200 XP

Exercise - Explore data type casting and conversion

30 minutes

There are multiple techniques to perform a data type conversion. The technique you choose depends on your answer to two important questions:

- Is it possible, depending on the value, that attempting to change the value's data type would throw an exception at run time?
- Is it possible, depending on the value, that attempting to change the value's data type would result in a loss of information?

In this exercise, you work your way through these questions, the implications of their answers, and which technique you should use when you need to change the data type.

Prepare your coding environment

This module includes hands-on activities that guide you through the process of building and running demonstration code. You're encouraged to complete these activities using Visual Studio Code as your development environment. Using Visual Studio Code for these activities helps you to become more comfortable writing and running code in a developer environment that's used by professionals worldwide.

> **Note**
>
> If you have completed other Microsoft Learn modules in this C# series, you may have already created a project folder for code samples. If that's the case, you can skip the following section of steps, and delete the code in the `Project.cs` file used for a previous exercise.

1. Open Visual Studio Code.

   You can use the Windows Start menu (or equivalent resource for another OS) to open Visual Studio Code.

2. On the Visual Studio Code **File** menu, select **Open Folder**.

3. In the **Open Folder** dialog, navigate to the Windows Desktop folder.

   If you have different folder location where you keep code projects, you can use that folder location instead. For this training, the important thing is to have a location that's easy locate and remember.

4. In the **Open Folder** dialog, select **Select Folder**.

   If you see a security dialog asking if you trust the authors, select **Yes**.

5. On the Visual Studio Code **Terminal** menu, select **New Terminal**.

   Notice that a command prompt in the Terminal panel displays the folder path for the current folder. For example:

   ```dos
   C:\Users\someuser\Desktop>
   ```

6. To create a new console application in a specified folder, at the Terminal command prompt type: `dotnet new console -o ./CsharpProjects/TestProject` and then press Enter.

   This .NET CLI command uses a .NET program template to create a new C# console application project in the specified folder location. The command creates the CsharpProjects and TestProject folders for you, and uses TestProject as the name of the `.csproj` file.

7. In the EXPLORER panel, expand the **CsharpProjects** folder.

   You should see the TestProject folder and two files, a C# program file named Program.cs and a C# project file named TestProject.csproj.

8. In the EXPLORER panel, to view your code file in the Editor panel, select **Program.cs**.

9. Delete the existing code lines.

   You use this C# console project to create, build, and run code samples during this module.

10. Close the Terminal panel.

Question: Is it possible that attempting to change the value's data type would throw an exception at

run time?

The C# compiler attempts to accommodate your code, but doesn't compile operations that could result in an exception. When you understand the C# compiler's primary concern, understanding why it functions a certain way is easier.

Write code that attempts to add an `int` and a `string` and save the result in an `int`

1. Ensure that you have Visual Studio Code open and Program.cs displayed in the Editor panel.

   > **Note**
   >
   > Program.cs should be empty. If if isn't, select and delete all code lines.

2. Type the following code into the Visual Studio Code Editor:

   ```C#
   int first = 2;
   string second = "4";
   int result = first + second;
   Console.WriteLine(result);
   ```

   Here, you're attempting to add the values `2` and `4`. The value `4` is of type `string`. Will this work?

3. On the Visual Studio Code **File** menu, select **Save**.

   The Program.cs file must be saved before building or running the code.

4. In the EXPLORER panel, to open a Terminal at your TestProject folder location, right-click **TestProject**, and then select **Open in Integrated Terminal**.

   A Terminal panel should open, and should include a command prompt showing that the Terminal is open to your TestProject folder location.

5. At the Terminal command prompt, to run your code, type `dotnet run` and then press Enter.

   You should see the following approximate output

Output

```
C:\Users\someuser\Desktop\csharpprojects\TestProject\Program.cs(3,14): error
CS0029: Cannot implicitly convert type 'string' to 'int'
```

> **Note**
>
> If you see a message saying "Couldn't find a project to run", ensure that the Terminal command prompt displays the expected TestProject folder location. For example: `C:\Users\someuser\Desktop\csharpprojects\TestProject>`

6. Take a minute to consider why the compiler was unable to run the first code sample.

   The important part of the error message, **`(3,14): error CS0029: Cannot implicitly convert type 'string' to 'int'`**, tells you the problem is with the use of the `string` data type.

   But why can't the C# Compiler just handle the error? After all, you can do the *opposite* to concatenate a number to a `string` and save it in a string variable. Here, you change the data type of the `result` variable from `int` to `string`.

7. Update your code in the Visual Studio Code Editor as follows:

   C#

   ```
   int first = 2;
   string second = "4";
   string result = first + second;
   Console.WriteLine(result);
   ```

8. Save your code file, and then use Visual Studio Code to run your code.

   You should observe the following output:

   Output

   ```
   24
   ```

   The output is mathematically incorrect but completes by combining the values as the characters "2" and "4".

9. Examine, once again, the first code example where the `result` variable is of type `int`. The code with the error message.

```
C#

int first = 2;
string second = "4";
int result = first + second;
Console.WriteLine(result);
```

Why can't the C# compiler figure out that you want to treat the variable `second` containing `4` as a number, not a `string`?

Compilers make safe conversions

The C# compiler sees a potential problem in the making. The variable `second` is of type `string`, so it might be set to a different value like `"hello"`. If the C# compiler attempted to convert `"hello"` to a number that would cause an exception at runtime. To avoid this possibility, the C# compiler doesn't implicitly perform the conversion from `string` to `int` for you.

From the C# compiler's perspective, the safer operation would be to convert `int` into a `string` and perform concatenation instead.

If you intend to perform addition using a string, the C# compiler requires you to take more explicit control of the process of data conversion. In other words, it forces you to be more involved so that you can put the proper precautions in place to handle the possibility that the conversion could throw an exception.

If you need to change a value from the original data type to a new data type and the change could produce an exception at run time, you must perform a **data conversion**.

To perform data conversion, you can use one of several techniques:

- Use a helper method on the data type
- Use a helper method on the variable
- Use the `Convert` class' methods

You look at a few examples of these techniques for data conversion later in this unit.

Question: Is it possible that attempting to change the value's data type would result in a loss of information?

1. Delete or use the line comment operator `//` to comment out the code from the previous exercise step, and add the following code:

   C#

   ```
   int myInt = 3;
   Console.WriteLine($"int: {myInt}");

   decimal myDecimal = myInt;
   Console.WriteLine($"decimal: {myDecimal}");
   ```

2. Save your code file, and then use Visual Studio Code to run your code.

   You should see the following output:

   Output

   ```
   int: 3
   decimal: 3
   ```

   The key to this example is this line of code:

   C#

   ```
   decimal myDecimal = myInt;
   ```

   Since any `int` value can easily fit inside of a `decimal`, the compiler performs the conversion.

   The term *widening conversion* means that you're attempting to convert a value **from** a data type that could hold *less* information **to** a data type that can hold *more* information. In this case, a value stored in a variable of type `int` converted to a variable of type `decimal`, doesn't lose information.

   When you know you're performing a widening conversion, you can rely on **implicit conversion**. The compiler handles implicit conversions.

Perform a cast

1. Delete or use the line comment operator `//` to comment out the code from the previous exercise step, and add the following code:

   C#

   ```
   decimal myDecimal = 3.14m;
   Console.WriteLine($"decimal: {myDecimal}");

   int myInt = (int)myDecimal;
   Console.WriteLine($"int: {myInt}");
   ```

   To perform a cast, you use the casting operator `()` to surround a data type, then place it next to the variable you want to convert (example: `(int)myDecimal`). You perform an **explicit conversion** to the defined cast data type (`int`).

2. Save your code file, and then use Visual Studio Code to run your code.

   You should see the following output:

   Output

   ```
   decimal: 3.14
   int: 3
   ```

   The key to this example is this line of code:

   C#

   ```
   int myInt = (int)myDecimal;
   ```

   The variable `myDecimal` holds a value that has precision after the decimal point. By adding the casting instruction `(int)`, you're telling the C# compiler that you understand it's possible you'll lose that precision, and in this situation, it's fine. You're telling the compiler that you're performing an intentional conversion, an **explicit conversion**.

Determine if your conversion is a "widening conversion" or a "narrowing conversion"

The term **narrowing conversion** means that you're attempting to convert a value from a data type that can hold *more* information to a data type that can hold less information. In this case, you may lose information such as precision (that is, the number of values after the decimal point). An example is converting value stored in a variable of type `decimal` into a variable of type `int`. If you print the two values, you would possibly notice the loss of information.

When you know you're performing a narrowing conversion, you need to perform a **cast**. Casting is an instruction to the C# compiler that you know precision may be lost, but you're willing to accept it.

If you're unsure whether you lose data in the conversion, write code to perform a conversion in two different ways and observe the changes. Developers frequently write small tests to better understand the behaviors, as illustrated with the next sample.

1. Delete or use the line comment operator `//` to comment out the code from the previous exercise step, and add the following code:

   C#

   ```
   decimal myDecimal = 1.23456789m;
   float myFloat = (float)myDecimal;

   Console.WriteLine($"Decimal: {myDecimal}");
   Console.WriteLine($"Float  : {myFloat}");
   ```

2. Save your code file, and then use Visual Studio Code to run your code.

   You should see the following output:

   Output

   ```
   Decimal: 1.23456789
   Float:   1.234568
   ```

   You can observe from the output that casting a `decimal` into a `float` is a narrowing conversion because you're losing precision.

Performing Data Conversions

Earlier, it was stated that a value change from one data type into another could cause a runtime exception, and you should perform data conversion. For data conversions, there are three

techniques you can use:

- Use a helper method on the variable
- Use a helper method on the data type
- Use the `Convert` class' methods

Use `ToString()` to convert a number to a `string`

Every data type variable has a `ToString()` method. What the `ToString()` method does depends on how it's implemented on a given type. However, on most primitives, it performs a widening conversion. While this isn't strictly necessary (since you can rely on implicit conversion in most cases) it can communicate to other developers that you understand what you're doing and it's intentional.

Here's a quick example of using the `ToString()` method to explicitly convert `int` values into `string`s.

1. Delete or use the line comment operator `//` to comment out the code from the previous exercise step, and add the following code:

   C#

   ```
   int first = 5;
   int second = 7;
   string message = first.ToString() + second.ToString();
   Console.WriteLine(message);
   ```

2. Save your code file, and then use Visual Studio Code to run your code. When you run the code, the output should display a concatenation of the two values:

   Output

   ```
   57
   ```

Convert a `string` to an `int` using the `Parse()` helper method

Most of the numeric data types have a `Parse()` method, which converts a string into the given data type. In this case, you use the `Parse()` method to convert two strings into `int` values, then

add them together.

1. Delete or use the line comment operator `//` to comment out the code from the previous exercise step, and add the following code:

   C#

   ```
   string first = "5";
   string second = "7";
   int sum = int.Parse(first) + int.Parse(second);
   Console.WriteLine(sum);
   ```

2. Save your code file, and then use Visual Studio Code to run your code. When you run the code, the output should display a sum of the two values:

   Output

   ```
   12
   ```

3. Take a minute to try to spot the potential problem with the previous code example? What if either of the variables `first` or `second` are set to values that can't be converted to an `int`? An exception is thrown at runtime. The C# compiler and runtime expects you to plan ahead and prevent "illegal" conversions. You can mitigate the runtime exception in several ways.

   The easiest way to mitigate this situation is by using `TryParse()`, which is a better version of the `Parse()` method.

Convert a `string` to a `int` using the `Convert` class

The `Convert` class has many helper methods to convert a value from one type into another. In the following code example, you convert a couple of strings into values of type `int`.

> **Note**
>
> The code samples in this exercise are designed based on en-US culture settings, and use a period (`.`) as the decimal separator. Building and running the code with a culture setting that uses a different decimal separators (such as a comma `,`) may give unexpected results or errors. To fix this issue, replace the period decimal separators in the code samples with your local decimal separator (such as `,`). Alternatively, to run a program using the en-US culture

setting, add the following code to the top of your program: `using System.Globalization;` and after any other `using` statements add `CultureInfo.CurrentCulture = new CultureInfo("en-US");`.

1. Delete or use the line comment operator `//` to comment out the code from the previous exercise step, and add the following code:

```
C#

string value1 = "5";
string value2 = "7";
int result = Convert.ToInt32(value1) * Convert.ToInt32(value2);
Console.WriteLine(result);
```

2. Save your code file, and then use Visual Studio Code to run your code.

   You should see the following output:

```
Output

35
```

> **Note**
>
> Why is the method name `ToInt32()`? Why not `ToInt()`? `System.Int32` is the name of the underlying data type in the .NET Class Library that the C# programming language maps to the keyword `int`. Because the `Convert` class is also part of the .NET Class Library, it is called by its full name, not its C# name. By defining data types as part of the .NET Class Library, multiple .NET languages like Visual Basic, F#, IronPython, and others can share the same data types and the same classes in the .NET Class Library.

The `ToInt32()` method has 19 overloaded versions allowing it to accept virtually every data type.

you used the `Convert.ToInt32()` method with a string here, but you should probably use `TryParse()` when possible.

So, when should you use the `Convert` class? The `Convert` class is best for converting fractional numbers into whole numbers (`int`) because it rounds up the way you would

expect.

Compare casting and converting a `decimal` into an `int`

The following example demonstrates what happens when you attempt to cast a `decimal` into an `int` (a narrowing conversion) versus using the `Convert.ToInt32()` method to convert the same `decimal` into an `int`.

1. Delete or use the line comment operator `//` to comment out the code from the previous exercise step, and add the following code:

   C#

   ```
   int value = (int)1.5m; // casting truncates
   Console.WriteLine(value);

   int value2 = Convert.ToInt32(1.5m); // converting rounds up
   Console.WriteLine(value2);
   ```

2. Save your code file, and then use Visual Studio Code to run your code.

   You should see the following output:

   Output

   ```
   1
   2
   ```

Casting truncates and converting rounds

When you're casting `int value = (int)1.5m;`, the value of the float is truncated so the result is `1`, meaning the value after the decimal is ignored completely. you could change the literal float to `1.999m` and the result of casting would be the same.

When you're converting using `Convert.ToInt32()`, the literal float value is properly rounded up to `2`. If you changed the literal value to `1.499m`, it would be rounded down to `1`.

Recap

You covered several important concepts of data conversion and casting:

- Prevent a runtime error while performing a data conversion
- Perform an explicit cast to tell the compiler you understand the risk of losing data
- Rely on the compiler to perform an implicit cast when performing an expanding conversion
- Use the `()` cast operator and the data type to perform a cast (for example, `(int)myDecimal`)
- Use the `Convert` class when you want to perform a narrowing conversion, but want to perform rounding, not a truncation of information

Check your knowledge

1. Which is the best technique to convert a decimal type to an int type in C#? *

- ◯ cast
- ◯ narrowing
- ◯ implicit conversion

2. Which of the following conversion rounds the value (versus truncate)? *

- ◯ `int cost = (int)3.75m;`
- ◯ `int cost = Convert.ToInt32(3.75m);`
- ◯ `uint cost = (uint)3.75m;`

Check your answers