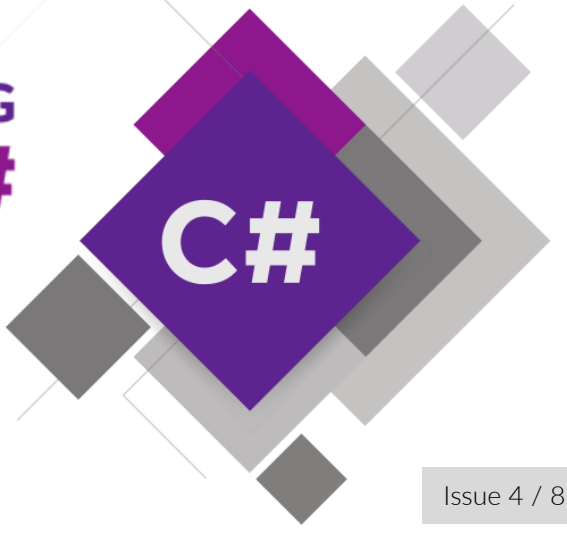
PROGRAMMING BASICS WITH C#



Issue 4 / 8

MORE COMPLEX CONDITIONS

Learn about using more complex if-else conditional statements, using switch-case, working with nested conditional statements and using more complex logical conditions with “AND”, “OR”, “NOT” and Parenthesis.

Get an idea how to build a simple Desktop GUI app “Point and Rectangle” in C# using Visual Studio.

Dr. Svetlin Nakov

Software University – https://softuni.org

More Complex Conditions

In this mini book, we are going to examine the **nested conditional statements** in the **C#** language, by which our program can contain **conditions** that contain other **nested conditional statements**. We call them **"nested"**, because **we put an `if` condition** into **another `if` condition**. We are going to examine the **more complex logical conditions** through proper examples.

Video: Overview

Watch this video to see what you will learn: https://youtu.be/qvbVrKXxsu0.

Introduction to Complex Conditions by Examples

Conditional statements can be nested, i.e. we can put **`if-else`** inside another **`if-else`** statement. Conditions in the **`if`** constructions can be complex, e.g. use **logical "AND"** or **logical "OR"**. Example:

```
var a = decimal.Parse(Console.ReadLine());
var b = decimal.Parse(Console.ReadLine());
if (a > 0 && b > 0 && a <= 100 && b <= 100)
{
    if (a * b >= 5000)
        Console.WriteLine($"Large size: {a*b}");
    else if (a * b > 1000 && a * b < 5000)
        Console.WriteLine($"Middle size: {a * b}");
    else
        Console.WriteLine($"Small size: {a * b}");
}
else
    Console.WriteLine($"Invalid size (a={a}, b={b})");
```

Run the above code example: https://repl.it/@nakov/nested-if-else-conditions-csharp.

The above code performs a **series of checks** using nested **`if-else`** conditional statements and logical operators like **&&** (logical **AND**) to check the input data for the following 4 cases:

- Size out of range (one of the sides is negative or bigger than 100).
- Large size (area >= 5000).
- Middle size (1000 < area < 5000)
- Small size (area <= 1000)

Let's explain in greater detail how to use **complex and nested `if-else` conditions** in C#.

Nested If-Else Conditions

Pretty often the program logic requires the use of **`if`** or **`if-else`** statements, which are contained one inside another. They are called **nested `if`** or **`if-else`** statements. As implied by the title **"nested"**, these are **`if`** or **`if-else`** statements that are placed inside other **`if`** or **`else`** statements.

```
if (condition1)
{
    if (condition2)
    {
        // body;
    }
    else
    {
        // body;
    }
}
```

Video: Nested Conditional Statements

Watch a video lesson about the nested if-conditions: https://youtu.be/4ugMAlkQAMo.

Deep Nesting

Nesting of **more than three conditional statements** inside each other is not considered a good practice and **has to be avoided**, mostly through optimization of the structure/the algorithm of the code and/or by using another type of conditional statement.

Nested If-Else Conditions – Examples

Let's take a few examples in order to gain experience about how to use **nested `if-else` conditions** in practice.

Example: Personal Titles

Depending on **age** (decimal number and **gender** (**m** / **f**), print a personal title:

- "**Mr.**" – a man (gender "**m**") – 16 or more years old.
- "**Master**" – a boy (gender "**m**") under 16 years.
- "**Ms.**" – a woman (gender "**f**") – 16 or more years old.
- "**Miss**" – a girl (gender "**f**") under 16 years.

Sample Input and Output

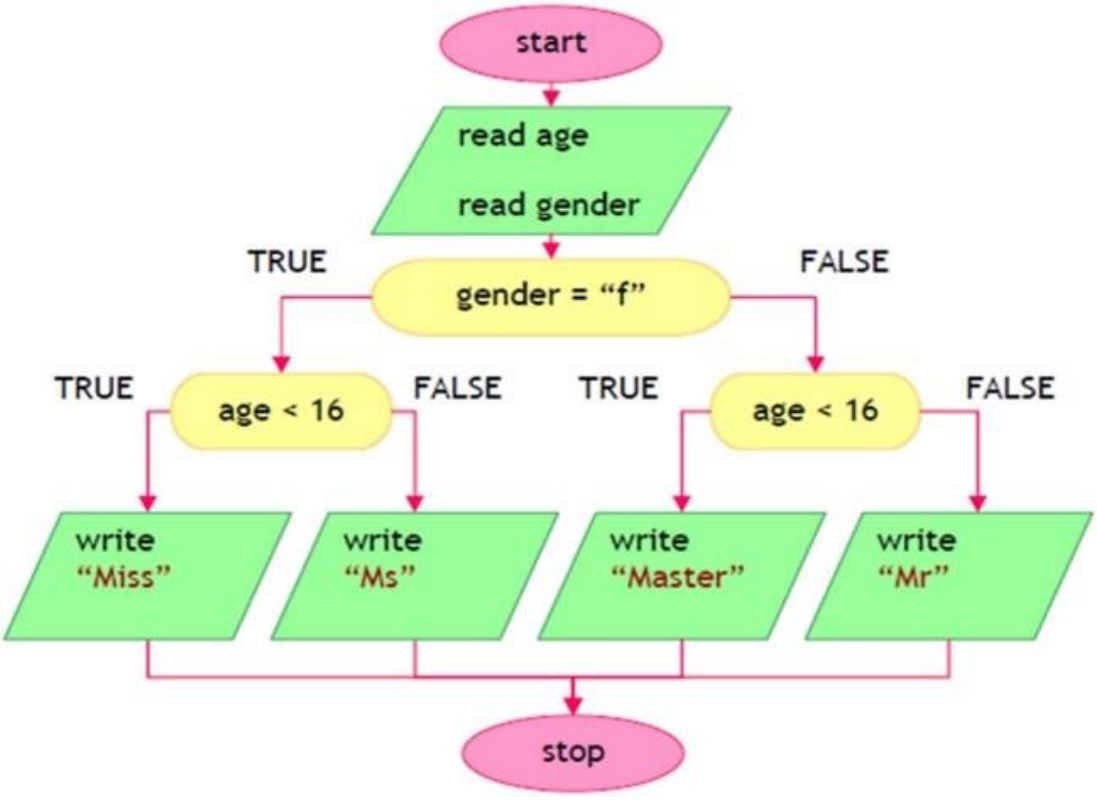
Input	Output	Input	Output
12 f	Miss	17 m	Mr.
Input	**Output**	**Input**	**Output**
25 f	Ms.	13.5 m	Master

Video: Personal Titles

Watch this video to learn how to solve this problem: https://youtu.be/7WiBbMOAc7Q.

Solution

We should notice that the output of the program depends on a few things. First, we have to check what is the entered gender and then check the age. Respectively, we are going to use a few `if-else` blocks. These blocks will be nested, meaning from the result of the first, we are going to define which one of the others to execute.



After reading the input data from the console, the following program logic should be executed:

```
var age = double.Parse(Console.ReadLine());
var gender = Console.ReadLine();
if (age < 16)
{
    if (gender == "m") Console.WriteLine("Master");
    else if (gender == "f") Console.WriteLine("Miss");
}
else
{
    if (gender == "m") Console.WriteLine("Mr.");
    else if (gender == "f") Console.WriteLine("Ms.");
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#0.

Example: Small Shop

A Bulgarian entrepreneur opens **small shops** in **a few cities** with different **prices** for the following **products**:

product / city	Sofia	Plovdiv	Varna
coffee	0.50	0.40	0.45
water	0.80	0.70	0.70
beer	1.20	1.15	1.10
sweets	1.45	1.30	1.35
peanuts	1.60	1.50	1.55

Calculate the price by the given **city** (string), **product** (string) and **quantity** (decimal number).

Sample Input and Output

Input	Output	Input	Output
coffee Varna 2	0.9	peanuts Plovdiv 1	1.5

Video: Small Shop

Watch this video to learn how to solve this problem: https://youtu.be/kU_ru7GK-Mg.

Solution

We **convert** all of the letters into **lower register** using the function **`.ToLower()`**, in order to compare products and cities **no matter** what the letters are – small or capital ones.

```
var product = Console.ReadLine().ToLower();
var town = Console.ReadLine().ToLower();
var quantity = double.Parse(Console.ReadLine());
if (town == "sofia")
{
    if (product == "coffee")
        Console.WriteLine(0.50 * quantity);
    // TODO: finish this …
}
if (town == "varna") // TODO: finish this …
    if (town == "plovdiv") // TODO: finish this …
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#1.

More Complex Conditions

Let's take a look at how we can create more **complex logical conditions** in programming. We can use the logical "AND" (`&&`), logical "OR" (`||`), logical **negation** (`!`) and **brackets** (`()`).

Logical "AND", "OR" and "NOT"

This is a short example that demonstrates the power of logical "AND", logical "OR" and logical "NOT":

```
var animal = "horse";
int speed = 45;

if ((animal == "horse" || animal == "donkey") && (speed > 40))
    Console.WriteLine("Run fast")
else if ((animal == "shark" || animal == "dolphin") && (speed > 45))
    Console.WriteLine("Swim fast")
else if (!(speed > 30 || animal == "turtle"))
    Console.WriteLine("Slow move")
```

We shall explain the logical AND (`||`), the logical OR (`||`), and the logical NOT (`!`) in the next few sections, along with examples and exercises.

The Parenthesis `()` Operator

Like the rest of the operators in programming, the operators `&&` and `||` have a priority, as in the case `&&` is with higher priority than `||`. The operator `()` serves for **changing the priority of operators** and is being calculated first, just like in mathematics. Using parentheses also gives the code better readability and is considered a good practice.

Example of checking whether a variable belongs to certain ranges:

```
if (x < 0) || ((x >= 5) && (x <= 10)) || (x > 20)
{
    ...
}
```

Logical "AND"

As we saw, in some tasks we have to make **many checks at once**. But what happens when in order to execute some code **more** conditions have to be executed and we **don't want to** make a **negation** (`else`) for each one of them? The option with nested `if` **blocks** is valid, but the code would look very **unordered** and for sure – **hard** to read and maintain.

The logical "AND" (operator `&&`) means a few conditions have to be **fulfilled simultaneously**. The following table of truthfulness is applicable:

a	b	a && b
true	true	true
true	false	false

a	b	a && b
false false	true false	false false

Video: Logical "AND"

Watch this video to learn how to use logical "AND" in programming: https://youtu.be/V86_z8GWarM.

How the && Operator Works?

The **&&** operator accepts a couple of Boolean (conditional) statements, which have a **true** or **false** value, and returns one bool statement as a result. Using it instead of a couple of nested **if** blocks, makes the code more readable, ordered and easy to maintain. But how does it work, when we put a few conditions one after another? As we saw above, the logical "AND" returns **true**, only when it accepts as arguments statements with value **true**. Respectively, when we have a sequence of arguments, the logical "AND" checks either until one of the arguments is over, or until it meets an argument with value **false**.

Example:

```
bool a = true;
bool b = true;
bool c = false;
bool d = true;
bool result = a && b && c && d;
// false (as d is not being checked)
```

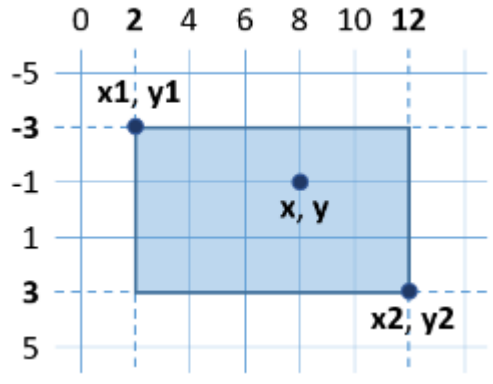
The program will run in the following way: It starts the check form **a**, reads it and accepts that it has a **true** value, after which it checks **b**. After it has accepted that **a** and **b** return **true**, it checks the next argument. It gets to **c** and sees that the variable has a **false** value. After the program accepts that the argument **c** has a **false** value, it calculates the expression before **c**, independent of what the value of **d** is. That is why the evaluation of **d** is being skipped and the whole expression is calculated as **false**.

```
if (x >= x1 && x <= x2 && y >= y1 && y <= y2)
```

Example: Point in a Rectangle

Checks whether point {x, y} is placed inside the rectangle {x1, y1} – {x2, y2}. The input data is read from the console and consists of 6 lines: the decimal numbers x1, y1, x2, y2, x and y (as it is guaranteed that x1 < x2 and y1 < y2).

Sample Input and Output

Input	Output	Visualization
2 -3 12 3 8 -1	Inside	

Solution

A point is internal for a given polygon, if the following four conditions are applied **at the same time**:

- The point is placed to the right from the left side of the rectangle.
- The point is placed to the left from the right side of the rectangle.
- The point is placed downwards from the upper side of the rectangle.
- The point is placed upwards from the down side of the rectangle.

```
var x1 = double.Parse(Console.ReadLine());
var y1 = double.Parse(Console.ReadLine());
var x2 = double.Parse(Console.ReadLine());
var y2 = double.Parse(Console.ReadLine());

var x = double.Parse(Console.ReadLine());
var y = double.Parse(Console.ReadLine());

if (x >= x1 && x <= x2 && y >= y1 && y <= y2)
{
    Console.WriteLine("Inside");
}
else
{
    Console.WriteLine("Outside");
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#2.

Logical "OR"

The logical "OR" (operator `||`) means that at least one among a few conditions is fulfilled. Similar to the operator `&&`, the logical "OR" accepts a few arguments of bool (conditional) type and returns `true` or `false`. We can easily guess that we obtain a value `true` every time when at least one of the arguments has a `true` value. Typical example of the logic of this operator is the following:

At school the teacher says: "John or Peter should clean the board". To fulfill this condition (to clean the board), it is possible either just for John to clean it, or just for Peter to clean it, or both of them to do it.

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Video: Logical "OR"

Watch this video to learn how to use logical "OR" in programming: https://youtu.be/e6i-2E66RNU.

How the || Operator Works?

We have already learned what the logical "OR" represents. But how is it actually being achieved? Just like with the logical "AND", the program checks from left to right the arguments that are given. In order to obtain `true` from the expression, it is necessary for just one argument to have a `true` value. Respectively, the checking continues until an argument with such value is met or until the arguments are over.

Here is one example of the `||` operator in action:

```
bool a = false;
bool b = true;
bool c = false;
bool d = true;
bool result = a || b || c || d;
// true (as c and d are not being checked)
```

The programs checks `a`, accepts that it has a value `false` and continues. Reaching `b`, it understands that it has a `true` value and the whole expression is calculated as `true`, without having to check `c` or `d`, because their values wouldn't change the result of the expression.

Example: Fruit or Vegetable

Let's check whether a given product is a fruit or a vegetable. The "fruits" are: banana, apple, kiwi, cherry, lemon and grapes. The "vegetables" are: tomato, cucumber, pepper and carrot. Everything else is "unknown".

Sample Input and Output

Input	Output
banana tomato java	fruit vegetable unknown

Solution

We have to use a few conditional statements with logical "OR" (**||**):

```
var s = Console.ReadLine();
if (s == "banana" || s == "apple" || s == "kiwi" ||
    s == "cherry" || s == "lemon" || s == "grapes")
    Console.WriteLine("fruit");
else if (s == "tomato" || s == "cucumber" || s == "pepper" || s == "carrot")
    Console.WriteLine("vegetable");
else
    Console.WriteLine("unknown");
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#3.

Logical Negation (NOT)

Logical negation (operator **!**) means a given condition is **not fulfilled**.

a	!a
true	false

The operator **!** accepts as an **argument** a bool variable and **returns** its value.

Video: Logical "NOT"

Watch this video to learn how to use the **logical "NOT"** operator in programming: https://youtu.be/4U7w2ZSAAW4.

Example: Invalid Number

A given **number is valid** if it is in the range [**100** ... **200**] or it is **0**. Do a validation for an **invalid** number.

Sample Input and Output

Input	Output
75	invalid

Input	Output
150	(no output)
220	invalid

Solution

```
var inRange = (num >= 100 && num <= 200) || num == 0;
if (!inRange)
    Console.WriteLine("invalid");
```

Testing in the Judge System

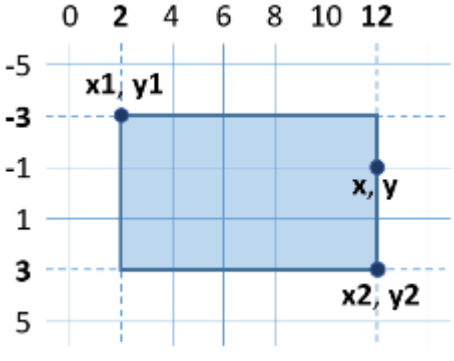
Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#4.

More Complex Conditions – Examples

Sometimes the conditions may be **very complex**, so they can require a long bool expression or a sequence of conditions. Let's take a look at a few examples.

Example: Point on a Rectangle Side

Write a program that checks whether a **point {x, y}** is placed **onto any of the sides of a rectangle {x1, y1} – {x2, y2}**. The input data is read from the console and consists of 6 lines: the decimal numbers **x1**, **y1**, **x2**, **y2**, **x** and **y** (as it is guaranteed that **x1 < x2** and **y1 < y2**). Print "**Border**" (if the point lies on any of the sides) or "**Inside / Outside**" (in the opposite case).



Sample Input and Output

Input	Output	Input	Output
2 -3 12 3 12 -1	Border	2 -3 12 3 8 -1	Inside / Outside

Solution

The point lies on any of the sides of the rectangle if:

- x coincides with x1 or x2 and at the same time y is between y1 and y2 or
- y coincides with y1 or y2 and at the same time x is between x1 and x2.

```
if (((x == x1 || x == x2) && (y >= y1) && (y <= y2)) ||
    ((y == y1 || y == y2) && (x >= x1) && (x <= x2)))
{
    Console.WriteLine("Border");
}
```

The previous evaluation might be simplified in the following way:

```
var onLeftSide = (x == x1) && (y >= y1) && (y <= y2);
var onRightSide = (x == x2) && (y >= y1) && (y <= y2);
var onUpSide = (y == y1) && (x >= x1) && (x <= x2);
var onDownSide = (y == y2) && (x >= x1) && (x <= x2);
if (onLeftSide || onRightSide || onUpSide || onDownSide)
{
    Console.WriteLine("Border");
}
```

The second way with the additional Boolean (bool) variables is longer, but much more understandable than the first one, isn't it? We recommend when you write Boolean conditions to make them **easy to read and understand**, instead of making them short. Use additional variables with meaningful names, if needed. The names of the bool variables have to hint what the value that is kept inside them represents.

What remains is to finish writing the code to print "**Inside / Outside**", if the point is not onto any of the sides of the rectangle.

Testing in the Judge System

After you finish writing the solution, you can test it here, in the judge system: https://judge.softuni.org/Contests/Practice/Index/508#5.

Example: Fruit Shop

A fruit shop during **week days** sells in the following **prices**:

Fruit	Price
banana	2.50
apple	1.20
orange	0.85
grapefruit	1.45
kiwi	2.70

Fruit	Price
pineapple	5.50
grapes	3.85

During the weekend days the prices are **higher**:

Fruit	Price
banana	2.70
apple	1.25
orange	0.90
grapefruit	1.60
kiwi	3.00
pineapple	5.60
grapes	4.20

Write a program that **reads** from the console a **fruit** (banana / apple / …), **a day of the week** (Monday / Tuesday / …) and **a quantity (a decimal number)** and **calculates the price** according to the prices from the tables above. The result has to be printed **rounded up to 2 digits after the decimal point**. Print **"error"** if it is an **invalid day** of the week or an **invalid name** of a fruit.

Sample Input and Output

Input	Output	Input	Output
orange Sunday 3	2.70	kiwi Monday 2.5	6.75

Input	Output	Input	Output
grapes Saturday 0.5	2.10	tomato Monday 0.5	error

Video: Fruit Store

Watch the video to learn how to solve the "Fruit Store" problem: https://youtu.be/6vZZzil9xBU.

Solution

```
if (day == "saturday" || day == "sunday")
{
    if (fruit == "banana") price = 2.70;
    else if (fruit == "apple") price = 1.25;
    // TODO: more fruits come here …
}
```

```
else if (day == "monday" || day == "tuesday" || day ==
   "wednesday" || day == "thursday" || day == "friday")
{
    if (fruit == "banana") price = 2.50;
    // TODO: more fruits come here …
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#6.

Example: Trade Fees

A company is giving the following **commissions** to its traders according to the **city**, in which they are working and the **volume of sales s**:

City	0 <= s <= 500	500 < s <= 1000	1000 < s <= 10000	s > 10000
Sofia	5%	7%	8%	12%
Varna	4.5%	7.5%	10%	13%
Plovdiv	5.5%	8%	12%	14.5%

Write a **program** that reads the name of a **city** (string) and the volume of **sales** (decimal number) and calculates the rate of the commission fee. The result has to be shown rounded **up to 2 digits after the decimal point**. When there is an **invalid city or volume of sales** (a negative number), print "**error**".

Sample Input and Output

Input	Output	Input	Output	Input	Output
Sofia 1500	120.00	Plovdiv 499.99	27.50	Paris -50	error

Video: Trade Fees

Watch the video about the "Trade Fees" problem and its solution: https://youtu.be/QqKBLJ4JzJ0.

Solution

When reading the input, we could convert the city into small letters (with the function **`.ToLower()`**). Initially we set the commission fee to **`-1`**. It will be changed if the city and the price range are found in the table of commissions. To calculate the commission according to the city and volume of sales, we need a few nested **`if`** **statements**, as in the sample code below:

```
var comission = -1.0;
if (town == "sofia")
{
    if (0 <= sales && sales <= 500) comission = 0.05;
    else if (500 < sales && sales <= 1000) comission = 0.07;
    // TODO: check the other price ranges …
}
else if (town == "varna") // TODO: check the price ranges …
else if (town == "plovdiv") // TODO: check the price ranges …
if (comission >= 0)
    Console.WriteLine("{0:f2}", sales * comission);
else Console.WriteLine("error");
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#7.

It is a good practice to use blocks that are enclosed with curly braces **{ }** after **if** and **else**. Also, it is recommended during writing to move aside the code after **if** and **else** with a single tabulation inward, in order to make the code more easily readable.

Switch-Case Conditional Statement

The **switch-case** condition works as a sequence of **if-else** blocks. Whenever the work of our program depends on the value of one variable, instead of making consecutive conditions with **if-else** blocks, we can use the conditional **switch** statement. It is being used for choosing between a list of possibilities. The statement compares a given value with defined constants and depending on the result, it takes an action.

We put the variable that we want to compare, inside the brackets after the operator **switch** and it is called a "selector". Here the type must be comparable (numbers, strings). Consecutively, the program starts comparing each value that is found after the **case** labels. Upon a match, the execution of the code from the respective place begins and continues until it reaches the operator **break**. In some programming languages (like C and C++) **break** might be skipped, in order to execute a code from other **case** construction, until it reaches another operator. In C# though, the presence of **break** is mandatory for every **case** that contains a program logic. When no matches are found, the **default** construction is being executed, if such exists.

```
switch (selector)
{
    case value1:
        construction;
        break;
    case value2:
        construction;
        break;
```

```
    case value3:
        construction;
        break;
    …
    default:
        construction;
        break;
}
```

Video: Switch-Case

Watch the video to learn how to use the switch-case conditional statement: https://youtu.be/mGJOc4xx5Ho.

Example: Day of the Week

Let's write a program that prints **the day of the week** (in English) depending on the **given number** (1 … 7) or "**Error!**" if an invalid input is given.

Sample Input and Output

Input	Output
1 7 -1	Monday Sunday Error!

Solution

```
int day = int.Parse(Console.ReadLine());
switch (day)
{
    case 1: Console.WriteLine("Monday"); break;
    case 2: Console.WriteLine("Tuesday"); break;
    …
    case 7: Console.WriteLine("Sunday"); break;
    default: Console.WriteLine("Error!"); break;
}
```

It is a good practice to put at the **first** place those **`case` statements** that process **the most common situations** and leave the **`case` constructions** processing **the more rear situations** at **the end, before the `default` construction**. Another **good practice** is to **arrange the `case` labels** in **ascending order**, regardless of whether they are integral or symbolic.

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#8.

Multiple Switch-Cases

In C# we have the possibility to use multiple `case` labels, when they have to execute the same code. This way, when our program finds a match, it will execute the next code, because after the respective `case` label there is no code for execution and a `break` operator.

```
switch (selector)
{
    case value1:
    case value2:
    case value3:
        construction;
        break;
    case value4:
    case value5:
        construction;
        break;
    …
    default:
        construction;
        break;
}
```

Example: Animal Type

Write a program that prints the type of the animal depending on its name:

- dog -> mammal
- crocodile, tortoise, snake -> reptile
- others -> unknown

Sample Input and Output

Input	Output	Input	Output	Input	Output
tortoise	reptile	dog	mammal	elephant	unknown

Solution

We can solve the task with `switch-case` conditions with multiple labels in the following way:

```
switch (animal)
{
    case "dog": Console.WriteLine("mammal"); break;
    case "crocodile":
    case "tortoise":
    case "snake": Console.WriteLine("reptile"); break;
    default: Console.WriteLine("unknown"); break;
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#9.

Exercises: More Complex Conditions

Now let's **exercise** our new skills with **complex conditions**. Let's solve a few practical tasks.

Video: Summary

Watch this video to review what we learned: https://youtu.be/QOhyJXZ0HHQ.

What We Learned?

Before proceeding ahead, let's remind ourselves about the new program constructs and techniques that we have learned.

Nested Conditions

```
if (condition1)
{
    if (condition2)
        // body;
    else
        // body;
}
```

Complex Conditions with &&, ||, ! and ()

```
if ((x == left || x == right) && y >= top && y <= bottom)
    Console.WriteLine(…);
```

Switch-Case Conditions

```
switch (selector)
{
    case value1:
        construction;
        break;
    case value2:
    case value3:
        construction;
        break;
    …
    default:
        construction;
        break;
}
```

Problem: Cinema

In a cinema hall the chairs are ordered in a **rectangle** shape in **r** rows and **c** columns. There are three types of screenings with tickets of **different** prices:

- **Premiere** – a premiere screening, with price **12.00** levs.
- **Normal** – a standard screening, with price **7.50** levs.
- **Discount** – a screening for children and students on a reduced price – **5.00** levs.

Write a program that enters a **type of screening** (string), number of **rows** and number of **columns** in the hall (integer numbers) and calculates **the total income** from tickets from a **full hall**. The result has to be printed in the same format as in the examples below – rounded up to 2 digits after the decimal point.

Sample Input and Output

Input	Output	Input	Output
Premiere 10 12	1440.00 leva	Normal 21 13	2047.50 leva

Hints and Guidelines

While reading the input, we could convert the screening type into small letters (with the function **`.ToLower()`**). We create and initialize a variable that will store the calculated income. In another variable we calculate the full capacity of the hall. We use a **`switch-case`** conditional statement to calculate the income according to the type of the projection and print the result on the console in the given format (look for the needed **C#** functionality on the internet).

Sample code (parts of the code are blurred with the purpose to stimulate your thinking and solving skills):

```
string type = Console.ReadLine().ToLower();
int rows = int.Parse(Console.ReadLine());
int columns = int.Parse(Console.ReadLine());

int full = rows * columns;
double income = 0;

switch (type)
{
    case "premiere":
        income = full * 12.00;
        break;
    case "normal":
        income = full * 7.50;
        break;
    case "discount":
        income = full * 5.00;
        break;
}

Console.WriteLine($"{income:f2} leva");
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#10.

Problem: Volleyball

Vladimir is a student, lives in Sofia and goes to his hometown from time to time. He is very keen on volleyball, but is busy during weekdays and plays **volleyball** only during **weekends** and on **holidays**. Vladimir plays **in Sofia** every **Saturday**, when **he is not working**, and **he is not traveling to his hometown** and also during **2/3 of the holidays**. He travels to his **hometown h times** a year, where he plays volleyball with his old friends on **Sunday**. Vladimir **is not working 3/4 of the weekends**, during which he is in Sofia. Furthermore, during **leap years** Vladimir plays **15% more** volleyball than usual. We accept that the year has exactly **48 weekends**, suitable for volleyball. Write a program that calculates **how many times Vladimir has played volleyball** through the year. **Round the result** down to the nearest whole number (e.g. 2.15 -> 2; 9.95 -> 9).

The input data is read from the console:

- The first line contains the word "**leap**" (leap year) or "**normal**" (a normal year with 365 days).
- The second line contains the integer **p** – the count of holidays in the year (which are not Saturday or Sunday).
- The third line contains the integer **h** – the count of weekends, in which Vladimir travels to his hometown.

Sample Input and Output

Input	Output	Input	Output
leap 5 2	45	normal 3 2	38

Input	Output	Input	Output
normal 11 6	44	leap 0 1	41

Hints and Guidelines

As usual, we read the input data from the console and, to avoid making mistakes, we convert the text into small letters with the function `.ToLower()`. Consequently, we calculate **the weekends spent in Sofia**, **the time for playing in Sofia** and **the common playtime**. At last, we check whether the year is **leap**, we make additional calculation when necessary and we print the result on the console **rounded down** to the nearest **integer** (look for a **C#** class with such functionality).

A sample code (parts of the code are blurred on purpose to stimulate independent thinking and solving skills):

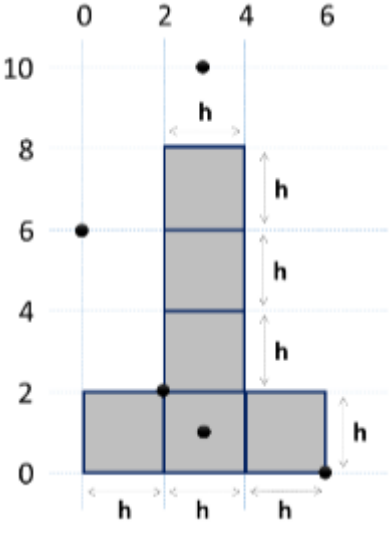
```
string year = Console.ReadLine().ToLower();
int holidays = int.Parse(Console.ReadLine());
int weekendsHome = int.Parse(Console.ReadLine());

int sofiaWeekends = 48 - weekendsHome;
double playSofia = 3.0 * sofiaWeekends / 4 + 2.0 * holidays / 3;
double playTotal = playSofia + weekendsHome;

if (year.Equals("leap"))
{
    playTotal = Math.Floor(playTotal * 15 / 100 + playTotal);
}
else if (year.Equals("normal"))
{
    playTotal = Math.Floor(playTotal);
}

Console.WriteLine(playTotal);
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#11.

Problem: * Point in the Figure

The figure consists of **6 blocks with size h * h**, placed as in the figure below. The lower left angle of the building is on position {0, 0}. The upper right angle of the figure is on position {**2*h**, **4*h**}. The coordinates given in the figure are for **h = 2**:



Write a program that enters an integer **h** and the coordinates of a given **point {x, y}** (integers) and prints whether the point is inside the figure (**inside**), outside of the figure (**outside**) or on any of the borders of the figure (**border**).

Sample Input and Output

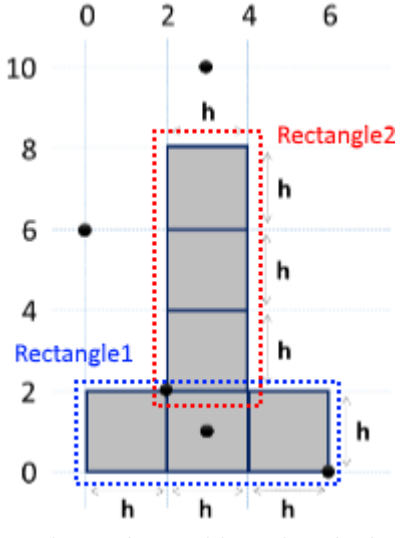
Input	Output	Input	Output
2 3 10	outside	2 3 1	inside

Input	Output	Input	Output
2 2 2	border	2 6 0	border
Input	Output	Input	Output
2 0 6	outside	15 13 55	outside
Input	Output	Input	Output
15 29 37	inside	15 37 18	outside
Input	Output	Input	Output
15 -4 7	outside	15 30 0	border

Hints and Guidelines

A possible logic for solving the task (not the only correct one):

- We might split the figure into **two rectangles** with a common side:



- A point is **outer (outside)** for the figure, when it is **outside** both of the rectangles.
- A point is **inner (inside)** for the figure, if it is inside one of the rectangles (excluding their borders) or lies on their common side.
- In **other case** the point lies on the border of the rectangle (**border**).

Implementation of the Proposed Idea

An exemplary implementation of the described idea (parts of the code are blurred with the purpose of stimulating logical thinking and solving skills):

```
int h = int.Parse(Console.ReadLine());
int x = int.Parse(Console.ReadLine());
int y = int.Parse(Console.ReadLine());

bool outRectangle1 = (x < 0 || x > 3 * h) || (y < 0 || y > h);
bool outRectangle2 = (x < h || x > 2 * h) || (y < h || y > 4 * h);

bool inRectangle1 = (x > 0 && x < 3 * h) && (y > 0 && y < h);
bool inRectangle2 = (x > h && x < 2 * h) && (y > h && y < 4 * h);

bool commonBorder = (x > h && x < 2 * h) && y == h;

if (outRectangle1 && outRectangle2)
{
    Console.WriteLine("outside");
}
else if (inRectangle1 || inRectangle2 || commonBorder)
{
    Console.WriteLine("inside");
}
else
{
    Console.WriteLine("border");
}
```

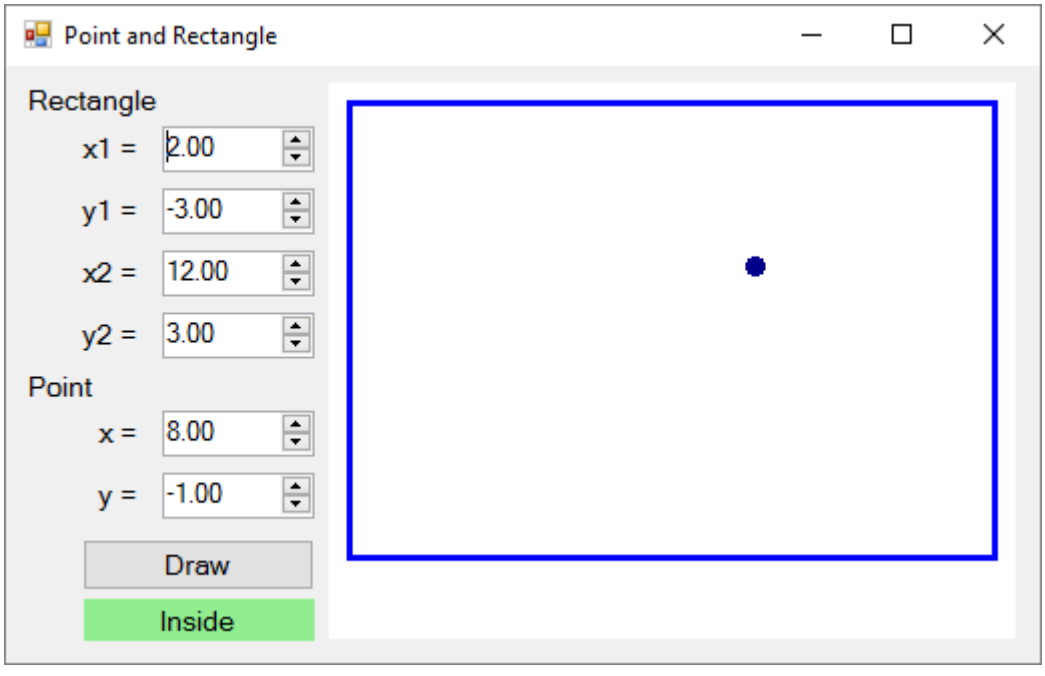
Testing in the Judge System

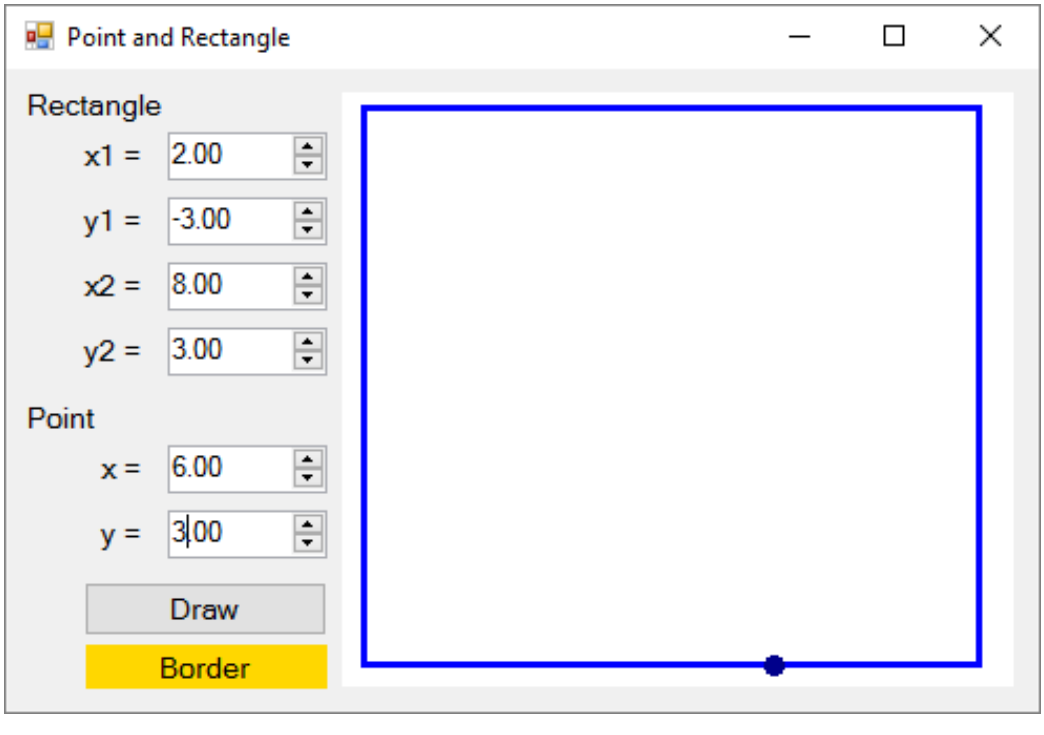
Test your solution here: https://judge.softuni.org/Contests/Practice/Index/508#12.

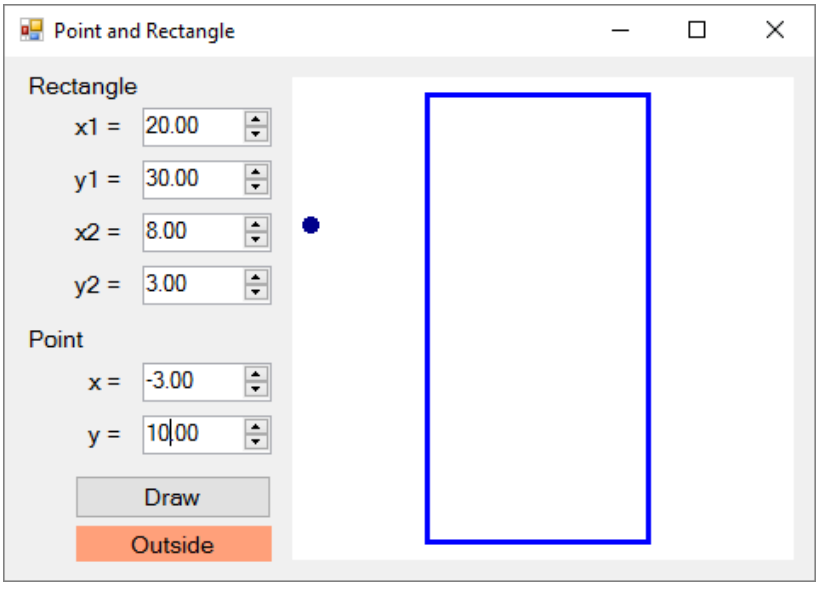
Lab: * GUI (Desktop) Application – Point and Rectangle

In this mini book we learned how we can make **statements with non-trivial conditions**. Now let's apply this knowledge to create something interesting: a **desktop (GUI) app that visualizes a point in a rectangle**. This is a wonderful visualization of one of the tasks from the exercises.

The task that we have is to develop a graphical (**GUI**) application for **visualizing a point and a rectangle**. The application must look like identically to the following:


Point and Rectangle
Rectangle
x1 = 2.00
y1 = -3.00
x2 = 12.00
y2 = 3.00
Point
x = 8.00
y = -1.00
Draw
Inside


Point and Rectangle
Rectangle
x1 = 2.00
y1 = -3.00
x2 = 8.00
y2 = 3.00
Point
x = 6.00
y = 3.00
Draw
Border

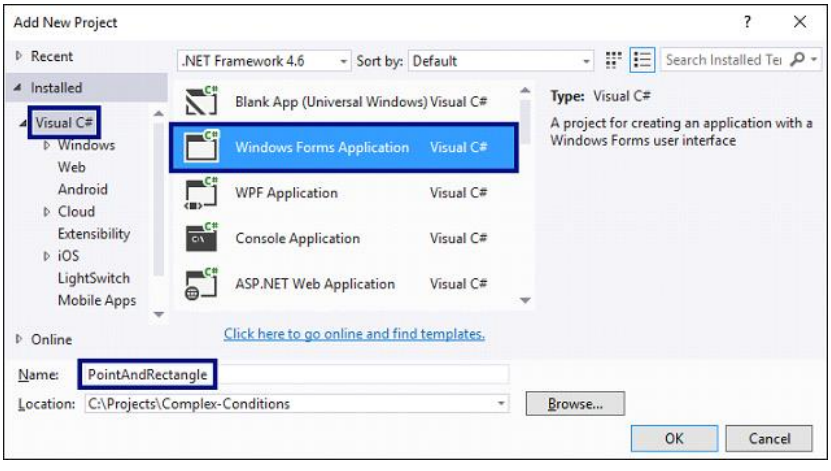


Using the controls on the left we set the coordinates of **two of the angles of the rectangle** (decimal numbers) and the coordinates of the **point**. The application **visualizes graphically** the rectangle and the point and prints whether the point is **inside** the rectangle (**Inside**), **outside** of it (**Outside**) or on one of its sides (**Border**). The application **moves and resizes** the coordinates of the rectangle and the point to be maximum large, but to fit the field for visualization in the right side of the application.

Attention: this application is significantly **more complex** than the previous graphical applications, which we have developed until now, because it requires using functions for drawing and non-trivial calculations for resizing and moving the rectangle and the point. Instructions for building the application step by step follow.

Creating a New C# Project and Adding Controls

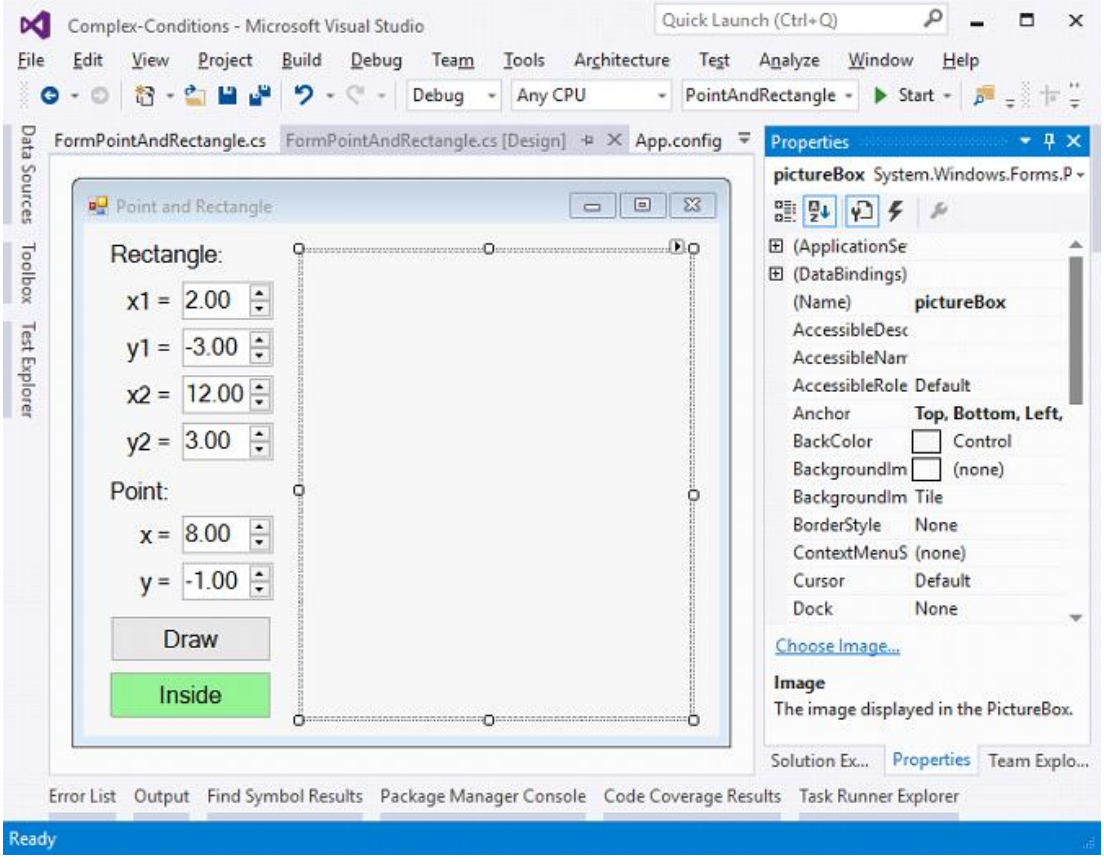
We create a new project **Windows Forms Application** with a suitable name, for example "Point-and-Rectangle":



We arrange the controls inside the form, as it is shown in the figure below:

- 6 boxes for entering a number (`NumericUpDown`).
- Labels (`Label`) before each box for entering a number.
- A button (`Button`) for drawing the rectangle and the point.
- A text block for the result (`Label`).

We set the sizes and properties of the controls to look as close as the ones in the picture:



Configuring the UI Controls

We set the following recommended settings of the controls:

- For the main form (`Form`) that contains all of the controls:
  - (name) = `FormPointAndRectangle`
  - `Text` = `Point and Rectangle`
  - `Font.Size` = `12`
  - `Size` = `700`, `410`
  - `MinimumSize` = `500`, `400`
  - `FormBorderStyle` = `FixedSingle`

- For the fields for entering a number (**NumericUpDown**):
  - (name) = **numericUpDownX1**; **numericUpDownY1**; **numericUpDownX2**; **numericUpDownY2**; **numericUpDownX**; **numericUpDownY**
  - **Value** = **2**; **-3**; **12**; **3**; **8**; **-1**
  - **Minimum** = **-100000**
  - **Maximum** = **100000**
  - **DecimalPlaces** = **2**
- For the button (**Button**) for visualization of the rectangle and the point:
  - (name) = **buttonDraw**
  - Text = **Draw**
- For the text block for the result (**Label**):
  - (name) = **labelLocation**
  - **AutoSize** = **false**
  - **BackColor** = **PaleGreen**
  - **TextAlign** = **MiddleCenter**
- For the field with the draft (**PictureBox**):
  - (name) = **pictureBox**
  - **Anchor** = **Top**, **Bottom**, **Left**, **Right**

Handling Events

We have to catch the following events to write the C# code that will be executed upon their occurrence:

- The event **Click** the button **buttonDraw** (it is called upon pressing the button).
- The event **ValueChanged** of the controls for entering numbers **numericUpDownX1**, **numericUpDownY1**, **numericUpDownX2**, **numericUpDownY2**, **numericUpDownX** and **numericUpDownY** (it is called upon changing the value in the control that enters a number).
- The event **Load** of the form **FormPointAndRectangle** (it is called upon starting the application, before the main form is shown on the display).
- The event **Resize** of the form **FormPointAndRectangle** (it is called upon changing the size of the main form).

All of the above-mentioned events will execute the same action – **Draw()**, which will visualize the rectangle and the point and show whether it's inside, outside or onto one of the sides. The code must look like this:

```
private void buttonDraw_Click(object sender, EventArgs e)
{
    Draw();
}
```

```
private void FormPointAndRectangle_Load(object sender, EventArgs e)
{
    Draw();
}

private void FormPointAndRectangle_Resize(object sender, EventArgs e)
{
    Draw();
}

private void numericUpDownX1_ValueChanged(object sender, EventArgs e)
{
    Draw();
}

 /* TODO: implement in the same way event handlers
    numericUpDownY1_ValueChanged,
    numericUpDownX2_ValueChanged,
    numericUpDownY2_ValueChanged,
    numericUpDownX_ValueChanged and
    numericUpDownY_ValueChanged */

private void Draw()
{
    // TODO: implement this a bit later …
}
```

Printing Point Position Compared to the Rectangle

Let's begin from the easier part: printing the information about the point's position (Inside, Outside or Border). The code must look like this:

```
private void Draw()
{
    // Get the rectangle and point coordinates from the form
    var x1 = this.numericUpDownX1.Value;
    var y1 = this.numericUpDownY1.Value;
    var x2 = this.numericUpDownX2.Value;
    var y2 = this.numericUpDownY2.Value;
    var x = this.numericUpDownX.Value;
    var y = this.numericUpDownY.Value;

    // Display the location of the point: Inside / Border / Outside
    DisplayPointLocation(x1, y1, x2, y2, x, y);
}

private void DisplayPointLocation(
    decimal x1, decimal y1, decimal x2, decimal y2, decimal x, decimal
y)
{
    var left = Math.Min(x1, x2);
```

```
    var right = Math.Max(x1, x2);
    var top = Math.Min(y1, y2);
    var bottom = Math.Max(y1, y2);
    if (x > left && x < right && …)
    {
        this.labelLocation.Text = "Inside";
        this.labelLocation.BackColor = Color.LightGreen;
    }
    else if (… || y < top || y > bottom)
    {
        this.labelLocation.Text = "Outside";
        this.labelLocation.BackColor = Color.LightSalmon;
    }
    else
    {
        this.labelLocation.Text = "Border";
        this.labelLocation.BackColor = Color.Gold;
    }
}
```

The code above takes the coordinates of the rectangle and the point and checks whether the point is inside, outside or on the borders of the rectangle. By visualizing the result, the color of the background of the text block that contains it is changed.

Think about how to finish the uncompleted (on purpose) conditions in the **`if`** statements! The code above purposely doesn't compile, because the purpose is to make you think about how and why it works and finish on your own the missing parts.

Visualization of the Rectangle and the Point

What remains is to implement the most complex part: visualization of the rectangle and the point in the control **`pictureBox`** with resizing. We can help ourselves with the code below, which makes some calculations and draws a blue rectangle and a dark blue circle (the point) according to the coordinates given in the form. Unfortunately, the complexity of the code exceeds the material learned until the present moment and it is complicated to explain in detail exactly how it works. There are comments for orientation. This is the full version of the action **`Draw()`**:

```
private void Draw()
{
  // Get the rectangle and point coordinates from the form
  var x1 = this.numericUpDownX1.Value;
  var y1 = this.numericUpDownY1.Value;
  var x2 = this.numericUpDownX2.Value;
  var y2 = this.numericUpDownY2.Value;
  var x = this.numericUpDownX.Value;
  var y = this.numericUpDownY.Value;

  // Display the location of the point: Inside / Border / Outside
  DisplayPointLocation(x1, y1, x2, y2, x, y);

  // Calculate the scale factor (ratio) for the diagram holding the
  // rectangle and point in order to fit them well in the picture box
```

```
    var minX = Min(x1, x2, x);
    var maxX = Max(x1, x2, x);
    var minY = Min(y1, y2, y);
    var maxY = Max(y1, y2, y);
    var diagramWidth = maxX - minX;
    var diagramHeight = maxY - minY;
    var ratio = 1.0m;
    var offset = 10;
    if (diagramWidth != 0 && diagramHeight != 0)
    {
      var ratioX = (pictureBox.Width - 2 * offset - 1) / diagramWidth;
      var ratioY = (pictureBox.Height - 2 * offset - 1) / diagramHeight;
      ratio = Math.Min(ratioX, ratioY);
    }

    // Calculate the scaled rectangle coordinates
    var rectLeft = offset + (int)Math.Round((Math.Min(x1, x2) - minX) *
ratio);
    var rectTop = offset + (int)Math.Round((Math.Min(y1, y2) - minY) *
ratio);
    var rectWidth = (int)Math.Round(Math.Abs(x2 - x1) * ratio);
    var rectHeight = (int)Math.Round(Math.Abs(y2 - y1) * ratio);
    var rect = new Rectangle(rectLeft, rectTop, rectWidth, rectHeight);

    // Calculate the scaled point coordinates
    var pointX = (int)Math.Round(offset + (x - minX) * ratio);
    var pointY = (int)Math.Round(offset + (y - minY) * ratio);
    var pointRect = new Rectangle(pointX - 2, pointY - 2, 5, 5);

    // Draw the rectangle and point
    pictureBox.Image = new Bitmap(pictureBox.Width, pictureBox.Height);
    using (var g = Graphics.FromImage(pictureBox.Image))
    {
      // Draw diagram background (white area)
      g.Clear(Color.White);

      // Draw the rectangle (scaled to the picture box size)
      var pen = new Pen(Color.Blue, 3);
      g.DrawRectangle(pen, rect);

      // Draw the point (scaled to the picture box size)
      pen = new Pen(Color.DarkBlue, 5);
      g.DrawEllipse(pen, pointRect);
    }
}

private decimal Min(decimal val1, decimal val2, decimal val3)
{
    return Math.Min(val1, Math.Min(val2, val3));
}
```

```
private decimal Max(decimal val1, decimal val2, decimal val3)
{
  return Math.Max(val1, Math.Max(val2, val3));
}
```

In the code above we can see a lot of conversion of types, because different types of numbers are used (decimal numbers, real numbers and integers) and sometimes it is required to do conversion between them.

Compiling and Testing the Application

In the end we compile the code. If there are errors, we eliminate them. The most probable reason for an error is an inconsistent name of some of the controls or if writing the code in the wrong place.

We start the application and test it. We enter different data to see whether it behaves correctly.

What's Next?

The "Programming Basics" series consists of publications for beginners in programming that covers the following topics:

1. First Steps in Programming – commands, programs, C#, Visual Studio,
2. Simple Calculations – variables, calculations, console input / output
3. Simple Conditions – conditional statements, the "if-else" construction
4. More Complex Conditions – nested if-else, logical "and", "or", "not"
5. Repetitions (Loops) – simple for-loops (repeat from 1 to n)
6. Nested Loops – nested loops and problem solving, drawing 2D figures
7. More Complex Loops – loops with a step, infinite loops with breaks
8. How to Become a Software Engineer?

The next topics are coming. Be patient!

Sign-Up to Study in SoftUni

The easiest way to become a software engineer is to go through the "Software University" training program at SoftUni. Signup now for the free Programming Basics training course:

https://softuni.org/apply

Join the SoftUni community and study programming for free in our interactive training platform. Get live support and mentoring from our trainers. Become a software developer now!