

Session: 4

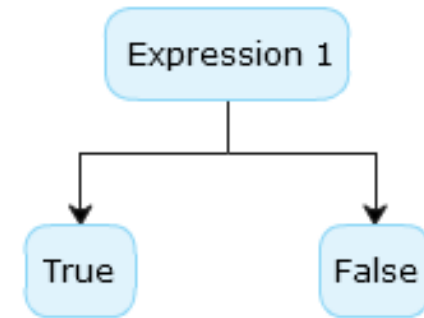
C# Programming Constructs

- ◆ Explain selection constructs
- ◆ Describe loop constructs
- ◆ Explain jump statements in C#



- ◆ A selection construct:

- ◆ Is a programming construct supported by C# that controls the flow of a program.
- ◆ Executes a particular block of statements based on a boolean condition, which is an expression returning true or false.
- ◆ Is referred to as a decision-making construct.
- ◆ Allow you to take logical decisions about executing different blocks of a program to achieve the required logical output.



- ◆ C# supports the following decision-making constructs:

- ◆ `if...else`
- ◆ `if...else...if`
- ◆ `Nested if`
- ◆ `switch...case`

The `if` Statement 1-3

- ◆ The `if` statement allows you to execute a block of statements after evaluating the specified logical condition.
- ◆ The `if` statement starts with the `if` keyword and is followed by the condition.
- ◆ If the condition evaluates to true, the block of statements following the `if` statement is executed.
- ◆ If the condition evaluates to false, the block of statements following the `if` statement is ignored and the statement after the block is executed.
- ◆ The following is the syntax for the `if` statement:

Syntax

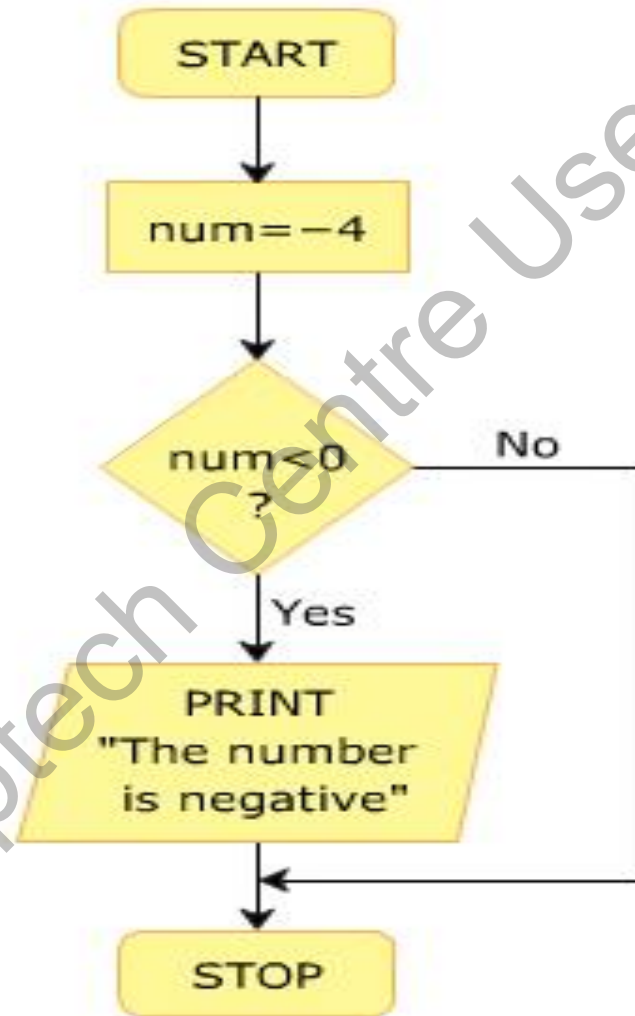
```
if (condition)
{
    // one or more statements;
}
```

where,

- ◆ **condition:** Is the boolean expression.
- ◆ **statements:** Are set of executable instructions executed when the boolean expression returns true.

The `if` Statement 2-3

- ◆ The following figure displays an example of the `if` construct:



- ◆ The following code displays whether the number is negative using the `if` statement:

Snippet

```
int num = -4;
if (num < 0)
{
    Console.WriteLine("The number is negative");
}
```

- ◆ In the code:
 - ◆ **num** is declared as an integer variable and is initialized to value -4.
 - ◆ The `if` statement is executed and the value of **num** is checked to see if it is less than 0.
 - ◆ The condition evaluates to `true` and the output “The number is negative” is displayed in the console window.

Output

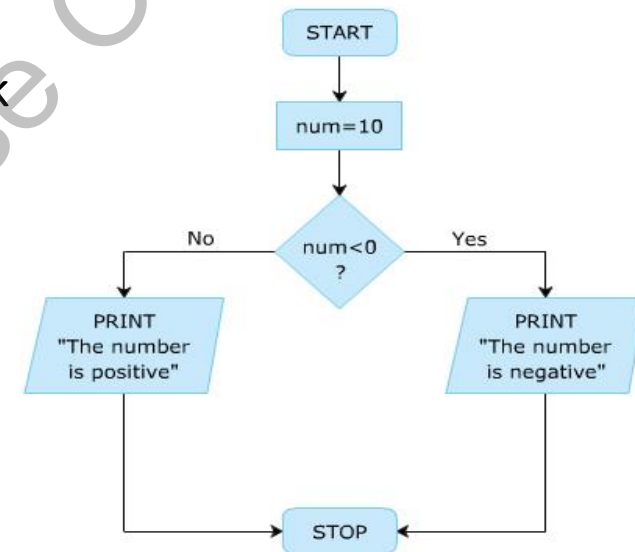
The number is negative.

The if...else Construct 1-2

- ◆ In some situations, it is required to define an action for a false condition by using an `if...else` construct.
- ◆ The `if...else` construct starts with the `if` block followed by an `else` block and the `else` block starts with the `else` keyword followed by a block of statements.
- ◆ If the condition specified in the `if` statement evaluates to false, the statements in the `else` block are executed.
- ◆ The following is the syntax for the `if...else` statement:

Syntax

```
if (condition)
{
    // one or more statements;
}
else
{
    //one or more statements;
}
```



The if...else Construct 2-2

- ◆ The following code displays whether a number is positive or negative using the `if . . . else` construct:

Snippet

```
int num = 10;
if (num < 0)
{
    Console.WriteLine("The number is negative");
}
else
{
    Console.WriteLine("The number is positive");
}
```

- ◆ In the code:
 - ◆ **num** is declared as an integer variable and is initialized to value 10.
 - ◆ The `if` statement is executed and the value of **num** is checked to see if it is less than 0.
 - ◆ The condition evaluates to `false` and the program control is passed to the `else` block and the output "The number is positive" is displayed in the console window.

Output

The number is positive.

The if...else...if Construct 1-3

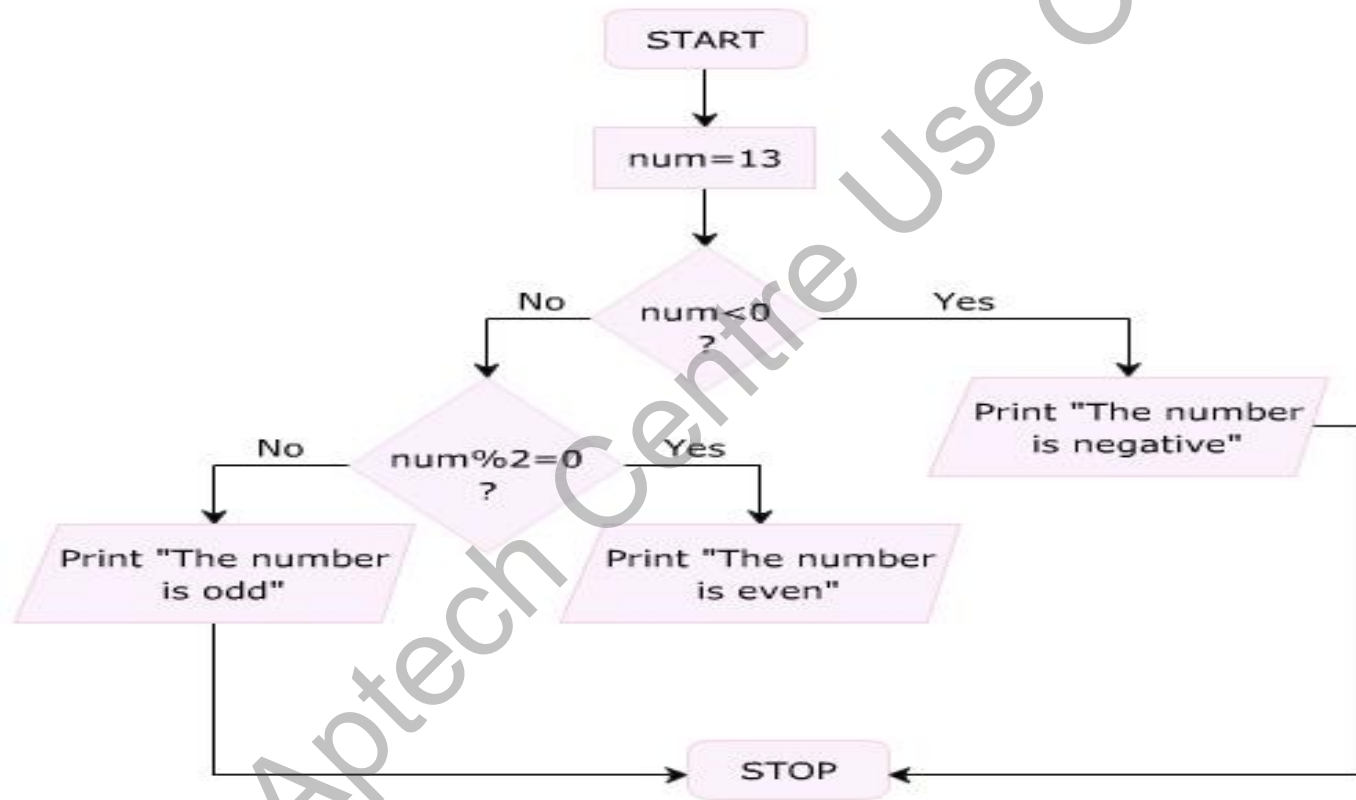
- ◆ The if...else...if construct allows you to check multiple conditions to execute a different block of code for each condition.
- ◆ It is also referred to as if-else-if ladder.
- ◆ The construct starts with the if statement followed by multiple else if statements followed by an optional else block.
- ◆ The conditions specified in the if...else...if construct are evaluated sequentially.
- ◆ The execution starts from the if statement. If a condition evaluates to false, the condition specified in the following else...if statement is evaluated.
- ◆ The syntax for the if...else...if construct is as follows:

Syntax

```
{  
    // one or more statements;  
}  
else if (condition)  
{  
    // one or more statements;  
}  
else  
{  
    // one or more statements;  
}
```

The if...else...if Construct 2-3

- ◆ The following figure displays an example of the if...else...if construct:



The `if...else...if` Construct 3-3

- ◆ The following code displays whether a number is negative, even, or odd using the `if...else...if` construct:

Snippet

```
int num = 13;
if (num < 0)
{
    Console.WriteLine("The number is negative");
}
else if ((num % 2) == 0)
{
    Console.WriteLine("The number is even");
}
```

- ◆ In the code:
 - ◆ **num** is declared as an integer variable and is initialized to value 13.
 - ◆ The `if` statement is executed and the value of **num** is checked to see if it is less than 0.
 - ◆ The condition evaluates to `false` and the program control is passed to the `else if` block.
 - ◆ The value of **num** is divided by 2 and the remainder is checked to see if it is 0. This condition evaluates to `false` and the control passes to the `else` block.
 - ◆ Finally, the output "The number is odd" is displayed in the console window.

Nested `if` Construct 1-3

- ◆ Following are the features of the nested `if` construct:

The nested `if` construct consists of multiple `if` statements.

The nested `if` construct starts with the `if` statement, which is called the outer `if` statement, and contains multiple `if` statements, which are called inner `if` statements.

In the nested `if` construct, the outer `if` condition controls the execution of the inner `if` statements. The compiler executes the inner `if` statements only if the condition in the outer `if` statement is `true`.

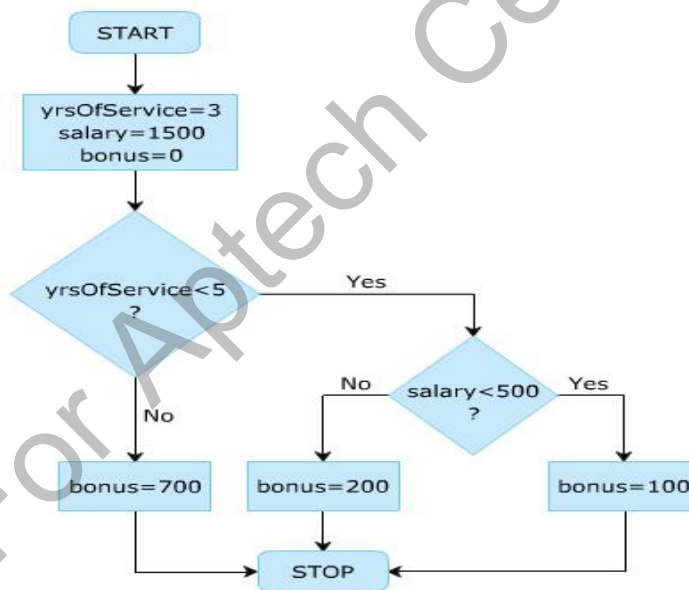
In addition, each inner `if` statement is executed only if the condition in its previous inner `if` statement is `true`.

- The following is the syntax for the nested `if` construct:

Syntax

```
if (condition)
{
    // one or more statements;
    if (condition)
    {
        // one or more statements;
        if (condition)
        {
            // one or more statements;
        }
    }
}
```

- The following figure displays an example of the nested `if` construct:



Nested if Construct 3-3

- ◆ The following figure displays the bonus amount using the nested if construct:

Snippet

```
int yrsOfService = 3;
double salary = 1500;
int bonus = 0;
if (yrsOfService < 5)
{
    if (salary < 500)
    {
        bonus = 100;
    }
    else
    {
        bonus = 200;
    }
}
else
{
    bonus = 700;
}

Console.WriteLine("Bonus amount: " + bonus);
```

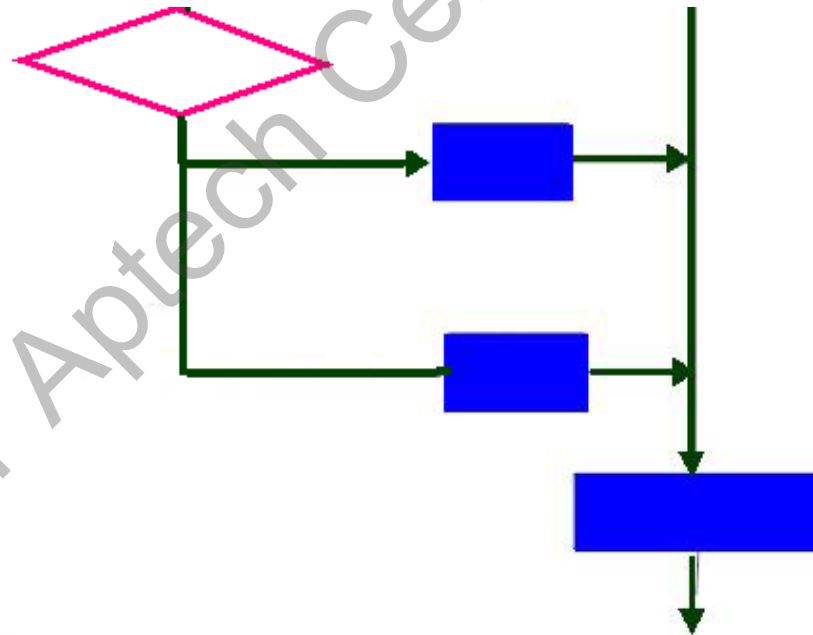
- ◆ In the code:
 - ◆ **yrsOfService** and **bonus** are declared as integer variables and initialized to values 3 and 0 respectively.
 - ◆ In addition, **salary** is declared as a **double** and is initialized to value 1500.
 - ◆ The first **if** statement is executed and the value of **yrsOfService** is checked to see if it is less than 5.
 - ◆ This condition is found to be true. Next, the value of **salary** is checked to see if it is less than 500.
 - ◆ This condition is found to be false. Hence, the control passes to the **else** block of the inner **if** statement. Finally, the bonus amount is displayed as 200.

Output

Bonus amount: 200

switch...case Construct 1-5

- ◆ A program is difficult to comprehend when there are too many `if` statements representing multiple selection constructs.
- ◆ To avoid using multiple `if` statements, in certain cases, the `switch...case` approach can be used as an alternative.
- ◆ The `switch...case` statement is used when a variable needs to be compared against different values.
- ◆ The following figure depicts the `switch...case` as a flow chart.



- ◆ The `switch...case` construct has the following components:
 - ◆ **switch:** The `switch` keyword is followed by an integer expression enclosed in parentheses. The expression must be of type `int`, `char`, `byte`, or `short`. The `switch` statement executes the `case` corresponding to the value of the expression.
 - ◆ **case:** The `case` keyword is followed by a unique integer constant and a colon. Thus, the `case` statement cannot contain a variable. The block following a particular `case` statement is executed when the `switch` expression and the `case` value match. Each `case` block must end with the `break` keyword that passes the control out of the `switch` construct.

- ◆ **default:** If no case value matches the switch expression value, the program control is transferred to the default block. This is the equivalent of the else block of the if...else...if construct.
- ◆ **break:** The break statement is optional and is used inside the switch...case statement to terminate the execution of the statement sequence. The control is transferred to the statement after the end of switch. If there is no break, execution flows sequentially into the next case statement. Sometimes, multiple case statements can be present without break statements between them.

switch...case Construct 4-5

- ◆ The following code displays the day of the week using the switch...case construct:

Snippet

```
int day = 5;
switch (day)
{
    case 1:
        Console.WriteLine("Sunday");
        break;
    case 2:
        Console.WriteLine("Monday");
        break;
    case 3:
        Console.WriteLine("Tuesday");
        break;

    case 4:
        Console.WriteLine("Wednesday");
        break;
    case 5:
        Console.WriteLine("Thursday");
        break;
    case 6:
        Console.WriteLine("Friday");
        break;
    case 7:
        Console.WriteLine("Saturday");
        break;
    default:
        Console.WriteLine("Enter a number between 1 to 7");
        break;
}
```

- ◆ In the code:
 - ◆ **day** is declared as an integer variable and is initialized to value 5.
 - ◆ The block of code following the `case 5` statement is executed because the value of **day** is 5 and the day is displayed as Thursday.
 - ◆ When the `break` statement is encountered, the control passes out of the `switch...case` construct.

Output

Thursday

Nested switch...case Construct 1-4

- ◆ C# allows the switch...case construct to be nested. That is, a case block of a switch...case construct can contain another switch...case construct.
- ◆ Also, the case constants of the inner switch...case construct can have values that are identical to the case constants of the outer construct.
- ◆ The following code demonstrates the use of nested switch:

Snippet

```
namespace Samsung
using System;
class Math
{
    static void Main(string[] args)
    {
        int numOne;
        int numTwo;
        int result = 0;
        Console.WriteLine("(1) Addition");
        Console.WriteLine("(2) Subtraction");
        Console.WriteLine("(3) Multiplication");
        Console.WriteLine("(4) Division");
        int input = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter value one");
        numOne = Convert.ToInt32(Console.ReadLine());
```

Nested switch...case Construct 2-4

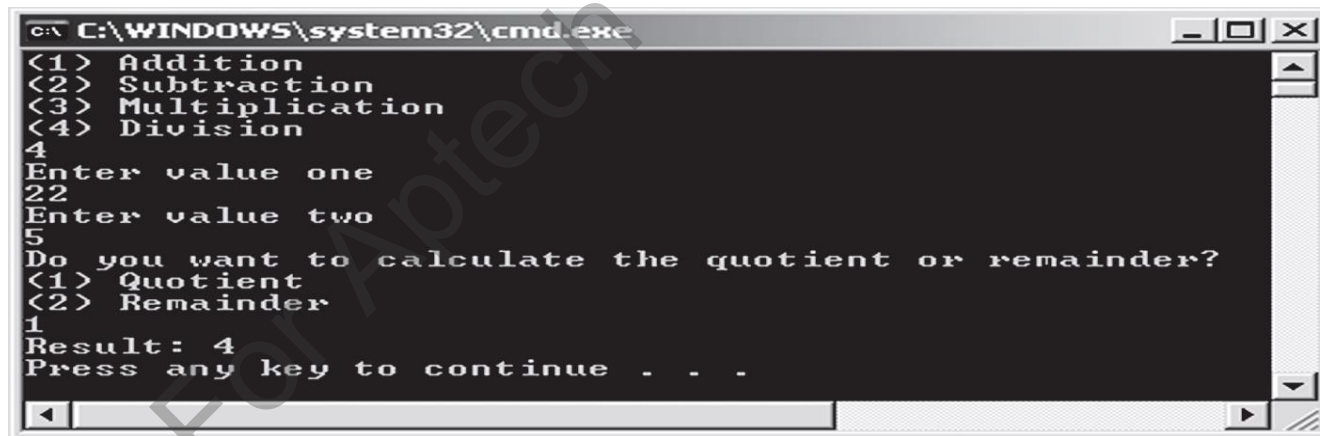
```
Console.WriteLine("Enter value two");
numTwo = Convert.ToInt32(Console.ReadLine());
switch (input)
{
    case 1:
        result = numOne + numTwo;
        break;
    case 2:
        result = numOne - numTwo;
        break;
    case 3:
        result = numOne * numTwo;
        break;
    switch (input)
    {
        case 1:
            result = numOne + numTwo;
            break;
        case 2:
            result = numOne - numTwo;
            break;
        case 3:
            result = numOne * numTwo;
            break;
        case 4:
            Console.WriteLine("Do you want to calculate
the quotient or remainder?");
            Console.WriteLine("(1) Quotient");
            Console.WriteLine("(2) Remainder");
            int choice = Convert.ToInt32
(Console.ReadLine());
```

Nested switch...case Construct 3-4

```
switch (choice)
{
    case 1:
        result = numOne / numTwo;
        break;
    case 2:
        result = numOne % numTwo;
        break;
    default:
        Console.WriteLine("Incorrect Choice");
        break;
}
break;
default:
    Console.WriteLine("Incorrect Choice");
    break;
}
Console.WriteLine("Result: " + result);
}
```

Nested switch...case Construct 4-4

- ◆ In the code:
 - ◆ The user is asked to choose the desired arithmetical operation. The user then enters the numbers on which the operation is to be performed.
 - ◆ Using the `switch...case` construct, based on the input from the user, the appropriate operation is performed.
 - ◆ The `case` block for the division option uses an inner `switch...case` construct to either calculate the quotient or the remainder of the division as per the choice selected by the user.
- ◆ The following figure demonstrates the nested `switch`:



```
c:\WINDOWS\system32\cmd.exe
<1> Addition
<2> Subtraction
<3> Multiplication
<4> Division
4
Enter value one
22
Enter value two
5
Do you want to calculate the quotient or remainder?
<1> Quotient
<2> Remainder
1
Result: 4
Press any key to continue . . .
```

- ◆ Following are the 'no-fall-through' rules:

In C#, the flow of execution from one `case` statement is not allowed to continue to the next `case` statement and is referred to as the 'no-fall-through' rule of C#.

Thus, the list of statements inside a `case` block generally ends with a `break` or a `goto` statement, which causes the control of the program to exit the `switch...case` construct and go to the statement following the construct.

The last `case` block (or the `default` block) also needs to have a statement like `break` or `goto` to explicitly take the control outside the `switch...case` construct.

C# introduced the no fall-through rule to allow the compiler to rearrange the order of the `case` blocks for performance optimization.

- ◆ Although C# does not permit the statement sequence of one `case` block to fall through to the next, it does allow empty `case` blocks (`case` blocks without any statements) to fall through.

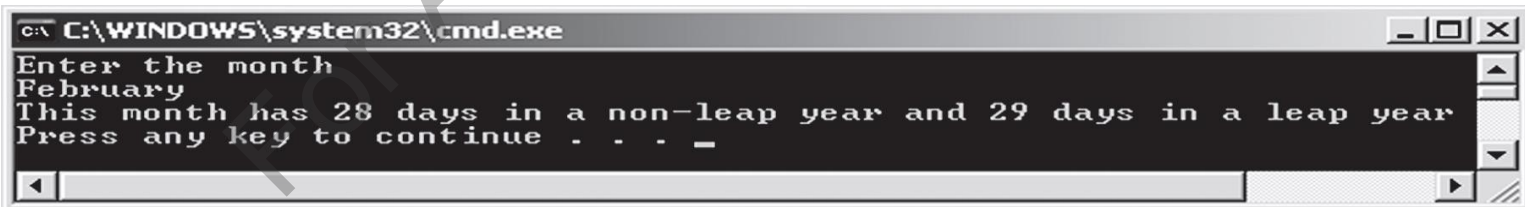
- ◆ A multiple case statement can be made to execute the same code sequence, as shown in the following code:

Snippet

```
namespace Samsung
using System;
class Months
{
    static void Main(string[] args)
    {
        string input;
        Console.WriteLine("Enter the month");
        input = Console.ReadLine().ToUpper();
        switch (input)
        {
            case "JANUARY":
            case "MARCH":
            case "MAY":
            case "JULY":
            case "AUGUST":
            case "OCTOBER":
            case "DECEMBER":
                Console.WriteLine ("This month has 31 days");
                break;
            case "APRIL":
            case "JUNE":
            case "SEPTEMBER":
```

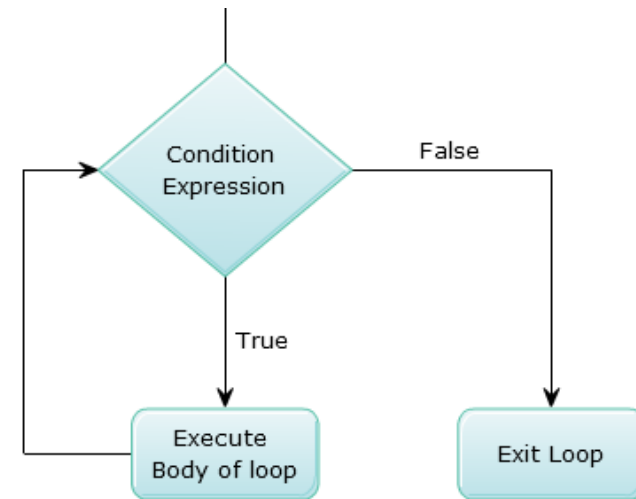
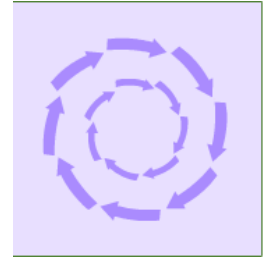
```
        case "NOVEMBER":  
            Console.WriteLine ("This month has 30 days");  
            break;  
        case "FEBRUARY":  
            Console.WriteLine("This month has 28 days in  
a non-leap year and 29 days in a leap year");  
            break;  
        default:  
            Console.WriteLine ("Incorrect choice");  
            break;  
    }  
}
```

- ◆ In the code:
 - ◆ The `switch...case` construct is used to display the number of days in a particular month.
 - ◆ Here, all months having 31 days have been stacked together to execute the same statement.
 - ◆ Similarly, all months having 30 days have been stacked together to execute the same statement.
 - ◆ Here, the no-fall-through rule is not violated as the stacked `case` statements execute the same code.
 - ◆ By stacking the `case` statements, unnecessary repetition of code is avoided.
- ◆ The figure shows the output of multiple case statements:



```
C:\WINDOWS\system32\cmd.exe  
Enter the month  
February  
This month has 28 days in a non-leap year and 29 days in a leap year  
Press any key to continue . . .
```

- ◆ Loops allow you to execute a single statement or a block of statements repetitively.
- ◆ The most common uses of loops include displaying a series of numbers and taking repetitive input.
- ◆ In software programming, a loop construct contains a condition that helps the compiler identify the number of times a specific block will be executed.
- ◆ If the condition is not specified, the loop continues infinitely and is termed as an infinite loop.
- ◆ The loop constructs are also referred to as iteration statements.
- ◆ C# supports four types of loop constructs such as:
 - ◆ The `while` loop
 - ◆ The `do..while` loop
 - ◆ The `for` loop
 - ◆ The `foreach` loop



The while Loop 1-3

- ◆ The `while` loop is used to execute a block of code repetitively as long as the condition of the loop remains true.
- ◆ The `while` loop consists of the `while` statement, which begins with the `while` keyword followed by a boolean condition.
- ◆ If the condition evaluates to true, the block of statements after the `while` statement is executed.
- ◆ After each iteration, the control is transferred back to the `while` statement and the condition is checked again for another round of execution.
- ◆ When the condition is evaluated to false, the block of statements following the `while` statement is ignored and the statement appearing after the block is executed by the compiler.
- ◆ The following is the syntax of the `while` loop:

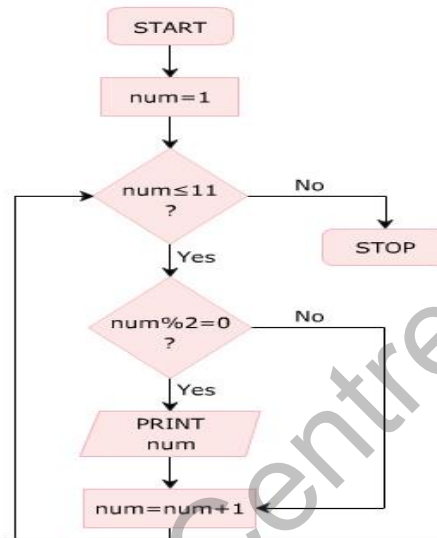
Syntax

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

where:

- ◆ `condition`: Specifies the boolean expression.

- The following figure depicts an example of a `while` loop using a flowchart:



- The following code displays even numbers from 1 to 10 using the `while` loop:

Snippet

```
public int num = 1;
Console.WriteLine("Even Numbers");
while (num <= 11)
{
    if ((num % 2) == 0)
    {
        Console.WriteLine(num);
    }
    num = num + 1;
}
```

- ◆ In the following code:
 - ◆ **num** is declared as an integer variable and initialized to value 1.
 - ◆ The condition in the `while` loop is checked, which specifies that the value of **num** variable should be less than or equal to 11.
 - ◆ If this condition is true, the value of the **num** variable is divided by 2 and the remainder is checked to see if it is 0.
 - ◆ If the remainder is 0, the value of the variable **num** is displayed in the console window and the variable **num** is incremented by 1.
 - ◆ Then, the program control is passed to the `while` statement to check the condition again.
 - ◆ When the value of **num** becomes 12, the `while` loop terminates as the loop condition becomes false.

Output

Even Numbers

2

4

6

8

10

Nested while Loop

- ◆ A `while` loop can be created within another `while` loop to create a nested `while` loop structure.
- ◆ The following code demonstrates the use of nested `while` loops to create a geometric pattern:

Snippet

```
using System;
class Pattern
{
    static void Main(string[] args)
    {
        int i = 0;
        int j;
        while (i <= 5)
        {
            j = 0;
            while (j <= i)
            {
                Console.Write("*");
                j++;
            }
            Console.WriteLine();
            i++;
        }
    }
}
```

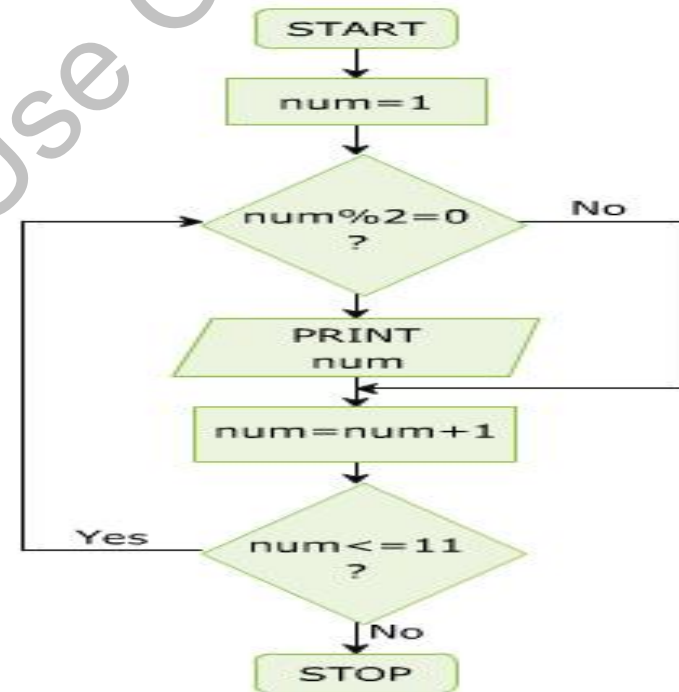
- ◆ In the code:
 - ◆ A pattern of a right-angled triangle is created using the asterisk (*) symbol. This is done using the nested `while` loop.

The do-while Loop 1-2

- ◆ The `do-while` loop is similar to the `while` loop; however, it is always executed at least once without the condition being checked.
- ◆ The loop starts with the `do` keyword and is followed by a block of executable statements.
- ◆ The `while` statement along with the condition appears at the end of this block.
- ◆ The statements in the `do-while` loop are executed as long as the specified condition remains true.
- ◆ When the condition evaluates to false, the block of statements after the `do` keyword are ignored and the immediate statement after the `while` statement is executed.
- ◆ The following is the syntax of the `do-while` loop:

Syntax

```
do
{
// one or more statements;
} while (condition);
```



The do-while Loop 2-2

- ◆ The following code displays even numbers from 1 to 10 using the do-while loop:

Snippet

```
int num = 1;
Console.WriteLine("EvenNumbers");
do
{
    if ((num % 2) == 0)
    {
        Console.WriteLine(num);
    }
    num = num + 1;
} while (num <= 11);
```

- ◆ In the code:
 - ◆ **num** is declared as an integer variable and is initialized to value 1.
 - ◆ In the do block, without checking any condition, the value of **num** is first divided by 2 and the remainder is checked to see if it is 0.
 - ◆ If the remainder is 0, the value of **num** is displayed and it is then incremented by 1.
 - ◆ Then, the condition in the while statement is checked to see if the value of **num** is less than or equal to 11.
 - ◆ If this condition is true, the do-while loop executes again.
 - ◆ When the value of **num** becomes 12, the do-while loop terminates.

Output

Even Numbers

2

4

6

8

10

- ◆ The `for` statement is similar to the `while` statement in its function.
- ◆ The statements within the body of the loop are executed as long as the condition is true.
- ◆ Here too, the condition is checked before the statements are executed.
- ◆ The following is the syntax of the `for` loop:

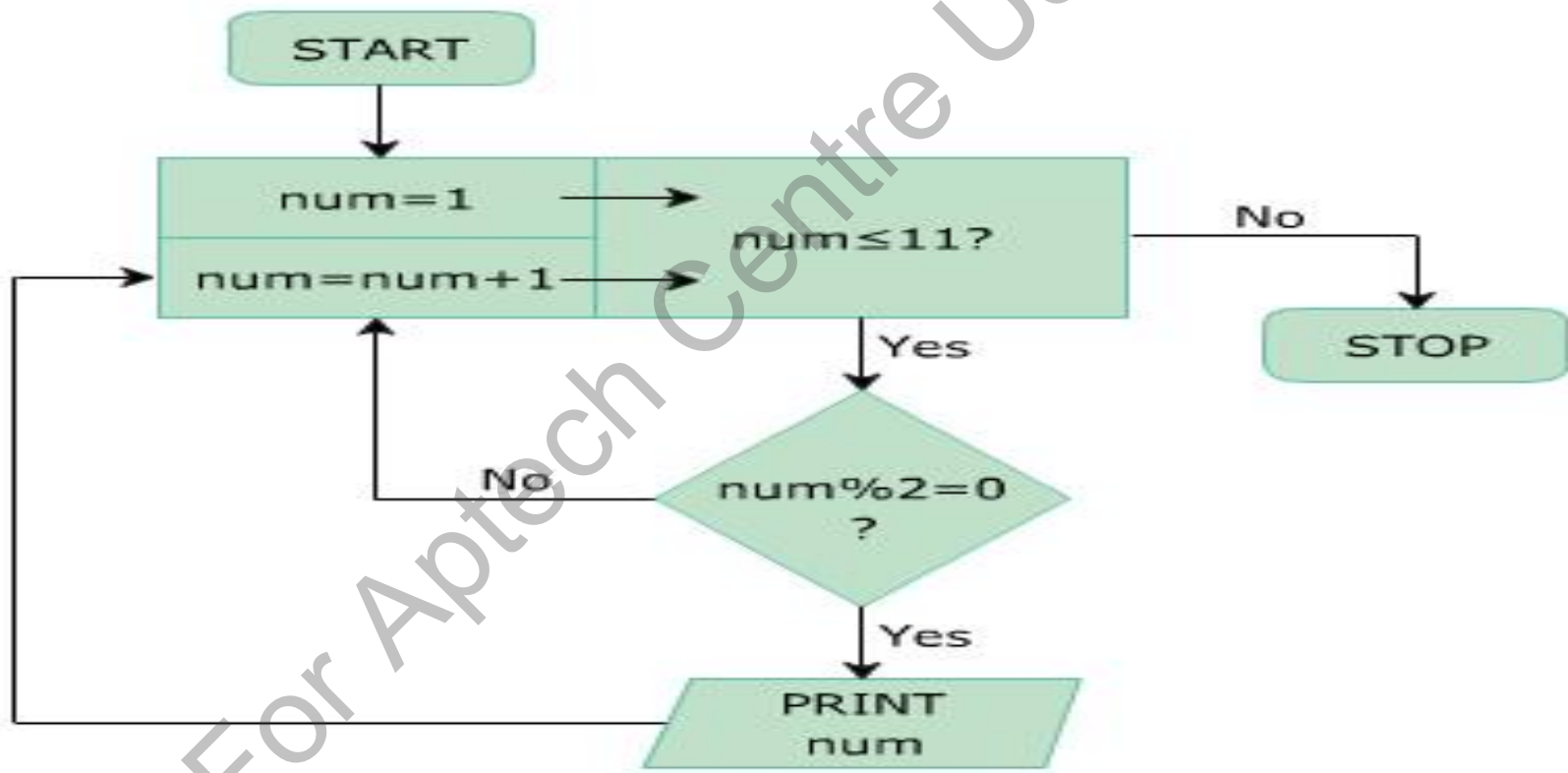
Syntax

```
for (initialization; condition; increment/decrement)
{
    // one or more statements;
}
```

where,

- ◆ `initialization`: Initializes the variable(s) that will be used in the `condition`.
- ◆ `condition`: Comprises the condition that is tested before the statements in the loop are executed.
- ◆ `increment/decrement`: Comprises the statement that changes the value of the variable(s) to ensure that the condition specified in the condition section is reached. Typically, increment and decrement operators such as `++`, `--` and shortcut operators such as `+=` or `-=` are used in this section. Note that there is no semicolon at the end of the increment/decrement expressions.

- ◆ The working of the `for` loop can be depicted using the following flowchart:



- ◆ The following code displays even numbers from 1 to 10 using the `for` loop:

Snippet

```
int num;
Console.WriteLine("Even Numbers");
for (num = 1; num <= 11; num++) {
    if ((num % 2) == 0)
    {
        Console.WriteLine(num);
    }
}
```

- ◆ In the code:
 - ◆ **num** is declared as an integer variable and it is initialized to value 1 in the `for` statement.
 - ◆ The condition specified in the `for` statement is checked for value of **num** to be less than or equal to 11.
 - ◆ If this condition is true, value of **num** is divided by 2 and the remainder is checked to see if it is 0.
 - ◆ If this condition is true, the control is passed to the `for` statement again.
 - ◆ Here, the value of **num** is incremented and the condition is checked again.
 - ◆ When the value of **num** becomes 12, the condition of the `for` loop becomes false and the loop terminates.

Nested for Loops

- ◆ The nested `for` loop consists of multiple `for` statements. When one `for` loop is enclosed inside another `for` loop, the loops are said to be nested.
- ◆ The `for` loop that encloses the other `for` loop is referred to as the outer `for` loop whereas the enclosed `for` loop is referred to as the inner `for` loop.
- ◆ The outer `for` loop determines the number of times the inner `for` loop will be invoked.
- ◆ The following code demonstrates a 2X2 matrix using nested `for` loops:

Snippet

```
int rows = 2;
int columns = 2;
for (int i = 0; i < rows; i++)
{
    for (int j = 0; j < columns; j++)
    {
        Console.WriteLine("{0} ", i*j);
    }
    Console.WriteLine();
}
```

- ◆ In the code:
 - ◆ The `rows` and `columns` are declared as integer variables and are initialized to value 2.
 - ◆ In addition, `i` and `j` are declared as integer variables in the outer and inner `for` loops respectively and both are initialized to 0. When the outer `for` loop is executed, the value of `i` is checked to see if it is less than 2.
 - ◆ Here, the condition is true; hence, the inner `for` loop is executed. In this loop, the value of `j` is checked to see if it is less than 2.
 - ◆ As long as it is less than 2, the product of the values of `i` and `j` is displayed in the console window. This inner `for` loop executes until `j` becomes greater than or equal to 2, at which time, the control passes to the outer `for` loop.

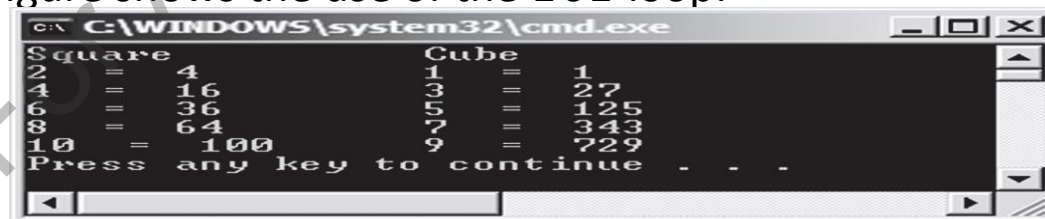
The for Loop with Multiple Loop Control Variables

- ◆ The `for` loop allows the use of multiple variables to control the loop.
- ◆ The following code demonstrates the use of the `for` loop with two variables:

Snippet

```
using System;
class Numbers
{
    static void Main(string[] args)
    {
        Console.WriteLine("Square \t\tCube");
        for (int i = 1, j = 0; i < 11; i++, j++)
        {
            if ((i % 2) == 0)
            {
                Console.Write("{0} = {1} \t", i, (i * i));
                Console.Write("{0} = {1} \n", j, (j * j * j));
            }
        }
    }
}
```

- ◆ In the code:
 - ◆ The initialization portion as well as the increment/decrement portion of the `for` loop definition contain two variables, `i` and `j`.
 - ◆ These variables are used in the body of the `for` loop, to display the square of all even numbers, and the cube of all odd numbers between 1 and 10.
- ◆ The following figure shows the use of the `for` loop:



```
C:\WINDOWS\system32\cmd.exe
Square          Cube
2 = 4           1 = 1
4 = 16          3 = 27
6 = 36          5 = 125
8 = 64          7 = 343
10 = 100        9 = 729
Press any key to continue . . .
```

The for Loop with Missing Portions 1-5

- ◆ The `for` loop definition can be divided into three portions, the initialization, the conditional expression, and the increment/decrement portion.
- ◆ C# allows the creation of the `for` loop even if one or more portions of the loop definition are omitted.
- ◆ In fact, the `for` loop can be created with all the three portions omitted.
- ◆ The following code demonstrates a `for` loop that leaves out or omits the increment/decrement portion of the loop definition:

Snippet

```
using System;
class Investment
{
    static void Main(string[] args)
    {
        int investment;
        int returns;
        int expenses;
        int profit;
        int counter = 0;
        for (investment=1000, returns=0; returns<investment;)
        {
            Console.WriteLine("Enter the monthly expenditure");
            expenses = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Enter the monthly profit");
            profit = Convert.ToInt32(Console.ReadLine());
            investment += expenses;
            returns += profit;
            counter++;
        }
        Console.WriteLine("Number of months to break even: "
+ counter);
    }
}
```


The `for` Loop with Missing Portions 2-5

◆ In the code:

- ◆ The `for` loop is used to calculate the number of months it has taken for a business venture to recover its investment and break even.
- ◆ The expenditure and the profit `for` every month is accepted from the user and stored in variables **`expenses`** and **`profit`** respectively.
- ◆ The total investment is calculated as the initial investment plus the monthly expenses. The total returns are calculated as the sum of the profits for each month.
- ◆ The conditional expression of the loop terminates the loop once the total returns of the business becomes greater than or equal to the total investment.
- ◆ The code then prints the total number of months it has taken the business to break even.
- ◆ The number of months equals the number of iterations of the `for` loop as each iteration represents a new month.

The for Loop with Missing Portions 3-5

- ◆ The following code demonstrates a `for` loop that omits the initialization portion as well as the increment/decrement portion of the loop definition:

Snippet

```
int investment = 1000;
int returns = 0;
for (; returns < investment; )
{
    // for loop statements
    ...
}
```

- ◆ In the code:
 - ◆ The initialization of the variables **investment** and **returns** has been done prior to the `for` loop definition. Hence, the loop has been defined with the initialization portion left empty.
 - ◆ If the conditional expression portion of the `for` loop is omitted, the loop becomes an infinite loop. Such loops can then be terminated using jump statements such as `break` or `goto` to exit the loop.
- ◆ The following code demonstrates the use of an infinite `for` loop:

Snippet

```
using System;
class Summation
{
    static void Main(string[] args)
    {
        char c;
        int numOne;
        int numTwo;
        int result;
        for ( ; ; )
```

The for Loop with Missing Portions 4-5

```
{
    Console.WriteLine("Enter number one");
    numOne = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Enter number two");
    numTwo = Convert.ToInt32(Console.ReadLine());
    result = numOne + numTwo;

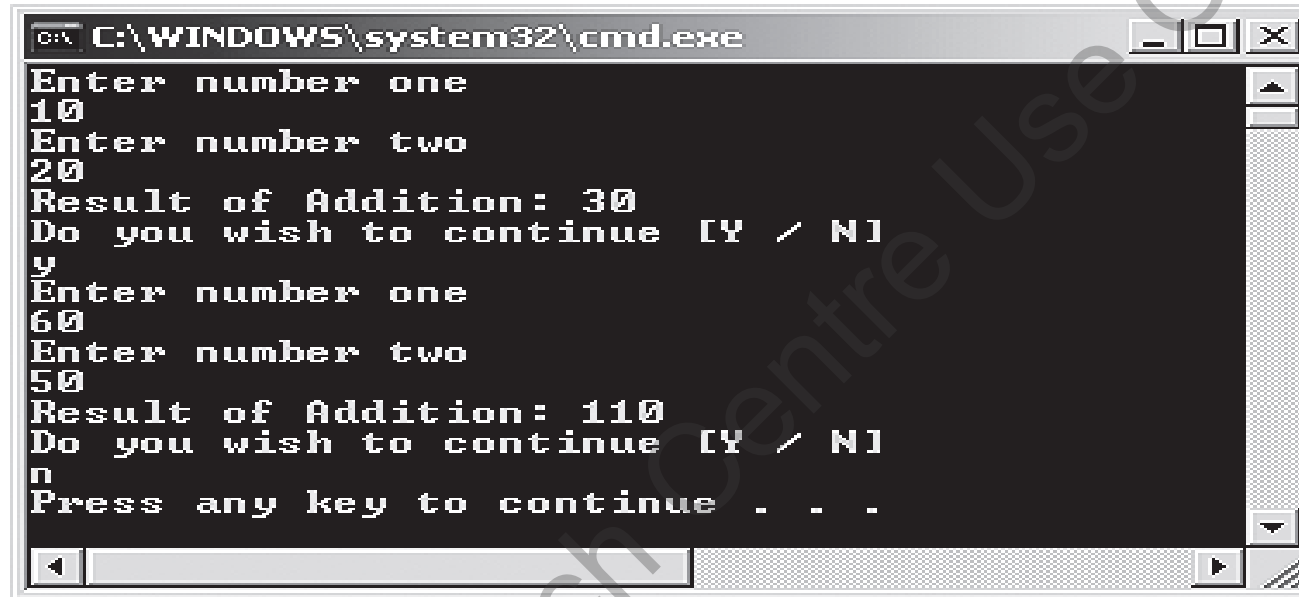
    Console.WriteLine("Result of Addition: " +
        result);
    Console.WriteLine("Do you wish to continue [Y / N]");
    c = Convert.ToChar(Console.ReadLine());
    if (c == 'Y' || c == 'y')
    {
        continue;
    }
    else
    {
        break;
    }
}
```

◆ In the code:

- ◆ An infinite `for` loop is used to repetitively accept two numbers from the user.
- ◆ These numbers are added and their result printed on the console.
- ◆ After each iteration of the loop, the `if` construct is used to check whether the user wishes to continue or not.
- ◆ If the answer is Y or y (denoting yes), the loop is executed again, else the loop is exited using the `break` statement.

The for Loop with Missing Portions 5-5

- ◆ The following figure shows the use of an infinite for loop:



```
C:\WINDOWS\system32\cmd.exe
Enter number one
10
Enter number two
20
Result of Addition: 30
Do you wish to continue [Y / N]
y
Enter number one
60
Enter number two
50
Result of Addition: 110
Do you wish to continue [Y / N]
n
Press any key to continue . . .
```

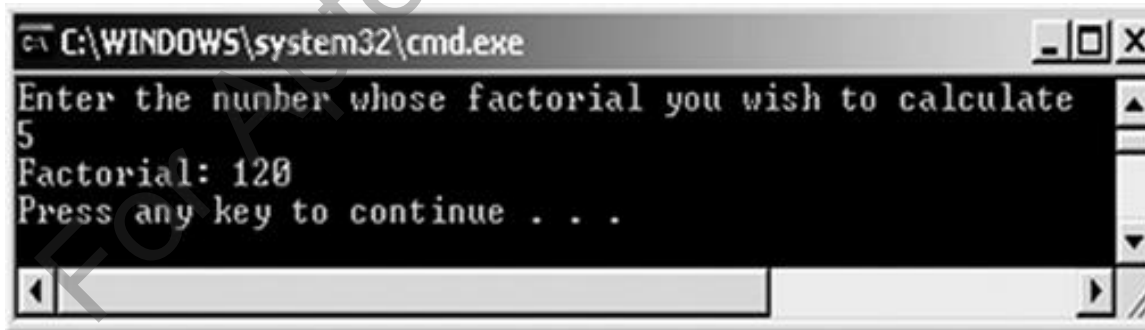
The for Loop without a Body

- ◆ In C#, a `for` loop can be created without a body.
- ◆ Such a loop is created when the operations performed within the body of the loop can be accommodated within the loop definition itself.
- ◆ The following code demonstrates the use of a `for` loop without a body:

Snippet

```
using System;
class Factorial
{
    static void Main(string[] args)
    {
        int fact = 1;
        int num, i;
        Console.WriteLine("Enter the number whose factorial you
        wish to calculate");
        num = Convert.ToInt32(Console.ReadLine());
        for (i = 1; i <= num; fact *= i++);
        Console.WriteLine("Factorial: " + fact);
    }
}
```

- ◆ In the code:
 - ◆ The process of calculating the factorial of a user-given number is done entirely in the `for` loop definition; the loop has no body.
- ◆ The following figure shows the `for` loop without a body:



Declaring Loop Control Variables in the Loop Definition 1-3

- ◆ The loop control variables are often created for loops such as the `for` loop. Once the loop is terminated, there is no further use of these variables. In such cases, these variables can be created within the initialization portion of the `for` loop definition.
- ◆ The following is the syntax for declaring the loop control variable within the loop definition:

Syntax

```
foreach (<datatype><identifier> in <list>)  
{  
    // one or more statements;  
}
```

where,

- ◆ `datatype`: Specifies the data type of the elements in the list.
- ◆ `identifier`: Is an appropriate name for the collection of elements.
- ◆ `list`: Specifies the name of the list.

Declaring Loop Control Variables in the Loop Definition 2-3

- ◆ The following figure displays the employee names using the `foreach` loop:

Snippet

```
string[] employeeNames = { "Maria", "Wilson", "Elton", "Garry" };  
Console.WriteLine("Employee Names");  
foreach (string names in employeeNames)  
{  
    Console.WriteLine("{0} ", names);  
}
```

- ◆ In the code:
 - ◆ The list of employee names is declared in an array of `string` variables called **employeeNames**.
 - ◆ In the `foreach` statement, the data type is declared as `string` and the identifier is specified as `names`.
 - ◆ This variable refers to all values from the **employeeNames** array.
 - ◆ The `foreach` loop displays names of the employees in the order in which they are stored.

Output

Employee Names

Maria

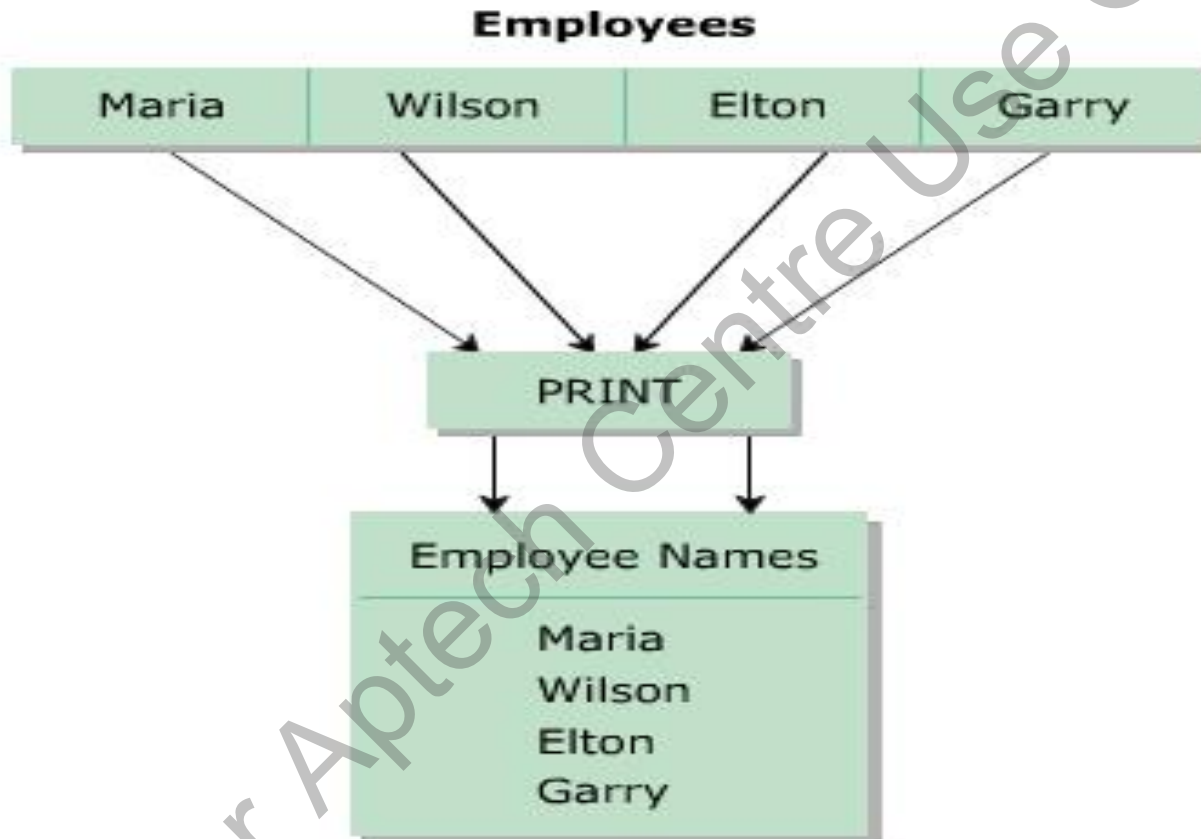
Wilson

Elton

Garry

Declaring Loop Control Variables in the Loop Definition 3-3

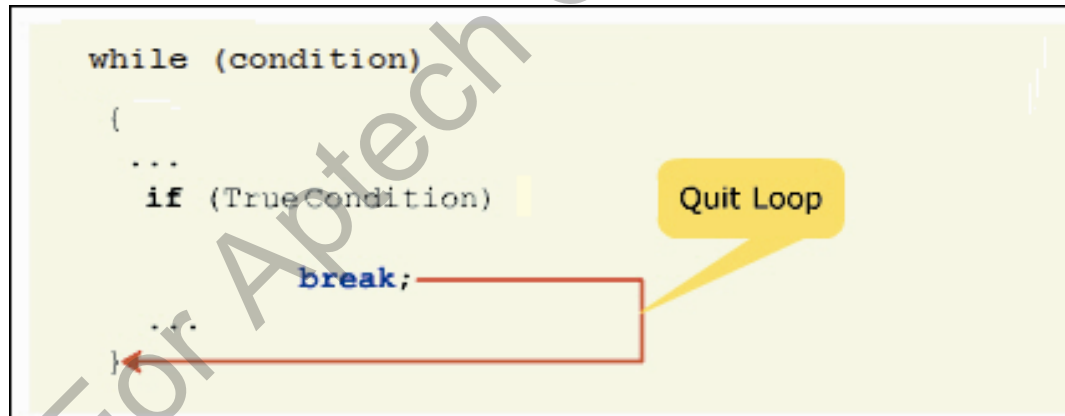
- ◆ The following figure depicts the working of the `foreach` loop:



- ◆ Jump statements are used to transfer control from one point in a program to another.
- ◆ There will be situations where you need to exit out of a loop prematurely and continue with the program.
- ◆ In such cases, jump statements are used. Jump statement unconditionally transfer control of a program to a different location.
- ◆ The location to which a jump statement transfers control is called the target of the jump statement.
- ◆ C# supports four types of jump statements. These are as follows:
 - ◆ `break`
 - ◆ `continue`
 - ◆ `goto`
 - ◆ `return`

The break Statement 1-2

- ◆ The `break` statement is used in the selection and loop constructs.
- ◆ It is most widely used in the `switch...case` construct and in the `for` and `while` loops.
- ◆ The `break` statement is denoted by the `break` keyword. In the `switch...case` construct, it is used to terminate the execution of the construct.
- ◆ In loops, it is used to exit the loop without testing the loop condition.
- ◆ In this case, the control passes to the next statement following the loop.
- ◆ The following figure depicts the `break` statement:



The break Statement 2-2

- ◆ The following figure displays a prime number using the `while` loop and the `break` statement:

Snippet

```
int numOne = 17;
int numTwo = 2;
while(numTwo <= numOne-1)
{
    if(numOne % numTwo == 0)
    {
        Console.WriteLine("Not a Prime Number");
        break;
    }
    numTwo++;
}
if(numTwo == numOne)
{
    Console.WriteLine("Prime Number");
}
```

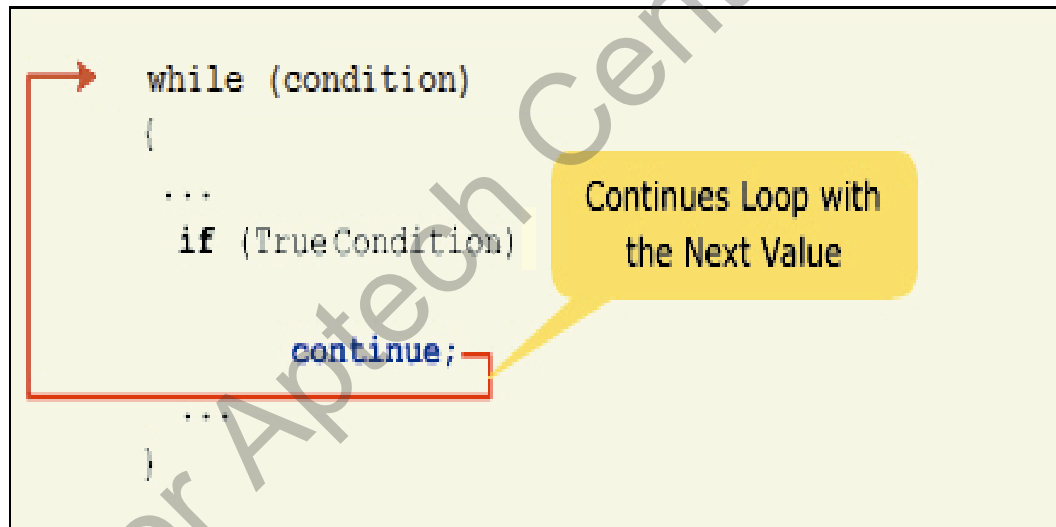
- ◆ In the code:
 - ◆ The **numOne** and **numTwo** are declared as integer variables and are initialized to values 17 and 2 respectively.
 - ◆ In the `while` statement, if the condition is true, the inner `if` condition is checked. If this condition evaluates to true, the program control passes to the `if` statement outside the `while` loop.
 - ◆ If the condition is false, the value of **numTwo** is incremented and the control passes to the `while` statement again.

Output

Prime Number

The continue Statement 1-2

- ◆ The `continue` statement is most widely used in the loop constructs and is denoted by the `continue` keyword.
- ◆ The `continue` statement is used to end the current iteration of the loop and transfer the program control back to the beginning of the loop. The statements of the loop following the `continue` statement are ignored in the current iteration.
- ◆ The following figure displays the working of the `continue` statement:



The continue Statement 2-2

- ◆ The following code displays the even numbers in the range of 1 to 10 using the `for` loop and the `continue` statement:

Snippet

```
Console.WriteLine("Even numbers in the range of 1-10");
for (int i=1; i<=10; i++)
{
    if (i % 2 != 0)
    {
        continue;
    }
    Console.Write(i + " ");
}
```

- ◆ In the code:
 - ◆ `i` is declared as an integer and is initialized to value 1 in the `for` loop definition.
 - ◆ In the body of the loop, the value of `i` is divided by 2 and the remainder is checked to see if it is equal to 0.
 - ◆ If the remainder is zero, the value of `i` is displayed as the value is an even number.
 - ◆ If the remainder is not equal to 0, the `continue` statement is executed and the program control is transferred to the beginning of the `for` loop.

Even numbers in the range of 1-10.

Output

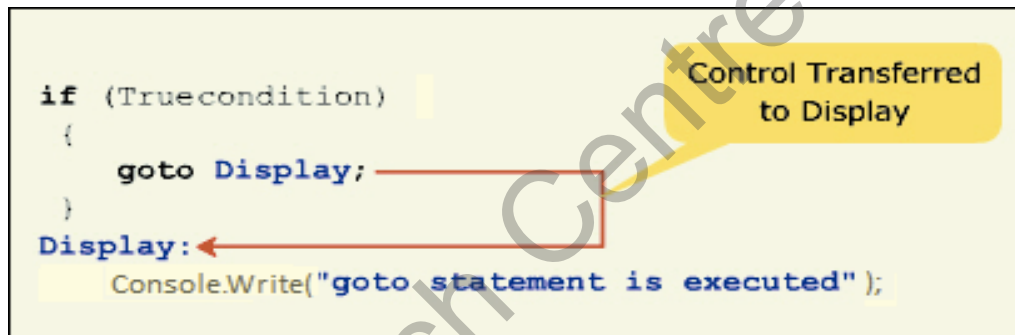
2 4 6 8 10

The goto Statement 1-4

- ◆ The `goto` statement allows you to directly execute a labeled statement or a labeled block of statements.
- ◆ A labeled block or a labeled statement starts with a label. A label is an identifier ending with a colon.
- ◆ A single labeled block can be referred by more than one `goto` statements.
- ◆ The `goto` statement is denoted by the `goto` keyword.
- ◆ The following code displays the `goto` statement:

```
if (Truecondition)
{
    goto Display;
}
Display:
    Console.WriteLine("goto statement is executed");
```

Control Transferred to Display

A diagram illustrating the execution of a goto statement. It shows a code snippet with an if block containing a goto Display; statement. Below the if block is a labeled block starting with Display: followed by a Console.WriteLine statement. A red arrow points from the goto statement to the Display: label. A yellow callout box with the text "Control Transferred to Display" points to the arrow.

- ◆ The following figure displays the output “Hello World” five times using the `goto` statement:

Snippet

```
int i = 0;
display:
Console.WriteLine("Hello World");
i++;
if (i < 5)
{
    goto display;
}
```

- ◆ In the code:
 - ◆ `i` is declared as an integer and is initialized to value 0.
 - ◆ The program control transfers to the display label and the message "Hello World" is displayed.
 - ◆ Then, the value of `i` is incremented by 1 and is checked to see if it is less than 5.
 - ◆ If this condition evaluates to true, the `goto` statement is executed and the program control is transferred to the display label. If the condition evaluates to false, the program ends.

Output

```
Hello World  
Hello World  
Hello World  
Hello World  
Hello World
```

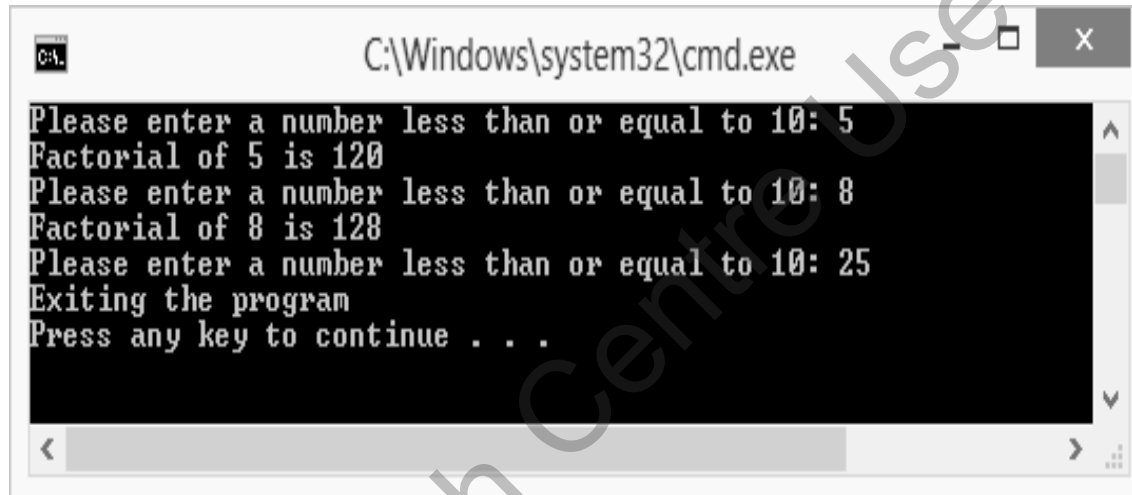

- ◆ The following code demonstrates the use of goto statement to break out of a nested loop:

Snippet

```
using System;
class Factorial
{
    static void Main(string[] args)
    {
        byte num = 0;
        while (true)
        {
            byte fact = 1;
            Console.Write("Please enter a number less than or equal to 10: ");
            num = Convert.ToByte(Console.ReadLine());
            if (num < 0)
            {
                goto stop;
            }
            for (byte j = num; j > 0; j--)
            {
                if (j > 10)
                {
                    goto stop;
                }
                fact *= j;
            }
            Console.WriteLine("Factorial of {0} is {1}", num, fact);
        }
        stop:
        Console.WriteLine("Exiting the program");
    }
}
```

The goto Statement 4-4

- ◆ The following figure shows the use of goto statement:



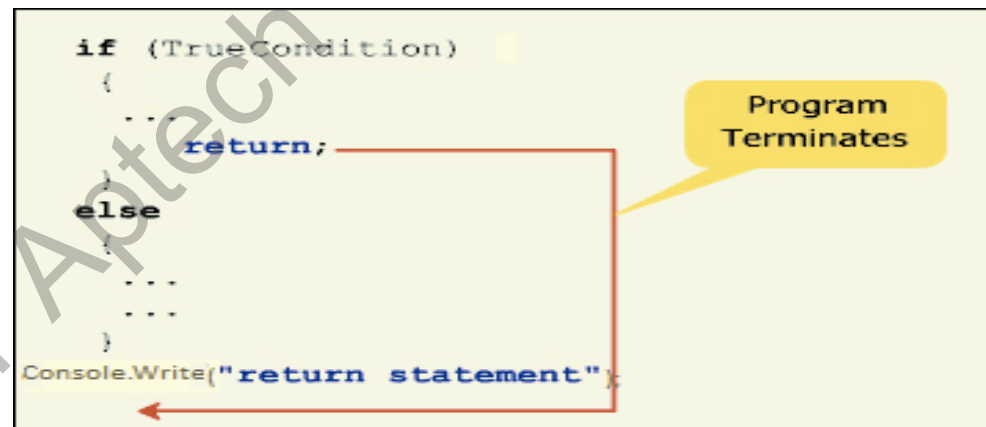
```
C:\Windows\system32\cmd.exe

Please enter a number less than or equal to 10: 5
Factorial of 5 is 120
Please enter a number less than or equal to 10: 8
Factorial of 8 is 128
Please enter a number less than or equal to 10: 25
Exiting the program
Press any key to continue . . .
```

The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window contains the output of a C# program that uses the goto statement. The program prompts the user to enter a number less than or equal to 10, calculates its factorial, and displays the result. The user enters 5, 8, and 25 in three separate runs. The program then exits and prompts the user to press any key to continue.

The return Statement 1-3

- ◆ The `return` statement is used to return a value of an expression or is used to transfer the control to the method from which the currently executing method was invoked.
- ◆ The `return` statement is denoted by the `return` keyword. The `return` statement must be the last statement in the method block.
- ◆ The following figure displays the working of the `return` statement:



The return Statement 2-3

- ◆ The following code displays the cube of a number using the return statement:

Snippet

```
static void Main(string[] args)
{
    int num = 23;
    Console.WriteLine("Cube of {0} = {1}", num, Cube(num));
}
static int Cube(int n)
{
    return (n * n * n);
}
```

- ◆ In the code:
 - ◆ The variable **num** is declared as an integer and is initialized to value 23.
 - ◆ The **Cube ()** method is invoked by the `Console.WriteLine()` method.
 - ◆ At this point, the program control passes to the **Cube ()** method, which returns the cube of the specified value. The return statement returns the calculated cube value back to the `Console.WriteLine()` method, which displays the calculated cube of 23.

Output

Cube of 23 = 12167

The return Statement 3-3

- ◆ The following code demonstrates the use of the `return` statement to terminate the program:

Snippet

```
class Factorial
{
    static void Main(string[] args)
    {
        int yrsOfService = 5;
        double salary = 1250;
        double bonus = 0;

        if (yrsOfService <= 5)
        {
            bonus = 50;
            return;
        }
        else
        {
            bonus = salary * 0.2;
        }
        Console.WriteLine("Salary amount: " + salary);
        Console.WriteLine("Bonus amount: " + bonus);
    }
}
```

- ◆ In the code:
 - ◆ **yrsOfService** is declared as an integer variable and is initialized to value 5.
 - ◆ In addition, **salary** and **bonus** are declared as `double` and are initialized to values 1250 and 0 respectively.
 - ◆ The value of the **yrsOfService** variable is checked to see if it is less than or equal to 5.
 - ◆ This condition is true and the bonus variable is assigned the value 50.
 - ◆ Then, the `return` statement is executed and the program terminates without executing the remaining statements of the program. Therefore, no output is displayed from this program.

- ◆ Selection constructs are decision-making blocks that execute a group of statements based on the boolean value of a condition.
- ◆ C# supports if...else, if...else...if, nested if, and switch...case selection constructs.
- ◆ Loop constructs execute a block of statement repeatedly for a particular condition.
- ◆ C# supports while, do-while, for, and foreach loop constructs.
- ◆ The loop control variables are often created for loops such as the for loop.
- ◆ Jump statements transfer the control to any labeled statement or block within a program.
- ◆ C# supports break, continue, goto, and return jump statements.