

CS6004NI

Application Development

samyush

Samyush.maharjan@islingtoncollege.edu.np

C# Operators

C# Operators

Binary operators

Operations such as **addition** and **multiplication** to operands such as variables and literal values.

```
var result = operandOne operator operandTwo;
```

Examples:

1. `var x = 10;`
2. `var y = 3;`
3. `int sum = x + y; // => 13`
4. `int product = x * y; // => 30`
5. `int quotient = x / y; // fractional part is truncated => 3`
6. `var doubleQuotient = (double)x / y; // non-integer division on int => 3.3333333333333335`
7. `int remainder = x % y; // => 1`

C# Operators

Unary operators

Operators that work on **a single operand**, and can apply **before or after the operand**.

```
var result = operand operator;  
var result = operator operand;
```

Examples:

1. `var x = 5;`
2. `int postIncrement = x++; // increment x after assigning`
3. `Console.WriteLine(postIncrement); // => 5`
4. `Console.WriteLine(x); // => 6`
5. `int preIncrement = ++x; // increment x before assigning`
6. `Console.WriteLine(preIncrement); // => 7`
7. `Console.WriteLine(x); // => 7`
8. `int postDecrement = x--; // decrement x after assigning`
9. `int preDecrement = --x; // decrement x before assigning`
10. `Console.WriteLine(x); // => 5`
11. `Type theTypeOfAnInteger = typeof(int);`
12. `int howManyBytesInAnInteger = sizeof(int);`

C# Operators

Assignment operators

Operators used to **assigning value** to a variable.

```
var variable = operand;
```

```
var variable operator= operand;
```

Examples:

```
1. var x = 5;
```

```
2. x += 3; // same as x = x + 3;
```

```
3. x -= 3; // same as x = x - 3;
```

```
4. x *= 3; // same as x = x * 3;
```

```
5. x /= 3; // same as x = x / 3;
```

C# Operators

Logical operators

Operators operate on **Boolean values**, so they return either `true` or `false`. Logical operators always **evaluate both the operands**.

- AND & logical operator

Both operands must be `true` for the result to be `true`.

```
1. var t = true;
2. var f = false;

3. Console.WriteLine(t & t); // => true
4. Console.WriteLine(t & f); // => false
5. Console.WriteLine(f & f); // => false
6. Console.WriteLine(f & t); // => false
```

C# Operators

Logical operators

- OR | logical operator

Ether operand can be `true` for the result to be `true`.

1. `var t = true;`
2. `var f = false;`
3. `Console.WriteLine(t | t); // => true`
4. `Console.WriteLine(t | f); // => true`
5. `Console.WriteLine(f | f); // => false`
6. `Console.WriteLine(f | t); // => true`

C# Operators

Logical operators

- XOR ^ logical operator

Ether operand can be **true** (but not both!) for the result to be **true**.

```
1. var t = true;
2. var f = false;

3. Console.WriteLine(t ^ t); // => false
4. Console.WriteLine(t ^ f); // => true
5. Console.WriteLine(f ^ f); // => false
6. Console.WriteLine(f ^ t); // => true
```


C# Operators

Conditional logical operators

Similar to logical operators, but **two symbols are used instead of one (&& or ||)**. Conditional logical operators **execute the second operand only if necessary**.

```
1. var t = true;
2. var f = false;

3. //Asume doWork function is available which returns a boolean value when executed.
4. Console.WriteLine(t && doWork()); // doWork gets executed
5. Console.WriteLine(f && doWork()); // doWork does not get executed

6. Console.WriteLine(t || doWork()); // doWork does not get executed
7. Console.WriteLine(f || doWork()); // doWork gets executed
```

C# Operators

Bitwise and binary shift operators

Bitwise operators (&, |, ^) affect the **bits in a number** and Binary shift operators (<< and >>) can perform some common **arithmetic calculations**.

Binary system:

| 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

```
1. var a = 10; // 00001010
2. var b = 6;  // 00000110

3. Console.WriteLine(a & b); // 2-bit column only
4. Console.WriteLine(a | b); // 8, 4, and 2-bit columns
5. Console.WriteLine(a ^ b); // 8 and 4-bit columns

6. // 01010000 left-shift a by three bit columns
7. Console.WriteLine(a << 3); // same as multiply a by 8
8. // 00000011 right-shift b by one bit column
9. Console.WriteLine(b >> 1); // same as divide b by 2
```

Note:

- When operating on **integer** values, the symbols (&, |, ^) are **bitwise operators**.
- When operating on **Boolean** values, the symbols (&, |, ^) are **logical operators**.

C# Operators

Ternary conditional operator

Evaluates a Boolean expression and **returns the result of one of the two expressions**.

```
var result = condition ? consequent : alternative;
```

Examples:

1. `var a = 10;`
2. `var b = 6;`
3. `var result = a > b ? "Larger" : "Smaller";`
4. `Console.WriteLine(result); // => Larger`

Read more about other [operators here](#).

C# Operators

What would the outputs of Console.WriteLine statements below?

```
1. var x = 15;
2. var y = 6;
3. var z = 2;
4. var t = true;
5. var f = false;

6. Console.WriteLine(x % y); // => ?
7. Console.WriteLine(z++); // => ?
8. Console.WriteLine(z); // => ?
9. Console.WriteLine(t ^ t); // => ?
10. Console.WriteLine(y << 2); // => ?
```

Binary system:

| 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |



C# Variables

What would the outputs of Console.WriteLine statements below?

```
1. var x = 15;
2. var y = 6;
3. var z = 2;
4. var t = true;
5. var f = false;

6. Console.WriteLine(x % y); // => 3
7. Console.WriteLine(z++); // => 2
8. Console.WriteLine(z); // => 3
9. Console.WriteLine(t ^ t); // => false
10. Console.WriteLine(y << 2); // => 24
```

Binary system:

| 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |



C# Selection statements

C# Selection statements

Branching with the if statement

The **if** statement determines which **branch** to follow by evaluating a **Boolean expression**.

```
1. var x = 15;
2. var y = 2;

3. if (x < y)
4. {
5.     Console.WriteLine("The x number is smaller than y.");
6. }
7. else if (x % 2 == 0) // we can have more than one else if blocks
8. {
9.     Console.WriteLine("The x number is a even number.");
10. }
11. else
12. {
13.     Console.WriteLine("None of the expressions were true.");
14. }
```

C# Selection statements

Branching with the if statement

- If there is only a single statement inside each block, curly braces are optional. **Avoid it for maintainability reason.**

```
1. if (x < y)
2.     Console.WriteLine("The x number is smaller than y.");
3. else
4.     Console.WriteLine("None of the expressions were true.");
```

- Local variable using **is** keyword.

```
1. object x = "6";

2. if (x is int i)
3.     Console.WriteLine($"The x variable is int and local variable i = {i}");
4. else
5.     Console.WriteLine("The x variable is not an int!");
```


C# Selection statements

Branching with the if statement

- Using negation operator.

```
1. var f = false;
2. if (!f)
3.     Console.WriteLine("The f variable is false.");
4. else
5.     Console.WriteLine("The f variable is false.");
```

- Using `is not` keyword.

```
1. object x = "6";

2. if (x is not int && x == "6")
3.     Console.WriteLine($"The x variable is string of number 6.");
4. else
5.     Console.WriteLine("The x variable is not string!");
```

C# Selection statements

Branching with the switch statement

The `switch` compares a **single expression** against a list of **multiple possible case statements**.

```
1. var number = (new Random()).Next(1, 5);  
  
2. switch (number) {  
3.     case 1:  
4.         Console.WriteLine("One");  
5.         break; // jumps to end of switch statement  
6.     case 2:  
7.     case 3: // multiple case section  
8.         Console.WriteLine("Two or three");  
9.         goto case 1;  
10.    default:  
11.        Console.WriteLine("Default");  
12.        break;  
13.} // end of switch statement
```

Every case section must end with:

- The `break`, `goto case`, `return` statement
- Or without statements
- Or the `goto` named label statement

C# Selection statements

Branching with the switch statement

- Using `goto` name label.

```
1. var number = (new Random()).Next(1, 5);
2. string message;

3. switch (number) {
4.     case 1:
5.         message = "One";
6.         goto MyLabel;
7.     case 2:
8.         message = "Two";
9.         goto LastLabel;
10.    default: // always evaluated last despite its current position
11.        message = "Default";
12.        break;
13.    case 3:
14.        message = "Three";
15.        break;
16. }

17. MyLabel:
18. Console.WriteLine("Some statements...");

19. LastLabel:
20. Console.WriteLine($"The message is {message}.");
```

C# Selection statements

Branching with the switch statement

- Using Switch expressions

```
1. var number = (new Random()).Next(-3, 3);

2. var value = number switch
3. {
4.     0 => "Zero",
5.     1 => "One",
6.     2 => "Two",
7.    -2 => "Negative Two",
8.    _ => "Default"
9. };

10. Console.WriteLine($"The value is {value}.");
```

C# Selection statements

What would the outputs of the code below?

```
1. var x = 10;
2. var y = 11;

3. if (--y >= x++)
4.     goto LastLabel;
5. else if (y == 10)
6.     Console.WriteLine("Message 1");
7. else
8.     goto FirstLabel;

9. FirstLabel:
10. Console.WriteLine("Message 2");

11. LastLabel:
12. Console.WriteLine("Message 3");
13. Console.WriteLine($"x={x}, y={y}");
```



C# Selection statements



What would the outputs of the code below?

```
1. var x = 10;
2. var y = 11;

3. if (--y >= x++)
4.     goto LastLabel;
5. else if (y == 10)
6.     Console.WriteLine("Message 1");
7. else
8.     goto FirstLabel;

9. FirstLabel:
10. Console.WriteLine("Message 2");

11. LastLabel:
12. Console.WriteLine("Message 3");
13. Console.WriteLine($"x={x}, y={y}");
```



Message 3
x=11, y=10

C# Iteration statements

C# Iteration statements

Looping with the while statement

The `while` statement **evaluates a Boolean expression** and continues to loop while it is `true`.

- Printing 0-9 number using `while` loop

```
1. var x = 0;

2. while (x < 10)
3. {
4.     Console.WriteLine(x);
5.     x++;
6. }
```


C# Iteration statements

Looping with the while statement

- Printing each elements of a `string[]` variable.

```
1. string[] names = { "John", "Matt", "Steve" };;  
2. var i = 0;  
  
3. while (i < names.Length)  
4. {  
5.     Console.WriteLine(names[i]);  
6.     i++;  
7. }
```

C# Iteration statements

Looping with the do-while statement

The `do-while` statement checks the Boolean expression at the bottom, which means that the block always **executes at least once**.

Example:

```
1. string? password;  
  
2. do  
3. {  
4.     Console.Write("Enter your password: ");  
5.     password = Console.ReadLine();  
6. }  
7. while (password != "SecretPassword");  
  
8. Console.WriteLine("Welcome!");
```

C# Iteration statements

Looping with the for statement

The **for** statement is like **while**, but it combines the **initializer**, **conditional**, and **iterator expressions**.

- **The initializer expression:** executes **once at the start** of the loop.
- **The conditional expression:** executes on **every iteration at the start of the loop** to check whether the looping should continue.
- **The iterator expression:** executes on every loop **at the bottom of the statement**.

- Printing each elements of a `string[]` variable

```
1. string[] names = { "John", "Matt", "Steve" };;  
  
2. for (int i = 0; i < names.Length; i++)  
3. {  
4.     Console.WriteLine(names[i]);  
5. }
```

C# Iteration statements

Looping with the foreach statement

Perform a block of statements on **each item in a sequence**.

- Printing each elements of a `string[]` variable

```
1. string[] names = { "John", "Matt", "Steve" };;  
  
2. foreach (var name in names)  
3. {  
4.     Console.WriteLine(name);  
5. }
```

C# Iteration statements

Looping with the foreach statement

The `foreach` statement will work on any type that follows these rules:

- The type must have a method named **GetEnumerator** that **returns an object**.
- The returned object must have a **property named Current** and a **method named MoveNext**.
- The **MoveNext** method must **change the value of Current** and **return true if there are more items to enumerate through or return false if there are no more items**.

*Note: There are interfaces under **System.Collections** namespace named `IEnumerable` and `IEnumerable<T>` that formally define these rules.*

The `foreach` pseudocode:

```
1. string[] names = { "John", "Matt", "Steve" };;
2. IEnumerator e = names.GetEnumerator();

3. while (e.MoveNext())
4. {
5.     var name = (string)e.Current;
6.     Console.WriteLine(name);
7. }
```

C# Selection statements

What would the outputs of the code below?

```
1. var i = 10;  
2. while (--i > 0)  
3. {  
4.     Console.WriteLine(i);  
5. }
```



C# Selection statements

What would the outputs of the code below?

```
1. var i = 10;  
2. while (--i > 0)  
3. {  
4.     Console.WriteLine(i);  
5. }
```



9
8
7
6
5
4
3
2
1



C# Exceptions handling

C# Exceptions handling

Try-catch statement

- Wrapping **error-prone code** in a **try** block

Example:

```
1. Console.WriteLine("Before parsing");
2. Console.Write("What is your age? ");
3. string? input = Console.ReadLine();
4. try
5. {
6.     int age = int.Parse(input);
7.     Console.WriteLine($"You are {age} years old.");
8. }
9. catch
10.{ // bad practice
11.}
12.Console.WriteLine("After parsing");
```

C# Exceptions handling

Try-catch statement

- Catching all exceptions

```
1. Console.WriteLine("Before parsing");
2. Console.Write("What is your age? ");
3. string? input = Console.ReadLine();
4. try
5. {
6.     int age = int.Parse(input);
7.     Console.WriteLine($"You are {age} years old.");
8. }
9. catch (Exception ex)
10.{
11.    Console.WriteLine($"{ex.GetType()} says {ex.Message}");
12.}
13.Console.WriteLine("After parsing");
```

C# Exceptions handling

Try-catch statement

- Catching specific exceptions

```
1. Console.WriteLine("Before parsing");
2. Console.Write("What is your age? ");
3. string? input = Console.ReadLine();
4. try
5. {
6.     int age = int.Parse(input);
7.     Console.WriteLine($"You are {age} years old.");
8. }
9. catch (FormatException)
10. {
11.     Console.WriteLine("The age you entered is not a valid number format.");
12. }
13. catch (Exception ex)
14. {
15.     Console.WriteLine($"{ex.GetType()} says {ex.Message}");
16. }
```

C# Exceptions handling

Try-catch statement

- Catching with filters

```
1. Console.Write("Enter an amount: ");
2. string? amount = Console.ReadLine();
3. try
4. {
5.     var amountValue = decimal.Parse(amount);
6. }
7. catch (FormatException) when (amount.Contains("$"))
8. {
9.     Console.WriteLine("Amounts cannot use the dollar sign!");
10.}
11.catch (FormatException)
12.{
13.    Console.WriteLine("Amounts must only contain digits!");
14.}
```

C# Exceptions handling

Checked statement

The checked statement tells .NET to **throw an exception when an overflow** happens at **runtime** instead of allowing it to happen silently by default.

Example:

```
1. int x = int.MaxValue - 1; // => 2_147_483_647 - 1

2. Console.WriteLine($"Initial value: {x}"); // => 2,147,483,646
3. x++;
4. Console.WriteLine($"After incrementing: {x}"); // => 2,147,483,647
5. x++;
6. Console.WriteLine($"After incrementing: {x}"); // => -2,147,483,648
7. x++;
8. Console.WriteLine($"After incrementing: {x}"); // => -2,147,483,647
```

C# Exceptions handling

Checked statement

Throwing overflow exceptions with the `checked` statement and exception catching using `try-catch`.

```
1. try
2. {
3.     checked
4.     {
5.         int x = int.MaxValue - 1; // => 2_147_483_647 - 1
6.         Console.WriteLine($"Initial value: {x}"); // => 2,147,483,646
7.         x++;
8.         Console.WriteLine($"After incrementing: {x}"); // => 2,147,483,647
9.         x++;
10.        Console.WriteLine($"After incrementing: {x}"); // => throws System.OverflowException
11.        x++;
12.        Console.WriteLine($"After incrementing: {x}");
13.    }
14. }
15. catch (OverflowException)
16. {
17.     Console.WriteLine("The code overflowed!!");
18. }
```

C# Exceptions handling

Unchecked statement

This **unchecked** statement **switches off overflow checks** performed by the **compiler** within a block of code.

Example:

```
1. unchecked
2. {
3.     int y = int.MaxValue + 1; // without unchecked, it won't compile
4.     Console.WriteLine($"Initial value: {y}"); // => -2,147,483,648
5.     y--;
6.     Console.WriteLine($"After decrementing: {y}"); // => 2,147,483,647
7.     y--;
8.     Console.WriteLine($"After decrementing: {y}"); // => 2,147,483,646
9. }
```

Questions?