Control Statements

if-else, switch, while, for, do-while

Conditional Statements

- So far statements of our programs execute sequentially one after another.
- What happens when
  - we want to execute a statement depending on a condition?
    - e.g. If there is enough money in the bank account, give the money
  - we want to execute one statement when a condition holds and another statement when a condition does not hold?
    - e.g. If dollar is high, sell dollar. Otherwise, buy dollar.
  - we want to select from many statements according to one or more criteria (*selection*).
    - e.g. If dollar is high and euro is low, sell dollar and buy euro. If dollar is low and euro is high, sell euro and buy dollar. If both of them are high, sell both and buy YTL.
- You achieve conditional execution with *if-else* statements

Syntax

```
if (<condition>)
{
  <statement_true_1>;
  ...
  <statement_true_N>;
}
else
{
  <statement_false_1>;
  ...
  <statement_false_N>;
}
```

- If ***condition*** is TRUE then
  statement_true_1 … statement_true_N
  are executed,
  if ***condition*** is FALSE
  statement_false_1 … statement_false_N
  are executed.

```
if (<condition>)
{
  <statement_true_1>;
  ...
  <statement_true_N>;
}
```

- else and *statement_false's* are optional
  - if condition is FALSE then nothing will be executed and execution continues with the next statement in the program
- ***<condition>*** must be in brackets

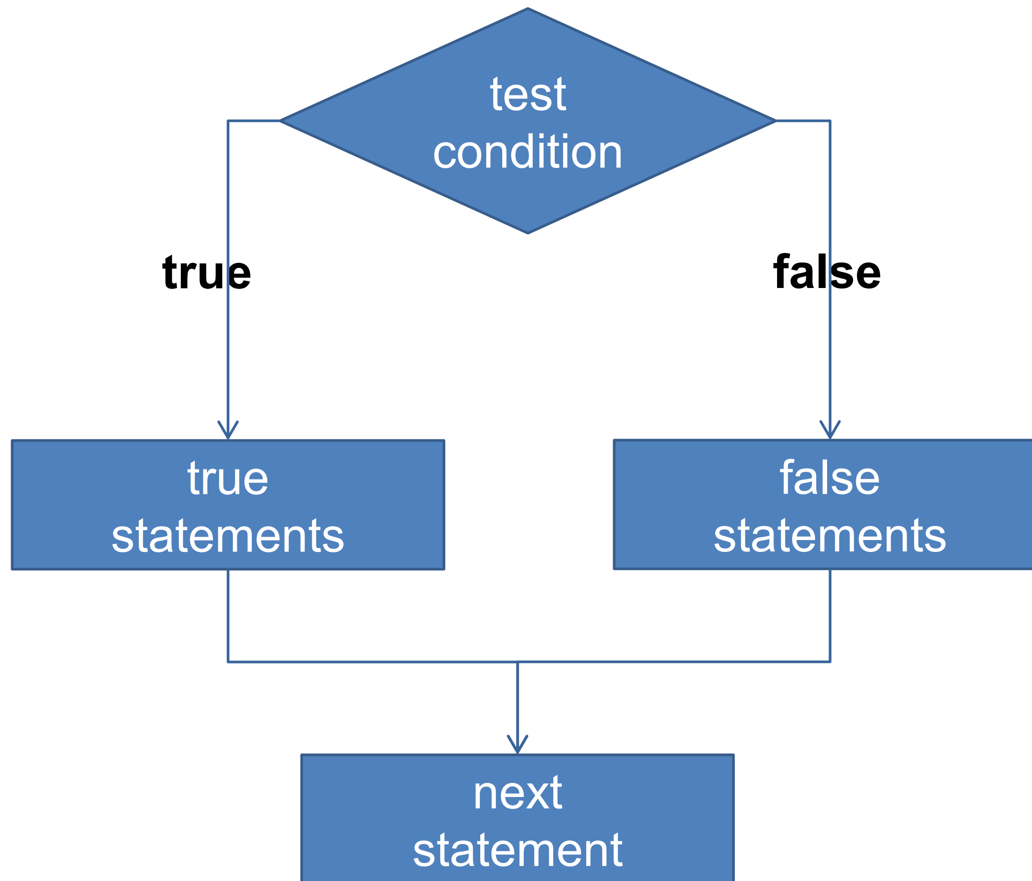
Another Syntax (without { })

```
if (<condition>)
  <statement_true>;
else
  <statement_false>;
```

```
if (<condition>)
  <statement_true>;
```

- Can be used when there is only one statement
- Not suggested (we will see why)

Flow diagram of if-else



`if-else` example

```
if ( grade >= 60 )
    Console.WriteLine( "Passed" );
else
    Console.WriteLine( "Failed" );
```

Boolean type and expressions

- The condition in an if statement must be a Boolean expression (named for George Boole)
  - Values are true or false
  - **bool** is a built-in value type like int, double

```
int      degrees;
bool      isHot = false;

Console.WriteLine("enter temperature: “);
degrees = Convert.ToInt32(Console.ReadLine());
if (degrees > 35)
{
    isHot = true;
}
```

Conditional operator (? :)

- The **conditional operator** (**?:**) can be used in place of an `if…else` statement.

```
Console.WriteLine( grade >= 60 ? "Passed" : "Failed" );
```

  - The first operand is a **boolean** expression that evaluates to **true** or **false**.
  - The second operand is the value if the expression is true
  - The third operand is the value if the expression is false. `

Relational Operators

- Relational operators are used to compare values:

<	less than	`number < 5`
<=	less than or equal	`number <= 0`
>	greater than	`num1 > num2`
>=	greater than or equal	`num1 >= num2`
==	equality check	`num1 == 0`
!=	inequality check	`num1 != num2`

- They take two operands
  - operands can be literals, variables or expressions
- Used for many types
  - numeric comparisons
  - string comparisons (alphabetical)

Logical operators

- Boolean expressions can be combined using logical operators: AND, OR, NOT
  - In C# we use && || !   respectively

A	B	A || B	A && B
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

A	! A
true	false
false	true

Example

- Range check: between 0 and 100 (includes 0 and 100), or not?
  If so, display a message saying that the number is in the range.
  If not, the message should say “out of the range”.
- Solution 1: using logical AND operator

```
if (num >= 0 && num <= 100)
     Console.Write("number is in the range");
else
     Console.Write("number is out of range");
```

- Solution 2: using logical AND and NOT operators

```
if ( ! (num >= 0 && num <= 100) )
     Console.Write("number is out of range");
else
     Console.Write("number is in the range");
```

- Solution 3: using logical OR operator

```
if (num < 0 || num > 100)
     Console.Write("number is out of range");
else
     Console.Write("number is in the range");
```

De Morgan's Rules

- Compare solution 2 and 3
  - two conditions are equivalent

```
( ! (num >= 0 && num <= 100) )

( num < 0 || num > 100 )
```

- De Morgan's Rules (assume a and b are two boolean expressions)

```
! (a   &&   b)  =   !a   ||   !b
! (a   ||   b)  =   !a   &&   !b
```

- De Morgan'a Rules can be generalized to several expressions (e.g. 4 boolean expressions case)

```
! (a && b && c && d) =  !a  ||  !b  ||  !c  ||  !d
! (a || b || c || d) =  !a  &&  !b  &&  !c  &&  !d
```

Operator Precedence - Revisited

- Upper operator groups have precedence

<table>
<tr><th>Operator</th><th>Explanation</th><th>Associativity</th></tr>
<tr><td>+ - !</td><td>plus and minus signs, logical NOT</td><td>right-to-left</td></tr>
<tr><td>* / %</td><td>multiplication, division and modulus</td><td>left-to-right</td></tr>
<tr><td>+ -</td><td>addition, subtraction</td><td>left-to-right</td></tr>
<tr><td><< >></td><td>stream insertion and extraction</td><td>left-to-right</td></tr>
<tr><td>< <= > >=</td><td>inequality comparison operators</td><td>left-to-right</td></tr>
<tr><td>== !=</td><td>equal, not equal comparison</td><td>left-to-right</td></tr>
<tr><td>&&</td><td>logical and</td><td>left-to-right</td></tr>
<tr><td>||</td><td>logical or</td><td>left-to-right</td></tr>
<tr><td>= += -=
*= /= %=</td><td>assignment operators</td><td>right-to-left</td></tr>
</table>

Nested if statements

- if/else statements are inside other if/else statements
- Method to select from multiple choices
- Example: input a numeric grade and convert to letter grade

90 .. 100	A
80 .. 89	B
70 .. 79	C
60 .. 69	D
0 .. 59	F
otherwise	F

Nested if statements

- This may be written in C# as

```
if ( grade >= 90 )
     Console.WriteLine( "A" );
  else
     if ( grade >= 80 )
        Console.WriteLine( "B" );
     else
        if ( grade >= 70 )
           Console.WriteLine( "C" );
        else
           if ( grade >= 60 )
              Console.WriteLine( "D" );
           else
              Console.WriteLine( "F" );
```

Nested if statements (Cont.)

- Most C# programmers prefer to use else if:

```
if ( grade >= 90 )
   Console.WriteLine( "A" );
else if ( grade >= 80 )
   Console.WriteLine( "B" );
else if ( grade >= 70 )
   Console.WriteLine( "C" );
else if ( grade >= 60 )
   Console.WriteLine( "D" );
else
   Console.WriteLine( "F" );
```

Short-circuit Evaluation

- Some subexpressions in Boolean expressions are not evaluated if the entire expression's value is already known using the subexpression evaluated so far.
- Rule: Evaluate the first (leftmost) boolean subexpression. If its value is enough to judge about the value of the entire expression, then stop there. Otherwise continue evaluation towards right.

```
if (count != 0 && scores/count < 60)
{
    Console.WriteLine("low average");
}
```

  - In this example, if the value of count is zero, then first subexpression becomes false and the second one is not evaluated.
  - In this way, we avoid "division by zero" error (that would cause to stop the execution of the program)
  - Alternative method to avoid division by zero without using short-circuit evaluation:

```
if (count != 0)
{
   if (scores/count < 60)
   {
       Console.WriteLine("low average");
   }
}
```

Dangling Else Problem

```
if ( x % 2  == 0)
    if ( x < 0 )
        Console.WriteLine("{0} is an even, negative number‟, x);
else
    Console.WriteLine("{0} is an odd number‟, x);
```

- What does it display for x=4?
- The problem is that it displays “odd number” message for positive even numbers and zero.
- Reason is that, although indentation says the reverse, else belongs to second (inner) if
  - else belongs to the most recent if
- Solution: use braces (see next slide)

Solution to Dangling Else Problem

```
if ( x % 2  == 0)
{
   if ( x < 0 )
      Console.WriteLine("{0} is an even, negative number", x);
}
else
{
   Console.WriteLine("{0} is an odd number", x);
}
```

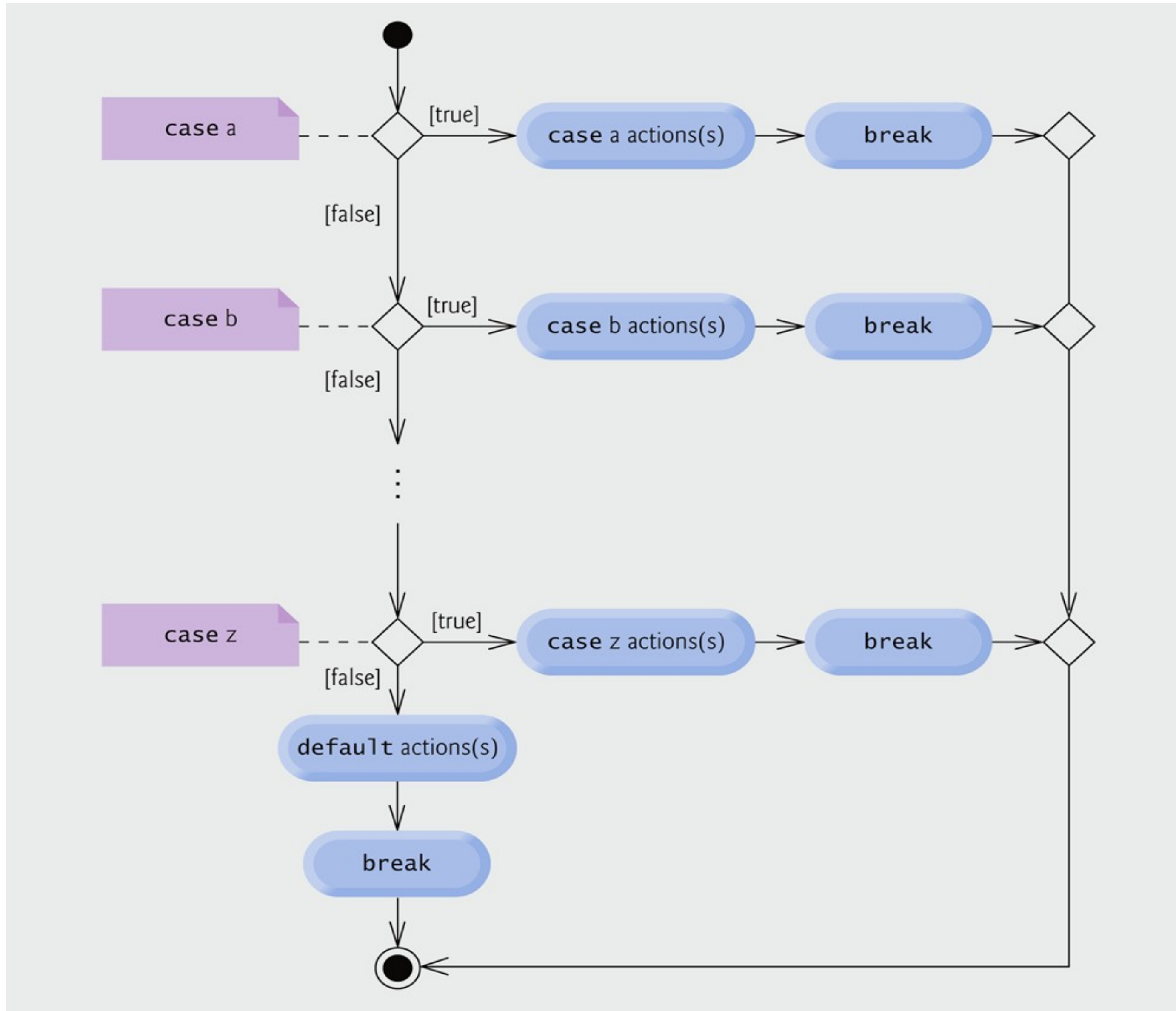
- Now else belongs to the first if
- if – else matching rule
  - Each else belongs to the nearest if for which there is no else and in the same compound block

`switch` statement

- The **switch multiple-selection** statement performs different actions based on the value of an expression.
- Each action is associated with the value of a **constant integral expression** or a **constant string expression** that the expression may assume.
- Let's see an example: GradeBookswitch.cs

```
private void IncrementLetterGradeCounter( int grade )
{
      switch ( grade / 10 )
      {
         case 9:          // grade was in the 90s
         case 10:         // grade was 100
            ++aCount;
            break;        // necessary to exit switch
         case 8:          // grade was between 80 and 89
            ++bCount;
            break;        // exit switch
         case 7:          // grade was between 70 and 79
            ++cCount;
            break;        // exit switch
         case 6:          // grade was between 60 and 69
            ++dCount;
            break;        // exit switch
         default:         // grade was less than 60
            ++fCount;
            break;        // exit switch
      }
 } // end method IncrementLetterGradeCounter
```

Flow diagram of `switch`



`switch` statement

- The expression after each `case` can be only a constant integral expression or a constant string expression.
- You can also use `null` and **character constants** which represent the integer values of characters.
- The expression also can be a **constant** that contains a value which does not change for the entire application.

From Selection to Repetition

- The if statement and if/else statement allow a *block* of statements to be executed selectively: based on a condition

```
Console.WriteLine("Please enter a non-negative number");
inputnumber = Convert.ToInt32(Console.ReadLine());
if (inputnumber < 0)
{
   Console.WriteLine(inputnumber + " is negative. Wrong Input");
}
```

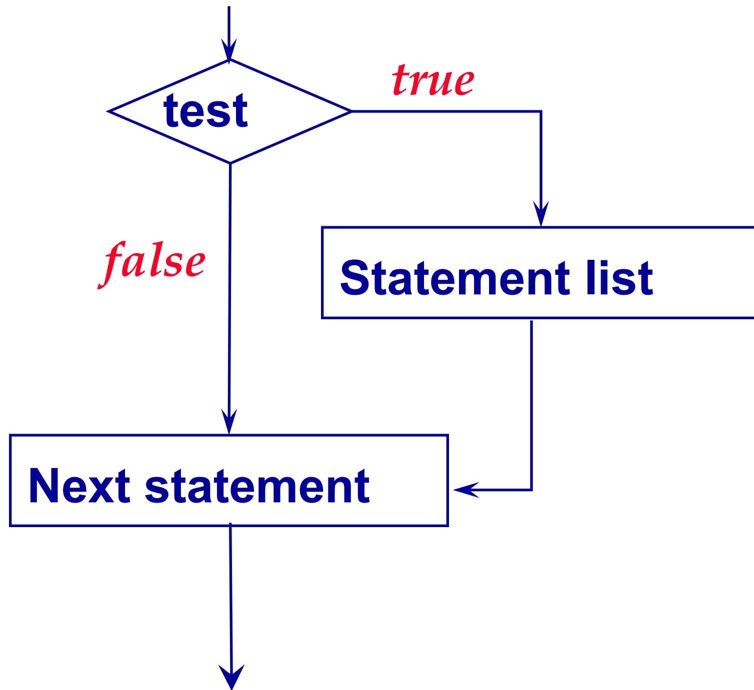
- This piece of code does not ask another input number if the number is negative.

- The while statement repeatedly executes a block of statements while the condition is true

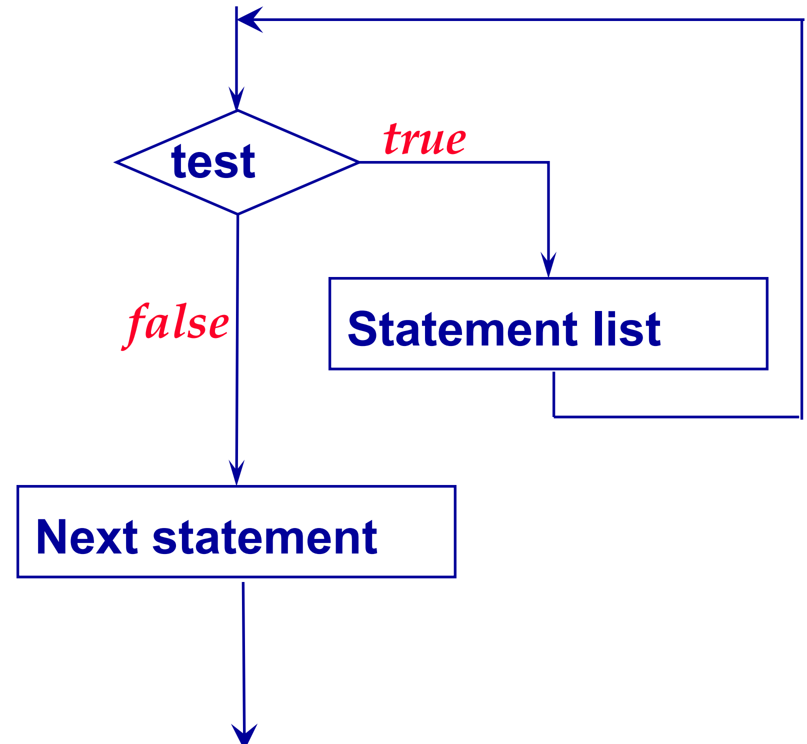
```
Console.WriteLine("Please enter a non-negative number");
inputnumber = Convert.ToInt32(Console.ReadLine());
while (inputnumber < 0)
{
    Console.WriteLine(inputnumber + " is negative! Try again");
    inputnumber = Convert.ToInt32(Console.ReadLine());
}
```

Flow diagram of while loop

```
if (test)
{
   statement list;
}
```




```
while (test)
{
    statement list;
}
```



`while` loop syntax

```
<initialization>
while (<test>)
{
    <statement1>;
    ...
    <statementN>;
   <update>
}
```

Counter-controlled loop example

- Consider the following problem statement:

  A class of 10 students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.
- The algorithm must input each grade, keep track of the total of all grades input, perform the averaging calculation and display the result.

Counter-controlled loop algorithm

set total to zero

set grade counter to one

while grade counter is less than or equal to 10

    prompt the user to enter the next grade

    input the next grade

    add the grade into the total

    add one to the grade counter

set the class average to the total divided by 10

display the class average

Counter-controlled loop code

```
// initialization
total = 0;              // initialize the total
gradeCounter = 1;       // initialize the loop counter

while ( gradeCounter <= 10 ) // test
{
   Console.Write( "Enter grade: " );                  // prompt the user
   grade = Convert.ToInt32( Console.ReadLine() ); // read grade
   total = total + grade;                   // add the grade to total
   gradeCounter = gradeCounter + 1;    // update
}

// termination phase
average = total / 10; // integer division yields integer result
```

Sentinel-controlled loop example

- Consider the following problem:

  Develop a class-averaging application that processes grades for an arbitrary number of students each time it is run.

- In this example, no indication is given of how many grades the user will enter during the application's execution.

Sentinel-controlled algorithm

initialize total to zero

initialize counter to zero

prompt the user to enter the first grade

input the first grade (possibly the sentinel)

while the user has not yet entered the sentinel

  add this grade into the running total

  add one to the grade counter

  prompt the user to enter the next grade

  input the next grade (possibly the sentinel)

if the counter is not equal to zero

  set the average to the total divided by the counter

  display the average

else

  display "No grades were entered"

Let’s see Sentinel_While.sln

```
// initialization phase
total = 0;              // initialize total
gradeCounter = 0;       // initialize loop counter

// prompt for and read a grade from the user
Console.Write("Enter grade or -1 to quit: ");
grade = Convert.ToInt32(Console.ReadLine());

// loop until sentinel value is read from the user
while (grade != -1)
{
    total = total + grade;               // add grade to total
    gradeCounter = gradeCounter + 1; // increment counter

    // prompt for and read the next grade from the user
    Console.Write("Enter grade or -1 to quit: ");
    grade = Convert.ToInt32(Console.ReadLine());
}
```

`for` loop syntax compared with `while`

```
<initialization>
while (<test>)
{
      <statement1>;
      ...
      <statementN>;
      <update>
}
```

```
for (<initialization>;
      <test>;
      <update> )
{
      <statement1>;
      ...
     <statementN>;
}
```

Example

- Calculate the sum of the integer numbers between 1 and 10

```
int sum = 0;          // this program piece
int i = 1;            // calculates the sum of
while (i <= 10)       // integers between and
{                     // including 1 and 10
   sum = sum + i;
   i = i + 1;
}
```

Same example with `for` loop

```
int sum = 0;
int i = 1;
while (i <= 10)
{
    sum = sum + i;
    i = i + 1;
}
```

```
int sum = 0;

for (int i=1; i <= 10; i=i+1)
{
    sum = sum + i;
}
```

Scope of the counter variable in `for`

```
for (int i=1; i <= 10; i=i+1)
```

- If the *initialization* expression declares the control variable, the control variable will not exist outside the for statement.
- This restriction is known as the variable's **scope**.
- Similarly, a local variable can be used only in the method that declares the variable and only from the point of declaration.

```
int i;
for (i=1; i <= 10; i=i+1)
```

`for` loop syntax

- Comma-separated lists that enable you to use multiple initialization expressions or multiple increment expressions:

```
for ( int i = 2; i <= 20; total += i, i += 2 )
    ; // empty statement
```

Increment and Decrement Operators

- C# provides operators for adding or subtracting 1 from a numeric variable
  - The unary **increment operator**, **++**
  - The unary **decrement operator**, **--**.

Operator	Called	Sample expression	Explanation
++	prefix increment	++a	Increments a by 1, then uses the new value of a in the expression.
++	postfix increment	a++	Uses the current value of a, then increments a by 1.
--	prefix decrement	--b	Decrements b by 1, then uses the new value of b.
--	postfix decrement	b--	Uses the current value of b, then decrements b by 1.

Bad loops

1.
```
for (int i = 10; i < 5; i=i+1)
{
    Console.WriteLine("How many times do I print?");
}
```

2.
```
for (int i = 10; i >= 1; i=i+1)
{
    Console.WriteLine("How many times do I print?");
}
```

3.
```
int i = 1;
while (i < 20)
{
    Console.WriteLine("How many times do I print?");
}
```

Infinite loops

- What is the problem with the code below?
  - cannot say infinite loop for sure, depends on input number
    - for example, if num is an odd number, then the loop is infinite

```
int num = Convert.ToInt32(Console.ReadLine());
int start = 0;
while (start != num)
{
    start += 2;
    Console.WriteLine(start);
}
```

- How to fix?
  - You can check whether num is even before starting the loop.

```
if (num % 2 == 0)
{    while (start != num)
     {  start += 2;
        Console.WriteLine(start);
     }
}
```

Other Common Problems

- Easy to iterate one more or one less times
- Test each loop with the inputs that cause:
  - zero iterations of the loop body
  - one iteration of the loop body
  - maximum number of iterations
  - one less than the maximum number of iterations
- Use the debugger and watch the variables.

The `do-while` loop

- Similar to while loop, but the test is after the execution of the loop body
- The while loop may never execute, do-while loop executes **at least once**

```
<initialization>
do
{
    <statement1>;
     ...
    <statementN>;
    <update>
} while (<condition>);
```

Don't forget

- Example: Prompt for a number between 0 and 100, loop until such a number is entered (user should enter at least one number)

```
do
{
    Console.WriteLine("enter number in range [0..100]");
    num = Convert.ToInt32(Console.ReadLine());
} while (num < 0 || num > 100 );
```

`foreach`

- Good with arrays or collections, we will revisit

Nested loops – Example

- Write a function to display a perpendicular isosceles triangle of stars (perpendicular side length is parameter)
  - e.g. if side length is 6 , the output should look like

```
*
**
***
****
*****
******
```

  - See drawtriangle.cs

break

- The break statement causes immediate exit from a statement.

```
1  // Fig. 6.12: BreakTest.cs
2  // break statement exiting a for statement.
3  using System;
4
5  public class BreakTest
6  {
7     public static void Main( string[] args )
8     {
9        int count; // control variable also used after loop teminates
10
11       for ( count = 1; count <= 10; count++ ) // loop 10 times
12       {
13          if ( count == 5 ) // if count is 5,
14             break; // terminate loop
15
16          Console.Write( "{0} ", count );
17       } // end for
18
19       Console.WriteLine( "\nBroke out of loop at count = {0}", count );
20    } // end Main
21 } // end class BreakTest
```

When `count` is 5, the `break` statement terminates the `for` statement.

```
1 2 3 4
Broke out of loop at count = 5
```

`break` and `continue`

- The **continue statement** skips the remaining statements in the loop body and tests whether to proceed with the next iteration of the loop.
- In a for statement, the increment expression executes, then the application evaluates the loop-continuation test.

Software Engineering

Some programmers feel that break and continue statements violate structured programming, since the same effects are achievable with structured programming techniques.

continue

```
1  // Fig. 6.13: ContinueTest.cs
2  // continue statement terminating an iteration of a for statement.
3  using System;
4
5  public class ContinueTest
6  {
7     public static void Main( string[] args )
8     {
9        for ( int count = 1; count <= 10; count++ ) // loop 10 times
10       {
11          if ( count == 5 ) // if count is 5,
12             continue; // skip remaining code in loop
13
14          Console.Write( "{0} ", count );
15       } // end for
16
17       Console.WriteLine( "\nUsed continue to skip displaying 5" );
18    } // end Main
19 } // end class ContinueTest
```

Skipping to the next iteration when `count` is `5`.

`Console.Write` skips `5` because of the `continue` statement.

```
1 2 3 4 6 7 8 9 10
Used continue to skip displaying 5
```