

<b>ICS4U0 UNIT 1 – USING GRAPHICS TO UNDERSTAND CLASSES AND STRUCTURES IN C#</b>	<b>1</b>
<b>THE CHECKERBOARD PROBLEM</b>	<b>2</b>
BACKGROUND RESEARCH	2
THE PROBLEM	2
<b>HINTS FOR CHECKERBOARD DRAWING PROBLEM</b>	<b>3</b>
DRAWING AND MANIPULATING SHAPES AND IMAGES	3
OUTLINE OF C# CODE FOR DRAWING A CHECKERBOARD	3
EXPLANATION OF VARIOUS STATEMENTS FROM THE PREVIOUS SECTION	4
HOW TO USE THE “GRAPHICS” CLASS FOR DRAWING	4
Overview	4
Creating a Graphics Object	4
<b>USING FRACTALS TO DEEPEN OUR UNDERSTANDING OF CLASSES</b>	<b>5</b>
WHAT ON EARTH IS A FRACTAL?	5
Classification of Fractals	5
FRACTALS IN NATURE	5
A FAMOUS FRACTAL – THE BOUNDARY OF THE MANDELBROT SET	6
The Geometry of the Mandelbrot Set	6
A Primer on Complex Numbers	6
Operations on Complex Numbers	7
The Mandelbrot Sequence	7
Generating a Picture of the Mandelbrot Set	8
Colouring the Exterior of the Mandelbrot Set	9
WRITING A C# PROGRAM TO GENERATE THE MANDELBROT SET	9
Transforming between Co-ordinates in the Complex Plane and Screen Co-Ordinates	10
How to Express this in C#	10
Deciding whether a Point $c$ in the Complex Plane belongs to the Mandelbrot Set	11
UNDERSTANDING THE COLOURING METHODS USED IN THE MANDELBROT PROGRAM	12
CREATING YOUR OWN FRACTAL	13
<b>USING ROBOTS TO UNDERSTAND HOW TO WRITE YOUR OWN CLASSES</b>	<b>16</b>

# THE CHECKERBOARD PROBLEM

## Background Research

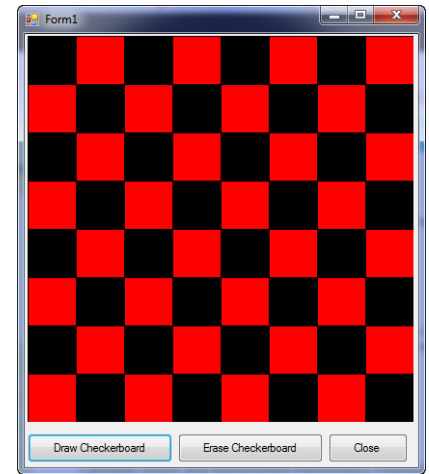
Use MSDN or any other resource to learn how to use the *Windows Graphics Device Interface* within the C# development environment. For the purposes of this course, you will need to understand the following:

- The “Graphics” class
- The “Pen” class
- The “Brush” class and classes derived from it (e.g. “SolidBrush,” “TextureBrush,” “HatchBrush,” etc)
- The “Font” class
- The “Pens” and “Brushes” structures
- The “Color” structure

The Graphics Device Interface used in the .NET framework is usually referred to as the *Windows GDI+*.

## The Problem

Write a C# program that draws a picture of a checkerboard on a picture box. An example is shown at the right.



## GDI

- Short for *Graphics Device Interface*, a Windows standard for representing graphical objects and transmitting them to output devices, such as monitors and printers.
- Windows GDI+ is a class-based *API* for C/C++ programmers.
- It enables applications to use graphics and formatted text on both the video display and the printer.
- Applications based on the Microsoft Win32 API (i.e. 32-bit Windows) do not access graphics hardware directly. Instead, GDI+ interacts with *device drivers* on behalf of applications. GDI+ is also supported by Microsoft Win64 (i.e. 64-bit Windows).
- For languages such as C# and Visual Basic, access to GDI+ is provided through .NET framework classes.
- Simple games that do not require fast graphics rendering use GDI. However, GDI is relatively difficult to use for advanced animation because it does not provide support for various graphics optimizations that are implemented through hardware. For instance, GDI lacks a mechanism for synchronizing with individual [video frames](#) in the [video card](#) and it also lacks hardware [rasterization](#) for 3D. Graphics-intensive applications usually use [DirectX](#) or [OpenGL](#) instead, which allow programmers to exploit the features of modern graphics hardware.

## API

- API, an abbreviation of *Application Programming Interface*, is a set or *library* of routines, protocols and other tools that allow software components to communicate with one another using a common interface.

## Device Driver

- A small program that acts as an *interpreter* between an operating system and a device.
- As such, a device driver simplifies programming because programmers can write the higher-level application code independently of whatever specific hardware the end-user is using.

# HINTS FOR CHECKERBOARD DRAWING PROBLEM

## *Drawing and Manipulating Shapes and Images*

After it is created, a `Graphics` object may be used to draw lines and shapes, render text, or display and manipulate images. The principal objects that are used with the `Graphics` object are:

- The `Pen` class—Used for drawing lines, outlining shapes, or rendering other geometric representations.
- The `Brush` class—Used for filling areas of graphics, such as filled shapes, images, or text.
- The `Font` class—Provides a description of what shapes to use when rendering text.
- The `Color` structure—Represents the different colors to display.

## *Outline of C# Code for Drawing a Checkerboard*

```
public partial class DrawCheckerboardForm : Form
{
    /**
     * The class 'Bitmap' encapsulates a GDI+ bitmap.
     *
     * A GDI+ bitmap consists of the pixel data for a graphics image, as well as the attributes
     * of the image. When the 'Bitmap' class is used to create a 'Bitmap' object, space is allocated
     * in memory to store the bitmap data. Whenever a drawing method is executed, the image data
     * are updated in memory; however, NOTHING is displayed until the 'Paint' event is fired on the
     * control on which the drawing is displayed. Drawing images in this way is much faster than
     * drawing directly to a form, picture box or other control on which images can be drawn.
     */
    Bitmap checkerBoardBitmap = new Bitmap(400, 400); // Bitmap size set to 400 pixels x 400 pixels.

    private void checkerBoardPictureBox_Paint(object sender, PaintEventArgs e)
    {
        e.Graphics.DrawImage(checkerBoardBitmap, 0, 0);
    } //End of method

    private void drawCheckerBoardButton_Click(object sender, EventArgs e)
    {
        // A 'Graphics' object represents a GDI+ drawing surface.
        Graphics checkerBoard = Graphics.FromImage(checkerBoardBitmap);

        // A 'SolidBrush' object represents a brush of a single colour.
        // Brushes are used to fill graphics shapes.
        SolidBrush fillColourBrush = new SolidBrush(Color.Red);

        checkerBoard.Clear(checkerBoardPictureBox.BackColor);

        // Code needs to be inserted here
        .
        .
        .
        checkerBoardPictureBox.Refresh(); // Fire the 'Paint' event on the picture box.

        // Release resources used by the brush and graphics objects.
        checkerBoard.Dispose();
        fillColourBrush.Dispose();

    } //End of method
    .
    .
    .
} // End of class
```

## Explanation of Various Statements from the Previous Section

The following statements are used to create new objects:

```
Bitmap checkerBoardBitmap = new Bitmap(400, 400);  
  
SolidBrush fillColourBrush = new SolidBrush(Color.Red);
```

Diagram illustrating the components of object creation statements:

- Class Name:** Points to `Bitmap` and `SolidBrush`.
- Object Name:** Points to `checkerBoardBitmap` and `fillColourBrush`.
- Create a new Object:** Points to the `new` keyword.
- Constructor Method Call:** Points to the parentheses containing parameters, such as `(400, 400)` and `(Color.Red)`.

The following statement also creates a new object:

```
Graphics checkerBoard = Graphics.FromImage(checkerBoardBitmap);
```

However, the “**new**” keyword is conspicuous by its absence! The reason for this is that the “Graphics” class does not expose any constructor methods (i.e. none of its constructors is visible outside the class). Therefore, “Graphics” objects are created by calling various methods such as the static method “FromImage” shown in the above example.

## How to use the “Graphics” Class for Drawing

The material found below is adapted from the following MSDN page:  
[How to: Create Graphics Objects for Drawing](#).

Before you can draw lines and shapes, render text or display and manipulate images with GDI+, you need to create a [Graphics](#) object. The [Graphics](#) object represents a GDI+ drawing surface and is the object that is used to create graphical images.

### Overview

1. Create a [Graphics](#) object.
2. Use the [Graphics](#) object to draw lines and shapes, render text or display and manipulate images.

### Creating a Graphics Object

As shown in the following table, there are several ways to create a “Graphics” object:

Method of Creating “Graphics” Object	Example
1. Receive a reference to a graphics object as part of the <a href="#">PaintEventArgs</a> in the <a href="#">Paint</a> event of a form or control. This is usually how you obtain a reference to a graphics object when creating painting code for a control. Similarly, you can also obtain a graphics object as a property of the <a href="#">PrintPageEventArgs</a> when handling the <a href="#">PrintPage</a> event for a <a href="#">PrintDocument</a> .	<pre>private void Form1_Paint(object sender,                         PaintEventArgs e) {     // Declare the Graphics object 'g'     // This is a reference to the object     // provided by the parameter 'e'     Graphics g=e.Graphics; }</pre>
2. Call the <a href="#">CreateGraphics</a> method of a control or form to obtain a reference to a <a href="#">Graphics</a> object that represents the drawing surface of that control or form. Use this method if you want to draw on a form or control that already exists.	<pre>// Create a Graphics object 'g' for a form Graphics g = this.CreateGraphics(); // Create a Graphics object 'gr' // for a picture box Graphics gr =PictureBoxName.CreateGraphics();</pre>
3. Create a <a href="#">Graphics</a> object from any object that inherits from <a href="#">Image</a> . This approach is useful when you want to alter an already existing image.	<pre>Bitmap myBitmap = new Bitmap("myPic.jpg"); Graphics g = Graphics.FromImage(myBitmap);</pre>

Adapted from  
[http://en.wikipedia.org/wiki/Constructor\\_%28object-oriented\\_programming%29](http://en.wikipedia.org/wiki/Constructor_%28object-oriented_programming%29)

In [object-oriented programming](#), a **constructor method** (often shortened to **constructor** and sometimes shortened to **ctor**) in a [class](#) is a special type of [subroutine](#) called to [create an object](#). It prepares the new object for use, often accepting [parameters](#) that the constructor uses to set [member variables](#) required for the object to reach a valid state. It is called a constructor because it constructs the values of data members of the class.

# USING FRACTALS TO DEEPEN OUR UNDERSTANDING OF CLASSES

## What on Earth is a Fractal?

A mathematically precise definition of fractals requires knowledge of mathematics that is far beyond the high school level. Therefore, we shall only consider an intuitive definition, which will allow us to understand the essential ideas without being burdened by the complexities of mathematical technicalities.

The term *fractal* denotes a shape that is *recursively constructed* or *self-similar*, that is, a shape that appears similar at all scales of magnification and is therefore often referred to as “infinitely complex.”

## Classification of Fractals

*Adapted from a Wikipedia article*

Fractals can be classified according to their self-similarity. There are three types of self-similarity found in fractals:

- **Exact Self-Similarity**

This is the strongest type of self-similarity; the fractal appears identical at all scales. Fractals defined by iterated function systems often display exact self-similarity.

- **Quasi-Self-Similarity**

This is a loose form of self-similarity; the fractal appears approximately (but not exactly) identical at all scales. Quasi-self-similar fractals contain small copies of the entire fractal in distorted and degenerate forms. Fractals defined by recurrence relations are usually quasi-self-similar but not exactly self-similar.

- **Statistical Self-Similarity**

This is the weakest type of self-similarity; the fractal has numerical or statistical measures that are preserved across all scales. Random fractals are examples of fractals that are statistically self-similar, but neither exactly nor quasi-self-similar.

## Fractals in Nature

*Adapted from a Wikipedia article*

Approximate fractals are easily found in nature. These objects display self-similar structure over an extended, but finite, scale range. Examples include *clouds*, *snowflakes*, *mountains*, *river networks* and *systems of blood vessels*. *Trees* and *ferns* are fractal in nature and can be modelled on a computer using recursive algorithms. The recursive nature is clear in these examples — a branch from a tree or a frond from a fern is a miniature replica of the whole, not identical, but similar in nature.



A fractal is formed when pulling apart two glue-covered acrylic sheets.



High voltage breakdown within a 4" block of acrylic creates a fractal Lichtenberg figure.



Fractal branching occurs on a microwave-irradiated DVD

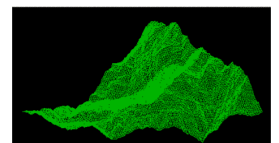
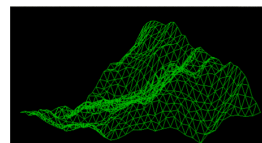
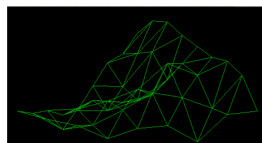
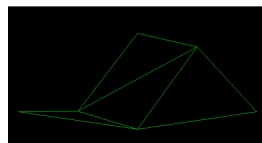
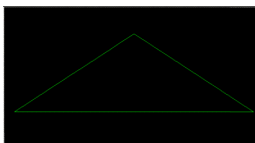


Romanesco broccoli showing very fine natural fractals



A fractal fern computed using an Iterated function system

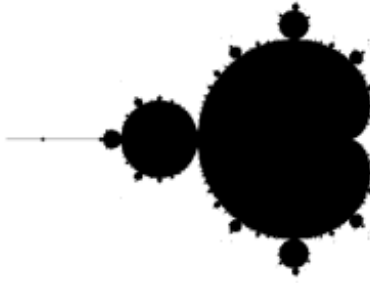
The surface of a mountain can be modelled on a computer using a fractal. Start with a triangle in 3D space and connect the central points of each side by line segments, resulting in 4 triangles. The central points are then randomly moved up or down within a defined range. The procedure is repeated, cutting the range in half after each iteration. The recursive nature of the algorithm guarantees that the whole is statistically similar to each detail.



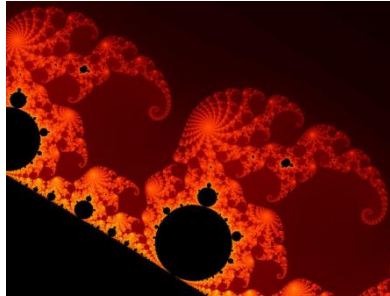


## A Famous Fractal – The Boundary of the Mandelbrot Set

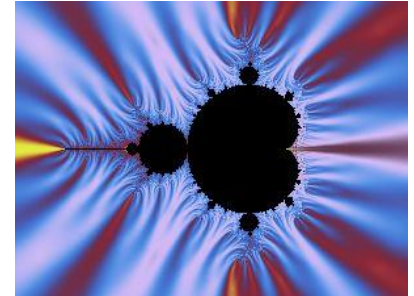
### The Geometry of the Mandelbrot Set



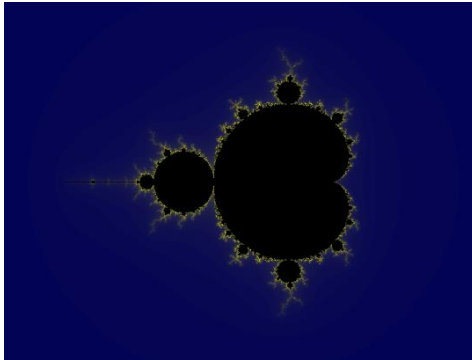
The Mandelbrot Set  
Notice the self-similarity at several scales.



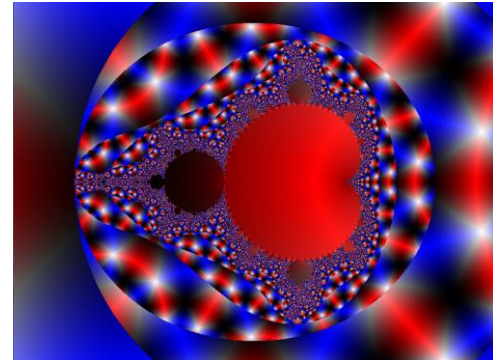
A Close-up View of the Boundary  
Self-similarity is evident here at a tiny scale.



The Exterior of the Mandelbrot Set Coloured  
using the “Triangle Inequality” Method



The Exterior of the Mandelbrot Set Coloured  
using the “Iterations” Method



The Interior and Exterior of the Mandelbrot Set  
Coloured using the “Trigonometric” Method

### A Primer on Complex Numbers

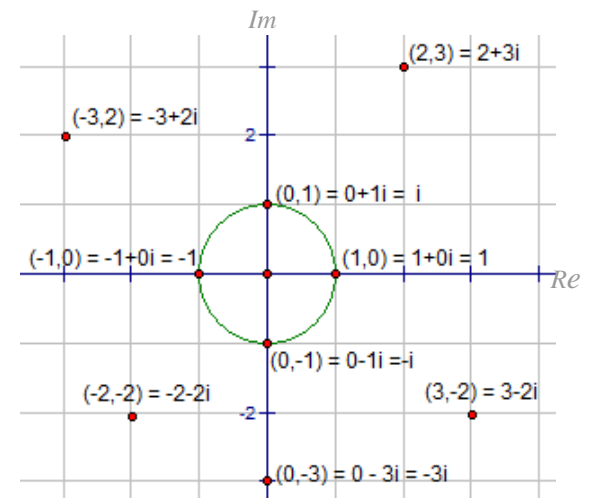
To understand how the Mandelbrot set is generated, it is necessary to have a basic understanding of **complex numbers**. Complex numbers are of the form  $a + bi$ , where  $a \in \mathbb{R}, b \in \mathbb{R}$  and  $i = \sqrt{-1}$ . The real number  $a$  is called the **real part** of  $a + bi$  and the real number  $b$  is called the **imaginary part** of  $a + bi$ .

Since  $i = \sqrt{-1}$ , it follows that  $i^2 = -1$ . Due to our intimate familiarity with real numbers, which have the property that  $x^2 \geq 0$  for all  $x \in \mathbb{R}$ , this at first appears to be a peculiar or even absurd notion. However, once we become well acquainted with the **geometry of complex numbers**, it becomes easier to accept the “reality” that the square of the imaginary number  $i$  actually equals  $-1$ .

Complex numbers are plotted by making use of the Cartesian plane. We only need to become accustomed to a few minor modifications.

- The Cartesian plane is renamed the **complex plane**.
- The  $x$ -axis is renamed the **real axis**.
- The  $y$ -axis is renamed the **imaginary axis**.

Using this framework, it becomes possible to give a geometric meaning to multiplication by the imaginary number  $i$ :



Multiplication by  $i$  is equivalent to a counter-clockwise rotation by  $90^\circ$  about the origin.

Let's examine how this works by starting at the complex number 1 on the real axis and following the unit circle.

$li = i$ (i.e. $(1,0) \rightarrow (0,1)$ )	$i(i) = i^2 = -1$ (i.e. $(0,1) \rightarrow (-1,0)$ )
$-li = -i$ (i.e. $(-1,0) \rightarrow (0,-1)$ )	$-i(i) = -i^2 = -(-1) = 1$ (i.e. $(0,-1) \rightarrow (1,0)$ )

### Operations on Complex Numbers

Addition	Subtraction	Multiplication	Division
$(a + bi) + (c + di)$ $= (a + c) + (b + d)i$ <b>e.g.</b> $(2 - 3i) + (-5 - i)$ $= (2 + (-5)) + (-3 + (-1))i$ $= -3 - 4i$	$(a + bi) - (c + di)$ $= (a - c) + (b - d)i$ <b>e.g.</b> $(2 - 3i) - (-5 - i)$ $= (2 - (-5)) + (-3 - (-1))i$ $= 7 - 2i$	$(a + bi)(c + di)$ $= ac + adi + bci + bdi^2$ $= (ac - bd) + (ad + bc)i$ <b>e.g.</b> $(2 - 3i)(-5 - i)$ $= 2(-5) + 2(-i) - 3i(-5) - 3i(-i)$ $= -13 + 13i$	$\frac{a + bi}{c + di}$ $= \left( \frac{a + bi}{c + di} \right) \left( \frac{c - di}{c - di} \right)$ $= \frac{(ac + bd) + (bc - ad)i}{c^2 + d^2}$

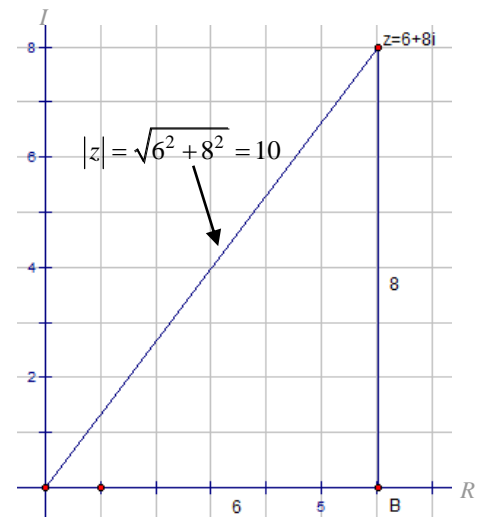
### The Modulus (Absolute Value) of a Complex Number

The **modulus** or **absolute value** of a complex number is an extremely important operation that is used to measure the “size” of a complex number. As shown in the diagram to the right, the modulus of a complex number  $z$ , denoted  $|z|$ , is equal to the **distance from the origin to  $z$** .

The following are some formal definitions, including the definition of  $|z|$ .

#### Definitions

1. The symbol  $\mathbb{C}$  is used to denote the set of complex numbers.
2. Suppose that  $z \in \mathbb{C}$ , where  $z = x + iy, x \in \mathbb{R}, y \in \mathbb{R}$ . Then  $\text{Re}(z) = x$  denotes the **real part of  $z$**  and  $\text{Im}(z) = y$  denotes the **imaginary part of  $z$** .
3. The **modulus** or **absolute value** of  $z = x + iy$  is denoted  $|z|$  and is equal to  $\sqrt{x^2 + y^2} = \sqrt{\text{Re}(z)^2 + \text{Im}(z)^2}$



### The Mandelbrot Sequence

For any **fixed value**  $c \in \mathbb{C}$ , consider the **Mandelbrot sequence**, which is defined **recursively** as follows for all  $n \in \mathbb{N}$ :

$$\begin{cases} z_n = 0, & \text{if } n = 1 \\ z_{n+1} = z_n^2 + c, & \text{if } n \geq 2 \end{cases}$$

For a particular value of  $c$ , there are **two** possibilities.

1. The value of  $|z_n|$  grows larger and larger indefinitely as  $n$  gets larger. That is,  $|z_n|$  “blows up” to infinity.
2. There is a constant  $D$  such that the value of  $|z_n| \leq D$  no matter how large  $n$  is made. In other words, in this case the value of  $|z_n|$  remains bounded. It does not “blow up” to infinity.

The **Mandelbrot set** consists of all the values of  $c \in \mathbb{C}$  for which the Mandelbrot sequence **does not “blow up” to infinity**.

### Generating a Picture of the Mandelbrot Set

To generate a picture of the Mandelbrot set, the following is done:

1. If the chosen value of  $c$  causes  $|z_n|$  to “blow up” to infinity, then  $c$  is *not plotted* on the complex plane.
2. If the chosen value of  $c$  does not cause  $|z_n|$  to “blow up” to infinity, then  $c$  is *plotted* on the complex plane.

As usual, a *specific example* should help to clarify matters. Suppose that we choose  $c = 0.5 + 0.5i$ . The following table, constructed using Microsoft Excel, gives the values of  $z_n$  and  $|z_n|$  for  $n = 1, \dots, 14$ .

$n$	$z_n$	$ z_n $
1	0	0
2	$0.5 + 0.5i$	0.70711
3	$0.5 + i$	1.11803
4	$-0.25 + 1.5i$	1.52069
5	$-1.6875 - 0.25i$	1.70592
6	$3.28515625 + 1.34375i$	3.54935
7	$9.48658752441406 + 9.328857421875i$	13.305
8	$3.46776206069622 + 177.498044870794i$	177.53192
9	$-31493.0305592448 + 1231.54197170139i$	31517.10122
10	$990294278.677464 - 77569977.3995685i$	993327669.9
11	$9.74665656987549E+017 - 1.53634209631866E+017i$	9.867E+17
12	$9.26369672541763E+035 - 2.99483975733211E+035i$	9.73577E+35
13	$7.68470118484163E+071 - 5.5486574506296E+071i$	9.47851E+71
14	$2.8267032795879E+143 - 8.52795489702672E+143i$	8.9842E+143

For  $c = 0.5 + 0.5i$ , we see that the Mandelbrot sequence is not bounded. After 6 iterations,  $|z_n|$  is already greater than 2 and by 14 iterations,  $|z_n|$  explodes to a value greater than  $10^{43}$  googols! Therefore,  $c = 0.5 + 0.5i$  is *not* in the Mandelbrot set and so, it is *not plotted* on the complex plane.

Now let's see if  $c = 0.2 + 0.1i$  fares any better than  $c = 0.5 + 0.5i$ .

$n$	$z_n$	$ z_n $
1	0	0
2	$0.2 + 0.1i$	0.22361
3	$0.23 + 0.14i$	0.26926
4	$0.2333 + 0.1644i$	0.28541
5	$0.22740153 + 0.17670904i$	0.28799
6	$0.220485371028619 + 0.180367812121662i$	0.28486
7	$0.216081251188073 + 0.17953692795453i$	0.28094
8	$0.214457598615653 + 0.177589128053755i$	0.27844
9	$0.2144541632011 + 0.176170675885312i$	0.27754
10	$0.214954481072396 + 0.175561069755114i$	0.27754
11	$0.215383739719543 + 0.175475277291451i$	0.27782
12	$0.215598582395064 + 0.175589042902713i$	0.27805
13	$0.21565123674327 + 0.175713497467862i$	0.27817
14	$0.215630222716514 + 0.17578566608286i$	0.27820

In this case, after 14 iterations  $|z_n|$  remains very small, which makes it very likely that  $c = 0.1 + 0.2i$  *is* a member of the Mandelbrot set. Since the Mandelbrot sequence appears to be bounded for  $c = 0.1 + 0.2i$ , then this point *is plotted* on the complex plane.



## Colouring the Exterior of the Mandelbrot Set

When it comes to colouring, the exterior of the Mandelbrot set is where all the action is! This is particularly true near the boundary of the set. The points lying outside the boundary of the Mandelbrot set all have something in common;  $|z_n|$  eventually “blows up” to infinity. More importantly for these points, however, is that  $|z_n|$  does not always “blow up” to infinity at the same rate. For some points,  $|z_n|$  goes to infinity rather slowly. For others,  $|z_n|$  approaches infinity very rapidly. We can use this as the basis for colouring (e.g. iterations method).

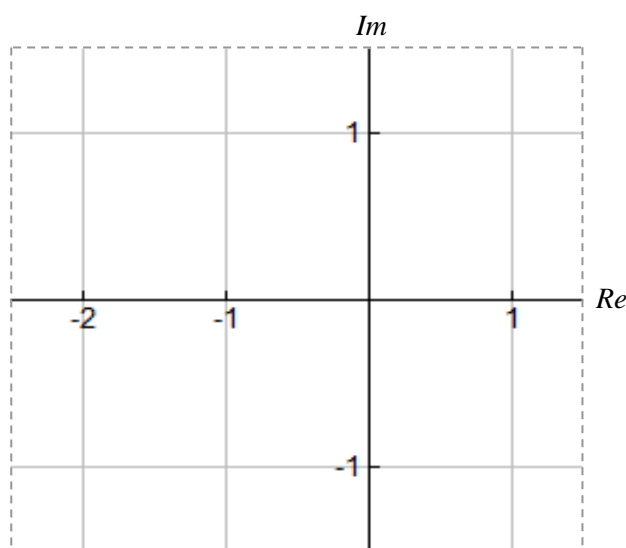
The following is a description of just a few colouring methods.

Iterations Method	Modulus Method	Exponential Smoothing Method
The Mandelbrot sequence is generated until $ z_n $ exceeds a certain fixed value. The number of iterations required to exceed this value is then used to determine the colour of the pixel located at $c$ on the complex plane.	The Mandelbrot sequence is generated until $ z_n $ exceeds a certain fixed value. The value of $ z_n $ is then used to determine the colour of the pixel located at $c$ on the complex plane.	On a small scale (i.e. high degree of magnification), the “iterations” method can lead to colour “banding” due to the rather abrupt transition from one colour to another. To prevent this problem, a second sequence is computed at the same time as the Mandelbrot sequence is generated: $s_{n+1} = s_n + e^{- z_{n+1} }$ The value of $s_n$ is used to determine the colour of the pixel located at $c$ on the complex plane. This allows for smoother colour transitions on a minute scale.

Other colouring methods include *decomposition*, *binary decomposition*, *orbit traps*, *direct orbit traps*, *distance estimator*, *Gaussian integer*, *gradient*, *triangle inequality average* and *lighting*.

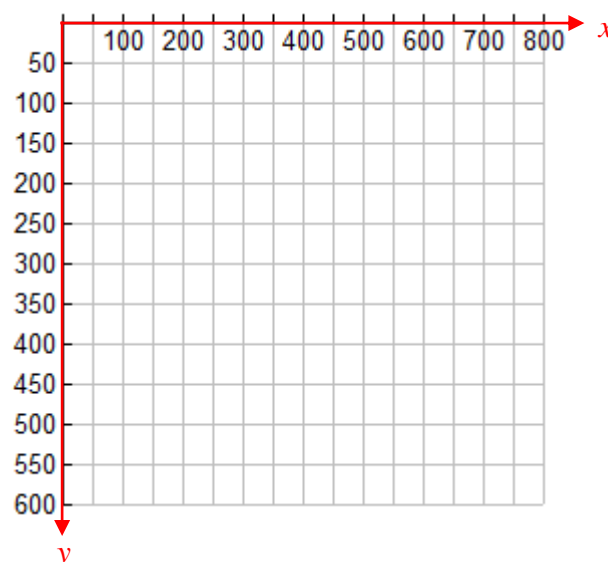
## Writing a C# Program to Generate the Mandelbrot Set

The Mandelbrot set lies in a region of the complex plane that is very close to the origin. Generally, the points in the Mandelbrot set and its immediate exterior are plotted for real values ranging from  $-2.5$  to  $1.5$  and for imaginary values ranging from  $-1.5$  to  $1.5$ . This poses a slight problem when writing computer programs because screen co-ordinates do not correspond to the ranges given above. Therefore, it is necessary to find equations that can transform between screen co-ordinates and actual complex plane co-ordinates.



The Mandelbrot set lies in this region of the complex plane.

To render the Mandelbrot set on a computer screen, co-ordinates in the range shown at the left must be translated to screen co-ordinates.

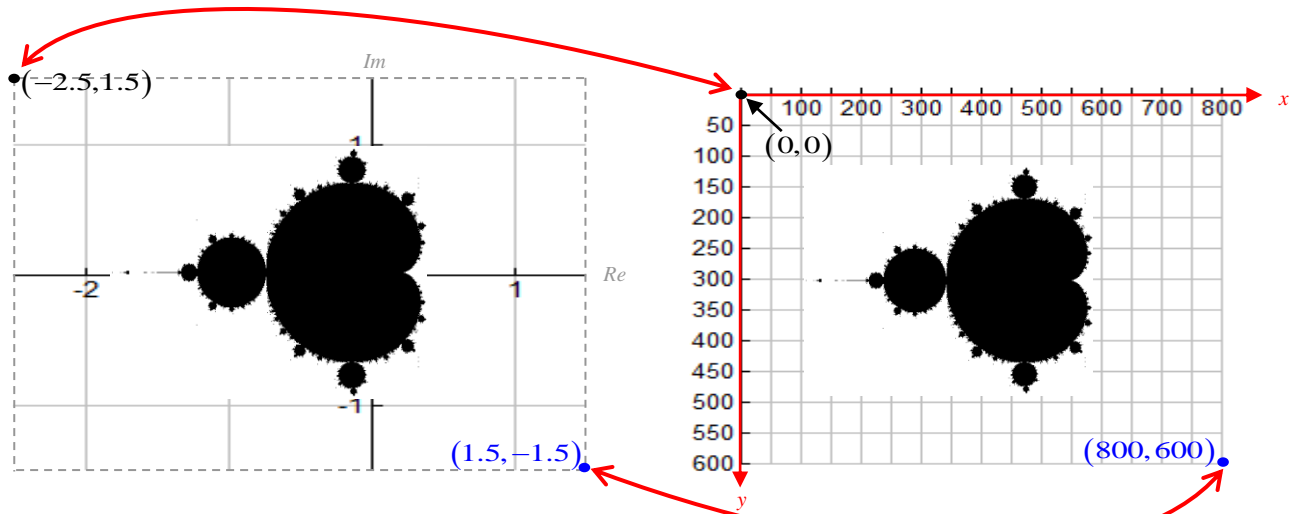


## Transforming between Co-ordinates in the Complex Plane and Screen Co-Ordinates

The following tables show correspondences between specific screen co-ordinates  $(x, y)$  and co-ordinates in the complex plane  $(\text{Re}(z), \text{Im}(z))$  ( $z$  represents a point in the complex plane).

$x$	$\text{Re}(z)$
0	-2.5
800	1.5

$y$	$\text{Im}(z)$
0	1.5
600	-1.5



When transforming between co-ordinate systems, it is important that proportions be preserved. In other words, the rendering of a picture in one co-ordinate system should look exactly the same as the rendering in any other co-ordinate system. To ensure that the change of co-ordinate systems does not distort the image in any way, the transformation between co-ordinate systems must be *linear*!

This makes it very easy to find equations that relate  $\text{Re}(z)$  to  $x$  and  $\text{Im}(z)$  to  $y$ . A little bit of reflection back to grade 9 mathematics should immediately bring to mind the familiar form  $y = mx + b$ . As shown below, the required equations are obtained by making a simple observation and by performing some simple calculations.

$$b = -2.5$$

$$m = \frac{1.5 - (-2.5)}{800 - 0} = \frac{4}{800}$$

$x$	$\text{Re}(z)$
0	-2.5
800	1.5

$y$	$\text{Im}(z)$
0	1.5
600	-1.5

$$b = 1.5$$

$$m = \frac{-1.5 - 1.5}{600 - 0} = -\frac{3}{600}$$

$$\therefore \text{Re}(z) = \frac{4}{800}x - 2.5 \text{ and } \text{Im}(z) = -\frac{3}{600}y + 1.5$$

## How to Express this in C#

//Instantiation of the 'Complex' class. The 'Complex' object 'c' is created.

Complex c = new Complex(0,0);

.

.

.

//The real and imaginary parts of the 'Complex' object 'c' are determined

//using the above equations translated into C#. (x and y represent screen co-ordinates of course)

c = new Complex(4.0d / mandelbrotSetBitmap.Width\*x - 2.5d, -3.0d / mandelbrotSetBitmap.Height\*y + 1.5d);

### *Deciding whether a Point $c$ in the Complex Plane belongs to the Mandelbrot Set*

```
private void displayMandelbrotSetButton_Click(object sender, EventArgs e)
{
    Graphics mandelbrotSetImage = Graphics.FromImage(mandelbrotSetBitmap);

    //Instantiations of the 'Complex' class. Create two complex objects, 'c' and 'z.'
    //For both 'Complex' objects, the real and imaginary parts are initially set to 0.
    Complex z = new Complex(0, 0);
    Complex c = new Complex(0,0);

    //For efficiency reasons, copy the 'Width' and 'Height' properties to variables.
    //This is called "caching" the values of properties.
    int width = mandelbrotSetBitmap.Width;
    int height = mandelbrotSetBitmap.Height;

    mandelbrotSetImage.Clear(mandelbrotSetPictureBox.BackColor);
    mandelbrotSetPictureBox.Refresh();

    //Traverse the bitmap one pixel at a time, column by column. Screen co-ordinates are (x,y).
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            //Transform screen co-ordinates (x,y) to co-ordinates in the complex plane
            c = new Complex(4.0d/width*x - 2.5d, -3.0d/height*y + 1.5d);
            z = 0;
            int iterations=0;

            //For the value of c corresponding to screen co-ordinates (x,y),
            //generate terms of the Mandelbrot sequence. The loop terminates
            //as soon as |z| exceeds 'maxModulus' or at the 100th term.
            do
            {
                z = Complex.Add(Complex.Pow(z, 2),c); //z=z^2+c
                iterations++;
            }while(Complex.Abs(z) < maxModulus && iterations<=100);

            mandelbrotSetBitmap.SetPixel(x, y, pixelColour(colouringMethod, iterations-1, z));

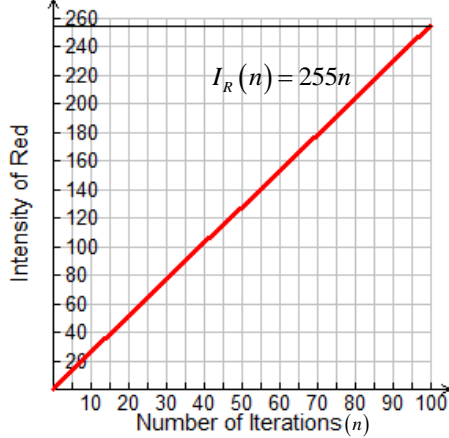
        } //end inner for
    } //end outer for

    ////Display the Mandelbrot set by firing the 'Paint' event on 'mandelbrotSetPictureBox.'
    mandelbrotSetPictureBox.Refresh();
}
```

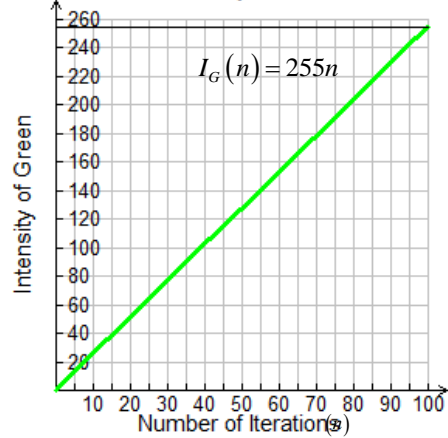
## Understanding the Colouring Methods used in the Mandelbrot Program

The colours that are displayed on computer monitor screens and mobile-device screens are “mixtures” of various intensities of the primary colours red, green and blue. For a colour model known as “RGB 24-bit,” the intensity of each primary colour is represented by an integer whose value is between 0 and 255. (This would be between 00000000 and 11111111 in binary.)

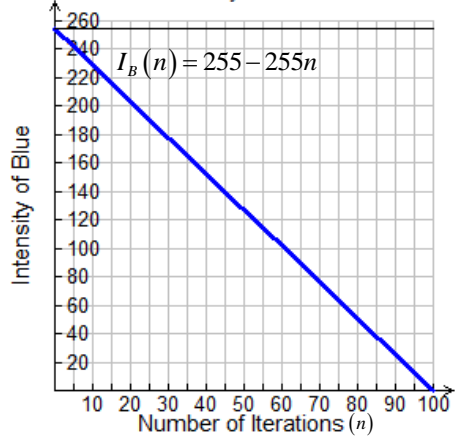
Red Colour Intensity for Iterations Method



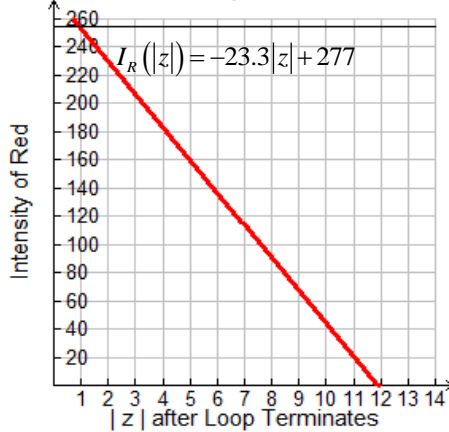
Green Colour Intensity for Iterations Method



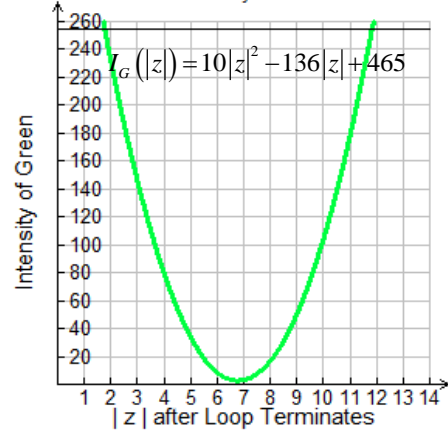
Blue Colour Intensity for Iterations Method



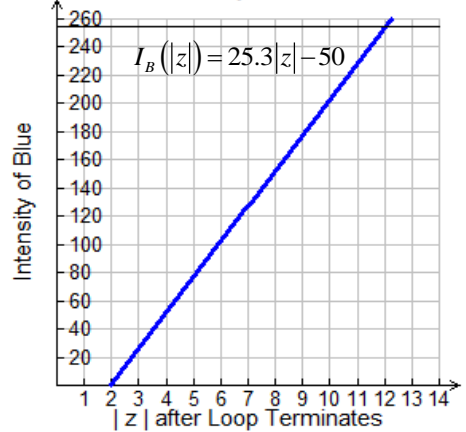
Red Colour Intensity for Modulus Method



Green Colour Intensity for Modulus Method



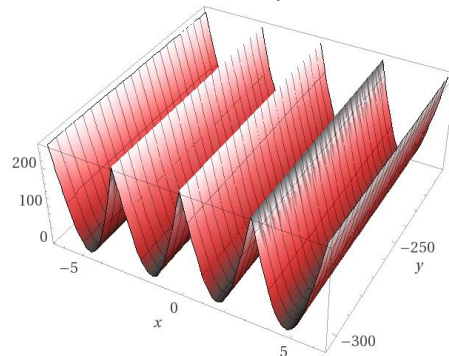
Blue Colour Intensity for Modulus Method



Intensity of Red for Trig Method

$$I_R(z) = 255 - 255|\sin(\operatorname{Re}(z))|$$

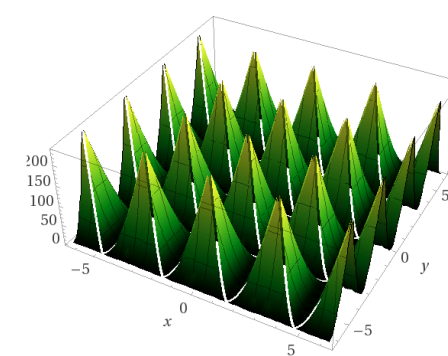
$$z = x + iy$$



Intensity of Green for Trig Method

$$I_G(z) = \min(255 - 255|\sin(\operatorname{Re}(z))|, 255 - 255|\cos(\operatorname{Im}(z))|)$$

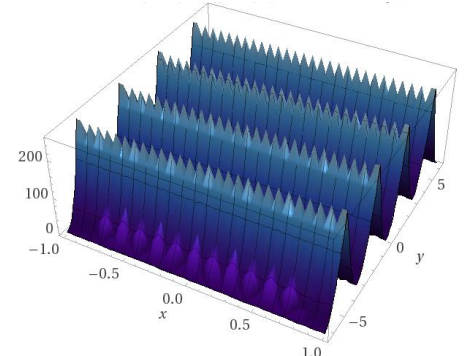
$$z = x + iy$$



Intensity of Blue for Trig Method

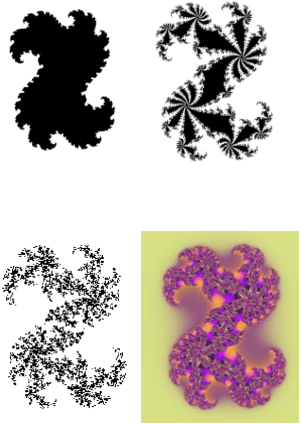
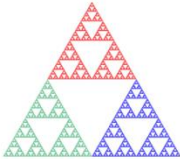
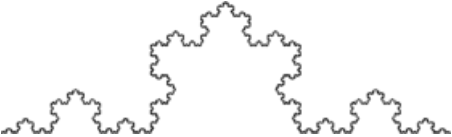
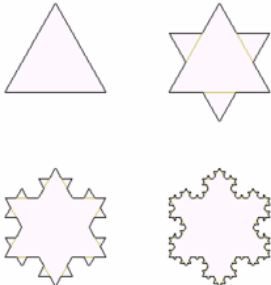
$$I_B(z) = 255 - 255|\cos(\operatorname{Im}(z))|$$

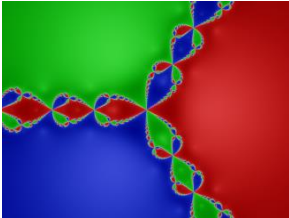
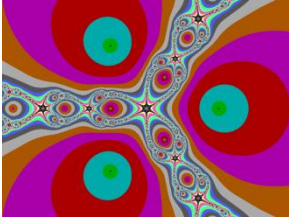
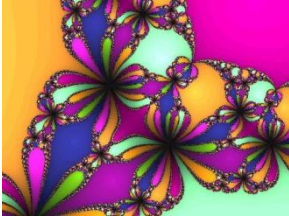
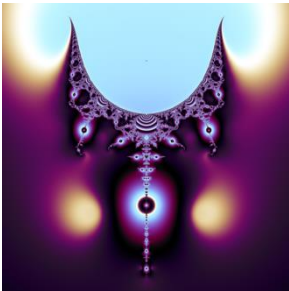
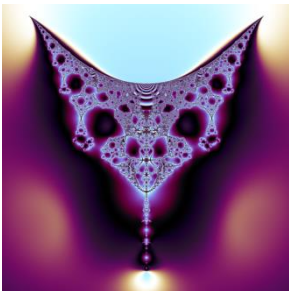
$$z = x + iy$$



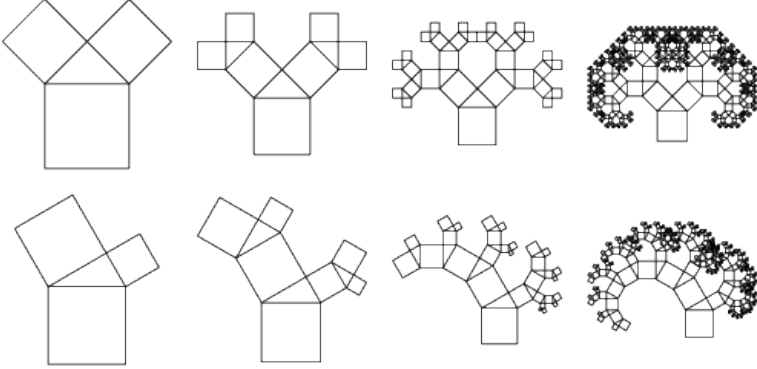
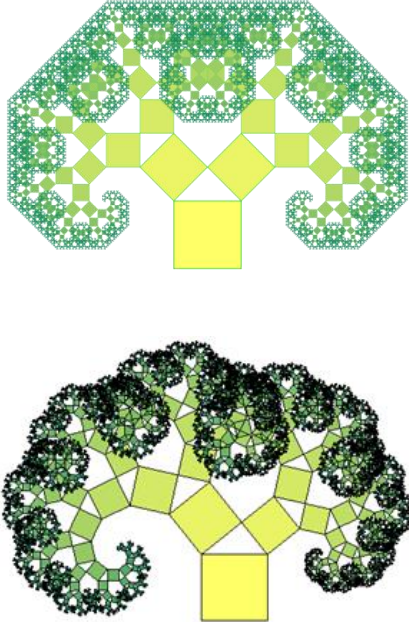
## Creating your own Fractal

Listed below are some examples of fractals that are appropriate for our fractal project. If you don't like any of these suggestions, you are free to choose any other fractal provided that your project can be completed in a reasonably short time. It would be a good idea to ask me about the appropriateness of your choice before forging ahead.

Fractal Name	Description	Sample Picture(s)
Julia Sets	<p>Julia sets are closely related to the Mandelbrot set. As with the Mandelbrot set, the border of a Julia set is a fractal and its exterior can be coloured in a variety of interesting ways. Unlike the Mandelbrot set, there are an infinite number of different Julia sets. The black and white pictures at the right are three examples of different Julia sets. The fourth picture is a coloured version of the third Julia set. Only points that are just outside the Julia set are coloured.</p> <p>To generate a Julia set use the following algorithm:</p> <ol style="list-style-type: none"> <li>1. Choose a point in the Mandelbrot set or just outside the Mandelbrot set. Call this value <math>c</math>. (The value chosen for <math>c</math> is known as the <i>index</i> of the Julia set.)</li> <li>2. Choose <math>z_1</math> in the complex plane in such a way that <math>-2 \leq \text{Re}(z_1) \leq 2</math> and <math>-1.5 \leq \text{Im}(z_1) \leq 1.5</math>.</li> <li>3. Using the value of <math>z_1</math> chosen in step 2 and the value of <math>c</math> chosen in step 1, generate the resulting Mandelbrot sequence until <math> z_n </math> exceeds 2 or a maximum number of iterations is exceeded.</li> <li>4. If <math> z_n  \leq 2</math>, then colour the pixel corresponding to <math>z_1</math> black. Otherwise, set the colour according to the colouring scheme that you have chosen.</li> <li>5. Repeat steps 2 to 4 until all pixels in the range have been coloured. (It is very important to understand that the value of <math>c</math> remains the same throughout the process.)</li> </ol>	
Fractal Mountains	<ol style="list-style-type: none"> <li>1. Begin with a triangle in 3-D space.</li> <li>2. Find the co-ordinates of the midpoint of each side of the triangle.</li> <li>3. Use line segments to connect the midpoints to each other. This produces 4 triangles.</li> <li>4. Move each midpoint up or down by a randomly selected amount.</li> <li>5. Repeat the same process on each of the four resulting triangles.</li> <li>6. Stop when the triangles become "smaller" than some fixed value.</li> </ol> <p><b>Note:</b> To display a 3-D object on a 2-D screen, some knowledge of projective geometry is required. In essence, each point <math>(x, y, z)</math> in 3-D space must be projected onto a point <math>(u, v)</math> in 2-D space.</p>	See page 5
Sierpinski Triangle	<p>If you are interested in this one, do a search on "Sierpinski Triangle" to find out more. This one is more interesting if the user is allowed to choose the vertices of the triangle (the triangle need not be equilateral). An even more interesting variation is to begin with other polygons such as quadrilaterals, pentagons, etc.</p>	
Koch Snowflake	<p>Begin with a single line segment and then recursively alter each line segment as follows:</p> <ol style="list-style-type: none"> <li>1. Divide the line segment into three segments of equal length.</li> <li>2. Draw an equilateral triangle that has the middle segment from step 1 as its base.</li> <li>3. Remove the line segment that is the base of the triangle from step 2.</li> </ol>  <p style="text-align: center;">The Koch Curve</p>	 <p style="text-align: center;">The first four iterations of the Koch Snowflake</p>

Fractal Name	Description	Sample Picture(s)
Newton Fractals	<p>Like the Mandelbrot set, the Newton fractal is based on a recursive formula:</p> $z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n)},$ <p>where <math>p(z)</math> is a complex polynomial function (i.e. <math>p</math> takes as “input” a complex value <math>z</math> and produces a complex “output” <math>p(z)</math>) and <math>p'(z)</math> is the <i>derivative</i> of <math>p(z)</math>.</p> <p>For any complex polynomial function <math>p(z)</math>, the above equation defines a sequence <math>z_1, z_2, z_3, \dots</math> in the complex plane. For some choices of the point <math>z_1</math>, the sequence will converge toward a <i>root</i> of the polynomial function. For other choices of <math>z_1</math>, the sequence <i>will not</i> converge toward a root.</p> <p>Newton fractals can be coloured in a variety of ways including the following...</p> <ul style="list-style-type: none"> <li>• The colour is determined by the number of iterations required for <math>z_n</math> to come within a specified distance from a root.</li> <li>• The colour is determined by the root toward which the sequence converges.</li> </ul> <p>The equation given above is used as the basis of what is known as Newton’s method or the Newton-Raphson method. This method provides us with a powerful algorithm for finding approximations of solutions of equations.</p>	 <p><math>p(z) = z^3 - 1</math></p> <p>The colour is determined by which root is reached.</p>  <p><math>p(z) = z^3 - 1</math></p> <p>The colour is determined by the number of iterations required to reach a root.</p>  <p><math>p(z) = z^8 + 15z^4 - 16</math></p> <p>This polynomial has 8 complex roots, allowing for 8 colours when the iterations method is used.</p>
Nova Fractal	<p>The Nova fractal is a generalization of the Newton fractal. It is based on the following recursive formula, which reduces to the Newton fractal recursive formula when <math>R = 1</math> and <math>c = 0</math>:</p> $z_{n+1} = z_n - R \frac{p(z_n)}{p'(z_n)} + c, \quad R \in \mathbb{R}, \quad c \in \mathbb{C}$ <p>For the Nova fractal, <math>p(z) = z^n - 1</math>, where <math>n</math> can have any complex value but is usually set to 3 or a value close to 3.</p>	 



<i>Fractal Name</i>	<i>Description</i>	<i>Sample Picture(s)</i>
Pythagoras Tree	<p>The Pythagoras tree is a plane (2-D) fractal constructed from squares. Invented by the Dutch mathematics teacher Albert E. Bosman in 1942, it is named after the ancient Greek mathematician Pythagoras because each triple of touching squares encloses a right triangle.</p> <p>The construction of the Pythagoras tree begins with a square. Upon this square are constructed two squares, each scaled down by a linear factor in such a way that the corners of the squares coincide pairwise. The same procedure is then applied recursively to the two smaller squares, <i>ad infinitum</i>.</p> <p>The illustrations below show the first few iterations in the construction process.</p> <p>The first row shows the construction process for the case in which the enclosed right triangle is isosceles. This leads to a “balanced” tree, that is, a tree exhibiting bilateral (“left-right”) symmetry. In the second row, the enclosed right triangle is not isosceles, leading to a tree that does not exhibit bilateral symmetry.</p>  <p>Many other variations are possible.</p>	

## USING ROBOTS TO UNDERSTAND HOW TO WRITE YOUR OWN CLASSES

Complete the following table. List as many properties (characteristics, attributes) and methods (actions, services) that you can imagine could be ascribed to the robot shown below.



Properties (Characteristics)	Methods (Actions)