



WriteLine() is a method, defined in the class Console, that writes text to standard output followed by a new line. The class Console is contained in the namespace (grouping of classes) System. If you want to avoid having to fully qualify Console by writing System.Console you can put using System; at the beginning of the file. Now we can write Console.WriteLine("Hello World!");

## Example 2

Our next example will demonstrate how to create and use user-defined classes as well as how to create dynamic link libraries. In your text editor create two new files. First, Apple.cs contains:

```
public class Apple {
    private string variety = "";

    public Apple(string appleVariety) {
        this.variety = appleVariety;
    }

    public void outputVariety() {
        System.Console.WriteLine(variety);
    }
}
```

Example2.cs contains:

```
class Example2 {
    static void Main() {
        Apple mac = new Apple("Macintosh");
        Apple gra = new Apple("Granny Smith");
        Apple cor = new Apple("Cortland");
        mac.outputVariety();
        gra.outputVariety();
        cor.outputVariety();
    }
}
```

First you'll notice that we've created a new user-defined class called Apple. While it was not necessary to put the class apple in its own file, it's a common object oriented programming practice to put each class in its own file for organization sake. We've added the modifier public to the declaration of the class (public class Apple) Apple so that other classes can create instances of it. The next line of code defines an instance variable called variety. The modifier private is used to make variety directly accessible only from within Apple. This is a common object-oriented programming practice called encapsulation where the details of how objects worked are hidden to the user of that object.

A helpful analogy demonstrating encapsulation in the real world is the keyboard you are using right now. You don't completely understand how keystrokes are sent to the controller (most of us don't anyway) you just understand how the interface works. We hit the '&' key in a text editor and an '&' pops up on the screen. If everyone had to understand the details of how a keyboard worked instead of just the interface, not many of us would use one.

The next three lines:

```
public Apple(string appleVariety) {
    this.variety = variety;
}
```

define the constructor for the class Apple. The constructor of a class is like the blue print describing how to create new instances of that class. We can distinguish a constructor from other methods in a class because the Constructor has the same name as the class. In our case the constructor for the class Apple takes one parameter, a string, that provides the name of the variety of the Apple we are creating. This value is stored in the instance variable variety. The last method of the class apple is an accessor method called outputVariety(). It's is called an accessor method because it provides an interface for accessing an instance variable.

Now let's look at the class Example2. The difference between our first example is that we are creating instances of and using our user-defined class Apple. We create three Apples using the new method (you never explicitly call the constructor of a class to create new objects - the new method takes care of calling the constructor for us). After creating three apples, we call each apples outputVariety method which outputs the name of the variety to the console.

So let's compile and run this example. First we must compile the Apple class into a dynamic link library. We do this with the following command:

```
>csc /target:library Apple.cs
```

`/target:library` means don't create an executable, instead create a .dll (dynamic link library) as the output. This should create a file called Apple.dll

Next, we compile Example2.cs and link it with Apple.dll with the following command:

```
>csc /reference:Apple.dll Example2.cs
```

Now we should have an executable file called Example2.exe that will output:

```
Macintosh  
Granny Smith  
Cortland
```

to the console.

### Example 3

In our final example we will look at topics of abstraction and polymorphism in C#. First let's define these two new terms. Abstraction works by abstracting common parts of objects and merging them into a single abstract class. In our example we will create an abstract class called shape. Each shape will have a method to return its color. Whether the shape is a square or a circle, returning the color works the same so this method that returns the color is abstracted from each shape class and put into a parent class. So if we had ten different shapes that had method to return the color now we only have one in the parent class. You can see how this shortens and simplifies code.

Polymorphism (literally meaning many forms), in terms of object-oriented programming, is the ability of an object or method to react differently depending upon its class. In our next example, the abstract class shape will have a method called `getArea()`. This method takes many forms depending on the type of shape (circle square or rectangle) it is working.

Here's the code:

```
public abstract class Shape {
    protected string color;

    public Shape(string color) {
        this.color = color;
    }

    public string getColor() {
        return color;
    }

    public abstract double getArea();
}

public class Circle : Shape {
    private double radius;

    public Circle(string color, double radius) : base(color) {
        this.radius = radius;
    }

    public override double getArea() {
        return System.Math.PI * radius * radius;
    }
}

public class Square : Shape {
    private double sideLen;

    public Square(string color, double sideLen) : base(color) {
        this.sideLen = sideLen;
    }
}
```

```

    public override double getArea() {
        return sideLen * sideLen;
    }
}

public class Example3
{
    static void Main()
    {
        Shape myCircle = new Circle("orange", 3);
        Shape myRectangle = new Rectangle("red", 8, 4);
        Shape mySquare = new Square("green", 4);
        System.Console.WriteLine("This circle is " + myCircle.getColor()
            + " and its area is " + myCircle.getArea() + ".");
        System.Console.WriteLine("This rectangle is " + myRectangle.getColor()
            + " and its area is " + myRectangle.getArea() + ".");
        System.Console.WriteLine("This square is " + mySquare.getColor()
            + " and its area is " + mySquare.getArea() + ".");
    }
}

```

The first class we define is Shape. It is defined as abstract because we don't want to create instances of this class only its derived classes. We are taking the common features of all shapes and putting those features in this class. Shape has one instance variable: area which is defined as protected. The protected modifier means that this element is not accessible to other classes except those that derive from this class (classes who share the common features of Shape). Next is the constructor and the accessor method getColor(). There's nothing new in either of these two methods, but the last method, getArea(), is defined as abstract. This is because each individual shape calculates its area differently so each shape must define this method.

The classes Circle, Rectangle and Square are all derived (sub) classes of Shape. They all share common features of shape. You see this by their definition. They all have:

```
public class <class name> : Shape {
```

The ": Shape" means I am deriving from the class Shape. Because these classes derive from Shape they automatically contain all the instance variables defined as public or protected. So color is an instance variables of Circle, Rectangle and Square. Each shape has its own constructor that sets the instance variable(s) of that shape (parts specific to each shape) then each shape calls for help from the base constructor (Shape's constructor) to set the common features (color). This is done with:

```
public Circle(string color, double radius) : base(color) {
```

The ": base(color)" means call the constructor of the base class with the parameter color. Finally we have the method getArea(), which demonstrates polymorphism. All shapes have the method getArea(), but depending on whether you have a circle, rectangle or square, a different method is called.

To run this example, put each file in the same directory and first run the command:

```
>csc /target:library /out:Shapes.dll Shapes.cs Circle.cs Rectangle.cs Square.cs
```

then run with the following command:

```
>csc /reference:Shapes.dll Example3.cs
```

Now when you run the newly generated file Example3.exe you should see the output:

```

This circle is orange and its area is 28.274333882308138.
This square is green and its area is 16.
This rectangle is red and its area is 0.

```

## Conclusion

You should be able to:

- create and compile a simple C# program

- understand the basics of an object oriented program and how it is implemented in C# (this includes creating user-defined classes, abstract and subclasses)
- send output to the console

For further information head to:

- [C# Language Reference](#)
- [C# Station](#)

Developed Under:

Microsoft Visual Studio .NET Beta 1

---

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved

Questions or Comments? [devcentral@itcentral.com](mailto:devcentral@itcentral.com)

**[PRIVACY POLICY](#)**