Properties & Constructors

1

OUTLINE

- Fields
- Properties
  - Properties with backing fields
  - Expression-bodied properties
  - Auto-implemented properties
  - Calculated properties
  - Accessing properties
- Constructing objects
  - Constructors
  - Expression-bodied constructors
  - Constructor overloading
  - Constructor chaining
  - Object initialisers
  - Static constructors
- The ‘this’ keyword

2

2

FIELDS

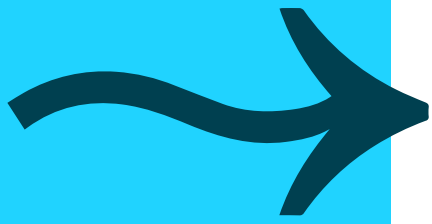
- A **field** is a variable that is declared directly in a class or struct
- A field can be an *instance* field: specific to the object instance
- A field can be a *static* field: shared amongst all objects of that type

```
public class Car {
    public string make;
    public string model;
    public int speed;
}
```

```
// fields do not provide validation beyond the datatype
Car c1 = new();
Car c2 = new();

c1.make = "Ford";
c2.make = "Ferrari";
c1.model = "Fiesta";
c2.model = "Flying submersible"; // not realistic
c1.speed = 9999; // not realistic
```

- A best practice recommendation is to declare fields as **private** or **protected** and define *properties*



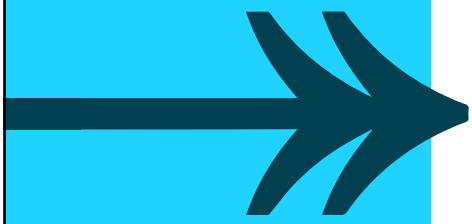
3

PROPERTIES

- **Properties** are special methods called *accessors*
- They provide a way to read, write, or compute the values of a private field whilst hiding the implementation

- A **get** property accessor is used to return the property value
- A **set** property accessor is used to assign a new **value**
- An **init** property accessor is used to assign a new **value** only during object construction

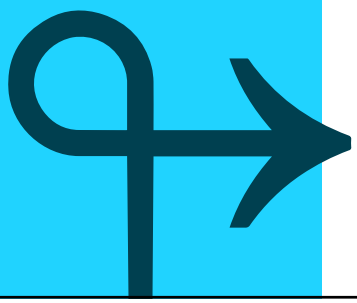
- Properties can be *read-write* (**get** and **set**)
- Properties can be *read-only* (**get**)
- Properties can be *write-only* (**set**)



4

4

PROPERTY SYNTAX



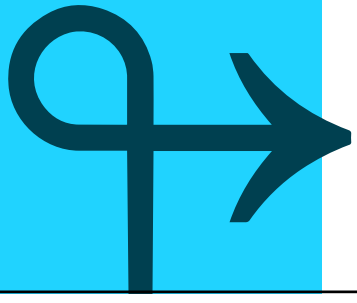
There are three different property syntaxes:

- Properties with backing fields
- Expression body definitions
- Auto-implemented properties

5

5

PROPERTIES WITH BACKING FIELDS



- The backing field holds the data
- The accessors can have different visibilities

```
public class Car
{
    private int speed;

    0 references
    public int Speed
    {
        get { return speed; }
        private set { speed = value; }
    }
}
```

6

6

EXPRESSION BODIED PROPERTIES

- Property accessors often consist of single-line statements that just assign or return the result of an expression
- Any single-line expression can be implemented using expression body syntax
- If the property is *read-only* (**get**) you can omit the **get** keyword
- If the property is *read-write* (**get** and **set**) you must use the **get** and **set** keywords

```
public class Car
{
    private int speed;

    0 references
    public int Speed => speed;
}
```

```
public class Car
{
    private int speed;

    0 references
    public int Speed
    {
        get => speed;
        private set => speed = value;
    }
}
```

7

7

AUTO-IMPLEMENTED PROPERTIES

- Use this syntax if there is no additional logic other than assigning or returning a value
- The C# compiler transparently creates the backing field for you and the implementation code to set / get to / from, the backing field
- You can use an optional initialiser to set a value

```
public class Car
{
    0 references
    public string Make { get; init; } = "Ford";
    1 reference
    public int Speed { get; private set; } = 42;
}
```

8

8

CALCULATED PROPERTY EXAMPLE

- **TempInDegreesCelsius** is a read-write auto-implemented property
- **TempInDegreesFahrenheit** is a calculated read-only property

```
public class Temperature
{
    1 reference
    public double TempInDegreesCelsius { get; set; }
    0 references
    public double TempInDegreesFahrenheit
    {
        get { return TempInDegreesCelsius * 1.8 + 32; }
    }
}
```

- **TempInDegreesFahrenheit** is a calculated read-only expression-bodied property

```
public class Temperature
{
    1 reference
    public double TempInDegreesCelsius { get; set; }
    0 references
    public double TempInDegreesFahrenheit => TempInDegreesCelsius * 1.8 + 32;
}
```

9

9

ACCESSING PROPERTIES

Property access looks like field access to the client:

```
// instantiate a Car object instance
Car c1 = new();

// set a property value
c1.Model = "Xr2i";

// get property values
Console.WriteLine(c1.Model);
Console.WriteLine(c1.Make);
Console.WriteLine(c1.Speed);
```

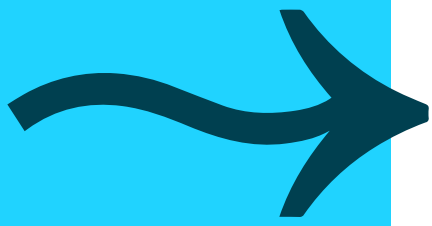
10

10

Constructing objects

11

OBJECT CONSTRUCTION



There are two ways to construct an object:

- Constructors
- Object initialisers

Constructors

- Define a constructor (or overloaded constructors) to include all combinations of *mandatory* fields
- This ensures the object is properly setup before being used

Object Initialisers

- Object initialisers can be used for *additional* optional fields that the object creator would like to set
- Object initialisers set properties or fields on the object *after* it has been constructed but before it is used

12

CONSTRUCTORS

- A constructor is called whenever a class or struct is created
- A class or struct can have multiple overloaded constructors
- A constructor is a method whose name is the same as the name of its type
- Constructors do not include a return type (not even void)
- Constructors are invoked using the **new** operator
- If no constructors are defined for a class or struct, the compiler creates a no-argument parameter-less constructor

```
Employee unknown = new(); //parameter-less constructor
Employee spiderman = new("Peter", "Parker", 1); // 3 arg constructor
```

13

13

CONSTRUCTOR EXAMPLE: EMPLOYEE WITH READ-ONLY FIELDS

You have an employee class with three mandatory values that should not be able to be changed once set: **firstName**, **lastName**, and **employeeID**.

Option 1

Define *readonly* fields and set their values in the constructor

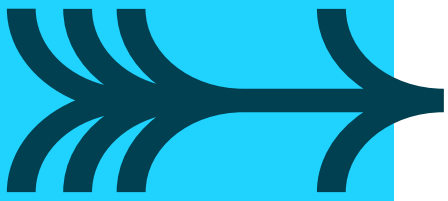
```
class Employee
{
    private readonly string firstName;
    private readonly string lastName;
    private readonly int employeeID;

    0 references
    public Employee(string firstName, string lastName, int employeeID)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.employeeID = employeeID;
    }
}
```

14

14

CONSTRUCTOR EXAMPLE: EMPLOYEE WITH AUTO-IMPLEMENTED PROPERTIES



You have an employee class with three mandatory values that should not be able to be changed once set: **firstName**, **lastName**, and **employeeID**.

Option 2

Define *auto-implemented properties* with **get** accessors only.

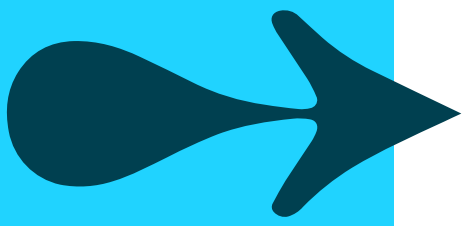
```
class Employee
{
    1 reference
    public Employee(string firstName, string lastName, int employeeID)
    {
        FirstName = firstName;
        LastName = lastName;
        EmployeeID = employeeID;
    }

    // auto-implemented properties
    1 reference
    public string FirstName { get; }
    1 reference
    public string LastName { get; }
    1 reference
    public int EmployeeID { get; }
}
```

15

15

CONSTRUCTOR EXAMPLE: EMPLOYEE WITH FULL PROPERTIES



Option 3

Define *full properties* with **get** and **init** accessors.

```
class Employee
{
    1 reference
    public Employee(string firstName, string lastName, int employeeID)
    {
        FirstName = firstName;
        LastName = lastName;
        EmployeeID = employeeID;
    }
    // full properties with backing fields
    2 references
    public string FirstName {
        get  { return firstName; }
        init {
            if (value.Length > 0)
            {
                firstName = value;
            }
        }
    }
    2 references
    public string LastName
    {
        get { return lastName.ToUpper(); }
        init
        {
            if (value.Length > 0)
            {
                lastName = value;
            }
        }
    }
    // backing fields
    private string firstName;
    private string lastName;

    // auto-implemented readonly property
    1 reference
    public int EmployeeID { get; }
}
```

16

16

EXPRESSION BODIED CONSTRUCTORS

If a constructor can be implemented as a single statement, you can use an expression body definition:

```
public class Location
{
    // private backing field
    private string locationName;

    // expression bodied constructor
    2 references
    public Location(string name) => Name = name;

    3 references
    public string Name
    {
        // expression bodied property accessors
        get => locationName;
        set => locationName = value;
    }
}
```

```
Location home = new("Home");
Location work = new("Work");
Console.WriteLine(home.Name);
Console.WriteLine(work.Name);
```

17

17

CONSTRUCTOR OVERLOADING

A constructor is a method and can therefore be overloaded:

```
public class Car
{
    // parameter-less constructor
    1 reference
    public Car()
    {
        Make = "Unkown";
    }
    // 1 arg constructor
    0 references
    public Car(string make)
    {
        Make = make;
    }
    // 2 arg constructor
    0 references
    public Car(string make, string model)
    {
        Make = make; ;
        Model = model;
    }
```

18

18

CONSTRUCTOR CHAINING

A constructor can invoke another constructor in the same object (constructor chaining) to avoid duplicating code, using the **this** keyword:

```
public class Car
{
    // parameter-less constructor
    1 reference
    public Car() :this("Unknown")
    { }

    // 1 arg constructor
    2 references
    public Car(string make) : this(make, "Unknown model")
    { }

    // 2 arg constructor
    2 references
    public Car(string make, string model)
    {
        Make = make; ;
        Model = model;
    }
```

```
Car c1 = new();
Car c2 = new("Audi");
Car c3 = new("BMW", "X5");

Console.WriteLine($"Car {nameof(c1)} is make: {c1.Make} and model: {c1.Model}");
Console.WriteLine($"Car {nameof(c2)} is make: {c2.Make} and model: {c2.Model}");
Console.WriteLine($"Car {nameof(c3)} is make: {c3.Make} and model: {c3.Model}");
// Output:
// Car c1 is make: Unknown and model: Unknown model
// Car c2 is make: Audi and model: Unknown model
// Car c3 is make: BMW and model: X5
```

19

19

OBJECT INITIALISERS

- To avoid creating many overloaded constructors, *object initialisers* can be used to initialise an object into a ready state
- Object initialisers are often used for *optional* values and constructors are defined for *mandatory* values
- An object initialiser invokes the parameterless constructor unless an explicit constructor is specified
- A **struct** always has a parameterless constructor
- A **class** needs to have an explicit parameterless constructor defined if at least one explicit non-parameterless constructor is defined

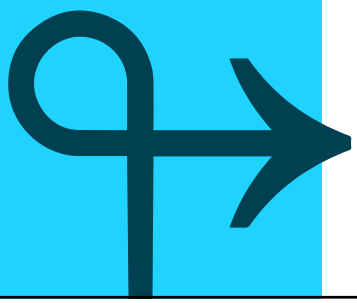
```
// parameter-less constructor is used implicitly with an object initializer
Car c4 = new Car { Make = "Audi", Model = "TT", Speed = 70 };
Console.WriteLine($"Car {nameof(c4)} is make: {c4.Make} and model: {c4.Model} and speed: {c4.Speed}");

// an explicit constructor can be used with an object initializer
Car c5 = new Car("Audi","TT") { Speed = 70 };
Console.WriteLine($"Car {nameof(c5)} is make: {c5.Make} and model: {c5.Model} and speed: {c5.Speed}");
```

20

20

CONSTRUCTOR ACCESS MODIFIERS



Constructors can be marked as

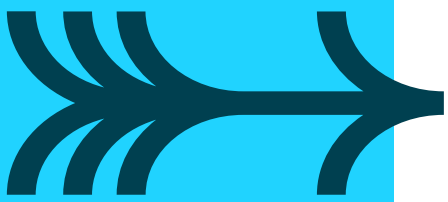
- public
- private
- internal
- protected internal
- private protected

These define how users of the class can construct an instance of the class.

21

21

STATIC CONSTRUCTORS



- Constructors can be marked as **static**
- A **static** constructor *initialises* any *static data* or performs an *action* that needs to be performed only *once*
- Static constructors do not have an *access modifier* or *parameters*
- Any class or struct can only have *one* static constructor
- Static constructors therefore **cannot** be *overloaded*
- Static constructors are called *automatically* by the CLR

```
class Employee
{
    static readonly string companyName;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    0 references
    static Employee()
    {
        companyName = "QA Ltd";
    }
}
```

22

22

THE 'THIS' KEYWORD

The **this** keyword has many uses:

- To qualify members hidden by similar names
- To pass an object as a parameter to other methods
- To chain constructors

```
class Employee
{
    private readonly string firstName;

    0 references
    public Employee(string firstName)
    {
        this.firstName = firstName;
    }
}
```

```
public Employee()
{
    BookSeatInTheOffice(this);
}
```

```
// parameter-less constructor
1 reference
public Car() :this("Unknown")
{ }

// 1 arg constructor
2 references
public Car(string make) : this(make, "Unknown model")
{ }
```

23

SUMMARY

- Fields
- Properties
  - Properties with backing fields
  - Expression-bodied properties
  - Auto-implemented properties
  - Calculated properties
  - Accessing properties
- Constructing objects
  - Constructors
  - Expression-bodied constructors
  - Constructor overloading
  - Constructor chaining
  - Object initialisers
  - Static constructors
- The 'this' keyword

24

25