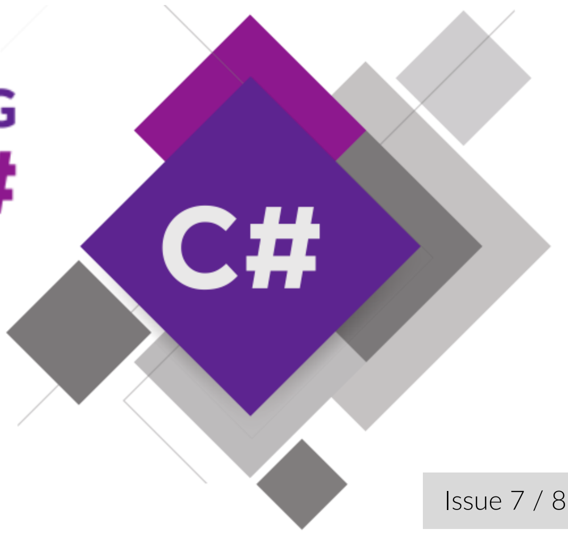
PROGRAMMING BASICS WITH C#



Issue 7 / 8

MORE COMPLEX LOOPS

Learn about how to write loops with a step, infinite loops with breaks, while loops, do-while loops and other loop constructions, together with a lot of practical exercises. Learn how to use "try-catch" to handle exceptions.

Get an idea how to build simple Web application "Fruits Game" in C# and ASP.NET MVC using Visual Studio.

Dr. Svetlin Nakov

Software University – https://softuni.org

More Complex Loops

Once we have learned what loops are and what **the `for` loops** serve for, now is the time to take a look at **other types of loops** as well as some **more complex loops constructions**. They will expand our knowledge and help us solve difficult and challenging problems. In particular, we will discuss how to use the following program constructions:

- loops **with step**
- `while` loops
- `do-while` loops
- **infinite** loops

In this mini book, we will also understand what the `break` operator is and **how** to **break** a loop. Also, using the `try-catch` construction, we will learn to keep track of **errors** during our program's execution.

Video: Overview

Watch a video lesson to review what shall we learn about the special and more complex types of loops: https://youtu.be/J18RgaaMi7U.

Introduction to More Complex Loops by Examples

Loops repeat a piece of code many times while a condition holds and usually changes the so called "**loop variable**" after each iteration. The loop variable using a **certain step**, e.g. 5 or -2. Example of a `for` loop from 10 down to 0, using a **step** -2:

```
for (int i = 10; i >= 0; i-=2)
    Console.Write(i + " ");
// Output: 10 8 6 4 2 0
```

Run the above code example: https://repl.it/@nakov/for-loop-step-minus-2-csharp.

One of the simplest loops in programming is the **while-loop**. It repeats a block of code while a condition is true:

```
int n = 5;
int factorial = 1;
while (n > 1)
{
    factorial = factorial * n;
    n--;
}
Console.WriteLine(factorial);
// Output: 120
```

Run the above code example: https://repl.it/@nakov/while-loop-factorial-csharp.

Another example of loops is the `do-while` **loop**. It repeats a code block while a condition holds. For example, we can calculate the minimum number **k**, such that $2^k > 1,000,000,000$, using the code below:

```
int num = 1, count = 0;
do
{
    count++;
    num = num * 2;
} while (num <= 1000000000);
Console.WriteLine("2^{0} = {1}", count, num);
// Output: 2^30 = 1073741824
```

Run the above code example: https://repl.it/@nakov/do-while-loop-power-of-2-calculation-csharp.

Sometime in programming we don't know in advance how many times to repeat a loop, neither we have a clear loop condition, so we may use **infinite loop with exit condition** inside the loop. For example, we want to print the first 5 results, matching certain condition, calculated inside a loop. We use **infinite loop** and exit it using **the `break` operator**:

```
int value = 0, min = 100000, count = 0;
while (true)
{
    value = 2 * value + 1;
    if (value > min)
    {
        Console.WriteLine(value);
        count++;
    }
    if (count == 5)
        break;
}
```

Run the above code example: https://repl.it/@nakov/infinite-loop-with-break-csharp.

Let's get into details on how to use **`for` loops with a step**, how to use **`while` loops**, how to use **`do-while` loops** and how to design a program logic, based on **infinite loops with a `break`**.

For Loop with Step

In this mini book we will pay **attention** to a particular and very important part of its construction, namely the **step**.

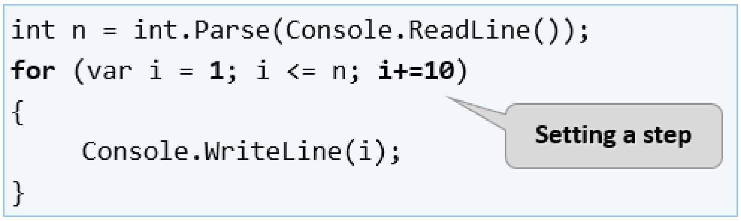
Video: Loop with a Step

Watch the following video lesson to learn how to use for-loops with a custom step: https://youtu.be/QZDpWHcb7dE.

Loop with a Step – Explanation

The step is that part of the `for` loop construction that tells how much to increase or decrease the value of its leading variable. It is declared the last in the skeleton of the `for` loop.

Most often, we have a size of `1`, and in this case, instead of writing `i += 1` or `i -= 1`, we can use the `i++` or `i--` operators. If we want our step to be different than 1, when increasing, we use the `i +=` + `step size` operator, and when decreasing, the `i -=` + `step size`. With step of 10, the loop would look like this:



```
int n = int.Parse(Console.ReadLine());
for (var i = 1; i <= n; i+=10)
{
     Console.WriteLine(i);
}
```

Here is a series of sample problems, the solution of which will help us better understand the use of the step in `for` loop.

Example: Numbers 1...N with Step 3

Write a program that prints the numbers from 1 to n with step of 3. For example, if n = 100, the result will be: 1, 4, 7, 10, ..., 94, 97, 100.

We can solve the problem using the following sequence of actions (algorithm):

- We read the number `n` from the console input.
- We run a `for` loop from 1 to `n` with step size of 3.
- In the body of the loop, we print the value of the current step.

```
int n = int.Parse(Console.ReadLine());

// by i+ =3 we increase the value of i with the size of the step
for (var i = 1; i <= n; i += 3)
{
    Console.WriteLine(i);
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#0.

Example: Numbers N...1 in Reverse Order

Write a program that prints the numbers from n to 1 in reverse order (step of -1). For example, if n = 100, the result will be: 100, 99, 98, ..., 3, 2, 1.

Video: Numbers N...1

Watch this video lesson to learn how to print the numbers from N down to 1 (in reverse order) using a for loop: https://youtu.be/LGPZ-ug3qh0.

Hints and Guidelines

We can solve the problem in the following way:

- We read the number `n` from the console input.
- We create a `for` loop by assigning `int i = n`.
- We reverse the condition of the loop: `i >= 1`.
- We define the size of the step: -1.
- In the body of the loop, we print the value of the current step.

```
int n = int.Parse(Console.ReadLine());

// Reversed condition: i >= 1
// Negative step: i--
for (int i = n; i >= 1; i--)
{
    Console.WriteLine(i);
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#1.

Example: Numbers from 1 to 2^n with a For Loop

In the following example, we will look at using the usual step with size of 1, combined with a calculation at each loop iteration.

Write a program that prints the numbers from 1 to 2^n (two in power of n). For example, if n = 10, the result will be: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

Video: Numbers 1 ... 2^n

Watch this video lesson to learn how to iterate over the number from 1 to 2^n using a for-loop: https://youtu.be/B2k_yx3EV0I.

Hints and Guidelines

The code below demonstrates how we can calculate the powers of 2 for given n using a for-loop with a calculation at the end of its body:

```
int n = int.Parse(Console.ReadLine());
int num = 1;
for (var i = 0; i <= n; i++)
{
    Console.WriteLine(num);
    num = num * 2;
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#2.

Example: Even Powers of 2

Print **the even** powers of **2** to **2^n**: **2^0**, **2^2**, **2^4**, **2^6**, **2^8**, **…**, **2^n**. For example, if **n = 10**, the result will be: **1, 4, 16, 64, 256, 1024**.

Video: Even Powers of 2

Watch this video lesson to learn how to print the even powers of 2 using a for loop: https://youtu.be/H8t4HGn_Ap4.

Hints and Guidelines

Here is how we can solve the problem using a **for-loop with a step**:

- We create a **`num`** variable for the current number to which we assign an initial **value of 1**.
- For **a step** of the loop, we set a value of **2**.
- In **the body of the loop**: we print the value of the current number and **increase the current number `num` 4 times** (according to the problem's description).

```
int n = int.Parse(Console.ReadLine());
int num = 1;
for (var i = 0; i <= n; i += 2)
{
    Console.WriteLine(num);
    num = num * 2 * 2;
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#3.

While Loop

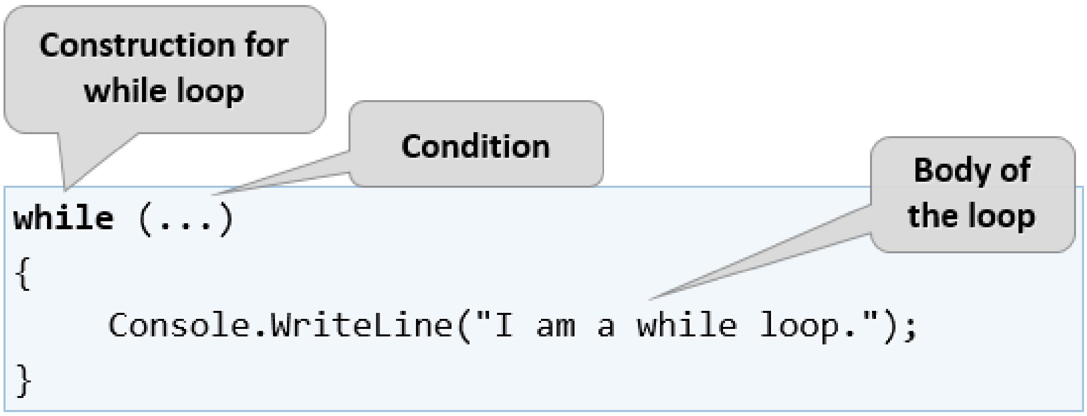
The next type of loops that we will get familiar with are called **`while` loops**. The specific thing about them is that they repeat a block of commands **while a condition is true**. As a structure, they differ from **`for`** loops, and even have a simple syntax.

Video: While Loop

Watch this video lesson to learn how to use the while-loop in C#: https://youtu.be/Jqnxl6k1V9w.

While Loop – Explanation

In programming **the `while` loop** is used when we want to **repeat** the execution of a certain logic while **a condition is in effect**. By "**condition**," we understand every **expression** that returns **`true`** or `false`. When **the condition** is **wrong**, **the `while` loop** is **interrupted**, the program **continues** to execute the remaining code after the loop. **The `while` loop** construction looks like this:



```
while (...)
{
    Console.WriteLine("I am a while loop.");
}
```

Here is a series of sample problems, the solution of which will help us better understand the use of the `while` loop.

Example: Sequence of Numbers 2k+1

Write a program that prints all **numbers ≤ n** of the series: **1, 3, 7, 15, 31, …**, assuming that each next number = **previous number * 2 + 1**.

Here is how we can solve the problem:

- We create a `num` variable for the current number to which we assign an initial **value of 1**.
- For a loop condition, we put **the current number <= n**.
- In **the body of the loop**: we print the value of the current number and increase the current number by using the formula from the problem's description.

Here is a sample implementation of this idea:

```
int n = int.Parse(Console.ReadLine());
int num = 1;
while (num <= n)
{
    Console.WriteLine(num);
    num = 2 * num + 1;
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#4.

Example: Number in Range [1…100]

Enter an integer in the range [**1 … 100**]. If the entered number is **invalid**, enter it **again**. In this case, an invalid number will be any number that **is not** within the specified range.

Video: Numbers in the Range [1…100]

Watch this video lesson to learn how to enter a number in the range [1…100]: https://youtu.be/8W-CIbF4cdA.

Hints and Guidelines

To solve the problem, we can use the following algorithm:

- We create a `num` variable to which we assign the integer value obtained from the console input.
- For a loop condition, we put an expression that is `true` if the number of the input **is not** in the range specified in the problem's description.
- In **the body of the loop**: we print a message "**Invalid number!**" on the console, then we assign a new value to `num` from the console input.
- Once we have validated the entered number, we print the value of the number outside the body of the loop.

Here's a sample implementation of the algorithm using a `while` **loop**:

```
var num = int.Parse(Console.ReadLine());
while (num < 1 || num > 100)
{
    Console.WriteLine("Invalid number!");
    num = int.Parse(Console.ReadLine());
}
Console.WriteLine("The number is: {0}", num);
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#5.

Greatest Common Divisor (GCD)

Before proceeding to the next problem, we will get familiar with the definition of **the greatest common divisor** (GCD), widely used in mathematics and numbers theory, and will learn how to calculate GCD.

Definition of GCD: the greatest common divisor of two **natural** numbers **a** and **b** is the largest number that divides **both a** and **b** without reminder.

a	b	GCD
24	16	8
67	18	1
12	24	12
15	9	3

a	b	GCD
10	10	10
100	88	4

Video: Greatest Common Divisor (GCD)

Watch the video lesson to learn about the Euclidean algorithm for calculating the GCD of given two integers: https://youtu.be/1-SEOWupvrA.

The Euclidean Algorithm

In the next problem we will use one of the first published algorithms for finding the GCD – **Euclid's algorithm**.

Until we reach a remainder of 0:

- We divide the greater number by the smaller one.
- We take the remainder of the division.

Euclid's algorithm **pseudo-code**:

```
while b ≠ 0
  var oldB = b;
  b = a % b;
  a = oldB;
print a;
```

Example: Greatest Common Divisor (GCD)

Enter **integers a** and **b** and find **GCD(a, b)**.

We will solve the problem through **Euclid's algorithm**:

- We create variables `a` and `b` to which we assign **integer** values taken from the console input.
- For a loop condition, we put an expression that is `true` if the number `b` **is different** from 0.
- In **the body of the loop** we follow the instructions from the pseudo code:
  - We create a temporary variable to which we assign **the current** value of `b`.
  - We assign a new value to `b`, which is the remainder of the division of `a` and `b`.
  - On the variable `a` we assign **the previous** value of the variable `b`.
- Once the loop is complete and we have found the GCD, we print it on the screen.

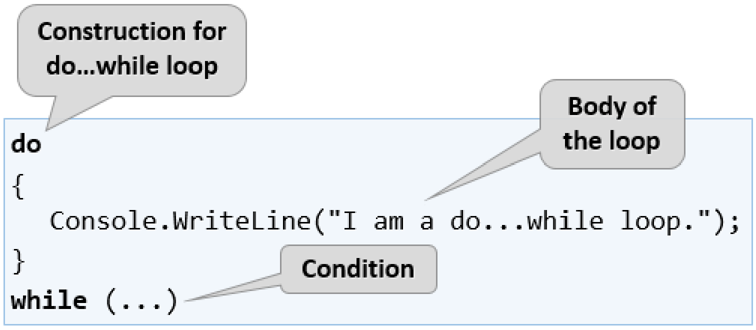
```
var a = int.Parse(Console.ReadLine());
var b = int.Parse(Console.ReadLine());
while (b != 0)
{
    var oldB = b;
    b = a % b;
    a = oldB;
}
Console.WriteLine("GCD = {0}", a);
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#6.

Do-While Loop

The next type of loops we will get familiar with are the **do-while** loops. By structure, this type of loop resembles the **while** loop, but there is a significant difference between them. It is that the **do-while** loop will execute its body at least once. Why is this happening? In the **do-while** loop construction, the condition is always checked after the body, which ensures that the first loop iteration will execute the code and the check for the end of the loop will be applied to each subsequent iteration of the **do-while**.



```
do
{
  Console.WriteLine("I am a do...while loop.");
}
while (...)
```

The usual set of sample problems follows. Their solutions will help us better understand the **do-while** loops.

Video: Do-While Loop

Watch a video lesson about the do-while loop and how to use it: https://youtu.be/hEJ9-INyahU.

Example: Calculating Factorial

For natural n number, calculate n! = 1 * 2 * 3 * ... * n. For example, if n = 5, the result will be: 5! = 1 * 2 * 3 * 4 * 5 = 120.

Here is how we can specifically calculate factorial:

- We create the variable **n** to which we assign an integer value taken from the console input.

- We create another variable – a **`fact`** which initial value is 1. We will use it for the calculation and storage of the factorial.
- For a loop condition, we will use **`n > 1`**, because each time we perform the calculations in the body of the loop, we will decrease the value of **`n`** by 1.
- In the body of the loop:
  - We assign a new value to **`fact`** that is the result of multiplying the current **`fact`** value to the current value of **`n`**.
  - We decrease the value of **`n`** by -1.
- Outside the body of the loop, we print the final factorial value.

```
var n = int.Parse(Console.ReadLine());
var fact = 1;
do
{
    fact = fact * n;
    n--;
}
while (n > 1);
Console.WriteLine(fact);
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#7.

Example: Summing Up Digits

Let's practice the **`do-while`** loop with the following exercise:

Sum up the digits of a **positive** integer **n**. Examples:

- If **n = 5634**, the result will be: 5 + 6 + 3 + 4 = **18**.
- If **n = 920**, the result will be: 9 + 2 + 0 = **11**.

Video: Sum of Digits

Watch this video lesson to learn how to sum the digits of given integer: https://youtu.be/sbzlzdoEbFc.

Hints and Guidelines

We can use the following **idea to solve the problem**: extract many times the last digit from the input number and sum the extracted digits until the input number reaches 0. Example:

- sum = 0
- n = 5634 → extract 4; sum += 4; n = 563
- n = 563 → extract 3; sum += 3; n = 56
- n = 56 → extract 6; sum += 6; n = 5

- n = 5 ⍰ extract 5; sum += 5; n = 0 ⍰ end

In more detail the above idea looks like this:

- We create the variable **`n`**, to which we assign a value equal to the number entered by the user.
- We create a second variable – **`sum`**, which initial value is 0. We will use it for the calculation and storage of the result.
- As a loop condition, we will use **`n > 0`** because after each calculation of the result in the body of the loop, we will remove the last digit of **`n`**.
- In the body of the loop:
    - We assign a new value of **`sum`** that is the result of the sum of the current value of **`sum`** with the last digit of **`n`**.
    - We assign a new value to **`n`**, which is the result of removing the last digit of **`n`**.
- Outside the body of the loop, we print the final value of the sum.

```csharp
var n = int.Parse(Console.ReadLine());
var sum = 0;
do
{
    sum = sum + (n % 10);
    n = n / 10;
}
while (n > 0);
Console.WriteLine("Sum of digits: {0}", sum);
```

`n % 10`: returns the last digit of the number `n`.
`n / 10`: deletes the last digit of `n`.

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#8.

Infinite Loops with Break

So far, we were introduced to various types of loops, learning what structures they have and how they are applied. Now, we need to understand what an infinite loop is, when it occurs, and how we can break it using the **`break`** operator.

Video: Infinite Loops with Break

Watch this video lesson to learn how to use infinite loops, along with the `break` operator: https://youtu.be/rpez6b9TpdA.

Infinite Loop – Explanation

We call an infinite loop one that repeats infinitely the performance of its body. In **while** and **do-while** loops the end check is a conditional expression that always returns **true**. Infinite **for** occurs when there is no condition to end the loop.

Here is what an infinite **while** loop looks like:

```
while (true)
{
    Console.WriteLine("Infinite loop");
}
```

And here is what an infinite **for** loop looks like:

```
for (;;)
{
    Console.WriteLine("Infinite loop");
}
```

The Operator "Break"

We already know that the infinite loop performs a certain code infinitely, but what if we want at some point under a given condition to go out of the loop? The **break** operator comes in handy in this situation.

⚠	The **break** operator stops the execution of a loop at the time it is called and continues from the first line after the end of the loop. This means that the current iteration of the loop will not be completed, accordingly, the rest of the code in the body of the loop will not be executed.

Example: Prime Number Checking

The next problem we are going to solve is to check whether given number is prime. An integer is prime if it cannot be decomposed to a product of other numbers. For example: 2, 5 and 19 are primes, while 9, 12 and 35 are composite.

Video: Prime Number Checking

Watch this video lesson to learn how to design and implement an algorithm to check if given number is prime: https://youtu.be/4lWOaPWKf0I.

Hints and Guidelines

Before proceeding to the hints about solving the "prime checking" problem, let's recall in bigger detail what prime numbers are.

Definition: an integer is prime if it is divisible only by itself and by 1. By definition, the prime numbers are positive and greater than 1. The smallest prime number is 2.

We can assume that an integer n is a prime number if n > 1 and n is not divisible by a number between 2 and n-1.

The first few prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, …

Unlike the prime numbers, **composite numbers** are integers which can be obtained by multiplying several prime numbers.

Here are some examples of **composite numbers**:

- **10** = 2 * 5
- **42** = 2 * 3 * 7
- **143** = 13 * 11

Positive integers, greater than 1, can be either **prime** or **composite** (product of primes). Numbers like **0** and **1** are not prime, but are also not composite.

We can **check if an integer is prime** following the definition: check if **n > 1** and **n** is divisible by **2**, **3**, ..., **n-1** without reminder.

- If it is divisible by any of the numbers, it is **composite**.
- If it is not divisible by any of the numbers, then it is **prime**.

We can optimize the algorithm instead of checking it to **`n-1`**, to check divisors to **`√n`**. Think what the reason for that is!

Prime Checking Algorithm

The most popular **algorithm** to check if a number **n** is prime is by checking if **n** is divisible by the numbers between 2 and √n.

The **steps** of the "prime checking algorithm" are given below in bigger detail:

- We create the variable **`n`** to which we assign an integer taken from the console input.
- We create an **`isPrime`** bool variable with an initial value **`true`**. We assume that a number is prime until proven otherwise.
- We create a **`for`** loop in which we set an initial value 2 for the loop variable, for condition **the current value `<= √n`**. The loop step is 1.
- In **the body of the loop**, we check if **`n`**, divided by **the current value**, has a remainder. If there is **no reminder** from the division, then we change **`isPrime`** to **`false`** and we exit the loop through the **`break`** operator.
- Depending on the value of **`isPrime`**, we print whether the number is prime (**`true`**) or composite (**`false`**).

Implementation of the Prime Checking Algorithm

Here is a **sample implementation** of the prime checking algorithm, described above:

```
var n = int.Parse(Console.ReadLine());
var prime = true;
for (var i = 2; i <= Math.Sqrt(n); i++)
{
    if (n % i == 0)
    {
        prime = false;
        break;
    }
}
if (prime)
{
    Console.WriteLine("Prime");
}
else
{
    Console.WriteLine("Not prime");
}
```

What remains is to add a **condition that checks if the input number is greater than 1**, because by definition numbers such as 0, 1, -1 and -2 are not prime.

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#9.

Example: Enter an Even Number

The next example will be to write a program that **enters an even number** from the console. If an odd number is entered, the program should enter a number again, until an even number is entered.

We shall use an **infinite loop with `break`** to solve this problem, because we don't know how many times the loop body will be repeated.

We shall check if a particular number **n** is even, and if it is, we will print it on the screen. An even number is one that can be divided by 2 without remainder. If an invalid number is entered, we will ask the user to enter a number again and will display a notification that the input number is not even.

Hints and Guidelines

Here is an idea how we can implement the above described logic:

- We create a variable **`n`** to which we assign an initial value of **0**.
- We create an infinite **`while`** loop and as condition we will set **`true`**.
- In **the body of the loop**:

  - We take an integer value from the console input and assign it to **`n`**.
  - If the number is even, we exit the loop by **`break`**.
  - Otherwise, we display a message stating that the number is not even. The iterations continue until an even number is entered.

- Finally, after the loop, print the even number on the screen.

Implementation

Here is an example implementation of the idea:

```csharp
var n = 0;
while (true)
{
    Console.Write("Enter even number: ");
    n = int.Parse(Console.ReadLine());
    if (n % 2 == 0)
    {
        break; // even number -> exit from the loop
    }
    Console.WriteLine("The number is not even.");
}
Console.WriteLine("Even number entered: {0}", n);
```

Note: Although the code above is correct, it will not work if the user enters text instead of numbers, such as "Invalid number". Then parsing the text to a number will break and the program will display an error message (exception). How to deal with this problem and how to capture and process exceptions using the **`try-catch`** construction will be learned later.

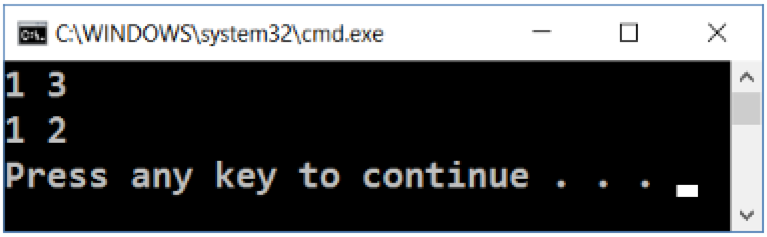
Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#10.

Nested Loops and Break

Once we have learned what the nested loops are and how the **`break`** operator works, it is time to figure out how they work together. For a better understanding, let's step by step write a program that should make all possible combinations of pairs of numbers. The first number of the combination is increasing from 1 to 3 and the second one is decreasing from 3 to 1. The problem must continue running until **`i + j`** is not equal to 2 (**`i = 1`** and **`j = 1`**).

The desired result is:



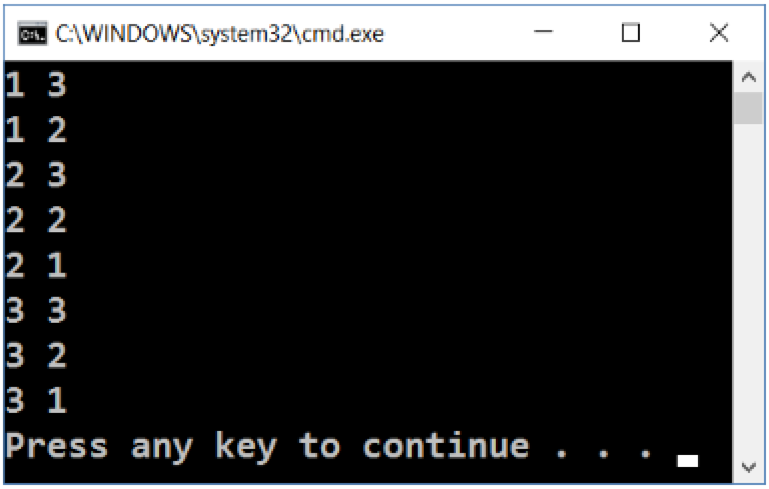
```
1 3
1 2
Press any key to continue . . .
```

Wrong Implementation

Here is a wrong solution that looks right at first sight:

```
for (int i = 1; i <= 3; i++)
{
    for (int j = 3; j >= 1; j--)
    {
        if (i + j == 2)
        {
            break;
        }
        Console.WriteLine(i + " " + j);
    }
}
```

If we leave our program that way, our result will be as follows:



```
1 3
1 2
2 3
2 2
2 1
3 3
3 2
3 1
Press any key to continue . . .
```

Why is it so? As we can see, the result is missing "1 1". When the program reaches that point that **`i = 1`** and **`j = 1`**, it enters the **`if`** check and executes the **`break`** operation. This way, it goes out of the inner loop, but then continues the performance of the outer one. **`i`** grows, the program enters the internal loop and prints the result.

When we use the **break** operator in a nested loop, it interrupts the execution of the inner loop only.

Correct Implementation

What is the right solution? One way to solve this problem is by declaring a **bool** variable to keep track if the loop rotation has to continue. If you need to exit (leave all nested loops), we set the variable to **true** and exit the inner loop with a **break**, and in the next check we exit the outer loop. Here is an example implementation of this idea:

```
bool hasToEnd = false;
for (int i = 1; i <= 3; i++)
{
    if (hasToEnd == false)
    {
        for (int j = 3; j >= 1; j--)
        {
            if (i + j == 2)
            {
                hasToEnd = true;
                break;
            }
            Console.WriteLine(i + " " + j);
        }
    }
}
```

Thus, when **i + j = 2**, the program will set the **hasToEnd = true** and exit the inner loop. Upon the next rotation of the outer loop, through the check, the program will not be able to reach the inner loop and will interrupt its execution.

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#11.

Handling Errors: Try-Catch

The last thing we will get familiar with is how to "capture" wrong data using the **try-catch** construction.

Video: Using Try-Catch

Watch this video lesson to learn how to use the **try-catch** statement to enter a valid integer number in certain range: https://youtu.be/0WLRjNcSh3I.

What is Try-Catch?

The `try-catch` construction is used to capture and handle exceptions (errors) during program execution.

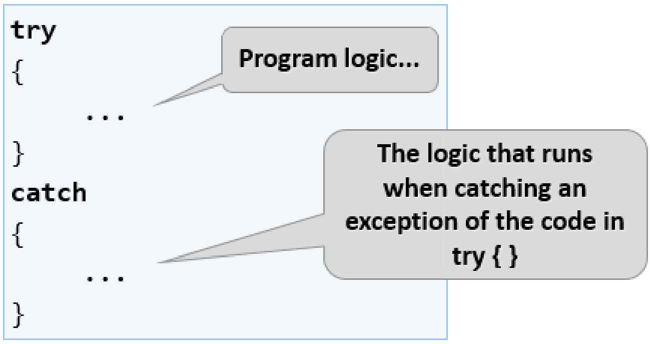
In programming, exceptions are a notification of an event that violates the normal operation of a program. Such exceptional events interrupt the execution of the program, and it is looking for something to process the situation. If it does not find it, the exception is printed on the console. If found, the exception is processed, and the program continues its normal execution. After a while, we'll see how this happens.

When an exception is found (e.g. when we divide an integer by zero), it is said that the exception was "thrown" (throw exception).

When the exception is handled and a piece of program logic recovers the program execution from the problem, we say the we "catch the exception".

The Try-Catch Construction

The `try-catch` construction in C# has different forms, but for now we will use the most basic of them:



```
try
{
    ...
}
catch
{
    ...
}
```

Program logic...
The logic that runs when catching an exception of the code in try { }

We have a piece of code (sequence of commands) inside the `try` block. If this code runs normally (without errors), all the commands in the `try` blocks are executed. If some of the commands in the `try` block throw and exception (in case of an error), the code execution is stopped, and the `catch` block is executed. In this case we say that we catch and handle the error (exception).

In the next task, we will see how to handle a situation where a user enters a non-numeric input (for example, a `string` instead of an `int`) by `try-catch`.

Example: Dealing with Invalid Numbers with Try-Catch

Write a program that checks if an n number is even, and if it is, prints it on the screen. If an invalid number is entered, the program should display a notification that the entered input is not a valid number and the entering of the number has to be done again.

Here's how we can solve the problem:

- We create an infinite `while` loop and as a condition we set `true`.
- In the body of the loop:
  - We create a `try-catch` construction.

- In the `try` block we write the programming logic for reading the user input, parsing it to a number, and the check for even number.
- If it is **an even number**, we print it and go out of the loop (with `break`). The program is done and ends.
- If it is **an odd number**, we print a message saying that an even number is required without leaving the loop (because we want it to be repeated again).
- If we **catch an exception** when executing the `try` block, we write a message for invalid input number (and the loop is repeated because we do not explicitly go out of it).

Enter Even Number – Implementation

Here is a **sample implementation** of the described idea:

```
while (true)
{
    try
    {
        Console.Write("Enter even number: ");
        int n = int.Parse(Console.ReadLine());
        if (n % 2 == 0)
        {
            Console.WriteLine("Even number entered: {0}", n);
            break;
        }
        Console.WriteLine("The number is not even.");
    }
    catch
    {
        Console.WriteLine("Invalid number.");
    }
}
```

Play with the above code. Try to enter invalid numbers (e.g. text messages), non-integer numbers, odd numbers and even numbers.

The solution should **work in all cases**: whether we are entering integer numbers, invalid numbers (for example, too many digits), or non-numbered text.

The above program logic will repeat in an infinite loop the process of **entering a value until a valid even integer is entered**.

- The `int.Parse()` method will **throw an exception** in case of an invalid integer.
- In case of a valid integer, the program will check if it is even. In this case a "*success*" message is shown, and the **loop is stopped** using `break`.
- In case of an odd integer, an **error message** is shown, and the **loop repeats again**.

- In case of an exception (error during the number parsing), an **error message** is shown, and the **loop repeats again**.

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#12.

Exercises: More Complex Loops

In this mini book, we got familiar with some new types of loops that can perform repetitions with more complex programming logic. Let's solve a few **practical problems** using these new constructs.

Video: Summary

Watch this video to review what we learned: https://youtu.be/6Wrna8Q0LFA.

What We Learned?

First, let's recall what we have learned.

We can use `for` loop with **a step**:

```
for (var i = 1; i <= n; i+=3)
{
   Console.WriteLine(i);
}
```

The `while` / `do-while` loops are repeated while a **condition** is in effect:

```
int num = 1;
while (num <= n)
{
   Console.WriteLine(num++);
}
```

If we have to **interrupt** the loop execution, we do it with the operator `break`:

```
var n = 0;
while (true)
{
   n = int.Parse(Console.ReadLine());
   if (n % 2 == 0)
   {
      break; // even number -> exit from the loop
   }
   Console.WriteLine("The number is not even.");
}
Console.WriteLine("Even number entered: {0}", n);
```

We can catch **errors** during the program execution:

```
try
{
   Console.Write("Enter even number: ");
   n = int.Parse(Console.ReadLine());
}
catch
   Console.WriteLine("Invalid number.");
}
// If int.Parse(…) fails, the catch { … } block will execute
```

Now, let's work on a few exercises to practice the new loop types, learned recently.

Problem: Fibonacci Numbers

Fibonacci's numbers in mathematics form a sequence that looks like this: 1, 1, 2, 3, 5, 8, 13, 21, 34, ….

The formula to form the Fibonacci sequence is:

```
F0 = 1
F1 = 1
Fn = Fn-1 + Fn-2
```

Sample Input and Output

Input (n)	Output	Comment
10	89	F(11) = F(9) + F(8)
5	8	F(5) = F(4) + F(3)
20	10946	F(20) = F(19) + F(18)
0	1	By definition
1	1	By definition

Enter an integer number n and calculate the n-number of Fibonacci.

Video: Fibonacci Numbers

Watch this video lesson to learn how to calculate the Fibonacci numbers using a **`for`** loop: https://youtu.be/1ZR0ZBFzB3c.

Hints and Guidelines

An idea to solve the problem:

- We create a variable `n` to which we assign an integer value from the console input.
- We create the variables `f0` and `f1` to which we assign a value of `1`, since the sequence starts.
- We create a `for` loop with condition the current value `i < n - 1`.

- In **the body of the loop**:
  - We create **a temporary** variable `fNext`, to which we assign the next number in the Fibonacci sequence.
  - To `f0` we assign the current value of `f1`.
  - To `f1` we assign the value of the temporary variable `fNext`.
- Out of the loop we print the nth number of Fibonacci.

Example implementation:

```
var n = int.Parse(Console.ReadLine());
var f0 = 1;
var f1 = 1;
for (var i = 0; i < n - 1; i++)
{
    var fNext = f0 + f1;
    f0 = f1;
    f1 = fNext;
}
Console.WriteLine(f1);
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#13.

Problem: Numbers Pyramid

Print **the numbers 1 … n in a pyramid** as in the examples below. On the first line we print one number, on the second line we print two numbers, on the third line we print three numbers, and so on, until the numbers are over. On the last line we print as many numbers as we get until we get to **n**.

Sample Input and Output

Input	Output	Input	Output	Input	Output
7	1 2 3 4 5 6 7	5	1 2 3 4 5	10	1 2 3 4 5 6 7 8 9 10

Video: Pyramid of Numbers

Watch this video lesson to learn how to draw a pyramid of numbers using nested loops and the `break` operator: https://youtu.be/SWU-gQa31QI.

Hints and Guidelines

We can solve the problem with two nested loops (by rows and columns) with printing in them and leaving when the last number is reached. Here is the idea, written in more details:

- We create a variable `n`, to which we assign an integer value from the console input.
- We create a variable `num` with an initial value of 1. It will keep the number of printed numbers. At each iteration we will increase it by 1 and print it.
- We create an outer `for` loop that will be responsible for the rows in the table. We name the variable of the loop `row` and set an initial value of 0. For condition, we set `row < n`. The size of the step is 1.
- In the body of the loop we create an inner `for` loop that will be responsible for the columns in the table. We name the variable of the loop `col` and set an initial value of 0. For a condition, we set `col < row` (`row` = number of digits per line). The size of the step is 1.
- In the body of the nested loop:
  - We check if `col > 1`, if yes –> we print space. If we do not do this, but directly print the space, we will have an unnecessary one at the beginning of each line.
  - We print the number `num` in the current cell of the table and increase it by 1.
  - We are checking for `num > n`. If `num` is greater than `n`, we interrupt the running of the inner loop.
- We print a blank line to move to the next one.
- Again, we check if `num > n`. If it is greater, we interrupt our program by `break`.

Implementation of the Idea

Here is an example implementation:

```
var n = int.Parse(Console.ReadLine());
var num = 1;
for (var row = 1; row <= n; row++)
{
    for (var col = 1; col <= row; col++)
    {
        if (col > 1)
        {
            Console.Write(" ");
        }
        Console.Write(num);
        num++;
        if (num > n)
        {
            break;
        }
    }
    Console.WriteLine();
    if (num > n)
    {
        break;
    }
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#14.

Problem: Numbers Table

Print the numbers 1 … n in a table as in the examples below.

Sample Input and Output

Input	Output	Input	Output
3	1 2 3 2 3 2 3 2 1	4	1 2 3 4 2 3 4 3 3 4 3 2 4 3 2 1

Video: Table with Numbers

Watch this video lesson to learn how to print a table of numbers like the shown above using nested loops: https://youtu.be/DVf7riptCwA.

Hints and Guidelines

We can solve the problem using two nested loops and little calculations inside them:

- We read from the console the table size in an integer variable **`n`**.
- We create a **`for`** loop that will be responsible for the rows in the table. We name the loop variable **`row`** and set assign it an initial value of 0. As a condition, we set **`row < n`**. The size of the step is 1.
- In the body of the loop we create a nested **`for`** loop that will be responsible for the columns in the table. We name the loop variable **`col`** and assign it an initial value of 0. As a condition, we set **`col < n`**. The size of the step is 1.
- In the body of the nested loop:
    - We create a **`num`** variable to which we assign the result of the current row + the current column + 1 (+1 as we start the count from 0).
    - We check for **`num > n`**. If **`num`** is greater than n, we assign a new value to **`num`** which is equal to two times n – the current value for **`num`**. We do this in order not to exceed **`n`** in any of the cells in the table.
    - We print the number from the current table cell.
- We print a blank line in the outer loop to move to the next row.

Implementation of the Idea

Here is a sample implementation of the described idea:

```
var n = int.Parse(Console.ReadLine());
for (int row = 0; row < n; row++)
{
    for (int col = 0; col < n; col++)
    {
        var num = row + col + 1;
        if (num > n)
        {
            num = 2 * n - num;
        }
        Console.Write(num + " ");
    }
    Console.WriteLine();
}
```

Testing in the Judge System

Test your solution here: https://judge.softuni.org/Contests/Practice/Index/514#15.

Lab: Web Application with Complex Loops

Now we know how to repeat a group of actions using **loops**. Let's do something interesting: **a web-based game**. Yes, a real game, with graphics and game logic. Let's have fun. It will be complicated, but if you do not understand how it works, there is no problem. We are now getting into programming. You will advance with coding and with the software technologies over the time. For now, just follow the steps.

Problem: Web Application "Fruits Game"

Description: Develop an **ASP.NET MVC Web Application** – a game in which the player **shoots fruits**, arranged in a table. Successfully hit fruits disappear and the player gets points for each target fruit. When you hit a **dynamite**, the fruits explode and the game ends (as in Fruit Ninja).

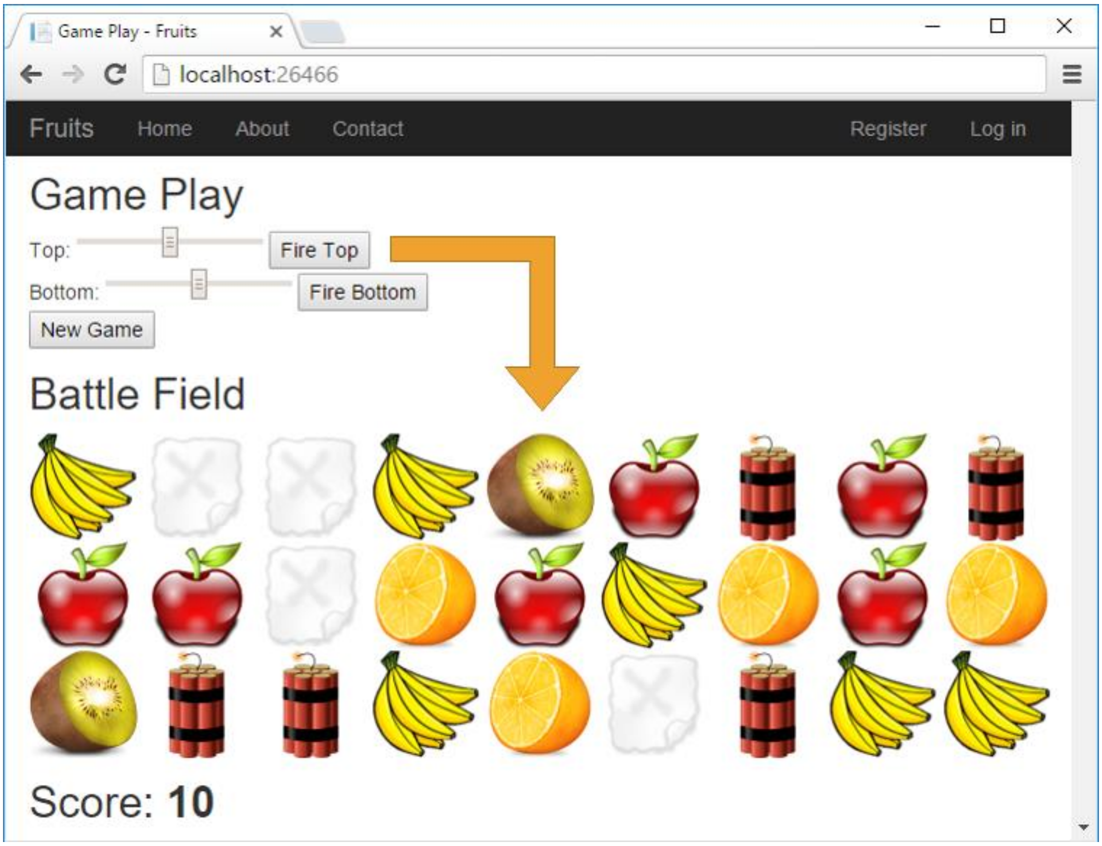
Video: Fruits Game – ASP.NET MVC Web App

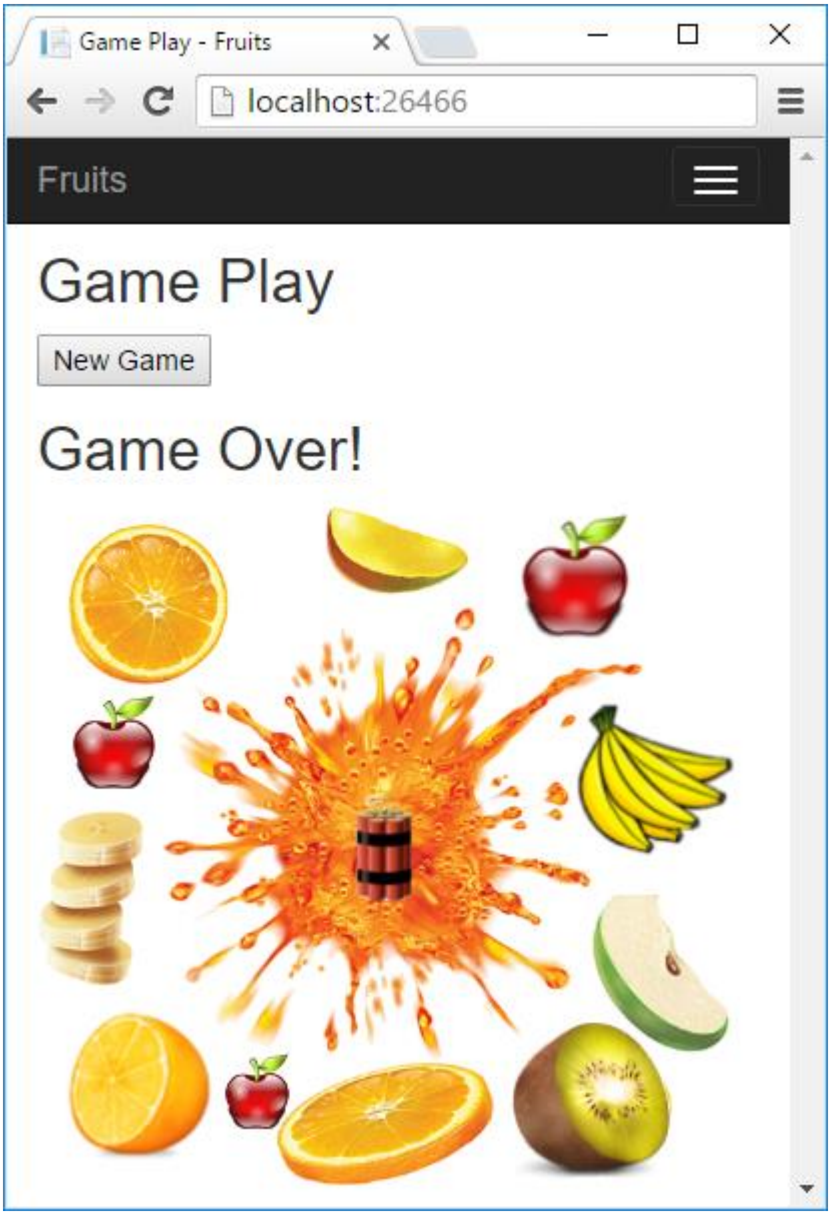
Watch this video lesson to learn how to build an ASP.NET MVC Web application "Fruits Game": https://youtu.be/inCr6SpHWC0.

Fruits Game Explained

Shooting is done by columns, top to bottom or bottom to top, and the location of impact (the column under fire) is set by scroll bar. Because of the inaccuracy of the scroller, the player is not quite sure which column they are going to shoot. Thus, every shot has a chance not to hit and this makes the game more interesting (like the sling in Angry Birds).

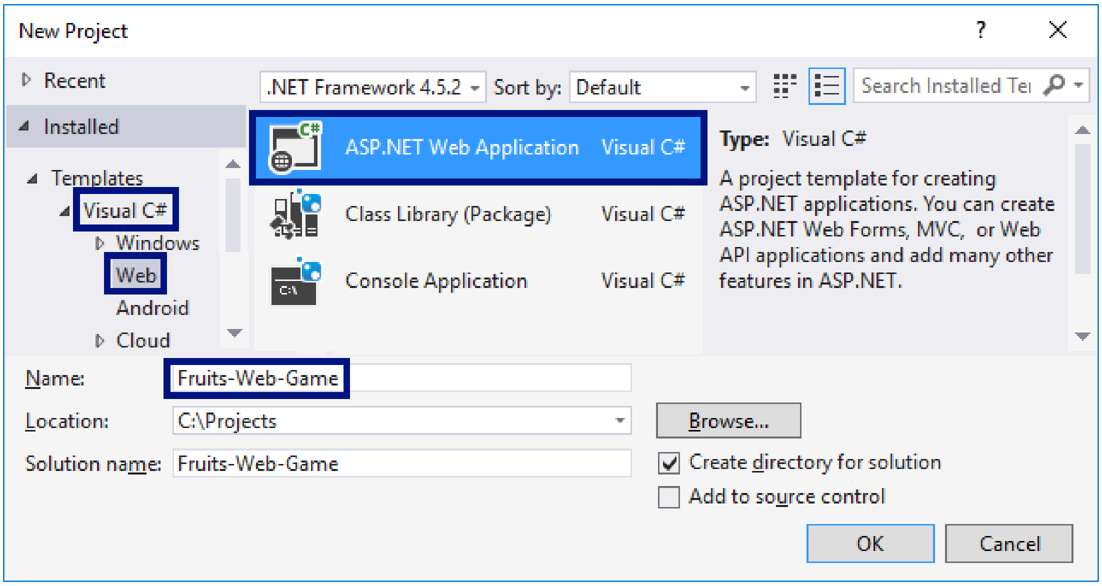
Our game should look like this:


Game Play - Fruits
localhost:26466
Fruits
Home
About
Contact
Register
Log in
Game Play
Top:
Fire Top
Bottom:
Fire Bottom
New Game
Battle Field
Score: 10



Create New C# Project

In Visual Studio, we create a new ASP.NET MVC web application with C# language. Add a new project from [File] → [New] → [Project…]. We give it a meaningful name, for example "Fruits-Web-Game":


New Project

Recent
Installed
Templates
Visual C#
Windows
Web
Android
Cloud

.NET Framework 4.5.2 Sort by: Default

ASP.NET Web Application Visual C#
Class Library (Package) Visual C#
Console Application Visual C#

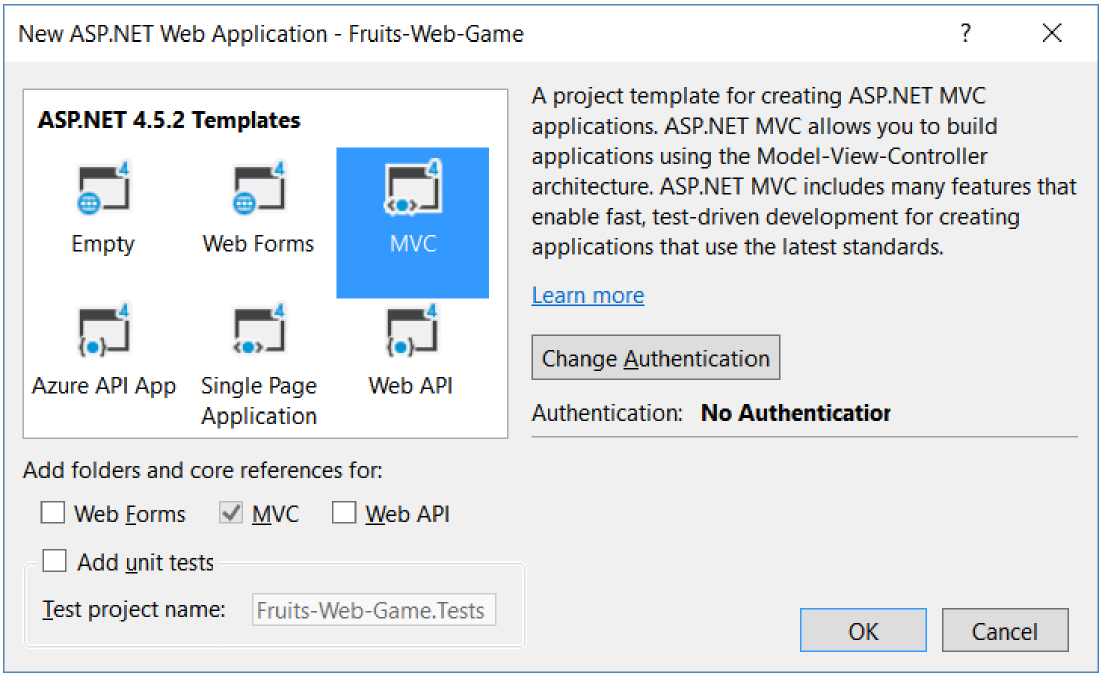
Type: Visual C#

A project template for creating ASP.NET applications. You can create ASP.NET Web Forms, MVC, or Web API applications and add many other features in ASP.NET.

Name: Fruits-Web-Game
Location: C:\Projects
Solution name: Fruits-Web-Game

☑ Create directory for solution
☐ Add to source control

Then we choose the type of web app "MVC":


New ASP.NET Web Application - Fruits-Web-Game

ASP.NET 4.5.2 Templates

Empty, Web Forms, MVC, Azure API App, Single Page Application, Web API

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Learn more

Authentication: **No Authenticatior**

Add folders and core references for:

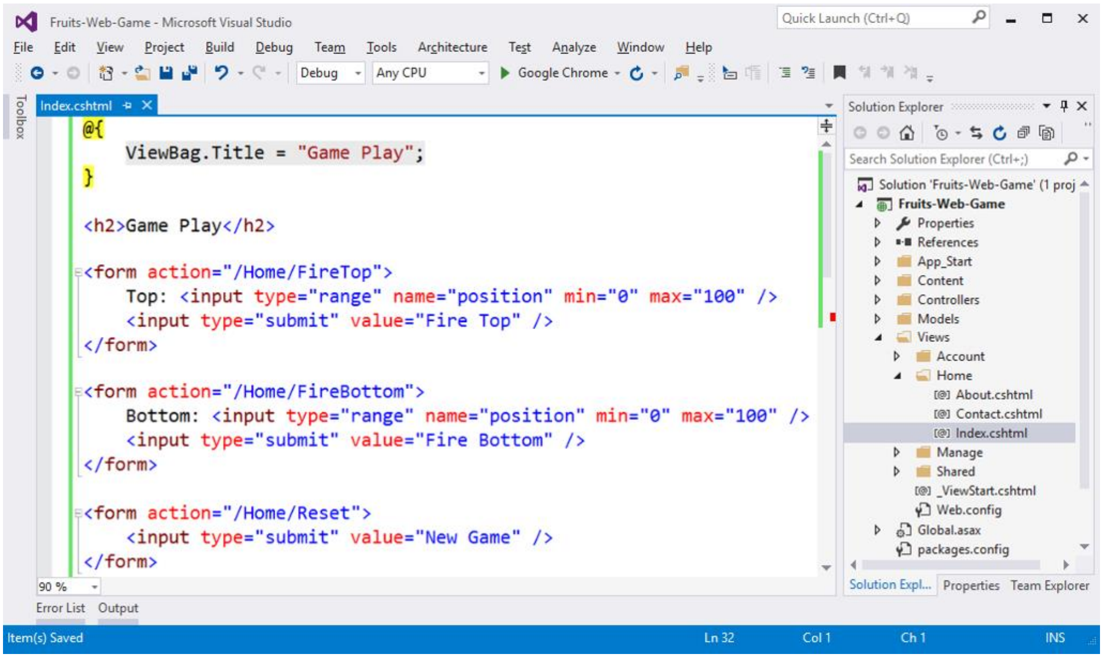
☐ Web Forms ☑ MVC ☐ Web API

☐ Add unit tests

Test project name: Fruits-Web-Game.Tests

Create Controls

Now we will create the controls for the game. The goal is to add scrolling bars by which the player is targeting, and a button for starting a new game. We need to edit the file `Views/Home/Index.cshtml`. We delete everything in it and write the code from the picture:

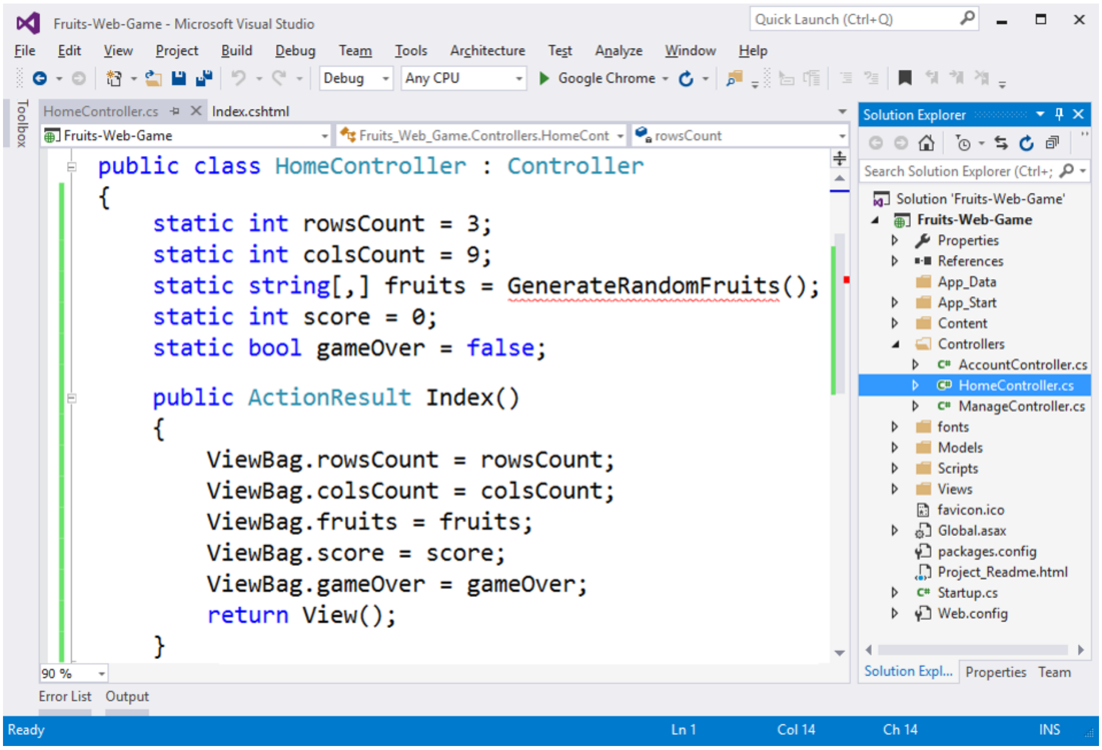


```
@{
    ViewBag.Title = "Game Play";
}

<h2>Game Play</h2>

<form action="/Home/FireTop">
    Top: <input type="range" name="position" min="0" max="100" />
    <input type="submit" value="Fire Top" />
</form>

<form action="/Home/FireBottom">
    Bottom: <input type="range" name="position" min="0" max="100" />
    <input type="submit" value="Fire Bottom" />
</form>

<form action="/Home/Reset">
    <input type="submit" value="New Game" />
</form>
```

This code creates an HTML form `<form>` with a scroller `position` for setting a number in the range [0 … 100] and a button [Fire Top] for sending the form data to the server. The action that will process the data is called `Home/FireTop`, which means `FireTop` method in the `Home` controller, which is located in the file `HomeController.cs`. There are two similar forms with the [Fire Bottom] and [New Game] buttons.

Prepare Fruits for the View

Now we have to prepare the fruits for drawing in the view. Add the following code to the controller: `Controllers/HomeController.cs`:


Fruits-Web-Game - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Debug Any CPU Google Chrome
HomeController.cs Index.cshtml
Fruits-Web-Game Fruits_Web_Game.Controllers.HomeCont rowsCount
Solution Explorer
Solution 'Fruits-Web-Game'
Fruits-Web-Game
Properties
References
App_Data
App_Start
Content
Controllers
AccountController.cs
HomeController.cs
ManageController.cs
fonts
Models
Scripts
Views
favicon.ico
Global.asax
packages.config
Project_Readme.html
Startup.cs
Web.config
Solution Expl... Properties Team
90 %
Error List Output
Ready Ln 1 Col 14 Ch 14 INS

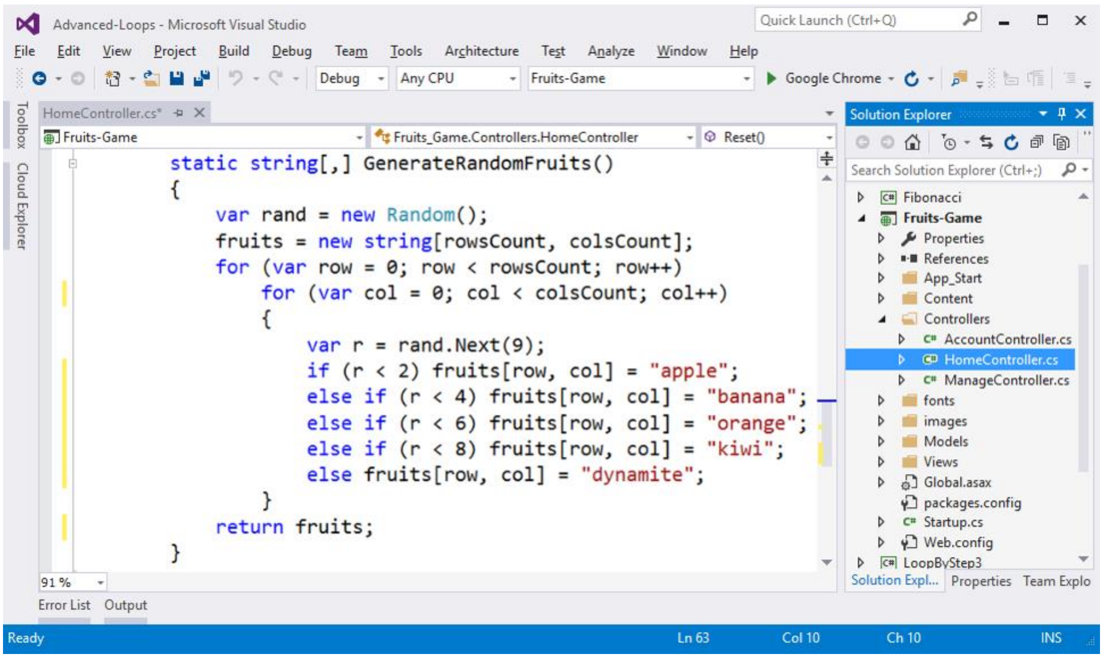
```csharp
public class HomeController : Controller
{
    static int rowsCount = 3;
    static int colsCount = 9;
    static string[,] fruits = GenerateRandomFruits();
    static int score = 0;
    static bool gameOver = false;

    public ActionResult Index()
    {
        ViewBag.rowsCount = rowsCount;
        ViewBag.colsCount = colsCount;
        ViewBag.fruits = fruits;
        ViewBag.score = score;
        ViewBag.gameOver = gameOver;
        return View();
    }
```

The above code defines the fields for number of rows, number of columns, fruit table (playing field), points accumulated by the player and information whether the game is active or ended (field `gameOver`). The playing field has 9 columns in 3 rows and contains for each field a text stating what is inside it: `apple`, `banana`, `orange`, `kiwi`, `empty` or `dynamite`. The main action `Index ()` prepares the game field by recording in the `ViewBag` the structure of the game elements and invoking the view that draws them into the game page of the web browser as HTML.

Generate Random Fruits

We need to generate random fruits. To do this, we need to write a `GenerateRandomFruits()` method with the code from the image below. This code records in the matrix `fruits` names of different images and thus builds the playing field. Each cell of the table records one of the following values: `apple`, `banana`, `orange`, `kiwi`, `empty` or `dynamite`. Next, to draw the corresponding image in the view, the text of the table will be merged with `.png` and this will give the name of the picture file that has to be inserted into the HTML page as part of the playing field. Filling in the playing field (9 columns with 3 rows) happens in the view `Index.cshtml` with two nested `for` loops (for row and column).

In order to generate random fruit for each cell, a random number is generated between 0 and 8 (see the class `Random` in .NET). If the number is 0 or 1, we place `apple`, if it is between 2 and 3, we place `banana` and so on. If the number is 8, we place `dynamite`. Obviously, the fruits appear twice as often as the dynamite. Here's the code:



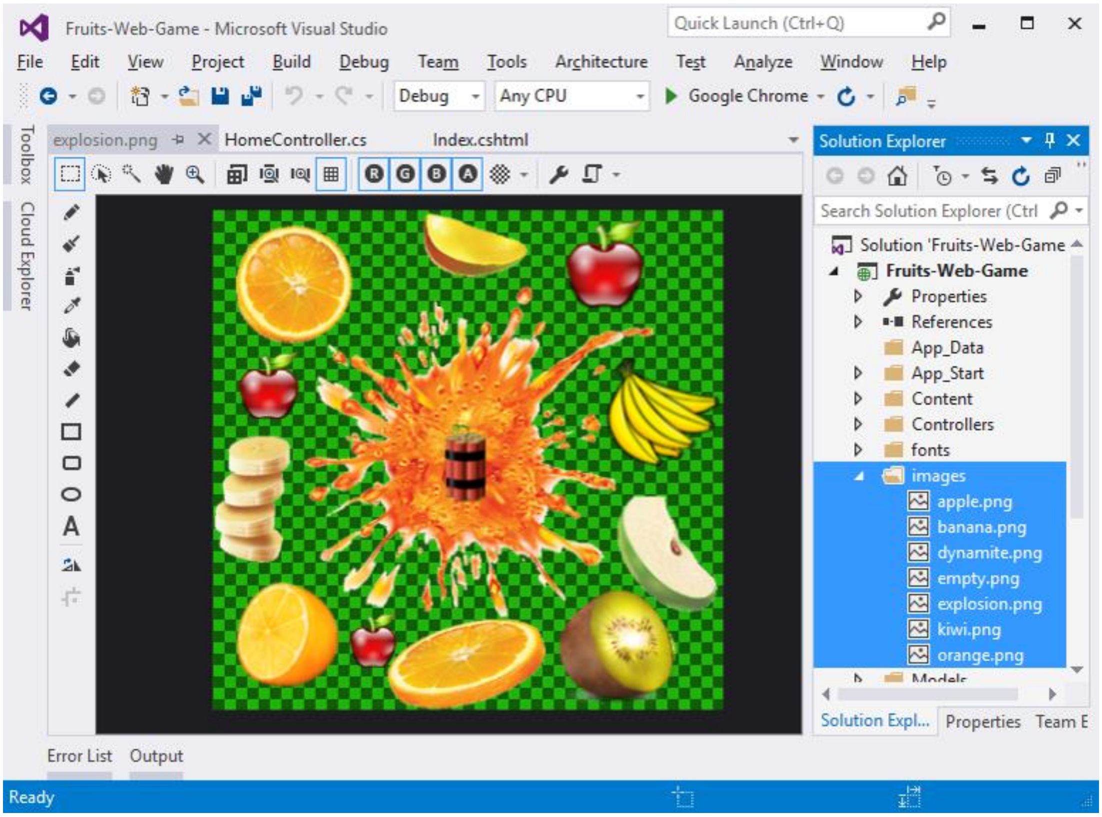
```
static string[,] GenerateRandomFruits()
{
    var rand = new Random();
    fruits = new string[rowsCount, colsCount];
    for (var row = 0; row < rowsCount; row++)
        for (var col = 0; col < colsCount; col++)
        {
            var r = rand.Next(9);
            if (r < 2) fruits[row, col] = "apple";
            else if (r < 4) fruits[row, col] = "banana";
            else if (r < 6) fruits[row, col] = "orange";
            else if (r < 8) fruits[row, col] = "kiwi";
            else fruits[row, col] = "dynamite";
        }
    return fruits;
}
```

Add Game Images

The next thing is to **add the images** for the game.

From [**Solution Explorer**] create folder **images** in the root directory of the project. We use the menu [**Add**] → [**New Folder**].

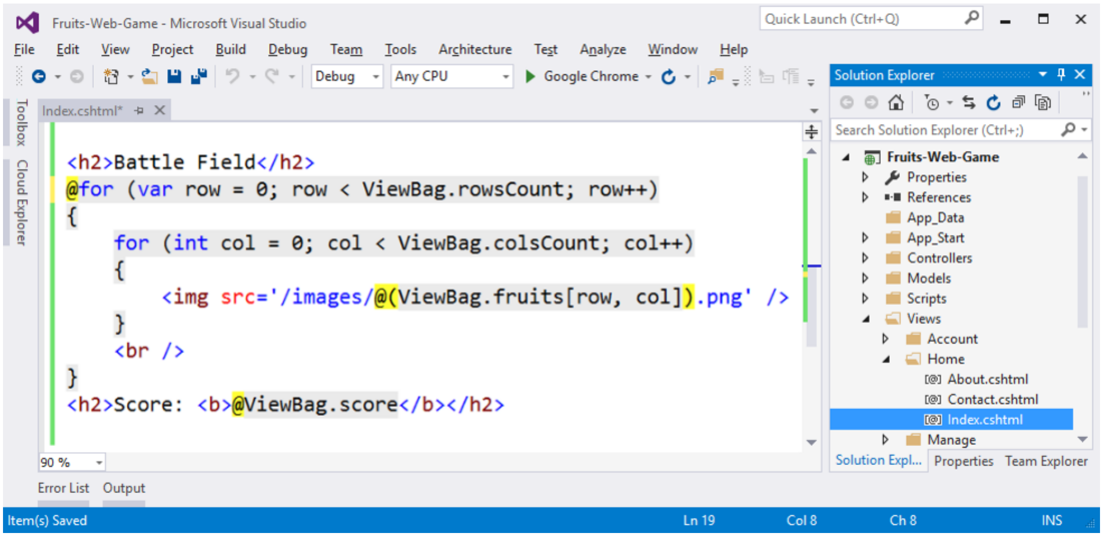
Now we add the **game images** (they are part of the project files for this project and can be downloaded from the book's GitHub repo: https://github.com/SoftUni/Programming-Basics-Book-CSharp-EN/tree/master/assets/chapter-7-assets). We copy them from Windows Explorer and put them in the **images** folder in [**Solution Explorer**] in Visual Studio with **copy/paste**.



Visualize Fruits

Drawing Fruits in `Index.cshtml`:

In order to **draw the playing field** with the fruits, we need to rotate **two nested loops** (for rows and columns). Each row consists of 9 images, each of which contains an `apple`, `banana` or other fruit, or empty `empty`, or `dynamite`. Images are drawn by printing an HTML tag to insert a picture of the type `<img src = "/images/apple.png" />`. Nine pictures are stacked one after the other on each row, followed by a new line with a `<br>`. This is repeated three times for the three lines. Finally, the player's points are printed. Here is what **the code** for drawing the playing field and points looks like:

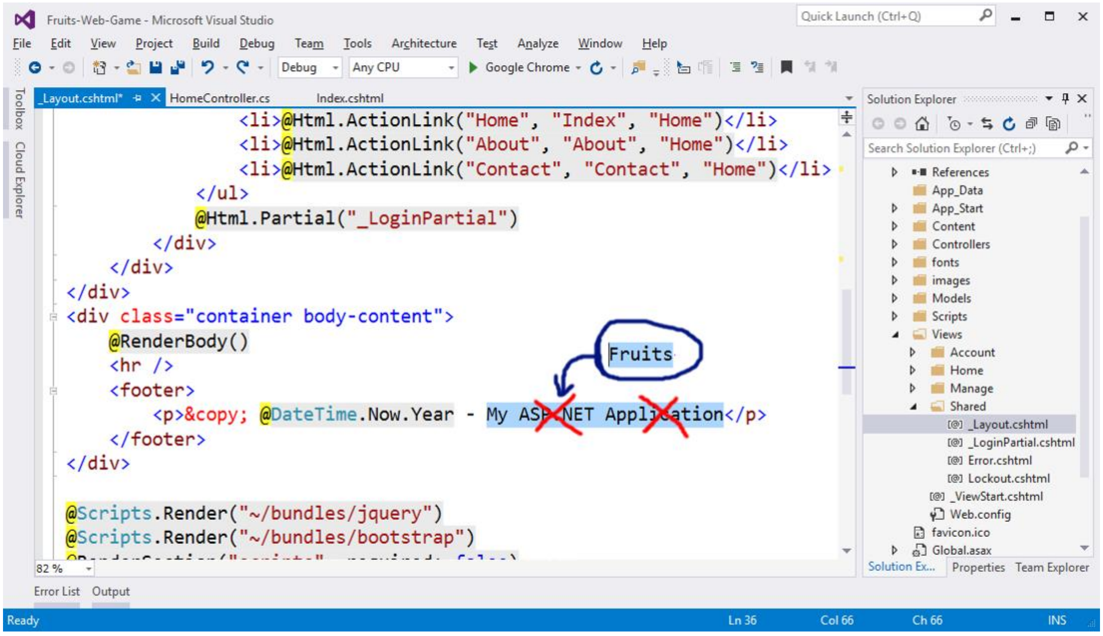


```
<h2>Battle Field</h2>
@for (var row = 0; row < ViewBag.rowsCount; row++)
{
    for (int col = 0; col < ViewBag.colsCount; col++)
    {
        <img src='/images/@(ViewBag.fruits[row, col]).png' />
    }
    <br />
}
<h2>Score: <b>@ViewBag.score</b></h2>
```

Take a look at the yellow characters **@** – they are used to switch between the C# and HTML languages and come from the Razor syntax for drawing dynamic web pages.

Change Text in Layout

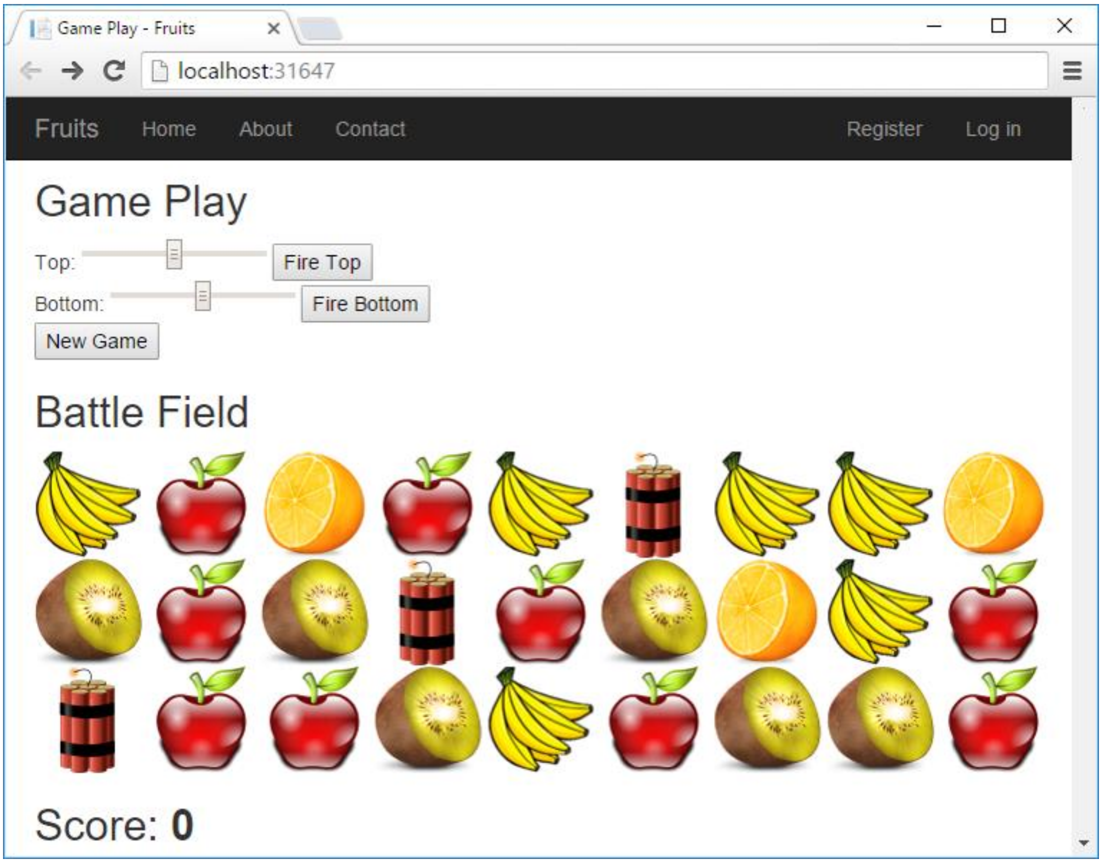
We need to adjust the texts in the `/Views/Shared/_Layout.cshtml` file. We replace `My ASP.NET Application` with more appropriate text, e.g. `Fruits`:



```
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @Html.Partial("_LoginPartial")
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
```

Test the Application

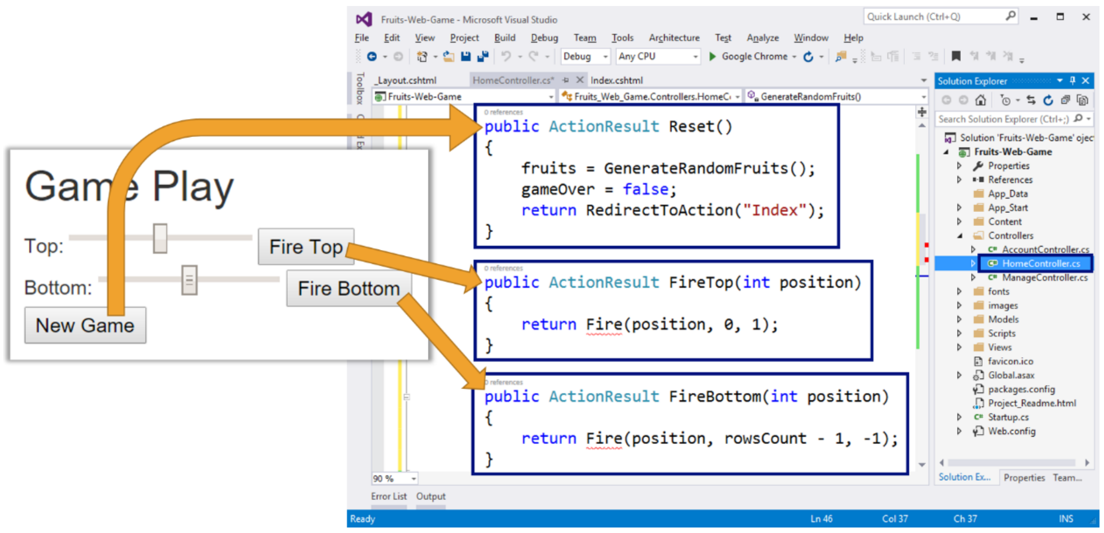
Start the project using [Ctrl + F5] and enjoy it. It is expected to generate a random 9-to-3 playing field with fruits and visualize it on the web page through a series of pictures:



Now the game is sort of done: the playing field is randomly generated and rendered successfully (if you have not made a mistake somewhere). What remains is to fulfill the essence of the game: shooting the fruits.

Shooting the Fruits

For the fruit shooting, we need to add the actions [Reset] and [Fire Top] / [Fire Bottom] to the controller `HomeController.cs`:



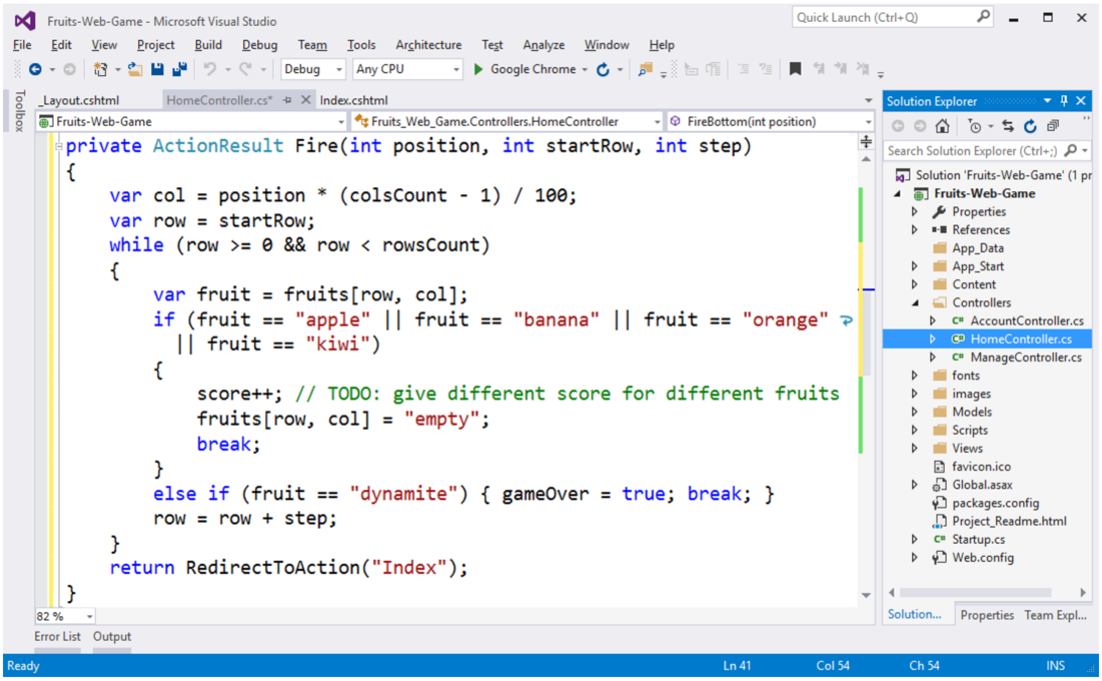
```
public ActionResult Reset()
{
    fruits = GenerateRandomFruits();
    gameOver = false;
    return RedirectToAction("Index");
}

public ActionResult FireTop(int position)
{
    return Fire(position, 0, 1);
}

public ActionResult FireBottom(int position)
{
    return Fire(position, rowsCount - 1, -1);
}
```

The above code defines three actions:

- `Reset()` – starts a new game by generating a new random playing field with fruits and explosives, resetting the player's points and making the game valid (`gameOver = false`). This action is pretty simple and can be immediately tested using [Ctrl + F5] before writing the other ones.
- `FireTop (position)` – shoots on row 0 at position `position` (number 0 to 100). The shooting is in direction down (+1) from row `0`(top). Shooting itself is more complicated as a logic and will be considered after a while.
- `FireBottom (position)` – shoots on row 2 at position `position` (number 0 to 100). The shooting is in direction up (-1) from row 2(bottom).

Implement the "Fire" Method

We implement the "firing" method `Fire (position, startRow, step)`:



```
private ActionResult Fire(int position, int startRow, int step)
{
    var col = position * (colsCount - 1) / 100;
    var row = startRow;
    while (row >= 0 && row < rowsCount)
    {
        var fruit = fruits[row, col];
        if (fruit == "apple" || fruit == "banana" || fruit == "orange"
          || fruit == "kiwi")
        {
            score++; // TODO: give different score for different fruits
            fruits[row, col] = "empty";
            break;
        }
        else if (fruit == "dynamite") { gameOver = true; break; }
        row = row + step;
    }
    return RedirectToAction("Index");
}
```

Shooting works like this: first calculate the column number `col` to which the player has targeted. The input number from the scroll bar (between 0 and 100) is reduced to a number between 0 and 8 (for each of the 9 columns). Line number `row` is either 0 (if the shot is on top) or the number of lines minus one (if the shot is below). Accordingly, the shooting direction (step) is 1 (down) or - 1 (upwards).

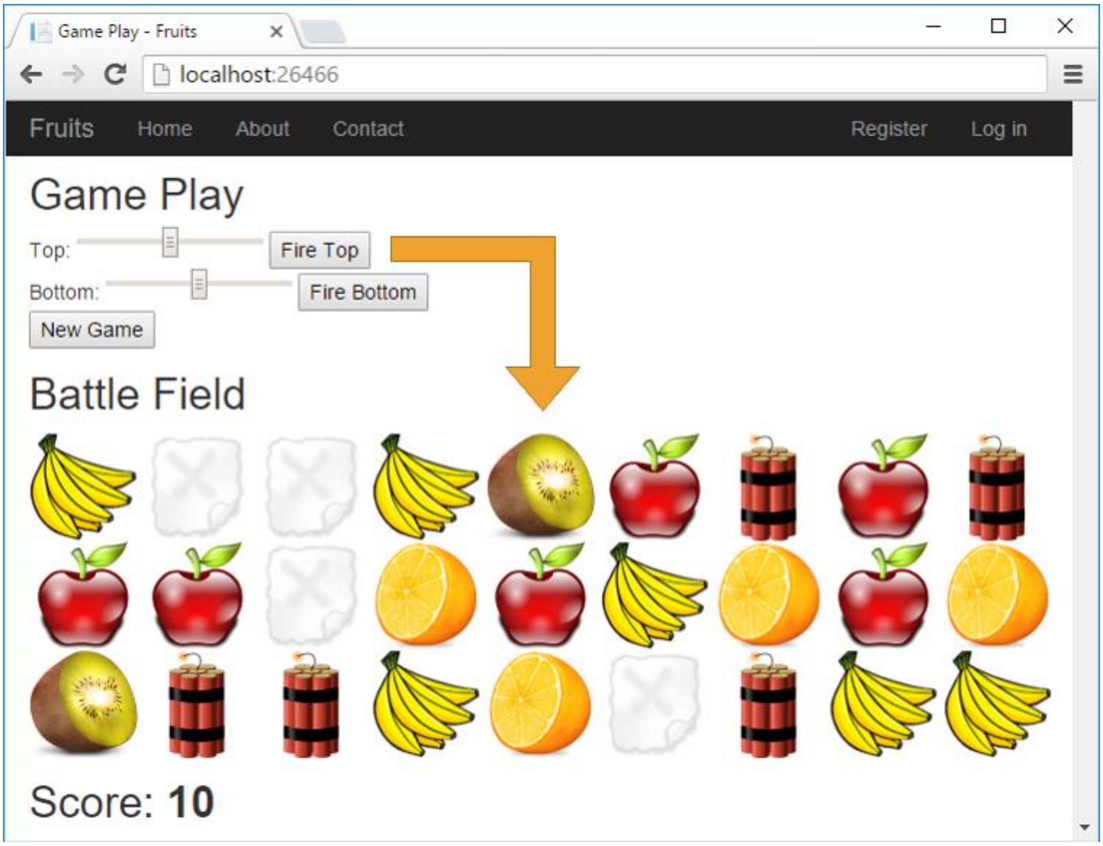
In order to find where the shot hits fruit or dynamite, go through a loop through all the cells in the playing field in the target column and from the first to the last attack row. If a fruit is hit, it disappears (replaced by `empty`) and points are given to the player. If the `dynamite` is hit, the game is marked as finished.

The more enthusiastic among you can implement a more complex behavior, for example, to give different points in the pursuit of a different fruit, to carry out animation with an explosion (this is not too easy), to take points in unnecessary firing in an empty column and so on.

Test the Application Again

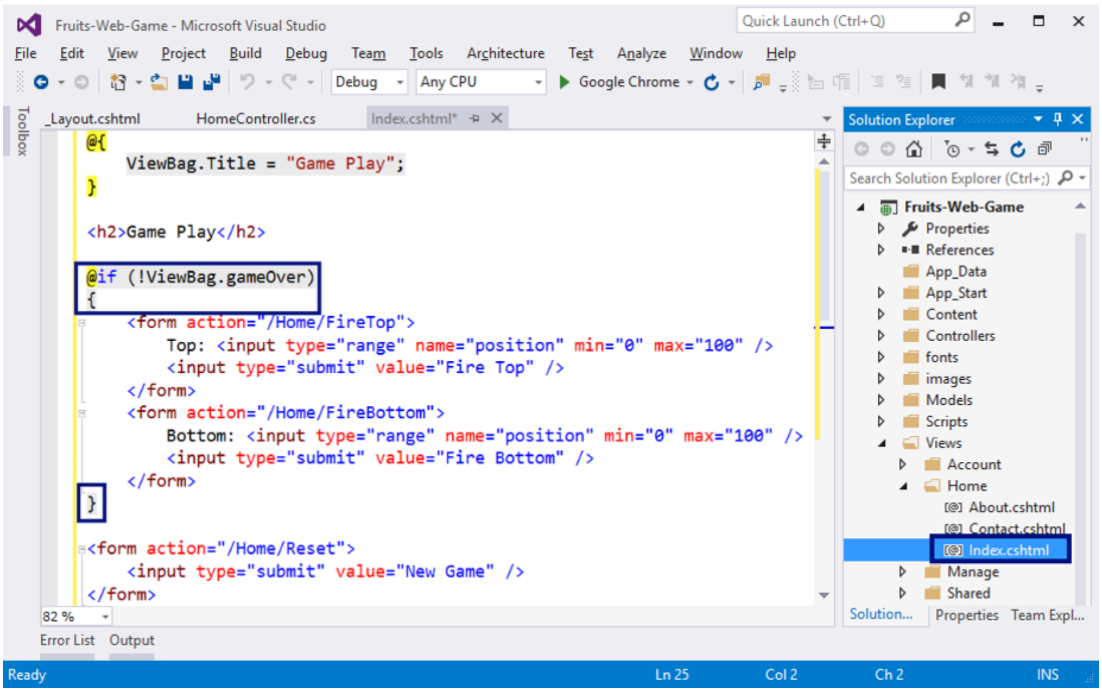
We are testing what created up until now by starting with [Ctrl + F5]:

- New Game → the new game button must generate a new playing field with randomly placed fruits and explosives and reset the player's points.
- Shooting from top → the top firing must remove the top fruit in the hit column or cause the game to end if there is dynamite. In fact, at the end of the game nothing is going to happen, because in the view this case is still not considered.
- Shooting from bottom → the shooting from bottom should remove the lowest fruit in the hit column or end the game when you hit the dynamite.

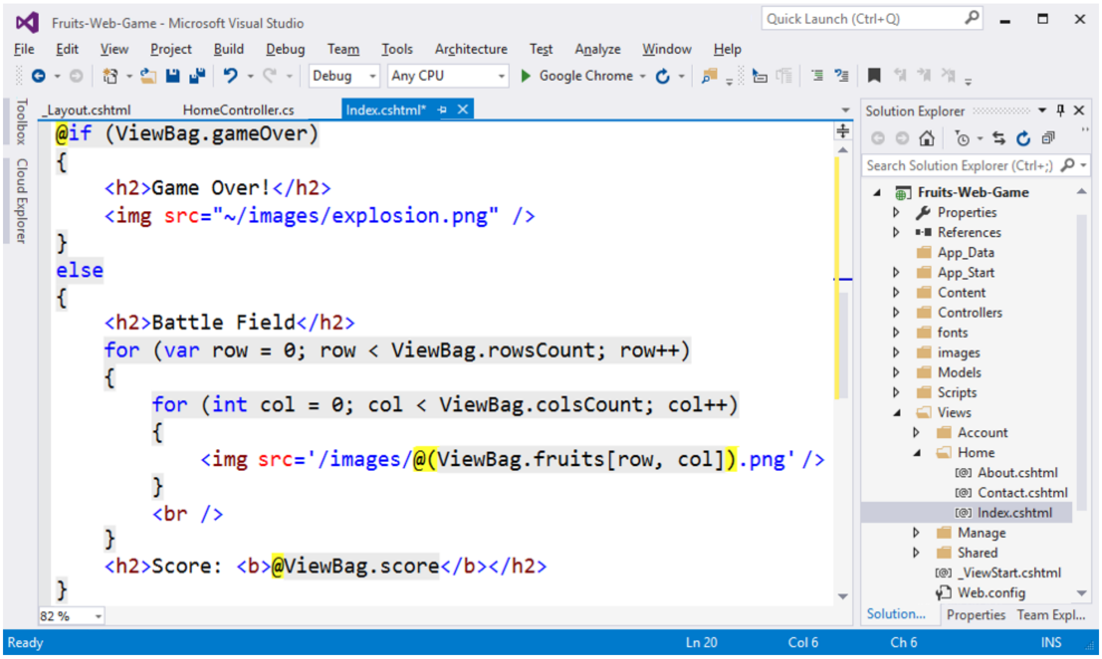


Implement "Game Over"

For now, at "End of the game" nothing happens. If a player reaches a dynamite, the controller says that the game is over (`gameOver = true`), but this fact is not visualized in any way. In order for the game to finish, we need to add several checks in the view:



```
@{
    ViewBag.Title = "Game Play";
}

<h2>Game Play</h2>

@if (!ViewBag.gameOver)
{
    <form action="/Home/FireTop">
        Top: <input type="range" name="position" min="0" max="100" />
        <input type="submit" value="Fire Top" />
    </form>
    <form action="/Home/FireBottom">
        Bottom: <input type="range" name="position" min="0" max="100" />
        <input type="submit" value="Fire Bottom" />
    </form>
}

<form action="/Home/Reset">
    <input type="submit" value="New Game" />
</form>
```

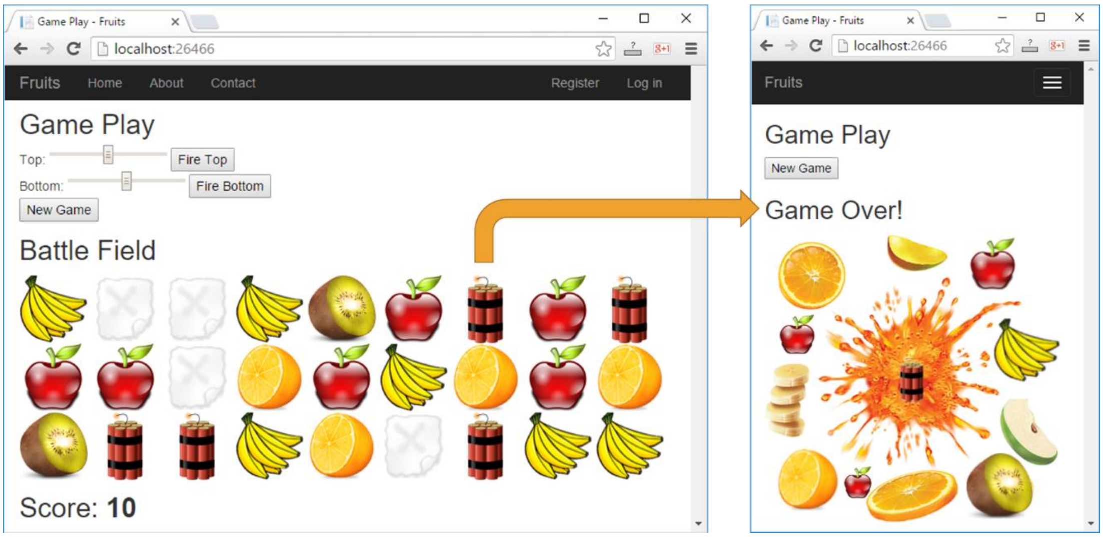


```
@if (ViewBag.gameOver)
{
    <h2>Game Over!</h2>
    <img src="~/images/explosion.png" />
}
else
{
    <h2>Battle Field</h2>
    for (var row = 0; row < ViewBag.rowsCount; row++)
    {
        for (int col = 0; col < ViewBag.colsCount; col++)
        {
            <img src='/images/@(ViewBag.fruits[row, col]).png' />
        }
        <br />
    }
    <h2>Score: <b>@ViewBag.score</b></h2>
}
```

The code above checks whether the game has finished and indicates accordingly the shooting controls and the playing field (active game) or exploding fruit picture at the end of the game.

Final Testing of the Application

After changing the code in the view, let's start by [Ctrl + F5] and test the game again:



This time, when you hit a dynamite, the right picture should appear and allow only the "new game" action (the [**New Game**] button).

Was it complicated? Did you manage to create the game? If you have not succeeded, relax, this is a relatively complex project that includes a great deal of non-studied matter.

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!