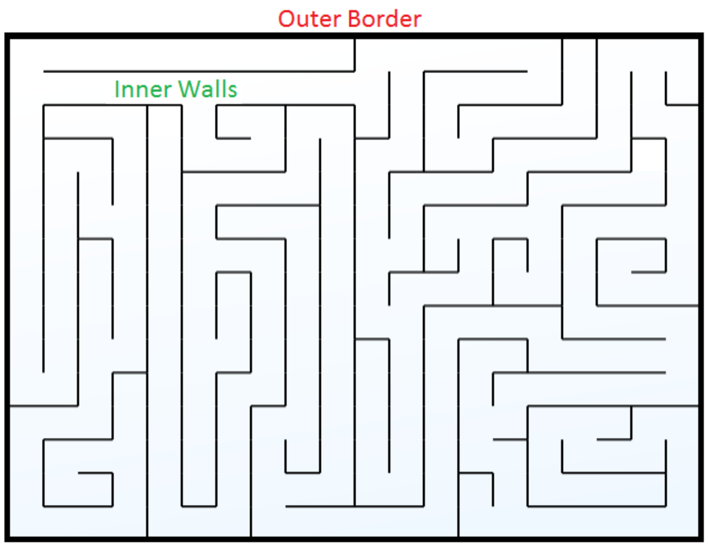
C# Programming II Final Project

Creating and Solving Perfect Mazes

For the final project you will use many of the skills you learned in this course to create and solve so-called "perfect" mazes. A maze is perfect if for any two points in the maze there is only one, unique path that connects them.

Example of a perfect maze:



Creating a Maze

To create a maze, first we must define some constraints:

- Mazes will be rectangular
- Mazes are made up of cells where the total number of cells equals the width times height
- From each cell there are four possible movement directions: up, down, left and right (north, south, west and east) – no diagonal movements
- Every cell can be visited (no unreachable cells)
- The outer borders of the maze cannot be crossed (no Pac Man-like secret passages that take you from one edge of the maze to the other)
- While traversing the maze you cannot hop over walls
- There is only one, unique path between any two cells in the maze
    - This implies that no looping is allowed
- Every cell can have zero to three walls
    - If a cell had four walls it would be inaccessible and therefore invalid
- The minimum size of a maze is 2 x 2, but for practicality we'll set the lower limit to 5 x 5

DirectionEnum & Constants Class

Teacher's Note: *For this project I created a class file called EnumsAndConstants.cs. I renamed the class in the file to "Constants" and made it static. I then added the DirectionEnum to the file above that class, but still within the namespace. In the Constants class I added the constants and readonly fields defined below to be used as "global" values throughout the project.*

As we have the need to define and use the cardinal directions of a compass, let's create an enum that will accomplish this:

```
[Flags]
public enum DirectionEnum
{
   None = 0,
   North = 1,
   South = 2,
   East = 4,
   West = 8,
   All = North | East | South | West
}
```

The values of enums are typically arbitrary, but notice that DirectionEnum has the **Flags** attribute which implies all values should be a power of 2. This also allows us to define the "*All*" enum value which is the combination of the four directions. We will use the *All* value frequently when defining and testing maze cells.

Remember, to test if a DirectionEnum value has a particular flag set, use the HasFlag method:

```
DirectionEnum test = DirectionEnum.North | DirectionEnum.East;

bool hasEast = test.HasFlag(DirectionEnum.East);  // Test for East
```

To add a flag, use the bitwise OR operator "|":

```
test = test | DirectionEnum.West;  // Add West
OR
test |= DirectionEnum.West;
```

To remove a flag, you have to use a combination of bitwise AND "&" and bitwise NOT "~":

```
test = test & ~DirectionEnum.North;  // Remove North
OR
test &= ~DirectionEnum.North;
```

Now, if you have not already done so, rename the EnumsAndConstants class to Constants, make it static, and provide the following items:

```
public static class Constants
{
  public const int MIN_WIDTH = 5;
  public const int MAX_WIDTH = 50;
  public const int MIN_HEIGHT = 5;
  public const int MAX_HEIGHT = 50;

  public static readonly DirectionEnum[] Cardinals = {
    DirectionEnum.North, DirectionEnum.South, DirectionEnum.East, DirectionEnum.West
  };

  // Additional definitions here...
}
```

EnumMethods

Extension methods are an extremely useful feature of C#. They allow you to define a method that appears to be part of a class, struct or even an enum. The trick is that the original type is not altered in any way – the extension method is completely external to the target type. This is how the Linq libraries were created. When you add the "using System.Linq;" statement to a code file, automatically, classes like List<T> and arrays get new functionality through the extension methods defined in that namespace. We will create a method that adds functionality to our DirectionEnum type. Specifically, we want a method to return the opposite direction to a given direction. You can define this class & method in the EnumsAndConstants.cs file.

```
public static class EnumMethods
{
    /// <summary>Gets the opposite direction from the one given</summary>
    /// <param name="direction">Original direction</param>
    /// <returns>Opposite direction</returns>
    /// <remarks>Input: East returns West.  Input: North, East returns South, West</remarks>
    public static DirectionEnum GetOpposite(this DirectionEnum direction)
    {
        DirectionEnum result = DirectionEnum.None;
        if (direction.HasFlag(DirectionEnum.East))
        {
            result |= DirectionEnum.West;
        }
        if (direction.HasFlag(DirectionEnum.West))
        {
            result |= DirectionEnum.East;
        }
        if (direction.HasFlag(DirectionEnum.North))
        {
            result |= DirectionEnum.South;
        }
        if (direction.HasFlag(DirectionEnum.South))
        {
            result |= DirectionEnum.North;
        }
        return result;
    }
}
```

Now, anywhere that the EnumMethods class is visible, you can get the opposite value of a given direction like this:

```
DirectionEnum north = DirectionEnum.North;
DirectionEnum opposite = dir1.GetOpposite();  // South
```

Position

When defining an individual cell in a maze (see MazeCell below) its position must be defined. In a rectangular maze like this we will define the position as a set of values, one to define the horizontal position and one for the vertical. We could simply use two ints everywhere we need to. However, since these two values tend to be used together, it makes sense to group them into a single data structure. Also, since the data consists of two relatively small value types, a struct is an excellent choice. Add a new struct to your project called "Position". Note: since Visual Studio does not have an "Add struct" menu option, simply add a class called Position and when the code file is displayed, change the definition to:

```
public struct Position
```

Next, add two fields for the position values:

```
public int X;
public int Y;
```

This is one of those rare occasions where I eschew the use of properties and create publicly exposed fields. This is due to how fields and properties work with structs as they are value types. Also, in this application we do not have a need for data validation when the fields are set. However, if you are more comfortable using the field/property structure you are welcome to use it.

Since this is a struct you cannot define your own version of the default constructor. However, we do want to define a parameterized constructor that accepts two ints, one for x and one for y. In the constructor, simply assign the input x value to the X field and do the same for Y.

```
public Position(int x, int y)
```

Next, define a ToString() override that outputs the X & Y values in standard Cartesian form. Example: (3, 4)

Next we want to create methods to allow for quick comparison of one Position object to another. To do this we will need to define the "==" and "!=" operators as well as override the Equals and GetHashCode methods. This way we will be able to compare objects using the standard (a == b) and (a != b) syntax of C#.

The "==" and "!=" overrides are relatively straight-forward:

```
public static bool operator ==(Position p1, Position p2)
{
    return (p1.X == p2.X && p1.Y == p2.Y);
}

public static bool operator !=(Position p1, Position p2)
{
    return !(p1 == p2);
}
```

Notice that both methods are static and accept two Position objects as parameters. When using the standard C# comparison syntax, such as "if (a == b)", the "a" value corresponds to the first parameter and the "b" value corresponds to the second parameter. The comparison logic checks that the X fields and Y fields for both objects are equal and returns true if they are. The "!=" operator utilizes the "==" operator so we don't have to write the comparison logic in multiple locations.

Next, define the Equals override. This is simply an instance-based version of "==" that all derivatives of object can override. By default, the Equals method returns true if the two objects being compared are the same physical object in memory. Since C# does not use pointers, it is impossible for two value types to share the same location in memory. Therefore, if we want a useful Equals method, we must do something similar to the "==" operator. In fact, we will use it.

```
public override bool Equals(object obj)
{
    if (obj is Position)
    {
        return (this == (Position)obj);
    }
    return false;
}
```

Since object's Equals method has no foreknowledge of derivatives, the input parameter must be checked for the correct type. If it is, we will compare "this" Position object with the one passed in.

GetHashCode is a method used to provide a "fingerprint" for an object. It is used mostly for hashtable and dictionary style, key-based collections. For our purposes we'll simply do an XOR (exclusive or) operation between X and Y:

```
public override int GetHashCode()
{
    return X ^ Y;
}
```

Next, we'll add some utility methods to make the Position class more useful. The first method is static and is called GetRandomPosition. Given a range of possible values for both x and y, a random Position object is generated.

```
public static Position GetRandomPosition(Range<int> horizontal, Range<int> vertical)
{
    int x = Utils.Rnd.Next(horizontal.Min, horizontal.Max + 1);
    int y = Utils.Rnd.Next(vertical.Min, vertical.Max + 1);
    return new Position(x, y);
}
```

Note that this method utilizes a Range struct which is pre-defined in the start project provided with this document.

Finally, we will add a function to move in a given direction.

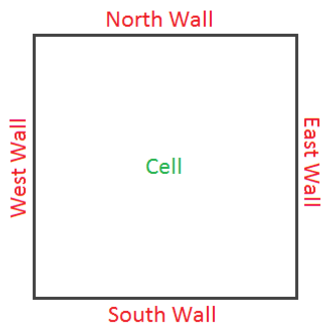
```
public Position MovePosition(DirectionEnum direction)
{
    int x = X;
    int y = Y;

    if (direction.HasFlag(DirectionEnum.North))
    {
        y--;
    }
    if (direction.HasFlag(DirectionEnum.South))
    {
        y++;
    }
    if (direction.HasFlag(DirectionEnum.East))
    {
        x++;
    }
    if (direction.HasFlag(DirectionEnum.West))
```

```
        {
            x--;
        }
        return new Position(x, y);
    }
```

MazeCell

Cells are the individual components that make up a maze. A cell can have zero to four walls in the standard cardinal directions (N, S, E, W). Although, once a maze has been created, there should never be four walls up as that would imply the cell is unreachable.

Example of a cell:



So, as we define our **public** class called *MazeCell*, we will need some mechanism to track its walls and its position within the maze. We can use DirectionEnum for the first part and Position for the second. Add two auto properties to the class:

```
    public DirectionEnum Walls { get; set; }
    public Position Position { get; set; }
```

It is OK to have the property name identical to the type as is done with Position.

Add two constructors to the MazeCell class.

Constructor 1

- accepts no parameters
- sets Position to a new Position object with X & Y set to 0
- sets Walls to DirectionEnum.All

Constructor 2

- accepts two parameters, one each for a Position object and a DirectionEnum object
- sets Position to the passed in Position object
- sets Walls to the passed in DirectionEnum object

Next, let's add a public method called *RemoveWall* that accepts a DirectionEnum parameter called *wallToRemove*. Recall how to remove flags from earlier in this document and use that logic to implement the method.

The ToString method is provided in the class already.

MazeCellArray

A maze is composed of *MazeCell* objects. We will organize these cells into a two-dimensional array. Instead of just using a *MazeCell[,]*, we will create a class that encapsulates a field of that type. Define the fields of MazeCellArray:

```
private MazeCell[,] m_Cells = null;
private Range<int>? m_XRange = null;
private Range<int>? m_YRange = null;
```

The instance field, *m_Cells*, will contain the actual array of maze cells. This arrangement is known as a "has a" relationship or encapsulation, rather than the "is a" relationship that inheritance provides.

The other instance fields, m_XRange and m_YRange, define the limits of the array. These will make determining whether a given Position is within the bounds of the maze easier.

Next, define the properties required by this class. Two of the properties are indexers that will allow us to read and write to the *m_Cells* field from outside (and inside) the class as if the class was an array itself. As indicated by the name, an indexer property is accessed by index. Indexers are named "**this**" and can have both **get** and **set** accessors. We'll create an indexer that accesses m_Cells via two int parameters and another by a Position object.

```
public MazeCell this[int x, int y]
{
    get
    {
        return m_Cells[x, y];
    }
    set
    {
        m_Cells[x, y] = value;
    }
}
```

```
public MazeCell this[Position pos]
{
    get
    {
        return m_Cells[pos.X, pos.Y];
    }
    set
    {
        m_Cells[pos.X, pos.Y] = value;
    }
}
```

If we have an object of type MazeCellArray, called *cells*, we could access the individual elements like an array:

```
MazeCell x = cells[4, 5];
```

Or

```
Position pos = new Position(4, 5);
MazeCell x = cells[pos];
```

If you wish to access this indexer from within the MazeCellArray class itself, you would use the "this" handle:

```
MazeCell x = this[4, 5];
```

Next add two public int properties with only get accessors, called *Width* and *Height*. In the body of each property's get accessor, check that *m_Cells* is not null. If not null, Width returns the width, or first dimension length, of m_Cells. Height does the same thing but returns the second dimension length. Since *m_Cells* is an array, you can use its *GetLength* method passing either 0 or 1 for width and height, respectively. If *m_Cells* is null, then return 0 for both properties.

The final two properties are used to return the Range objects m_XRange and m_YRange, respectively.

```
public Range<int> XRange
{
    get
    {
        if (m_XRange == null)
        {
            m_XRange = new Range<int>(0, Width - 1);
        }
        return m_XRange.Value;
    }
}

public Range<int> YRange
{
    get
    {
        if (m_YRange == null)
        {
            m_YRange = new Range<int>(0, Height - 1);
        }
        return m_XRange.Value;
    }
}
```

Now, let's move onto some of the methods of the class…

Reset()

Description: Resets all cells in the *m_Cells* array to new MazeCell objects
Signature: `public void Reset()`

The first method is called *Reset* which loops through all indices of *m_Cells* and sets each value to a new MazeCell object. Since *m_Cells* is a two-dimensional array, using a nested for loop is the easiest way to access each item. The outer for loop would cycle through all numbers between 0 and Width - 1 and the inner loop would do the same between 0 and Height - 1. Inside the inner loop you will create a new MazeCell and put that new cell in the appropriate pair of indices indicated by the for loop variables. Those for loop variables can also be used to set the new MazeCell's Position property. As for the Walls property of each new MazeCell, set that to DirectionEnum.All.

Resize()

Description: Resizes the *m_Cells* array to new width and height dimensions
Signature: `public void Resize(int width, int height)`

Resize is the method that the constructor and any consumer of the MazeCellArray class will use to set the dimensions of the maze.

The first thing the method must do is verify that width and height are legal. You can use the constants defined in the Constants class for legal values.

Next, set m_XRange and m_YRange to null. The next time their encapsulating properties (XRange and YRange) are accessed they will be re-created automatically. Check the property definitions above to see how this is done.

Next, *m_Cells* must be re-created with the following statement:

```
m_Cells = new MazeCell[width, height];
```

Finally, call Reset() to initialize all of the cells in *m_Cells*.

MazeCellArray Constructor

Description: Creates a new MazeCellArray object

Signature: `public MazeCellArray(int width, int height)`

All that happens in the constructor for this class is a call to Resize() passing through width and height.

GetNeighborCellsWithAllWalls()

Description: From a given cell position, get all of the adjacent cells from the four compass directions that have all walls up

Signature: `public Dictionary<DirectionEnum, MazeCell> GetNeighborCellsWithAllWalls(Position position)`

Requires: `using System.Collections.Generic;`

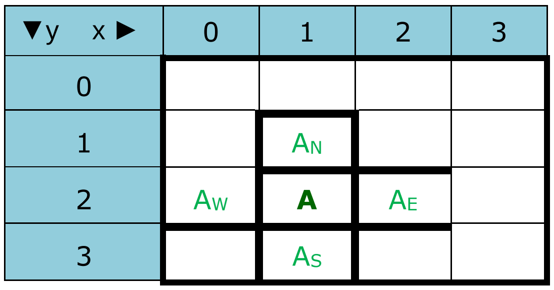
This method will accept a Position object called *position* and returns a Dictionary<DirectionEnum, MazeCell> object.

First, this method will create a Dictionary<DirectionEnum, MazeCell> called *neighbors*.

Next, using a foreach loop, cycle through each of the four cardinal directions. The Constants class contains an array of DirectionEnum values, called *Cardinals*, for this purpose. Inside the loop, you must determine the Position of the cell in the direction of the current loop step. **The Position class has a *MovePosition* method which, when passed a DirectionEnum value, returns a new Position at the given direction.** Using this new position, ensure it is within the ranges defined within XRange and YRange (note: Range objects have a *Contains* method). If the new position is in range, then retrieve the MazeCell at that position using the indexer that accepts a Position. If the MazeCell object's Walls property equals DirectionEnum.All, then add it to *neighbors* using the current direction as the key.

After the loop, return *neighbors*.

Consider the following maze array with 4 columns and 4 rows:



If A (position: (1, 2)) is the input only A_N and A_S would be returned. This is because A_W is missing its North wall and A_E is missing its East wall.

Note: A **Dictionary** is used so the caller can quickly check to see what directions has valid neighbors using the *ContainsKey* method of the return value.

Pseudo code:

```
1. Create a Dictionary<DirectionEnum, MazeCell> called neighbors
2. Loop through each DirectionEnum (dir) in Constants.Cardinals
    a. Create a new Position (neighbor) in the direction of dir from position
    b. If XRange contains neighbor.X AND YRange contains neighbor.Y AND the MazeCell at
       position neighbor has its Walls property set to DirectionEnum.All
        i. Add the MazeCell to neighbors using dir as the key
3. Return neighbors
```

GetAccessibleNeighborCells()

Description:	From a given cell position, get all of the adjacent cells that can be reached
Signature:	`public Dictionary<DirectionEnum, MazeCell> GetAccessibleNeighborCells(Position position, DirectionEnum exclude = DirectionEnum.None)`
Requires:	`using System.Collections.Generic;`

This method accepts a Position object, called *position*, and returns a Dictionary<DirectionEnum, MazeCell>. This method performs similarly to *GetNeighborCells* except it returns a list of cells that can be reached from the given cell position. Once again, create a new Dictionary<DirectionEnum, MazeCell> called *neighbors*. Then loop through each DirectionEnum in Constants.Cardinals. If the exclude parameter matches the current direction, skip to the next one as the caller specifically requested not to include any cells in that direction. Otherwise, check to make sure there is no wall in the current direction. A statement such as:

```
this[position].Walls.HasFlag(dir)
```

will test for a wall in the "*dir*" direction. If there is no wall in a given direction, get the cell in that direction and add it to neighbors (just like was done in GetNeighborCellsWithAllWalls).

After the loop, return *neighbors*.

Pseudo code:

```
1. Create a Dictionary<DirectionEnum, MazeCell> called neighbors
2. Loop through each DirectionEnum (dir) in Constants.Cardinals
    a. If dir does not equal exclude AND the wall in direction dir of the cell at position is not
       present
        i. Create a new Position (neighbor) in the direction of dir from position
        ii. Add the MazeCell to neighbors using dir as the key
3. Return neighbors
```

RemoveCellWall()

Description: Remove the wall of the cell in the given position in the provided direction. Also remove the adjacent cell's opposite wall.

Signature: `public void RemoveCellWall(Position position, DirectionEnum wallToRemove)`

The purpose of this method is to provide a simple mechanism for removing a wall in a given direction from a cell. The code to remove a cell wall is already defined within the MazeCell class. This method is provided to remove the given cell's indicated wall and the neighbor cell's opposing wall. So, if you need to remove a cell's west wall you also need to remove the west cell's east wall. This code is provided as it can be a tricky task to put all of the calls together:

```
public void RemoveCellWall(Position position, DirectionEnum wallToRemove)
{
    // Find adjacent cell in the direction of the wall to remove
    MazeCell cell = this[position];
    Position neighborPosition = cell.Position.MovePosition(wallToRemove);

    // Only remove walls if neighborPosition is within bounds,
    // otherwise the given cell is on the border and walls cannot come down.
    if (XRange.Contains(neighborPosition.X) && YRange.Contains(neighborPosition.Y))
    {
        cell.RemoveWall(wallToRemove);
        MazeCell neighbor = this[neighborPosition];
        neighbor.RemoveWall(wallToRemove.GetOpposite());
    }
}
```

Maze

Now, onto the fun part – making and solving a perfect maze. Remember, a perfect maze is a maze where for any two points there is only one, unique path between them.

The Maze class will contain both create and solve algorithms.

Class Setup (for maze creation)

Fields

MazeCellArray	*m_Cells*	The array data structure that contains the maze cells
bool	*m_MazeCreated*	Flag that indicates if the maze was successfully created

Properties

Indexer to access *m_Cells* using a pair of int values (x & y)
Indexer to access *m_Cells* using a Position object
Width property to return the width of *m_Cells* (get only)
Height property to return the height of *m_Cells* (get only)
StartPoint property of type Position (auto property) – Indicates starting position in maze
EndPoint property of type Position (auto property) – Indicates ending position in maze
Complexity property of type double (auto property) – Indicates complexity of the maze
Values: 0 to 1, where 0 = easy, 1 = hard and 0.5 = medium

Methods

Resize()

Description: Resizes the *m_Cells* object
Signature: `public void Resize(int width, int height)`

This method first validates width and height are valid values.
Next, *m_Cells* is assigned a new MazeCellArray, passing *width* and *height* as parameters.
Finally, *m_MazeCreated* is set to false.

Constructor 1

Description: Creates a new Maze object
Signature: `public Maze()`

This constructor calls Resize() using the minimum width and height constants defined in Constants.

Constructor 2

Description: Creates a new Maze object

Signature: `public Maze(int width, int height)`

This constructor calls Resize() using the given width and height values.

Maze Making Algorithm

There are many algorithms that will do the job, but we'll use a variation of the depth-first search algorithm. The variations we'll add is that while creating the maze we'll use random values, based on the input complexity, to determine how long "hallways" will be. A simple maze will have longer hallways than a more complex maze, which will have many twists and turns.

In plain language, the algorithm goes like this:

- Start at a random cell in the maze.
- Pick a random direction to go such that the destination cell has all of its walls up.
- Knock down that wall and move to the cell.
- If you reach a dead-end, then backtrack until you find a cell with a neighbor that has all of its walls up and go there.
- Continue this process until all cells have been processed.

Assumptions

- We have a property called *Complexity* that is a double value between 0 and 1
- We have a *MazeCellArray* field called *m_Cells* – where all walls are up (*DirectionEnum.All*)

CreateMaze()

Description: This method is the entry point for the maze creation algorithm. The majority of the actual maze creation will occur in ProcessCell().

Signature: `public void CreateMaze(double complexity)`

```
Complexity = complexity;
m_Cells.Reset();

// Create starting and ending point (must be distinct)
StartPoint = Position.GetRandomPosition(m_Cells.XRange, m_Cells.YRange);
do
{   // Ending point must be distinct
    EndPoint = Position.GetRandomPosition(m_Cells.XRange, m_Cells.YRange);
} while (EndPoint == StartPoint);

ProcessCell(m_Cells[StartPoint], DirectionEnum.None);
m_MazeCreated = true;
```

ProcessCell()

Description: This method processes a MazeCell and recursively calls itself to process additional cells.

Signature: `private void ProcessCell(MazeCell cell, DirectionEnum heading)`

Algorithm:

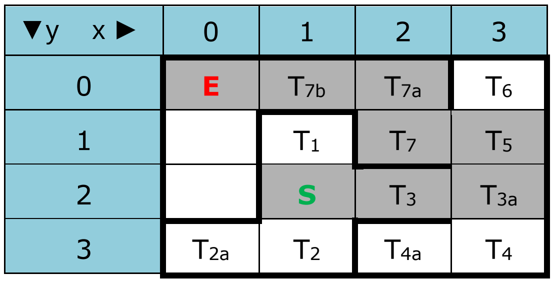
1. Use *m_Cells*' GetNeighborCellsWithAllWalls method, passing in *cell*'s *Position* property, to get a list of all neighbor cells that have not yet been processed, called *neighbors*
   a. If no valid neighbors exist then exit the method without any further action
2. Create a new DirectionEnum called *newHeading*
3. Create a new double called *chance* and set it equal to a new random value between 0 and 1
   a. You can use Utils.Rnd.NextDouble() to get a new random value
4. If *chance* is greater than Complexity **AND** *neighbors* contains a key that matches *heading* then set *newHeading* equal to *heading*
5. While *neighbors* has items do the following
   a. If *newHeading* equals DirectionEnum.None select a random DirectionEnum from the keys in *neighbors* and assign it to *newHeading*
      i. Note: the extension method in Utils called GetRandomValue provides the ability to get a random value from any collection, so you can call GetRandomValue on *neighbor*'s Key collection
   b. Retrieve the MazeCell from *neighbors* that has the *newHeading* key and store it in a MazeCell object called *neighbor*
   c. If *neighbor*'s Walls property equals DirectionEnum.All
      i. Remove *neighbor*'s wall in the direction of *newHeading*
         1. *m_Cells* contains a RemoveCellWall method which you can pass *cell*'s Position property and newHeading
      ii. Recursively call ProcessCell, using *neighbor* and *newHeading* as the parameters
   d. Remove the MazeCell from *neighbors* with the key *newHeading*
   e. Set *newHeading* equal to DirectionEnum.None

Maze Solving Algorithm

Next, we will define the algorithm to solve a maze. Like the maze creation process, there are many different algorithms to solve such mazes. We will use threads to assist us with this process. Basically, at any given cell within the maze you have three possibilities:

1. There are no valid paths to traverse from this position which implies a dead-end.
2. There is only one path to take, continue in that direction.
3. There are multiple paths that can be taken. Spawn a thread for each path that repeats this process.

Note: Care must be taken not to backtrack over cells that have already been visited.



In the above picture, the maze starts at position (1, 2), labeled S.

A thread is spawned for each valid direction from that cell, which in this case is North, South and East (order is arbitrary), labeled T_1, T_2 and T_3, respectively.

T_1 is a dead-end so the thread terminates.

T_2 can only proceed to the west and terminates in a dead-end.

T_3 continues east to T_{3a}. Here, it has two possible directions (N and S) and spawns two threads to go in those directions (T_4 and T_5).

T_4 has only one possible direction and dead ends at cell (2, 3).

T_5 reaches a junction at cell (3, 1) and spawns two new threads (T_6 and T_7).

T_6 is a dead-end so the thread terminates.

T_7 has only one possible path and follows it through cells (2, 1), (2, 0), (1, 0) and finally (0, 0) where the end of the maze is reached.

Note: At the time T_7 reaches E, it signals all other threads to quit as a solution has been found. Therefore, it is unlikely that cells (0, 1) and (0, 2) will ever be touched.

The solution path taken is shown in gray.

Since each thread is short-lived (exists long enough to check if it is at the end point or spawn new threads) we will use Tasks. This will also prevent overwhelming the system with possibly hundreds of managed threads.

Class Setup (for maze solving)

Fields

ManualResetEvent	*m_Done*	Manual reset event that is triggered when the solution is found to tell all tasks to stop processing

Properties

Solution property of type List<DirectionEnum> (auto property) – Contains the list of directions taken to reach the end point from the start point

SolveMaze()

Description: This method is the entry point for the maze solving algorithm. The majority of the actual maze solving will occur in SolveMazePath().

Signature: `public bool SolveMaze()`

Requires: `using System.Threading;`

```
if (!m_MazeCreated) throw new Exception("Maze not created!");

MazeCell start = m_Cells[StartPoint];
m_Done.Reset();
SolveMazePath(start, DirectionEnum.None, new List<DirectionEnum>());

if (!m_Done.WaitOne(30000))
{
    // Taking too long - something bad happened
    m_Done.Set();
    throw new TimeoutException("Maze solving taking too long.");
}
return true;
```

SolveMazePath()

Description: This method attempts to find the end point following the heading provided. If paths to either side are present, spawn threads to follow those paths.

Signature: `private void SolveMazePath(MazeCell cell, DirectionEnum heading, List<DirectionEnum> directions)`

Requires: `using System.Threading;`
`using System.Threading.Tasks;`

Algorithm:

1. Check if *m_Done* has been triggered using *m_Done*'s WaitOne() with a single tick. If WaitOne returns true, simply return ignoring any further processing – the threads have been asked to stop.
2. Check if *directions*' length is greater than the number of moves possible to solve the maze (width x height). If it is, call Set() on *m_Done* and throw an exception.

3. If *heading* is not DirectionEnum.None, add it to *directions*. This will keep track of how we got to where we are. This list will also be passed to any new threads down the line.
4. Check if *cell*'s Position property equals *EndPoint*. If it does, we have found the solution. Set *Solution* equal to *directions*, call Set() on *m_Done* and return from the method.

If step 5 is reached that means the cell must be processed further

5. Retrieve all accessible neighbor cells and store in a variable called *neighbors*
    a. Check the method called GetAccessibleNeighborCells of *m_Cells*. It requires the current *cell*'s position and any direction to ignore. Since we don't want to reprocess the cell we just came from, use that as the *exclude* parameter. ***Remember, we need the direction opposite to heading – that's what the GetOpposite() extension method is for.***
6. Process neighbors:
    a. If *neighbors* is empty, return as we've hit a dead end
    b. Else if neighbors contains more than one element, spawn a new task for each direction
        i. Use a foreach loop to cycle through each DirectionEnum in *neighbors*' Keys collection (assume *direction* is the loop variable)
            1. Spawn a task to go down the direction indicated by the current loop item. As this can be tricky if you are new to threading, here is the command to spawn the new Task:

```
Task.Run( () =>
{
    SolveMazePath(neighbors[direction], direction,
        new List<DirectionEnum>(directions));
});
```

Notice that the Task runs SolveMazePath. The first parameter is the cell at "direction" *heading from the current position. The heading of that cell is the second parameter. The third parameter is a clone of the* directions *list. This list needs to be cloned as we don't want separate threads accessing the same data. That would lead to unknown results and possible data corruption.*

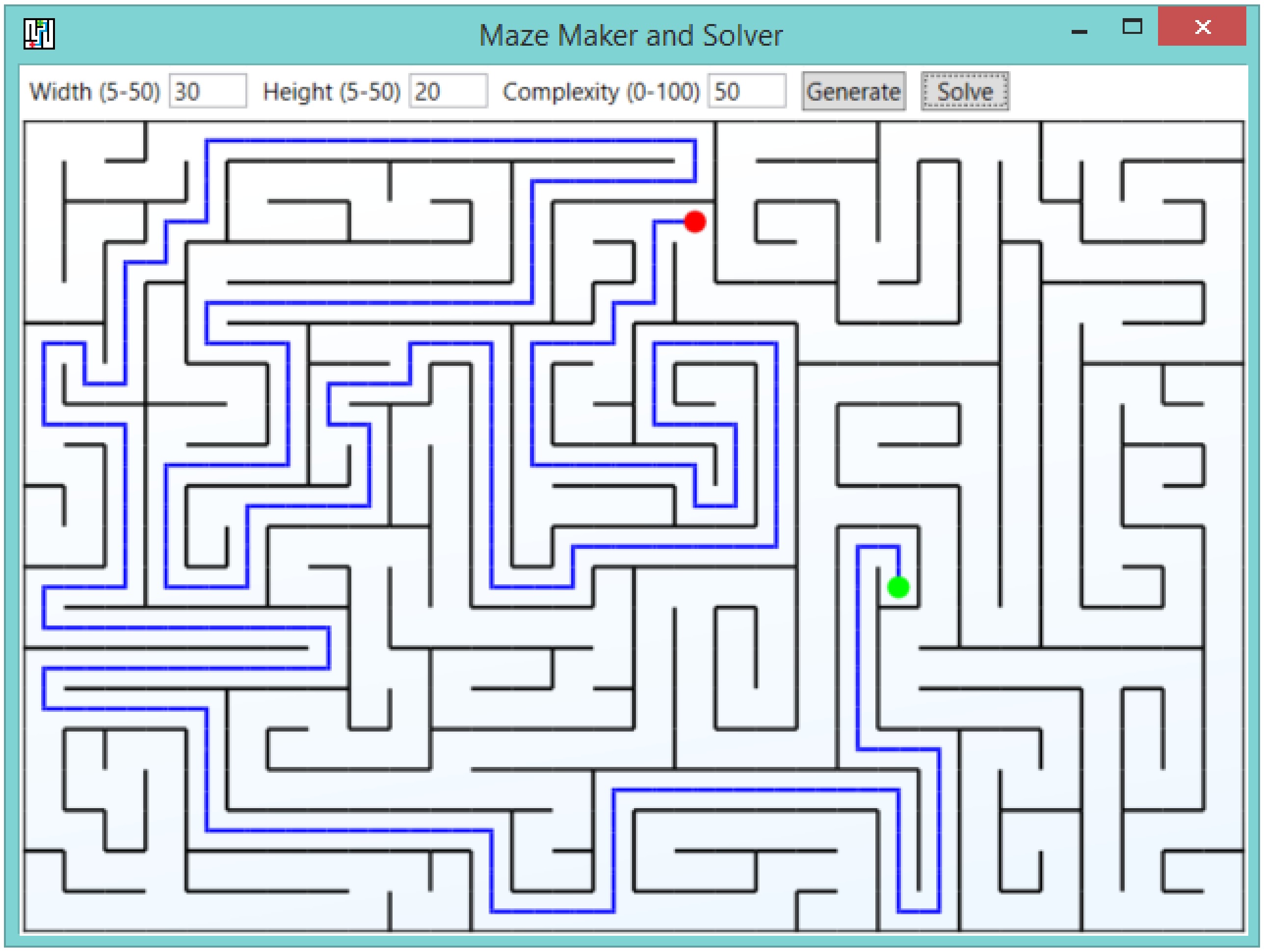
    c. Else, *neighbors* contains only one element. Instead of spawning another task to go down that path, use this existing thread by calling SolveMazePath recursively. This will minimize the number of threads that need to be created, thus optimizing the code a bit.

```
SolveMazePath(neighbors.Values.First(), neighbors.Keys.First(), directions);
```

Notice that the directions *parameter is not cloned as it was when tasks were spawned. This is because we are calling the method recursively* ***on the same thread*** *and there is no chance of another thread corrupting the data. For consistency, you could clone the list, but it will slow down the algorithm creating a new list of values.*

MainWindow

The WPF window, MainWindow, is the driver of the algorithms. From here you will fire off the code to create and solve mazes. Included in this class are methods to help visualize mazes and their solutions. Please check the code for more details.



Good Luck!