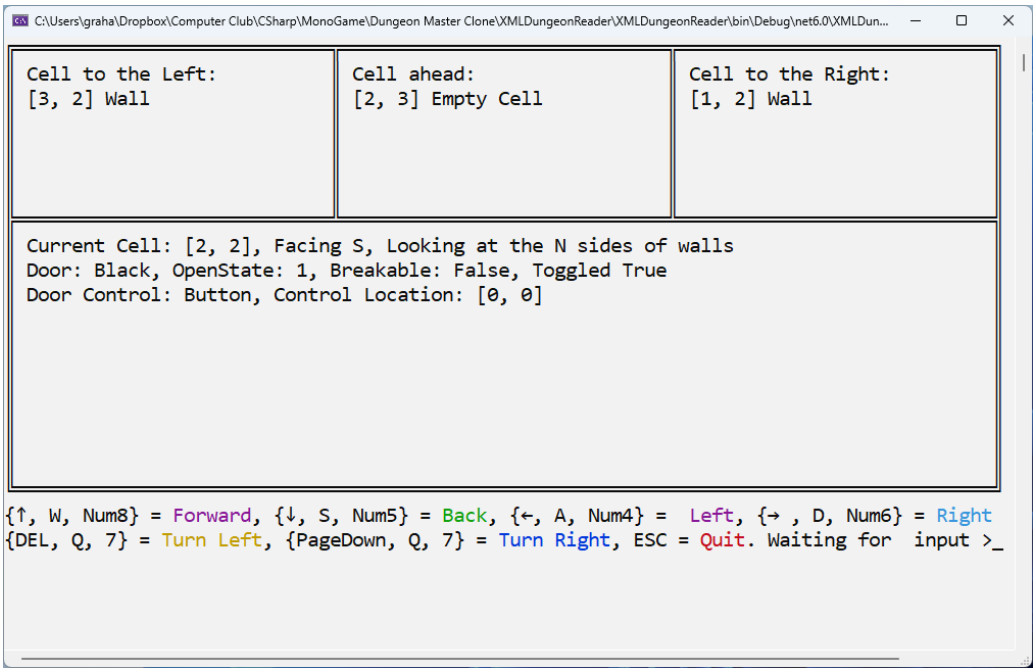
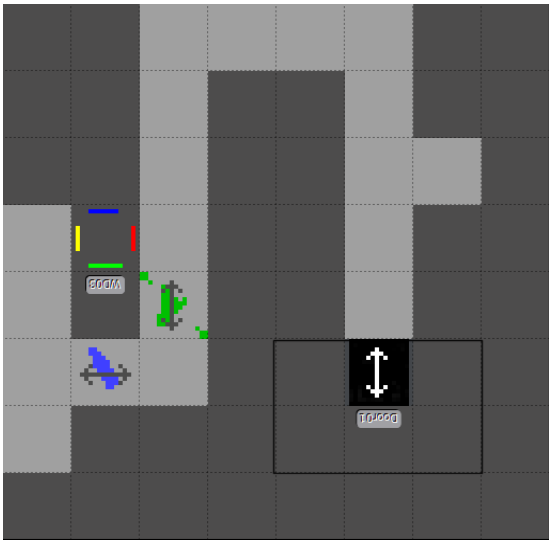
XMLDungeonReader VS2022 Console Application

Before diving into Monogame and dealing with the graphics, it seemed sensible to explore the idea of using a 2D array of objects to hold data from Level00.tmx and test it out:



The arrow keys only need to be pressed to make a move, or Esc to quit.
The level starts at grid ref 2,2 (3 blocks from the left, 3 blocks from the top) as we are using screen drawing coordinates.

What we are looking at here is this:



The map is upside-down as we are looking to the South.

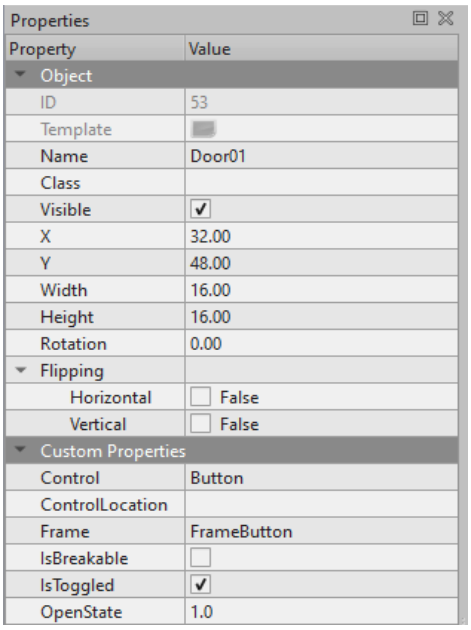
There is a 2x3 2D array containing the cell coordinates of the immediate area round the player, so descriptions can be given of the cell in front, to the left and right.

The player is currently standing on the square marked:

There is an object associated with a door

Door:

```
IsBreakable:                Boolean False

IsToggled:                  Boolean True

OpenState                   Float 1.0 (closed)

Control:                    Button

Frame:                      FrameButton
```



And the doorType calculated from the gid = “83” : Black

Put all this together and the screenshot makes sense:

```
Cell to the Left:
[3, 2] Wall

Cell ahead:
[2, 3] Empty Cell

Cell to the Right:
[1, 2] Wall

Current Cell: [2, 2], Facing S, Looking at the N sides of walls
Door: Black, OpenState: 1, Breakable: False, Toggled True
Door Control: Button, Control Location: [0, 0]
```

Pressing the down arrow twice moves back to the edge of the map and shows a warning message that you are at the border and does not move any further.

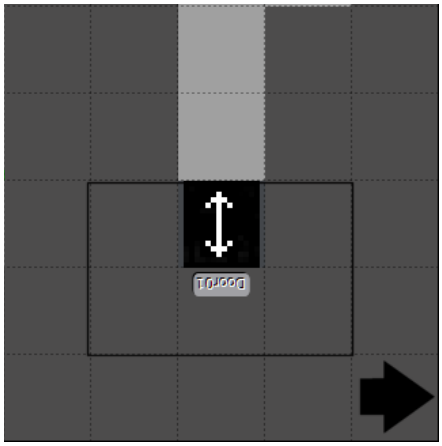
```
Current Cell: [2, 0], Facing S, Looking at the N sides of walls
Walls: N:Wall, E:Wall, S:Wall, W:Wall,

You are at the edge of the map
```

Pressing right arrow twice then Page=Down or NumPad9 or E takes you to the top left corner (bottom right on the screenshots above and below (south at the top) This is confirmed by the “Outside MapBorder” messages on the Cell ahead / Right

Cell to the Left: [0, 1] Wall	Cell ahead: Outside Map Border	Cell to the Right: Outside Map Border
Current Cell: [0, 0], Facing W, Looking at the E sides of walls Walls: N:Wall, E:Wall, S:Wall, W:Wall,		

Player is standing on the black arrow, facing East.

As reported in the screenshot above, cells ahead and to the right are outside the map border.



This has proved to be a valuable tool to test out the idea, and to learn how to read xml files in C#

The code is available on Github as part of the Monogame Dungeon Master Clone Project and the principles of translating a Tiled.tmx XML file into program code are described here.

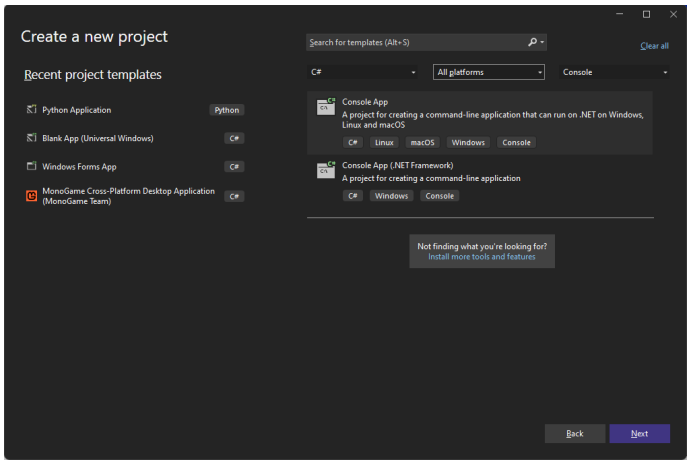
Creating XMLDungeonReader

Start Visual Studio

Create a new project

Narrow the choices with:

C# All platforms Console



Either will do, but I am choosing “Console App” to make it multi-platform. *Next.*

Choose your location and give it a name eg XMLDungeonReader. *Next.*

Framework .Net 6.0 (Long Term Support)

My personal choice is to tick the “Do not use top-level statements” box, as I prefer the usual Program.cs with Main()

Download or copy/paste the .cs files from Github

Make sure the namespaces are the same on all files

Create a Directory called “Data” in bin/debug/net6.0 directory of the project
You will also need these files placed in the Data directory:

- Champions.txt
- Items.txt
- Level00.tmx
- TiledToItem.txt

There are 6 object classes for some of the data:

Class Name	Purpose
Cell.cs	Each cell represents a Tile on the map. It has wall types, Floor and ceiling Doors
Champion.cs	24 Champion Objects contain all the data of the avaliable Champions
Door.cs	Each Door has the type, direction, OpenState, Toggled, etc
Floor.cs	Used for FloorActuators for Target, Toggled, Location etc
Item.cs	Used for all the available items and their properties
WallWriting.cs	Used for wall writing eg “Hall of Champions”

There is a Globals static class called Shared.cs for storing global variables and setting up the Dungeon from various text files.

There is a static class called PlayerMove.cs which handles keyboard input and key binding to move the player through the Dungeon.

Finally there is Program.cs containing static void Main().

<Static void Main()>

A couple of module level variables. These are currently set to output a white console with black text to save ink/toner when printing this manual, but changing these two will return the output to the default white on black.

```
static ConsoleColor Fore = ConsoleColor.Black;
static ConsoleColor Back = ConsoleColor.White;
```

When `static void Main(string[] args)` starts it sets the Console then calls Shared.LoadItems()

```
Console.BackgroundColor = Back;
Console.ForegroundColor = Fore;
Console.WindowWidth = 90;                       // Set Console width
Console.WindowHeight = 25;                      // Set Console height
Shared.LoadItems();
```

<Shared.LoadItems()>

This procedure loads the text file Items.txt and creates Item objects using Item.cs

```
internal class Item
{
    public string Name { get; set; }
    public string ImageName { get; set; }
    public List<string> ImageNames { get; set; } = new List<string>();
    public Dictionary<string, string> Properties { get; set; } = new Dictionary<string, string>
    {
        {"Weight", "0"  }, {"Action",""      }, {"Damage", "0"   }, {"Active",""        },
        {"Health", "0"  }, {"Stamina", "0"   }, {"Mana", "0"     }, {"Luck", "0"        },
        {"Strength","0" }, {"Dexterity","0"  }, {"Wisdom", "0"   }, {"Vitality", "0"    },
        {"AntiMagic","0"}, {"AntiFire","0"   }, {"Load", "0"     }, {"Distance", "0"    },
        {"Swing", "0"   }, {"Thrust", "0"    }, {"Club", "0"     }, {"Parry", "0"       },
        {"Brandish","0" }, {"Slash", "0"     }, {"Jab", "0"      }, {"Chop", "0"        },
        {"Melee", "0"   }, {"Stab", ""       }, {"Cleave","0"    }, {"Disrupt","0"      },
        {"Bash", "0"    }, {"Stun", "0"      }, {"Berzerk", "0"  }, {"Shoot Damage", "0"},
        {"Fireball", "0"}, {"Lightning", "0" }, {"Confuse", "0"  }, {"Value","0"        },
        {"Steal", "0"   }, {"Fight", "0"     }, {"Throw", "0"    }, {"Shoot", "0"       },
        {"Identify", "0"}, {"Heal", "0"      }, {"Influence", "0"}, {"Defend", "0"      },
        {"Light", "0"   }, {"Dispell", "0"   }, {"Armour", "0"   }, {"Sharp Resist", "0"},
        {"Block","0"    }, {"Hit", "0"       }, {"Shield", "0"   }, {"Food", "0"        },
        {"Water", "0"   }, {"Calm", "0"      }, {"Fireshield","0"}, {"Spellshield", "0" },
        {"Charges", "0" }, {"Skill13","0"    }, {"Skill14","0"   }, {"Skill15","0"      },
        {"Window",""    }, {"Freeze Life","0"}, {"Climb Down","0"}, {"Punch", "0"       },
        {"Blow Horn","0"}
    };
    public Item(string name)
    {
        Name = name;
        ImageName = name;
    }
}
```

All the numerous statistics of an individual item are stored in the Dictionary “Properties” which contain the default values as above.

The constructor just takes the name and image name. (ImageName is not used in this App as it is Console based!)

All the lines are read from the file via ProcessTextFile() which removes any blank lines and any starting with a #, and returns them as a List<string>

Example line from Items.txt:

```
Eye Of Time=Type:Weapon;Locations:Pouch;States:0,1;Weight:0.1;Charges:3;Active:Weapon;Punch:32:0;
Freeze Life:70:0
```

The line is split with “=” to give a key/value eg “Eye of Time” and

“`Type:Weapon;Locations:Pouch;States:0,1;Weight:0.1;Charges:3;Active:Weapon;Punch:32:0; Freeze Life:70:0`”

The value is split via “;” to a string[] array called data

1. `Type:Weapon`
2. `Locations:Pouch`
3. `States:0,1`
4. `Weight:0.1`
5. `Charges:3`
6. `Active:Weapon`
7. `Punch:32:0`
8. `Freeze Life:70:0`

This data is then passed to the Item object and stored in it’s Properties Dictionary:

```
if (Items[key].Properties.ContainsKey(properties[0]))
```

</Shared.LoadItems()>

The next stage is similar but loads in the Champions from Champions.txt.
Some filtering is required as the Champion may have multiple items in their Inventory, Quiver or Pouch.

Next stage is to load the data from the Tiled map Level00.tmx

Shared.GetLevelData("Level00.tmx");

to create the Cell objects, and fill Dictionaries of Objects

The text file TiledToItem is read in to create a `Dictionary<string, string> map` that holds the data in the text file. Eg the first line in the file is:

`1=Wall:Wall;Wall;Wall;Wall` which goes into the Dictionary as
`map["1"] = "Wall;Wall;Wall;Wall"`

A List would not work here as the indexes are not continuous eg there is no item "32"
Each of the indexes represnts a tile from the Tileset used to draw the dungeon and export it as .tmx. The first tile "1" is a solid wall, so all directions (NESW) are of type "Wall".

Line 10 (10=Wall:Alcove;Wall;Alcove;Wall) is a Tile with Alcoves on the East and West sides

When the Dictionary is complete the .tmx file is read directly using `System.Xml.Linq;` library:

```
XElement doc = XElement.Load(Path.Combine(AppPath, "Data", level));   // Load .tmx file
```

There are some local variables set directly from the XML file:

```
XElement doc = XElement.Load(Path.Combine(AppPath, "Levels", level));   // Load .tmx file
string sWidth = doc.Attribute("width").Value;                           // get width of map -> width="20"
string sHeight = doc.Attribute("height").Value;                         // get height of map -> height="19"
string sGridSizeX = doc.Attribute("tilewidth").Value;                   // get the width of each tile on the map
string sGridSizeY = doc.Attribute("tileheight").Value;                  // get the height of each tile on the map
int.TryParse(sWidth, out Shared.MapWidth);                              // set Shared. variables from string values
int.TryParse(sHeight, out Shared.MapHeight);                            // set Shared. variables from string values
int.TryParse(sGridSizeX, out mapGridSizeX);                             // set module variables from string values
int.TryParse(sGridSizeY, out mapGridSizeY);                             // set module variables from string values
```

Using these variables the Array of cells can be created

```
Cells = new Cell[Shared.MapWidth, Shared.MapHeight];                     // define array size eg 20 wide, 19 high
for (int col = 0; col < Cells.GetLength(0); col++)                      // populate array with Cell objects
{
    for (int row = 0; row < Cells.GetLength(1); row++)
    {
        Cells[col, row] = new Cell(col, row);
    }
}
```

Here comes the clever bit, Linq is used to create an iterable collection using SQL – like commands:

```
// create iterable of all "layer" items using xml.linq
IEnumerable<string> query = from item in doc.Descendants("layer")
                            select item.Attribute("name") + item.Value;
```

This is basically asking is to return all instances in the XML document called "layer" where "layer" is a direct descendant of the document root.
This means it will find:

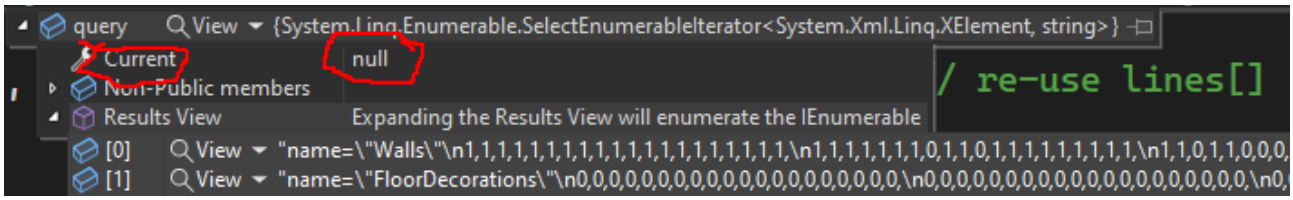
```
Only 2 descendants "layer"
<layer id="1" name="Walls" width="20" height="19">
<layer id="7" name="FloorDecorations" width="20" height="19" visible="0">
```

The `select item.Attribute("name") + item.Value` part will collect the “name” and the value of that tag, which is the grid of numbers representing the “Walls” or “FloorDecorations” data.

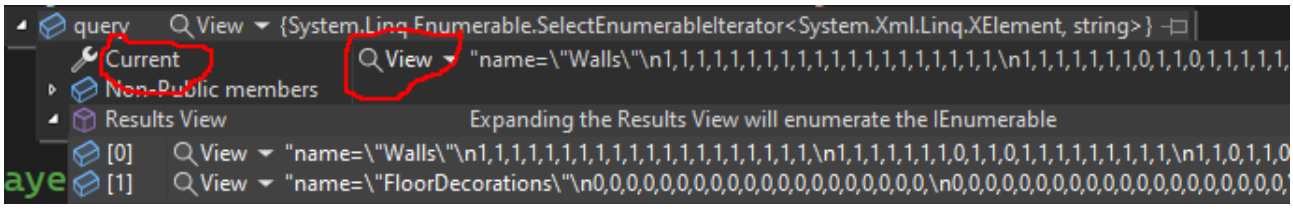
This can be observed in the foreach loop:

```
foreach (string layer in query)                                    // iterate layers to extract image information
{
    lines = layer.Split("\n", StringSplitOptions.RemoveEmptyEntries);   // re-use lines[]
    /*
    [0] "name=\"Walls\""
    [1] "1,1,1,1,1,1,1,1,1,1,1 -> "
    [2] "1,1,1,1,1,1,1,0,1,1,0,1 -> "
    */
    string name = lines[0].Substring(6).Replace("\"", "");              // clean lines[0] -> name="Walls" -> Walls
    ProcessData(name, lines);
}
```

On the first iteration the query is not fully populated and the Current value is Null



However you can see the 2 “layer” fields [0] and [1] in the results view.
On the next iteration the Current value is present:



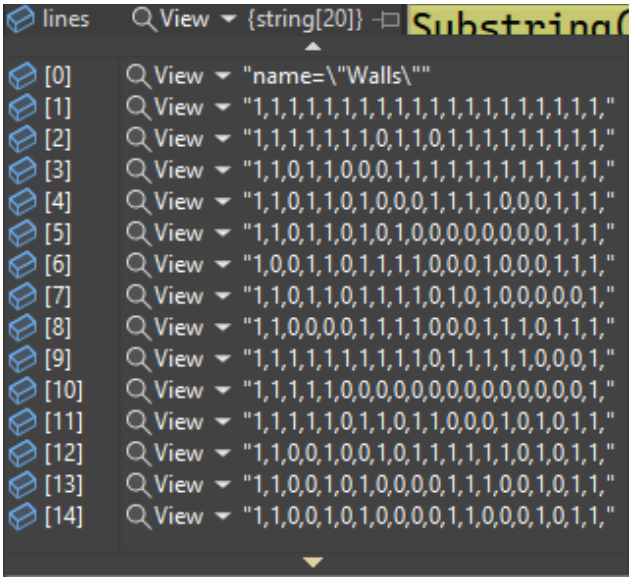
When the lines[] variable is re-populated with

```
lines = layer.Split("\n",
StringSplitOptions.RemoveEmptyEntries);
```

Which splits long strings shown above using the newline character.

It contains this: (screenshot) →

So we now have an array of rows from the .tmx containing the Tiles used



The name is extracted from the first line (lines[0]) eg “Walls” and passed to `ProcessData(name, lines);` for further processing

<ProcessData()>

The purpose of this procedure is to use nested for loops to iterate all the Tiles in the array and use the Tile type to add data to the previously created Cell objects.

Example using the second row lines[1] = 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,

In the nested loop each of these is examined.
If it is not “0” (it is not an empty cell) then an attempt to interpret the single number using the map Dictionary is made with:

```
if (map.TryGetValue(cellType, out string info))
```

In the case of cellType “1” this will return the string: "Wall;Wall;Wall;Wall"

This is passed to a for loop to change the cell properties. In the case of "Wall;Wall;Wall;Wall" the Cell Walls Dictionary is populated with:

```
public Dictionary<string, string> Walls { get; set; } = new Dictionary<string, string>
{
    {"N", "Wall" },
    {"E", "Wall" },
    {"S", "Wall" },
    {"W", "Wall" }
};
```

In the text file Null is used to represent an empty string eg:

46=WallWriting:Null;Text;Null;Null

This was done to minimise mistakes creating the file with nothing between the semicolons:
46=WallWriting:;Text;;
This is catered for with `imageTypes = ConvertNullToString(infos[1]);` which converts the word “Null” into an empty string:

Null;Text;Null;Null → "";"Text";"";""

</ProcessData()>

<GetTiledObjects("Doors", doc)>

This procedure goes through all the object layers in the .tmx file.
In this example “Doors” has been passed as a parameter.

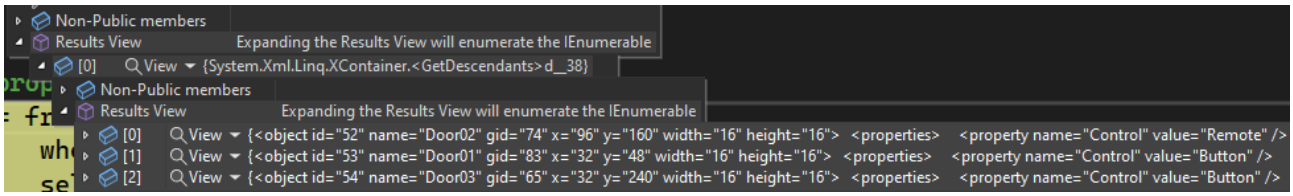
The Linq query:

```
var objectQuery = from item in doc.Descendants("objectgroup")
                  where item.Attribute("name").Value == objectName
                  select item.Descendants("object");
```

is asking : return “object” from the Descendants “objectgroup” where name == “Doors”

This should get the data from:

```
<objectgroup id="17" name="Doors" visible="0">
  <object id="52" name="Door02" gid="74" x="96" y="160" width="16" height="16">
   <properties>
    <property name="Control" value="Remote"/>
    <property name="ControlLocation" value="7,9"/>
    <property name="Frame" value="Frame"/>
    <property name="IsBreakable" type="bool" value="false"/>
    <property name="IsToggled" type="bool" value="false"/>
    <property name="OpenState" type="float" value="1"/>
   </properties>
  </object>
  <object id="53" name="Door01" gid="83" x="32" y="48" width="16" height="16">
   <properties>
    <property name="Control" value="Button"/>
    <property name="ControlLocation" value=""/>
    <property name="Frame" value="FrameButton"/>
    <property name="IsBreakable" type="bool" value="false"/>
    <property name="IsToggled" type="bool" value="true"/>
    <property name="OpenState" type="float" value="1"/>
   </properties>
  </object>
  <object id="54" name="Door03" gid="65" x="32" y="240" width="16" height="16">
   <properties>
    <property name="Control" value="Button"/>
    <property name="ControlLocation" value=""/>
    <property name="Frame" value="FrameButton"/>
    <property name="IsBreakable" type="bool" value="false"/>
    <property name="IsToggled" type="bool" value="true"/>
    <property name="OpenState" type="float" value="1"/>
   </properties>
  </object>
 </objectgroup>
```



You can see here eg [0] <object id=”52” name=”Door02” etc and the <properties> </properties>

The next query is used to distinguish objectgroups that do not have properties eg Champions:

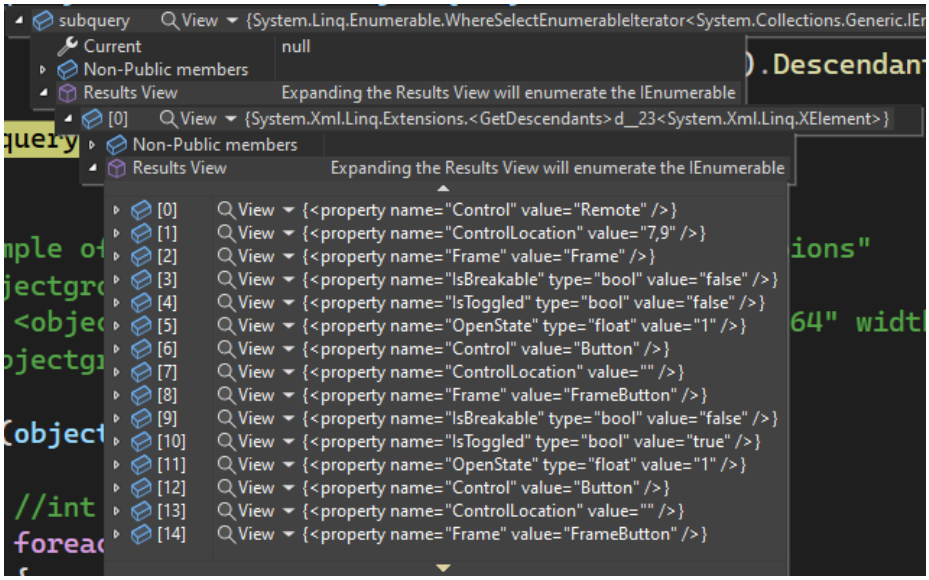
```
<objectgroup id="10" name="Champions" visible="0">
  <object id="25" name="C00" gid="43" x="176" y="64" width="16" height="16"/>
  <object id="26" name="C02" gid="43" x="240" y="48" width="16" height="16"/>
```

To separate these out this subquery is used:

```
// check if <properties> </properties> are available
var subquery = from item in objectQuery
               where item.Descendants("properties").Descendants("property").Count() > 0
               select item.Descendants("properties").Descendants("property");
```

Checking the subquery.Count() selects for lack of properties.

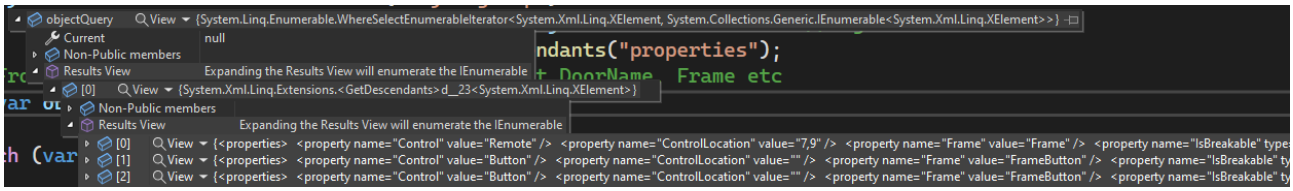
This is the output when searching for “Doors”, so clearly the .Count() is > 0



The Doors query does have properties so a new query is used to refine the search:

```
objectQuery = from item in doc.Descendants("objectgroup")
               where item.Attribute("name").Value == objectName           // eg DoorFrames
               select item.Descendants("object").Descendants("properties");
```

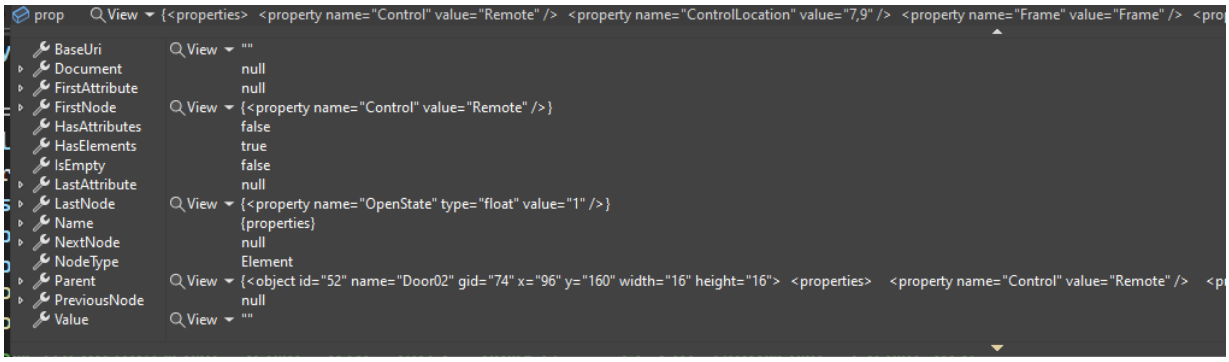
is asking return all the properties where the name of the objectgroup == “Doors”
The output of this query is:



So each of the 3 iterables is a set of properties for each of the doors.

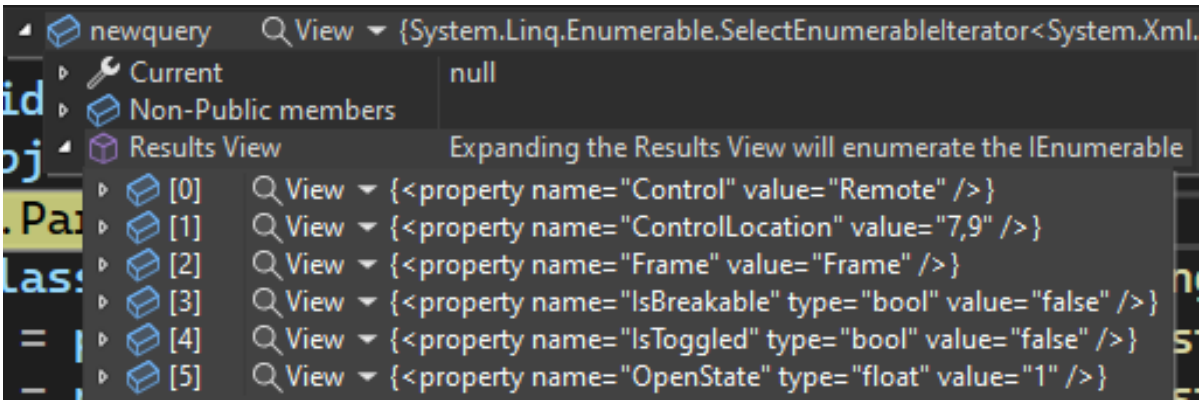
This goes via a nested foreach loop to obtain a list of properties for each Door:

```
foreach (var obj in objectQuery)
{
    foreach (var prop in obj)
    {
```



Finally the last query:

```
var newquery = from item in prop.Descendants("property") select item;
```



The Parent object is used to populate local variables as below

```
string name = prop.Parent.Attribute("name").ToString().Substring(6).Replace("\"", "");// name="Door01" -> Door01
string gid = prop.Parent.Attribute("gid").ToString().Substring(5).Replace("\"", "");  //gid="62" -> "62"
string objClass = "";
if (prop.Parent.Attribute("class") != null)
    objClass = prop.Parent.Attribute("class").ToString().Substring(7).Replace("\"", "");//"Decorations"-> Decorations
string x = prop.Parent.Attribute("x").ToString().Substring(3).Replace("\"", "");        //x="61" -> 61
string y = prop.Parent.Attribute("y").ToString().Substring(3).Replace("\"", "");        //y="32" -> 32
if (int.TryParse(x, out int colX))
     colX = colX / mapGridSizeX;
if (int.TryParse(y, out int rowY))
   rowY = (rowY - mapGridSizeY) / mapGridSizeY;
```

</GetTiledObjects("Doors", doc)>

This information is now passed to:

<SetInfoFromGID(“Doors”, “Door02”, “74”, 6, 9)>

This procedure checks the Dictionary for “74” and finds:

Door:Doortrack;Grate;Doortrack;Grate

This is separated into “Door” and “Doortrack;Grate;Doortrack;Grate”

A key is generated based on the current level (0) and the x,y coordinates of the cell eg “0-6-9”

A new Door object is added to the Dictionary Doors using the key above and passing
“0-6-9”, “Door02”, “Grate”, 6, 9 to the constructor

The Cell’s Dictionary Doors is populated:

```
public Dictionary<string, string> Doors { get; set; } = new Dictionary<string, string>
{
    {"N", "Doortrack" },
    {"E", "Grate" },
    {"S", "Doortrack" },
    {"W", "Grate" }
};
```

Similar processes are used for WallWriting, Stairs and FloorActuators

</SetInfoFromGID(“Doors”, “Door02”, “74”, 6, 9)/>

From static void Main()

```
Cell.PlayerFacing = "S";  // Level00.tmx starts facing S on coords [2,2]
Shared.CurrentCellCoords = new Point(2, 2);
```

An important concept to get your head round is the compass layout of the map, and the compass layout of the player party may be different.

The map has a fixed NESW layout, but the player may be facing eg South, so their view of the map is reversed: The North walls of each cell are immediately in front of the player.

When obtaining data from each cell it is important to take this into account.

The statement `Cell.PlayerFacing = "S";`

has this effect on the static Facing property of ALL cell objects:

```
public  static string PlayerFacing
{
    get { return playerFacing; }
    set
    {
        playerFacing = value;
        if (value == "N")
        {
            facing = "S";
            left = "E";
            right = "W";
        }
        else if (value == "E")
        {
            facing = "W";
            left = "S";
            right = "N";
        }
        else if (value == "S")
        {
            facing = "N";
            left = "W";
            right = "E";
        }
        else if (value == "W")
        {
            facing = "E";
            left = "N";
            right = "S";
        }
    }
}
public static string Facing
{
    get { return facing; }
    private set { facing = value; }
}
```

Note the use of static variables. When setting the PlayerFacing to “S”, it automatically changes the internal static values:

```
facing = "N";
left = "W";
right = "E";
```

This allows the accurate return of cell contents based on its observed position.

```
Shared.CurrentCellCoords = new Point(2, 2);
```

This sets the player position in the map, standing on Cells[2,2]

PlayerMoves can now be calculated from here.

Next create a “viewport”

This is a smaller group of Cell objects representing the cells closest to the player.
In the Monogame version, 4 rows deep are needed, 5 columns across at the back of the viewport so an Array of Cells 5 wide by 4 high is needed. For the Console app only 3 x 3 is required

<Point[,] viewport = GetNearbyCells()>

```
private static Point[,] GetNearbyCells()
{
    /*
    return a 3 row x 3 col array of Points of the cells around the player
    start with facing south on cell[2,2] then move S to cell[2,3]->
```

0,0	1,0	2,0
0,1	1,1	2,1
0,2	1,2	2,2

3,4	2,4	1,4
3,3	2,3	1,3
3,2	2,2	1,2

3,5	2,5	1,5
3,4	2,4	1,4
3,3	2,3	1,3

-1,-1	-1,-1	-1,-1
-1,-1	-1,-1	-1,-1
-1,-1	0, 0	1, 0

```
    */

    Point[,] retValue = new Point[3, 3];    // create 3,3, array of Points

    retValue[0, 0] = GetCorrectedCoordinate(-1, 2);
    retValue[1, 0] = GetCorrectedCoordinate(0, 2);
    retValue[2, 0] = GetCorrectedCoordinate(1, 2);
    retValue[0, 1] = GetCorrectedCoordinate(-1, 1);
    retValue[1, 1] = GetCorrectedCoordinate(0, 1);
    retValue[2, 1] = GetCorrectedCoordinate(1, 1);
    retValue[0, 2] = GetCorrectedCoordinate(-1, 0);
    retValue[1, 2] = GetCorrectedCoordinate(0, 0);
    retValue[2, 2] = GetCorrectedCoordinate(1, 0);

    return retValue;
}


private static Point GetCorrectedCoordinate(int toSide, int ahead)
{
/// return the coords of a cell toSide / ahead distances from the original toside can be -ve left, 0 centre +ve right
/// maxX and maxY are the sizes of the Shared.map - 1 to confine coordinates to 1 from border
/// example current cell [2,2]: fromX = 2, fromY = 2, toSide = -1, ahead = 2, facing = "S"
    int maxX = Shared.Cells.GetLength(0) - 1;       // eg level 1 Shared.map 20 X 19 Cells[,] = [20, 19] -> maxX = 19
    int maxY = Shared.Cells.GetLength(1) - 1;       // eg level 1 Shared.map 20 X 19 Cells[,] = [20, 19] -> maxY = 18
    int fromX = Shared.CurrentCellCoords.X;
    int fromY = Shared.CurrentCellCoords.Y;
    int x = -1;                                     // Point index X = 0
    int y = -1;                                     // Point index Y = 0
    if (Cell.PlayerFacing == "N") // looking up the map: x range 1-maxX; y range maxY-1 (ahead decreases row)
    {
        if (fromY - ahead >= 0)                             // eg 2 - 2 = 0 (facing N so ahead reduces row value)
            y = fromY - ahead;                              // eg 2 - 2 = 0 -> y = 0
        if (fromX + toSide <= maxX && fromX + toSide >= 0)  // eg 2 + (-1) = 1
            x = fromX + toSide;                             // eg 2 + (-1) = 1
                                                            // [1,0]
    }
    else if (Cell.PlayerFacing == "S") // looking down the map: x range maxX-1; y range 1-maxY (ahead = row+)
    {
        if (fromY + ahead <= maxY)                          // eg 2 + 2 = 4 (facing S so ahead increases row value)
            y = fromY + ahead;                              // eg 2 + 2 = 4 -> y = 4
        if (fromX - toSide >= 0 && fromX - toSide <= maxX)  // eg 2 - (-1) = 3
            x = fromX - toSide;                             // eg x = 2 - (-1) = 3
                                                            // [3,4]
    }
    else if (Cell.PlayerFacing == "E") // looking across the map to E: x range 1-maxX; y range 1-maxY (ahead = col+)
    {
        if (fromX + ahead <= maxX)                          // eg 2 + 2 = 4
            x = fromX + ahead;                              // eg 2 + 2 = 4 -> x = 4
        if (fromY + toSide <= maxY && fromY + toSide >= 0)  // eg 2 + (-1) = 1
            y = fromY + toSide;                             // eg y = 2 + (-1) = 1
                                                            // [4,1]
    }
    else if (Cell.PlayerFacing == "W") // looking across the map to W: x range maxX-1; y range 1-maxY (ahead = col-)
    {
        if (fromX - ahead >= 0)                             // eg 2 - 2 = 0
            x = fromX - ahead;                              // eg x unchanged -> 0
        if (fromY - toSide <= maxY && fromY - toSide >= 0)  // eg 2 - (-1) = 3
            y = fromY - toSide;                             // eg y = 2 - (-1) = 3
                                                            // [0,3]
    }
    return new Point(x, y);
}
```

The two functions above extract a 3x3 array of points of the cells at the side and in front of the player

Now the viewport is set, the game loop runs and the initial screen is drawn based on the viewport

<DrawScreen(Point[,] viewport)>

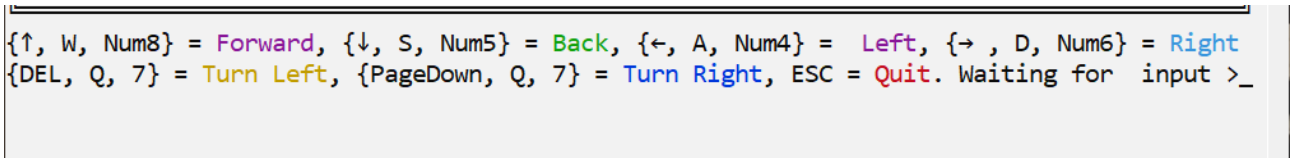
This procedure gets 4 sets of data from the cells ahead left and right of the player (items such as walls, writing, decorations, champions) and the contents of the cell the player is standing on.

This data is used to create the four panels shown earlier.

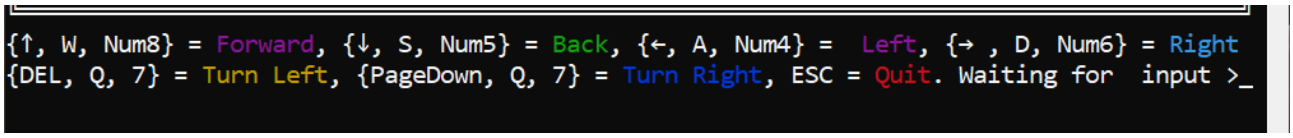
UTF8 characters are used to create the boxes along with PadLeft and PadRight for formatting. Finally an input field at the bottom, ready for the next key press.

```
lines += Instructions();

private static int Instructions()
{
    Console.ForegroundColor = Fore;
    Console.Write("{↑, W, Num8} =");
    Console.ForegroundColor = ConsoleColor.DarkMagenta;
    Console.Write(" Forward");
    Console.ForegroundColor = Fore;
    Console.Write(", {↓, S, Num5} =");
    Console.ForegroundColor = ConsoleColor.DarkGreen;
    Console.Write(" Back");
    Console.ForegroundColor = Fore;
    Console.Write(", {←, A, Num4} = ");
    Console.ForegroundColor = ConsoleColor.DarkMagenta;
    Console.Write(" Left");
    Console.ForegroundColor = Fore;
    Console.Write(", {→ , D, Num6} =");
    Console.ForegroundColor = ConsoleColor.DarkCyan;
    Console.WriteLine(" Right");
    Console.ForegroundColor = Fore;
    Console.Write("{DEL, Q, 7} =");
    Console.ForegroundColor = ConsoleColor.DarkYellow;
    Console.Write(" Turn Left");
    Console.ForegroundColor = Fore;
    Console.Write(", {PageDown, Q, 7} =");
    Console.ForegroundColor = ConsoleColor.DarkBlue;
    Console.Write(" Turn Right");
    Console.ForegroundColor = Fore;
    Console.Write(", ESC =");
    Console.ForegroundColor = ConsoleColor.DarkRed;
    Console.Write(" Quit");
    Console.ForegroundColor = Fore;
    Console.Write(". Waiting for  input >_ ");
    return 2;
}
```

This breaks the instructions into sections and changes the console colours to highlight the movements:



With reversed colour scheme:



<static class PlayerMove>

This is a **static** class with a constructor!

Static constructors run once when the class is first referenced, and is used here to populate a couple of Dictionaries of Dictionaries using ConsoleKeys as the key

The advantage of this is to keep the amount of selection code to a minimum and yet allow 3 sets of keys to be used (WASD / Arrow + DEL + Page Down / NumberPad).

When a key is pressed, it indexes one of the Dictionaries and is either returned a Point to change the current cell, or changes the Cell.Facing to suit the new direction thye player is facing.

Move() checks if any of the potential cells to move to are beyond the edge of the map. If not, the next cell is made current and the viewpoint is re-drawn.

</static class PlayerMove>

</Static void Main()>