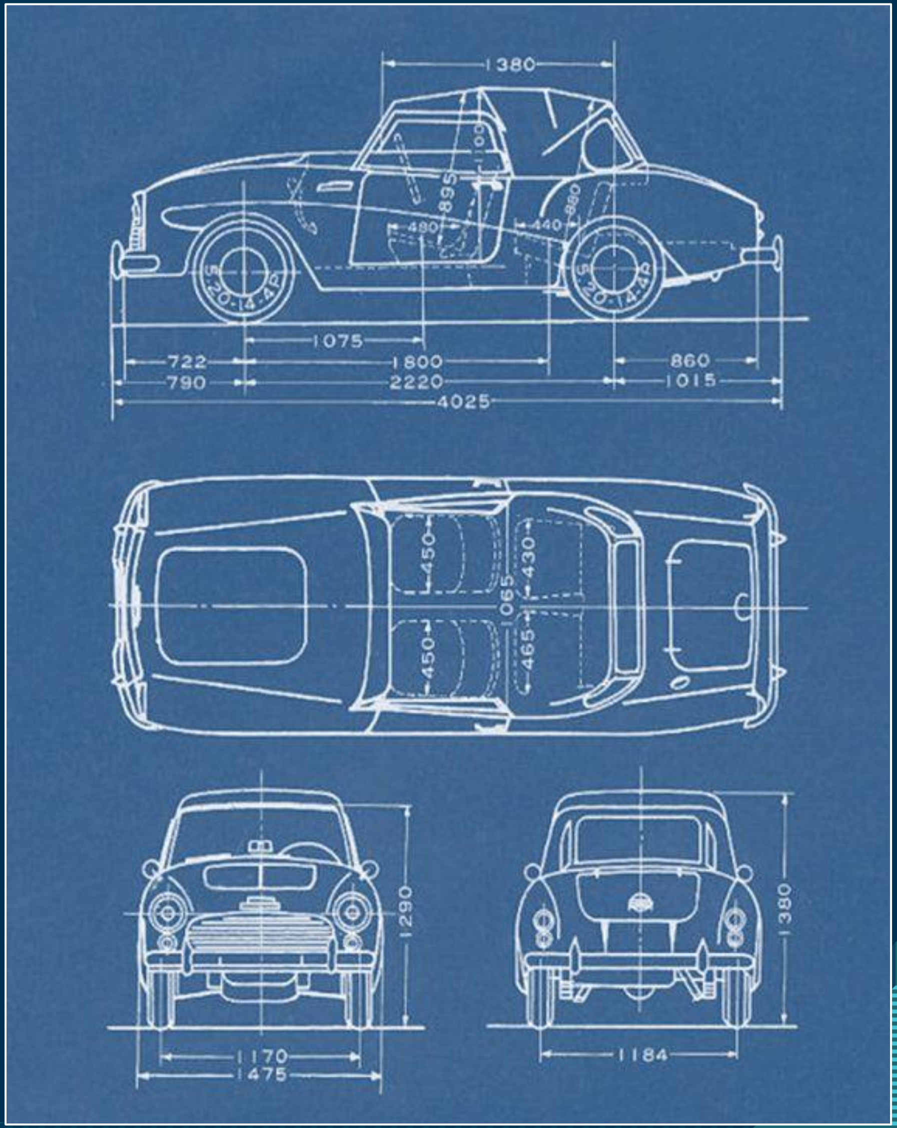
# Classes and Structs

## Introduction

- Let’s begin with a simple analogy to understand classes and their contents.
- Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal.
- What must happen before you can do this? Well, before you can drive a car, someone has to design it.
- A car typically begins as engineering drawings, similar to the blueprints used to design a house.
- These engineering drawings include the design for an accelerator pedal to make the car go faster.
- The pedal “hides” the complex mechanisms that actually make the car go faster, just as the brake pedal “hides” the mechanisms that slow the car and the steering wheel “hides” the mechanisms that turn the car.
- This enables people with little or no knowledge of how engines work to drive a car easily.



## Introduction

- Unfortunately, you can’t drive the engineering drawings of a car.
- Before you can drive a car, it must be built from the engineering drawings that describe it.
- A completed car will have an actual accelerator pedal to make the car go faster.

- A class is a construct that enables you to create your own custom types by grouping together variables of other types, methods and properties.
- When you define a class, you define a blueprint for a data type.
- This doesn't actually define any data, but it does define what an object of the class will consist of and what operations can be performed on such an object.

## Classes and Structs

- Classes and structs are essentially templates from which you can create objects.
- Each object contains data and has methods to manipulate and access that data.
- The class defines what data and behavior each particular object (called an instance) of that class can contain.

# Classes and Structs

- For example, if you have a class that represents a customer, it might define fields such as `customerId`, `firstName`, and `lastName`, which are used to hold information about a particular customer.
- It might also define functionality that acts upon the data stored in these fields.
- You can then instantiate an object of this class to represent one specific customer, set the field values for that instance, and use its functionality.

```
class PhoneCustomer
{
	public const string DayOfSendingBill = "Monday";
	private int _customerId;
	private string _firstName;
	private string _lastName;
}
```

Sheridan Get Creative

## Classes and Structs

- Structs differ from classes because they do not need to be allocated on the heap.
  - Classes are reference types and are always allocated on the heap.
  - Structs are value types and are stored on the stack.
- Also, structs cannot derive from a base struct.
- You typically use structs for smaller data types for performance reasons.
- Storing value types on the stack avoids garbage collection.

## Classes and Structs

- In terms of syntax, structs look very similar to classes.
- The main difference is that you use the keyword `struct` instead of `class` to declare them.
- For example, if you wanted all `PhoneCustomer` instances to be allocated on the stack instead of the heap, you could write the following:

```
struct PhoneCustomerStruct
{
        public const string DayOfSendingBill = "Monday";
        private int _customerId;
        private string _firstName;
        private string _lastName;
}
```

## Classes and Structs

- For both classes and structs, you use the keyword `new` to declare an instance.

```
var myCustomer1 = new PhoneCustomer();          // works for a class
var myCustomer2 = new PhoneCustomerStruct();    // works for a struct
```

- In most cases, you use classes much more often than structs.

- **Note:** An important difference between classes and structs is that objects of type of class are passed by reference, and objects of type of a struct are passed by value.

## Classes

- A class contains members, which can be static or instance members.
- A static member belongs to the class; an instance member belongs to the object.
- With static fields, the value of the field is the same for every object.
- With instance fields, every object can have a different value.
- Static members have the `static` modifier attached.

# Classes

- The kind of members are explained in the following table.

| MEMBER | DESCRIPTION |
|---|---|
| Fields | A field is a data member of a class. |
| Constants | Constants are associated with the class (although they do not have the static modifier). |
| Constructors | Constructors are special functions that are called automatically when an object is instantiated. |
| Methods | Methods are functions associated with a particular class. |
| Properties | Properties are substitutes for getter/setter methods in .NET. |

## Class Example

```
public class Product
{
   private int code;     // Fields
   private string _name;
   private double _price;

   public Product()      // Constructors
   {
      code = 0;
      _name = "";
      _price = 0.0;
   }
```

*(Continued…)*

```
   public Product(int code, string name,
                            double price)
   {
      this.code = code;
      _name = name;
      _price = price;
   }

   public double GetPrice() // Method
   {
      return _price;
   }
}
```

- Recently, the .NET Core team switched the naming convention to prefix field names by an underscore.
- This provides an extremely convenient way to identify fields in contrast to local variables.

## Object Instantiation

```
// creating object using parameterless constructor
Product p1 = new Product();

// creating object using parameterized constructor
Product p2 = new Product(101, "Laptop", 600.00);
```

## Object Instantiation

- When an object is instantiated, a *constructor* is executed to initialize the data that makes up the object.
- If a class doesn’t provide a constructor, a *default constructor* is executed.
  - The default constructor simply initializes all the data to default values.

- An application can create two or more instances of the same class. Each instance is a separate entity.
  - If you make changes to one object, the other objects created from the same class are not affected.

## Fields

- A variable that’s defined at the class level within a class is called a **field**.
- Usually, fields are set to `private`, to prevent other classes from accessing them.

```
public class Product
{
      private int _quantity; // A private field
      public double _price;  // A public field

      ...
      ...
}
```

## Read-only Fields

- The concept of a constant is something that C# shares with most programming languages.
- However, *constants don't necessarily meet all requirements*.
- On occasion, you may have a variable whose value shouldn't be changed but the value is not known until runtime.
- C# provides another type of variable that is useful in this scenario: the *readonly field*.

## Read-only Fields

- The `readonly` keyword provides a bit more flexibility than `const`, allowing for situations in which you want a field to be constant but you also need to carry out some calculations to determine its initial value.
- The rule is that you can assign values to a `readonly` field inside a constructor or while declaring it, but not anywhere else.
- `readonly` field can be an instance as well as static field.
- `const`, on the other hand, is always static, even if not explicitly defined.

# Read-only Fields

- For example, each document you edit has a creation date, which you wouldn’t want to allow the user to change.
- In example below, the field is public – you don’t normally need to make readonly fields private, because by definition they cannot be modified externally.
- The same principle also applies to constants.

```
public class Document
{
        public readonly DateTime CreationDate;

        public Document()
        {
                // Set the creation date of a document
                CreationDate = DateTime.Now;
        }
}
```

## Methods

- To provide other classes with access to a method, declare it using the `public` access modifier.
- The name of a method combined with its parameters form the method's signature.
  - Although, you can use the same name for more than one method, each method must have a unique signature (method overloading).

```
public class Product
{
        private int _quantity;
        private double _price;

        public double GetPrice()     // A public method
        {
                return _price;
        }

        public string GetData()      // Another public method
        {
                return $"Quantity = {_quantity}, Price = {_price};
        }
}
```

Get Creative

## Constructors

- The name of a constructor must be the same as the name of the class.
- A constructor is usually declared with the `public` access modifier, and it doesn't have a *return type*.

```
public class Product
{
        private int _quantity;
        private double _price;

        // A constructor
        public Product()
        {
                _quantity = 0;
                _price = 0.0;
        }
```

```
(Continued…)
        // another constructor
        public Product(int quantity, double price)
        {
                _quantity = quantity;
                _price = price;
        }
}
```

## Constructors

- It’s not necessary to provide a constructor for your class.
- If you don’t supply any constructor, the compiler will generate a default one behind the scenes.
- It will be a very basic constructor that just initializes all the member fields by zeroing them out:
  - Null for reference types, zero for numeric data types, and false for bools.

- If a default constructor is not adequate, you need to write your own constructors.

# Constructors

- If you supply any constructors that take parameters, the compiler ***will not*** automatically supply a default one.
- In the following example, because a one-parameter constructor is defined, the compiler assumes that this is the only constructor you want to be available, so it will not implicitly supply any others.

```
public class MyNumber
{
        private int _number;

        public MyNumber(int number)
        {
                _number = number;
        }
}
```

- Now, if you try instantiating a `MyNumber` object using a no-parameter constructor, you will get a compilation error.

```
// causes compilation error
MyNumber num = new MyNumber();
```

# Exercise

- Create an app that simulates a banking account.
- Create an `Account` class, that has:
  - Private field `_balance`
  - Public getter and setter methods for `_balance`
  - Public methods `Deposit()` and `Withdraw()` that receive the amount to deposit or withdraw
- The class `Account` should have two constructors, one that initializes the `_balance` to zero, and the other that initializes it to some specified amount.
- When the application runs, create an account either with no initial deposit, or with some initial deposit. Call the appropriate constructor for this.
- Handle the following issues:
  - Amount cannot be negative.
  - Withdraw amount cannot be more than `_balance`.

## Properties

- The idea of a property is that it is a redesigned getter and setter function.
- The purpose of a property is to get and/or set the field.
- The property combines the functionality of both getter and setter methods into a single unit.
- Therefore, the .NET developers don’t use getter and setter methods rather they make use of the properties to get and set the fields.

## Properties

### Class using getter/setter methods

```
public class PhoneCustomer
{
    private string _name;

    // getter
    public string GetName()
    {
        return _name;
    }

    // setter
    public void SetName(string name)
    {
        _name = name;
    }
}
```

### Class using property

```
public class PhoneCustomer
{
    private string _name;

    // property
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}
```

## Properties

- You use a property to get and set fields of a class.
- Each property has a corresponding private instance variable that stores the property’s value.
- It’s common to use the same name for the property and the related variable. The property name begins with an uppercase letter.
- You can code a *get accessor* to retrieve the value of the property and a *set accessor* to set the value.
- A property that has both a get and a set accessor is called a read/write property.
  - A property that has just a get accessor is called a read-only property.
  - A property that has just a set accessor is called a write-only property.

## Properties Examples

```
public string FirstName                        // A read/write property
{
       get { return _firstName; }
       set { _firstName = value; }
}
```

```
public string FirstName                        // A read-only property
{
       get { return _firstName; }
}
```

```
public string FirstName                        // A write-only property
{
       set { _firstName = value; }
}
```

```
customer.FirstName = "John";                   // Setting the property value
string firstName = customer.FirstName;  // Getting the property value
```

## Expression-Bodied Property Accessors

- With C# 7, you can also write property accessors as expression-bodied members.
- For example, the previously shown property `FirstName` can be written using `=>`.
- This new feature reduces the need to write curly brackets, and the `return` keyword is omitted with the `get` accessor.

```
private string _firstName;

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

## Auto-Implemented Properties

- If there isn't going to be any logic in the properties set and get, then auto-implemented properties can be used.
- Auto-implemented properties implement the backing member variable automatically.

```
public string FirstName { get; set; }
```

- The declaration of a private field is not needed. The compiler creates this automatically.
- You cannot access the field directly as you don't know the name the compiler generates.

## Access Modifiers for Properties

- C# does allow the `get` and `set` accessors to have differing access modifiers.
- This would allow a property to have a `public get` and a `private` or `protected set`.
- This can help control how or when a property can be set.
- In this code example, notice that the `set` has a `private` access modifier but the `get` does not.

```
public string FirstName
{
        get { return _firstName; }
        private set { _firstName = value; }
}
```

- In this case, the `get` takes the access level of the property.
- One of the accessors must follow the access level of the property.
- A compile error will be generated if the `get` accessor has the `protected` access level associated with it because that would make both accessors have a different access level from the property.

## Exercise

- Re-write the same `Account` class in the banking app using a property.
- Create an `Account` class, that has:
  - Private field `_balance`
  - Public property `Balance` which gets and sets the `_balance`
  - Public methods `Deposit()` and `Withdraw()` that receive the amount to deposit or withdraw
- The class `Account` should have two constructors, one that initializes the `Balance` to zero, and the other that initializes it to some specified amount.
- When the application runs, create an account either with no initial deposit, or with some initial deposit. Call the appropriate constructor for this.
- Handle the following issues:
  - Amount cannot be negative.
  - Withdraw amount cannot be more than `Balance`.

# Do It Yourself!

- **Exercise: Target-Heart-Rate Calculator:**
- While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors.
- According to the American Heart Association (AHA) (https://www.heart.org/en/healthy-living/fitness/fitness-basics/target-heart-rates), the formula for calculating your maximum heart rate in beats per minute is 220 minus your age in years.
- Your target heart rate is a range that is 50 – 85% of your maximum heart rate.

- Create a class called `HeartRates`. The class attributes should include the person's first name, last name, year of birth and the current year.
- Your class should have a constructor that receives this data as parameters.
- For each attribute provide a property with set and get accessors.
- The class also should include a property that calculates and returns the person's age (in years), a property that calculates and returns the person's maximum heart rate and properties that calculate and return the person's minimum and maximum target heart rates.
- Write an app that prompts for the person's information, instantiates an object of class `HeartRates` and displays the information from that object – including the person's first name, last name and year of birth – then calculates and displays the person's age in years, maximum heart rate and target-heart-rate range.

## Do It Yourself!

- **Exercise: Invoice Class:**
- Create a class called `Invoice` that a hardware store might use to represent an invoice for an item sold at the store.
- An `Invoice` should include four pieces of information as either instance variables or auto-implemented properties:
  - a part number (type `string`)
  - a part name (type `string`)
  - a quantity of the item being purchased (type `int`)
  - a price per item (`decimal`)
- Your class should have a constructor that initializes the four values.
- Provide a property with a get and set accessor for any instance variables.
- For the `Quantity` and `PricePerItem` properties, if the value passed to the set accessor is negative, the value of the instance variable should be left unchanged.
- Also, provide a method named `GetInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as a `decimal` value.

## 🛠 Do It Yourself!

- **Exercise: Set of Integers:**
- Create class `IntegerSet`.
- `IntegerSet` has an `int` array that can hold integers in the range `1-9`.
- The parameterless constructor initializes the array with random `int` values in the range `1-9`.
- Provide the following methods:
  a) Method `Union` creates a third set that's the union of two existing sets, Set-A and Set-B.
     i. Meaning, this new set contains all elements that are in Set-A or in Set-B (possibly both).

  b) Method `Intersection` creates a third set which is the intersection of two existing sets, Set-A and Set-B.
     ii. Meaning, this new set consists of all elements that are both in Set-A and Set-B.

  c) Method `ToString` returns a string containing a set as a list of numbers separated by spaces.
- Write an app to test class `IntegerSet`.

## 🛠 Do It Yourself!

- **Exercise: Tic-Tac-Toe:**
- Create class `TicTacToe` that will enable you to write a complete app to play the game of Tic-Tac-Toe.
- The class contains a private 3-by-3 rectangular array of integers.
- The constructor should initialize the empty board to all `0`s.
- Allow two human players.
- Wherever the first player moves, place a `1` in the specified square, and place a `2` wherever the second player moves.
- Each move must be to an empty square.
- After each move, determine whether the game has been won and whether it’s a draw.
- If you feel ambitious, modify your app so that the computer makes the moves for one of the players.
- Also, allow the player to specify whether he or she wants to go first or second.

# Structs in C#

## Introduction to Structs

- So far, you have seen how classes offer a great way to encapsulate objects in your program.
- Class objects are stored on the heap in a way that gives you much more flexibility in data lifetime but with a slight cost in performance.
- This performance cost is small thanks to the optimizations of managed heaps.
- However, in some situations all you really need is a small data structure.
- In those cases, a class provides more functionality than you need, and for best performance you probably want to use a struct.

## Introduction to Structs

- Consider the following example using a reference type:

```
public class Dimensions
{
	public double Length { get; }
	public double Width { get; }

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

## Introduction to Structs

- This previous code defines a class called `Dimensions`, which simply stores the length and width of an item.
- Suppose you're writing a furniture-arranging program that enables users to experiment with rearranging their furniture on the computer, and you want to store the dimensions of each item of furniture.
- All you have is two numbers, which you'll find convenient to treat as a pair rather than individually.
- There is no need for a lot of methods, or for you to be able to inherit from the class, and you certainly don't want to have the .NET runtime go to the trouble of bringing in the heap, with all the performance implications, just to store two `double`s.

## Introduction to Structs

- A struct would have been a better option.
- Just replace the keyword `class` with `struct`:

```
public struct Dimensions
{
      public double Length { get; }
      public double Width { get; }

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

Sheridan Get Creative

# Introduction to Structs

- Defining functions for structs is also the same as defining them for classes.
- You’ve already seen a constructor with the `Dimensions` struct.
- The following code demonstrates adding the property `Diagonal` to invoke the `Sqrt` method of the `Math` class:

```
public struct Dimensions
{
      public double Length { get; }
      public double Width { get; }

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

      public double Diagonal => Math.Sqrt(Length * Length + Width * Width);
}
```

Sheridan Get Creative

## Structs are Value Type

- Structs are value types, not reference types.
- This means they are stored in the stack and have the same lifetime restrictions as the simple data types.
- Structs do not support inheritance.
- If you do not supply a default constructor, the compiler automatically creates one and initializes the members to its default values.

## Structs Performance - Good or Bad

- The fact that structs are value types affects performance.
  - Depending on how you use struct, this can be good or bad.
- On the positive side, allocating memory for structs is very fast because this takes place on the stack.
- The same is true when they go out of scope.
- Structs are cleaned up quickly and don’t need to wait on garbage collection.
- On the negative side, whenever you pass a struct as a parameter or assign a struct to another struct (as in `A = B`, where `A` and `B` are structs), the full contents of the struct are copied, whereas for a class only the reference is copied.
- This results in a performance loss that varies according to the size of the struct, emphasizing the fact that structs are really intended for small data structures.

## Structs Performance - Good or Bad

- Note, that when passing a struct as a parameter to a method, you can avoid this performance loss by passing it as a `ref` parameter.
- In this case, only the address in memory of the struct will be passed in, which is just as fast as passing in a class.

Thank You

## Copyright

*The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.*

Sheridan Get Creative

## References

**Material has been taken from:**

- Visual C# 2012: How to Program:
- https://learning.oreilly.com/library/view/visual-c-2012/9780133380170/ch04.html

- Professional C# 7 and .NET Core 2.0:
- https://learning.oreilly.com/library/view/professional-c-7/9781119449270/c04.xhtml