Method, Operators, Conditional statement

Before start

- Understand variable, data type
- Visual Studio readiness

Lesson Objectives

- Method
- Operators
- Conditional statements

Section 1

OPERATORS

Statements and expressions

- A C# program is a set of tasks that perform to achieve the overall functionality of the program.
- To perform the tasks, programmers provide instructions. These instructions are called statements.
- A C# statement can contain expressions that evaluates to a value.
- In C#, a statement ends with a semicolon.

Statements

- Statements are used to specify the input, the process, and the output tasks of a program. Statements can consist of:
  - ✓ Data types
  - ✓ Variables
  - ✓ Operators
  - ✓ Constants
  - ✓ Literals
  - ✓ Keywords

Statements

- Statements help you build a logical flow in the program. With the help of statements, you can:
  - ✓ Initialize variables and objects
  - ✓ Take the input
  - ✓ Call a method of a class
  - ✓ Display the output

Types of Statements

Declaration statements	Expression statements	Selection statements	Iteration statements	Jump statements
• introduces a new variable or constant.	• calculate a value must store the value in a variable.	• if • else • switch • case	• do • for • foreach • in • while	• break • continue • default • goto • return • yield

Types of Statements

- Exception handling statements
- Checked and unchecked
- The await statement
- The yield return statement
- The fix statement
- The lock statement
- Labelled statement
- The empty statement

C# Operators

Expressions in C# comprise one or more operators that performs some operations on variables.

An operation is an action performed on single or multiple values stored in variables in order to modify them or to generate a new value with the help of minimum one symbol and a value.

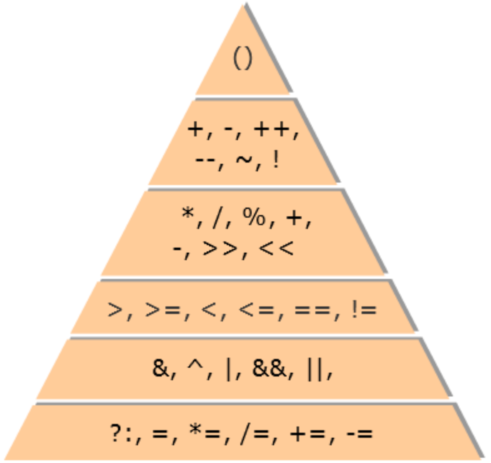
The symbol is called an operator and it determines the type of action to be performed on the value.

The value on which the operation is to be performed is called an operand.

An operand might be a complex expression. For example, (X * Y) + (X – Y) is a complex expression, where the + operator is used to join two operands.

Type of Operators

- Operators are used to simplify expressions.
- In C#, there is a predefined set of operators used to perform various types of operations.
- These are classified into six categories based on the action they perform on values:
    - ✓ Arithmetic Operators
    - ✓ Relational Operators
    - ✓ Logical Operators
    - ✓ Conditional Operators
    - ✓ Increment and Decrement Operators
    - ✓ Assignment Operators



Arithmetic Operators

- Arithmetic operators are binary operators because they work with two operands, with the operator being placed in between the operands.
- These operators allow you to perform computations on numeric or string data.

Arithmetic Operators

Operators	Description	Examples
+ (Addition)	Performs addition. If the two operands are strings, then it functions as a string concatenation operator and adds one string to the end of the other.	`40 + 20`
- (Subtraction)	Performs subtraction. If a greater value is subtracted from a lower value, the resultant output is a negative value.	`100 – 47`
* (Multiplication)	Performs multiplication.	`67 * 46`
/ (Division)	Performs division. The operator divides the first operand by the second operand and gives the quotient as the output.	`12000 / 10`
% (Modulo)	Performs modulo operation. The operator divides the two operands and gives the remainder of the division operation as the output.	`100 % 33`

Arithmetic Operators

- With Division: data type and value of result is based on operand
  - ✓ int/int -> int (floor). Example: 10/3 = 3
  - ✓ decimal/int -> decimal
  - ✓ float/int -> float
  - ✓ …..
- Check and convert data type first
- Always check the second operand to avoid DivideByZero

Operators Precedence - 1

Category	Operator	Associativity
Multiplicative	* / %	Left to right, top to bottom
Additive	+ -	Left to right, top to bottom
Use parentheses () to change the order of evaluation imposed by operator associativity		

Example:
int a = 13 / 5 / 2;   //// a = 1
int b = 13 / (5 / 2); //// b = 6		

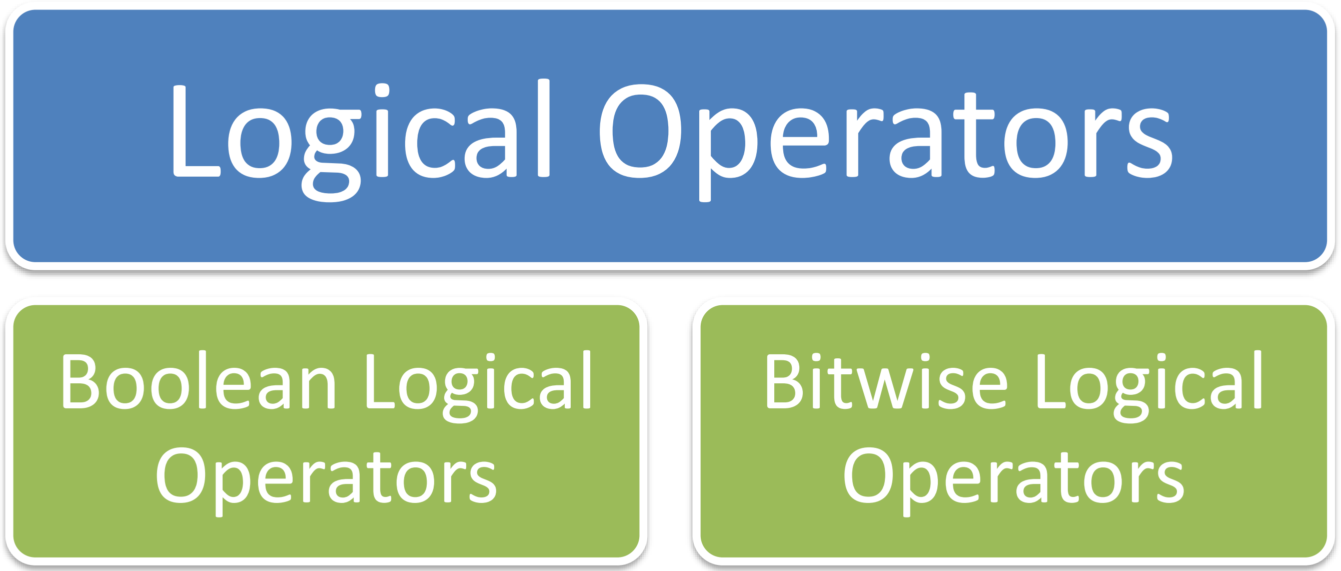
Relational Operators

- Relational operators make a comparison between two operands and return a Boolean value, **true**, or **false**.

Relational Operators	Description	Examples
==	Checks whether the two operands are identical.	`85 == 95`
!=	Checks for inequality between two operands.	`35 != 40`
>	Checks whether the first value is greater than the second value.	`50 > 30`
<	Checks whether the first value is lesser than the second value.	`20 < 30`
>=	Checks whether the first value is greater than or equal to the second value.	`100 >= 30`
<=	Checks whether the first value is lesser than or equal to the second value.	`75 <= 80`

Logical operators

- Logical operators are binary operators that perform logical operations on two operands and return a Boolean value.



Logical negation operator !

- The unary prefix ! operator computes logical negation of its operand. That is, it produces true, if the operand evaluates to false, and false, if the operand evaluates to true:

a	!a
true	false
false	true

Logical AND operator &

- The & operator computes the logical AND of its operands. The result of x & y is true if both x and y evaluate to true. Otherwise, the result is false.

x	y	y & y
true	true	true
true	false	false
false	true	false
false	false	false

Logical OR operator |

- The | operator computes the logical OR of its operands. The result of x | y is true if either x or y evaluates to true. Otherwise, the result is false.

x	y	y & y
true	true	true
true	false	true
false	true	true
false	false	false

Logical exclusive OR operator ^

- The ^ operator computes the logical exclusive OR, also known as the logical XOR, of its operands.

x	y	x ^ y
true	true	false
true	false	true
false	true	true
false	false	false

Conditional logical operator

- There are two types of conditional operators, conditional AND (&&) and conditional OR (||).
- Conditional operators are similar to the boolean logical operators but have the following differences:
  - ✓ The conditional AND operator evaluates the second expression only if the first expression returns true because this operator returns true only if both expressions are true.
  - ✓ The conditional OR operator evaluates the second expression only if the first expression returns false because this operator returns true if either of the expressions is true.

Ternary or Conditional Operator ?:

- If the first expression returns a true value, the second expression is evaluated, whereas if the first expression returns a false value, the third expression is evaluated.
- Syntax: <Expression 1> ? <Expression 2>: <Expression 3>;
  - ✓ Expression 1:Is a bool expression.
  - ✓ Expression 2: Is evaluated if expression 1 returns a **true** value.
  - ✓ Expression 3: Is evaluated if expression 1 returns a **false** value

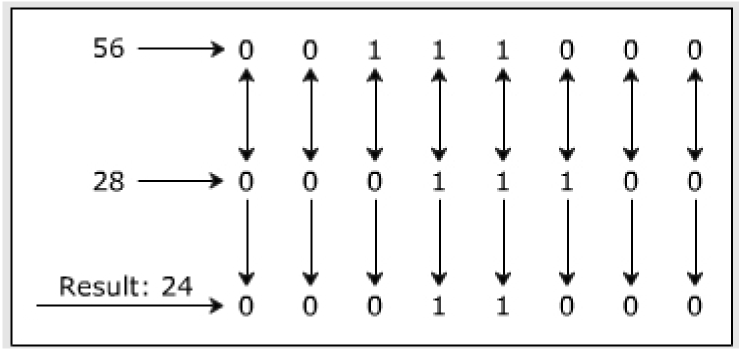
Bitwise Logical Operators

- The bitwise logical operators perform logical operations on the corresponding individual bits of two operands.
- The following table lists the bitwise logical operators along with their descriptions and an example of each type:

Logical Operators	Description	Examples
& (Bitwise AND)	Compares two bits and returns 1 if both bits are 1, else returns 0.	`00111000 & 00011100`
| (Bitwise Inclusive OR)	Compares two bits and returns 1 if either of the bits is 1.	`00010101 | 00011110`
^ (Bitwise Exclusive OR)	Compares two bits and returns 1 if only one of the bits is 1.	`00001011 ^ 00011110`

Bitwise Logical Operators

- The bitwise AND operator compares the corresponding bits of the two operands.
- It returns 1 if both the bits in that position are 1 or else returns 0.
- This comparison is performed on each of the individual bits and the results of these comparisons form an 8-bit binary number.
- This number is automatically converted to integer, which is displayed as the output.
- Same rules with OR and XOR



Increment and Decrement Operators

- Two of the most common calculations performed in programming are increasing and decreasing the value of the variable by 1.
- In C#, the increment operator (++) is used to increase the value by 1 while the decrement operator (--) is used to decrease the value by 1.
- If the operator is placed before the operand, the expression is called pre-increment or pre-decrement.
- If the operator is placed after the operand, the expression is called post-increment or post-decrement.

Increment and Decrement Operators

- The following table depicts the use of increment and decrement operators assuming the value of the variable valueOne is 5:

Expression	Type	Result
valueTwo = ++ValueOne;	Pre-Increment	valueTwo = 6
valueTwo = valueOne++;	Post-Increment	valueTwo = 5
valueTwo = --valueOne;	Pre-Decrement	valueTwo = 4
valueTwo = valueOne--;	Post-Decrement	valueTwo = 5

Assignment Operators

- Assignment operators are used to assign the value of the right side operand to the operand on the left side using the equal to operator (=).
- The assignment operators are divided into two categories in C#. These are as follows:
  - ✓ Simple assignment operators: The simple assignment operator is =, which is used to assign a value or result of an expression to a variable.
  - ✓ Compound assignment operators: The compound assignment operators are formed by combining the simple assignment operator with the arithmetic operators.

Assignment Operators

- The following table shows the use of assignment operators assuming the value of the variable valueOne is 10:

Expression	Description	Result
valueOne += 5;	valueOne = valueOne + 5	valueOne = 15
valueOne -= 5;	valueOne = valueOne - 5	valueOne = 5
valueOne *= 5;	valueOne = valueOne * 5	valueOne = 50
valueOne %= 5;	valueOne = valueOne % 5	valueOne = 0

Operators Precedence - 2

Precedence (where 1 is the highest)	Operator	Description	Associativity
1	()	Parentheses	Child to Parent
2	++ or --	Increment or Decrement	Right to Left
3	*, /, %	Multiplication, Division, Modulus	Left to Right
4	+, -	Addition, Subtraction	Left to Right
5	<, <=, >, >=	Less than, Less than or equal to, Greater than, Greater than or equal to	Left to Right
6	==, !=	Equal to, Not Equal to	Left to Right
7	&&	Conditional AND	Left to Right
8	||	Conditional OR	Left to Right
9	=, +=, -=, *=, /=, %=	Assignment Operators	Right to Left

Section 2

CONDITIONAL STATEMENT

Conditional Statements

- 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.

Conditional Statements

- C# supports the following decision-making constructs:
  - ✓ if…else
  - ✓ if…else…if
  - ✓ switch…case

The if statement

- 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.

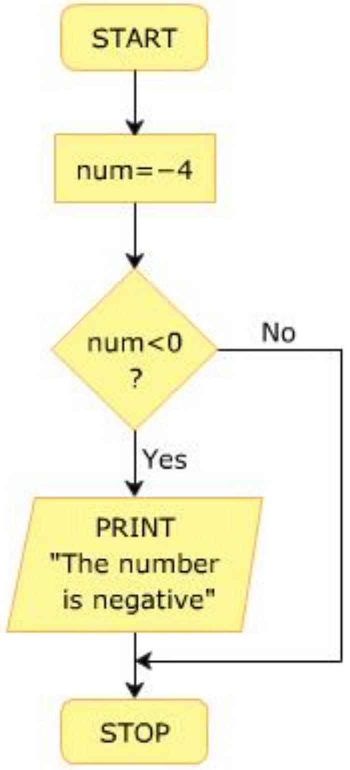
The if statement

- The following is the syntax for the if statement:

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

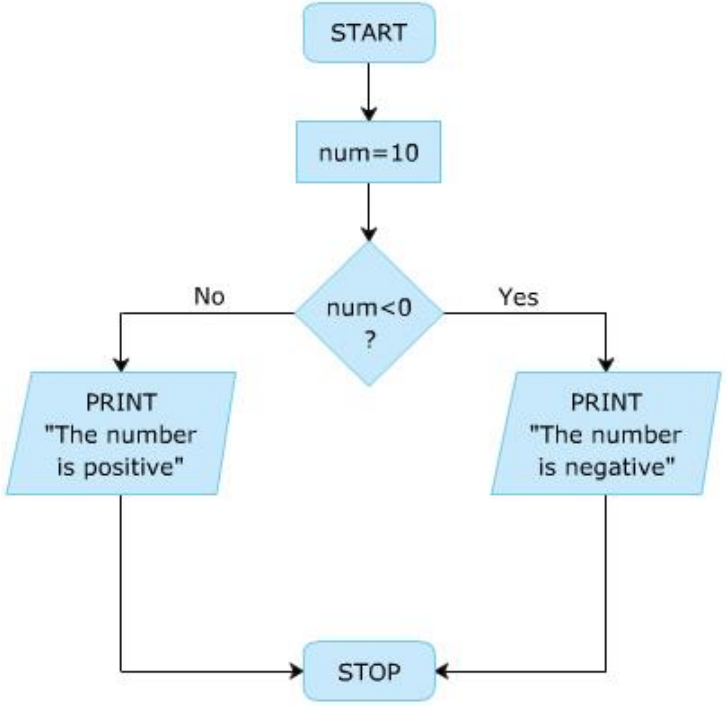
- where,
  - ✓ condition: Is the boolean expression.
  - ✓ statements: Are set of executable instructions executed when the boolean expression returns true.

The if statement



The if...else Construct

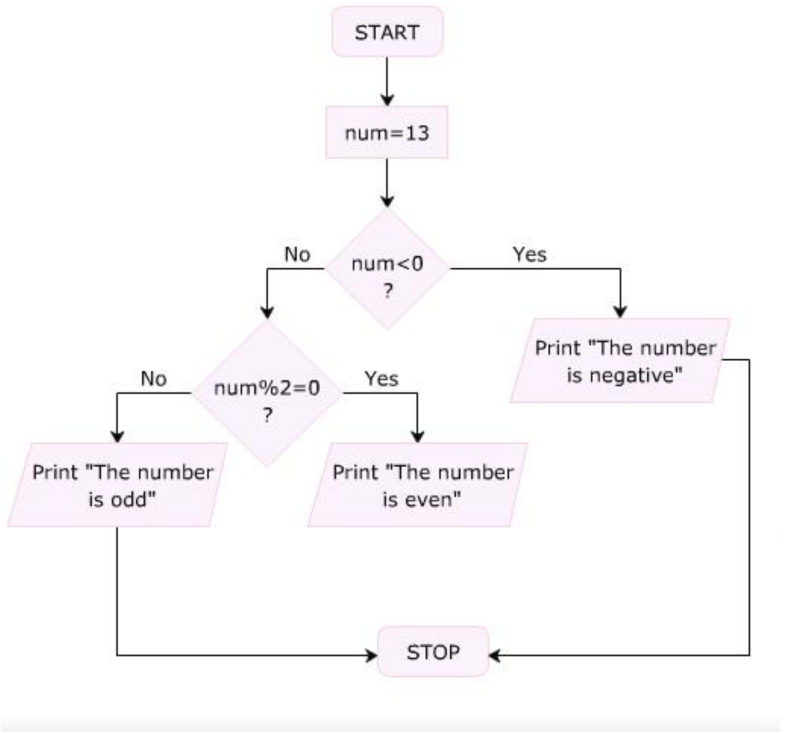
- In some situations, it is required to define an action for a false condition by using an if...else construct.
- The if...else construct starts with the if block followed by an else block and the else block starts with the else keyword followed by a block of statements.
- If the condition specified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


START
num=10
No
num<0 ?
Yes
PRINT "The number is positive"
PRINT "The number is negative"
STOP

The if…else…if Construct

- 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 if…else…if Construct


START
num=13
num<0 ?
No
Yes
Print "The number is negative"
num%2=0 ?
No
Yes
Print "The number is odd"
Print "The number is even"
STOP

Nested if Construct

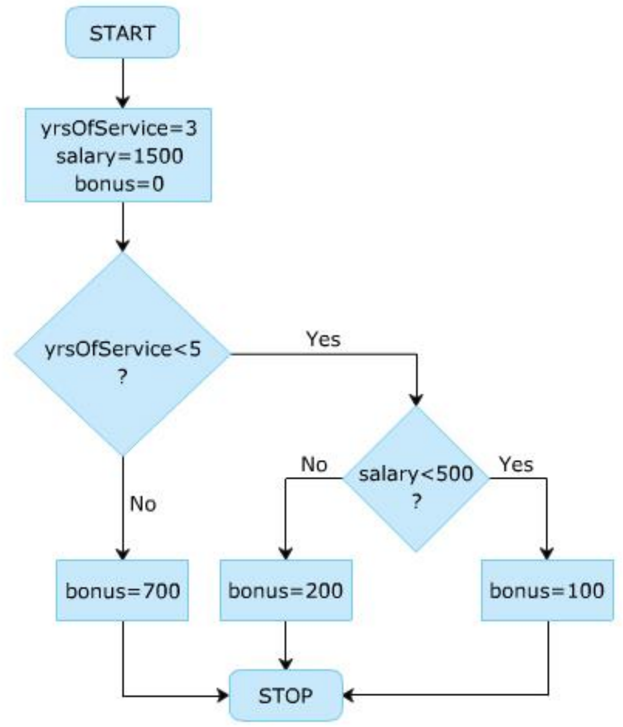
The nested if construct consists of multiple if statements.

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.

Nested if Construct



switch…case Construct

- 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.

switch…case Construct

- 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.

switch…case Construct

- The switch…case construct has the following components:
  - ✓ 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.

No-Fall-Through Rule

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.

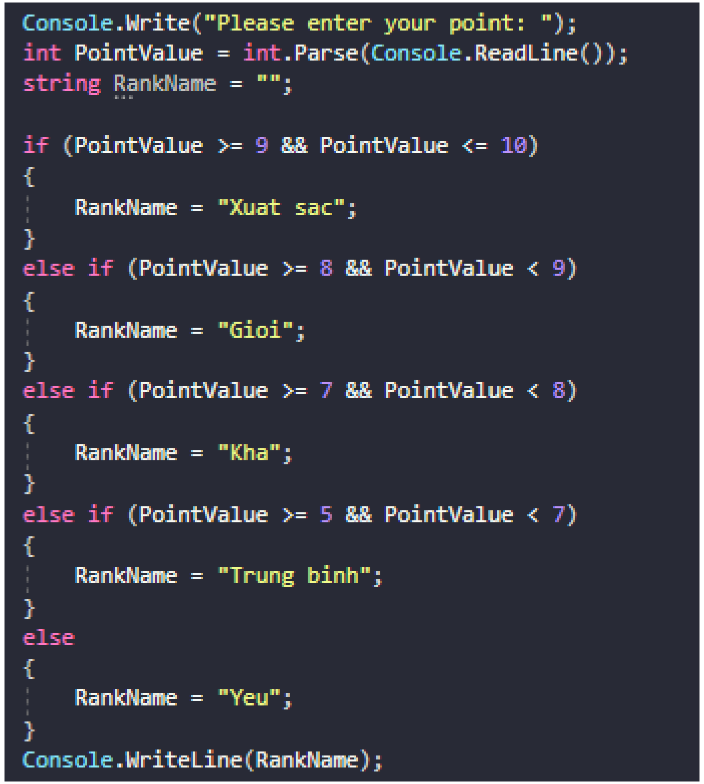
The last case block (or the default block) also needs to have a statement like break or goto to explicitly take the control outside the switch…case construct.

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

No-Fall-Through Rule

- Code will execute statements inside and bellow right case, or until break
- A multiple `case` statement can be made to execute the same code sequence.

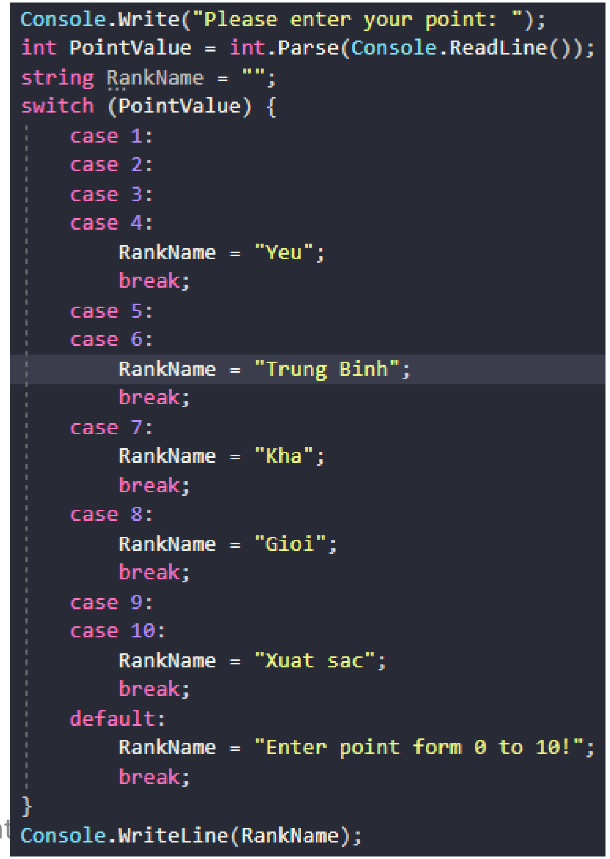
if – else VS switch



```
Console.Write("Please enter your point: ");
int PointValue = int.Parse(Console.ReadLine());
string RankName = "";

if (PointValue >= 9 && PointValue <= 10)
{
    RankName = "Xuat sac";
}
else if (PointValue >= 8 && PointValue < 9)
{
    RankName = "Gioi";
}
else if (PointValue >= 7 && PointValue < 8)
{
    RankName = "Kha";
}
else if (PointValue >= 5 && PointValue < 7)
{
    RankName = "Trung binh";
}
else
{
    RankName = "Yeu";
}
Console.WriteLine(RankName);
```

VS



```
Console.Write("Please enter your point: ");
int PointValue = int.Parse(Console.ReadLine());
string RankName = "";
switch (PointValue) {
    case 1:
    case 2:
    case 3:
    case 4:
        RankName = "Yeu";
        break;
    case 5:
    case 6:
        RankName = "Trung Binh";
        break;
    case 7:
        RankName = "Kha";
        break;
    case 8:
        RankName = "Gioi";
        break;
    case 9:
    case 10:
        RankName = "Xuat sac";
        break;
    default:
        RankName = "Enter point form 0 to 10!";
        break;
}
Console.WriteLine(RankName);
```

if – else VS switch

1. Check the Testing Expression

An **if-else** statement can test expressions based on ranges of values or conditions,

whereas

A **switch** statement tests expressions based only on a single integer, enumerated value, or String object.

if – else VS switch

2. Switch better for Multi way branching

When compiler compiles a **switch** statement, it will inspect each of the case constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression. Therefore, if we need to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of if-else.
The compiler can do this because **it knows that the case constants** are all the same type and simply must be compared for equality with the switch expression, while in case of if expressions, the compiler has no such knowledge.

if – else VS switch

3. if-else better for boolean values

If-else conditional branches are great for variable conditions that result into a boolean,

whereas

switch statements are great for fixed data values.

if – else VS switch

4. Speed

A **switch** statement might prove to be faster than ifs provided number of cases are good. If there are only few cases, it might not effect the speed in any case. Prefer switch if the number of cases are more than 5 otherwise, you may use if-else too.
If a switch contains more than **five items**, it's implemented using a lookup table or a hash list. This means that all items get the same access time, compared to a list of ifs where the last item takes much more time to reach as it has to evaluate every previous condition first.

if – else VS switch

5. Clarity in readability

A switch looks much cleaner when you have to combine cases. Ifs are quite vulnerable to errors too. Missing an else statement can land you up in havoc. Adding/removing labels is also easier with a switch and makes your code significantly easier to change and maintain.

if – else VS switch

6. Conversion

All switch statements can be convert to if-else

BUT

Not all if-else statements can be convert to switch

Give your decision

- Print day of week
- Print month of year
- Branching for check positive or negative number
- Student evaluation by final GPA

Section 3

METHOD

Code block

- Block of **statements** in **sequential** to implement **specify logical**
- C# use bracket to specify a code block {}
- Nested code block
- Sample:
  - ✓ Read name of user then say Hello
  - ✓ Get max value of 2 numbers
  - ✓ Find the greatest common divisor

Method

- A method is a code block that contains a series of statements.
- A program causes the statements to be executed by calling the method and specifying any required method arguments.
- In C#, every executed instruction is performed in the context of a method.
- Why use methods? To reuse code
  - ✓ define the code once
  - ✓ use it many times

Method signatures

- A signature of method to let compiler (and developer) understand main idea of the method
- A signature contains:
  - ✓ Access level: optional
  - ✓ Method name
  - ✓ Method parameters
  - ✓ Return type

Method name

- Following basic rules of name in C#
- Is verb or verb-phrase, describe main idea of the body
- **Use Pascal Casing** - First character of all words are Upper Case and other characters are lower case

Method parameters

- The method definition specifies the names and types of any parameters that are required.
- When calling code calls the method, it provides concrete values called arguments for each parameter.
- One method has zero, one or many parameters
- All parameters are placed in parentheses()

Method Overloading

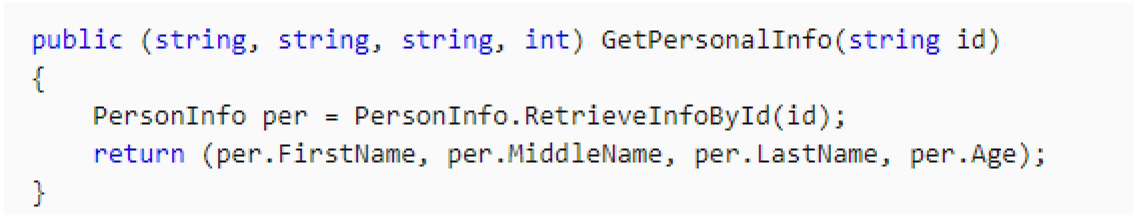
- With method overloading, multiple methods can have the same name with different parameters
- Overloading types:
  - ✓ Difference number of parameters
  - ✓ Difference data type of parameters
  - ✓ Difference order of parameters with difference data type
  - ✓ CANNOT: difference return values

Return values

- Methods can return a value to the caller.
- A statement with the return keyword followed by a variable, constant, or expression that matches the return type will return that value to the method caller.
- If method does not return any value, use void in method signature.
- Methods with a non-void return type are required to use the return keyword to return a value.
- The return keyword also stops the execution of the method.

Return values

- To return more than a single value, from C# 7.0, you can do this easily by using tuple types and tuple literals.
- The tuple type defines the data types of the tuple's elements.
- Tuple literals provide the actual values of the returned tuple.



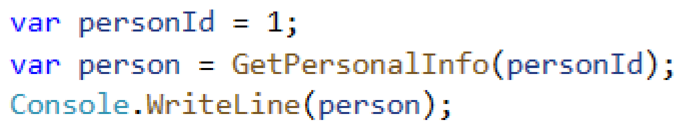
```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

Method body rules

- Method implement and only implement specify logical
- Method should use and only use defined parameters
  - ✓ Parameter is not used => remove from signature
  - ✓ Use other value => consider move it into parameter
- Method body should not exceed ONE screen of code
- DONOT use goto in method body
- Avoid to return anywhere in method

Method invocation

- Step 1: prepare arguments
- Step 2: Invoke method
- Step 3: (optional) receive return value



```
var personId = 1;
var person = GetPersonalInfo(personId);
Console.WriteLine(person);
```

Method parameters vs. arguments

- Parameter: specifies the names and data type in method signature
- Argument: concrete value that used to pass to method.
- The arguments must be compatible with the parameter type but the argument name (if any) used in the calling code doesn't have to be the same as the parameter named defined in the method.

Passing types

Passing by value

- Its copy is passed instead of the instance itself.
- changes to the argument have no effect on the original instance in the calling method.
- To pass a value-type instance by reference, use the **ref** keyword

Passing by reference

- a reference to the object is passed
- the change is reflected in the argument in the calling method

Variable scope

- Class Level Scope
  - ✓ Declaring the variables in a class but outside any method can be directly accessed anywhere in the class.
  - ✓ These variables are also termed as the fields or class members.
  - ✓ Class level scoped variable can be accessed by the non-static methods of the class in which it is declared.
  - ✓ Access modifier of class level variables doesn’t affect their scope within a class.
  - ✓ Member variables can also be accessed outside the class by using the access modifiers.

Variable scope

- Method Level Scope
  - ✓ Variables that are declared inside a method have method level scope. These are not accessible outside the method.
  - ✓ However, these variables can be accessed by the nested code blocks inside a method.
  - ✓ These variables are termed as the local variables.
  - ✓ There will be a compile-time error if these variables are declared twice with the same name in the same scope.
  - ✓ These variables don’t exist after method’s execution is over.

Variable scope

- **Block Level Scope**
  - ✓ These variables are generally declared inside the for, while statement etc.
  - ✓ These variables are also termed as the loop variables or statements variable as they have limited their scope up to the body of the statement in which it declared.
  - ✓ Generally, a loop inside a method has three level of nested code blocks(i.e. class level, method level, loop level).
  - ✓ The variable which is declared outside the loop is also accessible within the nested loops. It means a class level variable will be accessible to the methods and all loops. Method level variable will be accessible to loop and method inside that method.
  - ✓ A variable which is declared inside a loop body will not be visible to the outside of loop body.

Practice time

Lesson Summary

Thank you