.NET Basics

Session #02

dotnet training

Agenda
.NET Basics

Namespaces

Classes and Interfaces

Collections and Generics

Exception Handling

Manage app settings

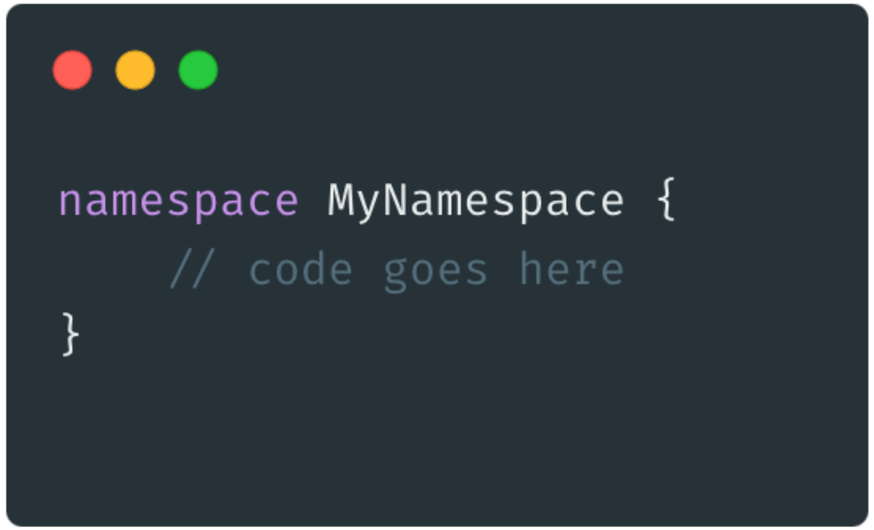
Create your libraries

Namespaces

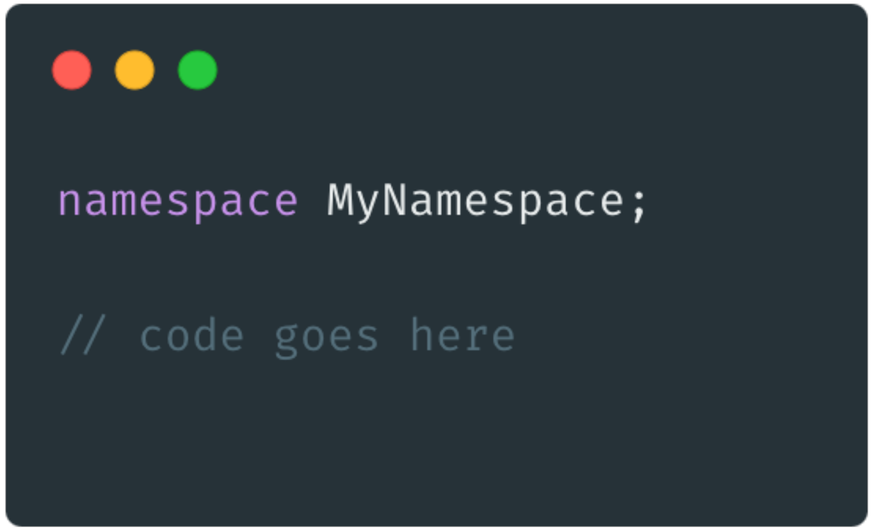
Namespaces

- Namespace is a way to organize code into logical groups, avoiding naming conflicts between different components of a software system
- Provide a way to group related classes, interfaces, enums, and other types, and they help to make code more organized, modular, and easier to understand
- Java developers may be familiar with a similar concept called **packages**, which are used in Java to organize related classes and avoid naming conflicts

Namespaces: How to define?



Before C#10

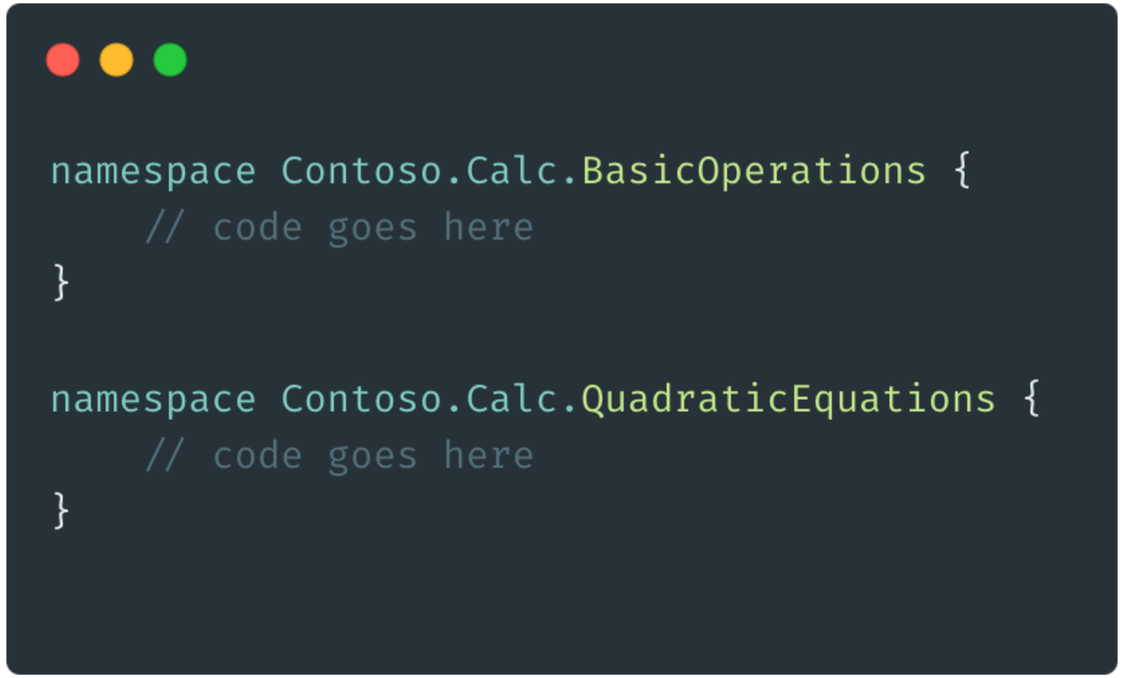


After C#10

Namespaces vs. Packages

- Namespace hierarchy is separate from the file system hierarchy
- Naming best practices
  - Use PascalCase for each namespace hierarchy part
  - <Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]

Namespaces: How to use?



```
namespace Contoso.Calc.BasicOperations {
    // code goes here
}

namespace Contoso.Calc.QuadraticEquations {
    // code goes here
}
```

Namespaces on .NET 6+

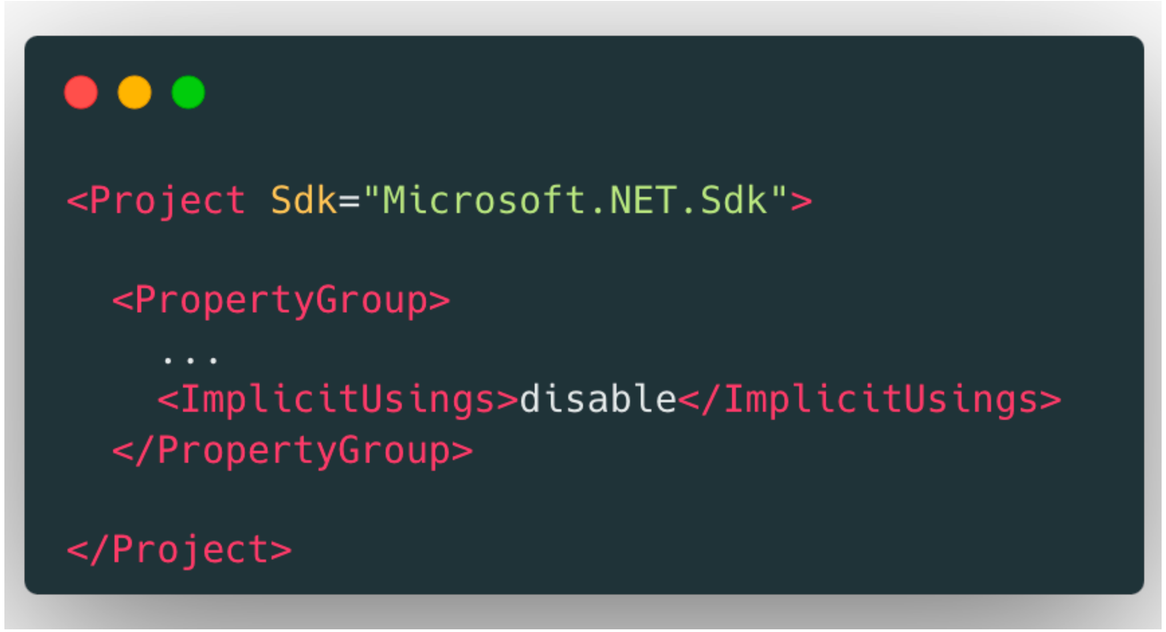
```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Implicit Usings

- Compiler automatically adds a set of using directives based on the project type.
- For console applications, the following directives are implicitly included in the application:
  - `using System;`
  - `using System.IO;`
  - `using System.Collections.Generic;`
  - `using System.Linq;`
  - `using System.Net.Http;`
  - `using System.Threading;`
  - `using System.Threading.Tasks;`
- For other project types, other `using` statements can be included

Implicit Usings

- If you want to remove this behavior and manually control all namespaces in your project, add `<ImplicitUsings>disable</ImplicitUsings>` to your project file in the `<PropertyGroup>` element, as shown in the following example:



```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    ...
    <ImplicitUsings>disable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

Global Usings

- A global using directive imports a namespace for your whole application instead of a single file. These global directives can be added either by adding a <Using> item to the project file, or by adding the global using directive to a code file.

- On .NET6+ is common to create a `GlobalUsings.cs` file on project root folder with `global using` statements to import namespace on all application files

Classes

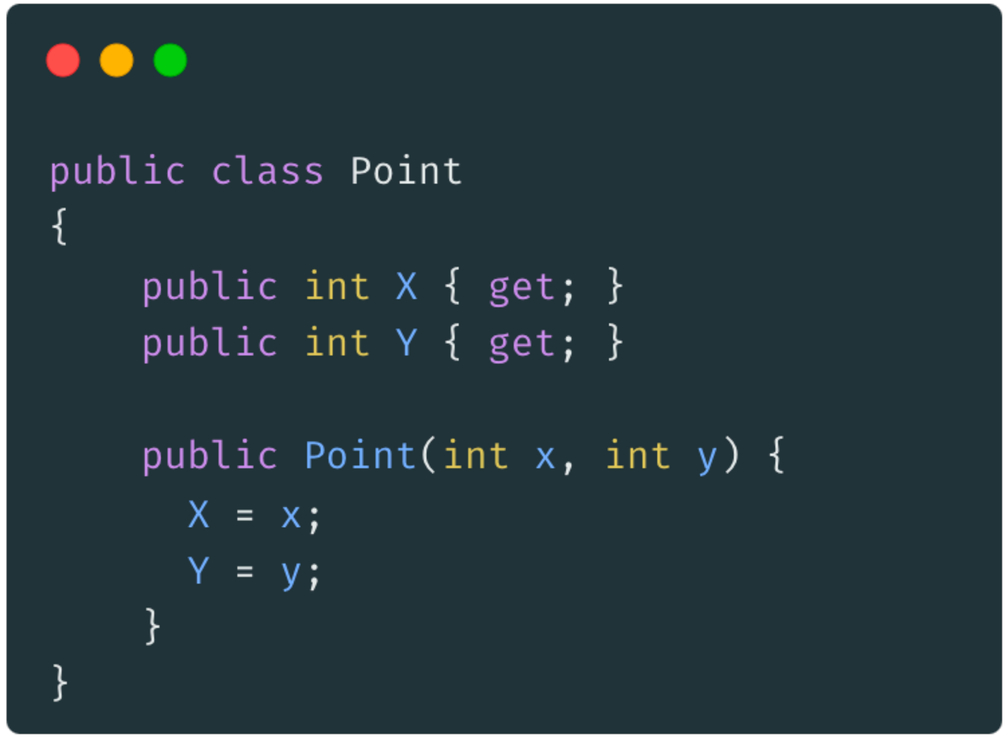
Classes

- Same conceptual meaning as in Java
- Define object state through Properties
- Property is implemented in Java with Getters and Setters
- An empty constructor is mandatory but can be defined implicitly
- Allow constructor overload
- All classes are inherited from namespace System.Object

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class
(System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
```

Constructor and instantiation

C# Classes



```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) {
      X = x;
      Y = y;
    }
}
```

Modifiers

C# Classes

- `public: Accessed by any other code in the same assembly or another assembly that references it`
- `private`: Accessed only by code in the same class
- `protected`: Accessed only by code in the same class, or in a class that is derived from that class
- `internal`: Accessed by any code in the same assembly, but not from another assembly. Is the ***default modifier*** when none is provided
- `protected internal`: Accessed by any code in the assembly in which it's declared, or from within a derived class in another assembly.
- `private protected`: Accessed by types derived from the class that are declared within its containing assembly.

Modifiers

C# Classes

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

Classes: Properties

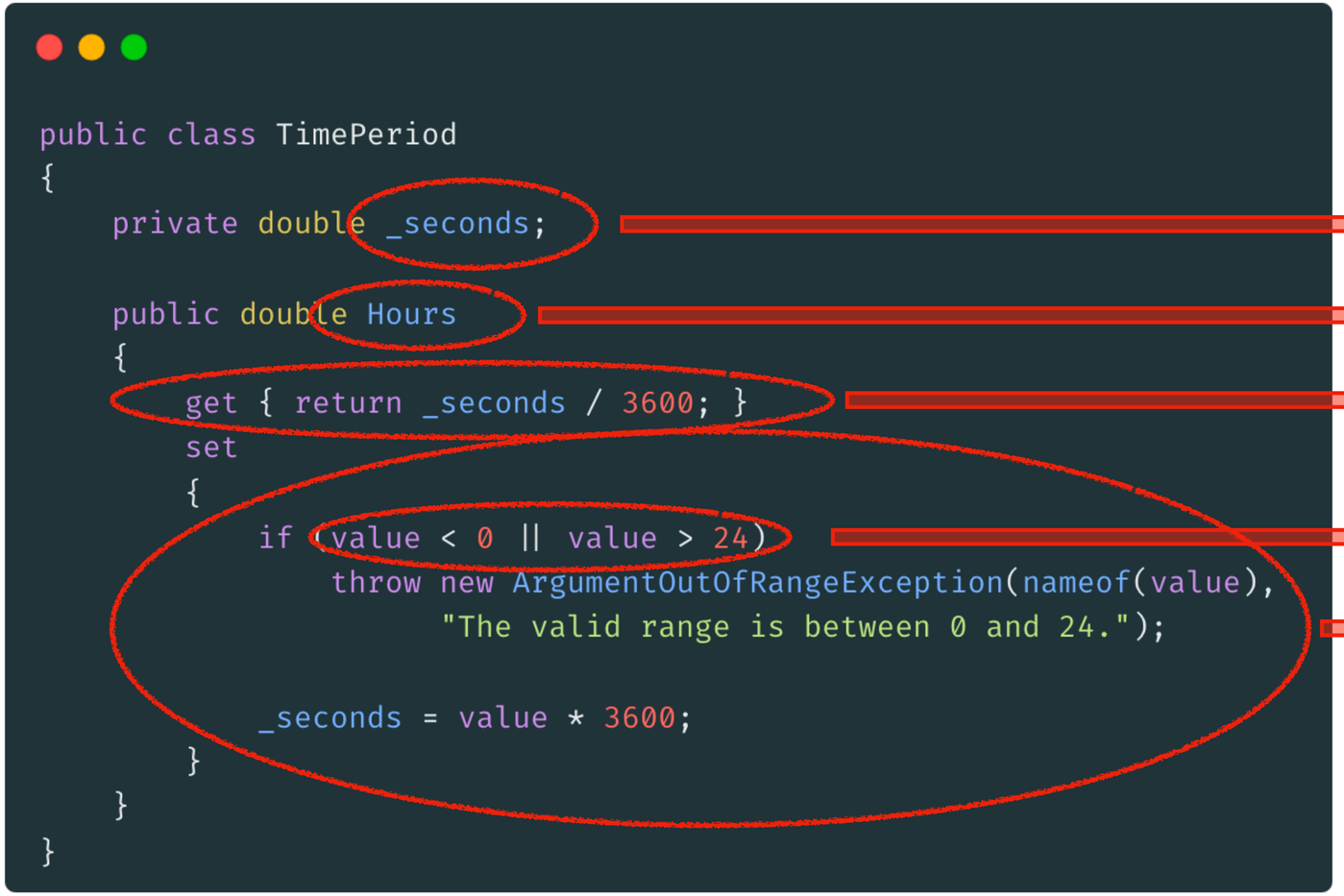
Properties

C# Classes

- Provides a flexible mechanism to read, write, or compute the value of a private ***field***
- Properties can be used as if they're public data members, but they're special methods called ***accessors***.
- This feature enables data to be accessed easily and still helps promote the safety and flexibility of methods
- Property have a ***get*** accessor to get the field value and a ***set*** accessor to update ***field*** value
- On set accessor you use ***value*** keyword to access provided value for property

Properties: Backing Fields

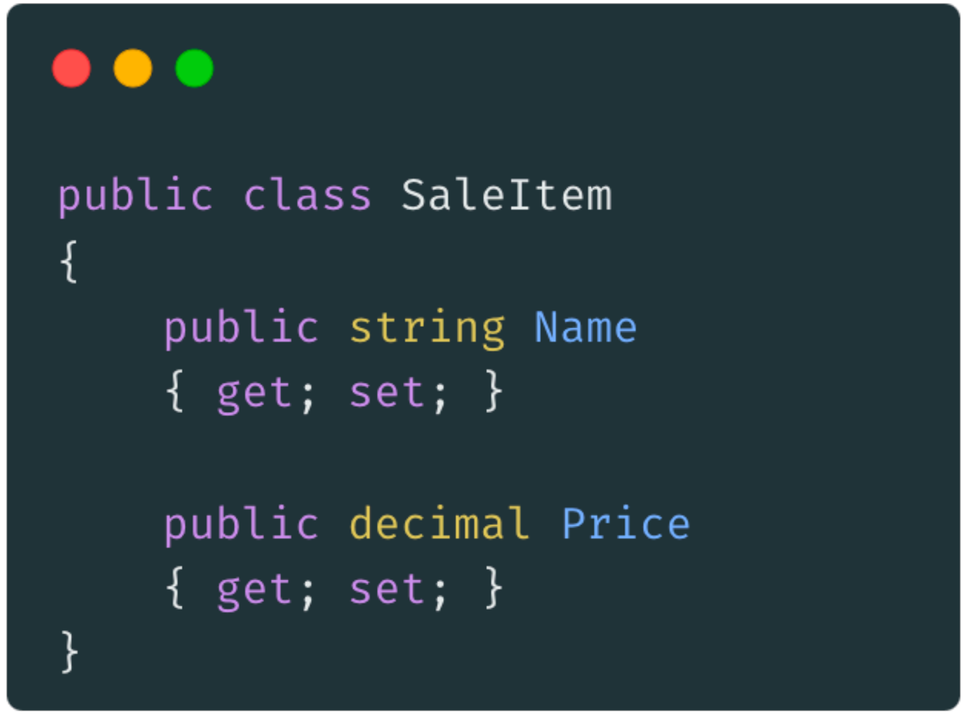
C# Classes



Properties: Auto-implemented

C# Classes

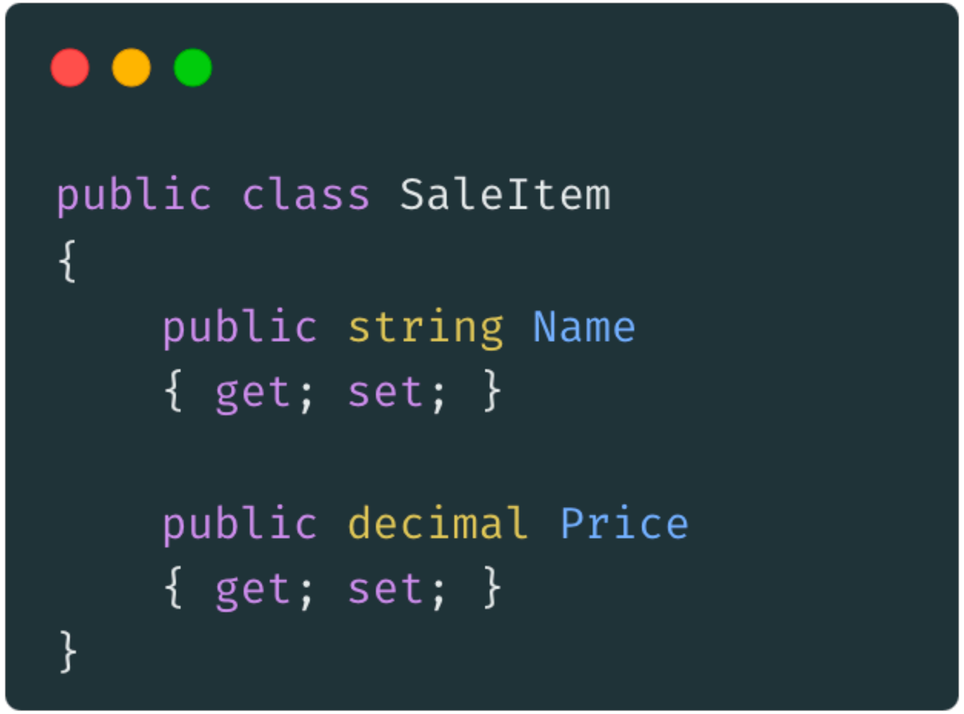
- Many times get and set accessors just assign or retrieve a value without including any extra logic
- By using auto-implemented properties, you can simplify your code while having the C# compiler providing the backing field for you
- When using this approach, both accessors must be auto-implemented



```
public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}
```

Properties: How to use?

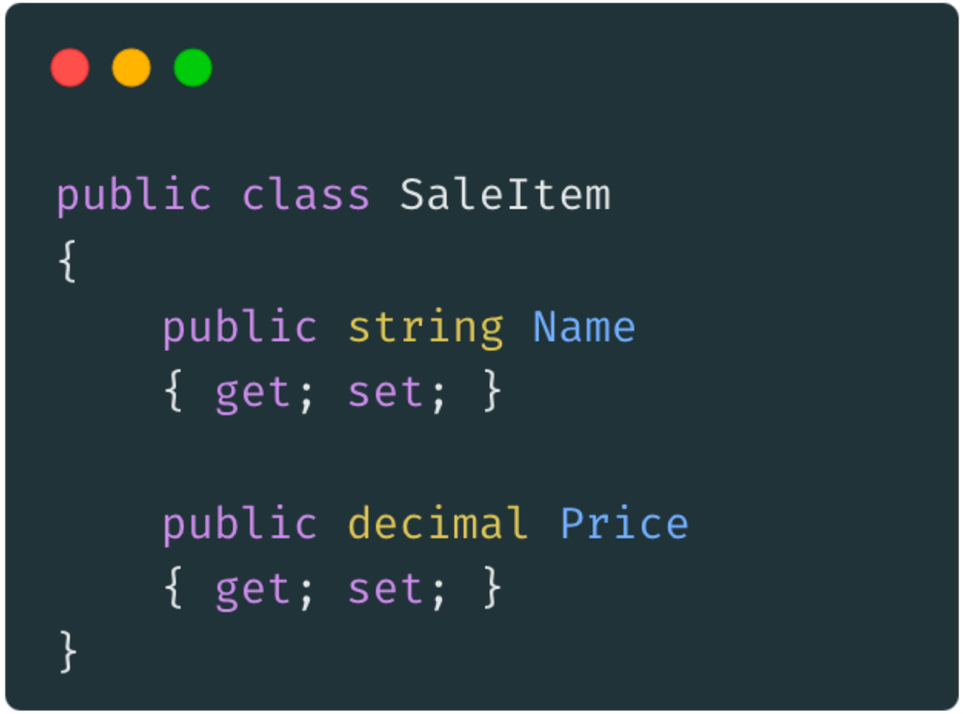
C# Classes



```
public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}
```

Properties: How to use?

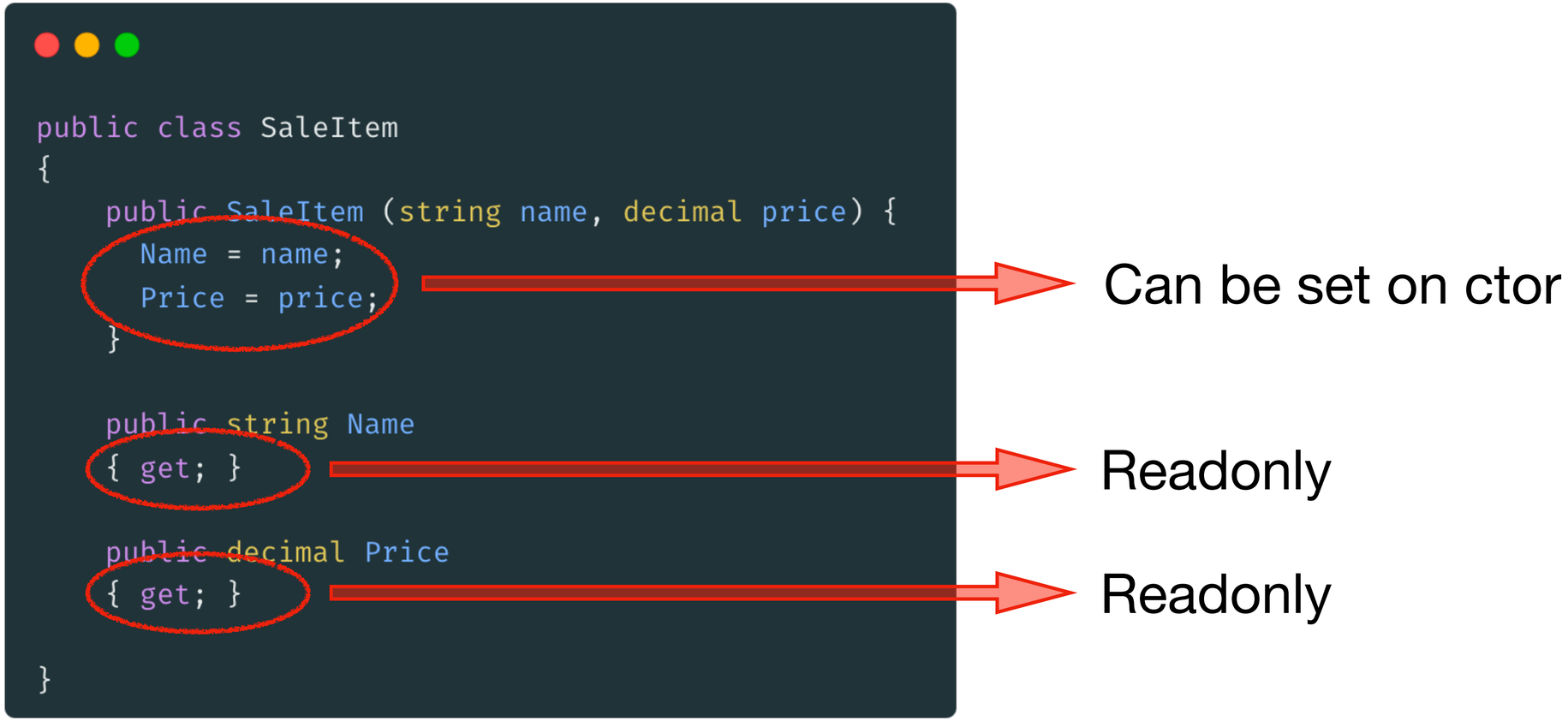
C# Classes



```
public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}
```

Properties: Readonly auto-implemented

C# Classes



Properties: Modifiers on properties

C# Classes

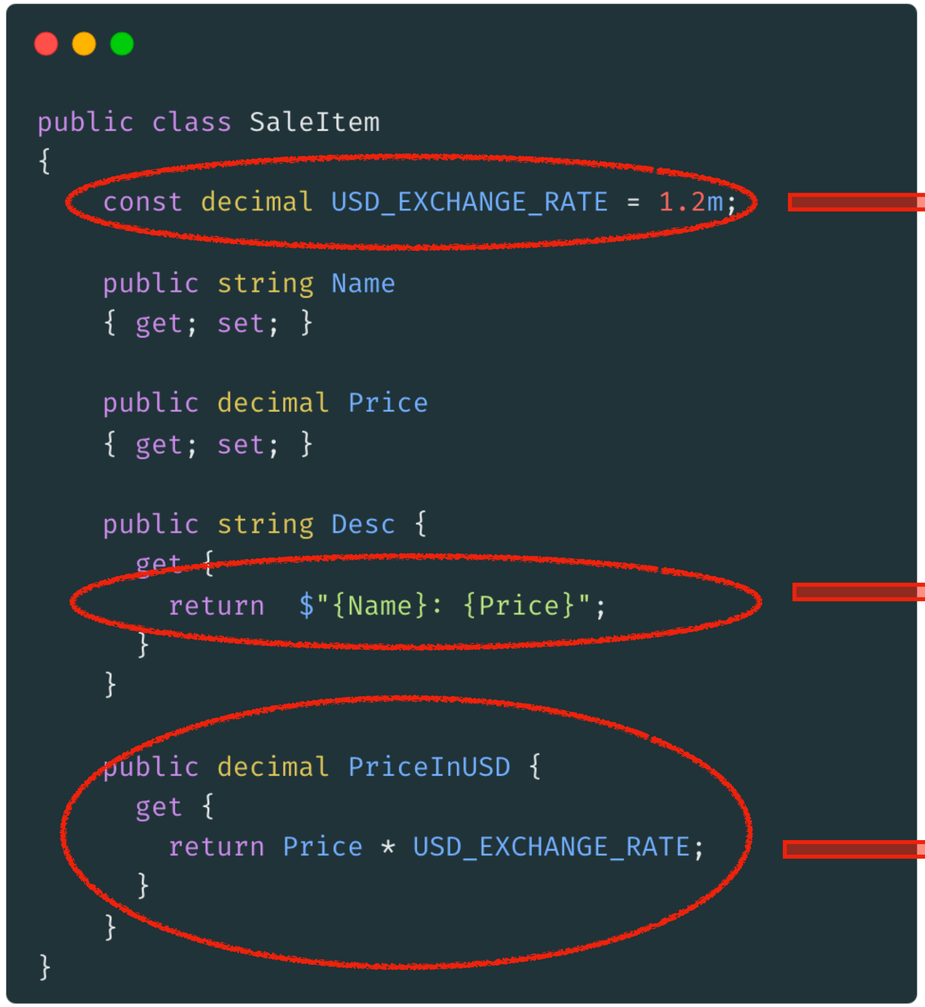
Properties with no direct Field

C# Classes

- Properties can be defined to return values that are not directly stored on one field
- Most of the time is used to get a return based on any fields/properties that exists
- Used to apply some simply logic on the returned value
- These properties are readonly

Properties with no direct Field

C# Classes



```
public class SaleItem
{
    const decimal USD_EXCHANGE_RATE = 1.2m;

    public string Name
    { get; set; }

    public decimal Price
    { get; set; }

    public string Desc {
      get {
        return  $"{Name}: {Price}";
      }
    }

    public decimal PriceInUSD {
      get {
        return Price * USD_EXCHANGE_RATE;
      }
    }
}
```

How to declare constants

String interpolation (C#11)

Using another property

Properties: Init-only and required

C# Classes

- ***init*** accessor allows to set value only on object initialiser making the property as readonly after that
- ***required*** keyword makes a property mandatory that needs to be set when the object is created
- This can be achieved using the object constructor or initialiser

Properties: Init-only and required

C# Classes

```
public class Person {
  public string FirstName { get; init; } = "John";
  public string LastName { get; init; } = "Doe";
  public required int Age { get; set; }

  public override string ToString() => $"{FirstName} {LastName} ({Age})";
}
```

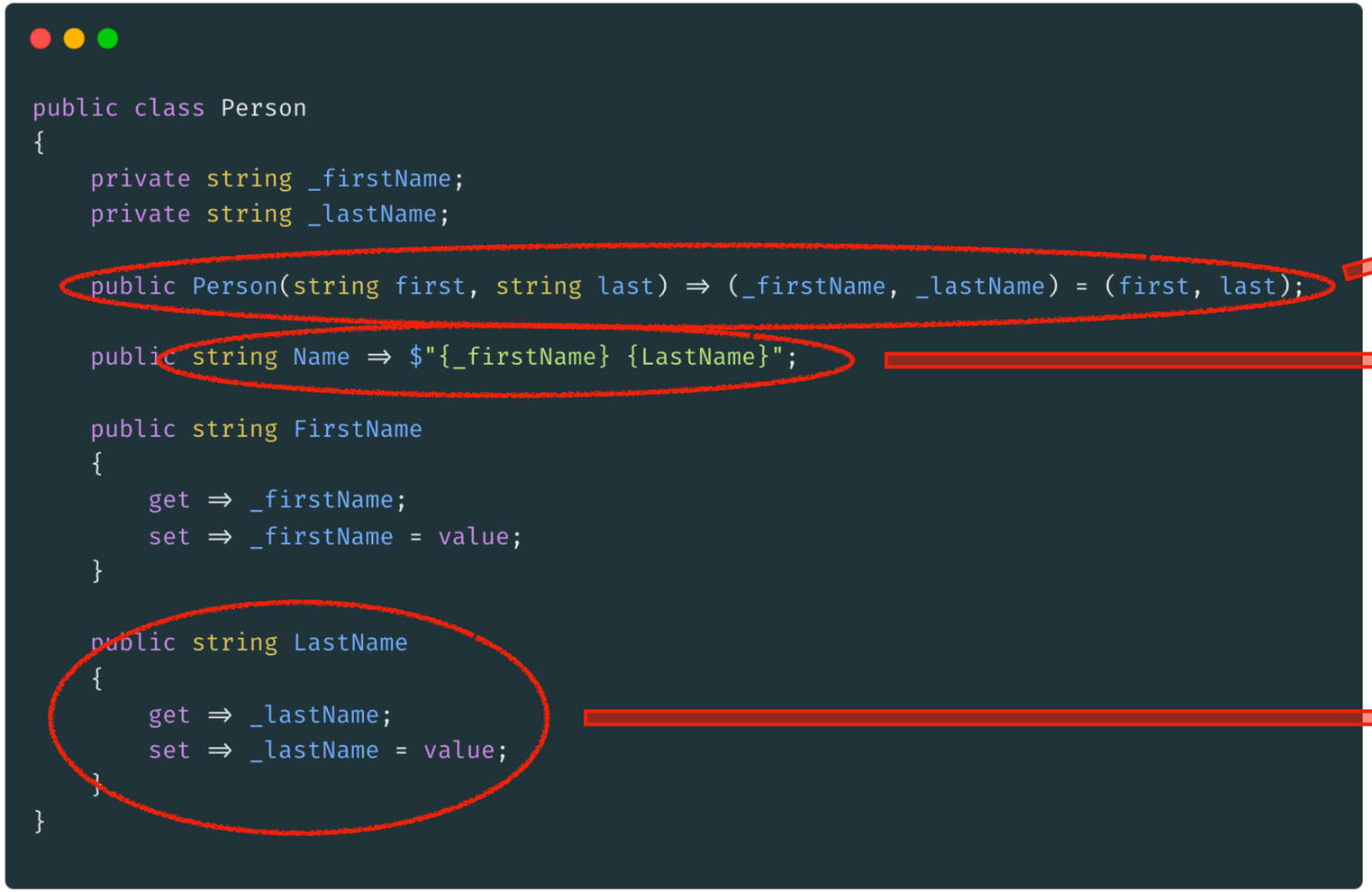
Properties: Expression body definitions

C# Classes

- Property accessors often consist of single-line statements that just assign or return the result of an expression
- You can implement these properties as expression-bodied members
- Expression body definitions consist of the ‘=>’ symbol followed by the expression to assign to or retrieve from the property

Properties: Expression body definitions

C# Classes



```
public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last) => (_firstName, _lastName) = (first, last);

    public string Name => $"{_firstName} {LastName}";

    public string FirstName
    {
        get => _firstName;
        set => _firstName = value;
    }

    public string LastName
    {
        get => _lastName;
        set => _lastName = value;
    }
}
```

Can be used to set values by order

Simplify readonly properties

Used on accessors

Classes: Methods

Methods

C# Classes

- A method is a code block that contains a series of statements
- A program causes the statements to be executed by calling the method and specifying any required method arguments
- In C#, every executed instruction is performed in the context of a method
- You define methods to declare behaviour on your class
- Like classes. You may have partial methods, where method signature is defined on one file and implementation on another file (mostly used with code generators)

Methods: Abstract and Virtual

C# Classes

- Beside modifiers, your methods can be abstract or virtual
- Abstract methods needs to be declared on an abstract class
- Abstract methods are mandatory to be implemented on derived class, like in Java
- Virtual methods needs to be explicitly set to allow them to be override on derived class
- In Java, all methods are virtual by default
- System.Object have 3 virtual methods:
  - ToString()
  - GetHashCode()
  - Equals(Object obj1, Object obj2)

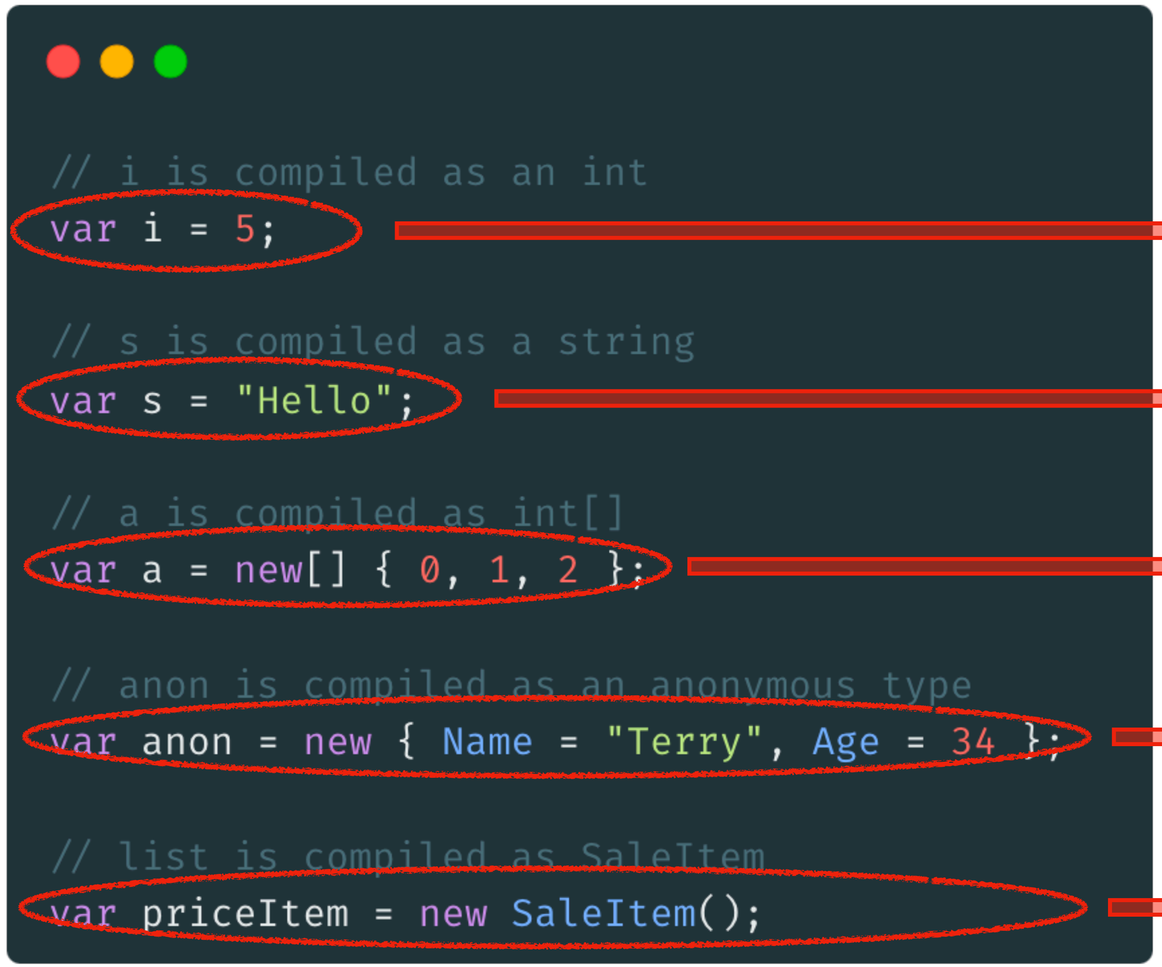
Methods: Implicit typed vars

C# Classes

- Local variables can be declared without giving an explicit type
- The `var` keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement
- The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET class library
- This vars will have always the same type during code execution

Methods: Implicit typed vars

C# Classes



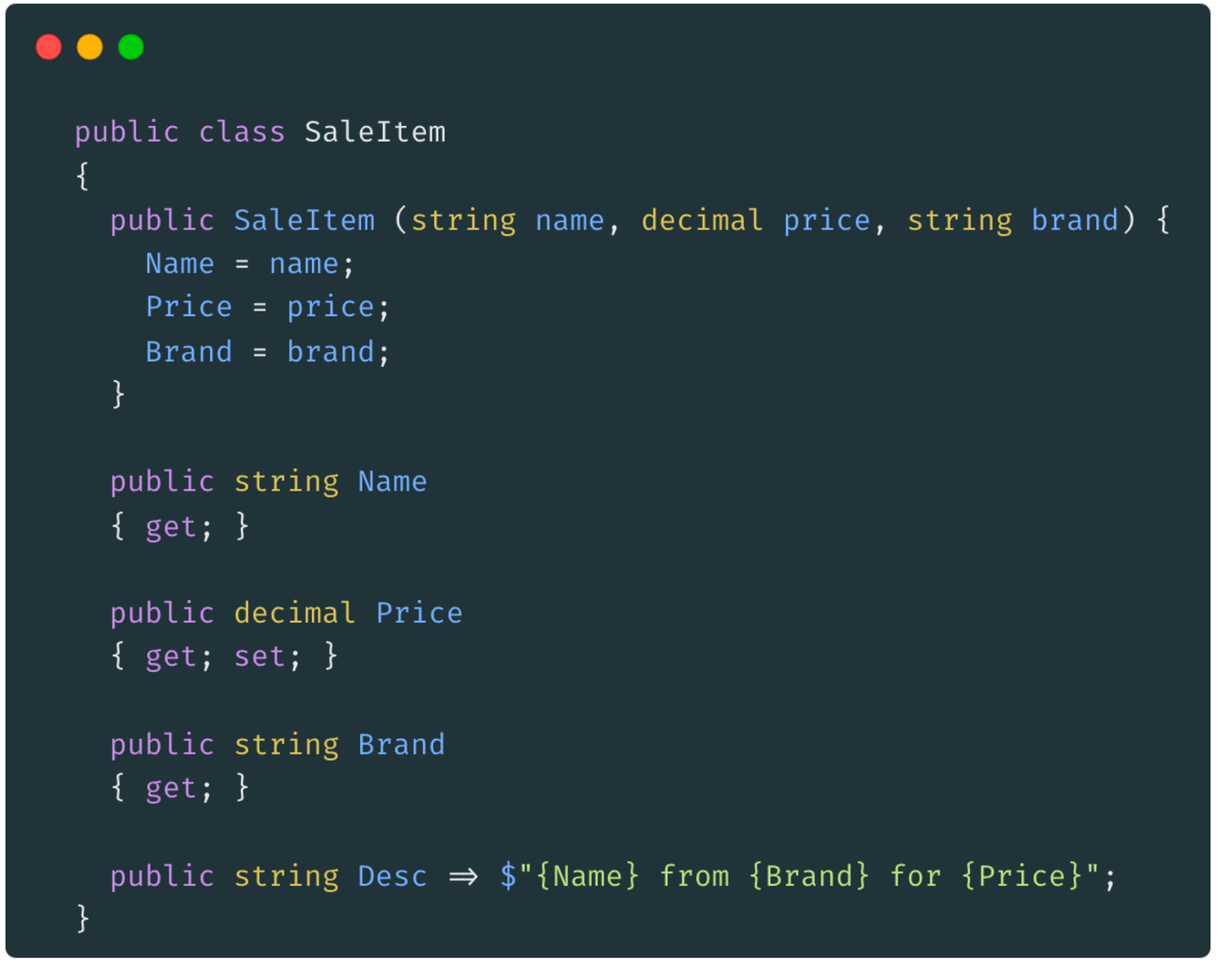
Methods: Named and Optional Parameters

C# Classes

- ***Named arguments*** enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list
- ***Optional arguments*** enable you to omit arguments for some parameters, providing default values
- Optional arguments can be used in methods and constructors

Methods: Named Parameters

C# Classes



```
public class SaleItem
{
  public SaleItem (string name, decimal price, string brand) {
    Name = name;
    Price = price;
    Brand = brand;
  }

  public string Name
  { get; }

  public decimal Price
  { get; set; }

  public string Brand
  { get; }

  public string Desc ⇒ $"{Name} from {Brand} for {Price}";
}
```

Methods: Optional Parameters

C# Classes

```
public class SaleItem
{
  const decimal USD_EXCHANGE_RATE = 1.1m;

  // class code

  public decimal PriceInUsd(decimal priceInEur, decimal exchangeRate = USD_EXCHANGE_RATE) {
    return priceInEur * exchangeRate;
  }
}
```

Methods: ref, in and out Parameters

C# Classes

- ***ref*** parameter makes argument to be sent to the method by reference
- ***in*** parameter causes arguments to be passed by reference but ensures the argument is not modified
- ***out*** parameter causes arguments to be passed by reference and makes mandatory for the argument to be set inside the method code

Methods: ref, in and out Parameters

C# Classes

```
public static int DoSameCalcs(ref int num1, in int num2, out int num3) {
  // num3 needs to be set before used
  num3 = 30;
  var result = num1 + num2 + num3;
  num1 = num2 + num3;

  // next line will produce and error
  // num2 = 5;

  return result;
}
```

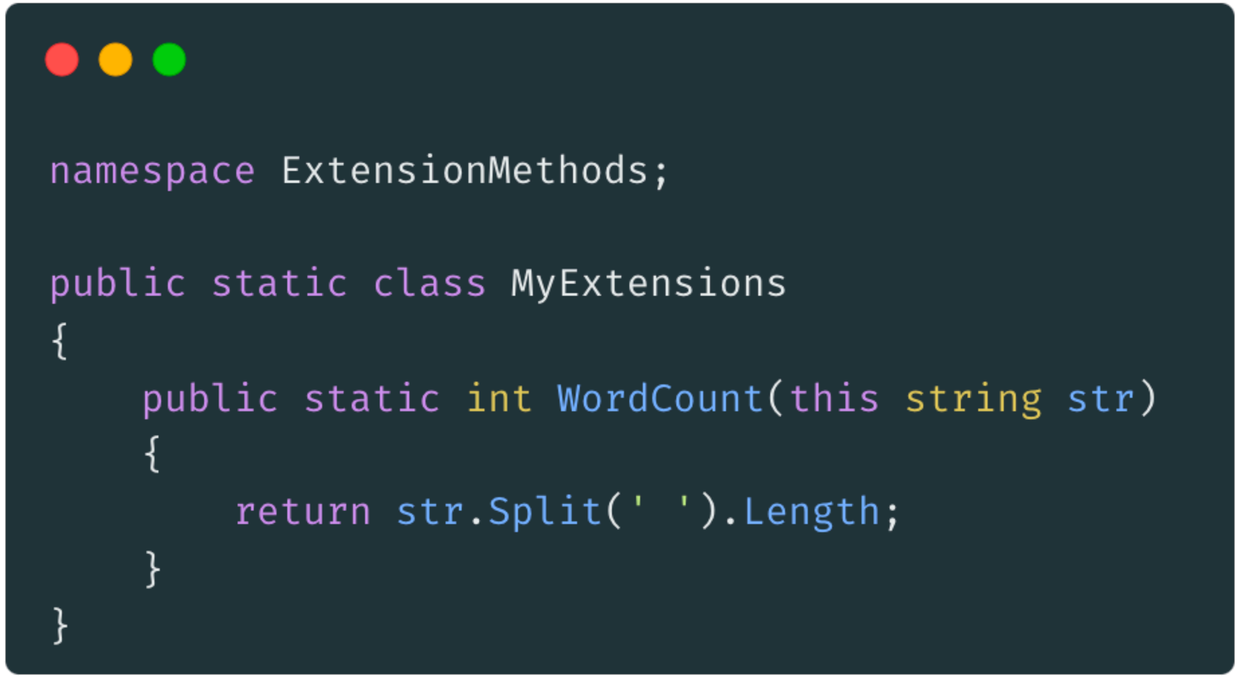
Extension methods

C# Classes

- Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type
- Extension methods are static methods, but they're called as if they were instance methods on the extended type
- There's no apparent difference between calling an extension method and the methods defined in a type.

Extension methods

C# Classes



```
namespace ExtensionMethods;

public static class MyExtensions
{
    public static int WordCount(this string str)
    {
        return str.Split(' ').Length;
    }
}
```

Classes: Inheritance

Inheritance

C# Classes

- Follows all common inheritance properties that exists in Java
- C# only allow **single inheritance** were a class only can inherit from another
- Uses ‘`:`’ caracter instead of ‘`extends`’
- User ‘`base`’ keyword instead of ‘`super`’

Inheritance

C# Classes

```
public class Person
{
  public Person (string first, string last) => (FirstName, LastName) = (first, last);
  public string FirstName { get; }
  public string LastName { get; }

  public override string ToString() => $"Name: {FirstName} {LastName}";
}
```

Inheritance

C# Classes

```
public class Employee : Person
{
  public Employee(int empId, string first, string last) : base (first, last) => (EmployeeId) = (empId);
  public int EmployeeId { get; set; }
  public override string ToString() => $"{base.ToString()}, Employee Id: {EmployeeId}";
}
```

Inheritance

C# Classes

```
Person person = new Person("John", "Doe");
Person employee = new Employee(last: "Smith", first: "Jane", empId: 123456);

Console.WriteLine(person);
Console.WriteLine(employee);

if (employee is Employee emp) {
  emp.EmployeeId = 98765;
}

Console.WriteLine(employee);

═══════════════════════════════════════════════════════════════════════

Name: John Doe
Name: Jane Smith, Employee Id: 123456
Name: Jane Smith, Employee Id: 98765
```

Records

Records

C# Basics

- Introduced in C# 9, a **record** is a class that provides special syntax and behaviour for working with data models
- A record can use positional parameters to create and instantiate a type with immutable properties
- A record provides value equality, meaning that two instances of a record type are equal if the types match and all property and field values match.
- A record supports nondestructive mutation, meaning that you can use a `with` expression to create a copy of an immutable object with new values in selected properties.
- A record can inherit from another record, but not from a class, and a class can’t inherit from a record
- Automatic ToString() method on a JSON format

Records

C# Basics

```
public record Person(string FirstName, string LastName);

public record Employee(string FirstName, string LastName) : Person(FirstName, LastName) {
    public int Salary => MonthWorkedHours * 10;
    public int MonthWorkedHours { get; set;} = 0;
}
```

Records: With keyword

C# Basics

```
public record Person(string FirstName, string LastName);

public record Employee(string FirstName, string LastName) : Person(FirstName, LastName) {
    public int Salary => MonthWorkedHours * 10;
    public int MonthWorkedHours { get; set;} = 0;
}
```

Structs

Struct

C# Basics

- A structure type (or struct type) is a **value type** that can encapsulate data and related functionality
- Structure types have **value semantics**. Means when used an instance of the type is copied
- Use structure types to design small data-centric types that provide little or no behaviour.
- .NET uses structure types to represent a number, a Boolean value, a Unicode character, a time instance.
- Because structure types have value semantics, is recommended to define immutable structure types using `readonly` keyword

Struct: Operators overloading

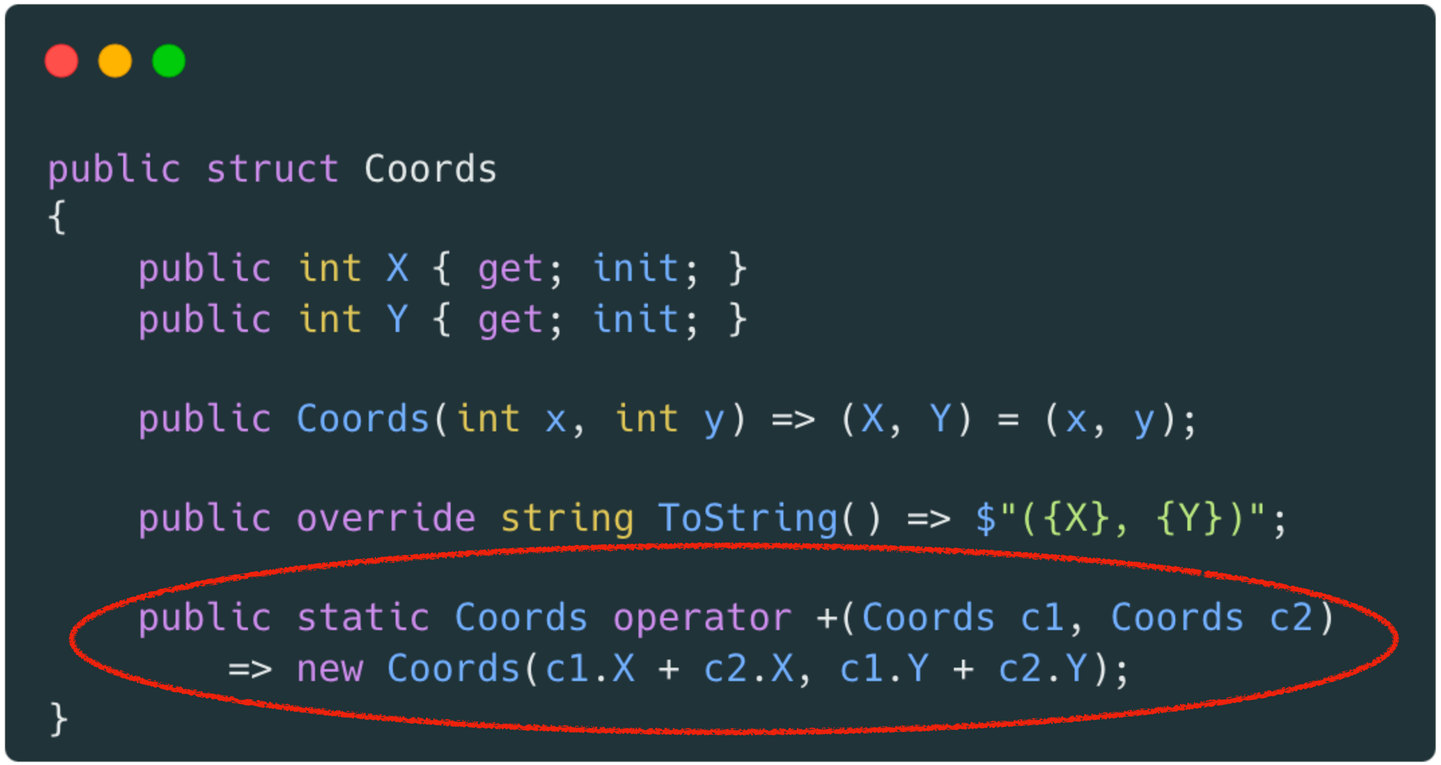
C# Basics

- Since you’re creating a value type, operator overloading can be a good way to use language semantics to perform actions on your struct
- Overloadable operators

Operators	Notes
+x, -x, !x, ~x, ++, --, true, false	The `true` and `false` operators must be overloaded together.
x + y, x - y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x >>> y	
x == y, x != y, x < y, x > y, x <= y, x >= y	Must be overloaded in pairs as follows: `==` and `!=`, `<` and `>`, `<=` and `>=`.

Struct: Operators overloading

C# Basics



```
public struct Coords
{
    public int X { get; init; }
    public int Y { get; init; }

    public Coords(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";

    public static Coords operator +(Coords c1, Coords c2)
        => new Coords(c1.X + c2.X, c1.Y + c2.Y);
}
```

Class, Record or Struct?

When to use?

Class, Record or Struct?

- Can your data type be a value type? **Go with struct**.
- Does your type describe a value-like, preferably immutable state? **Go with record.**
- Otherwise, **go with class**

Interfaces

Interfaces

C# Basics

- Similar concept as in Java
- Allow multiple inheritance by other interfaces
- Allow multiple implementation by a class
- Can define properties and methods

Interfaces

C# Basics

Naming Conventions

Naming Conventions

PascalCase vs camelCase

- PascalCase
  - Classes, records, structs and namespaces
  - Interfaces prefixing it with an ‘I’
  - Public members like fields, properties, methods, …
- camelCase
  - Private or internal fields, prefixing with ‘_’
  - Static private or internal fields, prefixing with ’s_’
  - Method parameters

Collections and Generics

Generics

What is Generics?

- Generics introduces the concept of type parameters to .NET
- Make possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code
- Generic classes and methods combine reusability, type safety, and efficiency in a way that their non-generic counterparts cannot
- Generics are most frequently used with collections and the methods that operate on them
- .NET Generics implementation is similar with JAVA implementation

Collections

.NET Collections

- System.Collections.Generic Classes
  - Collections using generics
  - Generic collection enforces strong typing by allowing only the desired data type to be added
- System.Collections Classes
  - Do not store elements as specifically typed objects, but as objects of type Object
  - Present due to legacy code
- System.Collections.Concurrent Classes
  - These collections should be use when you need both previous collections but with multiple threads are accessing the collection concurrently

System.Collections.Generic

.NET Collections

Class	Description
Dictionary<TKey,TValue>	Represents a collection of key/value pairs that are organized based on the key.
List<T>	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
Queue<T>	Represents a first in, first out (FIFO) collection of objects.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

LINQ: Language-Integrated Query

.NET Collections

- LINQ provides language-level querying capabilities, and a higher-order function API, that enable you to write expressive declarative code
- Allow you to query, select, aggregate and convert collections easily
- Can be used on any type of collections
- When integrated with EntityFramework (later on this training) makes really easy to query databases
- You can right LINQ with API or Query syntax
- Basically are the same, but you should use query syntax if you need to scope variables within your queries because of the complexity

LINQ: Language-Integrated Query

.NET Collections

```
var germanShepherds = dogs.Where(dog => dog.Breed == DogBreed.GermanShepherd);
```

API Syntax

```
var queryGermanShepherds = from dog in dogs
                           where dog.Breed == DogBreed.GermanShepherd
                           select dog;
```

Query Syntax

LINQ: Standard Query Operators

.NET Collections

Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last,
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

Exception Handling

Exception Handling

How To

- Exception handling on .NET works in a similar way as JAVA
- Main difference is in .NET don’t exists checked and unchecked exceptions
- With that declaring ‘`throws`’ statements are not mandatory
- To handle exceptions, you can use ‘`try-catch-finally`’ blocks

Manage app settings

App Setting

Sources

- How to add configuration to your app like JAVA properties
- Differs from how you are running your application, if you're using a GenericHost approach or not.
- .NET provides default configuration for the app in the following order, from highest to lowest priority
  - Command-line arguments
  - Environment variables
  - `appsettings.{Environment}.json` using the JSON configuration provider. For example, `appsettings.Production.json` and `appsettings.Development.json`.
  - `appsettings.json` using the JSON configuration provider.

App Setting

Sources

- To access your configuration values, you can use code like:

```
Configuration["Logging:LogLevel:Default"];
Configuration["Logging__LogLevel__Default"];
Configuration["MyKey"];
```

- To set environment variables, you use the exact same name as you use to get the variable in code

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information"
    }
  },
  "AllowedHosts": "*",
  "MyKey": "This is a key"
}
```

app.settings.json

Environments

App Settings

- Can be set on two variables, depending of the type of project you're running
  - DOTNET_ENVIRONMENT
  - ASPNETCORE_ENVIRONMENT
- Can be set using dotnet cli command parameter:

```
dotnet run --environment Production
```

- You can set any value you want but the platform have the following native values:
  - Development
  - Staging
  - Production: The default value

Create your libraries

Create Nuget Package

Create a library

- A Nuget package is a library with DLLs (compiled code) and metadata to identify your package
- To create a class library use dotnet CLI: `dotnet new classlib`
- Metadata is added to project file (.csproj) inside `<PropertyGroup>` tag

```
<PackageId>Contoso.08.28.22.001.Test</PackageId>
<Version>1.0.0</Version>
<Authors>your_name</Authors>
<Company>your_company</Company>
```

Nuget Package Metadata

Create a library

- Version needs to follow a semantic version and immutability principle
- PackageId needs to be unique on the package sources your are using (nuget.org or others)
- Additional properties can be set on the metadata, describe here: https://learn.microsoft.com/en-us/dotnet/core/project-sdk/msbuild-props#package-properties

Nuget Package Packing

Create a library

- To create the Nuget package you must create a .nupkg file
- .nupkg file is a archive file with all contents (unzip it to check the content)
- A dotnet CLI command help you on that: `dotnet pack`

Nuget Package Push

Create a library

- After creating your package, you can push it to a Nuget package source
- You can use nuget.org to publish public packages or you can use other private options to do it
- If you want to use nuget.org but not making your package immediately available for everyone, you can use Nuget integration environment at https://int.nugettest.org/
- To push packages you need to set some authn/authz dependent from the package source implementation
- On nuget.org the process uses API key

Q&A

Build your first app

Lab #02

dotnet training

Build your first app

Lab #02

- Learning Objectives
  - Create a Nuget library
  - Push your library to a remote repository
  - Create a console app using your class library
  - Use console for input and output
  - Control your app behaviour using app settings
  - Handle filesystem
- MD Link: https://github.com/tasb/dotnet-training/blob/main/labs/lab02.md
- HTML Link: https://tasb.github.io/dotnet-training/labs/lab02.html