codecademy

Learn C#: Classes and Objects

C# Classes

In C#, *classes* are used to create custom types. The class defines the kinds of information and methods included in a custom type.

```
using System;

namespace BasicClasses
{
    class Forest {
    public string name;
    public int trees;
    }
}

// Here we have the Forest class which has two
pieces of data, called fields. They are the
"name" and "trees" fields.
```

C# Constructor

In C#, whenever an instance of a class is created, its *constructor* is called. Like methods, a constructor can be overloaded. It must have the same name as the enclosing class. This is useful when you may want to define an additional constructor that takes a different number of arguments.

```
// Takes two arguments
public Forest(int area, string country)
{
  this.Area = area;
  this.Country = country;
}

// Takes one argument
public Forest(int area)
{
  this.Area = area;
  this.Country = "Unknown";
}


// Typically, a constructor is used to set initial values and run any code needed to “set up” an instance.

// A constructor looks like a method, but its return type and method name are reduced to the name of the enclosing type.
```

C# Parameterless Constructor

In C#, if no constructors are specified in a class, the compiler automatically creates a parameterless constructor.

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

public static void Main (string[] args)
{
  Freshman f = new Freshman();
  // name is null
  string name = f.FirstName;
}

// In this example, no constructor is defined in Freshman, but a parameterless constructor is still available for use in Main().
```

C# Access Modifiers

In C#, members of a class can be marked with *access modifiers*, including `public` and `private`. A `public` member can be accessed by other classes. A `private` member can only be accessed by code in the same class.

By default, fields, properties, and methods are private, and classes are public.

```
public class Speech
{
  private string greeting = "Greetings";

  private string FormalGreeting()
  {
    return $"{greeting} and salutations";
  }

  public string Scream()
  {
    return FormalGreeting().ToUpper();
  }

}

public static void Main (string[] args)
{
  Speech s = new Speech();
  //string sfg = s.FormalGreeting(); // Error!
  //string sg = s.greeting; // Error!
  Console.WriteLine(s.Scream());
}

// In this example, greeting and
FormalGreeting() are private. They cannot be
called from the Main() method, which belongs
to a different class. However the code within
Scream() can access those members because
Scream() is part of the same class.
```

C# Field

In C#, a *field* stores a piece of data within an object. It acts like a variable and may have a different value for each instance of a type.

A field can have a number of modifiers, including: `public`, `private`, `static`, and `readonly`. If no access modifier is provided, a field is `private` by default.

```
public class Person
{
   private string firstName;
   private string lastName;
}

// In this example, firstName and lastName are private fields of the Person class.

// For effective encapsulation, a field is typically set to private, then accessed using a property. This ensures that values passed to an instance are validated (assuming the property implements some kind of validation for its field).
```

C# this Keyword

In C#, the `this` keyword refers to the current instance of a class.

```
// We can use the this keyword to refer to the current class's members hidden by similar names:
public NationalPark(int area, string state)
{
  this.area = area;
  this.state = state;
}

// The code below requires duplicate code, which can lead to extra work and errors when changes are needed:
public NationalPark(int area, string state)
{
  area = area;
  state = state;
}
public NationalPark(int area)
{
  area = area;
  state = "Unknown";
}

// Use this to have one constructor call another:
public NationalPark(int area) : this (state, "Unknown")
{ }
```

C# Members

In C#, a class contains *members*, which define the kind of data stored in a class and the behaviors a class can perform.

```
class Forest
{
  public string name;
  public string Name
  {
    get { return name; }
    set { name = value; }
  }
}

// A member of a class can be a field (like name), a property (like Name) or a method (like get()/set()). It can also be any of the following:
// Constants
// Constructors
// Events
// Finalizers
// Indexers
// Operators
// Nested Types
```

C# Property

In C#, a *property* is a member of an object that controls how one field may be accessed and/or modified. A property defines two methods: a `get()` method that describes how a field can be accessed, and a `set()` method that describes how a field can be modified. One use case for properties is to control access to a field. Another is to validate values for a field.

```
public class Freshman
{
  private string firstName;

  public string FirstName
  {
    get { return firstName; }
    set { firstName = value; }
  }
}

public static void Main (string[] args) {
  Freshman f = new Freshman();
  f.FirstName = "Louie";

  // Prints "Louie"
  Console.WriteLine(f.FirstName);
}

// In this example, FirstName is a property
```

C# Auto-Implemented Property

In C#, an *auto-implemented* property reads and writes to a private field, like other properties, but it does not require explicit definitions for the accessor methods nor the field. It is used with the `{ get; set; }` syntax. This helps your code become more concise.

```
public class HotSauce
{
  public string Title
  { get; set; }

  public string Origin
  { get; set; }
}

// In this example, Title and Origin are auto-
implemented properties. Notice that
a definition for each field (like private
string title) is no longer necessary.
A hidden, private field is created for each
property during runtime.
```

C# Dot Notation

In C#, a member of a class can be accessed with dot notation.

```
string greeting = "hello";

// Prints 5
Console.WriteLine(greeting.Length);

// Returns 8
Math.Min(8, 920);
```

C# Class Instance

In C#, an object is an *instance* of a class. An object can be created from a class using the `new` keyword.

```
Burger cheeseburger = new Burger();
// If a class is a recipe, then an object is
a single meal made from that recipe.

House tudor = new House();
// If a class is a blueprint, then an object
is a house made from that blueprint.
```

C# Static Constructor

In C#, a *static constructor* is run once per type, not per instance. It must be parameterless. It is invoked before the type is instantiated or a static member is accessed.

```
class Forest
{
  static Forest()
  {
    Console.WriteLine("Type Initialized");
  }
}
// In this class, either of the following two
lines would trigger the static constructor
(but it would not be triggered twice if these
two lines followed each other in succession):
Forest f  = new Forest();
Forest.Define();
```

C# Static Class

In C#, a *static* class cannot be instantiated. Its members are accessed by the class name.
This is useful when you want a class that provides a set of tools, but doesn't need to maintain any internal data.

`Math` is a commonly-used static class.

```
//Two examples of static classes calling
static methods:

Math.Min(23, 97);
Console.WriteLine("Let's Go!");
```