**OOPDraw  
Learn the principles of OOP by writing a simple drawing program**

STUDENT WORKBOOK

Created by Richard Pawson

[Creative Commons License](https://creativecommons.org/licenses/by-sa/4.0/)  
This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

# Exercise 1: Using Turtle Graphics

In this first exercise we are going to write a program to create a simple drawing made up of squares and circles, using the *procedural programming* paradigm that you have mainly used so far.

You might well have encountered ‘Turtle Graphics’ earlier in your education - instructions are given to an imaginary Turtle to move forward/backward a specified distance, and to rotate (turn) a number of degrees (positive for clockwise, negative for anti-clockwise).

Start by downloading, unzipping, and then opening the OOPDraw project. Build the solution (**Build > Build Solution**) - this is necessary to get Visual Studio to download and install any packages (specifically *Nakov.TurtleGraphics* that it depends upon). After this the code should compile.

Find and edit the empty MyDrawings.Draw function, and then add this code into it:

namespace OOPDraw

{

public class MyDrawing

{

public static void Draw()

{

Turtle.PenColor = Color.Blue;

for (int i = 0; i < 4; i++)

{

Turtle.Forward(100);

Turtle.Rotate(90);

}

}

}

}

Run the program and copy a partial screenshot showing the resulting drawing here. Make sure you understand why this generates a square.

Experiment with different values in the Forward(100) function call.

Next we will extract the code for drawing a square into a separate function, passing the sideLength and Colour as paramaters, and also a positionX and position to specify where the square should be started (0,0 being the centre of the screen). Then we will call this function more than once to draw multiple squares.

public static void Draw()

{

DrawSquare(0,0,Color.Blue, 100);

DrawSquare(100, 0, Color.Red, 50);

}

private static void DrawSquare(float positionX, float positionY,   
 Color lineColor, float sideLength)

{

Turtle.X = positionX;

Turtle.Y = positionY;

Turtle.PenColor = lineColor;

for (int i = 0; i < 4; i++)

{

Turtle.Forward(sideLength);

Turtle.Rotate(90);

}

}

Make the changes highlighted above, run the program, and paste in a partial screenshot showing the resulting drawing.

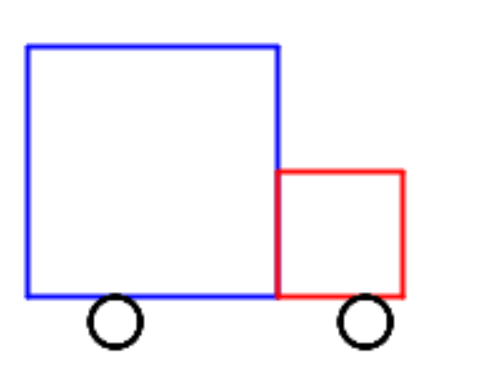
Now create a new function called DrawCircle, which takes four parameters. The first three are the same as for Square, but the last one should be a float named radius.

In Turtle graphics, the simplest way to draw a circle is to draw a polygon with 360 sides - rotating one degree each time. The length of each side can be calculated as:

(float) (2 \* Math.PI \* radius /360)

(The is needed to convert the result of the calculation from a double to a float, which is the type taken by the Forward function).

Within the Draw function, create two small circles positions such that your drawing resembles a truck:



Paste in the code for your Circle function and the Draw function that calls it.

# Exercise 2: Introducing objects as custom data types

It would be possible continue with the *procedural* approach adopted in Exercise 1 to create a sophisticated drawing, and indeed this is somewhat like the approach used by *Paint* programs - except that the functions to draw lines and shapes, to fill them with colour and so forth are triggered by the user’s mouse clicks and movements.

The limitation of this approach is that once a square has been drawn, that square exists only as a set of pixels on the screen. If we wanted to change the drawing (because, say, we weren’t happy with the size of the wheels) we would have to erase the necessary pixels and draw something afresh, copying all the parameters we need for the shape, and modifying the one(s) we want to change.

At minimum it would be nice to hold all the parameters that define one square in one holder and give it a meaningful name such as Cab, or Wheel, say. We can’t hold those parameters in an array because they are of different types: float, and color, so far.

An *object* can be thought of, in the first instance, as custom data type with multiple *properties* holding individual data items, each with a name and type. (Shortly, we will see that this is a very limited view of what an object is, but it will do for the moment). We define one of these custom types by creating a *class.* You can define a new class within the same code file as your drawing functions, but it is considered better practice to create each new class in a file of its own. Do this by right-clicking on the project icon and selecting Add > Class, giving the new class the name Square. Then edit the new file to look like this:

using System.Drawing;

namespace OOPDraw

{

public class Square

{

//Properties

public float PositionX { get; set; }

public float PositionY { get; set; }

public Color LineColor { get; set; }

public float SideLength { get; set; }

//The 'Constructor

public Square(float x, float y, Color lineColor, float sideLength)

{

LineColor = lineColor;

PositionX = x;

PositionY = y;

SideLength = sideLength;

}

}

}

When writing the four properties you can make use of the ‘prop’ *code snippet*: just type prop and then Tab twice, then edit the highlighted fields for type and name.

It is a convention to start each property name with a capital letter.

A  *class* can be thought of as a *template* that defines a type (Square) and from which you can create multiple objects (also known as *instances)* each having its own copy of the properties containing its own individual values.

The *constructor* is the function that is used to create a new instance of that type (in this case to create a new Square). Its parameters specify the values that you must provide to create an instance, and in the body of the constructor these parameters are used to set up the individual properties. (Many people comment that it looks a bit wasteful copying each parameter into a similarly-named property, but you’ll soon get used to it. Some programming languages have side-stepped this apparent repetition.)

You can use the ‘ctor’ code snippet to help write the constructor, though you will need to add all the parameters to it yourself.

Now we can change the code in our Draw function to create two instances of type Square as follows:

public static void Draw()

{

var body = new Square(0,0,Color.Blue, 100);

var cab = new Square(100, 0, Color.Red, 50);

Circle(125, -10, Color.Black, 10);

Circle(25, -10, Color.Black, 10);

}

Note the use of new to create a new instance. If you right click on the word Square and select **Go To Definition** you will be taken to the *constructor* on the Square class.

However, if you run the program now, the body and cab are no longer drawn. We’ve created holders for the properties of each (instances of Square) but we have not told the program to draw them. We need to call the DrawSquare function, but instead of passing it the individual parameters as before, we now just want to pass it an object of type Square, from which it can read the properties it needs. Make these changes:

public static void Draw()

{

var body = new Square(0,0,Color.Blue, 100);

DrawSquare(body);

var cab = new Square(100, 0, Color.Red, 50);

DrawSquare(cab);

DrawCircle(125, -10, Color.Black, 10);

DrawCircle(25, -10, Color.Black, 10);

}

private static void DrawSquare(Square sq)

{

Turtle.X = sq.PositionX;

Turtle.Y = sq.PositionY;

Turtle.PenColor = sq.LineColor;

for (int i = 0; i < 4; i++)

{

Turtle.Forward(sq.SideLength);

Turtle.Rotate(90);

}

}

Code of the form sq.PositionX may be read as ‘the PositionX property of the object sq’ and this is known as the *dot syntax.* As well as being easy to read, it has another advantage: if you type just ‘sq.’ Visual Studio will give you a pop up list showing all the properties you can access on that typefrom which you can select - or carry on typing (e.g. ‘sq.Pos’) to see matching properties. This *auto-complete* feature saves time and reduces errors and is one of the big advantages of statically-typed object-orientation languages such as Java, C# and VB - you won’t get this in a dynamically-typed language such as Python or JavaScript because the system is not able to determine the type of a variable or parameter until the program is run - so it can’t work out which properties are available.

Check that the program now draws the whole truck again.

Now add a new class named Circle, using appropriate properties for that type. Modify the DrawCircle function in a manner equivalent to DrawSquare. Then in the Draw method create two instances of Circle for the two wheels, and draw them. Paste in your completed code for the Circle class, the modified DrawCircle function, and the modified Draw function that uses both.

# Exercise 3: Adding behaviour to the object

So far we have used objects only for the purpose of holding multiple pieces data, all related but potentially of different types, in a single *instance* of a custom data type. Properly, though, object instances don’t just have properties (also known as *data* and collectively representing the object’s *state*) they have *methods* (also known as *functions* and collectively representing the object’s *behaviour*). In fact, in real OOP the *behaviour* of objects is considered the more important of the two things.

Another way of putting this is that objects have two types of *responsibility*:

* What they *know* (or their ‘know-whats’), represented by their properties.
* What they *know how to do* (or their ‘know-how-tos’), represented by their methods.

One of the goals of object-oriented design is to push as much of the behaviour of the program as a whole onto the objects that the program uses.

We’ll start this process by transferring the responsibility for drawing individual objects:

Start by creating a new function Draw() inside the Square class (properly known as a *method* when it is in a class) as follows:

using Nakov.TurtleGraphics;

using System.Drawing;

namespace OOPDraw

{

public class Square

{

//Properties

public float PositionX { get; set; }

public float PositionY { get; set; }

public Color LineColor { get; set; }

public float SideLength { get; set; }

//The 'Constructor

public Square(float x, float y, Color lineColor, float sideLength)

{

LineColor = lineColor;

PositionX = x;

PositionY = y;

SideLength = sideLength;

}

public void Draw()

{

Turtle.X = PositionX;

Turtle.Y = PositionY;

Turtle.PenColor = LineColor;

for (int i = 0; i < 4; i++)

{

Turtle.Forward(SideLength);

Turtle.Rotate(90);

}

}

}

}

Note that this is somewhat similar to the DrawSquare method in MyDrawing, but note these differences:

* Draw is public - which means that it can be accessed by code outside the class
* It does not need to take a Square parameter, because it has direct access to the properties defined in the class.
* It does not have the keyword static. This is because we want Draw to operate on a specific instance of the class, not on the class as a whole. If you try *temporarily* adding the static keyword to Draw, you will find that the properties (which always belong to a specific instance) can no longer be accessed, and you get compile errors.

Now modify the code in Draw function on MyDrawing, so that you call the Square’s new Draw function, as follows:

public class MyDrawing

{

public static void Draw()

{

var body = new Square(0,0,Color.Blue, 100);

body.Draw();

var cab = new Square(100, 0, Color.Red, 50);

cab.Draw();

var frontWheel = new Circle(125, -10, Color.Black, 10);

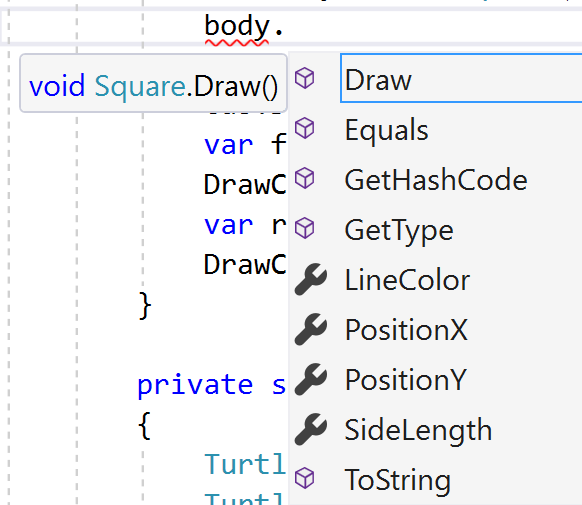
DrawCircle(frontWheel);

var rearWheel = new Circle(25, -10, Color.Black, 10);

DrawCircle(rearWheel);

}

Notice that we are invoking the new method using the *dot-syntax* again, only this time, the brackets after the name Draw indicate that it is a method (function) not a property. Also, in the auto-complete, you will see that properties have a small spanner icon next to them and methods don’t:



(We’ll find out where the other methods – such as Equals and ToString - come from later on).

You should now delete the original DrawSquare() function from MyDrawing, as it is no longer used.

Check that the program still runs

Now do the same for Circle:

* Add a Draw method to Circle, using the same structure as for Square but a different implementation
* Call the Draw instance method on the two circles representing the wheels, instead of the old DrawCircle function
* Delete the DrawCircle function.

Paste in your code for the new Draw function on Circle and the modified MyDrawing code.

# Exercise 4:

But what advantage have we gained from moving the Draw functionality from freestanding (static) functions into instance methods on each class.

* Code management
* Syntax
* Polymorphism (because they have the same name).

### Functionality is same; only immediate advantage is the dot-syntax, with auto-complete but later we will see more advantages to implementing behaviour as instance methods (Polymorphism)

### Create an array of squares and iterate over them calling Draw

### Add further methods for Erase, Move, Grow, and Rotate

## Add Circle object

Implement Draw, Move and Grow

Add two circles into drawing, but note that we can’t add them into our array because they are of a different type, have to treat them differently

## Introducing abstract types

Define abstract Shape and make Circle and Square inherit from it

Explain abstract - you can’t say new Shape

Now we can change type of array to Shape and add both Circles and Squares

But it still won’t let us call Draw or Move on our array members , because it doesn’t know that Shape has those methods

Define abstract methods for Draw, Move, and Grow, and change implementations to override them

Now we can call those methods on all shapes, *without knowing which specific type (Square or Circle) we are dealing with. This is called Polymorphism*

## *Introducing inherited functionality*

*Notice that the two implementations of Move are the same*

*Move the implementation up into Shape, but also need to move up the properties that this needs.*

*Now we are not just implementing the same methods, we are* inheriting  *some of them*

## *Introducing interfaces to provide some kinds of commonality*

*Exercise: Add Equilateral Triangle*

*Add Orientation property and Rotate method, and make this work in Draw*

*Do the same for Square; it makes no sense for Circle*

*How can we rotate all rotatable?*

## *Introducing association*

*Define Truck with Cab, Body, Wheels*

# *THINK ABOUT: ABSOLUTE vs. RELATIVE positions*

# *THINK ABOUT: Aggregation vs. Composition*

# THINK ABOUT: Adding Information Hiding