**Csharp for advanced level:**

# C# Methods

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once, and use it many times.

---

## Create a Method

A method is defined with the name of the method, followed by parentheses **()**. C# provides some pre-defined methods, which you already are familiar with, such as `Main()`, but you can also create your own methods to perform certain actions:

### Example

Create a method inside the Program class:

```
class Program
{
  static void MyMethod()
  {
    // code to be executed
  }
}
```

### Example Explained

- `MyMethod()` is the name of the method
- `static` means that the method belongs to the Program class and not an object of the Program class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

# Call a Method

To call (execute) a method, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, `MyMethod()` is used to print a text (the action), when it is called:

## Example

Inside `Main()`, call the `myMethod()` method:

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
}

// Outputs "I just got executed!"
```

A method can be called multiple times:

## Example

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
  MyMethod();
  MyMethod();
}

// I just got executed!
// I just got executed!
// I just got executed!
```

# C# Method Parameters

## Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `string` called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

## Example

```
static void MyMethod(string fname)
{
  Console.WriteLine(fname + " Refsnes");
}

static void Main(string[] args)
{
  MyMethod("Liam");
  MyMethod("Jenny");
  MyMethod("Anja");
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

# Multiple Parameters

You can have as many parameters as you like, just separate them with commas:

## Example

```
static void MyMethod(string fname, int age)
{
  Console.WriteLine(fname + " is " + age);
}

static void Main(string[] args)
{
  MyMethod("Liam", 5);
  MyMethod("Jenny", 8);
  MyMethod("Anja", 31);
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

# Default Parameter Value

You can also use a default parameter value, by using the equals sign (=).

If we call the method without an argument, it uses the default value ("Norway"):

## Example

```
static void MyMethod(string country = "Norway")
{
  Console.WriteLine(country);
}

static void Main(string[] args)
{
  MyMethod("Sweden");
  MyMethod("India");
  MyMethod();
  MyMethod("USA");
}

// Sweden
// India
// Norway
// USA
```

# C# Return Value

## Return Values

In the previous page, we used the `void` keyword in all examples, which indicates that the method should not return a value.

If you want the method to return a value, you can use a primitive data type (such as `int` or `double`) instead of `void`, and use the `return` keyword inside the method:

## Example

```
static int MyMethod(int x)
{
  return 5 + x;
}

static void Main(string[] args)
{
  Console.WriteLine(MyMethod(3));
}

// Outputs 8 (5 + 3)
```

This example returns the sum of a method's **two parameters**:

## Example

```
static int MyMethod(int x, int y)
{
  return x + y;
}

static void Main(string[] args)
{
```

```
  Console.WriteLine(MyMethod(5, 3));
}


// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

## Example

```
static int MyMethod(int x, int y)
{
  return x + y;
}

static void Main(string[] args)
{
  int z = MyMethod(5, 3);
  Console.WriteLine(z);
}

// Outputs 8 (5 + 3)
```

# Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

## Example

```
int MyMethod(int x)

float MyMethod(float x)

double MyMethod(double x, double y)
```

Consider the following example, which have two methods that add numbers of different type:

## Example

```
static int PlusMethodInt(int x, int y)

{

  return x + y;

}



static double PlusMethodDouble(double x, double y)

{

  return x + y;

}



static void Main(string[] args)

{

  int myNum1 = PlusMethodInt(8, 5);

  double myNum2 = PlusMethodDouble(4.3, 6.26);

  Console.WriteLine("Int: " + myNum1);

  Console.WriteLine("Double: " + myNum2);
```

```
}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the `PlusMethod` method to work for both `int` and `double`:

## Example

```
static int PlusMethod(int x, int y)
{
  return x + y;
}

static double PlusMethod(double x, double y)
{
  return x + y;
}

static void Main(string[] args)
{
  int myNum1 = PlusMethod(8, 5);
  double myNum2 = PlusMethod(4.3, 6.26);
  Console.WriteLine("Int: " + myNum1);
  Console.WriteLine("Double: " + myNum2);
}
```

# C# - What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

# Classes and Objects

You learned from the previous chapter that C# is an object-oriented programming language.

Everything in C# is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

---

# Create a Class

To create a class, use the `class` keyword:

Create a class named "`Car`" with a variable `color`:

```
class Car

{

  string color = "red";
```

# Create an Object

An object is created from a class. We have already created the class named `Car`, so now we can use this to create objects.

To create an object of `Car`, specify the class name, followed by the object name, and use the keyword `new`:

## Example

Create an object called "`myObj`" and use it to print the value of `color`:

```
class Car

{

  string color = "red";



  static void Main(string[] args)

  {

    Car myObj = new Car();

    Console.WriteLine(myObj.color);

  }

}
```

# Class Members

Fields and methods inside classes are often referred to as "Class Members":

## Example

Create a `Car` class with three class members: **two fields** and **one method**.

```
// The class
class MyClass
{
  // Class members
  string color = "red";         // field
  int maxSpeed = 200;           // field
  public void fullThrottle()    // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }
}
```

---

# Fields

In the previous chapter, you learned that variables inside a class are called fields, and that you can access them by creating an object of the class, and by using the dot syntax (`.`).

The following example will create an object of the `Car` class, with the name `myObj`. Then we print the value of the fields `color` and `maxSpeed`:

## Example

```
class Car
{
  string color = "red";
  int maxSpeed = 200;
```

```
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
  }
}
```

You can also leave the fields blank, and modify them when creating the object:

## Example

```
class Car
{
  string color;
  int maxSpeed;

  static void Main(string[] args)
  {
    Car myObj = new Car();
    myObj.color = "red";
    myObj.maxSpeed = 200;
    Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
  }
```

```
}
```

This is especially useful when creating multiple objects of one class:

## Example

```
class Car
{
  string model;
  string color;
  int year;

  static void Main(string[] args)
  {
    Car Ford = new Car();
    Ford.model = "Mustang";
    Ford.color = "red";
    Ford.year = 1969;

    Car Opel = new Car();
    Opel.model = "Astra";
    Opel.color = "white";
    Opel.year = 2005;

    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
```

```
  }
}
```

# Constructors

A constructor is a **special method** that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

## Example

Create a constructor:

```
// Create a Car class
class Car
{
  public string model;  // Create a field

  // Create a class constructor for the Car class
  public Car()
  {
    model = "Mustang"; // Set the initial value for model
  }

  static void Main(string[] args)
  {
    Car Ford = new Car();  // Create an object of the Car Class (this will call the constructor)
    Console.WriteLine(Ford.model);  // Print the value of model
  }
}
```

```
// Outputs "Mustang"
```

# Access Modifiers

By now, you are quite familiar with the `public` keyword that appears in many of our examples:

```
public string color;
```

The `public` keyword is an **access modifier**, which is used to set the access level/visibility for classes, fields, methods and properties.

C# has the following access modifiers:

| Modifier | Description |
| --- | --- |
| `public` | The code is accessible for all classes |
| `private` | The code is only accessible within the same class |
| `protected` | The code is accessible within the same class, or in a class that is inherited from that cla<br>more about inheritance in a later chapter |
| `internal` | The code is only accessible within its own assembly, but not from another assembly. Y<br>more about this in a later chapter |

There's also two combinations: `protected internal` and `private protected`.

For now, lets focus on `public` and `private` modifiers.

# Private Modifier

If you declare a field with a `private` access modifier, it can only be accessed within the same class:

## Example

```
class Car

{

  private string model = "Mustang";


  static void Main(string[] args)

  {

    Car myObj = new Car();

    Console.WriteLine(myObj.model);

  }

}
```

The output will be:

```
Mustang
```

# nheritance (Derived and Base Class)

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- **Derived Class** (child) - the class that inherits from another class
- **Base Class** (parent) - the class being inherited from

To inherit from a class, use the `:` symbol.

In the example below, the `Car` class (child) inherits the fields and methods from the `Vehicle` class (parent):

## Example

```
class Vehicle  // base class (parent)
{
  public string brand = "Ford";  // Vehicle field
  public void honk()             // Vehicle method
  {
    Console.WriteLine("Tuut, tuut!");
  }
}

class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}

class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();
```

```
    // Display the value of the brand field (from the Vehicle class) and
the value of the modelName from the Car class

    Console.WriteLine(myCar.brand + " " + myCar.modelName);

  }

}
```

---

# The sealed Keyword

If you don't want other classes to inherit from a class, use the `sealed` keyword:

If you try to access a `sealed` class, C# will generate an error:

```
sealed class Vehicle

{

  ...

}


class Car : Vehicle

{

  ...

}
```

The error message will be something like this:

```
'Car': cannot derive from sealed type 'Vehicle'
```

# Polymorphism and Overriding Methods

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; **Inheritance** lets us inherit fields and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called `Animal` that has a method called `animalSound()`. Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

## Example

```
class Animal  // Base class (parent)
{
  public void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Pig : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public void animalSound()
```

```
  {
    Console.WriteLine("The dog says: bow wow");
  }
}
```

Now we can create `Pig` and `Dog` objects and call the `animalSound()` method on both of them:

## Example

```
class Animal  // Base class (parent)
{
  public void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Pig : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}
```

```
class Dog : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}


class Program
{
  static void Main(string[] args)
  {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

The output will be:

```
The animal makes a sound
The animal makes a sound
The animal makes a sound
```

## Example

```
class Animal  // Base class (parent)
{
  public virtual void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Pig : Animal  // Derived class (child)
{
  public override void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public override void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
```

```
}


class Program
{
  static void Main(string[] args)
  {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

The output will be:

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

# Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The `abstract` keyword is used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal
{
  public abstract void animalSound();
  public void sleep()
  {
    Console.WriteLine("Zzz");
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // Will generate an error (Cannot create an instance of the abstract class or interface 'Animal')
```

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the Polymorphism chapter to an abstract class.

## Example

```
// Abstract class
abstract class Animal
{
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep()
  {
    Console.WriteLine("Zzz");
  }
}

// Derived class (inherit from Animal)
class Pig : Animal
{
  public override void animalSound()
  {
    // The body of animalSound() is provided here
    Console.WriteLine("The pig says: wee wee");
  }
}

class Program
{
  static void Main(string[] args)
```

```
  {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();  // Call the abstract method
    myPig.sleep();  // Call the regular method
  }
}
```

# Interfaces

Another way to achieve abstraction in C#, is with interfaces.

An `interface` is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies):

## Example

```
// interface
interface Animal
{
  void animalSound(); // interface method (does not have a body)
  void run(); // interface method (does not have a body)
}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class. To implement an interface, use the `:` symbol (just like with inheritance). The body of the interface method is provided by the "implement" class. Note that you do not have to use the `override` keyword when implementing an interface:

## Example

```
// Interface
interface IAnimal
```

```
{
  void animalSound(); // interface method (does not have a body)
}

// Pig "implements" the IAnimal interface
class Pig : IAnimal
{
  public void animalSound()
  {
    // The body of animalSound() is provided here
    Console.WriteLine("The pig says: wee wee");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
  }
}
```

# C# Enums

An enum is a special "class" that represents a group of **constants** (unchangeable/read-only variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the enum items with a comma:

## Example

```
enum Level
{
  Low,
  Medium,
  High
}
```

You can access enum items with the **dot** syntax:

```
Level myVar = Level.Medium;
Console.WriteLine(myVar);
```

---

# Enum inside a Class

You can also have an enum inside a class:

## Example

```
class Program
{
  enum Level
  {
    Low,
    Medium,
    High
  }
```

```
  static void Main(string[] args)
  {
    Level myVar = Level.Medium;
    Console.WriteLine(myVar);
  }
}
```

The output will be:

```
Medium
```

# Enum Values

By default, the first item of an enum has the value 0. The second has the value 1, and so on.

To get the integer value from an item, you must explicitly convert the item to an `int`:

## Example

```
enum Months
{
  January,    // 0
  February,   // 1
  March,      // 2
  April,      // 3
  May,        // 4
  June,       // 5
  July        // 6
}
```

```
static void Main(string[] args)
{
  int myNum = (int) Months.April;
  Console.WriteLine(myNum);
}
```

The output will be:

```
3
```

# C# Files

## Working With Files

The `File` class from the `System.IO` namespace, allows us to work with files:

### Example

```
using System.IO;  // include the System.IO namespace


File.SomeFileMethod();  // use the file class with methods
```

The `File` class has many useful methods for creating and getting information about files. For example:

| Method | Description |
| --- | --- |
| `AppendText()` | Appends text at the end of an existing file |
| `Copy()` | Copies a file |

| | |
|---|---|
| `Create()` | Creates or overwrites a file |
| `Delete()` | Deletes a file |
| `Exists()` | Tests whether the file exists |
| `ReadAllText()` | Reads the contents of a file |
| `Replace()` | Replaces the contents of a file with the contents of another file |
| `WriteAllText()` | Creates a new file and writes the contents to it. If the file already exists, i overwritten. |

---

# Write To a File and Read It

In the following example, we use the `WriteAllText()` method to create a file named "filename.txt" and write some content to it. Then we use the `ReadAllText()` method to read the contents of the file:

## Example

```
using System.IO;  // include the System.IO namespace


string writeText = "Hello World!";  // Create a text string
```

```
File.WriteAllText("filename.txt", writeText);  // Create a file and write
the content of writeText to it



string readText = File.ReadAllText("filename.txt");  // Read the contents
of the file

Console.WriteLine(readText);  // Output the content
```

The output will be:

```
Hello World!
```

# C# Exceptions - Try..Catch

## C# Exceptions

When executing C# code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C# will normally stop and generate an error message. The technical term for this is: C# will throw an **exception** (throw an error).

---

## C# try and catch

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

### Syntax

```
try

{
```

```
  //  Block of code to try
}
catch (Exception e)
{
  //  Block of code to handle errors
}
```

Consider the following example, where we create an array of three integers:

This will generate an error, because **myNumbers[10]** does not exist.

```
int[] myNumbers = {1, 2, 3};
Console.WriteLine(myNumbers[10]); // error!
```

The error message will be something like this:

```
System.IndexOutOfRangeException: 'Index was outside the bounds of
the array.'
```

If an error occurs, we can use `try...catch` to catch the error and execute some code to handle it.

In the following example, we use the variable inside the catch block (`e`) together with the built-in `Message` property, which outputs a message that describes the exception:

## Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
```

```
  Console.WriteLine(e.Message);
}
```

The output will be:

```
Index was outside the bounds of the array.
```

You can also output your own error message:

## Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
```

The output will be:

```
Something went wrong.
```

# Finally

The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

## Example

```
try
{
  int[] myNumbers = {1, 2, 3};
```

```
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
finally
{
  Console.WriteLine("The 'try catch' is finished.");
}
```

The output will be:

```
Something went wrong.
The 'try catch' is finished.
```

---

# The throw keyword

The `throw` statement allows you to create a custom error.

The `throw` statement is used together with an **exception class**. There are many exception classes available in C#: `ArithmeticException`, `FileNotFoundException`, `IndexOutOfRangeException`, `TimeOutException`, etc:

## Example

```
static void checkAge(int age)
{
  if (age < 18)
  {
```

```
    throw new ArithmeticException("Access denied - You must be at least 18 years old.");
  }
  else
  {
    Console.WriteLine("Access granted - You are old enough!");
  }
}

static void Main(string[] args)
{
  checkAge(15);
}
```

The error message displayed in the program will be:

```
System.ArithmeticException: 'Access denied - You must be at least 18 years old.'
```

If `age` was 20, you would **not** get an exception:

## Example

```
checkAge(20);
```

The output will be:

```
Access granted - You are old enough!
```