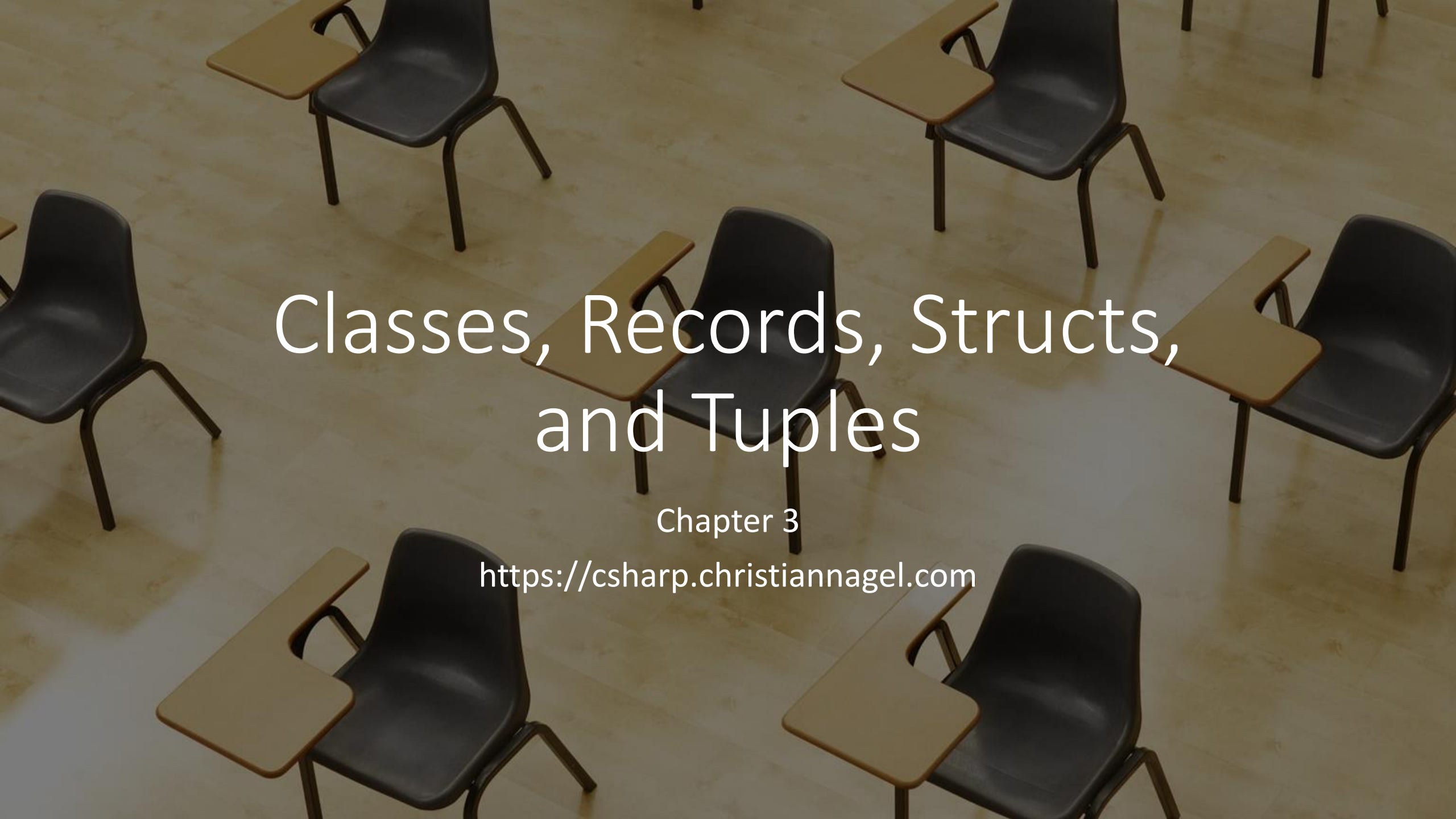

Classes, Records, Structs, and Tuples

Chapter 3

https://csharp.christiannagel.com

Topics

- Class, records, structs
- Methods, Properties, Constructors
- Local Functions
- Extension Methods
- Anonymous Types
- Tuples
- Pattern Matching

Types

- class
  - reference type
- struct
  - value type
- ref struct
  - value type
- record
  - reference type
  - class with built-in features

Pass by Value or by Reference

- struct – pass by value
- class – pass by reference

```
AStruct x1 = new() { A = 1 };
AStruct x2 = x1;
x2.A = 2;
Console.WriteLine($"original didn’t change with a struct: {x1.A}");

//...

public struct AStruct
{
  public int A;
}
```

Type Members

Fields

- Good practice: *private* access modifier

```
public class Person
{
  //...
  private string _firstName;
  private string _lastName;
  //...
}
```

static modifier

- Share field with all instances of the class

```
public class PeopleFactory
{
  //...
  private static int s_peopleCount;
  //...
}
```

Readonly Field

- Can’t change it after instantiation

```
public class Person
{
  //...
  public Person(string firstName, string lastName)
  {
    _firstName = firstName;
    _lastName = lastName;
  }

  private readonly string _firstName;
  private readonly string _lastName;
  //...
}
```

Properties

- get and set accessors

```
public class Person
{
  //...

  private int _age;
  public int Age
  {
    get => _age;
    set => _age = value;
  }
}
```

Auto Implemented Properties

- Field, Implementation of get and set accessor created by compiler

```
public class Person
{
  public int Age { get; set; }

  //...
}
```

Properties – Special Features

- Change access modifier

```
public int Age { get; private set; }
```

- Readonly Properties

```
private readonly string _firstName;
public string FirstName
{
  get => _firstName;
}
```

- Expression-Bodied Properties

```
public string FullName =>
  $“{FirstName} {LastName}“;
```

Init-Only Set Accessor

- init accessor

```
public class Book
{
  public Book(string title)
  {
    Title = title;
  }

  public string Title { get; init; }
  public string? Publisher { get; init; }
}
```

Methods

- Declare name, parameter types, and return type

```
public bool IsSquare(Rectangle rect)
{
  return (rect.Height == rect.Width);
}
```

Expression-Bodied Methods

- Declare name, parameter types, and return type

```
public class Math
{
  public int Value { get; set; }
  public int GetSquare() => Value * Value;
  public static int GetSquareOf(int x) => x * x;
}
```

```
// Call static members
int x = Math.GetSquareOf(5);
Console.WriteLine($"Square of 5 is {x}");

// Instantiate a Math object
Math math = new();

// Call instance members
math.Value = 30;
Console.WriteLine($"Value field of math variable contains {math.Value}");
Console.WriteLine($"Square of 30 is {math.GetSquare()}");
```

Method Overloading

- Use different parameter types and/or different parameter numbers

```
class ResultDisplayer
{
  public void DisplayResult(string result)
  {
    // implementation
  }

  public void DisplayResult(int result)
  {
    // implementation
  }
}
```

Named Arguments

- Use parameter names invoking the method

```
public void MoveAndResize(int x, int y, int width, int height) { }
```

```
r.MoveAndResize(x: 30, y: 40, width: 20, height: 40);
```

Optional Arguments

- Parameters can be optional – use a default
- Compiler changes the invocation to use the default value (don’t change the default value with a new version)

```
public void TestMethod(int notOptionalNumber, int optionalNumber = 42)
{
  Console.WriteLine(optionalNumber + notOptionalNumber);
}
```

```
TestMethod(11);
TestMethod(11, 12);
```

Variable Number of Arguments

- params keyword with array type

```
public void AnyNumberOfArguments(params object[] data)
{
```

```
AnyNumberOfArguments(1);
AnyNumberOfArguments(1, 3, 5, 7, 11, 13);
```

Constructors

- Name of the class
- No return type

```
public class MyNumber
{
  private int _number;
  private MyNumber(int number) => _number = number;
  //...
}
```

```
MyNumber n = new(42);
```

Calling Constructors from other Constructors

- constructor initializer with this keyword

```
class Car
{
  private string _description;
  private uint _nWheels;
  public Car(string description, uint nWheels)
  {
    _description = description;
    _nWheels = nWheels;
  }
  public Car(string description): this(description, 4)
  {
  }
}
```

Static Constructors

- Initialize Static members
- Invoked before any instance member

```
class MyClass
{
  static MyClass()
  {
    // initialization code
  }
  //...
}
```

Local Functions

- Can only be invoked within methods
- Without static modifier can access variables within the method (closure)

```
public static void IntroLocalFunctions()
{
  static int Add(int x, int y) => x + y;

  int result = Add(3, 7);
  Console.WriteLine("called the local function with this result: {result}");
}
```

Generic Methods

- Define the parameter type on use

```
class GenericMethods
{
  public static void Swap<T>(ref T x, ref T y)
  {
    T temp;
    temp = x;
    x = y;
    y = temp;
  }
}
```

```
int x = 1;
int y = 2;
GenericMethods.Swap(ref x, ref y);
```

```
string s1 = "a";
string s2 = "b";
GenericMethods.Swap(ref s1, ref s2);
```

Extension Methods

- Create methods that extend other types
- Can’t access private members of the type
- Convenience methods – compiler converts to invocation of static method

```
public static class StringExtensions
{
  public static int GetWordCount(this string s) => s.Split().Length;
}
```

Anonymous Types

- Compiler creates a class with read-only properties

```
var captain = new
{
  FirstName = "James",
  MiddleName = "Tiberius",
  LastName = "Kirk"
};
```

Nominal and Positional Records

- record instead of class keyword
- Designed for immutability

```
public record Book1
{
    public string Title { get; init; } = string.Empty;
    public string Publisher { get; init; } = string.Empty;
}
```

```
public record Book2(string Title, string Publisher)
{
  // add your members, overloads
}
```

Record Features

- Equality comparison
- With expressions

```
var aNewBook = book1a with { Title = "Professional C# and .NET – 2024" };
```

- Deconstruction (positional records only)

Structs

- Value Type

```
public readonly struct Dimensions
{
  public Dimensions(double length, double width)
  {
    Length = length;
    Width = width;
  }

  public double Length { get; }
  public double Width { get; }
  //...
}
```

Enum Types

- Named values

```
public enum Color
{
  Red,
  Green,
  Blue
}
```

```
[Flags]
public enum DaysOfWeek
{
  Monday = 0x1,
  Tuesday = 0x2,
  Wednesday = 0x4,
  Thursday = 0x8,
  Friday = 0x10,
  Saturday = 0x20,
  Sunday = 0x40
  Weekend = Saturday | Sunday,
  Workday = 0x1f,
  AllWeek = Workday | Weekend
}
```

ref, in, and out

- Pass by reference (ref)
- Readonly with method (in)
- Return value from method (out)

```
Console.Write("Please enter a number: ");
string? input = Console.ReadLine();
if (int.TryParse(input, out int x))
{
  Console.WriteLine();
  Console.WriteLine($"read an int: {x}");
}
```

```
void ChangeAValueType(ref int x)
{
  x = 2;
}
```

```
void PassValueByReferenceReadonly(in SomeValue data)
{
  // data.Value1 = 4; - you cannot change a value, it's a read-only variable!
}
```

Tuples

- Combine different types without creating a struct or class

```
(string AString, int Number, Book Book) tuple1 =
   ("magic", 42, new Book("Professional C#", "Wrox Press"));
```

Tuple Deconstruction

- Deconstruct into variables

```
var tuple1 = (AString: "magic",
   Number: 42, Book: new Book("Professional C#", "Wrox Press"));
 (string aString, int number, Book book) = tuple1;

 Console.WriteLine($"a string: {aString}, number: {number}, book: {book}");

 (_, _, var book1) = tuple1;
 Console.WriteLine(book1.Title);
```

Deconstruction with custom types

- Implement a Deconstruct method

```
public class Person
{
  //...
  public void Deconstruct(out string firstName, out string lastName,
    out int age)
  {
    firstName = FirstName;
    lastName = LastName;
    age = Age;
  }
}
```

Pattern Matching

Pattern Matching with Tuples

- switch expression with tuple pattern

```
(TrafficLight Current, TrafficLight Previous)
  NextLightUsingTuples(TrafficLight current, TrafficLight previous) =>
    (current, previous) switch
    {
      (Red, _) => (Amber, current),
      (Amber, Red) => (Green, current),
      (Green, _) => (Amber, current),
      (Amber, Green) => (Red, current),
      _ => throw new InvalidOperationException()
    };
```

Pattern Matching with Tuples

- switch expression with property pattern

```
TrafficLightState NextLightUsingRecords(TrafficLightState trafficLightState)
  => trafficLightState switch
  {
    { CurrentLight: AmberBlink } =>
      new TrafficLightState(Red, trafficLightState.PreviousLight, 3000),
    { CurrentLight: Red } =>
      new TrafficLightState(Amber, trafficLightState.CurrentLight, 200),
    { CurrentLight: Amber, PreviousLight: Red} =>
      new TrafficLightState(Green, trafficLightState.CurrentLight, 2000),
    //...
};
```

Summary

- Classes
- Records
- Structs
- Tuples
- Type Members