1

OUTLINE

- Methods
- Positional parameters
- Named and optional params
- Method overloading
- Return values
- Expression-bodied methods
- Passing parameters
- Passing value-type parameters
  - By value
  - By ref : ‘ref’, ‘out’ and ‘in’
- Passing reference-type parameters
  - By value
  - By ref
- Static methods and the using static directive
- Extension methods

2

2

METHODS

- A method is a code block that contains a series of statements
- Methods are called (invoked) by a program
- Methods are declared in a *class*, *struct*, or *interface* with:
  - An access modifier such as **public** or **private**
  - Optional modifiers such as **abstract**
  - The return type such as **void** or **int**
  - The name of the method
  - Any method parameters in brackets
- This definition is known as the method signature
- Methods can be passed arguments that map to parameters defined in the method signature
- Every C# application has a method called **Main** that is the entry point for the application

3

3

METHODS: POSITIONAL PARAMETERS

Method definition:

```
class Car
{
    2 references
    public void Register(string name, string address, string postCode, string country)
    {
        // store this in the DVLA database in Swansea
    }
}
```

Instantiate objects and invoke the method:

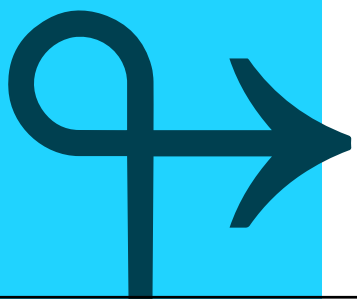
```
Car julieCar = new Car();
julieCar.Register("Julie Dooley", "1 Main Street", "CH12 9DL", "UK");


Car lisaCar = new Car();
lisaCar.Register("Lisa Simpson", "742 Evergreen Terrace", "97394", "USA");
```

- Name, address, postcode, and country are the method's parameters. When the method is invoked, the parameters are passed positionally:
  - The zeroth argument 'Julie Dooley' maps to parameter *name*
  - The first argument '1 Main Street' maps to parameter *address*
  - The second argument 'CH12 9DL' maps to parameter *postcode*
  - The third argument 'UK' maps to parameter *country*

4

4

NAMED AND OPTIONAL PARAMETERS



- Parameters can be passed by name
- When **named**, the parameters can be passed in any order

```
Car batMobile = new();
batMobile.Register(name: "Bruce Wayne", postCode: "Unknown",
           address: "1007 Mountain Drive, Gotham", country: "USA");
```

- Parameters can be given a default value to make them **optional**:

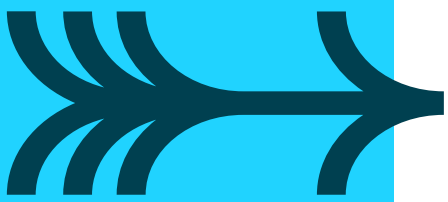
```
public void Register(string name, string address, string postCode, string country = "USA")
{
}
```

```
batMobile.Register(name: "Bruce Wayne", postCode: "Unknown",
           address: "1007 Mountain Drive, Gotham");
```

5

5

METHOD OVERLOADING



You might need to start different cars in different ways:

```
1 reference
public void Start() { }
0 references
public void Start(string keycode) { }
0 references
public void Start(Token token) { }
0 references
public void Start(Fingerprint print) { }
```

You can define different signatures for the same method.

This is an **overloaded method**:

```
batMobile.Start();
▲ 1 of 4 ▼ void Car.Start()
```

```
batMobile.Start();
▲ 2 of 4 ▼ void Car.Start(Fingerprint print)
```

```
batMobile.Start();
▲ 3 of 4 ▼ void Car.Start(string keycode)
```

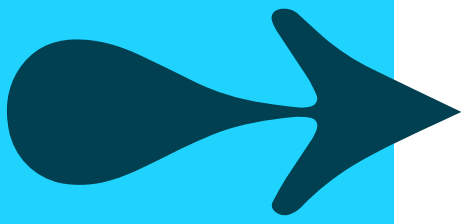
```
batMobile.Start();
▲ 4 of 4 ▼ void Car.Start(Token token)
```

6

6

METHOD RETURN VALUES

- Methods can return a value to the caller if the return type is not **void**
- The value is returned using the **return** keyword
- The type of the value must match the method signature
- The **return** keyword stops execution of the method
- A **void** method can use **return** without a value to stop execution
- A **void** method without return will stop execution at the end of the code block

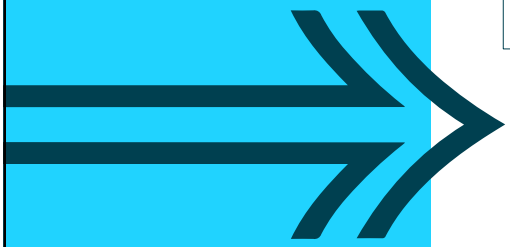


7

7

METHOD RETURN VALUES: EXAMPLE

```
// returns an int
0 references
public int AddTwoNumbers(int number1, int number2) // 2 parameters
{
    return number1 + number2;// execution stops and an int is returned
}

// returns a void
0 references
public void SquareANumber(int number) // 1 parameter
{
    number *= number;
    Console.WriteLine(number);
    return;// execution stops here
    Console.WriteLine(number);// unreachable code
}
```



8

8

EXPRESSION BODIED MEMBERS

- Expression body definitions let you provide a member's implementation in a very concise form
- Use an expression body definition whenever the logic required consists of a single expression
- Syntax:
  member => expression

- The following C# members support expression body definitions:
  - Methods
  - Properties
  - Constructors
  - Finalizers
  - Indexers

9

9

EXPRESSION BODIED METHODS

- An expression-bodied method consists of a single expression that returns a value whose type matches the method's return type, or, for void methods, performs some operation:

```
0 references
public int AddTwoNumbers(int number1, int number2) => number1 + number2;

0 references
public void SquareANumber(int number1) => Console.WriteLine(number1 *= number1);
```

- Block body methods

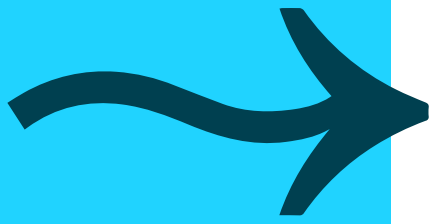
```
public int AddTwoNumbers(int number1, int number2)
{
    return number1 + number2;
}
```

```
public void SquareANumber(int number)
{
    Console.WriteLine(number *= number);
    return;
}
```

10

10

PASSING PARAMETERS



Value types

- When an instance of a **value** type is passed as a parameter to a method, its *copy* is passed instead of the instance
- Changes to the argument within the called method have *no effect* on the original instance in the calling code

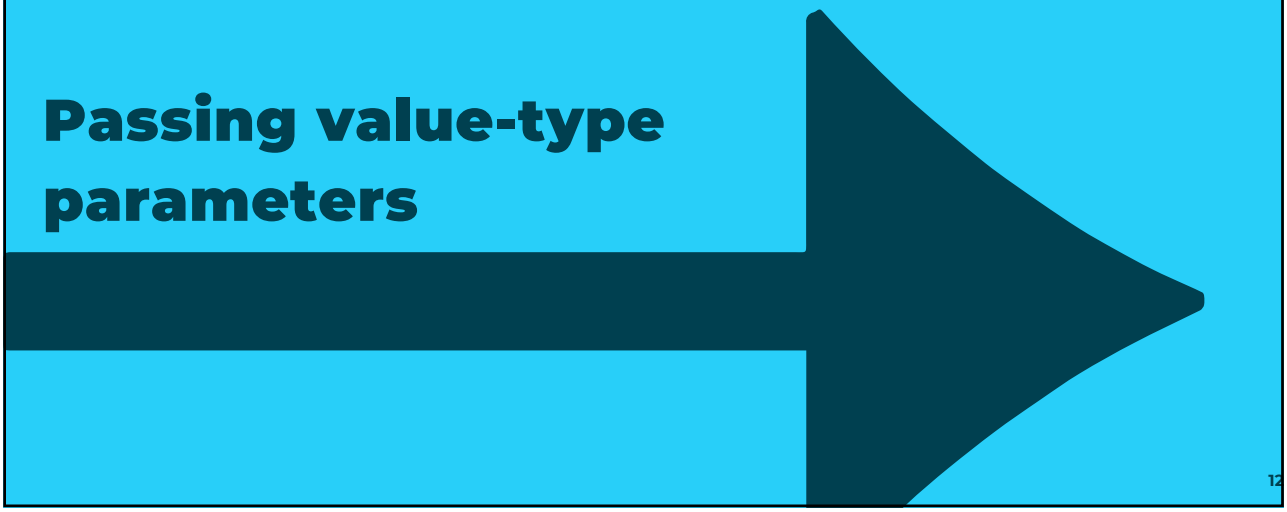
Reference types

- When an instance of a **reference** type is passed as a parameter to a method, a *reference* to the object is passed
- Changes to a member of the argument within the called method *are reflected* in the argument in the calling code

11

11

Passing value-type parameters



12

12

PASSING PARAMETERS: VALUE TYPES BY VALUE

The **value** type variable x is *passed by value* by default, so a copy of the value 7 is passed. This is then squared to give the value of 49 within the called method. The original variable is unchanged.

```
internal class PassingParams
{
    1 reference
    public void SquareANumber(int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

```
PassingParams pp = new PassingParams();

int x = 7;
System.Console.WriteLine("The value before calling the method: {0}", x);

pp.SquareANumber(x);  // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", x);

// Output:
// The value before calling the method: 7
// 49
// The value after calling the method: 7
```

13

13

PASSING PARAMETERS: VALUE TYPES BY REFERENCE ‘REF’

The value type variable x is *passed by reference* using the **ref** keyword in both the method declaration and the method call. A **reference** to the variable is passed. The value this reference points to is then squared to give the value of 49. The original variable is **changed**.

```
internal class PassingParams
{
    1 reference
    public void SquareANumber(ref int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

```
PassingParams pp = new PassingParams();

int x = 7;
System.Console.WriteLine("The value before calling the method: {0}", x);

pp.SquareANumber(ref x);  // Passing the variable by reference
System.Console.WriteLine("The value after calling the method: {0}", x);

// Output:
// The value before calling the method: 7
// 49
// The value after calling the method: 49
```

14

14

PASSING PARAMETERS: VALUE TYPES BY REFERENCE 'OUT'

The value type variables are *passed by reference* using the **out** keyword in both the method declaration and the method call. Unlike **ref**, variables do not need to be initialised before being passed.

```
void OutExampleMethod(out int number, out string text, out string optionalString)
{
    number = 42;
    text = "I'm output text";
    optionalString = null;
}

int argNumber;
string argText, argOptionalString;
OutExampleMethod(out argNumber, out argText, out argOptionalString);

Console.WriteLine(argNumber);
Console.WriteLine(argText);
Console.WriteLine(argOptionalString == null);

// Output:
//  42
//  I'm output text
//  True
```

15

15

PASSING PARAMETERS: VALUE TYPES BY REFERENCE 'OUT'

From C# 7, you can declare the **out** variables in the argument list of the method call rather than having to declare them beforehand:

```
int argNumber;
string argText, argOptionalString;
OutExampleMethod(out argNumber, out argText, out argOptionalString);
```

```
OutExampleMethod(out int argNumber, out string argText, out string argOptionalString);
```

16

16

TUPLES INSTEAD OF OUT

Out parameters are used to return multiple items from a method

An alternative is to return a **collection** when the values belong in a group of the same type e.g., List<string>

Another alternative is to return a **tuple:**

```
(int number, string text, string? optionalString)  OutExampleMethod()
{
    var number = 42;
    var text = "I'm output text";
    string? optionalString = null;
    return (number, text, optionalString);
}

var outputs = OutExampleMethod();
Console.WriteLine($"Outputs: number is {outputs.number}, text is {outputs.text}");
Console.WriteLine($"Outputs: optional string is {outputs.optionalString ?? "NULL"}");
```

A **tuple** is concise syntax to group multiple data elements in a lightweight data structure.

The most common use case is as a method return type within private or internal utility methods.

17

17

PASSING PARAMETERS: VALUE TYPES AS 'IN'

- The value type variable x is *passed by reference* using the **in** keyword
- **in** ensures the argument cannot be modified by the called method
- The **in** keyword is optional in the calling code because it is the default passing mechanism

```
PassingParams pp = new PassingParams();

int x = 5;
System.Console.WriteLine("The value before calling the method: {0}", x);

pp.SquareANumber(x);  // Passing the variable by reference with 'in'
System.Console.WriteLine("The value after calling the method: {0}", x);

// Output:
// The value before calling the method: 5
// 25
// The value after calling the method: 5
```

```
1 reference
public void SquareANumber(in int number)
{
    Console.WriteLine(number * number);
    return;
}
```

18

18

PASSING PARAMETERS: VALUE TYPES AS 'IN'

- The called method cannot modify the **in** parameter, so the code must be changed to reflect this restriction:

```
1 reference
public void SquareANumber(in int number)
{
    Console.WriteLine(number *= number);
    return;
}
```

(parameter) in int number

CS8331: Cannot assign to variable 'in int' because it is a readonly variable

```
1 reference
public void SquareANumber(in int number)
{
    Console.WriteLine(number * number);
    return;
}
```

19

19

Passing reference-type parameters

20

20

PASSING PARAMETERS: REFERENCE TYPES

When a **reference-type** parameter is *passed by value*, it is possible to change members belonging to the object but not the object itself.

When a **reference-type** parameter is *passed by reference*, it is possible to change the value of the object itself.

```
internal class SwappingStrings
{
    1 reference
    public void SwapStringsByValue(string s1, string s2)
    // The string parameter is passed by value.
    // Any changes on parameters will not affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }
    0 references
    public void SwapStringsByReference(ref string s1, ref string s2)
    // The string parameter is passed by reference.
    // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }
}
```

21

21

PASSING PARAMETERS: REF TYPES BY VALUE

The **reference-type** variables str1 and str2 are *passed by value*.

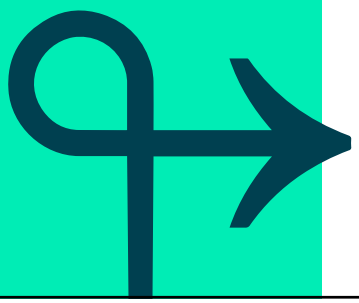
```
public void SwapStringsByValue(string s1, string s2)
// The string parameter is passed by value.
// Any changes on parameters will not affect the original variables.
{
    string temp = s1;
    s1 = s2;
    s2 = temp;
    System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
}
```

```
SwappingStrings ss = new SwappingStrings();

// ref types by value
string str1 = "Diana";
string str2 = "Prince";
System.Console.WriteLine("Before swapping: {0} {1}", str1, str2);

ss.SwapStringsByValue(str1, str2);   // Passing strings by value
System.Console.WriteLine("After swapping: {0} {1}", str1, str2);


/* Output:
    Before swapping: Diana Prince
    Inside the method: Prince Diana
    After swapping: Diana Prince
*/
```

22

22

PASSING PARAMETERS: REF TYPES BY REF



The **reference-type** variables str1 and str2 are *passed by reference*.

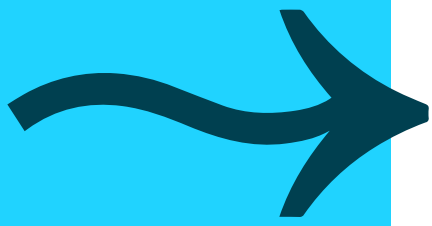
```
public void SwapStringsByReference(ref string s1, ref string s2)
// The string parameter is passed by reference.
// Any changes on parameters will affect the original variables.
{
    string temp = s1;
    s1 = s2;
    s2 = temp;
    System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
}
```

```
SwappingStrings ss = new SwappingStrings();

// ref types by reference
string str1 = "Diana";
string str2 = "Prince";
System.Console.WriteLine("Before swapping: {0} {1}", str1, str2);

ss.SwapStringsByReference(ref str1, ref str2);   // Passing strings by reference
System.Console.WriteLine("After swapping: {0} {1}", str1, str2);

/* Output:
    Before swapping: Diana Prince
    Inside the method: Prince Diana
    After swapping: Prince Diana
*/
```

23

23

STATIC METHODS



- Use the **static** modifier to declare a static member such as a class or method
- A **static** method belongs to the type itself rather than to a specific instance of the object
- Therefore, **static** methods do not require an object to be instantiated
- A **static** method can't be referenced through an instance
- **WriteLine** is an example of a **static** method of the **Console** class

```
Console.WriteLine();
```

class System.Console
Represents the standard input, output, and error streams for console applications.

24

24

STATIC METHODS: EXAMPLE

- The **System.Math** class provides constants and static methods for common mathematical functions
- The directive **using System;** is used in this example

```
int x = 5;
int y = 7;

int lowest = Math.Min(x, y);
int highest = Math.Max(x, y);

Console.WriteLine($"The lowest value is {lowest}");
Console.WriteLine($"The highest value is {highest}");
/* Output:
 * The lowest value is 5
 * The highest value is 7
 */

double price = 9.99;
double priceFloor = Math.Floor(price);
double priceRounded = Math.Round(price, 0);

Console.WriteLine(priceFloor); // 9
Console.WriteLine(priceRounded); // 10
```

25

25

STATIC METHODS: EXAMPLE 'USING STATIC'

- The directive **using static System.Math;** is used in this example

```
// using static System.Math
int x = 5;
int y = 7;

int lowest = Min(x, y);
int highest = Max(x, y);

Console.WriteLine($"The lowest value is {lowest}");
Console.WriteLine($"The highest value is {highest}");
/* Output:
 * The lowest value is 5
 * The highest value is 7
 */

double price = 9.99;
double priceFloor = Floor(price);
double priceRounded = Round(price, 0);

Console.WriteLine(priceFloor); // 9
Console.WriteLine(priceRounded); // 10
```

26

26

INSTANCE METHOD EXAMPLE

- For *instance* methods, instantiate an object, then call the method using that object instance

```
internal class PassingParams
{
    1 reference
    public void SquareANumber(int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

```
// instantiate object
PassingParams pp = new PassingParams();

int x = 7;
pp.SquareANumber(x); // call instance method
```

27

27

CREATE AND USE A STATIC METHOD

- For *static* methods, use the **static** modifier on the method definition, then call the method using the class
- Issue a *using directive* to import the static members of the class to make the code less verbose

```
internal class PassingParams
{
    1 reference
    public static void SquareANumber(int number)
    {
        Console.WriteLine(number *= number);
        return;
    }
}
```

```
int x = 7;
PassingParams.SquareANumber(x); // call static method

// using static PassingParams
int y = 10;
SquareANumber(y); // call static method
```

28

28

EXTENSION METHODS

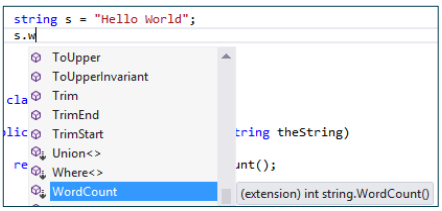
- Extension methods enable a type's functionality to be extended without editing the source code or inheriting from the type
- Extension methods are defined as **static** *methods* in a **static** *class*
- The first parameter defines the *type* that the method 'extends'
- The parameter type is preceded by the **this** modifier

29

29

EXTENSION METHOD EXAMPLE

```
static class StringUtils {
        public static int WordCount(this string theString)
        {return theString.Split(' ').Count(); }    }
```

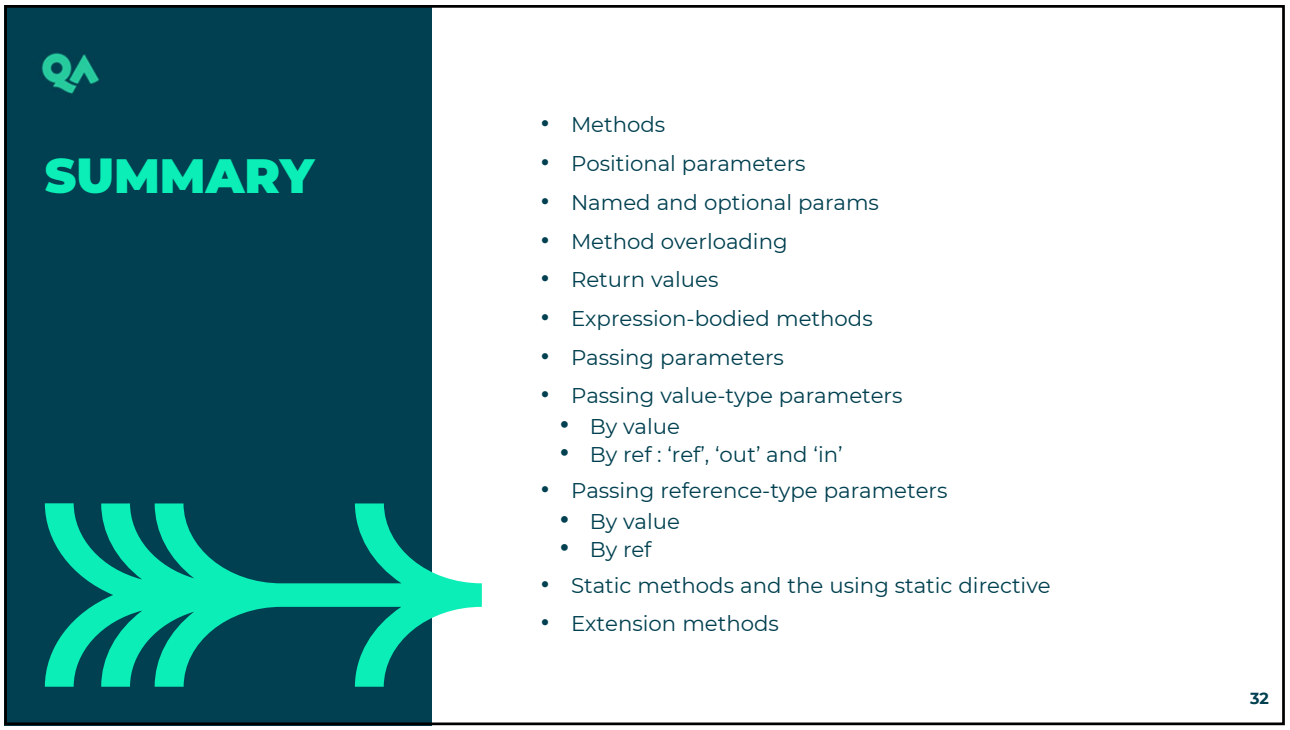


30

30


ACTIVITY:
Exercise 7
31

31


SUMMARY
Methods
Positional parameters
Named and optional params
Method overloading
Return values
Expression-bodied methods
Passing parameters
Passing value-type parameters
By value
By ref : 'ref', 'out' and 'in'
Passing reference-type parameters
By value
By ref
Static methods and the using static directive
Extension methods
32

32