Step 1: Intro

This is a .NET Console application written in C#. Select **Run Code** to try it out.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace myApp
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

o/p-

```
Hello Praneeth!
```

Step 2: Strings

Try modifying the code so that the console says hello to your name, instead of the world (for example, `Hello Ana!`). Select **Run Code** to see if you got it right.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace myApp
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Hello Praneeth!");
        }
    }
}
```

```
Hello Praneeth!
```

Step 3: Variables

Variables hold values that you can use elsewhere in your code.

Let's store your name in a variable, then read the value from that variable when creating the output message.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace myApp
{
    class Program
    {
        static void Main()
        {
            var name="Praneeth";
            Console.WriteLine("Hello" + name + "!");
        }
    }
}
```

```
o/p-Hello Praneeth!
```

Step 4: String interpolation

String interpolation lets you piece together strings in a more concise and readable way.

If you add a `$` before the opening quotes of the string, you can then include string values, like the `name` variable, inside the string in curly brackets. Try it out and select **Run Code**.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace myApp
{
    class Program
    {
        static void Main()
        {
            var name="Praneeth";
          Console.WriteLine($"Hello {name}!");

        }
    }
}
```

```
HelloPraneeth!
```

Step 5: Methods

Methods take inputs, do some work, and sometimes return a result.

`ToUpper()` is a method you can invoke on a string, like the `name` variable. It will return the same string, converted to uppercase.

Update the greeting to change the name of the person being greeted to uppercase, and select **Run Code**.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace myApp
{
    class Program
    {
        static void Main()
        {
            var name="Praneeth";
          Console.WriteLine($"Hello {name.ToUpper()}!");

        }
    }
}
```

```
Hello PRANEETH!
```

Step 6: Collections

Collections hold multiple values of the same type.

Replace the **name** variable with a **names** variable that has a list of names. Then use a **foreach** loop to iterate over all the names and say hello to each person.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace myApp
{
    class Program
    {
        static void Main()
        {
           var names = new List<string> { "Praneeth", "Felipe", "Emillia" };
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
        }
    }
}
```

```
Hello PRANEETH!
Hello FELIPE!
Hello EMILLIA!
```

Congratulations, You Finished!

Now that you've got the basics, you can keep learning with the Introduction to C# tutorials.

In the first tutorial you'll learn about working with numbers.

Explore integer math

- 30 minutes remaining

Run the following code in the interactive window. Select the **Enter focus mode** button. Then, type the following code block in the interactive window and select **Run**:

environment, you should follow the instructions for the local version instead.

You've seen one of the fundamental math operations with integers. The `int` type represents an **integer**, a positive or negative whole number. You use the `+` symbol for addition. Other common mathematical operations for integers include:

- `-` for subtraction
- `*` for multiplication
- `/` for division

Addition-

```
      int a = 18;

int b = 6;
int c = a + b;
Console.WriteLine(c);
```

o/p-24

Subtraction-

```
int a = 18;
int b = 6;
int c = a - b;
Console.WriteLine(c);
```

O/p-12

Multiplication-

```
int a = 18;
int b = 6;
int c = a * b;
Console.WriteLine(c);
```

o/p-108

Division-

```
int a = 18;
int b = 6;
int c = a / b;
Console.WriteLine(c);
```

o/p-3

Step 2-

Explore order of operations

The C# language defines the precedence of different mathematics operations with rules consistent with the rules you learned in mathematics. Multiplication and division take precedence over addition and subtraction. Explore that by running the following code in the interactive window:

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

o/p-13

The output demonstrates that the multiplication is performed before the addition.

You can force a different order of operation by adding parentheses around the operation or operations you want performed first:

```
int a = 5;
int b = 4;
int c = 2;
int d = (a + b) * c;
Console.WriteLine(d);
```

o/p-18

Explore more by combining many different operations. Replace the fourth line above with something like this:

```
int a = 5;
int b = 4;
int c = 2;
int d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
```

o/p-No output

You may have noticed an interesting behavior for integers. Integer division always produces an integer result, even when you'd expect the result to include a decimal or fractional portion.

If you haven't seen this behavior, try the following code:

```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
Console.WriteLine(d);
```

o/p-3

Step-3:

Explore integer precision and limits

- 19 minutes remaining

That last sample showed you that integer division truncates the result. You can get the **remainder** by using the **remainder** operator, the `%` character:

```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

o/p-
quotient: 3
remainder: 2

The C# integer type differs from mathematical integers in one other way: the `int` type has minimum and maximum limits. Run this code in the interactive window to see those limits:

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

o/p-
The range of integers is -2147483648 to 2147483647

If a calculation produces a value that exceeds those limits, you have an **underflow** or **overflow** condition. The answer appears to wrap from one limit to the other. Add these two lines to the interactive window to see an example:

```
int max = int.MaxValue;
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Notice that the answer is very close to the minimum (negative) integer. It's the same as `min + 2`. The addition operation **overflowed** the allowed values for integers. The answer is a very large negative number because an overflow "wraps around" from the largest possible integer value to the smallest.

There are other numeric types with different limits and precision that you would use when the `int` type doesn't meet your needs. Let's explore those types of numbers next.

Step-4:

Work with the double type

- 14 minutes remaining

The `double` numeric type represents a double-precision floating point number. Those terms may be new to you. A **floating point** number is useful to represent non-integral numbers that may be very large or small in magnitude. **Double-precision** is a relative term that describes the numbers of binary digits used to store the value. **Double precision** number have twice the number of binary digits as **single-precision**. On modern computers, it is more common to use double precision than single precision numbers. **Single precision** numbers are declared using the `float` keyword. Let's explore. Try the following code in the interactive window and see the result:

```
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

o/p-4.5

Notice that the answer includes the decimal portion of the quotient. Try a slightly more complicated expression with doubles:

```
double a = 19;
double b = 23;
double c = 8;
double d = (a + b) / c;
Console.WriteLine(d);
```

o/p-5.25

The range of a double value is much greater than integer values. Try the following code in the interactive window:

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

o/p-

```
The range of double is -1.79769313486232E+308 to
1.79769313486232E+308
```

These values are printed out in scientific notation. The number to the left of the `E` is the significand. The number to the right is the exponent, as a power of 10.

Just like decimal numbers in math, doubles in C# can have rounding errors. Try this code:

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

o/p- 0.333333333333333

You know that `0.3` is `3/10` and not exactly the same as `1/3`. Similarly, `0.33` is `33/100`. That's closer to `1/3`, but still not exact.

Challenge

Try other calculations with large numbers, small numbers, multiplication, and division using the `double` type. Try more complicated calculations.

Step-5:

Work with decimal types

- 9 minutes remaining

You've seen the basic numeric types in C#: integers and doubles. There's one other type to learn: the `decimal` type. The `decimal` type has a smaller range but greater precision than `double`. Let's take a look:

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

o/p-The range of the decimal type is -79228162514264337593543950335 to 79228162514264337593543950335

Notice that the range is smaller than the `double` type. You can see the greater precision with the decimal type by trying the following code:

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

o/p-0.333333333333333
0.3333333333333333333333333333

The `M` suffix on the numbers is how you indicate that a constant should use the `decimal` type.

Notice that the math using the decimal type has more digits to the right of the decimal point.

The `M` suffix on the numbers is how you indicate that a constant should use the `decimal` type. Otherwise, the compiler assumes the `double` type.

Note

The letter M was chosen as the most visually distinct letter between the double and decimal keywords.

Challenge

Now that you've seen the different numeric types, write code that calculates the area of a circle whose radius is 2.50 centimeters. Remember that the area of a circle is the radius squared multiplied by PI. One hint: .NET contains a constant for PI, Math.PI that you can use for that value. Math.PI, like all constants declared in the `System.Math` namespace, is a `double` value. For that reason, you should use `double` instead of `decimal` values for this challenge.

You should get an answer between 19 and 20.

Complete challenge

- 1 minutes remaining

Did you come up with something like this?

```
double radius = 2.50;
double area = Math.PI * radius * radius;
Console.WriteLine(area);
```

o/p- 19.6349540849362

Congratulations!

- 100% complete!

You've completed the "Numbers in C#" interactive tutorial. You can select the **Branches and Loops** link below to start the next interactive tutorial, or you can visit the .NET site to download the .NET Core SDK, create a project on your machine, and keep coding. The "Next steps" section brings you back to these tutorials.

You can learn more about numbers in C# in the following articles:

- Integral numeric types
- Floating-point numeric types
- Built-in numeric conversions

Learn conditional logic with branch and loop statements

This tutorial teaches you how to write code that examines variables and changes execution path based on those variables. You'll use your browser to write C# interactively and see the results of compiling and running your code. This tutorial contains a series of lessons that explore branching and looping constructs in C#. These lessons teach you the fundamentals of the C# language.

Make decisions using the if statement

- 48 minutes remaining

Run the following code in the interactive window. Select the **Enter focus mode** button. Then, type the following code block in the interactive window and select

```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

o/p-The answer is greater than 10

If you are running this on your environment, you should follow the instructions for the local version instead.

Modify the declaration of `b` so that the sum is less than 10:

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

o/p-No output

Select the **Run** button again. Because the answer is less than 10, nothing is printed. The **condition** you're testing is false. You don't have any code to execute because you've only written one of the possible branches for an `if` statement: the true branch.

Tip

As you explore C# (or any programming language), you'll make mistakes when you write code. The compiler will find those errors and report them to you. When the output contains error messages, look closely at the example code, and the code in the interactive window to see what to fix. That exercise will help you learn the structure of C# code.

This first sample shows the power of `if` and boolean types. A *boolean* is a variable that can have one of two values: `true` or `false`. C# defines a special type, `bool` for boolean variables. The `if` statement checks the value of a `bool`. When the value is `true`, the statement following the `if` executes. Otherwise, it's skipped.

This process of checking conditions and executing statements based on those conditions is powerful. Let's explore more.

Step-2:

Make if and else work together

- 44 minutes remaining

To execute different code in both the true and false branches, you create an `else` branch that executes when the condition is false. Try this:

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

O/p-The answer is not greater than 10

The statement following the `else` keyword executes only when the condition being tested is `false`. Combining `if` and `else` with boolean conditions provides all the power you need.

Important

The indentation under the `if` and `else` statements is for human readers. The C# language doesn't treat indentation or white space as significant. The statement following the `if` or `else` keyword will be executed based on the condition. All the samples in this tutorial follow a common practice to indent lines based on the control flow of statements.

Because indentation isn't significant, you need to use `{` and `}` to indicate when you want more than one statement to be part of the block that executes conditionally. C# programmers typically use those braces on all `if` and `else` clauses. The following example is the same as what you created. Try it.

```
 int a = 5;

int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

```
o/p-The answer is not greater than 10
```

Tip

Through the rest of this tutorial, the code samples all include the braces, following accepted practices.

You can test more complicated conditions:

```
int a = 5;
int b = 3;
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

o/p-The answer is not greater than 10

The `==` symbol tests for *equality*. Using `==` distinguishes the test for equality from assignment, which you saw in `a = 5`.

The `&&` represents "and". It means both conditions must be true to execute the statement in the true branch. These examples also show that you can have multiple statements in each conditional branch, provided you enclose them in `{` and `}`.

You can also use || to represent "or":

```
int a = 5;
int b = 3;
int c = 4;
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

o/p- The answer is greater than 10
Or the first number is equal to the second

Modify the values of `a`, `b`, and `c` and switch between `&&` and `||` to explore. You'll gain more understanding of how the `&&` and `||` operators work.

Use loops to repeat operations

- 34 minutes remaining

Another important concept to create larger programs is **loops**. You'll use loops to repeat statements that you want executed more than once. Try this code in the interactive window:

```
int counter = 0;
while (counter < 10)
{
  Console.WriteLine($"Hello World! The counter is {counter}");
  counter++;
}
```

o/p- Hello World! The counter is 0
Hello World! The counter is 1
Hello World! The counter is 2
Hello World! The counter is 3
Hello World! The counter is 4
Hello World! The counter is 5
Hello World! The counter is 6
Hello World! The counter is 7
Hello World! The counter is 8
Hello World! The counter is 9

Important

Make sure that the while loop condition does switch to false as you execute the code. Otherwise, you create an infinite loop where your program never ends. Let's not demonstrate that, because the engine that runs your code will time out and you'll see no output from your program.

The `while` loop tests the condition before executing the code following the `while`. The `do` ... `while` loop executes the code first, and then checks the condition. It looks like this:

```
int counter = 0;
do
{
  Console.WriteLine($"Hello World! The counter is {counter}");
  counter++;
} while (counter < 10);
```

o/p- Hello World! The counter is 0
Hello World! The counter is 1
Hello World! The counter is 2
Hello World! The counter is 3
Hello World! The counter is 4
Hello World! The counter is 5
Hello World! The counter is 6
Hello World! The counter is 7
Hello World! The counter is 8
Hello World! The counter is 9

Step-4:

Work with the for loop

- 28 minutes remaining

Another common loop statement that you'll see in C# code is the `for` loop. Try this code in the interactive window:

```
int counter = 0;
do
{
  Console.WriteLine($"Hello World! The counter is {counter}");
  counter++;
} while (counter < 10);
```

o/p-

Hello World! The counter is 0
Hello World! The counter is 1
Hello World! The counter is 2
Hello World! The counter is 3
Hello World! The counter is 4
Hello World! The counter is 5
Hello World! The counter is 6

```
Hello World! The counter is 7
Hello World! The counter is 8
Hello World! The counter is 9
```

This does the same work as the `while` loop and the `do` loop you've already used. The `for` statement has three parts that control how it works.

The first part is the **for initializer**: `int counter = 0;` declares that `counter` is the loop variable, and sets its initial value to `0`.

The middle part is the **for condition**: `counter < 10` declares that this `for` loop continues to execute as long as the value of counter is less than 10.

The final part is the **for iterator**: `counter++` specifies how to modify the loop variable after executing the block following the `for` statement. Here, it specifies that `counter` should be incremented by 1 each time the block executes.

Experiment with these yourself. Try each of the following:

- Change the initializer to start at a different value.
- Change the condition to stop at a different value.

When you're done, let's move on to write some code yourself to use what you've learned.

There's one other looping statement that isn't covered in this tutorial: the `foreach` statement. The `foreach` statement repeats its statement for every item in a sequence of items. It's most often used with *collections*, so it is covered in the next tutorial.

Step-5:

Created nested loops

- 23 minutes remaining

A `while`, `do`, or `for` loop can be nested inside another loop to create a matrix using the combination of each item in the outer loop with each item in the inner loop. Let's do that to build a set of alphanumeric pairs to represent rows and columns.

One `for` loop can generate the rows:

```
for (int row = 1; row < 11; row++)
{
  Console.WriteLine($"The row is {row}");
}
```

o/p-

```
The row is 1
The row is 2
The row is 3
The row is 4
The row is 5
The row is 6
The row is 7
The row is 8
The row is 9
The row is 10
```

Another loop can generate the columns:

```
for (char column = 'a'; column < 'k'; column++)
{
  Console.WriteLine($"The column is {column}");
}
```

o/p-

```
The column is a
The column is b
The column is c
The column is d
The column is e
The column is f
The column is g
The column is h
The column is i
The column is j
```

You can nest one loop inside the other to form pairs:

```
for (int row = 1; row < 11; row++)
{
  for (char column = 'a'; column < 'k'; column++)
  {
    Console.WriteLine($"The cell is ({row}, {column})");
  }
}
```

o/p-

```
The cell is (1, a)
The cell is (1, b)
The cell is (1, c)
The cell is (1, d)
The cell is (1, e)
The cell is (1, f)
The cell is (1, g)
The cell is (1, h)
The cell is (1, i)
The cell is (1, j)
The cell is (2, a)
The cell is (2, b)
The cell is (2, c)
The cell is (2, d)
The cell is (2, e)
The cell is (2, f)
The cell is (2, g)
The cell is (2, h)
The cell is (2, i)
The cell is (2, j)
The cell is (3, a)
The cell is (3, b)
The cell is (3, c)
The cell is (3, d)
The cell is (3, e)
The cell is (3, f)
The cell is (3, g)
The cell is (3, h)
```

The cell is (3, i)
The cell is (3, j)
The cell is (4, a)
The cell is (4, b)
The cell is (4, c)
The cell is (4, d)
The cell is (4, e)
The cell is (4, f)
The cell is (4, g)
The cell is (4, h)
The cell is (4, i)
The cell is (4, j)
The cell is (5, a)
The cell is (5, b)
The cell is (5, c)
The cell is (5, d)
The cell is (5, e)
The cell is (5, f)
The cell is (5, g)
The cell is (5, h)
The cell is (5, i)
The cell is (5, j)
The cell is (6, a)
The cell is (6, b)
The cell is (6, c)
The cell is (6, d)
The cell is (6, e)
The cell is (6, f)
The cell is (6, g)
The cell is (6, h)
The cell is (6, i)
The cell is (6, j)
The cell is (7, a)
The cell is (7, b)
The cell is (7, c)

The cell is (7, d)
The cell is (7, e)
The cell is (7, f)
The cell is (7, g)
The cell is (7, h)
The cell is (7, i)
The cell is (7, j)
The cell is (8, a)
The cell is (8, b)
The cell is (8, c)
The cell is (8, d)
The cell is (8, e)
The cell is (8, f)
The cell is (8, g)
The cell is (8, h)
The cell is (8, i)
The cell is (8, j)
The cell is (9, a)
The cell is (9, b)
The cell is (9, c)
The cell is (9, d)
The cell is (9, e)
The cell is (9, f)
The cell is (9, g)
The cell is (9, h)
The cell is (9, i)
The cell is (9, j)
The cell is (10, a)
The cell is (10, b)
The cell is (10, c)
The cell is (10, d)
The cell is (10, e)
The cell is (10, f)
The cell is (10, g)
The cell is (10, h)

The cell is (10, i)
The cell is (10, j)

You can see that the outer loop increments once for each full run of the inner loop. Reverse the row and column nesting, and see the changes for yourself.

Step-6:

Combine branches and loops

- 13 minutes remaining

Now that you've seen the `if` statement and the looping constructs in the C# language, see if you can write C# code to find the sum of all integers 1 through 20 that are divisible by 3. Here are a few hints:

- The `%` operator gives you the remainder of a division operation.
- The `if` statement gives you the condition to see if a number should be part of the sum.
- The `for` loop can help you repeat a series of steps for all the numbers 1 through 20.

Try it yourself. Then check how you did. As a hint, you should get 63 for an answer.

Step-7:

Complete challenge

- 1 minutes remaining

Did you come up with something like this?

```
int sum = 0;
for (int number = 1; number < 21; number++)
{
  if (number % 3 == 0)
  {
    sum = sum + number;
  }
}
Console.WriteLine($"The sum is {sum}");
```

o/p-The sum is 63

Step-8:

Congratulations!

- 100% complete!

You've completed the "branches and loops" interactive tutorial. You can select the **list collection** link below to start the next interactive tutorial, or you can visit the .NET site to download the .NET Core SDK, create a project on your machine, and keep coding. The "Next steps" section brings you back to these tutorials.

You can learn more about these concepts in these articles:

- If and else statement
- While statement
- Do statement
- For statement