing **`while`**, **`do`**, **`for`**, or **`foreach`** statement in which it appears
- The **`goto`** statement transfers the program control directly to a labeled statement. A common use of **`goto`** is to transfer control to a specific switch-case label or the default label in a switch statement

# Lecture contents

- Basic principles of C #, CLR. Operators
  - Operators
  - Program control statements
  - Use principles of structure with branch. Determining the appropriate iteration processing method

# Controlling program flow (1/3)

- All applications require some program flow options
- You iterate over a series of steps, but at some point, you might need to do something different depending on the outcome of some other action
- Code branching can be thought of as program flow moving to a different location in the code listing and then coming back to where it left off, or repeating lines of code to complete a set of tasks over and over

# Controlling program flow (2/3)

- Early attempts at program flow control used statements such as **goto** where labels were used in code and program flow was directed to code in a labeled section. These code branching statements created *spaghetti code*, making it hard to debug and maintain application code because it forced the programmer to jump from one code location to another and back again to try to make sense of the logic often getting lost in the process

- C# provides various program flow statements such as decision and repetition structures that allow the programmer to make decisions based on conditions and to iterate or repeat over code to accomplish necessary tasks

# Controlling program flow (3/3)

- Decision structures such as the if statement and the switch statement permit the programmer to compare values and direct code execution based on the result
- Repetition in code enables you to iterate over collections or arrays to act on the items contained in those structures. Repetition also enables you to perform the same code statement or set of statements to perform various other actions until a certain condition is met