Contents

Inputs and Outputs

Reading Input from the User

- Input and output in CLI
  - Our programs use a command-line interface, where input and output come from text printed in a "terminal" or "console"
  - We've already seen that `Console.WriteLine` prints text from your program on the screen to provide output to the user
  - The equivalent method for reading input is `Console.ReadLine()`, which waits for the user to type some text in the console and then returns it to your program
  - In general, the `Console` class represents the command-line interface
- Using `Console.ReadLine()`
  - Example usage:

```
using System;

class PersonalizedWelcomeMessage
{
  static void Main()
  {
    string firstName;
    Console.WriteLine("Enter your first name:");
    firstName = Console.ReadLine();
    Console.WriteLine($"Welcome, {firstName}!");
  }
}
```

    This program first declares a `string` variable named `firstName`. On the second line, it uses `Console.WriteLine` to display a message (instructions for the user). On the third line, it calls the `Console.ReadLine()` method, and assigns its return value (result) to the `firstName` variable. This means the program waits for the user to type some text and press "Enter", and then stores that text in `firstName`. Finally, the program uses string interpolation in `Console.WriteLine` to display a message including the contents of the `firstName` variable.

  - `Console.ReadLine` is the "inverse" of `Console.WriteLine`, and the way you use it is also the "inverse"
  - While `Console.WriteLine` takes an argument, which is the text you want to display on the screen, `Console.ReadLine()` takes no arguments because it does not need any input from your program – it will always do the same thing
  - `Console.WriteLine` has no "return value" - it does not give any output back to your program, and the only effect of calling it is that text is displayed on the screen
  - `Console.ReadLine()` does have a return value, specifically a `string`. This means you can use the result of this method to assign a `string` variable, just like you can use the result of an arithmetic expression to assign a numeric variable.
  - The `string` that `Console.ReadLine()` returns is **one line of text** typed in the console. When you call it, the computer will wait for the user to type some text and then press "Enter", and everything the user typed before pressing "Enter" gets returned from `Console.ReadLine()`

Parsing user input

- `Console.ReadLine()` always returns the same type of data, a `string`, regardless of what the user enters
  - If you ask the user to enter a number, `ReadLine` will output that number as a `string`
  - For example, if you ask the user to enter his/her age, and the user enters 21, `Console.ReadLine()` will return the string `"21"`
- If we want to do any kind of arithmetic with a number provided by the user, we will need to convert that `string` to a numeric type like `int` or `double`. Remember that casting cannot be used to convert numeric data *to or from* `string` data.
- When converting numeric data to `string` data, we use string interpolation:

```
int myAge = 29;
//This does not work:
//string strAge = (string) myAge;
string strAge = $"{myAge}";
```

- In the other direction, we use a method called Parse to convert strings to numbers:

  ```
  string strAge = "29";
  //This does not work:
  //int myAge = (int) strAge;
  int myAge = int.Parse(strAge);
  ```

- The int.Parse method takes a string as an argument, and returns an int containing the numeric value written in that string

- Each built-in numeric type has its own Parse method
  - int.Parse("42") returns the value 42
  - long.Parse("42") returns the value 42L
  - double.Parse("3.65") returns the value 3.65
  - float.Parse("3.65") returns the value 3.65f
  - decimal.Parse("3.65") returns the value 3.65m

- The Parse methods are useful for converting user input to useable data. For example, this is how to get the user's age as an int:

  ```
  Console.WriteLine("Enter your age:");
  string ageString = Console.ReadLine();
  int age = int.Parse(ageString);
  ```

More detail on the Parse methods

- Console.WriteLine is a method that takes input from your program, in the form of an argument, but does not return any output. Meanwhile, Console.ReadLine is a method that does not have any arguments, but it returns output to your program (the user's string).

- int.Parse is a method that both takes input (the string argument) and returns output (the converted int value)

- When executing a statement such as

  ```
  int answer = int.Parse("42");
  ```

  the computer must evaluate the expression on the right side of the = operator before it can do the assignment. This means it calls the int.Parse method with the string "42" as input. The method's code then executes, converting "42" to an integer, and it returns a result, the int value 42. This value can now be assigned to the variable answer.

- Since the return value of a Parse method is a numeric type, it can be used in arithmetic expressions just like a numeric-type variable or literal. For example, in this statement:

```
double result = double.Parse("3.65") * 4;
```

To evaluate the expression on the right side of the = operator, the computer must first call the method `double.Parse` with the input `"3.65"`. Then the method's return value, `3.65`, is used the math operation as if it was written `3.65 * 4`. So the computer implicitly converts `4` to a `double` value, performs the multiplication on `double`s, and gets the resulting value `14.6`, which it assigns to the variable `result`.

- Another example of using the result of `Parse` to do math:

  ```
  Console.WriteLine("Please enter the year.");
  string userInput = Console.ReadLine();
  int curYear = int.Parse(userInput);
  Console.WriteLine($"Next year it will be {curYear +
   ↪  1}");
  ```

  Note that in order to do arithmetic with the user's input (i.e. add 1), it must be a numeric type (i.e. `int`), not a `string`. This is why we often call a `Parse` method on the data returned by `Console.ReadLine()`.

- The previous example can be made shorter and simpler by combining the `Parse` and `ReadLine` methods in one statement. Specifically, you can write:

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

  In this statement, the return value (output) of one method is used as the argument (input) to another method. When the computer executes the statement, it starts by evaluating the `int.Parse(...)` method call, but it cannot actually execute the `Parse` method yet because its argument is an expression, not a variable or value. In order to determine what value to send to the `Parse` method as input, it must first evaluate the `Console.ReadLine()` method call. Since this method has no arguments, the computer can immediately start executing it; the `ReadLine` method waits for the user to type a line of text, then returns that text as a `string` value. This return value can now be used as the argument to `int.Parse`, and the computer starts executing `int.Parse` with the user-provided string as input. When the `Parse` method returns an `int` value, this value becomes the value of the entire expression `int.Parse(Console.ReadLine())`, and the computer assigns it to the variable `curYear`.

- Notice that by placing the call to `ReadLine` inside the argument to `Parse`, we have eliminated the variable `userInput` entirely. The `string` returned by `ReadLine` does not need to be stored any-where (i.e. in a variable); it only needs to exist long enough to be sent to the `Parse` method as input.

Correct input formatting

- The Parse methods *assume* that the string they are given as an argument (i.e. the user input) actually contains a valid number. But the user may not follow directions, and invalid input can cause a variety of errors.
- If the string does not contain a number at all – e.g. `int badIdea = int.Parse("Hello");` – the program will fail with the error `System.FormatException`
- If the string contains a number with a decimal point, but the `Parse` method is for an integer datatype, the program will also fail with `System.FormatException`. For example, `int fromFraction = int.Parse("52.5");` will cause this error. This will happen even if the number in the string ends in ".0" (meaning it has no fractional part), such as `int wholeNumber = int.Parse("45.0");`.
- If the string has extraneous text before or after the number, such as `"$18.95"` or `1999!`, the program will fail with the error `System.FormatException`
- If the string contains a number that cannot fit in the desired datatype (due to overflow or underflow), the behavior depends on the datatype:
  - For the integer types (`int` and `long`), the program will fail with the error `System.OverflowException`. For example, `int.Parse("3000000000")` will cause this error because 3000000000 is larger than $2^{31} - 1$ (the maximum value an `int` can store).
  - For the floating-point types (`float` and `double`), no error will be produced. Instead, the result will be the same as if an overflow or underflow had occurred during normal program execution: an overflow will produce the value `Infinity`, and an underflow will produce the value `0`. For example, `float tooSmall = float.Parse("1.5e-55");` will assign `tooSmall` the value 0, while `double tooBig = double.Parse("1.8e310");` will assign `tooBig` the value `double.Infinity`.
- Acceptable string formats vary slightly between the numeric types, due to the different ranges of values they can contain
  - `int.Parse` and `long.Parse` will accept strings in the format `([ws])([sign])[digits]([ws])`, where `[ws]` represents empty spaces and groups in parentheses are **optional**. This means that a string with leading or trailing spaces will not cause an error, unlike a string with other extraneous text around the number.
  - `decimal.Parse` will accept strings in the format `([ws])([sign])([digits],)[digits](`

  Note that you can optionally include commas between groups of digits, and the decimal point is also optional. This means a string like `"18,999"` is valid for `decimal.Parse` but not for `int.Parse`.
- `float.Parse` and `double.Parse` will accept strings in the format `([ws])([sign])([digits],)[digits](.[digits])(e[sign][digits])([ws])`. As with `decimal`, you can include commas between groups of digits. In addition, you can write the string in scientific notation with the letter "e" or "E" followed by an exponent, such as `"-9.44e15"`.

Output with Variables

Converting from numbers to strings

- As we saw in a previous lecture (Datatypes and Variables), the `Console.WriteLine` method needs a `string` as its argument
- If the variable you want to display is not a `string`, you might think you could cast it to a `string`, but that will not work – there is no explicit conversion from `string` to numeric types
  - This code:

    ```
    double fraction = (double) 47 / 6;
    string text = (string) fraction;
    ```

    will produce a compile error
- You *can* convert numeric data to a `string` using string interpolation, which we've used before in `Console.WriteLine` statements:

  ```
  int x = 47, y = 6;
  double fraction = (double) x / y;
  string text = $"{x} divided by {y} is {fraction}";
  ```

  After executing this code, `text` will contain "47 divided by 6 is 7.8333333"
- String interpolation can convert any expression to a `string`, not just a single variable. For example, you can write:

  ```
  Console.WriteLine($"{x} divided by {y} is {(double) x
   ↪  / y}");
  Console.WriteLine($"{x} plus 7 is {x + 7}");
  ```

  This will display the following output:

  ```
  47 divided by 6 is 7.8333333
  47 plus 7 is 54
  ```

Note that writing a math expression inside a string interpolation statement does not change the values of any variables. After executing this code, x is still 47, and y is still 6.

The `ToString()` method

- String interpolation does not "magically know" how to convert numbers to strings – it delegates the task to the numbers themselves
- This works because all data types in C# are objects, even the built-in ones like `int` and `double`
  - Since they are objects, they can have methods
- **All** objects in C# are guaranteed to have a method named `ToString()`, whose return value (result) is a `string`
- Meaning of `ToString()` method: "Convert this object to a `string`, and return that `string`"
- This means you can call the `ToString()` method on any variable to convert it to a `string`, like this:

  ```
  int num = 42;
  double fraction = 33.5;
  string intText = num.ToString();
  string fracText = fraction.ToString();
  ```

  After executing this code, `intText` will contain the string "42", and `fracText` will contain the string "33.5"
- String interpolation calls `ToString()` on each variable or expression within braces, asking it to convert itself to a string
  - In other words, these three statements are all the same:

    ```
    Console.WriteLine($"num is {num}");
    Console.WriteLine($"num is {intText}");
    Console.WriteLine($"num is {num.ToString()}");
    ```

    Putting `num` within the braces is the same as calling `ToString()` on it.

String Concatenation

- Now that we've seen `ToString()`, we can introduce another operator: the concatenation operator
- Concatenation basics
  - Remember, the + operator is defined separately for each data type. The "`double + double`" operator is different from the "`int + int`" operator.

  - If the operand types are `string` (i.e. `string + string`), the + operator performs concatenation, not addition
  - You can concatenate `string` literals or `string` variables:
    ```
    string greeting = "Hi there, " + "John";
    string name = "Paul";
    string greeting2 = "Hi there, " + name;
    ```
    After executing this code, `greeting` will contain "Hi there, John" and `greeting2` will contain "Hi there, Paul"
- Concatenation with mixed types
  - Just like with the other operators, both operands (both sides of the +) must be the same type
  - If one operand is a `string` and the other is not a `string`, the `ToString()` method will automatically be called to convert it to a `string`
  - Example: In this code:
    ```
    int bananas = 42;
    string text = "Bananas: " + bananas;
    ```
    The + operator has a `string` operand and an `int` operand, so the `int` will be converted to a `string`. This means the computer will call `bananas.ToString()`, which returns the string "42". Then the `string + string` operator is called with the operands "Bananas:" and "42", which concatenates them into "Bananas: 42".

Output with concatenation

- We now have two different ways to construct a string for `Console.WriteLine`: Interpolation and concatenation
- Concatenating a string with a variable will automatically call its `ToString()` method, just like interpolation will. These two `WriteLine` calls are equivalent:
  ```
  int num = 42;
  Console.WriteLine($"num is {num}");
  Console.WriteLine("num is " + num);
  ```
- It's usually easier to use interpolation, since when you have many variables the + signs start to add up. Compare the length of these two equivalent lines of code:
  ```
  Console.WriteLine($"The variables are {a}, {b}, {c},
   ↪  {d}, and {e}");
  Console.WriteLine("The variables are " + a + ", " + b
   ↪  + ", " + c + ", " + d + ", and " + e);
  ```
- Be careful when using concatenation with numeric variables: the meaning of + depends on the types of its two operands

- If both operands are numbers, the + operator does addition
- If both operands are strings, the + operator does concatenation
- If *one* argument is a string, the other argument will be converted to a string using `ToString()`
- Expressions in C# are always evaluated **left-to-right**, just like arithmetic
- Therefore, in this code:

  ```
  int var1 = 6, var2 = 7;
  Console.WriteLine(var1 + var2 + " is the result");
  Console.WriteLine("The result is " + var1 + var2);
  ```

  The first `WriteLine` will display "13 is the result", because `var1` and `var2` are both `int`s, so the first + operator performs addition on two `int`s (the resulting number,13, is then converted to a `string` for the second + operator). However, the second `WriteLine` will display "The result is 67", because both + operators perform concatenation: The first one concatenates a string with `var1` to produce a string, and then the second one concatenates this string with `var2`
- If you want to combine addition and concatenation in the same line of code, use parentheses to make the order and grouping of operations explicit. For example:

  ```
  int var1 = 6, var2 = 7;
  Console.WriteLine((var1 + var2) + " is the
   ↪  result");
  Console.WriteLine("The result is " + (var1 +
   ↪  var2));
  ```

  In this code, the parentheses ensure that `var1 + var2` is always interpreted as addition.