01 Hello World

May 12, 2022

1 Hello World

1.1 What is C#?

It's a powerful and widely used programming language that you can use to make websites, games, mobile apps, desktop apps, and more. C# is part of the .NET programming platform, which includes libraries for all those applications. Learn C#, get started, and it'll give you the world.

1.2 C#: Hello World

[1]:
```
Console.WriteLine("Hello World!");
```

```
Hello World!
```

`Console.WriteLine` is a method that is used to print a message to a text console. In this case, you specified "Hello World" as the output.

2 Declare and use variables

A **variable** is a symbol you can use to run the same code with different values. For example, you can declare a new variable named `aFriend` that you can use with `Console.WriteLine` to output a string. You can declare this variable by using the type `string` or by using the `var` keyword that will automatically figure out the type for you.

> Go ahead and run the following code to see the output of **Jayme**.
>
> Next, change the name in the variable and run it again to see something different.

[2]:
```
var aFriend = "Jayme";
Console.WriteLine(aFriend);
```

```
Jayme
```

3 Combining Strings

You may have noticed that the word "Hello" was missing from the last code block. You can fix it by combining multiple string together using `+` to create a new string that it output to the console.

[3]:
```
Console.WriteLine("Hello " + aFriend + "!");
```

```
Hello Jayme!
```

4 String Interpolation

You just used `+` to build a new string from a **variable** and a **constant**. There's a better way to do this by placing the variable between `{`and `}` to tell C# to replace that text with the value of the variable. This is called **string interpolation**. You can then add a `$` before the opening quote to enable string interpolation for the string.

[4]:
```
aFriend = "string interpolation";
Console.WriteLine($"Hello {aFriend}!");
```

```
Hello string interpolation!
```

[ ]:
```

```

02 The Basics of Strings

May 12, 2022

1 The Basics of Strings

1.1 What is a String?

A string is a sequence of characters. A handy metaphor is a friendship bracelet, where you string together letters to make a name.

1.2 Strings and String Literals

`firstFriend` and `secondFriend` are variables of strings. The line within `Console.WriteLine` is also a string. It's a **string literal**. A string literal is text what represents a constant string.

Try that out with the following code. Press play and see what comes out.

Next, try changing the variables to see different names.

[1]:
```
string firstFriend = "Maria";
string secondFriend = "Sophia";
Console.WriteLine($"My friends are {firstFriend} and {secondFriend}");
```

```
My friends are Maria and Sophia
```

1.3 String Properties

As you explore more with strings, you'll find that strings are more than a collection of letters. You can find the length of a string using `Length`. `Length` is a **property** of a string and it returns the number of characters in that string.

Try that out by seeing how long the names of the friends are:

[2]:
```
Console.WriteLine($"The name {firstFriend} has {firstFriend.Length} letters.");
Console.WriteLine($"The name {secondFriend} has {secondFriend.Length} letters.
↪");
```

```
The name Maria has 5 letters.
The name Sophia has 6 letters.
```

2 String Methods

2.1 Leading and Trailing Spaces

Suppose your strings have leading or trailing spaces (also called **white space**) that you don't want to display. You want to trim the spaces from the strings. The `Trim` method and related methods `TrimStart` and `TrimEnd` do that work. You can just use those methods to remove leading and trailing spaces.

> Play around with trimming in the following code. The brackets are there to help you see all the white space.

[3]:
```
string greeting = "      Hello World!       ";
Console.WriteLine($"[{greeting}]");

string trimmedGreeting = greeting.TrimStart();
Console.WriteLine($"[{trimmedGreeting}]");

trimmedGreeting = greeting.TrimEnd();
Console.WriteLine($"[{trimmedGreeting}]");

trimmedGreeting = greeting.Trim();
Console.WriteLine($"[{trimmedGreeting}]");
```

```
[      Hello World!       ]
[Hello World!       ]
[      Hello World!]
[Hello World!]
```

2.2 Replace

You can also replace substrings with other values. For example, in the code below, you can take "Hello World!" and replace "Hello" with "Greetings", to make "Greetings World!"

> Try it out. What else could you replace "Hello" with?

[4]:
```
string sayHello = "Hello World!";
Console.WriteLine(sayHello);
sayHello = sayHello.Replace("Hello", "Greetings");
Console.WriteLine(sayHello);
```

```
Hello World!
Greetings World!
```

2.3 Changing Case

Sometimes you need your strings to be all UPPERCASE or all lowercase. `ToUpper` and `ToLower` do just that. > The following example seems a bit mixed up. Can you fix it so "whisper" is all lowercase, and "shout" is all uppercase?

[5]:
```
Console.WriteLine("WhiSPer".ToUpper());
Console.WriteLine("sHoUt".ToLower());
```

```
WHISPER
shout
```

[ ]:
```

```

03 Searching Strings

May 12, 2022

1 Searching Strings

1.1 Contains

Does your string contain another string within it? You can use `Contains` to find out! The `Contains` method returns a *boolean*. That's a type represented by the keyword `bool` that can hold two values: `true` or `false`. In this case, the method returns `true` when sought string is found, and `false` when it's not found. > Run the following code. > > What else would or wouldn't be contained? > > Does case matter? > > Can you store the return value of the `Contains` method> Remember the type of the result is a `bool`.

[1]:
```
string songLyrics = "You say goodbye, and I say hello";
Console.WriteLine(songLyrics.Contains("goodbye"));
Console.WriteLine(songLyrics.Contains("greetings"));
```

```
True
False
```

1.2 StartsWith and EndsWith

`StartsWith` and `EndsWith` are methods similar to `Contains`, but more specific. They tell you if a string starts with or ends with the string you're checking. It has the same structure as `Contains`, that is: `bigstring.StartsWith(substring)` > Now you try! > In the following code, try searching the line to see if it starts with "you" or "I". > Next, see if the code ends with "hello" or "goodbye".

[2]:
```
string songLyrics = "You say goodbye, and I say hello";
```

[ ]:
```

```

04 Numbers and Integer Math

May 12, 2022

1 Numbers and Integer Math

1.1 Integer Math

You have a few `integers` defined below. An `integer` is a positive or negative whole number. > Before you run the code, what should c be?

1.2 Addition

[2]:
```
int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

```
24
```

1.3 Subtraction

[3]:
```
int c = a - b;
Console.WriteLine(c);
```

```
12
```

1.4 Multiplication

[4]:
```
int c = a * b;
Console.WriteLine(c);
```

```
108
```

1.5 Division

[5]:
```
int c = a / b;
Console.WriteLine(c);
```

```
3
```

2 Order of operations

C# follows the order of operation when it comes to math. That is, it does multiplication and division first, then addition and subtraction. > What would the math be if C# didn't follow the order of operation, and instead just did math left to right?

[6]:
```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

```
13
```

2.1 Using parenthesis

You can also force different orders by putting parentheses around whatever you want done first

[7]:
```
int d = (a + b) * c;
Console.WriteLine(d);
```

```
18
```

You can make math as long and complicated as you want. > Can you make this line even more complicated?

[8]:
```
int d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

```
25
```

2.2 Integers: Whole numbers no matter what

Integer math will always produce integers. What that means is that even when math should result in a decimal or fraction, the answer will be truncated to a whole number. > Check it out. WHat should the answer truly be?

[9]:
```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
Console.WriteLine(d);
```

```
3
```

[ ]:
```

```

05 Numbers and Integer Precision

May 12, 2022

1 Numbers and Integer Precision

Like you learned in the last module, when doing math with integers, you only get integers as a result, no decimals or fractions. The numbers are `truncated`, which just means that the remainder is cut off. You can find the remainder with `%`, the remainder operator. The remainder is the left over amount from a division problem.

[1]:
```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

```
quotient: 3
remainder: 2
```

What is this saying? Well when you take 11 and divide it by 3, there are 3 3s that fit into 11, with two leftover, or remaining. That's why 3 is the quotient, and 2 is the remainder

1.1 Minimum and Maximum Integer Size

Because of how integers are structured in coding, there is a limit to their size.

[4]:
```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

```
The range of integers is -2147483648 to 2147483647
```

That's still a pretty big range! > But what happens if you try to go beyond?

[3]:
```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

```
An example of overflow: -2147483646
```

That number, which should be really big, is now close to the minimum! This is because an `overflow` "wraps," going back to the minimum and then continuing to count.

1.2 Doubles: Precision and Size

Doubles are another form of numbers. They can hold and answer in floating point. > Repeat the same code from the beginning, and see the difference a double makes.

[6]:
```
double a = 7;
double b = 4;
double c = 3;
double d = (a + b) / c;
Console.WriteLine(d);
```

```
3.6666666666666665
```

[7]:
```
double a = 19;
double b = 23;
double c = 8;
double d = (a + b) / c;
Console.WriteLine(d);
```

```
5.25
```

Find out the range of doubles:

[8]:
```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

```
The range of double is -1.7976931348623157E+308 to 1.7976931348623157E+308
```

That's pretty big! Much larger than integers.

Of course, doubles aren't perfect. They also have rounding errors. > Check out this rounding:

[9]:
```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

```
0.3333333333333333
```

Technically, 1/3 converted to decimal should be 3 repeating infinitely, but that isn't practical in coding. It's good to be aware of though, if you're working in extremely precise variables.

[ ]:
```

```

06 Numbers and Decimals

May 12, 2022

1 Numbers and Decimals

1.1 Working With Fixed Point types

The `Decimal` type is similar to `double`s. They don't have as big a range, but they do have much higher precision.

What's the range of a decimal type?

[1]:
```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

```
The range of the decimal type is -79228162514264337593543950335 to
79228162514264337593543950335
```

How do doubles and decimals compare in precision?

[2]:
```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

```
0.3333333333333333
0.33333333333333333333333333
```

You can see that a decimals have a higher precision than doubles.

[ ]:
```

```

07 Branches (if)

May 12, 2022

1 Branches (if)

1.1 Basic If

If statements makes decisions.

Try out our first if statement:

[1]:
```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

```
The answer is greater than 10.
```

In words, this if statement is saying "if a plus b is greater than 10, write the line 'The answer is greater than 10' ". > What would happen if a + b is less than 10? Try editing the previous code to see.

Did nothing happen? That's great! Since the the **conditions** (a + b is greater than 10) of the if statement weren't met, the code didn't go into the if statement, and therefore had nothing to print. ## What's a condition?

The `condition` is the statement in parentheses after the `if`. A `condition` is a boolean, which means it has to return a true or false. that means using symbols such as `>`, `<`, `<=`, `>=` or `==`. > Practice boolean statements. Try out some different symbols and numbers to see the answer.

[2]:
```
bool outcome = 3 > 5;
Console.WriteLine("This condition is " + outcome);
```

```
This condition is False
```

1.2 Else

Before, if the conditions of the `if` statement weren't met, the entire if statement was skipped. But what if you want something to happen in both cases? `else` is what happens if the conditional comes out false.

Run this code and change the conditional a couple times to see the different outcomes.

[3]:
```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

```
The answer is not greater than 10
```

1.3 Multi-line If statements

What if you want more complex code in your `if` statements? That's great, just add curly braces around what you want done.

Try it out! Run the following code.

[4]:
```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

```
The answer is not greater than 10
Or the first number is not equal to the second
```

`&&` means "and". It's a way to link up multiple conditionals. You can also use `||` as "or".

The `if` conditional above checks that adding a, b, and c up is greater than 10 AND that a equals b. If both are true, it goes into the `if` statement; otherwise, it goes into the `else` part.

[ ]:
```

```

08 What Are Loops?

May 12, 2022

1 What Are Loops?

Loops are a way to repeat an action multiple times. You can use `while` to do that. > Try out this first loop:

[1]:
```
int counter = 0;
while (counter < 10)
{
  Console.WriteLine($"Hello World! The counter is {counter}");
  counter++;
}
```

```
Hello World! The counter is 0
Hello World! The counter is 1
Hello World! The counter is 2
Hello World! The counter is 3
Hello World! The counter is 4
Hello World! The counter is 5
Hello World! The counter is 6
Hello World! The counter is 7
Hello World! The counter is 8
Hello World! The counter is 9
```

So what happened here? `while` checks if the condition ('counter < 10') is true. If it is, it goes through the loop. Then it checks it again. It will keep going through the loop until the condition is false. When figuring out branches and loops, it can be helpful to put code into human language. This code is saying: "Set up counter to equal 0. While the counter is less than 10, print 'Hello World! the counter is {counter}' and then increase the counter by 1".

1.1 ++

`++` is a quick quick way to add one to a variable. You can also do the same with `--`, which subtracts one from the variable.

1.2 Infinite loops

It's easy to accidentally make an infinite loop. **Editors note: You can't tell them to create an infinite loop, because the notebook doesn't ever stop itself** If you hit an infinite loop, you have two options: you can always exit out of your program, or you can hit CTRL+C.

> Challenge: You can't show one on this notebook, because the code doesn't catch itself, but as a challenge, try making an infinite loop in Visual Studio! What are the different ways you might accidentally make it infinite? It can be scary to make a mistake, but always know you can CTRL+C or exit the program.

1.3 do

You can also make a loop with `do`. > Try out the following code and see if there's anything different between this and the `while` loop.

[3]:
```
int counter = 0;
do
{
  Console.WriteLine($"Hello World! The counter is {counter}");
  counter++;
} while (counter < 10);
```

```
Hello World! The counter is 0
Hello World! The counter is 1
Hello World! The counter is 2
Hello World! The counter is 3
Hello World! The counter is 4
Hello World! The counter is 5
Hello World! The counter is 6
Hello World! The counter is 7
Hello World! The counter is 8
Hello World! The counter is 9
```

Hmm, nothing much seems to different, right?

> Try changing the conditional to `(counter < 0)` on both codes. What happens?

In the `while` loop, nothing is printed out, but in the `do` loop, you get one print out. The `do` loop does the action first, then checks the conditional. In contrast, the `while` loop checks the conditional first, then does an action.

1.4 for loop

`for` is one the most common loops. It will repeat an action for a specific amount of turns. It can look a little intimidating. > Before you learn all the ins and outs of the code, run the following for loop just to see it working:

[4]:
```
for (int counter = 0; counter < 10; counter++)
{
  Console.WriteLine($"Hello World! The counter is {counter}");
}
```

```
Hello World! The counter is 0
Hello World! The counter is 1
Hello World! The counter is 2
```

```
Hello World! The counter is 3
Hello World! The counter is 4
Hello World! The counter is 5
Hello World! The counter is 6
Hello World! The counter is 7
Hello World! The counter is 8
Hello World! The counter is 9
```

1.4.1 What's in the parentheses?

There are three parts in the `for` loop parentheses: `int counter = 0`, `counter < 10`, and `counter++`. You can think of them as the start point, the end point, and the step size. `int counter = 0` is just setting up the counter. You're starting at 0. `counter < 10` is the conditional that gets checked at the start of every loop. Once the conditional is false (in this case, once counter is not less than 10), the loop is done and the code moves on. `counter++` is increasing counter by one, taking one step closer to the end. The step is taken at the end of each loop. > Try messing with the `for` loop set up. How does it change?

[ ]:

09 Combining Branches and Loops

May 12, 2022

1 Combining Branches and Loops

1.1 Tips and tricks

- The `%` operator gives you the remainder of a division operation.
- The `if` statement gives you the condition to see if a number should be part of the sum.
- The `for` loop can help you repeat a series of steps for all the numbers 1 through 20.
- Getting a weird answer? try making some print statements of your variables, to see it they're changing unexpectedly.

[ ]:

10 Arrays, Lists, and Collections

May 12, 2022

1 Arrays, Lists, and Collections

Arrays, lists, and collections can be pretty useful. Try looking at a list: > Run the following code. Does it print out what you expected?

[1]:
```
using System;
using System.Collections.Generic;

var names = new List<string> { "<name>", "Ana", "Felipe" };
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

```
Hello <NAME>!
Hello ANA!
Hello FELIPE!
```

1.1 So what does that code mean?

- **System.collections.Generic**: This is a namespace that has lists in it. If you don't tell the code that you're using it, you have to write "Systems.Collections.Generic.List" every time you want to use a list. This saves some typing!
- **var**: It's what you put when you have a variable, but don't know/care what the variable type is.
- **List<string>**: This means that you're making a list of strings. In place of `string`, you can put in `int`, `double`, or any other variable.
- **foreach**: This is another for loop! It goes through every item in a list.
- **name in names**: This is a style that a lot of people prefer. `names` is the whole list that contains plural names. `name` is an individual item in `names`.

1.2 Alternative method

The previous code is a bit more clear to read for human than the code below, but the code below has some more recognizable code, based off of what we've learned. These are really just two different styles of writing the same code. Feel free to use whatever makes the most sense to you! > Run the following code. > > Can you identify similar parts of code between the two different methods? > > Which method do you prefer?

[2]:
```
using System;
using System.Collections.Generic;

var names = new List<string> { "<name>", "Ana", "Felipe" };
for (int i = 0;i < names.Count; i++)
{
    Console.WriteLine($"Hello {names[i].ToUpper()}");
}
```

```
Hello <NAME>
Hello ANA
Hello FELIPE
```

1.3 Add

You can add names to lists pretty easily. Lists have the method `Add()`, which tacks on a new item to the end of the list. > Run the code. > > Then try adding your own name instead.

[3]:
```
var names = new List<string> { "<name>", "Ana", "Felipe" };

names.Add("Sophia");

foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

```
Hello <NAME>!
Hello ANA!
Hello FELIPE!
Hello SOPHIA!
```

1.4 Remove

You can also remove names. Try that out:

Run the code.

Then choose which name you want removed.

What happens when you try to remove something that isn't there?

[4]:
```
var names = new List<string> { "<name>", "Ana", "Felipe" };

names.Remove("<name>");

for (int i = 0;i < names.Count; i++)
{
    Console.WriteLine($"Hello {names[i].ToUpper()}");
}
```

```
Hello ANA
Hello FELIPE
```

1.5 Printing a specific item

What if you don't want to print out all of your friends? What if you just want to print out one friend? That's where brackets come in. > Run the code. > > Try printing a different spot in the list. > > Do you need a 0 or 1 to print the first item in a list?

[5]:
```
var names = new List<string> { "<name>", "Sophia", "Felipe" };
Console.WriteLine(names[1]);
```

```
Sophia
```

Don't forget that lists are "0" based. The first spot is the "0th" spot.

[ ]:
```

```

11 Search, Sort, and Index Lists

May 12, 2022

1 Search, Sort, and Index Lists

1.1 Search

In the last notebook, you learned how to find what item was stored at a specific index. Now, given an item, find out its index.

Run the code

What index is Sophia at?

What index is "Scott" at?

[1]:
```
using System;
using System.Collections.Generic;
var names = new List<string> { "Sophia", "Ana", "Jayme", "Bill" };
string name = "Ana";
var index = names.IndexOf(name);
Console.WriteLine($"Found {name} at {index}");
```

```
Found Ana at 1
```

1.2 What does -1 mean?

If `IndexOf()` returns -1, then that means it couldn't find the item in the list. In fact, you can make a little if statement that works in not finding the item:

Run the code.

Try out a few different names.

[2]:
```
var names = new List<string> { "Sophia", "Ana", "Jayme", "Bill" };
string name = "Scott";
var index = names.IndexOf(name);
if(index == -1){
    Console.WriteLine($"{name} not found");
} else {
    Console.WriteLine($"Found {names[index]} at {index}");
}
```

```
Scott not found
```

1.3 Sort

Until now, you've just been putting in names in a random order. But sometimes it's nice to have a list be sorted. `Sort()` takes a list and organizes it. It looks at the variable types and organizes in the most reasonable way it can see - if it's strings, it sorts alphabetically, if it's numbers it organizes from smallest to largest.

Note that you don't need to write `sortedList = names.Sort()`, you just have to write `names.Sort()`. `Sort()` changes the list itself and you don't have to save the action to a new object.

Run the code!

Feel free to add in any other names to see them get sorted.

[3]:
```
var names = new List<string> { "Sophia", "Ana", "Jayme", "Bill" };
Console.WriteLine("Pre Sorting:");
foreach(var name in names )
{
    Console.WriteLine(name);
}

names.Sort();

Console.WriteLine();
Console.WriteLine("Post Sorting:");
foreach(var name in names )
{
    Console.WriteLine(name);
}
```

```
Pre Sorting:
Sophia
Ana
Jayme
Bill

Post Sorting:
Ana
Bill
Jayme
Sophia
```

[ ]:

12 Lists of Other Types

May 12, 2022

1 Lists of Other Types

You've been practicing lists of strings, but you can make a list of anything! Here's a number example.

1.1 Fibonacci

Fibonacci is a cool number sequence. It adds up the last two numbers up to make the next number. You start with 1 and 1 $1 + 1 = 2$ (1, 1, 2) $1 + 2 = 3$ (1, 1, 2, 3) $2 + 3 = 5$ (1, 1, 2, 3, 5) $3 + 5 = 8$ (1, 1, 2, 3, 5, 8) and so on. There are lots of stuff in nature that follow this number sequence, and has lots of cool stuff if you want to look it up!

> Start with the base numbers: Here's a list with just 1, 1 in it. Run it and see what happens.

[1]:
```
var fibonacciNumbers = new List<int> {1, 1};

foreach (var item in fibonacciNumbers)
    Console.WriteLine(item);
```

```
1
1
```

Now, you don't want just 1,1 in it! You want more of the sequence. In this code, you're using the last two numbers of the list, adding them together to make the next number, then adding it to the list.

> Run the code to try it out.

[3]:
```
var fibonacciNumbers = new List<int> {1, 1}; // Starting the list off with the␣
 ↪basics

var previous = fibonacciNumbers[fibonacciNumbers.Count - 1]; // Take the last␣
 ↪number in the list
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2]; // Take the␣
 ↪second to last number in the list

fibonacciNumbers.Add(previous + previous2); // Add the previous numbers␣
 ↪together, and attach the sum to the end of the list
```

```
foreach (var item in fibonacciNumbers) // Print out the list
    Console.WriteLine(item);
```

```
1
1
2
```

1.2 Count -1

Why do you need to do `fibonacciNumbers.Count -1` to get the last number of the list? Well, `Count` tells you how many items are in a list. However, the index of an item starts at zero. So, if you only had one item in your list, the count would be one, but the index of the item would be 0. The index and count of the last item is always one off.

2 Challenge: Fibonacci to 20th number

We've given you a base of code that deals with Fibonacci. Can you make a list that has the first 20 fibonacci numbers?

> Make and print a list that has the first 20 fibonacci numbers.

[4]:
```
Console.WriteLine("Challenge");
```

```
Challenge
```

2.1 Tips and tricks

- The final number should be 6765.
- Could you make a `for` loop? A `foreach` loop? A `while` loop? Which kind of loop do you prefer and which would be more useful?
- Are you getting close, but are you one number off? That's a really common issue! Remember that `>` and `>=` are similar, but they end up being one off from the other. Try playing around with that?
- Remember that you're starting with two items in the list already.

[ ]:
```

```

13 Objects and Classes

May 12, 2022

1 Objects and Classes

1.1 Object Oriented Programming

Objects are a way to mimic the real world in coding. If you take the concept of a person, they can have a name, address, height, all of these properties that change from person to person. Object oriented coding packages that type of information, so that you can easily make a person with all those details. There are lots of stuff you can do with objects, but for now you can start with the basics.

1.2 Making a Bank

Over the next few modules, you'll make a bank account object with these attributes:

It has a 10-digit number that uniquely identifies the bank account.

It has a string that stores the name or names of the owners.

The balance can be retrieved.

It accepts deposits.

It accepts withdrawals.

The initial balance must be positive.

Withdrawals cannot result in a negative balance.

You can categorize these goals:

- **Properties**: details about the object (how much money it has, the name of the account).
- **Actions**: things the object can do (accept deposits and withdrawals).
- **Rules**: guidelines for the object so that it doesn't try to do impossible things (make sure that the account can never go negative).

1.3 Make it yourself

Below is a blank `BankAccount` object that you're going to create. You'll add in code step by step.

1.4 #1: Properties

Properties are a nice little list of values each object holds. `get` and `set`: sometimes you want to only want a user to see a variable but not change it. Other times you want the user to be able to change a variable. `get` lets you see the variable, `set` lets you change it. (right?)

Copy the code below and paste it into the `BankAccount` object under `//Properties`

```
public string Number { get; }
public string Owner { get; set; }
public decimal Balance { get; }
```

[3]:
```
public class BankAccount
{
    // Properties (paste under here)

    // Constructor

    // Functions
}
```

1.5 #2: Constructor

This method is what creates a specific instance of an object. Making a `BankAccount` class, like you're doing now, is like making a template for all bank accounts. It's not a singular individual account. The constructor is what will make a singular account, with all the persons actual details. You give the constructor all the details you want for a specific account, and it assigns the details to the new object's properties.

`this` is a styling choice. It makes explicit that the variable "Owner" is the variable of that specific instance. In the future, you'll have two instances of an object to interact, and `this` will become a bit more explicitly helpful. You can also write `Owner` instead of `this.Owner` if you want!

You're taking the variables `name` and `initialBalance` and creating a bank account that contains these variables.

Copy paste the constructor into the `BankAccount` below, under `//Constructor`

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

[4]:
```
public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    // Constructor (Paste here!)

    // Functions
}
```

1.6 #3: Making an Instance

Now that you have the code written out, see what happens if you make a `BankAccount`!

> Run the two code cells code below to create a specific bank account. Does it do what you expected?
>
> Change the code to make a bank account for yourself. How much money do you want in your bank account?

[5]:
```
public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Balance = initialBalance;
    }
    // Functions
}
```

[6]:
```
var account = new BankAccount("Kendra", 1000);
Console.WriteLine($"Account{account.Number} was created for {account.Owner}␣
 ↪with {account.Balance} dollars");
```

```
Account was created for Kendra with 1000 dollars
```

1.7 What about the `account.Number`?

You may have noticed that the code didn't print out anything for `account.Number`. That's okay! You haven't put anything in it yet. You'll learn about it in the next module.

1.8 #4: Functions

Functions exist to do actions with an object or change the object variables. These two functions will make a deposit (add money) and make a withdrawal (take out money). You'll be adding stuff in the methods later, but for now you just want to add the empty versions.

> copy the functions below and add them to `BankAccount` under `//Functions`

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
```

```
{
}
```

[7]:
```
public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Balance = initialBalance;
    }

    // Functions (paste here!)
}
```

2 Review

Here's the version of `BankAccount` you end up with in this module. You'll be adding more to it in the next module, but why don't you try out stuff, just to see what you need to learn?

> Can you add a 10-digit code? What would your object need to know to make sure the code was unique?
>
> Try out adding to the deposit function! What do you want it to do?
>
> How might you check that the initial balance is positive?

[8]:
```
public class BankAccount
{
    // Variables (#1)
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    // Constructor (#2)
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Balance = initialBalance;
    }

    // Functions (#4)
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
```

```
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

[9]:
```
//Make an instance (#3)
var account = new BankAccount("Kendra", 1000);
Console.WriteLine($"Account{account.Number} was created for {account.Owner}␣
 ↪with {account.Balance} dollars");
```

```
Account was created for Kendra with 1000 dollars
```

[ ]:
```

```

14 Methods and Members

May 12, 2022

1 Methods and Members

Here's your bank account so far! It doesn't do much right now, only prints out the owner and balance. It doesn't even have an account number yet. You'll work on a transaction class, which has been added as an empty class for you.

> Run each code chunk below and see what gets printed. This is what you ended up with last time.

[2]:
```
public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Balance = initialBalance;
    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

[3]:
```
var account = new BankAccount("Kendra", 1000);
Console.WriteLine($"Account{account.Number} was created for {account.Owner}␣
↪with {account.Balance} dollars");
```

```
Account was created for Kendra with 1000 dollars
```

1.1 #1: Account Number

You need a starting number, that you can base the new account numbers off of, to insure that all the accounts are unique. Below is the code for this number "seed". What does it mean?

- **Private**: This means that no client can see this number. It's internal, part of the internal workings of the code.
- **Static**: This mean the number is universal amongst all individual accounts. If one account changes it, then that number is updated for all the other accounts. This is how you can make it a great way to make sure the account numbers are all unique! Once an bank account uses it for it's bank number, it can add one to the account seed, and the next new bank account has a new number.

  Copy the code below and paste it in the `// Properties` section of the `BankAccount` class.

```
private static int accountNumberSeed = 1234567890;
```

Copy this next bit of code and add it to the constructor.

```
this.Number = accountNumberSeed.ToString();
accountNumberSeed++;
```

Run this code and see what happens!

[4]:
```
public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }
    //(Paste first bit here!)

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Balance = initialBalance;
        //(Paste second part here!)

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

[5]:
```
var account = new BankAccount("Kendra", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}␣
↪with {account.Balance} dollars");
```

```
Account  was created for Kendra with 1000 dollars
```

1.2 #2: Transaction Properties

The next part you need is a balance! One way you could do this is just keep a running tab. However, another way to do it is to create a history of transactions. To do that, you're going to make a little transaction class, that records one transaction.

paste the properties below into the class `Transaction`

```
public decimal Amount { get; }
public DateTime Date { get; }
public string Notes { get; }
```

[6]:
```
public class Transaction
{
    // Properties (Paste here!)

    // Constructor
}
```

1.3 #3: Transaction Constructor

Next, you need to add the constructor to the class!

Add the following code to the `Transaction` class, under constructor.

```
public Transaction(decimal amount, DateTime date, string note)
{
    this.Amount = amount;
    this.Date = date;
    this.Notes = note;
}
```

[7]:
```
public class Transaction
{
    // Properties
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes { get; }

    // Constructor (Paste here!)
}
```

1.4 #4: Update BankAccount to match

Now that you have a transaction class, you can use that in our bank account. First, you need to make a list of transactions.

Copy the following code into the Properties section.

```
private List<Transaction> allTransactions = new List<Transaction>();
```

[8]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance{ get;}
    private static int accountNumberSeed = 1234567890;
    //(Paste here!)

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Balance = initialBalance;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

1.5 #5: Updating Balance

Now that you have a list of transactions that you can use, you need to attach `Balance` to that. What you want to do is, whenever someone wants to get balance, the code checks the list of transactions and tallies it all up, before returning the answer. You can do this by attaching some instructions to the `get` in Balance!

In `BankAccount`, replace `public decimal Balance { get;}` with the following code:

```
public decimal Balance
```

```
{
    get
    {
        decimal balance = 0;
        foreach (var item in allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

[9]:
```
public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }  // replace this line!
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Balance = initialBalance;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;
    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

1.6 #6: Fixing errors

You may have noticed a red squiggly line under `this.Balance`. There's a new error you created! Because whenever you're getting Balance, it goes through a process of summing up the list of transactions, you can't just say that `Balance` is initial balance. You won't fix this entirely in this module, but you can make the code usable for now.

Remove the line `this.Balance = initialBalance`.

[11]:
```
public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get
        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }
    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();


    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        //this.Balance = initialBalance; //delete this line
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;
    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

2 Review: Where did Initial Balance go?

Here's our final code for this module below. There's a problem though! You no longer have an initial balance and have 0 money! Since you tied up your balance with transactions, you're gonna

need to be able to make deposits and withdrawals to put money in the bank. You'll learn that in the next module!

Run the code cells below.

Try making your own transaction methods before the next module! Where are you getting stuck? What do you need to learn?

[12]:
```
public class Transaction
{
    // Properties (#2)
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes
    {
        get;

    }

    // Constructor (#3)
    public Transaction(decimal amount, DateTime date, string note)
    {
        this.Amount = amount;
        this.Date = date;
        this.Notes = note;
    }
}
```

[13]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance  //(#5)
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }
```

```
    }
    private static int accountNumberSeed = 1234567890; //(#1)
    private List<Transaction> allTransactions = new List<Transaction>(); //(#4)


    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        //(#6: deleted "this.Balance = initialBalance;")
        this.Number = accountNumberSeed.ToString(); //(#1)
        accountNumberSeed++; //(#1)

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

[14]:
```
var account = new BankAccount("Kendra", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}␣
↪with {account.Balance} dollars");
```

```
Account 1234567890 was created for Kendra with 0 dollars
```

[ ]:
```

```

15 Methods and Exceptions

May 12, 2022

1 Methods and Exceptions

Below is the code you've made so far. Balance is gotten by summing up the lists of transactions, but you haven't written a way to add a transaction. This happens plenty of times in coding, where to make something more robust, you have to take a step back before going forward.

Run the code cells below.

[1]:
```
public class Transaction
{
    // Properties
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes
    {
        get;

    }

    // Constructor
    public Transaction(decimal amount, DateTime date, string note)
    {
        this.Amount = amount;
        this.Date = date;
        this.Notes = note;
    }
}
```

[2]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get
```

```
        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }

    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

[3]:
```
// Testing Code

var account = new BankAccount("Kendra", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}␣
 ↪with {account.Balance} dollars");
```

```
Account 1234567890 was created for Kendra with 0 dollars
```

1.1 #1: Adding deposits

First things first, time to make a deposit function. This addition will make a transaction listing the amount, date, and a note that you're depositing, and then adds it to the transaction list.

Add this code inside MakeDeposit.

```
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
```

[5]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }


    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        //Add code here!
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
```

```
}
```

1.2 #2: Exceptions

But what if someone tries to deposit negative money? That doesn't make logical sense, but currently the method allows for that. What you can do is make an exception. Before doing anything, you check that the amount deposited is more than 0. If it is, awesome, the code moves on to adding the transaction. If not, the code throws an exception, where it stops the code and prints out the issue.

Place this code in the very beginning of the `MakeDeposit` method.

```
if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit must be positi
    }
```

[6]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }

    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;
```

```
    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        //Add Code here!

        var deposit = new Transaction(amount, date, note);
        allTransactions.Add(deposit);
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

1.3 #3: Adding withdrawal

Now you need to do the same thing for the withdrawal!

Add this code to MakeWithdrawal.

```
if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be pos
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
```

[7]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
```

```
            {
                balance += item.Amount;
            }

            return balance;
        }


    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
 ↪deposit must be positive");
        }
        var deposit = new Transaction(amount, date, note);
        allTransactions.Add(deposit);
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
        //Add code here!
    }
}
```

1.4 #4: Creating initial deposit

Now that you have deposits and withdrawals, you can finally make an initial deposit again. What you'll do is create a deposit of the initial amount when you're first constructing the bank account.

Add this code to the `BankAccount` constructor.

```
MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
```

[8]:
```
using System.Collections.Generic;

public class BankAccount
{
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }


    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {
        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;
        //(Paste here!)

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
 ↪deposit must be positive");
        }
        var deposit = new Transaction(amount, date, note);
        allTransactions.Add(deposit);
    }
```

```
    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
 ↪withdrawal must be positive");
        }
        if (Balance - amount < 0)
        {
            throw new InvalidOperationException("Not sufficient funds for this␣
 ↪withdrawal");
        }
        var withdrawal = new Transaction(-amount, date, note);
        allTransactions.Add(withdrawal);
        }
}
```

2 Check and test your work

There's some added line in the test code, because you can now make deposits and withdrawals. Test it out!

Run the following cells, including the new stuff in the test code.

Make your own deposit and withdrawal.

[9]:
```
public class Transaction
{
    // Properties
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes { get; }

    // Constructor
    public Transaction(decimal amount, DateTime date, string note)
    {
        this.Amount = amount;
        this.Date = date;
        this.Notes = note;
    }
}
```

[10]:
```
using System.Collections.Generic;

public class BankAccount
{
```

```
    // Properties
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance
    {
        get

        {
            decimal balance = 0;
            foreach (var item in allTransactions)
            {
                balance += item.Amount;
            }

            return balance;
        }


    }
    private static int accountNumberSeed = 1234567890;
    private List<Transaction> allTransactions = new List<Transaction>();

    // Constructor
    public BankAccount(string name, decimal initialBalance)
    {

        this.Owner = name;
        this.Number = accountNumberSeed.ToString();
        accountNumberSeed++;
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance"); //(#4)

    }

    // Functions
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        //(#2)
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
↪deposit must be positive");
        }
        //(#1)
        var deposit = new Transaction(amount, date, note);
        allTransactions.Add(deposit);
    }
```

```
    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
        //(#3)
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of␣
 ↪withdrawal must be positive");
        }
        if (Balance - amount < 0)
        {
            throw new InvalidOperationException("Not sufficient funds for this␣
 ↪withdrawal");
        }
        var withdrawal = new Transaction(-amount, date, note);
        allTransactions.Add(withdrawal);
    }
}
```

[11]:
```
var account = new BankAccount("Kendra", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}␣
 ↪with {account.Balance} dollars");

account.MakeWithdrawal(500, DateTime.Now, "Rent payment");  //Added test code
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

```
Account 1234567890 was created for Kendra with 1000 dollars
500
600
```

3 Review

You did it! You've now successfully made a bank account class that has the following attributes:

It has a 10-digit number that uniquely identifies the bank account.

It has a string that stores the name or names of the owners.

The balance can be retrieved.

It accepts deposits.

It accepts withdrawals.

The initial balance must be positive.

Withdrawals cannot result in a negative balance.

[ ]: