Cheatsheets / **Learn C#**

Learn C#: Encapsulation

Encapsulation

Encapsulation is the process of bundling related data and methods behind a barrier, such as a class definition, in order to hide the internal implementation details and control how data can be accessed and modified.

C# Access Modifiers

In C#, members of a class can be marked with an *access modifier*, such as `public` or `private`. A `public` member can be accessed by any class. A `private` member can only be accessed by code within the same class it is defined.
By default, fields, properties, and methods are private, and classes are public.

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

  // FormalGreeting() can access greeting since they are part of the same class
  private string FormalGreeting()
  {
    return $"{greeting} and salutations";
  }

  // Scream() can access FormalGreeting() since they are part of the same class
  public string Scream()
  {
    return FormalGreeting().ToUpper();
  }

}

public static void Main (string[] args)
{
  Speech s = new Speech();

  // greeting and FormalGreeting() are private, so they cannot be accessed outside the Speech class
  // string sg = s.greeting; // Error!
  // string sfg = s.FormalGreeting(); // Error!

  // Scream is public, so it can be accessed anywhere
  Console.WriteLine(s.Scream());
}
```

C# Auto-Implemented Property

In C#, an *auto-implemented property* automatically creates a private backing field and does not require explicit accessor methods. The body of an auto-implemented property will look like `{ get; set; }`, which helps us write more concise code for basic properties.

```
public class HotSauce
{
  // Title is a property with an explicitly defined backing private field, title. Title also has an explicit getter and setter for accessing and modifying the field
  private string title;
  public string Title
  {
    get
    {
      return title;
    }
    set
    {
      title = value;
    }
  }

  // Origin is an auto-implemented property. A hidden, private backing field is automatically created during runtime, which the property will use behind the scenes. Explicit getters and setters are not need. This Origin property is functionally equivalent to the preceding Title property
  public string Origin
  { get; set; }
}
```

C# Property

In C#, a *property* is a member of an object that controls how one field may be accessed and modified. A property defines two *accessor methods*:

- a `get()` method, which defines how its associated field can be accessed
- a `set()` method, which defines how the same field can be modified

Accessor methods default to `public` when no access modifier is specified.

```
public class Freshman
{
  // FIELD
  private string firstName;

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

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

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

C# Static Constructor

In C#, a *static constructor* is run once per type, not per instance. It cannot take any parameters or be defined with an access modifier. A static constructor cannot be invoked directly — it is invoked automatically exactly once, before one of the following happens for the first time:

- an instance of the type is instantiated
- a static member of the type is accessed

```
class Forest
{
  static Forest()
  {
    Console.WriteLine("Type
Initialized");
  }

  public static void Define()
  {
    Console.WriteLine("A forest is a
large area where trees are the primary
life-form.")
  }
}

// For this class, either of the
following two lines would trigger the
static constructor, but only the first
time one of them occurs:
Forest f  = new Forest();
Forest.Define();
```

C# Static Class

In C#, a *static class* can only contain static members and cannot be instantiated. All of 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, built-in static class.

```
// Math and Console are both static
classes that are built-in to the C#
languages
// the methods Min() and WriteLine() are
accessed using the corresponding class
name and dot notation, like
ClassName.MethodName()

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

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.

C# Constructor

In C#, whenever an instance of a class is created, its *constructor* is called. It must have the same name as the enclosing class. Like other methods, a constructor can be overloaded. 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
it does not include a return type and it
must have the same name as 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;
}

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(). All fields will be set to a default value according to their type and remain unchanged until updated manually.
```

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.

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

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

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";
}

// When "this" is used as a method
// it instructs one constructor to call another
// the following achieves the same as the preceding 2 constructors
public NationalPark(int area) : this(are, "Unknown")
{ }
```

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);
```

Print Share