More On Classes

Destructors

As constructors are used when a class is instantiated, **destructors** are automatically invoked when an object is destroyed or deleted.
Destructors have the following attributes:
- A class can only have **one** destructor.
- Destructors cannot be called. They are invoked automatically.
- A destructor does not take modifiers or have parameters.
- The name of a destructor is exactly the same as the class prefixed with a **tilde (~)**.

For Example:

```
class Dog
{
  ~Dog()
  {
    // code statements
  }
}
```

Destructors can be very useful for releasing resources before coming out of the program. This can include closing files, releasing memory, and so on.

Destructors

Let's include WriteLine statements in the destructor and constructor of our class and see how the program behaves when an object of that class is created and when the program ends:

```
class Dog
{
  public Dog() {
    Console.WriteLine("Constructor");
  }
  ~Dog() {
    Console.WriteLine("Destructor");
  }
}
static void Main(string[] args) {
  Dog d = new Dog();
}
/*Outputs:
Constructor
Destructor
/*
```

Try It Yourself

When the program runs, it first creates the object, which calls the constructor. The object is deleted at the end of the program and the destructor is invoked when the program's execution is complete.

This can be useful, for example, if your class is working with storage or files. The constructor would initialize and open the files. Then, when the program ends, the destructor would close the files.

Static

Now it's time to discuss the static keyword.
You first noticed it in the Main method's declaration:

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

Class members (variables, properties, methods) can also be declared as static. This makes those members belong to the class itself, instead of belonging to individual objects. No matter how many objects of the class are created, there is only **one** copy of the static member.
For example:

```
class Cat {
  public static int count=0;
  public Cat() {
    count++;
  }
}
```

In this case, we declared a public member variable **count**, which is static. The constructor of the class increments the **count** variable by one.

> No matter how many **Cat** objects are instantiated, there is always only one **count** variable that belongs to the **Cat** class because it was declared **static**.

Static

Because of their global nature, static members can be accessed directly using the **class name** without an object.
For example:

```
class Cat {
  public static int count=0;
  public Cat() {
    count++;
  }
}
static void Main(string[] args)
{
  Cat c1 = new Cat();
  Cat c2 = new Cat();

  Console.WriteLine(Cat.count);
}
//Outputs 2
```

Try It Yourself

As you can see, we can access the static variable using the class name: **Cat.count**.
The **count** variable is shared between all Cat objects. For this class, each time an object is created, the static value is incremented. The program above demonstrates this when 2 is displayed after creating two objects of that class.

> You must access **static** members using the class name. If you try to access them via an object of that class, you will generate an error.

Static Methods

The same concept applies to static methods.
For example:

```
class Dog
{
  public static void Bark() {
    Console.WriteLine("Woof");
  }
}
static void Main(string[] args)
{
  Dog.Bark();
}
// Outputs "Woof"
```

Try It Yourself

Static methods can access **only** static members.

> The Main method is **static**, as it is the starting point of any program. Therefore any method called directly from Main had to be **static**.

Static

Constant members are static by definition.
For example:

```
class MathClass {
  public const int ONE = 1;
}
static void Main(string[] args) {
  Console.Write(MathClass.ONE);
}
//Outputs 1
```

Try It Yourself

As you can see, we access the property **ONE** using the name of the class, just like a static member. This is because all **const** members are **static** by default.

Static Constructors

Constructors can be declared **static** to initialize static members of the class.
The static constructor is automatically called once when we access a static member of the class.
For example:

```
class SomeClass {
  public static int X { get; set; }
  public static int Y { get; set; }

  static SomeClass() {
    X = 10;
    Y = 20;
  }
}
```

Try It Yourself

The constructor will get called once when we try to access SomeClass.X or SomeClass.Y.

Static Classes

An entire class can be declared as static.
A static **class** can contain only static members.
You cannot instantiate an object of a static class, as only one instance of the static class can exist in a program.
Static classes are useful for combining logical properties and methods. A good example of this is the **Math** class.
It contains various useful properties and methods for mathematical operations.
For example, the **Pow** method raises a number to a power:

```
Console.WriteLine(Math.Pow(2, 3));
//Outputs 8
```

Try It Yourself

You access all members of the Math class using the class name, without declaring an object.

Tap next to learn about the available methods of the Math class.

Static Classes

There are a number of useful static methods and properties available in C#:

Math

Math.**PI** the constant PI.
Math.**E** represents the natural logarithmic base e.
Math.**Max**() returns the larger of its two arguments.
Math.**Min**() returns the smaller of its two arguments.
Math.**Abs**() returns the absolute value of its argument.
Math.**Sin**() returns the sine of the specified angle.
Math.**Cos**() returns the cosine of the specified angle.
Math.**Pow**() returns a specified number raised to the specified power.
Math.**Round**() rounds the decimal number to its nearest integral value.
Math.**Sqrt**() returns the square root of a specified number.

Array
The **Array** class includes some static methods for manipulating arrays:

```
int[] arr = {1, 2, 3, 4};

Array.Reverse(arr);
//arr = {4, 3, 2, 1}

Array.Sort(arr);
//arr = {1, 2, 3, 4}
```

String

```
string s1 = "some text";
string s2 = "another text";

String.Concat(s1, s2); // combines the two strings

String.Equals(s1, s2); // returns false
```

DateTime
The **DateTime** structure allows you to work with dates.

```
DateTime.Now; // represents the current date & time
DateTime.Today; // represents the current day

DateTime.DaysInMonth(2016, 2);
//return the number of days in the specified month
```

Try It Yourself

The **Console** class is also an example of a static class. We use its static **WriteLine()** method to output to the screen, or the static **ReadLine()** method to get user input.
The **Convert** class used to convert value types is also a static class.

The this Keyword

The **this** keyword is used inside the class and refers to the current instance of the class, meaning it refers to the current object.
One of the common uses of **this** is to distinguish class members from other data, such as local or formal parameters of a method, as shown in the following example:

```
class Person {
  private string name;
  public Person(string name) {
    this.name = name;
  }
}
```

Here, **this.name** represents the member of the class, whereas **name** represents the parameter of the constructor.

Another common use of **this** is for passing the current instance to a method as parameter: ShowPersonInfo(**this**);

The readonly Modifier

The **readonly** modifier prevents a member of a class from being modified after construction. It means that the field declared as **readonly** can be modified only when you declare it or from within a constructor.
For example:

```
class Person {
  private readonly string name = "John";
  public Person(string name) {
    this.name = name;
  }
}
```

If we try to modify the **name** field anywhere else, we will get an error.
There are three major differences between **readonly** and **const** fields.
First, a constant field must be initialized when it is declared, whereas a readonly field can be declared without initialization, as in:

```
readonly string name; // OK
const double PI; // Error
```

Second, a **readonly** field value can be changed in a constructor, but a constant value cannot.
Third, the **readonly** field can be assigned a value that is a result of a calculation, but constants cannot, as in:

```
readonly double a = Math.Sin(60); // OK
const double b = Math.Sin(60); // Error!
```

> The readonly modifier prevents a member of a class from being modified after construction.

Indexers

An indexer allows objects to be indexed like an array.
As discussed earlier, a string variable is actually an object of the **String** class. Further, the String class is actually an array of Char objects. In this way, the string class implements an indexer so we can access any character (Char object) by its index:

```
string str = "Hello World";
char x = str[4];
Console.WriteLine(x);
//Outputs "o"
```

Try It Yourself

> Arrays use integer indexes, but indexers can use any type of index, such as strings, characters, etc.

Indexers

Declaration of an indexer is to some extent similar to a property. The difference is that indexer accessors require an **index**.
Like a property, you use **get** and **set** accessors for defining an indexer. However, where properties return or set a specific data member, indexers return or set a particular value from the object instance.
Indexers are defined with the **this** keyword.
For example:

```
class Clients {
  private string[] names = new string[10];

  public string this[int index] {
    get {
      return names[index];
    }
    set {
      names[index] = value;
    }
  }
}
```

As you can see, the indexer definition includes the **this** keyword and an index, which is used to get and set the appropriate value.
Now, when we declare an object of class Clients, we use an index to refer to specific objects like the elements of an array:

```
Clients c = new Clients();
c[0] = "Dave";
c[1] = "Bob";

Console.WriteLine(c[1]);
//Outputs "Bob"
```

Try It Yourself

You typically use an indexer if the class represents a list, collection, or array of objects.

Operator Overloading

Most operators in C# can be **overloaded**, meaning they can be redefined for custom actions.
For example, you can redefine the action of the plus (+) operator in a custom class.
Consider the **Box** class that has **Height** and **Width** properties:

```
class Box {
  public int Height {get; set;}
  public int Width {get; set;}
  public Box(int h, int w) {
    Height = h;
    Width = w;
  }
}
static void Main(string[] args) {
  Box b1 = new Box(14, 3);
  Box b2 = new Box(5, 7);
}
```

We would like to add these two Box objects, which would result in a new, bigger Box.
So, basically, we would like the following code to work:

```
Box b3 = b1 + b2;
```

The Height and Width properties of object b3 should be equal to the sum of the corresponding properties of the b1 and b2 objects.

This is achieved through **operator overloading**. Tap next to learn more!

Operator Overloading

Overloaded operators are methods with special names, where the keyword **operator** is followed by the symbol for the operator being defined.
Similar to any other method, an overloaded operator has a return type and a parameter list.
For example, for our **Box** class, we overload the + operator:

```
public static Box operator+ (Box a, Box b) {
  int h = a.Height + b.Height;
  int w = a.Width + b.Width;
  Box res = new Box(h, w);
  return res;
}
```

The method above defines an overloaded **operator +** with two Box object parameters and returning a new Box object whose Height and Width properties equal the sum of its parameter's corresponding properties.
Additionally, the overloaded operator must be **static**.
Putting it all together:

```
class Box {
  public int Height { get; set; }
  public int Width { get; set; }
  public Box(int h, int w) {
    Height = h;
    Width = w;
  }
  public static Box operator+(Box a, Box b) {
    int h = a.Height + b.Height;
    int w = a.Width + b.Width;
    Box res = new Box(h, w);
    return res;
  }
}
static void Main(string[] args) {
  Box b1 = new Box(14, 3);
  Box b2 = new Box(5, 7);
  Box b3 = b1 + b2;

  Console.WriteLine(b3.Height); //19
  Console.WriteLine(b3.Width); //10
}
```

Try It Yourself

All arithmetic and comparison operators can be overloaded. For instance, you could define greater than and less than operators for the boxes that would compare the Boxes and return a **boolean** result. Just keep in mind that when overloading the greater than operator, the less than operator should also be defined.

End.