100 XP

Exercise - Perform math operations

10 minutes

Now that you understand the basics of addition and more importantly, implicit type conversion between numeric and string data types, let's look at several other common mathematical operations on numeric data.

Perform basic math operations

Write code to perform addition, subtraction, multiplication, and division with integers

1. Select all of the code in the .NET Editor, and press `Delete` or `Backspace` to delete it.
2. Enter the following code in the .NET Editor:

C#

```csharp
int sum = 7 + 5;
int difference = 7 - 5;
int product = 7 * 5;
int quotient = 7 / 5;

Console.WriteLine("Sum: " + sum);
Console.WriteLine("Difference: " + difference);
Console.WriteLine("Product: " + product);
Console.WriteLine("Quotient: " + quotient);
```

3. Run the code. You should see the following output:

Output

```
Sum: 12
Difference: 2
Product: 35
Quotient: 1
```

As you can see:

- `+` is the addition operator
- `-` is the subtraction operator
- `*` is the multiplication operator
- `/` is the division operator

However, the resulting quotient of the division example may not be what you may have expected. The values after the decimal are truncated from the `quotient` since it is defined as an `int`, and `int` cannot contain values after the decimal.

Add code to perform division using literal decimal data

To see division working properly, you need to use a data type that supports fractional digits after the decimal point like `decimal`.

1. Delete the code from the previous steps and enter the following code into the .NET Editor:

   C#

   ```
   decimal decimalQuotient = 7.0m / 5;
   Console.WriteLine($"Decimal quotient: {decimalQuotient}");
   ```

2. Run the code. You should see the following output:

   Output

   ```
   Decimal quotient: 1.4
   ```

For this to work, the quotient (left of the assignment operator) must be of type `decimal` **and** at least one of numbers being divided must also be of type `decimal` (both numbers can also be a `decimal` type).

Here are two additional examples that work equally well:

C#

```
decimal decimalQuotient = 7 / 5.0m;
decimal decimalQuotient = 7.0m / 5.0m;
```

However, the following lines of code won't work (or give inaccurate results):

C#

```csharp
int decimalQuotient = 7 / 5.0m;
int decimalQuotient = 7.0m / 5;
int decimalQuotient = 7.0m / 5.0m;
decimal decimalQuotient = 7 / 5;
```

Add code to perform division using literal decimal data

What if you are not working with literal values? In other words, what if you need to divide two variables of type `int` but do not want the result truncated? In that case, you must perform a data type cast from `int` to `decimal`. Casting is one type of data conversion that instructs the compiler to temporarily treat a value as if it were a different data type.

To cast `int` to `decimal`, you add the cast operator before the value. You use the name of the data type surrounded by parentheses in front of the value to cast it. In this case, you would add `(decimal)` before the variables `first` and `second`.

1. Delete the code from the previous steps and enter the following code into the .NET Editor:

C#

```csharp
int first = 7;
int second = 5;
decimal quotient = (decimal)first / (decimal)second;
Console.WriteLine(quotient);
```

2. Run the code. You should see the following output:

Output

```
1.4
```

Note

You've seen three uses for the parenthesis operator: method invocation, order of operations and casting.

Write code to determine the remainder after integer division

The modulus operator `%` tells you the remainder of `int` division. What you really learn from this is whether one number is divisible by another. This can be useful during long processing operations when looping through hundreds or thousands of data records and you want to provide feedback to the end user after every 100 data records have been processed.

1. Delete the code from the previous steps and enter the following code into the .NET Editor:

   C#

   ```csharp
   Console.WriteLine($"Modulus of 200 / 5 : {200 % 5}");
   Console.WriteLine($"Modulus of 7 / 5 : {7 % 5}");
   ```

2. Run the code. You should see the following output:

   Output

   ```
   Modulus of 200 / 5 : 0
   Modulus of 7 / 5 : 2
   ```

When the modulus is 0, that means the dividend is divisible by the divisor.

Order of operations

As you learned in the previous exercise, you can use the `()` symbols as the *order of operations* operators. However, this isn't the only way the order of operations is determined.

In math, PEMDAS is an acronym that helps students remember the order of operations. The order is:

1. **P**arentheses (whatever is inside the parenthesis is performed first)
2. **E**xponents
3. **M**ultiplication and **D**ivision (from left to right)
4. **A**ddition and **S**ubtraction (from left to right)

C# follows the same order as PEMDAS except for exponents. While there's no exponent operator in C#, you can use the System.Math.Pow method. The module "Call methods from the .NET Class Library using C#" will feature this method and others.

Write code to exercise C#'s order of operations

1. Delete the code from the previous steps and enter the following code into the .NET Editor:

C#

```
int value1 = 3 + 4 * 5;
int value2 = (3 + 4) * 5;
Console.WriteLine(value1);
Console.WriteLine(value2);
```

Here you see the difference when performing the same operations in a different order.

2. Run the code. You should see the following output:

Output

```
23
35
```

Recap

Here's what you've learned so far about mathematical operations in C#:

- Use operators like `+`, `-`, `*`, and `/` to perform basic mathematical operations.
- The division of two `int` values will result in the truncation of any values after the decimal point. To retain values after the decimal point, you need to cast the divisor or dividend (or both) from `int` into a floating point number like `decimal` first, then the quotient must be of the same floating point type as well in order to avoid truncation.
- Perform a cast operation to temporarily treat a value as if it were a different data type.
- Use the `%` operator to capture the remainder after division.
- The order of operations will follow the rules of the acronym PEMDAS.

Module complete:

Unlock achievement

.NET Editor

Press CTRL + M, TAB to exit the editor

Clear Run

Output