



# Structs, Enums, Exceptions & Files

## Structs

A **struct** type is a value type that is typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle or the characteristics of an item in an inventory. The following example shows a simple **struct** declaration:

```
struct Book {  
    public string title;  
    public double price;  
    public string author;  
}
```

Structs share most of the same syntax as classes, but are more limited than classes. Unlike classes, structs can be instantiated without using a new operator.

```
static void Main(string[] args) {  
    Book b;  
    b.title = "Test";  
    b.price = 5.99;  
    b.author = "David";  
  
    Console.WriteLine(b.title);  
    //Outputs "Test"  
}
```

Try It Yourself

Structs do **not** support inheritance and cannot contain virtual methods.

## Structs

Structs can contain methods, properties, indexers, and so on. Structs cannot contain default constructors (a **constructor** without parameters), but they can have constructors that take parameters. In that case the **new** keyword is used to instantiate a **struct** object, similar to class objects.

For example:

```
struct Point {  
    public int x;  
    public int y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}  
  
static void Main(string[] args) {  
    Point p = new Point(10, 15);  
    Console.WriteLine(p.x);  
    // Outputs 10  
}
```

Try It Yourself

## Structs vs Classes

In general, classes are used to model more complex behavior, or data, that is intended to be modified after a class object is created. Structs are best suited for small data structures that contain primarily data that is not intended to be modified after the `struct` is created. Consider defining a `struct` instead of a class if you are trying to represent a simple set of data.

All standard C# types (`int`, `double`, `bool`, `char`, etc.) are actually structs.

---

## Enums

The `enum` keyword is used to declare an enumeration: a type that consists of a set of named constants called the enumerator list.

By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1.

For example, in the following enumeration, Sun is 0, Mon is 1, Tue is 2, and so on:

```
enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

You can also assign your own enumerator values:

```
enum Days {Sun, Mon, Tue=4, Wed, Thu, Fri, Sat};
```

In the example above, the enumeration will start from 0, then Mon is 1, Tue is 4, Wed is 5, and so on. The value of the next item in an Enum is one increment of the previous value. Note that the values are comma separated.

You can refer to the values in the Enum with the dot syntax.

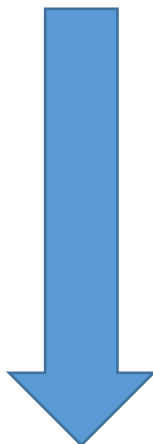
In order to assign Enum values to `int` variables, you have to specify the type in parentheses:

```
enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

static void Main(string[] args) {
    int x = (int)Days.Tue;
    Console.WriteLine(x);
    //Outputs 2
}
```

Try It Yourself

Basically, Enums define variables that represent members of a fixed set. Some sample Enum uses include month names, days of the week, cards in a deck, etc.



---

## Enums

Enums are often used with **switch** statements.  
For example:

```
enum TrafficLights { Green, Red, Yellow };

static void Main(string[] args) {
    TrafficLights x = TrafficLights.Red;
    switch (x) {
        case TrafficLights.Green:
            Console.WriteLine("Go!");
            break;
        case TrafficLights.Red:
            Console.WriteLine("Stop!");
            break;
        case TrafficLights.Yellow:
            Console.WriteLine("Caution!");
            break;
    }
    //Outputs "Stop!"
}
```

Try It Yourself

Tap Try It Yourself to play around with the code!

---

## Exceptions

An **exception** is a problem that occurs during program execution. Exceptions cause abnormal termination of the program.

An **exception** can occur for many different reasons. Some examples:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications.
- Insufficient memory and other issues related to physical resources.

For example, the following code will produce an **exception** when run because we request an index which does not exist:

```
int[] arr = new int[] { 4, 5, 8 };
Console.Write(arr[8]);
```

Try It Yourself

As you can see, exceptions are caused by user error, programmer error, or physical resource issues. However, a well-written program should handle all possible exceptions.



## Handling Exceptions

C# provides a flexible mechanism called the **try-catch** statement to handle exceptions so that a program won't crash when an error occurs. The try and catch blocks are used similar to:

```
try {  
    int[] arr = new int[] { 4, 5, 8 };  
    Console.Write(arr[8]);  
}  
catch(Exception e) {  
    Console.WriteLine("An error occurred");  
}  
//Outputs "An error occurred"
```

Try It Yourself

The code that might generate an **exception** is placed in the **try** block. If an **exception** occurs, the catch blocks is executed without stopping the program. The type of **exception** you want to catch appears in parentheses following the keyword **catch**. We use the general **Exception** type to handle all kinds of exceptions. We can also use the **exception** object **e** to access the **exception** details, such as the original error message (**e.Message**):

```
try {  
    int[] arr = new int[] { 4, 5, 8 };  
    Console.Write(arr[8]);  
}  
catch(Exception e) {  
    Console.WriteLine(e.Message);  
}  
// Index was outside the bounds of the array.
```

Try It Yourself

You can also catch and handle different exceptions separately. Tap next to learn more!

## Handling Multiple Exceptions

A single **try** block can contain multiple **catch** blocks that handle different exceptions separately. Exception handling is particularly useful when dealing with user input. For example, for a program that requests user input of two numbers and then outputs their quotient, be sure that you handle division by zero, in case your user enters 0 as the second number.

```
int x, y;  
try {  
    x = Convert.ToInt32(Console.Read());  
    y = Convert.ToInt32(Console.Read());  
    Console.WriteLine(x / y);  
}  
catch (DivideByZeroException e) {  
    Console.WriteLine("Cannot divide by 0");  
}  
catch (Exception e) {  
    Console.WriteLine("An error occurred");  
}
```

The above code handles the **DivideByZeroException** separately. The last **catch** handles all the other exceptions that might occur. If multiple exceptions are handled, the **Exception** type must be defined last.

Now, if the user enters 0 for the second number, "Cannot divide by 0" will be displayed. If, for example, the user enters non-integer values, "An error occurred" will be displayed.

The following exception types are some of the most commonly used: `FileNotFoundException`, `FormatException`, `IndexOutOfRangeException`, `InvalidOperationException`, `OutOfMemoryException`.

---

## finally

An optional **finally** block can be used after the **catch** blocks. The **finally** block is used to execute a given set of statements, whether an exception is thrown or not.

For example:

```
int result=0;
int num1 = 8;
int num2 = 4;
try {
    result = num1 / num2;
}
catch (DivideByZeroException e) {
    Console.WriteLine("Error");
}
finally {
    Console.WriteLine(result);
}
```

Try It Yourself

The **finally** block can be used, for example, when you work with files or other resources. These should be closed or released in the **finally** block, whether an exception is raised or not.

---

## Writing to Files

The **System.IO namespace** has various classes that are used for performing numerous operations with files, such as creating and deleting files, reading from or writing to a file, closing a file, and more.

The **File** class is one of them. For example:

```
string str = "Some text";
File.WriteAllText("test.txt", str);
```

The **WriteAllText()** method creates a file with the specified path and writes the content to it. If the file already exists, it is overwritten.

The code above creates a file **test.txt** and writes the contents of the **str** string into it.

To use the **File** class you need to use the **System.IO namespace**: **using System.IO;**

---

---

## Reading from Files

You can read the content of a file using the `ReadAllText` method of the `File` class:

```
string txt = File.ReadAllText("test.txt");  
Console.WriteLine(txt);
```

Try It Yourself

This will output the content of the `test.txt` file.

The following methods are available in the `File` class:

**AppendAllText()** - appends text to the end of the file.

**Create()** - creates a file in the specified location.

**Delete()** - deletes the specified file.

**Exists()** - determines whether the specified file exists.

**Copy()** - copies a file to a new location.

**Move()** - moves a specified file to a new location

All methods automatically close the file after performing the operation.

---

# End.