C# Basics

Topics Covered

- Unassigned Variables
- Memory Usage
- Initializing Variables
- System.Math
- Creating Informative Strings
  - Concatenation
  - Formatting
  - Interpolation
- Object Methods: ToString()
- Console.ReadLine()
- double.Parse()
- double.TryParse()
- static and instance Methods
- Professional Best Practices
- Using a while Loop

Calculate the Area of a Circle

Create another Console application and name it AreaOfCircle. Add code to the Main method as shown below.

```
static void Main(string[] args)
{
    double radius, area;  // Uninitialized variables
    radius = 20;          // Assignment
    area = radius * radius * Math.PI;   // Compute and assign to area variable

    // Show result
    Console.WriteLine("The area of a circle of radius " + radius + " is " + area);
}
```

In this program, we declare two variables of type double.

When we declare a variable, the C# compiler sets aside a piece of memory large enough to store the value of that variable.

The double Datatype

The amount of memory required to represent a variable depends on its type.

double is a floating-point type, meaning that it represents a number with a decimal point.

C# variables of type double can represent numbers ranging from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$.

Each variable of type double requires 8 bytes (64 bits) of memory.

Initializing Variables

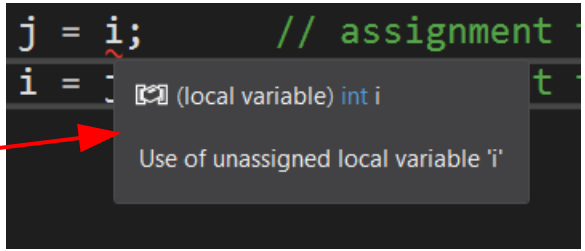
```
static void Main(string[] args)
{
    double radius, area;  // Uninitialized variables
    radius = 20;          // Assignment
    area = radius * radius * Math.PI;   // Compute and assign to area variable

    // Show result
    Console.WriteLine("The area of a circle of radius " + radius + " is " + area);
}
```

The first line of code declares the variables, but does not give them a value.

These are called “uninitialized variables”. The C# compiler does not allow us to use any variable until after it has been assigned an initial value.

Declaring and Initializing Variables

```
int i;      // declaration without initialization
int j = 10; // declare and initialize
j = i;      // use of an uninitialized variable is an error.
i = j;      // assignment to an uninitialized variable is valid.
```



Variables are used to store and manipulate information.

Variables can be declared with or without initialization.

Use of an uninitialized variable is an error. Visual Studio flags errors with a squiggly red underscore.

Hover the mouse over the error, and Visual Studio will describe the problem.

Values in Memory

In the first line of our program, both variables are unassigned. Technically, the C# compiler fills the memory with zeros, giving them a value 0, but we are still not allowed to use them.

```
double radius, area;
```

Memory	
Variable	Value
radius	Empty (0)
area	Empty (0)

In the 2nd line, we assign a value of 20 to the variable named *radius*.

```
radius = 20;
```

Memory	
Variable	Value
radius	20
area	Empty (0)

Since it is now assigned, we can use it in the calculation on line 3.

```
area = radius * radius * Math.PI;
```

Memory	
Variable	Value
radius	20
area	1256.63706143592

The result of the calculation is assigned to the variable named *area*. Since both variables are now assigned, we can use them in the last line.

The Math Class

```
area = radius * radius * Math.PI;
```

The calculation on line 3 uses the Math class defined in the System namespace.

Math provides the constants **π** (the ratio of a circle's circumference to its diameter) and **e** (the base of the natural logarithms).

Math also defines numerous methods, for example to calculate square roots, perform exponentiation, and to calculate logarithms or trigonometric values.

String Concatenation

```
Console.WriteLine("The area of a circle of radius " + radius + " is " + area);
```

In the last line of code, we call the Console class' WriteLine method with an argument of type string.

We construct this string 'on the fly' by concatenating two literal strings with the our radius and area variables.

When we join strings together in this way, the operation is called 'concatenation'. The result is this:

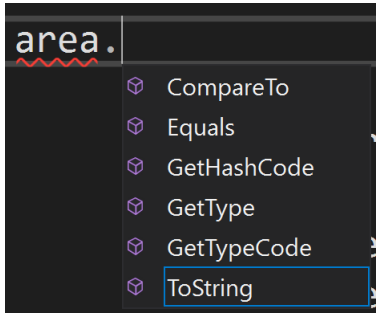
```
The area of a circle of radius 20 is 1256.63706143592
```

Converting Numbers (or Anything) to Strings

The output of our program is OK, but whoa – all those decimal places!

When our numbers are concatenated to the strings, under the hood, the compiler is calling a method on the numbers.

Numbers have methods? Yes!! In C#, every variable you will ever use is an Object, and all objects have methods. You can see the methods on our area variable by typing “area.”. As soon as you type that last ‘.’, Visual Studio will show you all of the methods that are available for that object:



We see here that our area has a method named **ToString**.
This is the method that the compiler uses to convert our doubles to strings.

The dot operator is used to access any variable’s members.

String Formatting

The ToString method has an optional argument that lets us provide formatting instructions.

For floating-point types, we can pass an “F” followed by the number of decimal places we want to see:

```
Console.WriteLine("The area of a circle of radius " + radius.ToString("F3") + " is " + area.ToString("F3"));
```

```
The area of a circle of radius 20.000 is 1256.637.
```

You can find a description of all of the standard numeric format strings here.

String Interpolation

There is an alternative way to embed variable values into strings that is easier to use, called string interpolation.

This line of code produces exactly the same output as our previous example:

```
Console.WriteLine($"The area of a circle of radius {radius:F3} is {area:F3}.");
```

```
The area of a circle of radius 20.000 is 1256.637.
```

Interpolation is activated by preceding a string literal with a dollar sign - '$'.

Within the interpolated string, we can include variables and C# expressions enclosed inside curly braces.

The value of the variable or the result of the expression then becomes embedded in the output string. Also notice how formatting instructions are provided.

Accepting User Input

Our program is fine if we only care about circles with radius 20. It would be more useful if we ask the user to enter the radius.

We can accomplish this by using another method of the Console class named ReadLine:

```
double radius, area;
Console.Write("Please enter a radius: ");
string sRadius = Console.ReadLine();
radius = double.Parse(sRadius);
```

First, we use another method, Console.Write. This differs from WriteLine in that it leaves the cursor at the end of the line just after the text that was written.

Console.ReadLine is explained on the next page.

Console.ReadLine()

Console.ReadLine is what is called a *blocking call*. Execution of the program freezes until the user presses <Enter>. This gives the user as much time as needed to enter the radius.

Here is the signature* of the method: `public static string ReadLine ();`

ReadLine has a return type of string. Our program captures the returned string in a variable of the same type:

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

Notice how we declare and initialize sRadius in the same line of code.

* A method's *signature* is just its declaration, without its body.

Converting a string to a double

We've seen how to convert a double to a string, but how do we go the other direction?

There's a method for that!

```
radius = double.Parse(sRadius);
```

Cool. But think about it – what if the user of our program was confused and entered "xyz" or any other text that cannot possibly represent a number?

We can answer that by reading the documentation for double.Parse. It states that if the argument "does not represent a number in a valid format", the method will generate a FormatException.

Exceptions occur when a program is asked to do something impossible.

We'll deal with this problem shortly, but first ...

area.ToString() vs. double.Parse()

We've now seen two methods related to doubles: ToString and Parse.

We called them rather differently. We called ToString using a variable: area.ToString();

We called Parse using the name of the type: double.Parse().

What is the difference? Here are the method signatures defined by the double type:

```
public string ToString (string format);
public static double Parse (string s);
```

The difference is the keyword static. We access static methods through the type name.

We access non-static (or "instance") methods through an object of that type.

Entire classes can be declared static. The Console class is a static class.
This means that all of its methods are static.

Safe Parsing

We have a string input by our program's user, and we call double.Parse. If the string cannot be parse, a runtime error will occur and our program will crash!

We could handle the exception, but in this case there is a better way. We can use another of double's static methods: TryParse. Here is its signature:

```
public static bool TryParse (string s, out double result);
```

TryParse does not throw exceptions! If parsing fails, it simply returns false. If parsing succeeds, it returns true, and it sets the *result* parameter to the parse result.

result is an out parameter. out parameters are another way that methods can return values to a caller.

Using double.TryParse

```
bool success = double.TryParse(sRadius, out radius);
if (!success)
{
    Console.WriteLine($"{sRadius} is not a number.  Sorry!");
    return;
}
```

We use TryParse as shown above. We pass *radius* as an out parameter, and we capture the method's return value in a variable named *success*.

Remember that we cannot use unassigned variables until they have been assigned. Why can we use *radius* as an out parameter?

When methods declare out parameters, they guarantee that the variable will be assigned within the method call. This allows us to use unassigned variables as out parameters.

Professional Code

We've written these examples to illustrate important points about the C# language, but we've done a few things that professional software engineers rarely do. A "cleaned up" version of our Main method is shown below.

```
Console.Write("Please enter a radius: ");
string sRadius = Console.ReadLine();
if (!double.TryParse(sRadius, out double radius))
{
    Console.WriteLine($"{sRadius} is not a number.  Sorry!");
    return;
}
double area = radius * radius * Math.PI;
Console.WriteLine($"The area of a circle of radius {radius:F3} is {area:F3}.");
```

Try to identify how this code differs from previous examples.

Best Practices

Avoid uninitialized variables. Whenever possible, initialize variables when they are declared.

If a local variable is used only once, consider removing it and replacing its use with an expression.

Declare out parameters inline when possible.

The professionalized code makes most of these changes, though astute readers may notice it still includes one local variables that is used only once.

Why?

Make it More Useful

Our AreaCalculator is looking good, but think a bit deeper about how someone might use it. Perhaps our user is a geometry student tackling an assignment. They might need to calculate just one area, but we should give them the option of calculating multiple areas.

Let's add a loop to our program to let them calculate one or more areas.

We can do this by including our code within a while loop, then making just a few minor changes:

- Modify the prompt – tell the user how to quit the application.
- Test the input for the quit condition and if true, use return to exit the application.
- In case of a parsing failure, modify the message, then use continue instead of return.

The Finished Product

```
static void Main(string[] args)
{
    while(true)
    {
        Console.WriteLine();  // Provide space to make the screen more readable.
        Console.Write("Please enter a radius, or q to quit: ");
        string sRadius = Console.ReadLine();
        if (sRadius.ToLower() == "q") return;
        if (!double.TryParse(sRadius, out double radius))
        {
            Console.WriteLine($"{sRadius} is not a number.  Please try again.");
            continue;
        }
        double area = radius * radius * Math.PI;
        Console.WriteLine($"The area of a circle of radius {radius:F3} is {area:F3}.");
    }
}
```

Topics Covered

- Unassigned Variables
- Memory Usage
- Initializing Variables
- System.Math
- Creating Informative Strings
  - Concatenation
  - Formatting
  - Interpolation
- Object Methods: ToString()
- Console.ReadLine()
- double.Parse()
- double.TryParse()
- static and instance Methods
- Professional Best Practices
- Using a while Loop

Questions

- When programs declare variables, what happens?
- What is concatenation?
- In C#, what is an Object?
- What is a blocking call?
- Do you expect that there might be methods int.Parse and int.TryParse?
- How about byte.Parse and byte.TryParse?
- What is the difference between static and non-static methods?
- What are the objectives of coding “best practices”?
- Math.PI and Math.E are not methods. What are they?
- What are class “members”?

Exercises

- Write a program that accepts a number from the user and reports the square root of that number.
  - Handle the situations where the user enters either a non-number or a negative number.
- Explore the BackgroundColor and ForegroundColor properties of the Console class. Use these properties to give your program some character.