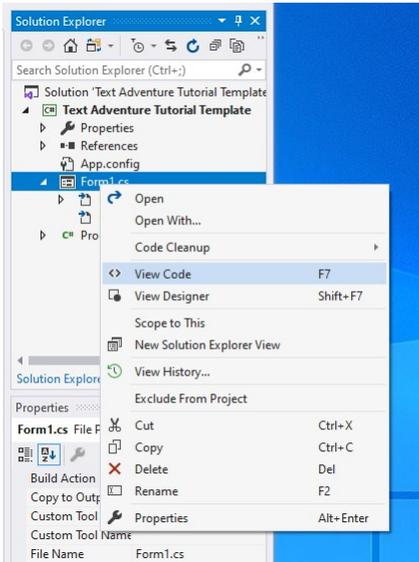


Text Adventure Code

This is a continuation of Part 1 which described how to setup the Form and it's controls.

Although there is a difference between Functions and Procedures needed to get extra exam marks, I will be using the word Function to refer to both! This is partly due to extensive use of Lua, which uses the keyword "function" for both.



Right-Click on Form1 and select 'View Code'

Modify the code already present to look like this: (Highlighted text is not already present and has been added) Delete any comments / Summaries. The form might be called "MainForm" or something else if you renamed it.

```
using System;
using System.Windows.Forms;

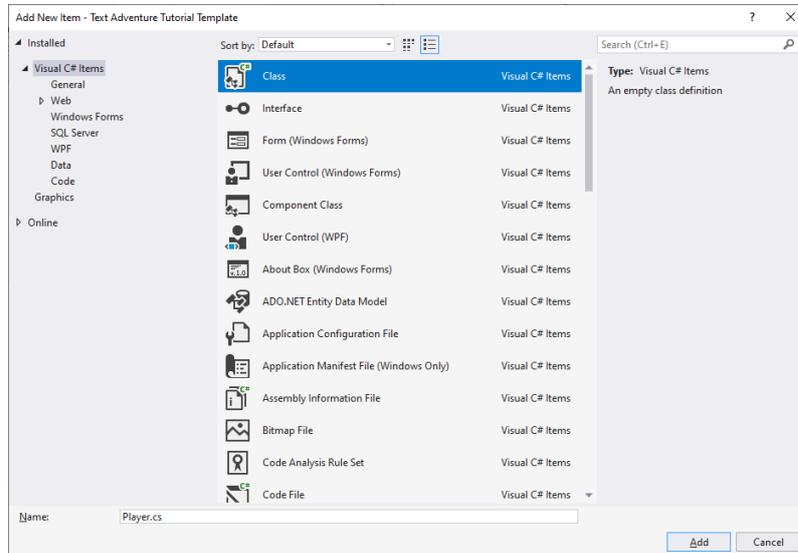
namespace <Your Namespace> //Depends what you called your project
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            this.CenterToScreen();
        }
    }
}
```

Test it out: Click the Run button

The form should open in the centre of the screen, displaying all the controls you added. Nothing will happen yet, as you have not coded it. Close the Window.

Time to get back to the idea of 'Classes'

- 1) Right-Click on the 'Text Adventure Tutorial' and select Add → New Item



- 2) Select 'Class'
- 3) Change the name from Class 1.cs to Player.cs
- 4) Click 'Add'

You should now have a new code panel:

```
namespace <Your Namespace>
{
    class Player
    {
    }
}
```

Modify it so it looks like this:

```
{
    static class Player
    {
    }
}
```

A brief description of classes:

A class can be thought of as a way of defining a new multi-purpose variable.

In the Player class created above, code can be added to hold other variables, such as Name, Strength, Health.

As we only need one Player, the class can be declared as static, so we can just use it with the dot notation, e.g. to write the player name to the Console, just use

```
Player.Name = "Fred";  
Console.WriteLine(Player.Name);
```

On the other hand, if we create an 'Item' class, there will be a number of different items needed, so each one has to be created as an instance of the class (an object) by using the keyword 'new':

(Examples only. These lines not needed in the game code)

```
Item item1 = new Item();  
item1.Name = "Torch";  
  
Item item2 = new Item();  
item2.Name = "Rock";
```

To differentiate between the two ways of using a class, the keyword 'static' is used to mean a class that cannot have instances made from it.

The same technique is used in Java.

Python's equivalent is a 'Code Module' which has to be imported into the main script

Time to add some variables to the class: (Note the variables have to also be declared as static)

```
using System;  
  
namespace Text_Adventure_Tutorial  
{  
    static class Player  
    {  
        public static string Name;  
        public static string Character;  
    }  
}
```

These variables are 'Properties' and can be directly read or written to from outside of the class if declared as public .

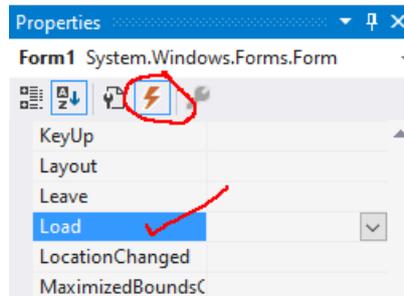
It is considered bad practice to use public variables directly, even in a static class, so "getters and setters" can be used.

These have recently been renamed to some obscure term such as "accessors" or similar. This knowledge is only required for extra marks in an exam... Look it up for yourself.

Time to test the new class

Go back to the Form1.cs [Design] window.

- 1) Click on an empty part of the Form.
- 2) Click on the Lightning bolt icon in the Properties Window
- 3) Double-Click the 'Load' event



This will automatically create some new code in the Form1.cs Tab:

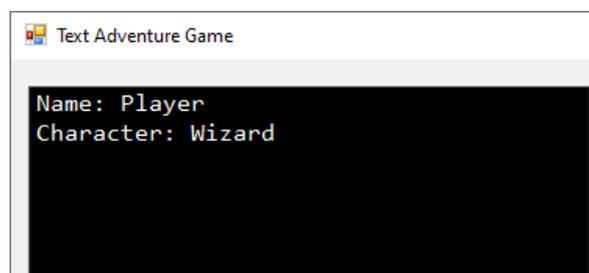
```
private void Form1_Load(object sender, EventArgs e)
{
}
}
```

Add the following 2 (highlighted) lines:

```
private void Form1_Load(object sender, EventArgs e)
{
    Player.Name = "Player";
    Player.Character = "Wizard";
    lblGame.Text = "Name: " + Player.Name;
    lblGame.Text += "\nCharacter: " + Player.Character;
}
}
```

Run the program again

As it starts up, the Form Load event is called, which runs the code in the highlighted lines above, giving the Player the name of "Player" and printing it to the Game output window:



Note how the text is built up by adding new text to the text already present.

Create the Item class

Create a new class called Item.cs using the same method as before (Page 2)

This one will be used numerous times to create all the Item objects needed in the game.

This class will **not** be using the static keyword.

To help create instances of this class, it is easier to use a 'Constructor'.

```
namespace Text_Adventure_Tutorial
{
    class Item
    {
        //constructor
        public Item()
        {

        }
    }
}
```

The Constructor has the same name as the class, and can take values fed to it when it is created:

```
public Item(string name, string description)
```

These values can be used to run code in the constructor. In this case they are used to set the values of the Name and Description fields.

Add the above values to the Constructor, and the lines of code shown below:

```
namespace Text_Adventure_Tutorial
{
    public class Item
    {
        public string Name { get; set; }
        public string Description { get; set; }
        //constructor
        public Item(string name, string description)
        {
            Name = name;
            Description = description;
        }
    }
}
```

Note the use of upper and lower case letters. These are **different** variables, so the temporary ones used in the constructor are passed to the Property fields (variables).

The Name and Description are Auto-implemented properties. They are the equivalent of the Python:

```
@property
def name():
    return self._name

@name.setter
def name(self, value):
    self._name = value
```

How does all this class / object /constructor stuff work?

Think of a class as a template. As an example, when the game starts you may need a torch and a key as two essential items. In the `Form1_Load()` event add some lines of code to create these items:

To make a torch use the following code:

```
Item torch1 = new Item("torch", "a flaming wooden torch");
```

To make a key use:

```
Item key1 = new Item("iron key", "a rusty old key.");
```

When these lines of code are used an analogy of Discworld's 'imps in the machine' carrying out instructions can be useful:

The imp in the `Form1_Load()` department sees the instruction '`Item torch`' and thinks:

- OK I have to make a torch. Luckily I have a template called 'Item' which I can use to make it.
- The template (class) expects two variables: a string called name, and a string called description. The finished Item object is called 'torch1'
- The instructions the coder has given me are: Make me a torch variable (object) called 'torch1'. the name is 'torch' and the description is 'a flaming wooden torch'
- He sends a text message to his friend in the 'Items' class with these two pieces of information.

The imp in the Items class receives a text message on his iConstructor ® © device:

```
public Item(string name, string description)
```

which is automatically translated to:

```
public Item(name = "torch", description = "a flaming wooden torch")
```

You can see this in action if the program is stepped through. All he has to do is to take this information and make an 'Item' object in memory and set it's 2 properties accordingly:

```
Name = name;  
Description = description;
```

A new Item (object) called 'torch1' has been created. The same procedure is used to create the key1 object.

If the coder wants to find out the properties of his new torch1 object he can use the 'getter':

(Pseudocode)

```
Print(torch1.Name) → public string Name {get; set;} → 'torch'
```

Creating the player's inventory (backpack, shopping bag, etc.)

The player needs to hold a number of items in his/her inventory.

An inventory could be a list of items, but a Dictionary works better, as the index is the name of the item, not a random number

You need to create a Dictionary in the Player class, to keep stock of all the items he/she is carrying.

Go back to the Player class and add this line:

```
public static Dictionary<string, Item> Inventory;
```

This reserves memory space for a Dictionary of Items, called Inventory.

The Player class now looks like this:

```
using System.Collections.Generic;
```

```
namespace Text_Adventure_Tutorial
{
    static class Player
    {
        public static string Name;
        public static string Character;
        public static Dictionary<string, Item> Inventory = new Dictionary<string, Item>();
    }
}
```

So what is a 'Dictionary' and how does it work?

A dictionary is similar to a list, but uses strings to index it:

| List Index (integer) | Dictionary Index (string) | Item |
|----------------------|---------------------------|--|
| 0 | torch | {Name = "torch"; Description = "a flaming wooden torch"} |
| 1 | key | {Name = "key"; Description = "a large iron key"} |
| 2 | Key card | {Name = "key card"; Description = "a plastic card"} |

- You can hold any of the basic variable types in a dictionary, and any objects, both in-built and created by you. In the case of 'Items' you can keep all the items you create in a dictionary of items.
- You can .Add items to the dictionary
- You can .Remove an item from the dictionary.
- The items in the dictionary are not ordered.

Dictionaries have to be initialised before you can use them. The best place to do this is in the constructor (dynamic classes)

Static classes have an implicit constructor, that is used when the first reference to it is made, but you can specifically add one in the same way as a normal class, and carry out any initialisation there on first use.

The Player class is now ready for use, and can be expanded if required to hold more data, such as age, strength, health etc.

There will be Methods added to it later

REMINDER:

1. The Player Class is static.
2. You cannot make Player objects.
3. You can refer to its variables directly using `Player.<variable name>`
4. Once created, it remains in memory while the Program is active.
5. The Item Class is used to create Item Objects.
6. You can have as many Item objects as you need, each one has a different name (torch1, torch2, key1, treasure etc.)
7. Items can be stored in Lists and Dictionaries
8. Items can be deleted. The memory they used is reclaimed.

Preparing for the next steps:

The code you used to test the game earlier was put into the `MainFormLoad()` event for testing purposes, but this is not the best place to put it. It is better to make a new function specifically to create the game. An obvious name for this function is `CreateGame()`. Delete the lines in the `MainFormLoad` Event

Type this new procedure:

```
private void CreateGame()
{
    //Create the player

    Player.Name = "Inksaver"; // put your name here or write a starting form to get details?
    Player.Health = 100; // not used so far. Enemies are needed!
    Player.Strength = 50; // not used so far. Enemies are needed!
}
```

To use this new function when the Form Loads, add the following line to the now empty

`void MainFormLoad()` event and **move** the `{}` brackets to the same line (Do NOT add new ones!):

```
void MainFormLoad(object sender, EventArgs e) {CreateGame();}
```

Note the whole of this event has been compacted to use just a single line. This has no effect on the way the code runs, it just takes up less space.

So far, the game has a Player, and Items, but nowhere to play.

The next class is used to create the world the game runs in.

Make a new class called Location.cs

Each Location represents an area of the game the player can visit.

These could be traditional dungeons, or a more modern twist, using classrooms in a school, or even a hotel.

This tutorial uses a run-down hotel, where the player wakes up in a hotel room, and has no idea where he is, or how he got there. Getting out would be good.

Draw a plan on paper so you can refer to it as you build your game. The hotel has 11 locations, and each location has a name

world plan:

```
|---|---|---|---|
| 0  1  2 | 9 |
|---|- -|---|  |
|   | 3 |||||  |
|---|- -|---| 10
| 5  4 |||||  |
|---|- -|---|  |
|   | 6  7  |
|---|- -|---|
|   | 8 |
|---|
```

0 = bedroom
1 = coridoor
2 = lift
3 = hallway
4 = diner
5 = store
6 = bar
7 = games
8 = wc
9 = reception
10 = home

Doors are shown as gaps between each room.

With North at the top of the map, you can see the first Location is a bedroom (0), with a single door to the East, leading to coridoor (1), east to lift(2) etc.

The Location class needs the following Properties:

- Name: the same short name used in the plan above, used as an identifier (NO DUPLICATES)
- DisplayName: A name that looks better when used in the text. E.g "You are in the games lounge" reads better than "You are in games"
- Description: The better you can describe each location, the more atmosphere your game has. E.g. "a flickering light illuminates the moth-eaten sofas"
- LocationToNorth: Empty string ("") if no exit, otherwise the short .Name of the Location
- LocationToEast
- LocationToSouth
- LocationToWest
- List of Items found in that location or empty list if no items present

The complete Location class:

```
using System.Collections.Generic;
namespace Text_Adventure_Tutorial
{
    public class Location
    {
        /// <summary>
        /// This class is used to create the locations in the game world
        /// It has a dictionary of Items in that location which can be
        /// added in the constructor, or as the player drops items as he/she goes along.
        ///
        /// There are string LocationToXXX directions which determine whether there is an
        /// exit in that direction. The string is the dictionary key to another Location
        /// </summary>

        #region Public Properties
        public string Name { get; set; }
        public string DisplayName { get; set; }
        public string Description { get; set; }
        public string LocationToNorth { get; set; }
        public string LocationToEast { get; set; }
        public string LocationToSouth { get; set; }
        public string LocationToWest { get; set; }
        public Dictionary<string, Item> Items { get; set; } = new Dictionary<string, Item>();
        public Item ItemRequired { get; set; }
        #endregion
        #region Constructor
        public Location(string name,
                       string displayname,
                       string description,
                       string locationToNorth = "",
                       string locationToEast = "",
                       string locationToSouth = "",
                       string locationToWest = "",
                       List<Item> listitem = null,
                       Item itemRequired = null)
        {
            Name = name;
            DisplayName = displayname;
            Description = description;
            LocationToNorth = locationToNorth;
            LocationToEast = locationToEast;
            LocationToSouth = locationToSouth;
            LocationToWest = locationToWest;
            if (listitem != null)
                CreateDictionary(listitem);
            ItemRequired = itemRequired;
        }
        #endregion
        #region Methods
        private void CreateDictionary(List<Item> itemList)
        {
            /// Used from the constructor to convert a list of Items into a Dictionary
            foreach(Item item in itemList)
            {
                Items.Add(item.Name, item);
            }
        }
        #endregion
    }
}
```

This is a fairly complex class. It currently holds data about it's name, description, the names of surrounding locations, any Items found in the Location.

Note in the Constructor parameters, default values are given to the last 6. This is to make sure that if no value is passed, the default is used.

All this should be familiar to you now except the lines:

```
if (listitem != null)
    CreateDictionary(listitem);
```

If the parameter listitem contains a list of Items, these are added to the dictionary 'Items' using the private method CreateDictionary()

So we now have a Player static class, an Item class for generic items such as rocks, torches etc, and a Location class for moving around the game.

The next thing needed are some specialised Item classes:

- Weapons: for attacking Enemies
- Potions: For healing (or poisoning!) the Player
- Notes: For giving the Player information and clues

These can be based on the Item class, and inherit the .Name and .Description properties.

Create 3 new classes called Weapon, Potion and Note: They all inherit from Item.

```
namespace Text_Adventure_Tutorial
{
    internal class Weapon : Item
    {
        /// <summary>
        /// This class inherits from the Item class
        /// so gets the Name and Description properties from
        /// the base class
        /// It can be used to inflict damage on Items or Enemies
        /// </summary>

        public int Damage { get; set; }
        public Weapon(string name, string description, int damage) : base(name, description)
        {
            Damage = damage;
        }
    }
}
```

```

namespace Text_Adventure_Tutorial
{
    internal class Potion : Item
    {
        /// <summary>
        /// This class inherits from the Item class
        /// so gets the Name and Description properties from
        /// the base class
        /// It can be used to allow the player to receive a healing potion (or poison if Healing
is negative!)
        /// </summary>
        public int Healing { get; set; }
        public Potion(string name, string description, int healing) : base(name, description)
        {
            Healing = healing;
        }
    }
}

```

```

namespace Text_Adventure_Tutorial
{
    internal class Note : Item
    {
        /// <summary>
        /// This class inherits from the Item class
        /// so gets the Name and Description properties from
        /// the base class
        /// It can be used to allow the player to read notices, maps, clues etc.
        /// </summary>

        public string Message { get; set; }
        public Note(string name, string description, string message) : base(name, description)
        {
            Message = message;
        }
    }
}

```

You can see each class has a speciality Property, Damage in the Weapon, Healing in the Potion and Message in the Note.

So now we need to complete game setup in the CreateGame function, using these classes and the dictionaries for storing them in.

The best place to store all the game objects is in Dictionaries of the object type

Add these lines to the MainForm Code:

```
public partial class MainForm : Form
{
    #region Class variables
    private Location currentLocation;
    private string NorthExit;
    private string EastExit;
    private string SouthExit;
    private string WestExit;
    // Dictionary of all locations in the game
    private Dictionary<string, Location> dictLocation = new Dictionary<string, Location>();
    // Dictionary of all Items in the game
    private Dictionary<string,Item> dictItems = new Dictionary<string,Item>();
    #endregion
}
```

Add these lines to the CreateGame() function;

```
private void CreateGame()
{
    // create 4 Item objects, a Weapon, a Potion and a Note object

    dictItems.Add("torch", new Item("torch", "a flaming wooden torch. It probably should not be in this
        adventure.."));
    dictItems.Add("LED torch",new Item("LED torch", "a new-fangled battery powered monstrosity"));
    dictItems.Add("iron key",new Item("iron key", "a rusty old key suitable for a dungeon door"));
    dictItems.Add("hotel card", new Item("hotel card", "a plastic key card with magnetic strip. Probably not
        much use in a dungeon"));
    // add a Weapon to the same Dictionary, as Weapon is a sub-class of Item
    dictItems.Add("knife", new Weapon("knife", "a poor quality kitchen knife from the Pound Shop", 100));
    // add a Potion to the same Dictionary, as Potion is a sub-class of Item
    dictItems.Add("aspirin", new Potion("aspirin", "a tablet useful after a late night...", 100));
    // add a Note to the same Dictionary, as Note is a sub-class of Item
    dictItems.Add("note", new Note("note", "a hotel provided acrylic plaque", "Welcome to the adventure.
        Don't forget the hotel key card before you leave"));
}
```

These lines create 4 items: torch, LED torch, iron key and hotel card and add them to the Items Dictionary.

One of each sub-class Weapon, Potion and Note are also added. It is important the Items are created before Locations, as there will be some Items added to one or two of the Locations. If those Dictionary keys do not exist, errors will occur.

Next the locations can be created:

```

/*
world plan:

|---|---|---|---|
| 0  1  2 | 9 |
|---|- -|---|
|   | 3 | |||| |
|---|- -| |||| | 10
| 5  4 | |||| |
|---|- -|---|
|   | 6  7 |
|---|- -|---|
| 8  |
|---|

0 = bedroom
1 = coridoor
2 = lift
3 = hallway
4 = diner
5 = store
6 = bar
7 = games
8 = wc
9 = reception
10 = home

```

```

create 11 locations in a dictionary
*****
*These could be read from a text file, so many different games can be created*
*****
lstLocation.Add(new Location("short one word name suitable for the room/area",
    "start with 'a' then a suitable name to expand on the one used above",
    "description of the room/area, as long as you like",
    Locations to the N,E,S,W No exit = "", other exits use the short one word name

e.g. "coridoor1"
    new List<Item>{dictItems["item name1"]})); <- List of any items found in this
location
*/
dictLocation.Add("bedroom", new Location("bedroom",
    "a hotel room",
    "smelling of stale cigarettes (the room, not you..)",
    "", "coridoor", "", "",
    new List<Item> { dictItems["torch"], dictItems["hotel card"] },
    dictItems["hotel card"])); // Note the last Item added here to allow entry
dictLocation.Add("coridoor", new Location("coridoor",
    "a coridoor",
    "a dimly lit passage with a stained carpet",
    "", "lift", "hallway", "bedroom"));
dictLocation.Add("lift", new Location("lift",
    "a lift",
    "a dangerous structure with a hand-operated sliding door",
    "", "", "", "coridoor",
    new List<Item>{ dictItems["LED torch"] }));
dictLocation.Add("hallway", new Location("hallway",
    "a hallway",
    "a narrow area with peeling wallpaper",
    "coridoor", "", "diner", ""));
dictLocation.Add("diner", new Location("diner",
    "the dining room",
    "a large room filled with plastic tables and chairs",
    "hallway", "", "bar", "store"));
dictLocation.Add("store", new Location("store",
    "a store-room",
    "a broom cupboard filled with junk",
    "", "diner", "", "",
    new List<Item> { dictItems["knife"], dictItems["iron key"] }));
dictLocation.Add("bar", new Location("bar",
    "'The Snug Bar'",
    "very classy, even has sawdust on the floor",
    "diner", "games", "", ""));
dictLocation.Add("games", new Location("games",
    "the games lounge",
    "a flickering light illuminates the moth-eaten sofas",
    "", "reception", "wc", "bar"));

```

```

dictLocation.Add("wc",          new Location("wc",
                                     "the toilets",
                                     "a shared facility in worse condition than found at Glastonbury",
                                     "games", "", "", ""));
dictLocation.Add("reception", new Location("reception",
                                     "the reception area",
                                     "a delightful receptionist states: Computer says 'No'",
                                     "", "home", "", "games"));
dictLocation.Add("home",       new Location("home",
                                     "your bed at home",
                                     "in a cold sweat, after a bad dream...",
                                     "", "", "", ""));

//Create the player

Player.Name   = "Inksaver";           // put your name here or write a starting form to obtain Player
details
Player.Health = 100;                  // not used so far. Enemies are needed!
Player.Strength = 50;                // not used so far. Enemies are needed!
Player.AddToInventory(dictItems["aspirin"]); // Add aspirin to the player's inventory
Player.GetItemFromInventory(dictItems["note"]); // Put note in player's hand
currentLocation = dictLocation["bedroom"]; // Set starting location (bedroom)
Intro();           // show intro text
Play();           // Play the game

```

The Constructor of each Location is made up of the following parts:

```

public Location(string name,
               string displayname,
               string description,
               string locationToNorth = "",
               string locationToEast = "",
               string locationToSouth = "",
               string locationToWest = "",
               List<Item> listitem = null,
               Item itemRequired = null)

```

It is essential the LocationToN/E/S/W is correctly completed. Mis-spelt names of the neighbouring Locations will cause a crash or unpredictable gameplay. Use the .Name property of each location as a reference.

Also the List of Items needs the same care. Separate multiple items with commas.

It would be a good idea to consider loading all this data from a text file, which would allow multiple versions of the game to work, and even allow load/save.

There are two Methods that appear at the bottom of this listing:

```

Intro(); // show intro text
Play(); // Play the game

```

Let's add those:

```
private void Intro()
{
    lblGame.Text = "Welcome to the worst adventure game you have ever played.\n" +
        "You have a serious hangover and you have no idea where you are.\n\n" +
        "Check your inventory. Anything in your sweaty hand?\n\n" +
        "Take a look around and see what you can do...\n";
}

private void Play()
{
    //fill description
    DisplayStory($"You are in {currentLocation.DisplayName}, {currentLocation.Description}" ,
true);
    this.Text = $"{currentLocation.DisplayName} - {currentLocation.Description}";
    NorthExit = currentLocation.LocationToNorth;
    EastExit = currentLocation.LocationToEast;
    SouthExit = currentLocation.LocationToSouth;
    WestExit = currentLocation.LocationToWest;
    DisplayItemList();
    DisplayInventory();
    DisplayCompass();
    DisplayHand();
}
```

As the game is Form-based, it relies on the user taking some action: clicking on a button, selecting an entry in a listbox, using a menu etc. This means **there is no need for any kind of "Game Loop"**. Windows handles all the drawing of the form and its controls, all we have to do is code events such as button clicks.

The buttons available are the four directions (N E S W), and those concerned with examining and moving items in both the world, and the player inventory.

Starting with the direction buttons:

Each button should move the player into the Location referred to.

The first thing needed in the code is a means of keeping track of all the following:

1. Current Location
2. Current Items in that Location
3. Current Items in the Player's Inventory
4. Current Item in the Player's hand

Number 1 has already been taken care of: `private Location currentLocation;` in the Form Class Variables and

```
currentLocation = dictLocation["bedroom"]; // Set starting location (bedroom)
```

in the CreateGame() function

Number 2 was fulfilled when the Locations were created and added to the Dictionary:

```
"" , "coridoor", "", "",
new List<Item> { dictItems["torch"], dictItems["hotel card"] }));
```

Number 3 has already been done, there is a Dictionary<string, Item> in the Player class

Number 4 is easy:

Add this line to the Player Class:

```
private static Item ItemInHand;
```

This variable can be set at the start of the game, and then updated every time the player moves into a new location.

Now the world has been created, it can be tested to see if the user can actually play the game.

This game has:

- Four direction buttons (N E S W)
- Two buttons to interact with Items in the game (Examine, Take)
- Two Inventory buttons (Hold Selected Item, Examine)
- Three buttons dealing with Items in the players hand (Drop, Return to backpack, Use)
- Two ListBoxes that can be clicked to highlight Items

Four string variables are needed to hold the key of any neighbouring Locations.

If there is not an entrance in one of the directions, the value should be "" (empty string)

In the world example set earlier, Location["bedroom"], has only got an exit to the East ("coridoor"), the others are set to an empty string ("").

Make the following changes to the Player class:

```
#region Public Properties
public static int Strength;           // using a public read/write variable (Not currently used)
public static string Name { get; set; } // using an auto-property
public static int Health { get; set; } // using an auto-property
public static Dictionary<string, Item> Inventory = new Dictionary<string, Item>();
public static Item ItemInHand;       // using a public read/write variable
#endregion
```

Some custom Methods are used:

This one adds a new Item to the inventory:

```
public static void AddToInventory(Item item)
{
    Inventory.Add(item.Name, item);
}
```

This one is a bit more clever. When used, it:

Returns whatever Item is currently in the Player's hand back to the inventory,
Puts the newly selected Item in the players hand.

Removes the Item now in the Player's hand from the inventory.

```
public static void GetItemFromInventory(string itemName)
{
    /// Overload for this function, takes string parameter
    /// Converts it to Item object
    /// Then passes it to the relevant overload
    GetItemFromInventory(Inventory[itemName]);
}
```

This one returns an Item held in the hand back to the inventory, then sets the ItemInHand to null.

```
public static void ReturnItemToInventory()
{
    if(ItemInHand != null)
    {
        Inventory.Add(ItemInHand.Name, ItemInHand);
    }
    ItemInHand = null;
}
```

The full listing for the Player class:

```
using System.Collections.Generic;

namespace Text_Adventure_Tutorial
{
    public static class Player
    {
        /// <summary>
        /// There is only one player, so a static class is used
        /// There is no real need to encapsulate variables in a static class
        /// but some have been given auto-properties to show how it can be done
        /// </summary>

        #region Public Properties
        public static int Strength;           // using a public read/write variable (Not in use)
        public static string Name { get; set; } // using an auto-property
        public static int Health { get; set; } // using an auto-property
        public static Dictionary<string, Item> Inventory = new Dictionary<string, Item>();
        public static Item ItemInHand;       // using a public read/write variable
        #endregion

        #region Static Constructor
        static Player()
        {
            /// Static classes do not usually require a constructor
            /// They run once only, when the first reference to the
            /// class is made, and can be used to initialise data, etc
        }
        #endregion

        #region Public Methods
        public static void AddToInventory(Item item)
        {
            Inventory.Add(item.Name, item);
        }
        public static void GetItemFromInventory(Item item)
        {
            /// Overloaded function, This version takes an Item parameter.
            /// It can also accept a string parameter in another overload
            /// which sends it back here as an Item for processing
            ReturnItemToInventory();
            ItemInHand = item;
            Inventory.Remove(item.Name);
        }
        public static void GetItemFromInventory(string itemName)
        {
            /// Overload for this function, takes string parameter
            /// Converts it to Item object
            /// Then passes it to the relevant overload
            GetItemFromInventory(Inventory[itemName]);
        }
        public static void ReturnItemToInventory()
        {
            if(ItemInHand != null)
            {
                Inventory.Add(ItemInHand.Name, ItemInHand);
            }
            ItemInHand = null;
        }
        #endregion
    }
}
```

You will note the function `GetItemFromInventory(Item item)` appears twice, with the second version taking different parameters: `GetItemFromInventory(string itemName)`

This is an example of “overloading” a function, so it can accept different parameters

Run the game again. It should load OK, but absolutely nothing works. Not even any display in the lblGame Label!

Another function is needed in the MainForm() to add text to the lblGame Label

Write the DisplayStory() function

Add this new function above the CreateGame() function

```
private void DisplayStory(string AddText, bool clear = false)
{
    if (clear)
    {
        lblGame.Text = string.Empty;
    }
    lblGame.Text += AddText;
}
```

This function works by taking the text to be added as a parameter, along with a Boolean value determining whether to clear the display first.

If clear is passed, clear the Label

The new text is added to the existing text.

Write the Play() function

The Play() function is needed to do the following:

- Add a description of the current location to lblGame
- Display the current Location in the window titlebar
- Set the exit variables
- (Update the Listbox showing any available Items in this area)
- (Update the Inventory Listbox)
- (Enable / Disable direction buttons to indicate available exits)
- (Update the label and buttons associated with the Player’s hand.)

The last four (in brackets) are best done with a number of sub-functions called from Play() , so start off with this in the MainForm by adding it after the CreateGame() function:

Positioning of Procedures/Functions is not important, but it is easier to add them in alphabetical order so you can find them easily

```

private void Play()
{
    DisplayStory($"You are in {currentLocation.DisplayName}, {currentLocation.Description}" , true);
    this.Text = $"{currentLocation.DisplayName} - {currentLocation.Description}";
    NorthExit = currentLocation.LocationToNorth;
    EastExit = currentLocation.LocationToEast;
    SouthExit = currentLocation.LocationToSouth;
    WestExit = currentLocation.LocationToWest;
    DisplayItemList();
    DisplayInventory();
    DisplayCompass();
    DisplayHand();
}

```

Going through this function step by step:

DisplayStory() adds the name and description of the current location to the display (lblGame)

this.Text = \$" changes the Window Title to the current Location name / description

<direction>Exit = sets the exit names to the relevant location key

Write the following four functions, which are called at the end of Play()

1 DisplayItemList()

This function is used to update the ListBox lblItems, so it will show a list of any Items found in the current Location

```

private void DisplayItemList()
{
    lblLocationItems.Text = "Items in " + currentLocation.DisplayName;
    lstLocation.Items.Clear();
    btnExamine.Enabled = false;
    btnTake.Enabled = false;
    if(currentLocation.Items.Count > 0)
    {
        foreach (KeyValuePair<string, Item> item in currentLocation.Items)
        {
            lstLocation.Items.Add(item.Value.Name);
        }
        btnExamine.Enabled = true;
        btnTake.Enabled = true;
    }
}

```

The actions carried out are:

- Set the label lblLocationItems to show eg "Items in a hotel room"
- Clear the ListBox
- Disable the Examine and Take buttons in case there are no items present
- Check if there are any items present in the current Location
- Add any Items to the ListBox
- Enable the buttons if Items are present.

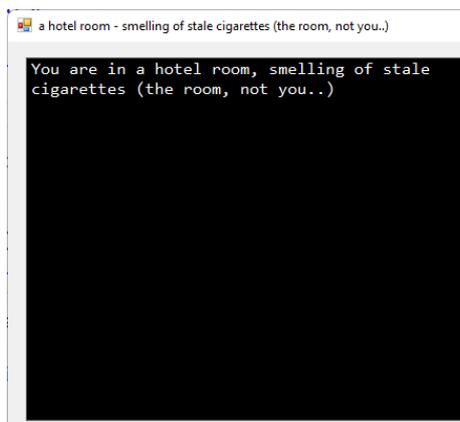
Note the lines:

```
foreach (KeyValuePair<string, Item> item in currentLocation.Items)
{
    lstLocation.Items.Add(item.Value.Name);
}
```

Because the Items are stored in a Dictionary we have to use KeyValuePair in the foreach loop.

The variable `item` has two Properties: `.Key` and `.Value` providing the Dictionary key and value respectively, so `item.Value` returns the Item object, and `item.Value.Name` returns it's `.Name` property

Run the game to test it out:



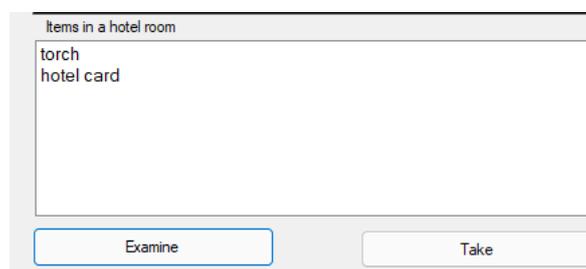
The lblGame Label is working: It tells the user where the Player is standing.

2 DisplayInventory()

This function is used to update the ListBox lstInventory, so it will show a list of any Items found in the Player's inventory

```
private void DisplayInventory()
{
    lstInventory.Items.Clear();
    if(Player.Inventory.Count > 0)
    {
        foreach (KeyValuePair<string,Item> item in Player.Inventory)
        {
            lstInventory.Items.Add(item.Value.Name);
        }
    }
}
```

The code is similar to that used to update the Items lisbox, except it uses the Player.Inventory public variable.



3 DisplayCompass()

This function enables / disables the four direction buttons and changes the background colour of each button, depending whether there is an exit from the current Location in that direction.

As the same code is used on all four buttons, it saves typing if you write **YAF** (Yet Another Function) to do the job:

```
private void DisplayCompass()
{
    DisableButton(btnNorth);
    DisableButton(btnEast);
    DisableButton(btnSouth);
    DisableButton(btnWest);
    if (NorthExit != "") { EnableButton(btnNorth); }
    if (EastExit != "") { EnableButton(btnEast); }
    if (SouthExit != "") { EnableButton(btnSouth); }
    if (WestExit != "") { EnableButton(btnWest); }
}

private void DisableButton(Button button)
{
    button.Enabled = false;
    button.BackColor = Color.LightGray;
    button.ForeColor = Color.Gainsboro;
}
```

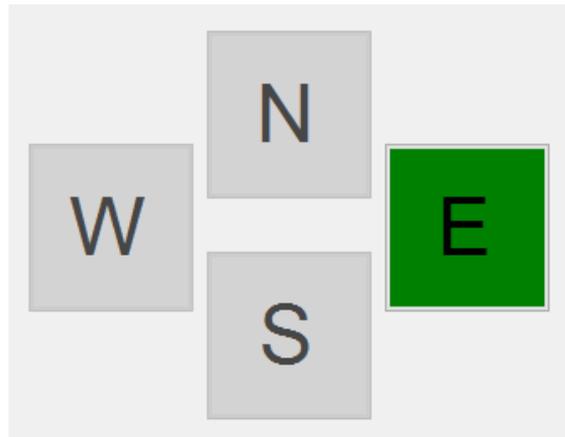
The first 4 lines of the DisplayCompass() function call the DisableButton() function in turn, passing the name of each Direction button as a parameter.

The DisableButton() function disables the button, and changes the ForeColor and BackColor properties.

The last 4 lines of DisplayCompass() will re-enable the button if it has an exit string that is not empty, by using **YAF** EnableButton(): Add the DisableButton() and EnableButton() functions

```
private void EnableButton(Button button)
{
    button.Enabled = true;
    button.BackColor = Color.Green;
    button.ForeColor = Color.Black;
}
```

Run the game again:



The button btnEast now has a green background, a black Font and is Enabled.

You can now click this button and go East!

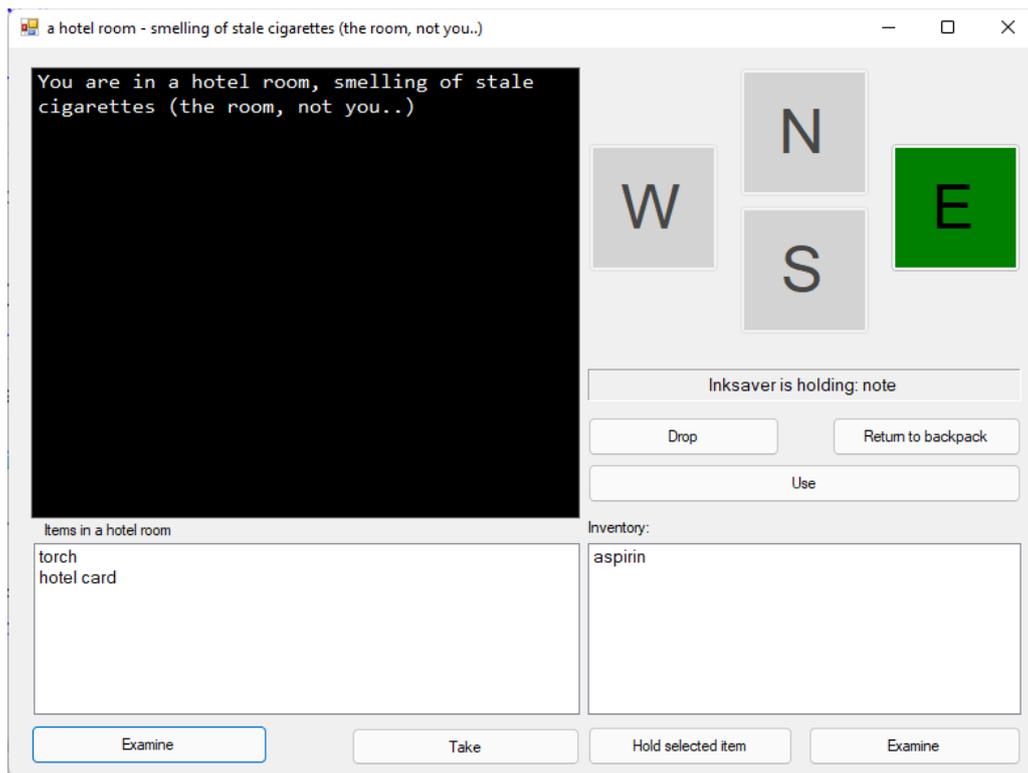
Er.. Well you have not coded that yet, so not at the moment.

One more thing to do before coding the direction buttons:

4 DisplayHand();

This function updates the label and buttons that represent the Player's hand:

```
private void DisplayHand()
{
    if(Player.ItemInHand != null)
    {
        lblHand.Text = Player.Name + " is holding: " + Player.ItemInHand.Name;
        btnDrop.Enabled = true;
        btnReturn.Enabled = true;
        btnUse.Enabled = true;
    }
    else
    {
        lblHand.Text = Player.Name + " is holding: nothing";
        btnDrop.Enabled = false;
        btnReturn.Enabled = false;
        btnUse.Enabled = false;
    }
}
```



Now the graphical display part is complete:

- The Window title bar is updated
- The lblGame Label is telling the story
- The Items Lisbox is displaying items in the current location
- The label above the listbox is updated
- The direction buttons are set for the next move
- The Player's hand is updated
- The Player's Inventory is updated

Time to sort out the Direction buttons

The Direction Buttons

A function called Go() is used with each of the buttons. Add this to the FormMain()

```
private void Go(string Direction)
{
    //has to be an exit here else button would not be enabled
    Location checkLocation = currentLocation;
    switch (Direction)
    {
        // CheckExit will change currentLocation to new value
        case "north": CheckExit(dictLocation[NorthExit]); break;
        case "east":  CheckExit(dictLocation[EastExit]);  break;
        case "south": CheckExit(dictLocation[SouthExit]); break;
        case "west":  CheckExit(dictLocation[WestExit]);  break;
    }
    if (checkLocation != currentLocation)
    {
        Play();
    }
}
```

The same function is called by each button, but with a different Parameter.

The btnNorth calls Go("north"), btnEast calls Go("east") etc.

The switch statement chooses the matching Direction and executes the code in the matching case.

If you remember earlier the NorthExit, EastExit etc. variables contain a string value corresponding to the key of the Location List.

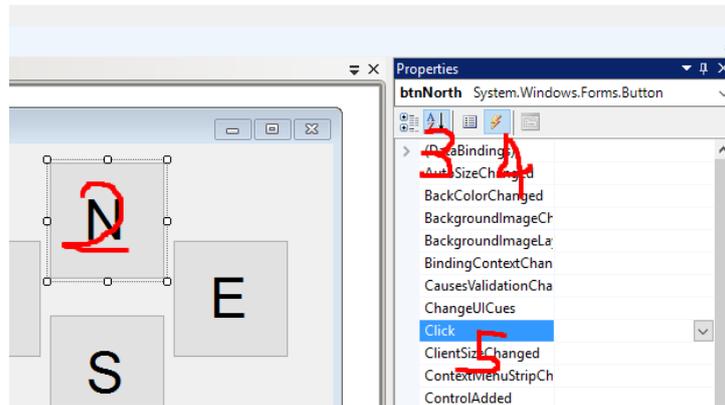
```
private void CheckExit(Location exitLoc)
{
    if(exitLoc.ItemRequired == null)
    {
        currentLocation = exitLoc;
    }
    else
    {
        if(exitLoc.ItemRequired == Player.ItemInHand)
        {
            currentLocation = exitLoc;
        }
        else
        {
            DisplayStory($"{exitLoc.ItemRequired.Name} in your hand to enter {exitLoc.DisplayName}");
        }
    }
}
```

The CheckExit() function updates the currentLocation

The function Play() is called to update the GUI

To link each button to the underlying code use the following method for each button in turn:

- 1) Click on the 'Design' tab
- 2) Select btnNorth / btnEast / btnSouth / btnWest
- 3) Click the A->Z button in the Properties window
- 4) Click the lightning bolt 'Events' in the Properties window
- 5) Double-Click the 'Click' event



This will add new code to the MainForm() Source at the end of the class:

```
private void BtnNorthClick(object sender, EventArgs e)
{
}

```

re-Format this to occupy one line with Go("north"); enclosed:

Repeat this for all the buttons in turn:

```
private void BtnNorthClick(object sender, EventArgs e)           {Go("north");}
private void BtnEastClick(object sender, EventArgs e)           {Go("east");}
private void BtnSouthClick(object sender, EventArgs e)          {Go("south");}
private void BtnWestClick(object sender, EventArgs e)            {Go("west");}
```

OPTIONAL - compact the code and move it to the top, along with the MainFormLoad() event and any other events as you create them:

```
public MainForm()
{
    InitializeComponent();
    this.CenterToScreen();
}
#region Event Handlers
private void MainFormLoad(object sender, EventArgs e)           {CreateGame();}
private void BtnDropClick(object sender, EventArgs e)           {DropItem();}
private void BtnNorthClick(object sender, EventArgs e)          {Go("north");}
private void BtnEastClick(object sender, EventArgs e)           {Go("east");}
private void BtnSouthClick(object sender, EventArgs e)          {Go("south");}
private void BtnWestClick(object sender, EventArgs e)            {Go("west");}

```

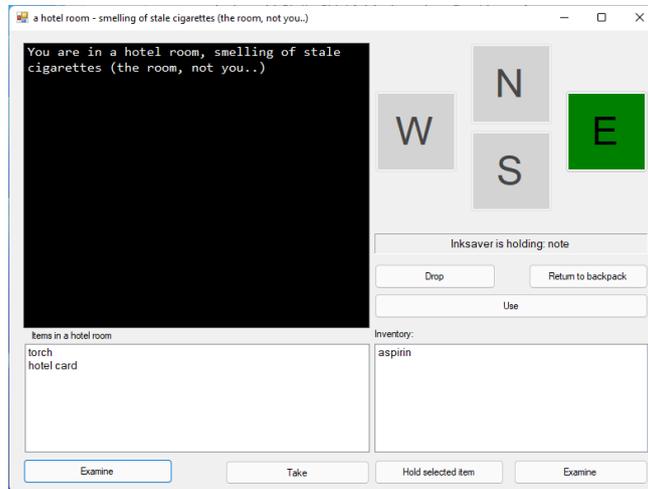
The only reason for doing this is to keep the code tidy.

As the game grows bigger, more Events are added, and more user-created functions are written.

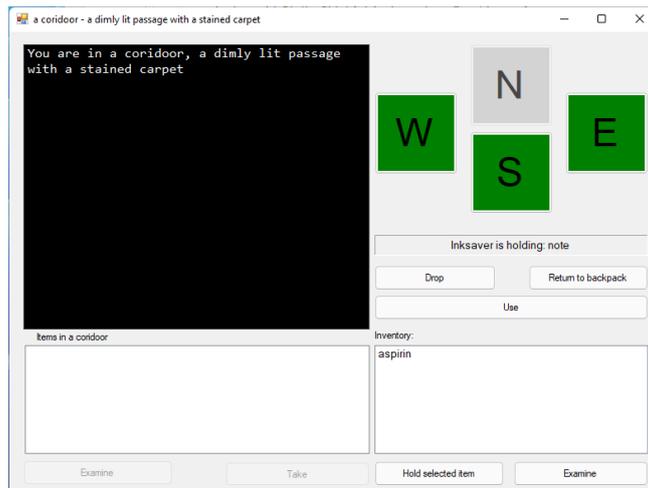
If all Events are moved to the top, and a new function written for each event, these can be kept to one line each.

User-defined functions can be placed in alphabetical order under all events.

Run the game again:



Click on the button btnEast ('E')



There are now three buttons active: 'E', 'S' and 'W'

The story reads:

You are in a hotel room, smelling of stale cigarettes (the room, not you..)

You are in a corridor, a dimly lit passage with a stained carpet

Finally, the game is running, and you can move around the world.

Interacting with the World

The next stage, now you can move around, is to be able to examine objects in each location, pick them up and drop them elsewhere if needed, and of course make use of them.

Using the same method as before with the Direction buttons, do the same for the remaining seven buttons to get the following code:

```
private void BtnExamineClick(object sender, EventArgs e)           {ExamineItem(1stLocation);}
private void BtnTakeClick(object sender, EventArgs e)             {TakeItem();}
private void BtnHoldClick(object sender, EventArgs e)             {HoldItem();}
private void BtnReturnClick(object sender, EventArgs e)           {ReturnItem();}
private void BtnUseClick(object sender, EventArgs e)              {UseItem();}
private void BtnExamineInventoryClick(object sender, EventArgs e) {ExamineItem(1stInventory);}
```

Unfortunately that means another six functions need writing...

ExamineItem()

This function is used to read the description of an item when it appears in the Items Listbox 1stLocation or 1stInventory

- Check if there are any selected items
- Force the variable Type Item by using (Item) in brackets before the listBox.SelectedItem.
- Add the .Name and .Description to the story.

```
private void ExamineItem(ListBox lb)
{
    if (lb.SelectedItems.Count > 0)
    {
        Item item = dictItems[lb.SelectedItem.ToString()]; // find the item by name
        DisplayStory($"{ "\nYou look at the {item.Name} ");
        DisplayStory(item.Description);
    }
}
```

TakeItem()

This function does the following:

- Checks if an Item is selected
- Adds it to the Player's inventory
- Adds comments to the story
- Removes it from the current Location
- Updates the GUI

```

private void TakeItem()
{
    if (lstLocation.SelectedItems.Count > 0)
    {
        // Take item from location; Put in inventory
        // add to inventory items
        Player.AddToInventory(dictItems[lstLocation.SelectedItem.ToString()]);
        DisplayStory($"\\nYou take the {lstLocation.SelectedItem}");
        // remove item from current location
        currentLocation.RemoveItem(lstLocation.SelectedItem.ToString());
        DisplayItemList();
        DisplayInventory();
    }
}

```

HoldItem()

This function does the following:

- Checks if an Item is selected
- Puts the selected Item in the Player's hand
- Adds comments to the story
- Updates the GUI

```

private void HoldItem()
{
    if (lstInventory.SelectedItems.Count > 0)
    {
        //return the current item held to inventory and take new item
        Player.GetItemFromInventory(lstInventory.SelectedItem.ToString());
        DisplayStory($"\\nYou get the {Player.ItemInHand.Name} from your backpack");
        DisplayInventory();
        DisplayHand();
    }
}

```

DropItem()

This function does the following:

- Adds comments to the story
- Adds the Item in the players hand to the current Location
- Removes it from the player's hand
- Updates the GUI

```

private void DropItem()
{
    DisplayStory($"\\nYou drop the {Player.ItemInHand.Name}");
    currentLocation.AddItem(Player.ItemInHand);
    Player.ItemInHand = null;
    DisplayItemList();
    DisplayHand();
}

```

ReturnItem()

This function does the following:

- Return the Item in the Player's hand back to the Inventory
- Updates the GUI

```
private void ReturnItem()
{
    Player.ReturnItemToInventory();
    DisplayInventory();
    DisplayHand();
}
```

UseItem()

An Item can be used on:

- An exit from a location (key)
- As a weapon (if there was an Enemy class...)
- On another Item to Craft something new (If there were two hands...)

```
private void UseItem()
{
    /// Use the item currently in your hand
    /// If it is a key, allow use into a location
    /// If it is a weapon, use it against an enemy
    /// If it is a note, read it
    Item currentItem = Player.ItemInHand;
    if (currentItem == null)
    {
        DisplayStory("\nYou are not holding anything in your hand!", true);
    }
    else
    {
        if (currentItem is Weapon)
        {
            Weapon weapon = (Weapon)currentItem;
            DisplayStory( $" \nYou use the {weapon.Name} to scratch the wall with" +
                $" {weapon.Damage} points of damage");
        }
        else if (currentItem is Potion)
        {
            Potion potion = (Potion)currentItem;
            DisplayStory( $" \nYou use the {potion.Name} for a health boost of" +
                $" {potion.Healing} and feel so much better!");
            Player.ItemInHand = null;
            DisplayHand();
        }
        else if (currentItem is Note)
        {
            Note note = (Note)currentItem;
            DisplayStory( $" \nThe {note.Name} in your hand is " +
                $" {note.Description} which reads: \n' {note.Message}'");
        }
    }
}
```

Modify the Location class:

Add these 3 methods

```
#region Methods
public void AddItem(Item item)
{
    Items.Add(item.Name, item);
}
public void RemoveItem(Item item)
{
    Items.Remove(item.Name);
}
public void RemoveItem(string itemName)
{
    Items.Remove(itemName);
}
}
#endregion
```

Run the game again:

It starts the same as before, with the E button green.

Notice the message: Inksaver is holding: note

Click on the "Use" button.

Read the message

Click on the "Drop" button

Click on the word "aspirin" in the Inventory list box

Click "Hold selected item" → Inksaver is holding: aspirin

Click on the "use" button

Click on 'E' and as before you are moved to the corridor.

```
You are in a hotel room, smelling of stale
cigarettes (the room, not you..)
The note in your hand is a hotel provided
acrylic plaque which reads:
'Welcome to the adventure. Don't forget the
hotel key card before you leave'
```

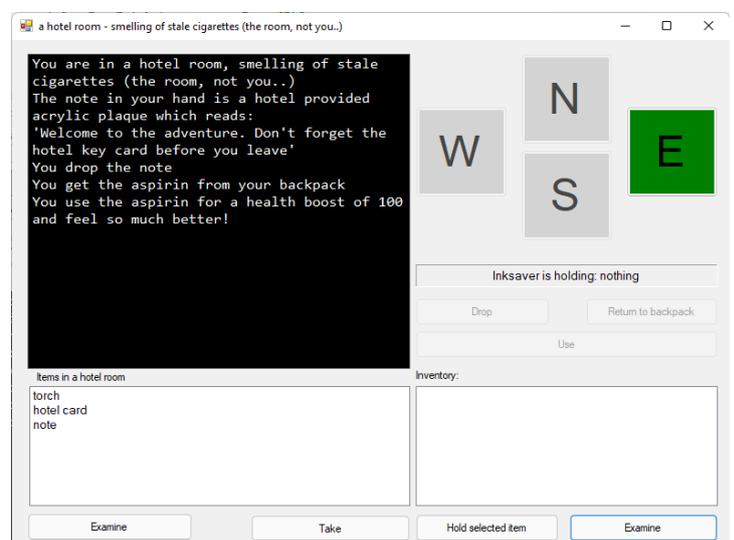
Click on the "Drop" button

Click on the word "aspirin" in the Inventory list box

Click "Hold selected item" → Inksaver is holding: aspirin

Click on the "use" button

Click on 'E' and as before you are moved to the corridor.



Click on 'W' to return to the bedroom, but instead of returning you get this message:

```
You are in a coridoor, a dimly lit passage  
with a stained carpet  
You need the hotel card in your hand to enter  
a hotel room
```

'You need the hotel card to enter a hotel room'

Whoops! Start again!

Make sure you select the hotel card from the left hand list, 'Examine' and 'Take' it first so it goes into your inventory

When attempting to return to your room:

- Click on the 'hotel card' in the inventory
- Click on the button 'Hold Selected Item'
- The GUI changes to show the Player holding the Hotel Card
- The story adds 'You get the hotel card from your backpack'

Now you can go West back to the hotel room

This adventure is based on the theme of a hotel. Make your own adventure using a different theme with different objects, with more locations. You can put Items in different Locations which have to be picked up and placed in the Player's hand to allow you to move further along Improvements to the game require a lot more code, but can include puzzles, quests, enemies, bonus points, loss of health from enemies or traps in some areas and all the traditional things associated with an adventure game.

But that is the topic of another tutorial.