



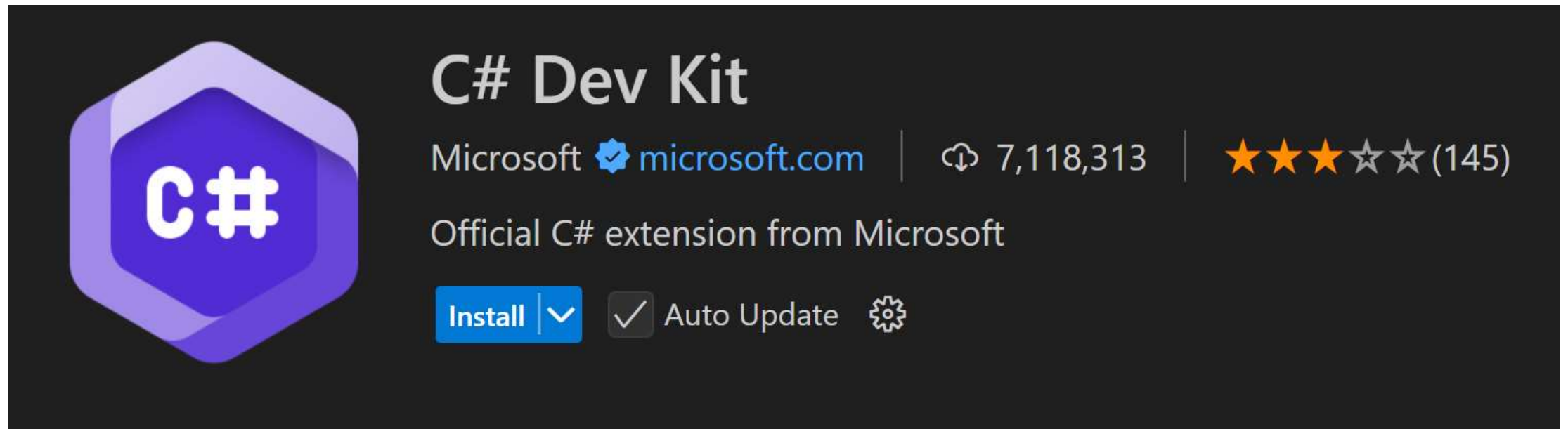
C# Programming Language

Stephen Marz
COSC365

Quick Actions




- <https://learn.microsoft.com/en-us/dotnet/csharp/>
- Language: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/>
- API: <https://learn.microsoft.com/en-us/dotnet/api/>
- Async/await: <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/>
- Download: <https://dotnet.microsoft.com/download>
- Playground: <https://dotnet.microsoft.com/en-us/platform/try-dotnet>

Development Plugin (VSCode)





The image shows a dark-themed card for the 'C# Dev Kit' extension in VS Code. On the left is a purple hexagonal icon with 'C#' in white. To the right, the title 'C# Dev Kit' is in large white font. Below it, 'Microsoft' is followed by a verified publisher icon and 'microsoft.com'. Further right is a download icon and '7,118,313'. To the right of that is a star rating of 4.5 stars (3 orange, 2 grey) and '(145)'. Below the title, it says 'Official C# extension from Microsoft'. At the bottom, there is a blue 'Install' button with a dropdown arrow, a checked 'Auto Update' checkbox, and a settings gear icon.

C# Dev Kit

Microsoft  microsoft.com |  7,118,313 |  (145)

Official C# extension from Microsoft

[Install](#)  ☒ Auto Update 

C#

- C# is:
 - imperative
 - procedural
 - compiled
 - object oriented
 - polymorphic
 - modular
 - strongly typed
 - statically typed
 - garbage collected
 - widely used in both government and industry

Imperative Programming

- A series of instructions to change the state of the machine (e.g., memory, registers, I/O).
- Examples: C, C++, C#, Java

Procedural Programming

- A series of **procedures** (e.g., functions, lambdas, etc) that change state.
- Procedural programming is a subset of **imperative programming**.

Compiled

- The language is transformed into machine code or intermediate code before running.
- Ahead-of-time (AoT) compilation – what you're used to with C, C++. The code is compiled to **native** instructions before it is run.
- Just-in-time (JIT) compilation – translation from a **bytecode** into **native instructions**.
 - Interpreted languages, such as Python, are not JIT.
 - C# is JIT when using its intermediate language (MSIL).

Object Oriented

- A style of programming that links data (fields) and actions (methods) in a set of memory called **objects**.
- An object is a **blueprint** until it is **instantiated**.
 - Instantiation – allocating memory to store an object.

Polymorphic

- A style of object-oriented programming allowing traits of one object to be derived to another.
- Example:
 - Animal class
 - Animals can run, walk, eat
 - Tiger is an Animal
 - Don't have to reimplement run, walk, eat

Modular

- Sections of code can be substituted.
- C/C++ does this with **headers** and **libraries**.
- C# takes this even farther with **partial** class definitions.
 - Classes can defined spanning multiple C# source code files.

Strongly Typed

- Data must be explicitly converted.
 - Except for coercion (i.e., auto-casting)
- C# does allow for **unsafe** typing, but this puts the onus on the programmer.

Statically Typed

- C#'s types are checked at run time.
- This differs from languages, such as Python, where the type is determined at **run time**.
 - Most progressively interpreted languages are dynamically typed.

Garbage Collected

- Heap memory is automatically freed by the C# runtime.
 - Therefore, there is no **delete** keyword.

Naming Conventions

- Interface names start with a capital I.
- Identifiers shouldn't contain two consecutive underscore (__) characters.
 - Those names are reserved for compiler-generated identifiers.
- Use PascalCase for class names and method names.
- Use camelCase for method parameters and local variables.
- Use PascalCase for constant names, both fields and local constants.
- Private instance fields start with an underscore (_) and the remaining text is camelCased.

.NET

- .NET is a framework of many classes that run on the CLR
- CLI – Common Language Infrastructure
 - Many languages (C#, C++, and Visual Basic) run on the same virtual machine
- CLR – Common Language Runtime
 - This is C#'s virtual machine that interprets MSIL
- MSIL – Microsoft Intermediate Language
 - C#'s bytecode
 - When you compile C#, it transforms it into bytecode, not native assembly

C# Compiler

- On hydra/tesla: "mcs"
 - -langversion:7.2
 - -out:myprogram.exe
 - `mcs -langversion:7.2 -out:myprogram.exe myprogram.cs`
- Creates a .exe file
 - Called a "portable executable" or PE for short.
- Must use "mono" to run .exe files:
 - `mono myprogram.exe`

C# Compiler and Runtime

- C# compiles into MSIL
 - Microsoft Intermediate Language
 - MS's "bytecode"
 - Linker takes MSIL and generates a "portable executable" (PE)
- C# runs on top of .NET
 - .NET is a virtual machine much like Java's VM
 - We use the "mono" virtual machine on Hydra/Tesla machines

Libraries

- Libraries are added in the compiler command
 - Libraries are called **assemblies**.
 - Adding libraries is called linking an assembly.
- System library is automatically linked.
 - Many things under the System library
 - using System;
 - Subclasses are not necessarily in the same assembly.
 - using System.Collections;
 - Is NOT in
 - using System;

Entry Point

- Just like Java: C#'s Main() must be in a class.
 - `public static void Main(string[] args)`
- Notice the capitalization of Main and string[].
 - This differs from Java's convention.
 - Java: `public static void main(String[] args)`

Primitives

- Primitives: int, char, byte, long, string, uint (unsigned), ulong, float, double
- Objective Primitives: Int16, Int32, Int64, UInt16, UInt32, UInt64
- Arrays: int[], char[], long[], string[]
- Structs are stack-based values
- Classes are heap-based objects
 - Allocate a class using the “new” operator
 - There is NO delete
 - Uses garbage collection (GC)
- Pointers function much the same way as C++
 - `char *character_pointer;`
 - `int *integer_pointer;`
 - Pointers should be avoided in C#

Arrays

- Arrays are allocated on the heap
- `double[] many_doubles = new double[50];`
 - Notice the data type itself has empty brackets []
 - Otherwise, the new operator functions much like C++
- `int[] many_ints = new int[] {1, 2, 3, 4, 5, 6, 7};`
 - This form creates a new integer array with 7 elements
- `int[] ref_to_many_ints = many_ints;`
 - In this case, `ref_to_many_ints` and `many_ints` point to the same memory location

Operators

- Identical to C++
 - + - * / % ++ --
 - == != >= <= < >
 - && || !
 - & ^ | ~ << >>
 - = += -= *= /= %= <<= >>= &= ^= |=
 - sizeof() typeof()

Overloading

- Many operators can be overloaded (defined by the programmer)
 - Fewer than C++ however
 - All operators are static and first argument is class object
- Overloadable operators:
 - (unary and binary) +, -, !, ~, ++, --, true false
 - ==, !=, <, >, >=, <=
 - &, |, ^, <<, >>
- Non-overloadable:
 - &&, ||, [] (use indexers instead)
 - +=, -=, /=, %=, |=, ^=, >>=, <<=, and essentially any other combination operator

Flow Control

- C# uses many of the same statements as C++
 - if
 - if...else if
 - if...else
 - switch
 - Ternary operator: ?

Auto-Derived Data

- var keyword

```
var a = 1234;    // a is an integer due to the literal 1234  
var b = 1.2;     // b is a double due to the literal 1.2
```

Converting and Casting

- Casting is like C
 - `var new_data = (type)data;`
- Class polymorphism uses as keyword
 - This is covered under the object-oriented section.
- Converting data types can use the Convert class
 - <https://learn.microsoft.com/en-us/dotnet/api/system.convert>
- Converting from string can use the Parse() functions.

Interpolated Strings

```
string hello = "Hello";  
string value = $"{hello} world.";   
Console.WriteLine(value);
```

```
for (var i = 0; i < 5; i+=1) {  
    Console.WriteLine($"Line {i + 1}");  
}
```

Formatted Strings

- `${i, 10}"`
 - Prints the variable `i` in a right-justified field, 10 characters wide
- `${i, -10}"`
 - Prints the variable `i` in a left-justified field, 10 characters wide
- `${dbl, 6:F2}"`
 - Prints the variable `dbl` in a right-justified field, 6 characters wide, in fixed notation with two digits of precision.

String Formats

- E - scientific notation
- *Fdigits* - fixed notation with *digits* number of digits right of the decimal.
- N - number (puts in commas and decimal points)
- P - percent
- X - hexadecimal
 - Capital X -- all digits capitalized
 - Lowercase x -- all digits lowercase
 - Does not print leading 0x

String Class

<https://learn.microsoft.com/en-us/dotnet/api/system.string>

<code>String(Char, Int32)</code>	Initializes a new instance of the <code>String</code> class to the value indicated by a specified Unicode character repeated a specified number of times.
<code>String(Char[], Int32, Int32)</code>	Initializes a new instance of the <code>String</code> class to the value indicated by an array of Unicode characters, a starting character position within that array, and a length.
<code>String(Char[])</code>	Initializes a new instance of the <code>String</code> class to the Unicode characters indicated in the specified character array.
<code>String(Char*, Int32, Int32)</code>	Initializes a new instance of the <code>String</code> class to the value indicated by a specified pointer to an array of Unicode characters, a starting character position within that array, and a length.
<code>String(Char*)</code>	Initializes a new instance of the <code>String</code> class to the value indicated by a specified pointer to an array of Unicode characters.

Handling Null

```
string fileName = Console.ReadLine()?.Trim() ?? string.Empty;
```

<https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#null-conditional-operators--and->

Console.ReadLine()

- Reads a line of input from the console and returns it as a string.
- If the user enters nothing and just presses Enter, it will return an empty string (""), not null.
- ?.Trim()
 - The null conditional operator (?.) ensures that the Trim method is only called if Console.ReadLine() does not return null.
 - If Console.ReadLine() is null, the entire expression Console.ReadLine()?.Trim() will evaluate to null, and Trim will not be called, avoiding a NullReferenceException.
- ?? string.Empty
 - The null coalescing operator (??) provides a default value if the left-hand side is null.
 - If Console.ReadLine()?.Trim() evaluates to null (because Console.ReadLine() returned null), then fileName will be assigned an empty string (string.Empty).

Nullable Value Types

- Value types do not hold null by default.
 - <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types>
- The ? syntax when declaring tells C# to also let the value type hold null.

```
double? pi = 3.14;  
char? letter = 'a';
```

```
int m2 = 10;  
int? m = m2;
```

```
bool? flag = null;
```

```
// An array of a nullable value type:  
int?[] arr = new int?[10];
```


Data Conversions

- System.Convert
 - <https://learn.microsoft.com/en-us/dotnet/api/system.convert?view=net-8.0>

```
var result = Convert.ToBoolean(value);
```

```
var result = Convert.ToInt32(7.95);
```

```
var result = Convert.ToSingle(7);
```

Built In String Parsing

- System.String
 - <https://learn.microsoft.com/en-us/dotnet/api/system.string?view=net-8.0>
 - Strings can be converted to int/float by using Parse

```
int value = int.Parse("123");
```

```
double value = double.Parse("7.59");
```

```
string input = "Hello world!";
```

```
char[] string_chars = input.ToCharArray(0, input.Length);
```

Console

- `Console.Write("Something");`
 - Writes the string to standard out.
- `Console.WriteLine("Something");`
 - Writes the string and prints a newline at the end to standard out.
- `Console.Read()`
 - Reads a single character from standard in as a char.
- `Console.ReadLine()`
 - Reads a line from standard in as a string.

File I/O

- Files are controlled through System.IO
 - BinaryReader
 - BinaryWriter
 - File
 - Path
 - FileInfo
 - FileNotFoundException
 - FileStream
 - StreamReader // implements TextReader for streams
 - StreamWriter // implements TextWriter for streams
 - TextReader // abstract base class
 - TextWriter // abstract base class
 - StringReader // istringstream equivalent
 - StringWriter // ostream equivalent

Exceptions

- Just like C++
 - try/catch/throw
- Try/Catch
 - <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/exception-handling>
- You can create your own exceptions:
 - <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/creating-and-throwing-exceptions>

Exception Handling

- `try { }`
 - Execute statements inside this block that might throw an exception
- `catch { }`
 - Execute statements inside this block when an exception was thrown
 - use: `catch (ExceptionName e1) { }` to catch specific exceptions
- `finally { }`
 - Execute a “final” set of statements regardless of whether or not an exception was thrown
- `throw` – Throws an exception to a `try{ } catch{ }` block

Loops

- for
- foreach
- while
- do-while

For Loop

- Same as C/C++/Java

```
for (initializer; condition; step) {  
    body  
}
```


Foreach Loop

- Same as Java/C++'s foreach loop
 - Except uses the keyword **foreach**

```
double[] array = new double[] { 1.1, 2.2, 3.3, 4.4, 5.5 };  
foreach (var i in array) {  
    Console.WriteLine($"{i}");  
}
```

- Can be used on primitives or classes that implement `IEnumerable`

While and Do-While Loops

- Exactly like C++

```
while (condition) {  
    // body  
}
```

```
do {  
    // body  
} while (condition);
```

Classes

- Much like C++
- Each element has access protection (not a state like it is in C++)
- Can contain: member functions, properties, variables, and events
- All class objects inherit the top-level class called "object"
 - object is essentially a generic variable

Object Oriented Classes in C#

- Everything is in a class
 - Can have several types of classes:
 - abstract – A class that cannot be instantiated, but can be inherited
 - sealed – A class that cannot be inherited
 - interface – An outline of a class, with no actual code
 - Classes have access protection just like C++
 - internal
 - public
 - protected
 - private
 - Classes' method's definitions are written in the class.
 - virtual – A method, property, or event that can be overridden by a derived class
 - Virtual methods invoke the **derived** class' methods even when passed as the **base** class.

Writing a Class

```
class MyClass
{
    public MyClass() {
        // Constructor that takes 0 arguments
    }
    public string Name { get; set; } // Property that is R/W
    private int Secret;             // Variable that is private
    public readonly int year;       // Public / readonly integer
}
```

Class Access Protection

`class MyClass { }` (is the same as `internal class MyClass { }`)

This class can only be seen by your "assembly"

Assemblies are what C# call libraries

`public class MyClass { }`

This class can be seen by any library or any other programmer's code that imports your library.

Class vs Struct

- Classes are reference based.
 - When a class is created – it is considered a reference (heap based).
 - When a class is passed – it is passed as a pointer.
- Structs are value based.
 - When a struct is created – it is considered a value (stack based).
 - When a struct is passed – it is passed by value.

Access Protection

- The `readonly` keyword allows a variable to be written once in either the declaration or constructor.
 - `public readonly int variable = 22;`
 - OR
 - `public readonly int variable;`
 - `public MyClass() { variable = 22; }`
- The `static` keyword allows a method to be available without an instance of the class. This means it cannot access member variables not specifically passed to it:
 - `public static void MyMethod() { }`

Default Access Protection / Restrictions

Members of	Default member accessibility	Allowed declared accessibility of the member
enum	public	None
class	private	public
		protected
		internal
		private
		protected internal
		private protected
interface	public	None
struct	private	public
		internal
		private

Indexer

- An indexer is what C# calls the [] operator.
- The indexer has a special syntax using "this"

```
class MyClass {  
    private int[] myArray = {1, 2, 3, 4, 5, 6};  
    public int this[int index] {  
        get { return myArray[index]; }  
        set { myArray[index] = value; }  
    }  
}  
  
MyClass mc = new MyClass;  
Console.WriteLine(mc[0]); // Prints 1  
Console.WriteLine(mc[4]); // Prints 5
```

Collections Generics

- [System.Collections.Generic](#)
 - `List<type>`
 - `LinkedList<type>`
 - `HashSet<type>`
 - `SortedSet<type>`
 - `Queue<type>`
 - `Stack<type>`
 - `Dictionary<keytype, valuetype>`

Example List

```
using System.Collections.Generic;
public static void MyFunction() {
    List<double> list_of_doubles = new List<double>();
    for (int i = 0; i < 100000; i++) {
        list_of_doubles.Add(i)
    }
    Console.WriteLine(list_of_doubles.Count); // # of elements
    Console.WriteLine(list_of_doubles[0]);    // First double
    list_of_double.Sort();
    foreach (double d in list_of_doubles)
        Console.WriteLine(d); // Print out every double in the list
}
```

Inheritance

- Much like C++, classes, abstract classes, and interfaces can be derived or “inherited”. The syntax is simply a colon after the name of a class or more separated by a comma.
- Multiple inheritance is not allowed in C#. However, you can implement multiple interfaces.

```
class Derived : SomeBaseClass {  
    // This class now has all of the members of SomeBaseClass  
}
```

Interface

- Interfaces contain the “signature” of a method, property, or event.
 - C# programmers typically put a capital I in front of the name to denote that it is an interface: `IMyInterface`

```
interface IMyInterface {  
    void SomeMethod(); // Notice no code goes here  
}
```

```
class MyClassThatImplementsIMyInterface : IMyInterface {  
    // This class is telling C# that it will implement all methods in IMyInterface  
    void SomeMethod() { // This implements IMyInterface.SomeMethod }  
}
```

All interface members are public and cannot be changed. Notice there is no "public void SomeMethod()" in the class. Doing so would cause a compiler error.

Override

- The override keyword is required to modify an abstract or virtual implementation of a method, property, event, or indexer.
 - You cannot override a non-virtual or non-abstract method!

```
abstract class SomeClass {  
    abstract public int SomeMethod();  
}
```

```
class MyClass : SomeClass {  
    public override int SomeMethod() {  
        // override is required here to tell C# that  
        // you know you're overriding the  
        // abstract class' method  
    }  
}
```

New

- What if I don't want to override a base class' method, and instead, I want to create a brand new one?
 - Use the new keyword

```
public class Base {  
    public virtual void Func() {}  
}  
public class Derived : Base {  
    public new void Func() { }  
}
```

- Derived.Func() is now a totally new function, which hides Base.Func()
 - You can still call Base.Func() from Derived, you just have to specify it as "Base.Func()"
- DO NOT use override and new together. It makes no sense and will cause a compiler error

Accessing Base Classes

- “as” keyword
- `var i = someobject as String;`
 - the “as” attempts to typecast someobject into a String using dynamic casting
- IF the “as” keyword cannot dynamically cast, then var i will be `null`.
 - Use conventional typecasting to have C# throw an exception
 - `var i = (string)someobject;`

Events

- Events allow a delegate (function) to be "registered". When the event is signaled, each function registered gets called.
- Events must have a delegate to denote the type of arguments the event handlers receive
- Events must have an "event" for observers to subscribe to

Event Example

```
class MyClass {  
    // This delegate shows that any event handler we subscribe  
    // will take a generic sender and EventArgs arguments  
    public delegate EventHandler(object sender, EventArgs e);  
    // The event keyword creates the event. We put EventHandler  
    // so that when we subscribe to SomeEvent, we know which  
    // arguments we'll get (sender and e).  
    public event EventHandler SomeEvent;  
}
```

Calling Event Handlers

- You can call all of the event handlers subscribed to an event by calling the event as a function:

```
public event EventHandler SomeEvent;  
public void SendEvent() {  
    // This means we have subscribers  
    if (SomeEvent != null)  
        // This calls all event handlers  
        SomeEvent(this, new EventArgs());  
}
```

Lambda Functions

- Functions in C# can be **anonymous** functions, called lambda functions.
 - <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions>
- Syntax: (parameter list) => { return value; }
- Useful with delegates to pass as a first-class parameter.

Lambda Example

```
var lst = new List<int>();  
lst.Add(10);  
lst.Add(20);  
lst.Add(30);  
lst.Add(40);  
lst.ForEach((i) => { Console.WriteLine(i); });
```

Parallel Programming

- System.Threading
- System.Threading.Tasks
 - Contains the Task<> objects
 - Contains the Parallel Class
 - Parallel.For
 - Parallel.ForEach
 - Parallel.Invoke
- <https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel>

Easiest Invocation

- Use the `Parallel.*` suite of functions
 - C# will automatically manage the threads for you
 - Must have: `using System.Threading.Tasks;`
- `Parallel.For(start, finish, function for each iterator)`
 - start is inclusive, and is usually 0
 - finish is exclusive, and is usually N
 - The function is usually a lambda function, which is given the iterator i
 - `Parallel.For(0, 10, i => { });`
 - Executes the lambda function for each iteration in the for loop in parallel.

Parallel For Example

```
using System.Threading.Tasks;
class Px {
    public static void Main(string[] args) {
        long[] z = new long[20];
        // The following executes a for loop where i = [0..20].
        // This means that 21 threads are spawned and ran concurrently
        // We use i => which is the lambda where i is which i the For loop is running
        // Be careful because i will NOT be 0, 1, 2, 3, .., 20 in that order.
        Parallel.For(0, 20, i => {
            z[i] = DateTime.Now.Ticks;
        });
        // z[0] will store when the parallel for executed this thread with i = 0
        // z[1] will store when the parallel for executed this thread with i = 1
        // ...etc...
    }
}
```

Task Async Pattern (TAP)

- C# offers two keywords for asynchronous function calls:
 - `async` and `await`
 - These require using the `Task` and `Task<>` objects from `System.Threading.Tasks`

Other Patterns

- C# also offers two other patterns (DO NOT USE THESE IN COSC365!)
 - Event Async Pattern (EAP)
 - Asynchronous Programming Model (APM)
- These models generate odd code for the newer CLR virtual machines, so they should be avoided, but they still may be used!

Task.Run

- In System.Threading.Tasks
 - Task.Run(() => { //Do something here });
 - This will run a new task in its separate thread. Task.Run returns a Task structure or Task<T> structure if you need to return a value from Task.Run

```
public async Task<T> MyFunc() {  
    return await Task.Run(() => {  
        int ret = 0;  
        for (int i = 0; i < 100000000; i++)  
            ret += i;  
        return ret;  
    });  
}
```

Task and Task<T> structure

- Task<T> structure returns a result in the property .Result
 - T is a generic type, so I can return a result as any type
- Task and Task<T> have two useful members
 - The property IsCompleted. Returns true if the task is done, false otherwise
 - The function .Wait(). This function will pause your current thread until the task is done.

Asynchronous File Operations

- StreamReader contains ReadToEndAsync() using the TAP pattern
- Your code can call ReadToEndAsync() and move on to other things until reading the file is completed.
- ReadToEndAsync() returns a reference to a Task that you can poll to see when it completes.

Synchronization

- C# has a “lock” statement that will lock a critical section of code

```
private Object i = new Object();  
public void SomeFunction() {  
    lock (i) {  
        // Execute code in here  
        // If another thread tries to use i, the lock statement will  
        // hang until the lock statement is released.  
    }  
}
```

Synchronization Types

- [System.Threading](#)
- Semaphore
- Mutex – Mutual exclusion
- Monitor – Locks a single instance of an object
- Barrier – a synchronization point that waits until all threads reach the barrier. Threads can be asynchronous before and after the barrier, but all threads will wait until ALL threads have reached the barrier.

Files

- Files use a file stream object.
 - StreamReader/StreamWriter for text files.
 - BinaryReader/BinaryWriter for binary files.
 - There are others, but these are the easiest to use for most cases.
- using System.IO;
 - The System.IO library contains all the file streaming classes.
- <https://learn.microsoft.com/en-us/dotnet/api/system.io?view=net-9.0>

StreamReader Example

```
using (StreamReader sr = new StreamReader("TestFile.txt")) {  
    string line;  
    // Read and display lines from the file until the end of  
    // the file is reached.  
    while ((line = sr.ReadLine()) != null) {  
        Console.WriteLine(line);  
    }  
}  
  
// The sr reference goes out of scope after the using statement  
// and it is "finalized" (i.e., closed).
```

Splitting Lines

- ReadLine() gets you an entire line, but what if you need individual data?
 - Split the line and convert the data.

```
StreamReader sr = new StreamReader("TestFile.txt");
string line = sr.ReadLine();
string[] data = line.Split(' ', StringSplitOptions.RemoveEmptyEntries);
if (data.Length >= 2) {
    int left = int.Parse(data[0]);
    int right = int.Parse(data[1])
}
sr.Close();
```

IDisposable

- The using () {} statement automatically runs an IDisposable interface.
 - <https://learn.microsoft.com/en-us/dotnet/api/system.idisposable?view=net-9.0>
- This allows a "Dispose" method to be called after the using statement closes.

Binary Files

- Binary files use the BinaryReader/BinaryWriter classes.
- These classes have a Write that is overloaded (like crazy).
 - <https://learn.microsoft.com/en-us/dotnet/api/system.io.binarywriter.write?view=net-9.0>
- The overloads allow any basic data types to be written.
- The Read methods have ReadXXX(), where XXX is the data type to return.
 - ReadByte()
 - ReadInt32()
 - ReadDouble()

Opening Binary Files

- BinaryReader/BinaryWriter require a stream object to wrap.
 - File.Open(filename, FileMode.Create) -- for writing
 - File.Open(filename, FileMode.Open) -- for reading
- Hand the FileStream object to the BinaryReader/BinaryWriter.

```
using (var stream = File.Open(fileName, FileMode.Create)) {  
    using (var writer = new BinaryWriter(stream)) {  
        writer.Write(1.250F);  
        writer.Write("String");  
        writer.Write(10);  
        writer.Write(true);  
    }  
}
```