**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.Execute function, and then add this code into it:

namespace OOPDraw

{

public class MyDrawing

{

public static void Execute()

{

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 Execute()

{

DrawSquare(50,50,Color.Blue, 100);

DrawSquare(125, 25, Color.Red, 50);

}

private static void DrawSquare(float centreX, float centreY,   
 Color lineColor, float sideLength)

{

Turtle.Angle = 0;

Turtle.X = centreX-sideLength/2; //To ensure shape is centred correctly

Turtle.Y = centreY-sideLength/2;

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.

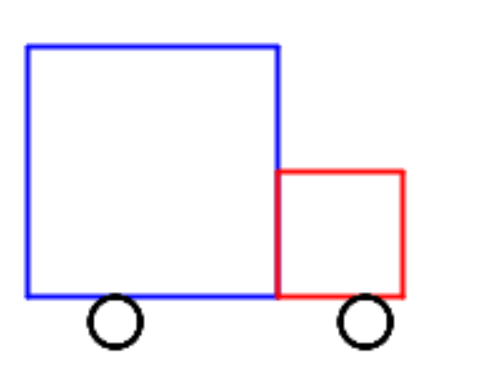
Now create a new function called DrawCircle, which takes four parameters. The first three are the same as for Square, but the last one should be a float named radius.

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 Execute function, create two small circles positions such that your drawing resembles a truck:



Paste in the code for your Circle function and the Execute 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 CentreX { get; set; }

public float CentreY { get; set; }

public Color LineColor { get; set; }

public float SideLength { get; set; }

//The 'Constructor

public Square(float centreX, float centreY, C  
 olor lineColor, float sideLength)

{

CentreX = centreX;

CentreY = y;

LineColor = lineColor;

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 Execute function to create two instances of type Square as follows:

public static void Execute()

{

var body = new Square(50, 50, Color.Blue, 100);

var cab = new Square(125, 25, Color.Red, 50);

}

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 Execute()

{

var body = new Square(50, 50, Color.Blue, 100);

DrawSquare(body);

var cab = new Square(125, 25, Color.Red, 50);

DrawSquare(cab);

}

private static void DrawSquare(Square sq)

{

Turtle.Angle = 0;

Turtle.X = sq.CentreX-SideLength/2; //To ensure shape is centred correctly

Turtle.Y = sq.CentreY-SideLength/2;

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 Execute 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 Execute 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.

The idea that objects have both types of responsibility is known as *encapsulation* - an object encapsulates all the responsibilities associated with the thing that it represents. Thus a Square object should know everything that we need to known about a square (for the purpose of the application we are developing, that is) and it should know how to do everything that we might want to do to a square.

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 CentreX { get; set; }

public float CentreY { get; set; }

public Color LineColor { get; set; }

public float SideLength { get; set; }

//The 'Constructor

public Square(float centreX, float centreY, Color lineColor, float sideLength)

{

LineColor = lineColor;

CentreX = centreX;

CentreY = centreY;

SideLength = sideLength;

}

public void Draw()

{

Turtle.Angle = 0;

Turtle.X = CentreX-SideLength/2; //To ensure shape is centred

Turtle.Y = CentreY-SideLength/2;

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 Execute function on MyDrawing, so that you call the Square’s new Draw function, as follows:

public class MyDrawing

{

public static void Execute()

{

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);

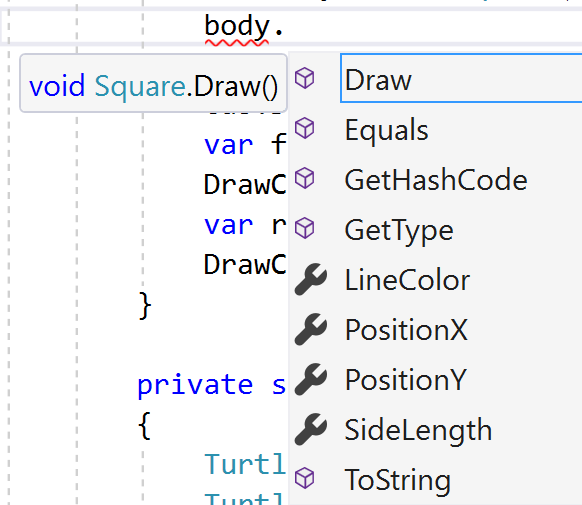
DrawCircle(frontWheel);

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 method on Circle and the modified MyDrawing code.

# Exercise 4: Introducing Polymorphism

But what advantage have we gained from moving the Draw functionality from freestanding (static) functions into instance methods on each class? On can argue that the *dot-syntax* is slightly easier to read; or that it makes the management of code (in a larger system) easier, because all the code relating to using or manipulating squares is located in the Square class; but these are small benefits.

The big benefit comes from *Polymorphism,* another of the key principles of OOP, and, arguably, the most important. Here is a definition for polymorphism:

“Where two of more objects have a property or method with the same signature (name, return type, and, for a method, parameters), even though the implementations are different, then it is possible to access that property or method on the object, *without having two know the specific type of that object.*”

In the context of our drawing program, Square and Circle both have a method called Draw, which has the same signature, though the implementation (how it is drawn) are quite different. They also have several properties (CentreX, CentreY, LineColor) in common.

Therefore, if we had a collection (which could be a List or an Array) of shapes, made up of some Squares, Circles and other forms of shape, then, with polymorphism, we could just call the Draw method on each shape without having to know that the first shape is a Square, the second a Circle and so on. Later on we could add further methods such as Move, or Grow, and again call those on each member.

Think of how this works in PowerPoint. You can create a variety of shapes from the shapes menu. Then you can select multiple shapes, of different types, and then move them, grow them, colour them in one go, even though each form of shape has to do different things in response to those instructions.

## Implementation

First, how can we create a collection of disparate shapes? We can create a List of type object and add any type of object into it as follows:

var shapes = new List<object>();

var body = new Square(50, 50, Color.Blue, 100);

shapes.Add(body);

var cab = new Square(125, 25, Color.Red, 50);

shapes.Add(cab);

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

shapes.Add(frontWheel);

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

shapes.Add(rearWheel);

Then we would like to call the Draw method on each shape as follows:

foreach (var shape in shapes)

{

shape.Draw();

}

But if you add that code into your Execute method you will find that it does not compile. Mouse over the error (where the red squiggle-underline appears) and note the error.

What compile error message do you get?

The C# compiler is saying that there is no method named Draw on every object: after all you could be adding strings and integers into the list of shapes – try it!

(If we were using a dynamically typed language such as Python or JavaScript, we could get away with this. The system wouldn’t check whether each object *actually* has a Draw method until you get to it - if it encounters an object that does not have a Draw method you get an error. This can sometimes be more convenient, but the advantage of static typing is that you get more checking up front).

The solution is to define a new class called Shape, which does have a Draw method, and then specify that both Square and Circle (and any new shape type we define) are in fact Shapes. So here’ the new class:

namespace OOPDraw

{

public abstract class Shape

{

public abstract void Draw();

}

}

Notice that the method Draw in Shape does not define a body (i.e. no { } ). This is because the method is also marked abstract - which means that the implementation (body) must be provided by sub-types of Shape - Square and Circle so far.

So we need to change Square state that it is a *sub-type* (also known as a sub-class here) of Shape. (Conversely, Shape is then said to be a *super-class* of Square). We also need to add the word override to each of their Draw methods to indicate that this is its implementation of the abstract method of the same name defined in Shape:

public class Square : Shape

{

//Properties

public float CentreX { get; set; }

public float CentreY { get; set; }

public Color LineColor { get; set; }

public float SideLength { get; set; }

//The 'Constructor

public Square(float centreX, float centreY,   
 Color lineColor, float sideLength)

{

LineColor = lineColor;

CentreX = centreX;

CentreY = centreY;

SideLength = sideLength;

}

public override void Draw()

{ //code after this line is not changed

Make equivalent changes to the Circle class

Now we can change our list of shapes to be a list of type Shape:

public static void Execute()

{

var shapes = new List<Shape>();

Your code should now compile, and run. Confirm that it still draws the truck correctly.

Congratulations, you are now using polymorphism: we are successfully invoking the draw method on each of the shapes in our list of shapes, *without having to know whether each shape is a Square or a Circle,* even though the code to do the drawing is different in each case.

Let’s do something more interesting. Into the Shape class, add another abstract method definition to grow the shape by a specified factor (e.g. by 2):

public abstract class Shape

{

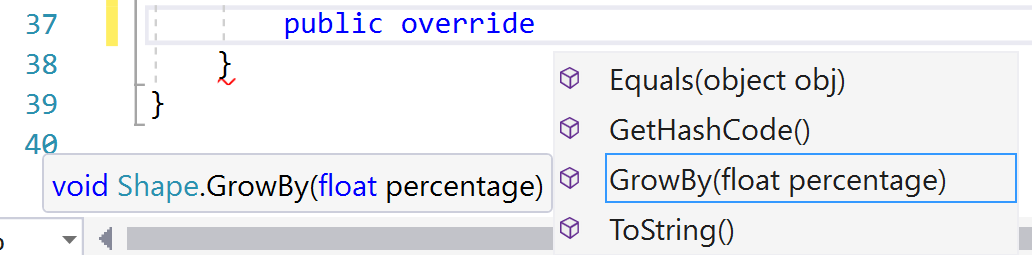
public abstract void Draw();

public abstract void GrowBy(float factor);

}

If you re-build the code you will get a new compile error for both Square and Circle.

What does the error message say?

We fix the error by providing a suitable method marked with override. In fact as soon as we have added the word override (and a space) Visual Studio prompts us with the various methods we could override but have not already overridden:

(Ignore the Equals, GetHashCode and ToString for now). If we double click on GrowBy, Visual Studio will provide a skeleton version of the method. We then need to replace the default implementation (throw new NotImplementedException();) with our own code.

For Square:

public override void GrowBy(float factor)

{

SideLength \*= factor;

}

For Circle:

public override void GrowBy(float factor)

{

Radius \*= factor;

}

Now modify the Execute method such that before drawing each shape, it grows it by a factor of 2:

foreach (var item in list)

{

item.GrowBy(2);

item.Draw();

}

Run the program and paste in a screenshot of the resulting drawing. Is it what you expected?

The problem is that growing each of the shapes means that they are no longer in the correct position relative to each other. We can fix this by moving the centre of each drawing by the same factor. Sdd these two lines of code into each of the two implementations of the draw method:

CentreX \*= factor;

CentreY \*= factor;

Run again and paste in a screenshot of the new double-sized truck.

# Exercise 5: Introducing inheritance

Let’s add a method into the Square class to move its centre to a new position:

public void MoveCentreBy(float xDiff, float yDiff)

{

CentreX += xDiff;

CentreY += yDiff;

}

But notice that if we wanted to add an equivalent method to Circle also then, *in this case*, the implementation would be *exactly* the same. This would violate the DRY principle (Don’t Repeat Yourself) - not only is it wasteful but it runs the risk that you might unintentionally introduce a difference at a future point where you don’t want any difference.

In this case, we can move the entire method (signature and implementation) into the common *super-class* of Square and Circle i.e. Shape. We will also need to move the CentreX and CentreY properties into Shape, *deleting them from Square and Circle*. We might as well also move the LineColor property, just because this further reduces duplication:

public abstract class Shape

{

//Properties

public float CentreX { get; set; }

public float CentreY { get; set; }

public Color LineColor { get; set; }

//Abstract methods - to be implemeted in sub-types

public abstract void Draw();

public abstract void GrowBy(float factor);

//Concrete methods

public void MoveCentreBy(float xDiff, float yDiff)

{

CentreX += xDiff;

CentreY += yDiff;

}

}

We can now modify the Execute method to move all the shapes a little to the right of where they would have been, before drawing them. Make the change and then confirm that this works.

foreach (var item in list)

{

item.MoveCentreBy(20, 0);

item.GrowBy(2);

item.Draw();

}

Now create an EquilateralTriangle class, inheriting from Shape. It can have a SideLength property just like Square, but will need its own implementation of Draw. To calculate the correct offset from the centre, and orient the triangle so it has a flat base, *start* the Draw method with these three lines:

Turtle.Angle = -90;

Turtle.X = CentreX - SideLength / 2; //To ensure shape is centred correctly

Turtle.Y = CentreY - (float) (SideLength \* Math.Sin(60) /3); //Centre is at a third of the height

Modify the execute method such that it draws a small red warning triangle at road level behind the (broken down!) truck. Paste in your drawing and your code for the Equilateral Triangle class and the modified Execute function.

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

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