Reading Input and Displaying Output

Principles of Computer Programming I

Spring/Fall 20XX

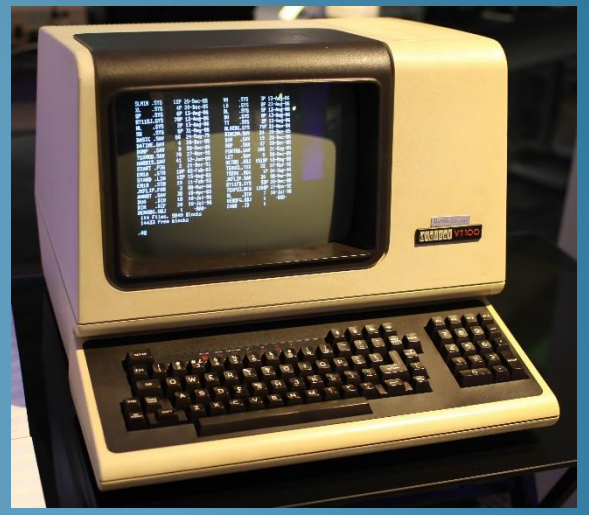
AUGUSTA UNIVERSITY

Outline

- Reading user input
- Parsing user input
  - Parse methods
  - Failures and correct formatting
- Converting numbers to strings
  - The ToString method
- String concatenation
- String modifications

Input from the User

- With a CLI, output = print text to screen, input = read text from keyboard
- `Console` represents the "terminal" interface



Output:

Text to output

```
Console.WriteLine("Hi!");
```

Result: Terminal displays "Hi!"

Input:

Waits until user presses "Enter"

```
Console.ReadLine();
```

Result: A `string` value containing text typed in at terminal (one line)

User Input Example

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

Declare a string variable named firstName

Print text to the console

Wait for a line of text to be typed in

Insert the text from firstName into this string and print it

Assign the text received from the console to firstName

Method Input and Output

- `WriteLine` gets input from your program: the argument

```
Console.WriteLine("Enter your first name");
```

Argument: a `string` value

```
myVar = Console.WriteLine("Hi!");
```

✖ `WriteLine` does not return a value, nothing to assign

- `ReadLine` gives output to your program: the **return value**

```
string name = Console.ReadLine();
```

Result of expression: a `string` value (the return value)

No argument

```
int result = 3 * 9 + 1;
```

Result of expression: an `int` value

Outline

- Reading user input
- **Parsing user input**
  - Parse methods
  - Failures and correct formatting
- Converting numbers to strings
  - The ToString method
- String concatenation
- String modifications

User Input and Data Types

- Expressions have exactly one result type; methods have exactly one return type

int double

```
365 / 7.0
```

double

```
Console.ReadLine()
```

string

- `ReadLine` always returns a `string`, no matter what user enters

```
Console.WriteLine("Enter your age:");
string age = Console.ReadLine();
Console.WriteLine("Enter your height in meters:");
string height = Console.ReadLine();
```

Gets the value "30"

Gets the value "1.77" (not 1.77)

String Conversions

- Recall: Converting `int` to `string` requires a specific function

```
int myAge = 30;
string strAge = (string) myAge;
```

Error! Can't convert `int` to string

```
string strAge = $"{myAge}";
```

- Similarly, converting `string` to `int` uses a method

```
string strAge = "30";
int myAge = (int) strAge;
```

Error! Can't convert `string` to `int`

```
int myAge = int.Parse(strAge);
```

Result: `int`

Parse Methods

- Each built-in numeric type has a "Parse" method

```
int.Parse("42") ⟶ 42
long.Parse("42") ⟶ 42L
double.Parse("3.65") ⟶ 3.65
float.Parse("3.65") ⟶ 3.65f
```

- Useful for turning user input into correct data type:

Gets "30"

Gets 30

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

Method Input and Output

- Parse methods have both **argument** (input) and **return value** (output)

Argument: a `string` value

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

Assigned value: 42

Result of method call: an `int` value

- Return value can be used in expressions

Implicitly convert to `double`

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

Assigned value: `14.6`

Return value: `3.65`

Parsing User Input

- Input must be parsed before doing 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}");
```

String interpolation will print result of expression

`int + int`

- Can call both methods in one statement:

Assign return value to `curYear`

```
Console.WriteLine("Please enter the year.");
int curYear = int.Parse(Console.ReadLine());
```

`Parse` converts the `string`, returns an `int`

Argument to `Parse` is an expression, so evaluate it

Execute method, return result (a `string`)

Outline

- Reading user input
- Parsing user input
  - Parse methods
  - **Failures and correct formatting**
- Converting numbers to strings
  - The ToString method
- String concatenation
- String modifications

What Could Go Wrong?

- What if the string is not a number?

```
int badIdea = int.Parse("Hello!");
```

→ `FormatException!`

  - Answer: Program crashes with `System.FormatException`

- The string is a number, but the wrong type of number?

```
int fromFraction = int.Parse("52.5");
```

→ `FormatException!`

```
int wholeNumber = int.Parse("45.0");
```

→ `FormatException!`

What Could Go Wrong?

- The user adds extra text to their number?

```
decimal money = decimal.Parse("$18.95");
int year = int.Parse("1999!");
```

→ `FormatException!`

→ `FormatException!`

- The string is a number that can't fit the desired type?

```
int tooBig = int.Parse("-3000000000");
float tooSmall = float.Parse("1.5e-55");
```

→ `OverflowException!`

→ `tooSmall` gets `0`

  - Integer types: Crash with `System.OverflowException`
  - Floating-point types: Normal overflow/underflow behavior

Correct Number Formatting

- Acceptable string formats vary by number type
  - `int` and `long`:

Sign is optional

[ws][sign][digits][ws]

Can begin or end with whitespace

```
"+6961   "
"-18005"
"    13    "
```

  - `decimal`:

Decimal digits, optional

[ws][sign][digits],[digits][.digits][ws]

Can use commas!

Not a valid `int`

```
"+18,999   "
"286669.4"
"   -6.378"
```

Correct Number Formatting

- `float` and `double`:

Decimal digits

[ws][sign][digits],[digits](.[digits])(e[sign][digits])[ws]

Can use commas

"Exponent" notation

```
"-9.44e15"
```

```
"165,324,888.01"
```

```
"+950E-18"
```

```
"6.02e23   "
```

Parse will succeed, but value is stored as `9.5e-16`

- References: Double.Parse, Int32.Parse, Decimal.Parse

Outline

- Reading user input
- Parsing user input
  - Parse methods
  - Failures and correct formatting
- **Converting numbers to strings**
  - The ToString method
- String concatenation
- String modifications

String Interpolation

- Casting won't convert numbers to `strings`, interpolation will

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

"47" "6" "7.8333333"

- Interpolation can convert any expression, not just a variable

"7.8333333"

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

"54"

This does **not** change the value of x or y:

```
Console.WriteLine($"x={x}, y={y}");
```

Prints "x=47, y=6"

Behind the Scenes

- Interpolation doesn't know how to convert numbers to strings
- All data types in C# are objects, even `int`
- All objects in C# have a `ToString()` method – "convert this object to a `string`"

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

Call method `ToString` on object `num`

Result: "42"

Result: "33.5"

Call method `ToString` on object `fraction`

Interpolation and ToString

- Interpolation calls `ToString()` on variables to convert them

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

```
x.ToString();        y.ToString();        fraction.ToString();

"47"                 "6"                  "7.833333333333333"

"47 divided by 6 is 7.833333333333333"
```

Interpolation and ToString

- What about expressions? How do they get converted?
- Result of expression is an object, so we can call `ToString` on it

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

A double object

```
x.ToString();   y.ToString();   7.833333333333333

                                7.833333333333333.ToString()

"47"            "6"             "7.833333333333333"

"47 divided by 6 is 7.833333333333333"
```

Storing and Printing Strings

- To convert a single variable to a string, either works

```
int num = 42;
string intText = num.ToString();
```

```
int num = 42;
string intText = $"{num}";
```

- To display a single variable on the screen, no conversion needed

```
Console.WriteLine(num);
```

← `WriteLine(int)` internally converts num to a string

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

← Convert num to a string, then call `WriteLine(string)`

- To insert *multiple* variables into a string, use interpolation

```
Console.WriteLine($"x={x}, y={y}");
```

```
string text = $"x={x}, y={y}";
```

Outline

- Reading user input
- Parsing user input
  - Parse methods
  - Failures and correct formatting
- Converting numbers to strings
  - The ToString method
- **String concatenation**
- String modifications

String Concatenation

- The + operator is a different function for each operand type
- If the operand types are `string`, it performs **concatenation**

```
string greeting = "Hi there, " + "John";
```

Result: "Hi there, John"

`string + string` operator

- Works as expected for `string` variables:

```
string name = "Paul";
string greeting2 = "Hi there, " + name;
```

Result: "Hi there, Paul"

Mixed Types with +

- What if we use + operator with a string and a number?

```
int bananas = 42;
string text = "Bananas: " + bananas;
```

- Answer: Converts the other argument to a string with ToString()

```
string text = "Bananas: " + bananas;
```

bananas.ToString()

"Bananas: " + "42" → "Bananas: 42"

string + string, concatenation

Interpolation and Concatenation

- Two different ways to insert variables into a `string`:

```
int num = 2500;
Console.WriteLine($"num is {num}");
Console.WriteLine("num is " + num);
```

- Interpolation is easier to write with many variables

```
Console.WriteLine($"The variables are {a}, {b}, {c}, {d}, and {e}");
Console.WriteLine("The variables are " + a + ", " + b + ", "
     + c + ", " + d + ", and " + e);
```

  - Also, only interpolation allows format specifiers

```
Console.WriteLine($"num is {num:N}");
```

Concatenation Puzzle

- Code executes left-to-right, binary operators (like +) are grouped left-to-right
- What does this produce?

int + int → int + string

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

"13 is the result"

"The result is 67"

string + int → string + int

Parentheses Define Order

- Parentheses make order, grouping explicit
- Use to ensure compiler does what you mean

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

"13 is the result"

"The result is 13"

Evaluates second, concatenates string with result of parentheses

Evaluates first, so + operator sees `int + int`

Outline

- Reading user input
- Parsing user input
  - Parse methods
  - Failures and correct formatting
- Converting numbers to strings
  - The ToString method
- String concatenation
- **String modifications**

Formatting Strings with Methods

- `strings` are objects, and have methods (including `ToString`)
- Some are useful for formatting input/output
- `ToUpper()`: Returns an UPPERCASE version of the string

```
string greeting = "Hi there, " + "John";
Console.WriteLine(greeting.ToUpper());
```

```
HI THERE, JOHN
```

- `ToLower()`: Returns a lowercase version of the string

```
Console.WriteLine(greeting.ToLower());
```

```
hi there, john
```

  - These do **not** modify the original string

```
Console.WriteLine(greeting);
```

```
Hi there, John
```

Formatting Strings with Methods

- `Trim()`: Removes leading and trailing whitespace (in a copy)

Gets the value "George"

```
string name = "   George   ";
string trimmedName = name.Trim();
```

name is still " George "

- Useful for "fixing" user input

```
Console.WriteLine("Enter your first name:");
string inputName = Console.ReadLine();
string name = inputName.Trim();
Console.WriteLine($"Welcome, {name}!");
```

User enters " George "

name gets "George"

Prints Welcome, George!

Without `Trim()`, output would be Welcome, George !