Learn C#

by Building a Simple RPG

Scott Lilly

2nd Edition, June 2016

This work is a reproduction of an online tutorial by Scott Lilly. The original tutorial can be reached at scottlilly.com.

If you find an error in the text or the code, have a suggestion on improving this book, or just want to say hi, let me know at feedback@scottlilly.com, or leave a comment at scottlilly.com.

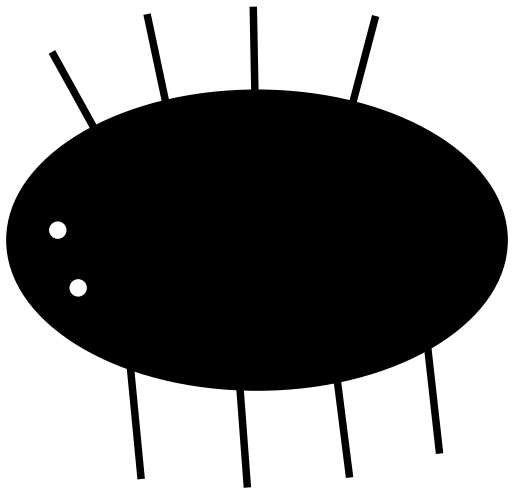
Thanks and have fun!

If you want to write a Role Playing Game, but don't know how to program, or just want to learn how to program in C#, then you're at right the place.

These lessons will take you from a complete beginner, to being an author of a Role Playing Game, for free.

Now, this isn't the world's greatest game. In fact, it's very short and kind of ugly.

However, as you create it, you'll learn the most common C# programming practices and techniques. Then, if you want, you can improve the game, adding more features and your own special touch to it.



Mandatory Giant Spider!

TABLE OF CONTENTS

About this tutorial 1

General information about programming in C# 3

Parts of Visual Studio 4

Building the base game

1 Defining classes and objects for the game 9

2 Building the solution for the game 11

3 Building the first screen 13

4 Creating the Player class and its properties 16

5 Creating objects from classes 19

6 Creating the remaining classes 23

7 Inheritance and base classes 27

8 Setting properties with a class constructor 30

8.1 Using class constructors with derived classes 33

9 Using your classes as datatypes 36

10 Creating collections of objects 38

11 Using a static class 44

12 Add the remaining UI controls 53

13 Functions, procedures, and methods 56

13.1 Creating functions to handle user input 60

14 Variables 61

14.1 IF, ELSE, ELSE IF statements 64

14.2 FOREACH loops 69

15 Getting random numbers for the game 72

16 Writing the function to move the player 75

16.1 Refactoring the player movement function 92

16.2 Functions to use weapons and potions 102

17 Running the game on another computer 109

18 Future enhancements for the game 111

Enhancements to the game

19.1 Scroll to the bottom of a rich text box 115
19.2 Use a calculated value for a property 116
19.3 Clean up the source code by converting FOREACH loops to LINQ 119
19.4 Saving and loading the player information (XML) 124
19.5 Changing dropdown default values 137
19.6 Increase maximum hit points when the player gains a level 142

Improving SuperAdventure's code quality by refactoring

20 Refactoring the SuperAdventure program 145
20.1 Binding a custom object's properties to UI controls 146
20.2 Binding list properties to datagridviews 151
20.3 Binding child list properties to a combobox 159
20.4 Moving the game logic functions from the UI project to the Engine project. . 168

Adding a vendor to locations (with buying and selling items)

21 Plans for adding a vendor to locations 181
21.1 Adding a price to game items 182
21.2 Create the vendor class and add it to locations 186
21.3 Add a button and create its eventhandler in code, without the UI design screen 189
21.4 Completing the trading screen 192

Use SQL to save and restore player's game data

22 Installing MS SQL Server on your computer 205
22.1 Creating database tables from classes 206
22.2 Creating the SQL to save and load the saved game data. 213

Creating a console UI for SuperAdventure

23 Creating a console front-end for the game. 231

Final refactoring (cleanup) of the SuperAdventure source code

24 Make the SuperAdventure source code easier to understand and modify 247

About this tutorial

What is in these lessons?

These lessons will teach you the basics of C#, by building a very simple role-playing game (RPG).

What will be in the game?

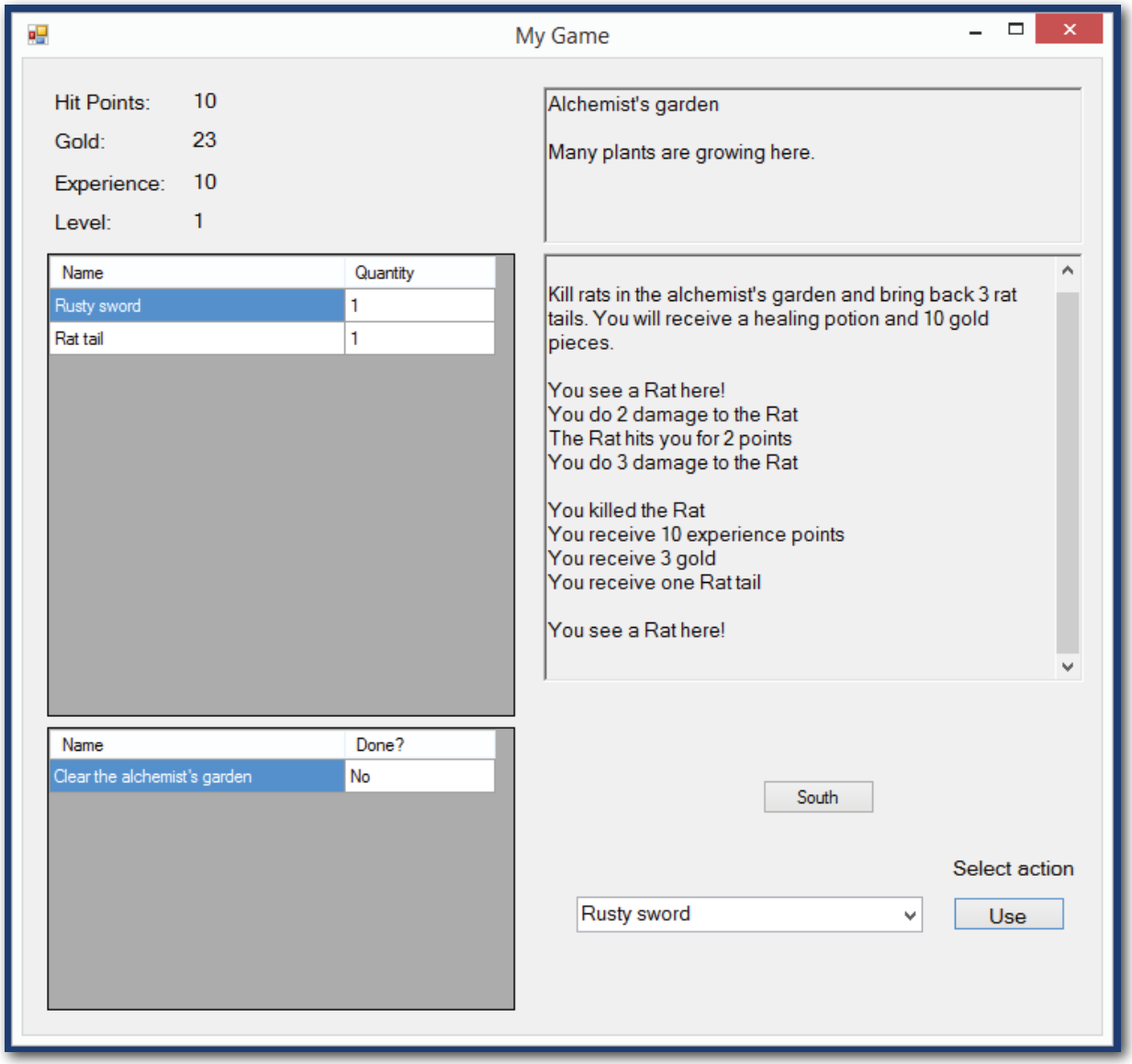
The game will let you create a player, set up locations for them to travel to, discover quests to complete, and fight monsters.

When I say that it's *very simple*, that's what I mean?

There are no graphics, there are only nine locations, three different types of monsters, two different weapons, and two quests. The battles are very simple, and there is no armor, magic, or crafting. However, at the end, you will know the basics of C# programming. That's the goal of these lessons – not to build the next Skyrim or World of Warcraft.

What will the game look like?

Like this:



What will I learn?

The focus of these lessons is to show you the most common things you'll need to do to create a C# program – such as creating classes, handling user inputs, and doing common calculations. Think of the Pareto Principle. You'll learn the 20% of the things that you'll need to do 80% of the time, when writing a program in C#.

Will I learn the *best* way to write a C# program?

There's always a better way. In fact, there will be some things that I know could be done better, but are more complex than I want to introduce to you right now. This will get you started, with the ability to create a program that works. But if you want to do more advanced things in your programs, you'll need to learn, and practice, more.

Can I add more to the game?

Yes! If you want to add more locations, creatures, quests, weapons, potions, etc., you'll be able to easily do that. If you want to expand the game with more features (armor, crafting, buying/selling, poisons, spells, repeatable quests, etc.), let me know. I may be able to find some time to expand the game.

General information about programming in C#

Before you write your first program in C#, you need to know a few general things about C# and programming.

C# is case-sensitive. When you name anything in C# (class, variable, function, etc.), the casing you use is important. *Player* is different from *player* and *PLAYER*. It usually doesn't matter what you use, as long as you are consistent. However, there are some standards that many C# programmers use. If you get used to them now, that will make it easier for you to read other programs, or have other programmers read your programs. The standards I'll use while building this game are fairly common. Visual Studio is the **editor** we will use. An editor is basically a word processor for your program. With a word processor, you can type a document, makes changes to it, and have it check if the spelling and grammar is correct. An editor lets you do similar things with your programs.

C# is a *high-level* language. This means is that C# looks similar to natural English – well, at a certain level. Computers really only understand machine language, a very low-level language, which doesn't look anything like English. So it's much easier to program in a high-level language, and letting the computer compile that into a low-level language.

What are functions, procedures, and methods? These are all names we use to describe a small piece of a program that does a specific thing. For example, in this game we're going to fight monsters. When the player clicks on the *Attack* button, we need to determine a random amount of damage to apply to the monster. So, we'll create a function to determine this random number. By having this piece of the program in a function, we can use it in several different places in the program. Once the player defeats a monster, we want to randomly determine what *loot* the player gets from the monster. So, we can use this same random number function for that.

What is compiling/building? Before a computer can run a C# program, the program needs to be converted into something a computer can read. This is called compiling. In Visual Studio, you compile your program by either selecting the *Build* menu option or the *Start* menu option (which compiles the program and then runs it). We are going to use Visual Studio Express 2013 for Windows Desktop. It's free for you to download and use. You can get it at visualstudio.com.

Make sure you have enough space on your hard drive as the installation can be quite demanding (upwards of 10 gigabytes). If you happen have a newer version of Visual Studio installed, you're ready to go.

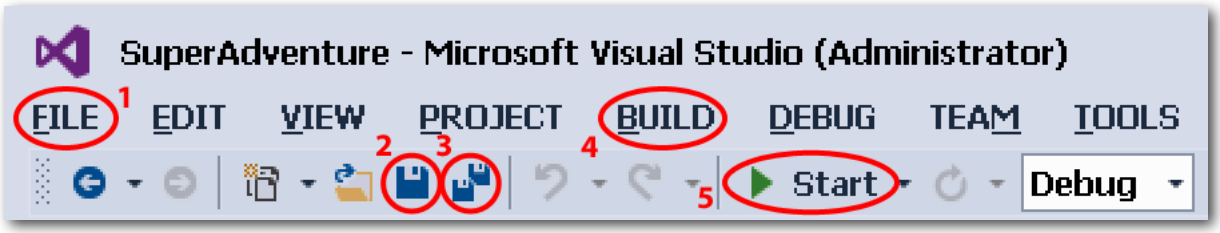
Parts of Visual Studio

Before you start using Visual Studio, you should get familiar with the different parts of it. Here is what Visual Studio will look like after you start building your game.

Menu

Just like many programs, the *menu options* are at the top. The ones you'll use most often are:

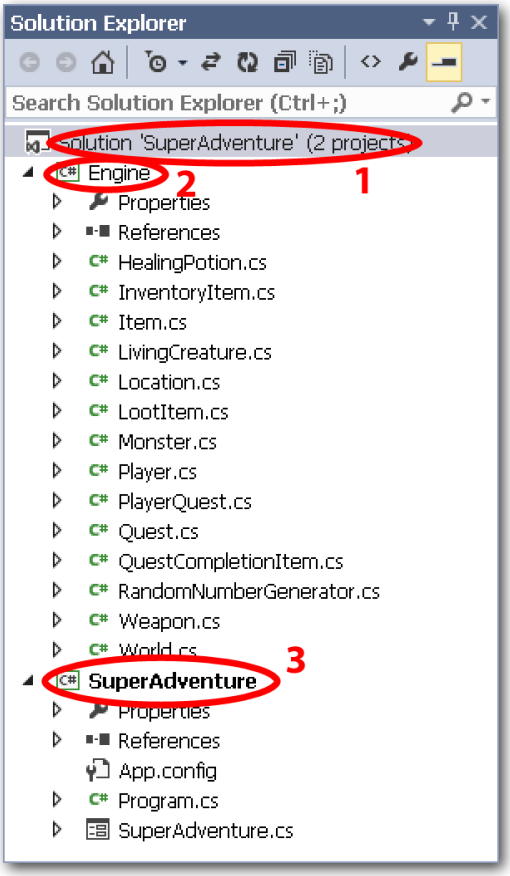
1. File – To open your solution, and start working on it.
2. Save – Save the current file you're working on.
3. Save All – Save all changes in the current solution you're working on.
4. Build – This **compiles** your solution – converts it from C# code to code that the computer can understand. This will also tell you if there are any problems in your solution.
5. Start – This builds (compiles) your program and runs it – so you can actually use it.



Solution Explorer

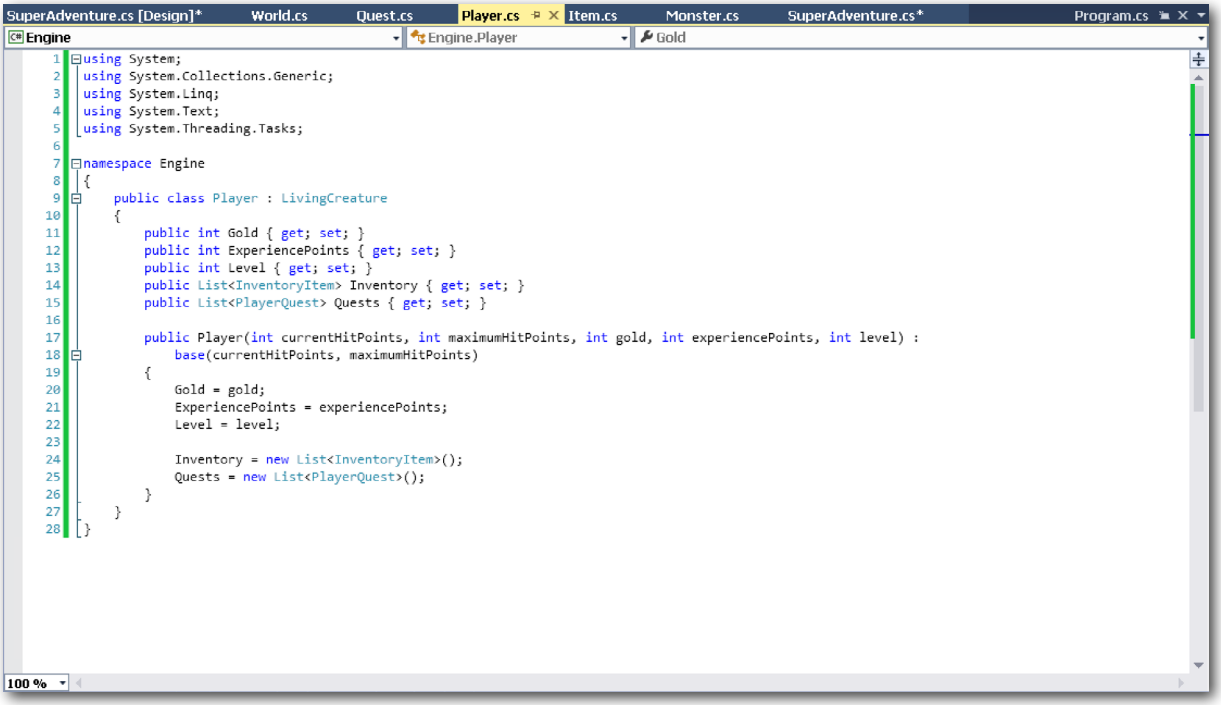
This section shows all the projects in your solution and all the files in the projects. This is where you will add files to your projects, rename or move existing files, and (sometimes) delete files from your project.

1. The SuperAdventure Solution – The top level grouping of your program/application.
2. Engine project – Where we will put the **logic** of the program.
3. SuperAdventure project – Where we will put the screen/display part of the program.



Workspace

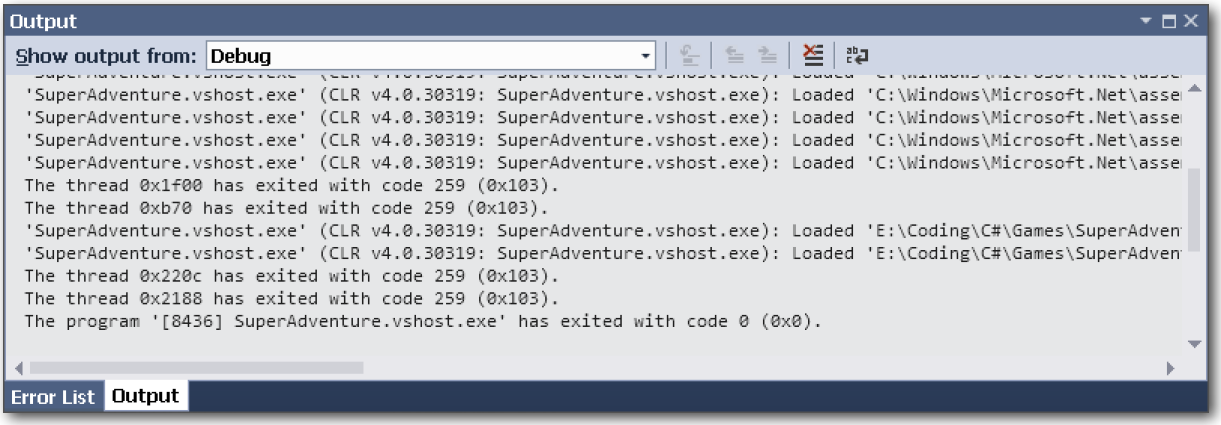
This is the space where you actually work on a part of your program. You select what you want to work on, from the Solution Explorer, and work on it here. You can have several files open on your workspace at one time (see the tabs at the top of the workspace), but you'll only have one *on top,* that you're actually working on at the moment.




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Engine
{
    public class Player : LivingCreature
    {
        public int Gold { get; set; }
        public int ExperiencePoints { get; set; }
        public int Level { get; set; }
        public List<InventoryItem> Inventory { get; set; }
        public List<PlayerQuest> Quests { get; set; }

        public Player(int currentHitPoints, int maximumHitPoints, int gold, int experiencePoints, int level) :
            base(currentHitPoints, maximumHitPoints)
        {
            Gold = gold;
            ExperiencePoints = experiencePoints;
            Level = level;

            Inventory = new List<InventoryItem>();
            Quests = new List<PlayerQuest>();
        }
    }
}
```

Output

When you build, or run, your program, this is where you'll receive status messages. If everything is OK, you see that everything succeeded. If there were any problems, you'll see where they are, so you can go fix them.

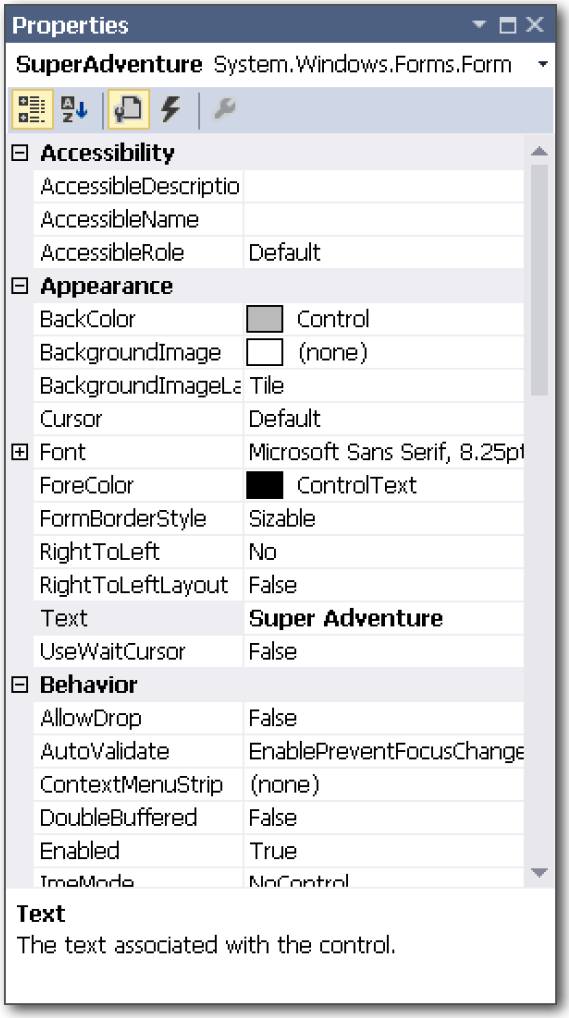


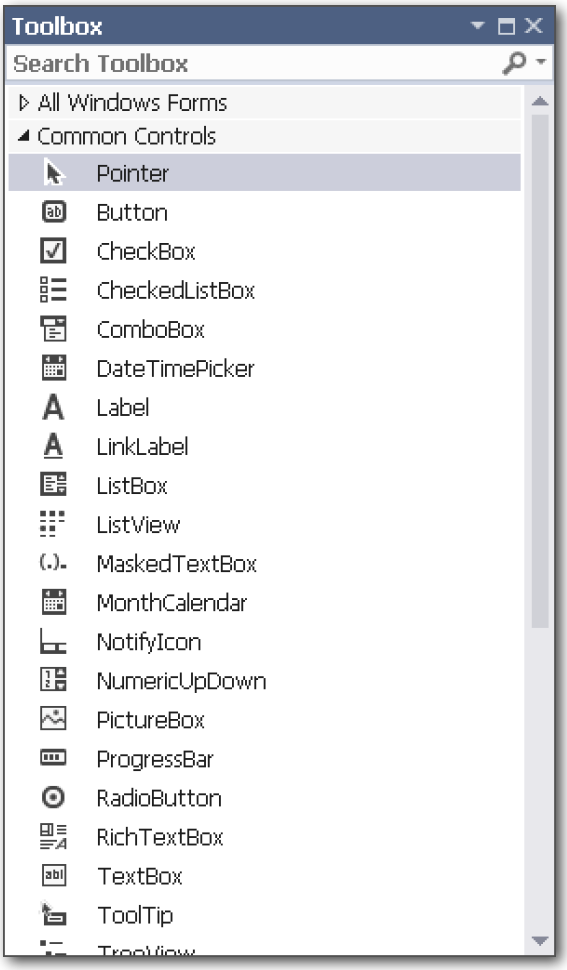
Properties

When you're working with the parts of the game that appear on the screen – from the main form to the individual buttons and boxes on it – the Properties section will show you things you change about your currently selected object.

For instance, you can set the height and width of a form. You can set the words you want displayed on a button. You can set whether or not something is visible.

We'll set some properties for the objects we create from this section of Visual Studio. Some others will be set from other places inside the program.





Toolbox

Just like the *Properties* section, the Toolbox area is filled when you're currently working on the parts of your program that are displayed on the screen.

This section shows you all the things you can add to the forms in your program – buttons, labels (text), checkboxes, radio buttons, etc. You can select what you want to add to the form and drag it to the place where you want it located on your form. You might here this called **drag-and-drop programming.**

1-18

Building the base game

Defining classes and objects for the game

Building the first screen

Using your classes as data types

Creating collections of objects

Functions, procedures, and methods

IF, ELSE, ELSE IF statements

and more...

1 Defining classes and objects for the game

Lesson objectives

At the end of this lesson, you will know...

- ➔ The most important thing to know about object-oriented programming
- ➔ How to discover the classes/objects your program will need
- ➔ The difference between classes and objects

What is object-oriented programming? There are whole books devoted to answering this question. However, the main aspect of object-oriented programming (OOP) is that the program tries to imitate the real world by thinking of things as the objects they represent in the real world. So, if you're writing a program for a bank, your classes/objects will be things such as customers, accounts, deposits, withdrawals, checks, loans, etc.

There are other important aspects of OOP, and we'll talk about some of them as we write the game. However, if you keep this one thing in mind, you'll be halfway there.

What are the classes/objects in our game? An easy way to figure out what your objects are going to be is to write down what you want your program to do, and underline all the nouns. For our game, we want to do these things:

- The **player** goes to locations.
- The player may need to have certain **items** to enter a **location.**
- The location might have a **quest** available.
- To complete a quest, the player must collect certain items and turn them in.
- The player can collect items by going to a location and fighting monsters there.
- The player fights monsters with **weapons.**
- The player can use a **healing potion** while fighting.
- The player receives loot items after defeating a monster.
- After turning in the quest, the player receives reward items.

So, the nouns (classes/objects) in this game will be *Player, Location, Item, Quest, Monster, Weapon and Healing Potion*. We'll need a few others, but we can start with these.

What's the difference between a class and an object? A class is basically a blank form, or blueprint, for an object. It defines the object, but it isn't the object. Think back to the days of playing Dungeons & Dragons when you had character sheets on paper. The blank character sheet has spaces for your character's name, class, level, experience points, attributes, armor, weapons, inventory, skills, spells, etc. But, until you filled in those spaces, it was just an outline – **a class.**

Once you filled in your character's information, you had a character – **an object.** Thinking of physical things, a blueprint for a house is a class (it says what the house will be like). Once you have builders follow the blueprint, with wood, steel, concrete, etc., you'll have a house (the object).

Summary

There's an old programmer's saying:

"You aren't writing a program for the computer to read, you're writing a program for a programmer to read."

It means that your program should be easy to understand, so a future programmer (who may be you) can easier fix or change things in your program. One of the most difficult parts of programming is for the programmer to keep track of everything in their mind. With objects that mirror real-world ones, it's easier to remember what's happening – you can almost picture it.

2 Building the solution for the game

Lesson objectives

At the end of this lesson, you will know...

- ➔ How to create a program (solution) in Visual Studio
- ➔ How, and why, to have separate projects in your solution

How to build your first solution in Visual Studio

People generally use *program, application,* and *project* interchangeably. However, Visual Studio calls a program a *solution.* A *solution* is broken into smaller pieces that Visual Studio calls *projects*. The solution must have at least one project.

In these lessons, I'll use *program* to talk about the game, *solution* for all the files used to make the game, and *project* for the different groups of files within the solution.

Deciding how many projects the solution needs

NOTE:
For a large business program, you might have dozens of projects in a solution – which sometimes leads to problems with keeping track of everything.

When I create a solution, there are always at least two projects. One is for the user interface (UI), the part of the program that reads and writes to the screen. The second project is for the *logic* of the program. I do this is for a couple of reasons. First, it makes the program easier to test. We won't do it in these tutorials, but it's possible to write automated tests to make sure the program does what it's supposed to do. If you have *logic* in the user interface, this is difficult to automate.

Second, sometimes you'll want to use one of your projects in another program. One example might be the code to read from, and write to, a database. That could be useful for any program you write that needs to access a database. If the database code is in the UI project, you won't be able to use it in another project, unless it needs to use the same screens from the UI project. So you put all that database access code into a separate project, and you can copy it to your next project that uses a database.

When we write the game, you'll see how I decide what to put in the Engine project.

STEP 1 Start Visual Studio.

STEP 2 Create a new solution with a *Windows Form* project (File ➡ New Project ➡ Windows Form Application).

NOTE:
When you create the solution, Visual Studio says *New Project…*, when it really should say *New Solution…*, to use the correct term.

A Windows Form application is a user interface project, one that will display the program on the screen, wait for the user's input, and update the information on the screen. We'll name the program *SuperAdventure,* and save it in the *C:\Projects* directory.

STEP 3 Now we'll create the *logic* project. Right-click on the name of the solution in the Solution Explorer and add a new *Class Library* project named *Engine* (Add… ➡ New Project… ➡ Class Library). The project didn't need to be named *Engine*. That's just what I like to name it. However, use *Engine* when you create yours, so it will match what's in the other lessons.

NOTE:
If you are using Visual Studio Community Edition 2015, you want the project type that just says *Class Library,* **not** *Class Library (Portable)* or *Class Library (Package).*

STEP 4 When you create a Class Library project, Visual Studio automatically adds a *Class1* class. You don't need it, so right-click on it and select delete, to get rid of it.

STEP 5 When you create a Windows Form project, Visual Studio creates a *Form1* screen by default. You don't need to rename it, but I always do. Right-click on Form1.cs in the SuperAdventure project, select *Rename,* and rename it to SuperAdventure (be sure to leave the *.cs* on at the end). The form name does not need to match the project name. That's just what I chose to name it for this program.

STEP 6 For the UI project to use the code in the Engine project, we need to connect them. This is called **setting a reference.** In the SuperAdventure project, right-click on *References* ➡ *Add References…* ➡ Solution (which will show you the other projects in the solution) ➡ Engine ➡ OK. Now you can see that *Engine* is listed as a reference in SuperAdventure. This means that the SuperAdventure project will be able to use the code in the Engine project.

STEP 7 Now you can run the program. Click on the *Start* button to see what the game looks like. It's not much yet, but you have to start somewhere.

Summary

Now, everything is ready to start writing the program. The solution has two projects – one for the UI, the other for the code (Engine) to do all the calculations and logic for the game.

3 Building the first screen

Lesson objectives

At the end of this lesson, you will know...

- ➔ How to add things (buttons, labels, etc.) to the user interface
- ➔ How the user interface connects with the logic in the program

Now we have a running program, but it doesn't do anything yet. Let's change that.

The first thing we need to do is make the form big enough to hold everything we're going to show on it.

Preparing the form

STEP 1 Start Visual Studio.

STEP 2 Open the SuperAdventure solution.

STEP 3 In the Solution Explorer (upper-right corner of Visual Studio), double-click on the SuperAdventure.cs file, to open it.

STEP 4 Select the whole form by clicking on it in the main editing window of Visual Studio (the center part of the screen). You'll know it's selected when you see the dashed line, with little boxes, along its edge. You'll also see *SuperAdventure* listed in the Properties part of Visual Studio, in the lower-right corner.

Properties are the small pieces of information about the form – its name, size, color, etc. We're only going to work with a few of those properties in these lessons.

STEP 5 Scroll through the Properties list to find the *Text* property. Change its value from *Form1* to *My Game*. Notice that the top line of the form has changed to *My Game* now.

STEP 6 Scroll through the properties list to find *Size,* and expand it by clicking on the plus sign (+) to its left (if it isn't already expanded). Change the *Width* to 735, and the *Height* to 690. Notice that the size of the form has changed.

Adding text (words) to the form

Now that the form is big enough to place everything we need on it, let's add the first few pieces of information.

If you remember from the earlier lesson, the finished game has the player's hit points, gold, experience, and level displayed in the upper-left corner of the form. That information is displayed with things called *controls.*

Controls are the things you see on a program's, or a website's, screen – text, boxes to enter values, buttons. In Visual Studio, when you're working with a form, you'll see a list of controls on the left side. We're going to use some of the common controls.

The first controls we're going to add are *labels***.** These are used when you want to display a small piece of text.

To add a label to a form, click on *Label,* in the controls list (left-hand side of the screen), and drag it over onto the form. You can put it anywhere, since we are going to manually change its location in its properties.

You'll need to add eight labels to the screen, and change their properties.

Here's a list of the values you need to change. It's easiest if you change each one, right after you add it to the form.

NOTE:
For the labels that will hold values that never change (for example, *Hit Points:*), leave them with whatever name they had when they were created. They don't really need a better name, since we are never going to connect to them from the *logic* part of the program.

UI control positions

Type	Name	Text	Location (X)	Location (Y)
Label	don't change	Hit Points:	18	20
Label	don't change	Gold:	18	46
Label	don't change	Experience:	18	74
Label	don't change	Level:	18	100
Label	lblHitPoints		110	19
Label	lblGold		110	45
Label	lblExperience		110	73
Label	lblLevel		110	99

The labels that we will change need a good name. You'll see why in a minute.
You probably also noticed that I added *lbl* to the front of each of the labels that I gave a name to. This is so it's easier for me to find all my labels, when writing code in the logic part of the program (again, you'll see why in a minute).

Add a button, to do something

Let's see how the program can change the labels we have. In a future lesson, we'll fill in the labels with names that start with *lbl* with the values for the player's hit points, experience, etc. For now, we'll just do a little test.

STEP 1 From the controls list on the left-hand side, drag a *button* onto an empty space on the form.

STEP 2 Change its *Name* property to *btnTest* (I use the *btn* prefix for all my buttons). Change its text property to *Test*.

STEP 3 Double-click on the btnTest button.

This will take you to the *logic* code for this form. You'll see that the cursor is in the middle of something named *btnTest_Click*. We'll take about this more later, but for now, you just need to know that this is the logic that will run when the player clicks on the btnTest button.

STEP 4 In the middle of the *btnTest_click* section, type this line:

```
lblGold.Text = "123";
```

If you noticed, when you typed *lbl,* Visual Studio showed you all the labels we created earlier the start with *lbl*. That's why I name them that way – it's easy to find them when writing the logic code.

Remember how you cleared out the *Text* property, after creating the label? Now the logic code is going to fill it in. In this case, it's filling it in with *123*. The semi-colon, at the end of the line you just wrote? That's there to let the computer know you're done with the line – kind of like a period/full stop at the end of a sentence.

STEP 5 Start the program.

Notice that the only things you see on the form are the labels that had a value in the *Text* property.

Now, click the *Test* button. See that the *lblGold,* which was just empty, now shows *123,* the value the logic code said to display when the player clicks the *Test* button.

That's a quick introduction to how the code we're going to write is going to connect with the form on the screen. The player will do something to one of the controls on the form (such as clicking a button). Then, the logic code will figure out what happens to the player's character in the game world, and updates the screen.

Summary

The game has a button! And the button does something!

OK, so it's not all that exciting, but you need some way for the player to interact with the *brains* of the program, and this is the first step.

After we write the logic in the Engine project, we'll come back to the UI and add the remaining controls, so you can start playing the game.

4 Creating the Player class and its properties

Lesson objectives

At the end of this lesson, you will know...

- ➔ How to detrmine the properties for a class
- ➔ How to select the correct datatype for each property
- ➔ How to create a class
- ➔ Two different ways to store property values

What are properties? The properties of a class, or object, are the pieces of information about it that can be different in each object, or could change while the program is running. In an RPG, that would be things such as the amount of gold, experience points, items in inventory, etc. We need a place to hold those values for the object, and the ability to change those values.

What are datatypes? The **datatype** is just a geeky way to answer the question, *"What kind of information are we going to store in this property, or variable?"*

NOTE:

Adding together strings that contain numbers in them acts differently from adding together numeric datatypes.

For example, if you try to add together the integers 1 and 2, you'll get an answer of 3. But when you add the strings "1" and "2", you get 12. This is called **concatenating strings.**

To start out, we're only going to look at two datatypes: **integers** and **strings.** An integer, if you remember back to math class, is a whole number – one that doesn't have any decimal value. This is what we need for the player's experience points, gold, and level. A string is letters, miscellaneous characters (such as, @#$%^&*), or numbers treated as letters.

We don't need any strings for our Player class, but we will have them in other classes. If you want to save a number to a string property or variable, it needs to be in quotes ("1", instead of 1), or you need to convert it from a number to a string (we'll see that a little later).

Once you declare the datatype for a property or variable, you can only store that type of data in it. So, you can't say that ExperiencePoints has an integer datatype, then try to save a string in it.

The properties of the Player class

Our Player class is going to be very simple. The only things we care about it (for now) are:

- The player's current hit points (in case they've lost any during battle)
- The player's maximum hit points (when they are fully healed)
- The amount of gold the player has
- The amount of experience points the player has
- The player's current level

The datatype for all these properties will be *integer,* since we only need to store whole numbers in them. In a future lesson, we'll add their items in an inventory and the quests they have.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 In the solution explorer, right-click on the *Engine* project. Select *Add,* then *Class.* Change the name of the class from *Class1.cs* to *Player.cs* (be sure to keep the original *.cs* at the end, but don't add a second *.cs*). Press the *Add* button. Now you are editing the code for the Player class.

STEP 3 Change the line where it says `class Player` to `public class Player`. We'll discuss why in a minute.

STEP 4 Move the cursor right after the first open *curly brace* ({) following `Player` and press the enter key to make some space to add properties.

NOTE:
To speed up the process of making properties, you can type **prop** at the start of the line and press the TAB key twice. This will create the default property code. Change the name and datatype (if necessary) of each property.

STEP 5 Type in these lines:

```
public int CurrentHitPoints { get; set; }
public int MaximumHitPoints { get; set; }
public int Gold { get; set; }
public int ExperiencePoints { get; set; }
public int Level { get; set; }
```

Notice that we don't have a space between the words *current, hit,* and *points* (along with the MaximumHitPoints and ExperiencePoints). You can't have a space in the middle of a property, or variable name. So, we use upper-case letters to show where each word starts. This isn't a requirement of C#, but it is how many programmers write property names. It's known as the **Pascal case** style. There's also the **Camel case** style, but I'll leave that one for you to discover on your own.

NOTE:
In C#, matching upper and lower case is important. A variable, or property, named *test* is different from one named *Test.*

What did you just do?

OK, you've typed in the code, but what does all of it mean? First, we'll start with *public.* This is known as the **scope.** It signifies what other parts of the solution can see the class or property. **Public** means that it is visible throughout the whole solution. This way, the UI project will be able to create a Player object and read the values of its properties. We'll talk about some of the other scope options in later lessons.

Next, what's happening on lines 11 through 15?

These are the properties of the Player class. They're public, since we need to have them visible everywhere. The *int* says that their datatype is an integer. Then we have the name of the property. Finally, there is a **{ get; set; }** after each property. Those signify that we have a property where we can *get* a value (read what is stored in the property) and *set* a value (store a value in the property). We'll see these in action soon.

What's on lines 1 through 5? That is a bunch of stuff we don't need for this class. But, when you create a new class in Visual Studio, it adds that in automatically, in case you will need it. For now, don't worry about it.

Summary

Now you've created the Player class. Once your program creates (instantiates) a Player object, it will be able to read and change the values of the object's properties.

5 Creating objects from classes

Lesson objectives

At the end of this lesson, you will know...

- ➔ How your program can create objects (instantiation)
- ➔ How to assign values to an object's properties
- ➔ How to use (reference) classes in different projects
- ➔ How to display the values from an object's properties on the UI

Instantiating your first object

Now that we have the Player class created, we can create an **instance** of the class – a Player object. This is how we go from just having the outline (the class) to having a Player (object) that the game can use.

STEP 1 Start Visual Studio, and open the solution.

Before we do anything else, let's remove the test button we added in the previous lesson. In the SuperAdventure UI project, double-click on SuperAdventure.cs to see the form. Click on the *Test* button once, to select it. You'll see that it has dashes around it. Press the *Delete* key to remove it.

In the Solution Explorer, right-click on SuperAdventure.cs and select *View Code,* to see the code for this form. Delete these lines:

```
private void btnTest_Click(object sender, EventArgs e)
{
    lblGold.Text = "123";
}
```

Now we are ready to create our Player variable.

STEP 2 We're going to use this one Player object in several places in the SuperAdventure screen. So, we need a place to store it. That's exactly what a variable does – stores a value so we can retrieve it later. In this case, we need to create a **class-level** variable. That means it can be seen by everything in the class.

You create a class level variable by having it inside your class (in this case, the SuperAdventure form – which is a class), but outside of any functions or methods.

In SuperAdventure.cs, find the line

```
public partial class SuperAdventure : Form
```

and place your cursor behind the first open curly brace ({) that follows. Press the enter key to create a new line. Here, type in

```
private Player _player;
```

So now, the code for your class will look like this:

```
public partial class SuperAdventure : Form
{
    private Player _player;

    public SuperAdventure()
    {
        InitializeComponent();
    }
}
```

Just like with the properties we created for the Player class, the variable has three parts:

- First is the *scope*. The scope here is *private,* since we don't need anything outside of this screen to use the variable.
- The *datatype* is *Player,* because we want to store a Player object in it.
- And the *name* is *_player.* You could name it anything, and it doesn't need to start with an underscore. That's just how I like to name my private variables.

Notice that Player has a red squiggly line under it. That means there is a problem. In this case, it's because the SuperAdventure form doesn't know where the Player class is located.

STEP 3 Go to line 10 (where all the lines above start with *using*), press the enter key, then type in this line:

```
using Engine;
```

The beginning of the class will look like this:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

using Engine;
```

The red line disappears, because the SuperAdventure form knows to find the Player class in the Engine project. You need to do this for each class, when it uses classes from a different projects.

Now we have a place to store the Player object, although we haven't created one yet.

STEP 4 Just under our newly created Player variable you'll notice some code that looks like this:

```
public SuperAdventure()
{
    InitializeComponent();
}
```

This is called the **constructor** for the class (the form, like everything in .Net, is a class). The code in the constructor gets called when we create a new object of the class – in this case, the form. This is where we'll instantiate a Player object and store it in the *_player* variable.

Place your cursor at the end of *InitializeComponent();* and press the enter key twice. You don't really need to do it twice, but it's nice to have some blank space in your code.

Here, type in

```
_player = new Player();
```

NOTE:
When you see a single equal sign in C#, it's for **assignment** – it's assigning the result of whatever happens on the right side of the equal sign to the property or variable on the left side. We'll get to what you use to compare if two values are equal to each other later on.

This creates a new Player object (that's what is happening on the right side of the equal sign). Then it assigns that object to the _player variable we created in step 2.

Now that we have a Player object, we can start working with it – setting its properties and reading them.

STEP 5 Add a blank line after *_player = new Player();* then add these lines after it:

```
_player.CurrentHitPoints = 10;
_player.MaximumHitPoints = 10;
_player.Gold = 20;
_player.ExperiencePoints = 0;
_player.Level = 1;
```

Your constructor will look like this:

```
public SuperAdventure()
{
    InitializeComponent();

    _player = new Player();

    _player.CurrentHitPoints = 10;
    _player.MaximumHitPoints = 10;
    _player.Gold = 20;
    _player.ExperiencePoints = 0;
    _player.Level = 1;
}
```

On these lines, we're assigning values to the properties of the _player object.

STEP 6 Add a blank line behind *_player.Level = 1;* then add these lines after it:

```
lblHitPoints.Text = _player.CurrentHitPoints.ToString();
lblGold.Text = _player.Gold.ToString();
lblExperience.Text = _player.ExperiencePoints.ToString();
lblLevel.Text = _player.Level.ToString();
```

Now, you should have this:

```
public SuperAdventure()
{
    InitializeComponent();

    _player = new Player();

    _player.CurrentHitPoints = 10;
    _player.MaximumHitPoints = 10;
    _player.Gold = 20;
    _player.ExperiencePoints = 0;
    _player.Level = 1;

    lblHitPoints.Text = _player.CurrentHitPoints.ToString();
    lblGold.Text = _player.Gold.ToString();
    lblExperience.Text = _player.ExperiencePoints.ToString();
    lblLevel.Text = _player.Level.ToString();
}
```

In these lines, we're getting the values of the properties from the _player object, and assigning them to the text of the labels on the screen.

Remember how I mentioned earlier that you cannot assign an integer value to a string, or vice versa? Since the Text property is a string, and the CurrentHitPoints, Gold, ExperiencePoints, and Level properties are all integers, we need to add the *ToString()* at the end of them. This is a common way to convert numbers to strings.

STEP 7 Save the code and Start the program. Now you should see the values we assigned to the _player object properties showing up on the screen. If you want to, you can stop the program, change the values for the _player properties, re-start the program, and see how they change.

Summary

Now you can create a variable, instantiate an object from a class in a different project, assign the object to the variable, change the values on the object's properties, and display the changes on the UI. That's a huge part of the foundation of writing programs in any object-oriented language.

6 Creating the remaining classes

Lesson objectives

At the end of this lesson, you will know...

➔ Nothing new. However, you do need to create the rest of the classes, so you can eventually play the game.

Time to add more classes

Now we can create the rest of the classes you'll use in the game. By the way, some programmers would call these classes the **business classes,** or **business objects.** Even though this is a game, and not a business application, that's a term you might hear.

The code for each class is on the next three pages.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Right-click on the *Engine* project, in Solution Explorer, then select *Add,* and then *Class*. The classes we need to create are *HealingPotion, Item, Location, Monster, Quest* and *Weapon*. Enter the name of the class you want to create, and add the properties in the class – just like you did with the Player class.

It may not be obvious what all the properties will be used for, but we'll get to that in a later lesson.

When you're done

Make sure that your Engine project has all of these classes, and that you didn't miss any.

- HealingPotion
- Item
- Location
- Monster
- Player (already created in a previous lesson)
- Quest
- Weapon

Summary

There wasn't really anything new in this lesson. We just need to add these classes to make the game actually do something.

HealingPotion.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public class HealingPotion
10      {
11          public int ID { get; set; }
12          public string Name { get; set; }
13          public string NamePlural { get; set; }
14          public int AmountToHeal { get; set; }
15      }
16  }
```

Item.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public class Item
10      {
11          public int ID { get; set; }
12          public string Name { get; set; }
13          public string NamePlural { get; set; }
14      }
15  }
```

Location.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public class Location
10      {
11          public int ID { get; set; }
12          public string Name { get; set; }
13          public string Description { get; set; }
14      }
15  }
```

continued on next page

Monster.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public class Monster
10      {
11          public int ID { get; set; }
12          public string Name { get; set; }
13          public int MaximumHitPoints { get; set; }
14          public int CurrentHitPoints { get; set; }
15          public int MaximumDamage { get; set; }
16          public int RewardExperiencePoints { get; set; }
17          public int RewardGold { get; set; }
18      }
19  }
```

Quest.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public class Quest
10      {
11          public int ID { get; set; }
12          public string Name { get; set; }
13          public string Description { get; set; }
14          public int RewardExperiencePoints { get; set; }
15          public int RewardGold { get; set; }
16      }
17  }
```

continued on next page

Weapon.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Engine
8  {
9      public class Weapon
10     {
11         public int ID { get; set; }
12         public string Name { get; set; }
13         public string NamePlural { get; set; }
14         public int MinimumDamage { get; set; }
15         public int MaximumDamage { get; set; }
16     }
17 }
```

7 Inheritance and base classes

Lesson objectives

At the end of this lesson, you will know...

- ➔ Why you might use inheritance
- ➔ How to use inheritance and base classes

There is a lot of repetition in the class properties

Right now, we have seven classes in the Engine project: *HealingPotion, Item, Location, Monster, Player, Quest, and Weapon*. You may have noticed that some of the classes have very similar properties. Some of the classes also represent the same type of thing.

For instance:

- HealingPotion, Item, and Weapon all have the properties *ID, Name, and NamePlural*. They all also represent items that a player may collect during a game.
- Player and Monster share *ID, CurrentHitPoints, and MximumHitPoints*. They are both *living creatures* in the game.

When you have classes that represent the same type of thing, and have similar properties (or functions, as we'll see later) that mean the same thing, you may want to use **inheritance** – another aspect of object-oriented programming.

We're going to make the Item class the **base class** for HealingPotion and Weapon, since all of those classes have the properties that are in the Item class. Then, we're going to create a new LivingCreature base class, to hold the shared CurrentHitPoints and MaximumHitPoints properties of the Monster and Player classes.

Here is how we do that.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Double-click on the HealingPotion class, in the Engine project. Find the line where it says `public class HealingPotion` and add `: Item` to the end of the line.

You should end up with this:

```
public class HealingPotion : Item
```

Adding the colon and Item class name after the HealingPotion class name is how we show that HealingPotion has a base class of Item. So, all the public properties and methods from the Item class now automatically show up in the HealingPotion class, even after we delete the lines for the ID, Name, and NamePlural properties.

```
public class HealingPotion : Item
{
    public int AmountToHeal { get; set; }
}
```

HealingPotion is now a **child class**, or **derived class**, from the Item class. Those are the common terms you'll hear other programmers use, for classes that inherit from another class.

STEP 3 Do the same thing that you just did to the HealingPotion class to the Weapon class – add the Item base class and delete the lines with the three properties that are already in the Item class.

```
public class Weapon : Item
{
    public int MinimumDamage { get; set; }
    public int MaximumDamage { get; set; }
}
```

STEP 4 We don't already have a class with the properties we have in common with the Monster and Player classes, so create a new class, named *LivingCreature*, with the integer properties for CurrentHitPoints and MaximumHitPoints. Be sure to make this class *public*. A base class needs to be at least as *visible* (with regards to its scope) as its child classes.

```
public class LivingCreature
{
    public int CurrentHitPoints { get; set; }
    public int MaximumHitPoints { get; set; }
}
```

STEP 5 Change the Monster and Player classes so they have LivingCreature as a base class (add the colon and LivingCreature to the lines that start with *public class*), and remove the CurrentHitPoints and MaximumHitPoints properties from those two classes – since they'll now use the properties from the base class.

```
public class Monster : LivingCreature
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int MaximumDamage { get; set; }
    public int RewardExperiencePoints { get; set; }
    public int RewardGold { get; set; }
}


public class Player : LivingCreature
{
    public int Gold { get; set; }
    public int ExperiencePoints { get; set; }
    public int Level { get; set; }
}
```

STEP 6 In the UI project, view the code of the SuperAdventure form (right-click on SuperAdventure.cs). Notice that there is no red line under the Player CurrentHitPoints and MaximumHitPoints properties – even though you just deleted those properties from the Player class. That's because the class inherited those properties from its base class – LivingCreature.

STEP 7 Start the program, so you can see that the CurrentHitPoints value is still correctly displayed on the screen.

Summary

Now you know how, and why, you would create base classes for the classes in your program. This is especially helpful when you write larger applications, and you want to make sure that classes that are similar all act in a similar manner.

Base classes also let you share functions (Lesson 09), so you don't have to duplicate the same code in several places – and worry about accidentally mistyping something in one of them.

There are some more advanced conditions to what is visible from the base class, in the *child* classes. But we won't get into those in this guide.

8 Setting properties with a class constructor

Lesson objectives

At the end of this lesson, you will know...

- ➔ What a class constructor is
- ➔ How to use a class constructor to set the properties of an object, during the object's instantiation
- ➔ How a child (derived) class can call the constructor of its base class

Let's clean up the code a little

Right now, in the SuperAdventure class, the code instantiates a Player object, then has five lines to set the properties of the player object. That works fine, but we can make that a little cleaner by doing all that in one line of code. We do this by creating a new **class constructor.**

Every class starts with a default constructor. It's what is run when you call *new Player()*. It builds and returns a new Player object. However, we can make our own constructors that do more than this default action. So, we are going to make constructors that accept the values we want to store in the properties.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 To see how this works, we're going to create a location object in our UI code (for a while). In the SuperAdventure project, right-click on SuperAdventure.cs and select *View Code*.

STEP 3 Place your cursor at the end of *InitializeComponent();* and press the enter key twice to give us some space. Then, add this code:

```
Location location = new Location();
location.ID = 1;
location.Name = "Home";
location.Description = "This is your house.";
```

Here, we create a new Location object, and save it to the variable *location* (notice that the variable name is all lower-case, to make it different from the class). Then we assign values to the properties of that object.

Your SuperAdventure constructor should look like this (the three dots represent the code we added in lesson 5, steps 5 and 6):

```
public SuperAdventure()
{
    InitializeComponent();

    Location location = new Location();
    location.ID = 1;
    location.Name = "Home";
    location.Description = "This is your house.";
    ...
}
```

STEP 4 Double-click on the Location class, in the Engine project, to start working with it. Go to the end of line *public string Description { get; set; }*, hit enter twice and add this code:

```
public Location(int id, string name, string description)
{
    ID = id;
    Name = name;
    Description = description;
}
```

This is our new constructor code for the Location class.

It starts out with *public,* since we want to be able to create a new Location object from anywhere in the solution. Then it has the name of the class. After that, we see three *parameters* between the parentheses.

When you run a function, or method (like a constructor), it can accept values. These are called **parameters**. In C#, you need to declare the datatype of the parameter and its name. In this constructor, there are three parameters: *id* (an integer), *name* (a string), and *description* (another string).

Notice that the names of the parameters match the names of the properties, except they are all lower-case. By having a different case from the property, it's more obvious when you're working with the property and when you're working with the parameter value. They don't need to match the property name, but I do that to make it obvious what the parameters are going to be used for.

So, now when you call the Location constructor, you'll *need* to pass in these three values.

Between the curly braces, we assign the values of the parameters to the properties in the class. Notice that the parameters (lower-case) are all on the right of the equal sign, and the properties (mixed-case) are on the left. The value on the right of the equal sign gets assigned to the property, or variable, on the left.

STEP 5 Go back to the SuperAdventure code and you'll see there is a red line under the *new Location()* code. That's because we aren't using the default constructor in the Location class anymore – we have our own *custom* constructor that *wants* three parameters.

If you remove the *new Location(),* and re-type it, you'll see that as soon as you type the open parentheses (( ) there is a tooltip that shows you the **method signature** (the parameters you need to pass to this method/constructor). So, let's add them:

```
Location location = new Location(1, "Home", "This is your house.");
```

Now, when you construct a new Location object, you can pass in the parameters you want assigned to the properties. This makes your code a little shorter/cleaner. It is also a way of double-checking that you've set all the properties. If you manually set each one, like before, it's easy to forget one. But with constructor that accepts parameters, you'll know exactly how many have to be passed to it. If you forget one, you'll see that red line.

You can now delete the individual property assignments for *location* we created in step 3: *location.ID, location.Name* and *location.Description*.

STEP 6 Double-click on the Quest class, in the Engine project, to start editing it. Add the following lines after the properties:

```
public Quest(int id, string name, string description,
    int rewardExperiencePoints, int rewardGold)
{
    ID = id;
    Name = name;
    Description = description;
    RewardExperiencePoints = rewardExperiencePoints;
    RewardGold = rewardGold;
}
```

Now the Quest class, too, has a custom constructor that accepts values to use to set its properties.

NOTE:

The parameters in the Quest class constructor are on two lines. I did this becuase the line was getting too long for the page. If you did the same in Visual Studio, it wouldn't be a problem. Since the last character of the first line is a comma, the computer knows that the next line is a continuation of the first line.

In C#, the end of a line is signified by a semi-colon (;), a closing parentheses ( )), or a closing curly brace (}).

Summary

You're going to see custom constructors in many C# projects. It's an easy way to ensure that your class has all needed properties set. You can also create one to do some other special tasks, which we'll see a bit later.

8.1 Using class constructors with derived classes

Lesson objectives

At the end of this lesson, you will know...

➔ How a child (derived) class calls the constructor of its base class

Let's clean up the code a little

Adding a constructor to a base class is a little more involved than adding one to a class that doesn't have any derived (child) classes.

When you create an object from a derived class, it uses the constructor for both that class and its base class. So, if you add a custom constructor to your base class, you need to have a way for the derived class to pass values to it.

If you think about this, it makes sense why this needs to be done. With a custom constructor, you are saying, *"I need these values before I can create this object."* When you create an object from a derived class, the base class still needs to have those values. So, you have to pass them to the derived class' constructor, which passes them on to the base class' constructor.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Double-click on the Item class, in the Engine project, to start editing it. After the properties, add these lines, to make your constructor:

NOTE:
If you try to build the solution right now, you'll see there are errors with the HealingPotion and Weapon classes. These are the classes that are derived from the Item class, and (right now) they don't have any values to pass to the Item class' new constructor.

```
public Item(int id, string name, string namePlural)
{
    ID = id;
    Name = name;
    NamePlural = namePlural;
}
```

STEP 3 Open the HealingPotion class, and add these lines after the properties, to make the new constructor:

```
public HealingPotion(int id, string name, string namePlural,
    int amountToHeal) : base(id, name, namePlural)
{
    AmountToHeal = amountToHeal;
}
```

Notice that the HealingPotion constructor has parameters for the one property it has (AmountToHeal) and the three properties it uses from the base class (ID, Name, and NamePlural).

Inside the constructor, it sets the AmountToHeal property to the value that was passed through the amountToHeal parameter.

Also notice that after the constructor has a list of its parameters, it has this piece of code:

```
: base(id, name, namePlural)
```

What that does is take the values from the parameters in the HealingPotion constructor (id, name, and namePlural) and passes them on to the constructor of the Item class. This is how we get parameters into the base class, when instantiating a derived class.

STEP 4 Edit the Weapon class, by adding this constructor code:

```
public Weapon(int id, string name, string namePlural,
    int minimumDamage, int maximumDamage) :
        base(id, name, namePlural)
{
    MinimumDamage = minimumDamage;
    MaximumDamage = maximumDamage;
}
```

After you add this, you can try rebuilding the solution. Now, since the derived classes are passing the required values to the base class, it will build without any errors.

STEP 5 Edit the LivingCreature class. Add this constructor code:

```
public LivingCreature(int currentHitPoints,
    int maximumHitPoints)
{
    CurrentHitPoints = currentHitPoints;
    MaximumHitPoints = maximumHitPoints;
}
```

Again, if you try to build the solution now, you'll get an error that the Monster and Player classes (the classes that derive from LivingCreature) have a problem.

STEP 6 Edit the Monster class, by inserting this constructor:

```
public Monster(int id, string name, int maximumDamage,
    int rewardExperiencePoints, int rewardGold,
        int currentHitPoints, int maximumHitPoints) :
            base(currentHitPoints, maximumHitPoints)
{
    ID = id;
    Name = name;
    MaximumDamage = maximumDamage;
    RewardExperiencePoints = rewardExperiencePoints;
    RewardGold = rewardGold;
}
```

STEP 7 Insert this constructor code into the Player class:

```
public Player(int currentHitPoints, int maximumHitPoints,
    int gold, int experiencePoints, int level) :
        base(currentHitPoints, maximumHitPoints)
{
    Gold = gold;
    ExperiencePoints = experiencePoints;
    Level = level;
}
```

Now, if we try building the solution, we get an error. That's because in the SuperAdventure UI code, we are instantiating a Player object without any parameters. Now that we have a custom constructor in the Player class, we need to use that and pass in the appropriate parameters.

STEP 8 Right-click on SuperAdventure.cs, in the SuperAdventure project, then select *View Code*. Find the line *_player = new Player();* and change it to this:

```
_player = new Player(10, 10, 20, 0, 1);
```

Now that we are passing all the values in the constructor, to set the player object's properties, we can delete the lines, where we were setting the properties individually.

You could leave those individual property assignments, they'll still work, and they won't hurt anything. But you're already setting the properties to those values, so you might as well remove them and make your code a little cleaner.

Summary

You'll probably need to use some amount of inheritance in any decent-sized program. And now you know how to make a custom constructor work in classes that use inheritance.

This may be overkill for the simple game that we're building. However, you're going to need to know how inheritance works if you program in C#, or another object-oriented programming language. Hopefully, with small, simple classes like these, it's easy to understand.

9 Using your classes as datatypes

Lesson objectives

At the end of this lesson, you will know...

➔ How to create a property in a class that uses one of your other classes as its datatype

Adding properties with custom datatypes

Currently, all of the properties in our classes are either strings or integers. But sometimes we need to store something more complex in a property.

For example, in the Location class, we want to possibly store an item required to enter the location (such as a key). If there is a quest available at that location, we need to store that value. We also might have a monster that exists in that location. So we need to add some new properties to the Location class, to store these values.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Edit the Location class, in the Engine project. Add these new properties:

```
public Item ItemRequiredToEnter { get; set; }
public Quest QuestAvailableHere { get; set; }
public Monster MonsterLivingHere { get; set; }
public Location LocationToNorth { get; set; }
public Location LocationToEast { get; set; }
public Location LocationToSouth { get; set; }
public Location LocationToWest { get; set; }
```

Now we have properties to hold these values for each location. Because we need to store an Item object in the ItemRequiredToEnter property, its datatype is Item. Just like using a string datatype when we want to store some text in a property.

STEP 3 Now, let's add parameters to the constructor to accept some of these values and save them to the properties. Change the constructor code to this:

```
public Location(int id, string name, string description,
Item itemRequiredToEnter = null,
    Quest questAvailableHere = null,
        Monster monsterLivingHere = null)
{
    ID = id;
    Name = name;
    Description = description;
    ItemRequiredToEnter = itemRequiredToEnter;
    QuestAvailableHere = questAvailableHere;
    MonsterLivingHere = monsterLivingHere;
}
```

Notice that after each of the new parameters, there is *=null* . Some locations won't have an item required to enter them, a quest available at them, or a monster living there. This lets us call the Location constructor without passing these three values. The constructor will know to use the default values, which, in this case, is the *null.* So, both of these lines would work the same way:

```
Location test1 = new Location(1, "Your House",
    "This is your house");
Location test2 = new Location(1, "Your House",
    "This is your house", null, null, null);
```

STEP 4 Edit the Quest class, in the Engine project. Add this one property:

```
public Item RewardItem { get; set; }
```

The RewardItem property will store what item the player receives when they complete the quest.

Summary

Now you can create properties that hold more complex values. This way, you'll be able to do things such as get the name of the monster at the location, without needing to do extra lookups somewhere outside of your object.

10 Creating collections of objects

Lesson objectives

At the end of this lesson, you will know...

➔ How to store groups of objects in one property or variable, and read them later

How to handle groups of objects

You may have noticed that there are properties missing from some of our classes. When outlining the program, we mentioned that the player will be able to have items in their inventory. The player will have quests. Quests have items that need to be turned in, to complete them. Monsters have items that can be looted from them.

We have a Quest class, and an Item class, but we don't have any place to store them in the Player or Monster classes. We also need a way to store more than one item, since a Player will probably have more than one item in their inventory and could have more than one quest.

We do this with lists, or collections.

But before we create these list properties, we're going to need a place to store some additional information for the items and quests. For example, with the items in the Player's inventory, we need to store the quantity of each items they have. It's similar to what we'll need to do with the Quest class – store the quantity of each item that is needed to complete the quest.

For the player's quests, we also need a place to record whether or not the player has completed the quest.

In the Monster class, we want a list of the items that can be *looted* from the monster. This is called the **loot table.** We need to store the item, the percentage of times it is *dropped,* and if the item is a default loot item – in case the random number generator says that none of the loot items were dropped.

There are different ways to do this, but I'm going to show you a way that uses some additional classes.

NOTE:

We use a new datatype for two of these classes – **bool.** This stands for **Boolean** and is used to store a value of *true* or *false.* For example, in the PlayerQuest class, the IsCompleted property will store a value of false, until the player finishes the quest. Then we'll store true in it.

We're going to need four new classes, so add them to the Engine project:

- InventoryItem
- PlayerQuest
- QuestCompletionItem
- LootItem

The code for each class is on the next two pages. After you've created all the classes and added the code, we'll look at adding them as *list properties* to the existing Player, Monster, and Quest classes.

InventoryItem.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public class InventoryItem
10      {
11          public Item Details { get; set; }
12          public int Quantity { get; set; }
13
14          public InventoryItem(Item details, int quantity)
15          {
16              Details = details;
17              Quantity = quantity;
18          }
19      }
20  }
```

LootItem.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public class LootItem
10      {
11          public Item Details { get; set; }
12          public int DropPercentage { get; set; }
13          public bool IsDefaultItem { get; set; }
14
15          public LootItem(Item details, int dropPercentage, bool isDefaultItem)
16          {
17              Details = details;
18              DropPercentage = dropPercentage;
19              IsDefaultItem = isDefaultItem;
20          }
21      }
22  }
```

continued on next page

PlayerQuest.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public class PlayerQuest
10      {
11          public Quest Details { get; set; }
12          public bool IsCompleted { get; set; }
13
14          public PlayerQuest(Quest details)
15          {
16              Details = details;
17              IsCompleted = false;
18          }
19      }
20  }
```

QuestCompletionItem.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public class QuestCompletionItem
10      {
11          public Item Details { get; set; }
12          public int Quantity { get; set; }
13
14          public QuestCompletionItem(Item details, int quantity)
15          {
16              Details = details;
17              Quantity = quantity;
18          }
19      }
20  }
```

Now that we have the new classes, we can create properties for them in the Player, Quest, and Monster classes.

STEP 1 Edit the Player class, by double-clicking on it in the Engine project. Look at line 2 of the Player class (in case it's missing, add it to any class that will be using lists):

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

NOTE:

We didn't need to set the string, integer, and Boolean properties to a default value because those datatypes have a built-in default value. However, Lists are *null* (non-existent), until you set them to an emplty list (a new List object, with no values in it yet).

In order to use lists – one variable or property to hold a collection of objects that are the same class – we need to have this line included. This is where your program will find everything it needs, in order to work with collections.

Add these two properties to the Player class:

```
public List<InventoryItem> Inventory { get; set; }
public List<PlayerQuest> Quests { get; set; }
```

Now you have two new properties that can hold lists containing InventoryItem and PlayerQuest objects.

Then, in the constructor code, add these new lines:

```
Inventory = new List<InventoryItem>();
Quests = new List<PlayerQuest>();
```

These two lines set the value of the new properties to empty lists. If we didn't do this, the value for those properties would be *null* – nothing. By setting them to an empty list, we can add items to them later, because you can add objects to an empty list, but you can't add object to nothing (null).

STEP 2 Edit the Quest class. Add this new property:

```
public List<QuestCompletionItem> QuestCompletionItems
    { get; set; }
```

In the constructor, add this new line, so the QuestCompletionItems list will be ready to have objects added to it:

```
QuestCompletionItems = new List<QuestCompletionItem>();
```

STEP 3 Edit the Monster class, by adding this property:

```
public List<LootItem> LootTable { get; set; }
```

And add this to the constructor, so the new property it isn't *null* :

```
LootTable = new List<LootItem>();
```

Your Player, Quest and Monster classes should now look like this:

NOTE:

We didn't create new parameters in the constructor to pass in values for these new properties. We could have, but we are going to populate them a little differently – a way that I think is a little easier for someone new to programming.

```
public class Player : LivingCreature
{
    public int Gold { get; set; }
    public int ExperiencePoints { get; set; }
    public int Level { get; set; }
    public List<InventoryItem> Inventory { get; set; }
    public List<PlayerQuest> Quests { get; set; }

    public Player(int currentHitPoints, int maximumHitPoints,
        int gold, int experiencePoints, int level) :
            base(currentHitPoints, maximumHitPoints)
    {
        Gold = gold;
        ExperiencePoints = experiencePoints;
        Level = level;

        Inventory = new List<InventoryItem>();
        Quests = new List<PlayerQuest>();
    }
}


public class Quest
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public int RewardExperiencePoints { get; set; }
    public int RewardGold { get; set; }
    public List<QuestCompletionItem> QuestCompletionItems
        { get; set; }
    public Item RewardItem { get; set; }

    public Quest(int id, string name, string description,
        int rewardExperiencePoints, int rewardGold)
    {
        ID = id;
        Name = name;
        Description = description;
        RewardExperiencePoints = rewardExperiencePoints;
        RewardGold = rewardGold;
        QuestCompletionItems =
        new List<QuestCompletionItem>();
    }
}
```

```
public class Monster : LivingCreature
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int MaximumDamage { get; set; }
    public int RewardExperiencePoints { get; set; }
    public int RewardGold { get; set; }
    public List<LootItem> LootTable { get; set; }

    public Monster(int id, string name, int maximumDamage,
        int rewardExperiencePoints, int rewardGold,
            int currentHitPoints, int maximumHitPoints) :
                base(currentHitPoints, maximumHitPoints)
    {
        ID = id;
        Name = name;
        MaximumDamage = maximumDamage;
        RewardExperiencePoints = rewardExperiencePoints;
        RewardGold = rewardGold;
        LootTable = new List<LootItem>();
    }
}
```

Summary

Almost every program I've written has included some type of list of collection. In the real world, we usually work with several items (e.g., display all the deposits into, and payments from, our bank account).

We'll show how to add to, and work with, the objects in these lists very soon.

11 Using a static class

Lesson objectives

At the end of this lesson, you will know...

- ➔ What a static class is, and how to create one
- ➔ When you want, and don't want, to use static classes

What is a static class? All the classes we've created so far have been public classes. That means they are visible in all of your program. However, in order to use one of them, you need to create an object of the class. Then you work with that object.

A static class is one that is always available to the rest of your program – you don't need to create an object from it. In fact, you can't create an object from a static class. It's just there.

A static class is a place to hold shared resources (variables and methods), since there will only be the one **instance** of it, and everything else in your program will use that one, shared set of methods and variables.

Why would you need a static class? For our game, we need to store some things that will be used in several places in the program. Things like the list of locations, the types of items, the types of monsters, etc. This information is never going to change. We're going to populate these lists at the start of the game, and never change it. We're also going to use those lists in several places in the game.

Using a static class to hold all this information is one way to make all this information available everywhere.

When else would you use a static class? Another thing you can do with a static class (and a static variable) is to hold values such as a system-wide counter. Let's say you want a program to hand out unique, sequential numbers to the people who use it. You could create a static NumberAssigner class, with a static GetNextNumber method, that keeps track of a static _nextNumber variable.

```
namespace Engine
{
    public static class NumberAssigner
    {
        static int _nextNumber = 0;

        public static int GetNextNumber()
        {
            _nextNumber = (_nextNumber + 1);

            return _nextNumber;
        }
    }
}
```

When you start the program, _nextNumber has a value of 0. When a user calls the GetNext-Number method, the code adds 1 to _nextNumber and returns the value (in this case, 1) to the program. The next time the GetNextNumber method is called, it adds 1 to _nextNumber (resulting in 2 this time) and returns 2 to the program.

What problems can happen with static classes? The problem with static methods and variables, is that sometimes you don't want a shared resource, you want each user to have their own copy of the object or variable.

The game we're creating is a single-player one. So, we don't really have a problem using static variables. However, if we were to make a UI for this game a website on the Internet, we might have several people playing it at the same time.

So, let's say we stored the player's current hit points somewhere as a static variable – CurrentHitPoints. When player A is attacked, the program would subtract their damage and change the value of CurrentHitPoints. But if a different player did something in the game (attacked a monster or healed themselves with a potion), since we only have a static, single, shared CurrentHitPoints variable, they'd be using the value from player A, and not their real current hit points value.

That's how static classes and variables can be dangerous. When you use a static variable to hold a value, make sure it's one that you really want to be shared for every user.

Populating our game world in a static class

Now that you have an understanding of static classes and variables, we're going to create a *World* class to hold lists of all the things in our game – locations, items, monsters, and quests. Since we're only going to read from it, once we do the initial population of the values, it's OK to use a static class.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Create a new class by right-clicking on the Engine project and selecting *Add,* then *Class…* . Name the class *World.cs.*

STEP 3 Copy the code for the World class from the next five pages into World.cs.

We have a lot of code to add, so be careful if you're typing it in (Visual Studio's **IntelliSense**/Autocomplete can be of great help), or when copy-pasting. Red squiggly lines usually mean there's an error present. But don't start solving those until you've entered all the code. We'll talk about what it does after.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Engine
{
    public static class World
    {
        public static readonly List<Item> Items = new List<Item>();
        public static readonly List<Monster> Monsters = new List<Monster>();
        public static readonly List<Quest> Quests = new List<Quest>();
        public static readonly List<Location> Locations = new List<Location>();

        public const int ITEM_ID_RUSTY_SWORD = 1;
        public const int ITEM_ID_RAT_TAIL = 2;
        public const int ITEM_ID_PIECE_OF_FUR = 3;
        public const int ITEM_ID_SNAKE_FANG = 4;
        public const int ITEM_ID_SNAKESKIN = 5;
        public const int ITEM_ID_CLUB = 6;
        public const int ITEM_ID_HEALING_POTION = 7;
        public const int ITEM_ID_SPIDER_FANG = 8;
        public const int ITEM_ID_SPIDER_SILK = 9;
        public const int ITEM_ID_ADVENTURER_PASS = 10;

        public const int MONSTER_ID_RAT = 1;
        public const int MONSTER_ID_SNAKE = 2;
        public const int MONSTER_ID_GIANT_SPIDER = 3;

        public const int QUEST_ID_CLEAR_ALCHEMIST_GARDEN = 1;
        public const int QUEST_ID_CLEAR_FARMERS_FIELD = 2;

        public const int LOCATION_ID_HOME = 1;
        public const int LOCATION_ID_TOWN_SQUARE = 2;
        public const int LOCATION_ID_GUARD_POST = 3;
        public const int LOCATION_ID_ALCHEMIST_HUT = 4;
        public const int LOCATION_ID_ALCHEMISTS_GARDEN = 5;
        public const int LOCATION_ID_FARMHOUSE = 6;
        public const int LOCATION_ID_FARM_FIELD = 7;
        public const int LOCATION_ID_BRIDGE = 8;
        public const int LOCATION_ID_SPIDER_FIELD = 9;

        static World()
        {
            PopulateItems();
            PopulateMonsters();
            PopulateQuests();
            PopulateLocations();
        }

```

continued on next page

```
52          private static void PopulateItems()
53          {
54              Items.Add(new Weapon(ITEM_ID_RUSTY_SWORD, "Rusty sword", "Rusty swords", 0, 5));
55              Items.Add(new Item(ITEM_ID_RAT_TAIL, "Rat tail", "Rat tails"));
56              Items.Add(new Item(ITEM_ID_PIECE_OF_FUR, "Piece of fur", "Pieces of fur"));
57              Items.Add(new Item(ITEM_ID_SNAKE_FANG, "Snake fang", "Snake fangs"));
58              Items.Add(new Item(ITEM_ID_SNAKESKIN, "Snakeskin", "Snakeskins"));
59              Items.Add(new Weapon(ITEM_ID_CLUB, "Club", "Clubs", 3, 10));
60              Items.Add(new HealingPotion(ITEM_ID_HEALING_POTION,
                    "Healing potion", "Healing potions", 5));
61              Items.Add(new Item(ITEM_ID_SPIDER_FANG,  "Spider fang", "Spider fangs"));
62              Items.Add(new Item(ITEM_ID_SPIDER_SILK,  "Spider silk", "Spider silks"));
63              Items.Add(new Item(ITEM_ID_ADVENTURER_PASS,
                    "Adventurer pass", "Adventurer passes"));
64          }
65
66          private static void PopulateMonsters()
67          {
68              Monster rat = new Monster(MONSTER_ID_RAT, "Rat", 5, 3, 10, 3, 3);
69              rat.LootTable.Add(new LootItem(ItemByID(ITEM_ID_RAT_TAIL), 75, false));
70              rat.LootTable.Add(new LootItem(ItemByID(ITEM_ID_PIECE_OF_FUR), 75, true));
71
72              Monster snake = new Monster(MONSTER_ID_SNAKE, "Snake", 5, 3, 10, 3, 3);
73              snake.LootTable.Add(new LootItem(ItemByID(ITEM_ID_SNAKE_FANG), 75, false));
74              snake.LootTable.Add(new LootItem(ItemByID(ITEM_ID_SNAKESKIN), 75, true));
75
76              Monster giantSpider = new Monster(MONSTER_ID_GIANT_SPIDER,
                    "Giant spider", 20, 5, 40, 10, 10);
77              giantSpider.LootTable.Add(new LootItem(
                    ItemByID(ITEM_ID_SPIDER_FANG), 75, true));
78              giantSpider.LootTable.Add(new LootItem(
                    ItemByID(ITEM_ID_SPIDER_SILK), 25, false));
79
80              Monsters.Add(rat);
81              Monsters.Add(snake);
82              Monsters.Add(giantSpider);
83          }
84
```

continued on next page

```
85          private static void PopulateQuests()
86          {
87              Quest clearAlchemistGarden = new Quest(QUEST_ID_CLEAR_ALCHEMIST_GARDEN,
88                  "Clear the alchemist's garden",
89                  "Kill rats in the alchemist's garden and bring back 3 rat tails.
                        You will receive a healing potion and 10 gold pieces.", 20, 10);
90
91              clearAlchemistGarden.QuestCompletionItems.Add(new QuestCompletionItem(
                        ItemByID(ITEM_ID_RAT_TAIL), 3));
92
93              clearAlchemistGarden.RewardItem = ItemByID(ITEM_ID_HEALING_POTION);
94
95              Quest clearFarmersField = new Quest(QUEST_ID_CLEAR_FARMERS_FIELD,
96                  "Clear the farmer's field",
97                  "Kill snakes in the farmer's field and bring back 3 snake fangs.
                        You will receive an adventurer's pass and 20 gold pieces.", 20, 20);
98
99              clearFarmersField.QuestCompletionItems.Add(new QuestCompletionItem(
                        ItemByID(ITEM_ID_SNAKE_FANG), 3));
100
101             clearFarmersField.RewardItem = ItemByID(ITEM_ID_ADVENTURER_PASS);
102
103             Quests.Add(clearAlchemistGarden);
104             Quests.Add(clearFarmersField);
105         }
106
107         private static void PopulateLocations()
108         {
109             // Create each location
110             Location home = new Location(LOCATION_ID_HOME, "Home",
                    "Your house. You really need to clean up the place.");
111
112             Location townSquare = new Location(LOCATION_ID_TOWN_SQUARE,
                    "Town square", "You see a fountain.");
113
114             Location alchemistHut = new Location(LOCATION_ID_ALCHEMIST_HUT,
                    "Alchemist's hut", "There are many strange plants on the shelves.");
115             alchemistHut.QuestAvailableHere = QuestByID(QUEST_ID_CLEAR_ALCHEMIST_GARDEN);
116
117             Location alchemistsGarden = new Location(LOCATION_ID_ALCHEMISTS_GARDEN,
                    "Alchemist's garden", "Many plants are growing here.");
118             alchemistsGarden.MonsterLivingHere = MonsterByID(MONSTER_ID_RAT);
119
120             Location farmhouse = new Location(LOCATION_ID_FARMHOUSE,
                    "Farmhouse", "There is a small farmhouse, with a farmer in front.");
121             farmhouse.QuestAvailableHere = QuestByID(QUEST_ID_CLEAR_FARMERS_FIELD);
122
123             Location farmersField = new Location(LOCATION_ID_FARM_FIELD,
                    "Farmer's field", "You see rows of vegetables growing here.");
124             farmersField.MonsterLivingHere = MonsterByID(MONSTER_ID_SNAKE);
125
126             Location guardPost = new Location(LOCATION_ID_GUARD_POST,
                    "Guard post", "There is a large, tough-looking guard here.",
                        ItemByID(ITEM_ID_ADVENTURER_PASS));
127
```

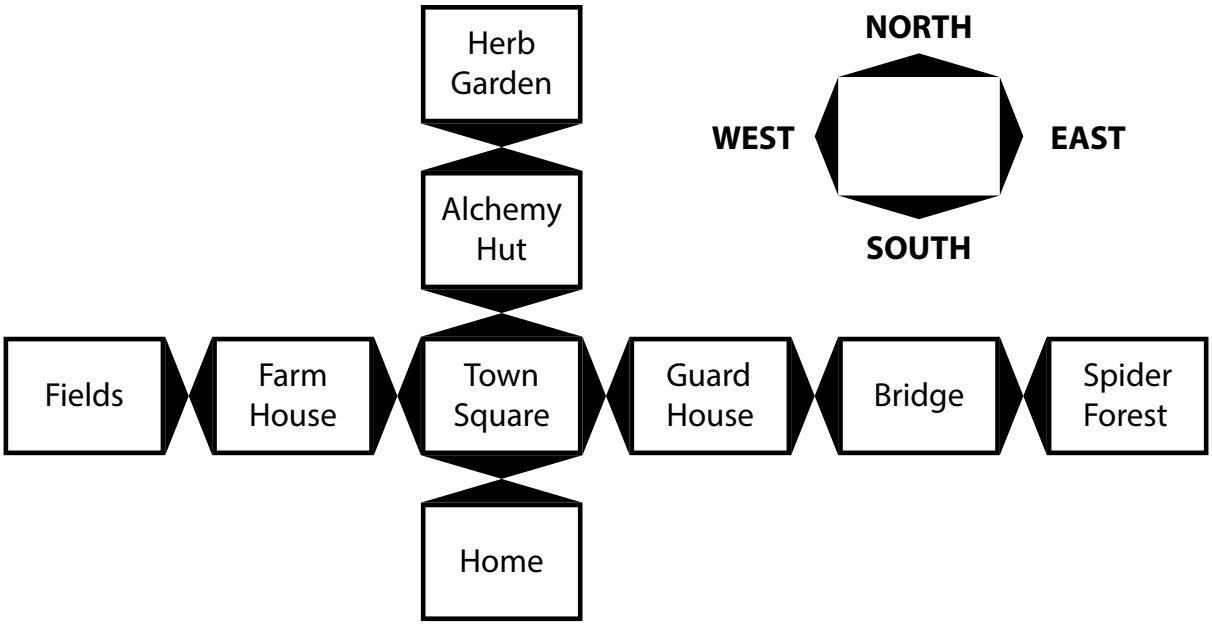
continued on next page

```
128             Location bridge = new Location(LOCATION_ID_BRIDGE,
                "Bridge", "A stone bridge crosses a wide river.");
129
130             Location spiderField = new Location(LOCATION_ID_SPIDER_FIELD,
                "Forest", "You see spider webs covering covering the trees in this forest.");
131             spiderField.MonsterLivingHere = MonsterByID(MONSTER_ID_GIANT_SPIDER);
132
133             // Link the locations together
134             home.LocationToNorth = townSquare;
135
136             townSquare.LocationToNorth = alchemistHut;
137             townSquare.LocationToSouth = home;
138             townSquare.LocationToEast = guardPost;
139             townSquare.LocationToWest = farmhouse;
140
141             farmhouse.LocationToEast = townSquare;
142             farmhouse.LocationToWest = farmersField;
143
144             farmersField.LocationToEast = farmhouse;
145
146             alchemistHut.LocationToSouth = townSquare;
147             alchemistHut.LocationToNorth = alchemistsGarden;
148
149             alchemistsGarden.LocationToSouth = alchemistHut;
150
151             guardPost.LocationToEast = bridge;
152             guardPost.LocationToWest = townSquare;
153
154             bridge.LocationToWest = guardPost;
155             bridge.LocationToEast = spiderField;
156
157             spiderField.LocationToWest = bridge;
158
159             // Add the locations to the static list
160             Locations.Add(home);
161             Locations.Add(townSquare);
162             Locations.Add(guardPost);
163             Locations.Add(alchemistHut);
164             Locations.Add(alchemistsGarden);
165             Locations.Add(farmhouse);
166             Locations.Add(farmersField);
167             Locations.Add(bridge);
168             Locations.Add(spiderField);
169         }
170
```

continued on next page

```
171         public static Item ItemByID(int id)
172         {
173             foreach(Item item in Items)
174             {
175                 if(item.ID == id)
176                 {
177                     return item;
178                 }
179             }
180 
181             return null;
182         }
183 
184         public static Monster MonsterByID(int id)
185         {
186             foreach(Monster monster in Monsters)
187             {
188                 if(monster.ID == id)
189                 {
190                     return monster;
191                 }
192             }
193 
194             return null;
195         }
196 
197         public static Quest QuestByID(int id)
198         {
199             foreach(Quest quest in Quests)
200             {
201                 if(quest.ID == id)
202                 {
203                     return quest;
204                 }
205             }
206 
207             return null;
208         }
209 
210         public static Location LocationByID(int id)
211         {
212             foreach(Location location in Locations)
213             {
214                 if(location.ID == id)
215                 {
216                     return location;
217                 }
218             }
219 
220             return null;
221         }
222     }
223 }
```

What does the code do? The purpose of the World class is to have one place to hold everything that exists in the game world. In it, we'll have things such as the monster that exist at a location, the loot items you can collect after defeating a monster, etc. It will also show how the locations connect with each other, building our game map.



Here is what is happening in the different parts of the World class.

Lines 11 – 14: Static list variables. These work similar to the properties in a class. We'll populate them with all the things in our game world, then read from them in the rest of the program.

Line 16 – 42: Constants. Constants look, and work, like variables, except for one big difference – they can never have their values changed.

We're going to use these constants, which have a sort-of English name, so we don't have to remember the numeric ID for each of the different games objects. Without them, if we wanted to create a giant spider for the player to fight, we'd need to remember that the giant spider monster has an ID value of 2. With the constants, we can say, get me a monster where the ID equals MONSTER_ID_GIANT_SPIDER.

If you don't fully understand how we'll do that, it should become clearer when we get to the lesson where we start having the player move around and we need to lookup locations, quests, monsters, and items.

Lines 44 – 50: This is the static constructor. You might be thinking, *"Wait! We can't instantiate a static class, so why does it have a constructor? After all, that's what a constructor is used for – instantiating an object!"*

With a static class, the constructor code is run the first time someone uses anything in the class. So, when we start the game and want to display information about the player's current location, and we try to get that data from the World class, the constructor method will be run, and the lists will get populated.

Inside the constructor, we call the four methods to populate the different lists. We don't need to have separate methods, and we could have put all the code from lines 52 through 169 into the constructor. But breaking them up makes them easier to read and work with.

Lines 52 – 169: These are the methods we use to create the game objects and add them to the static lists.

By calling the Add() method on a list variable or property, we add an object to that list.

Look at line 54. Here we are adding a new Weapon object to the Items list. When we call *new Weapon()*, the constructor of the Weapon class returns a Weapon object with the parameters passed in. Since that's all inside *Items.Add()*, that object gets added to the Items list.

You might hear that called **inlining,** since we did multiple things (created the value and added it to the list), all in one line.

On line 68, we create a new Monster object and save it to the variable *rat*. On lines 69 and 70, we add items to the (list) property of PotentialLootItems that you might find on the rat. Then, on line 80, we add the rat variable to the static Monsters list.

Lines 171 – 221: These methods are ones we can call to get values from the static lists. We could access the lists from lines 11 through 14 directly, since they are public. But these **wrapper methods** make it a little clearer exactly what we want to do.

We pass in the ID of the object we want to retrieve from its list (by using the constants from lines 16 through 42). The method will look at each item in the list (using the FOREACH) and see if the ID we passed in matches the ID of the object. If so, it returns that object to us. If we get to the end of the list, and nothing matches (which should really never happen), the method returns *null* – nothing.

Summary

Now we have a populated *world* for the game. We can use the static methods from this static class at any place in the rest of our program, and get the information we need about the objects in our game world.

12 Add the remaining UI controls

Lesson objectives

At the end of this lesson, you will know...

➔ Some of the other controls you can add to a form

Now that we have the game world built, we can start connecting it up with the user interface, so the player can start moving around, battling monsters, and completing quests.

So, we'll add the rest of the buttons, for the player to perform actions, and RichTextBoxes and DataGridViews, so we can display what is happening for the player.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Right-click on the SuperAdventure.cs form, in the SuperAdventure project, then select *Open* or *View Designer*.

STEP 3 Just like when you added the labels for Hit Points and Gold, you're going to drag some controls on the form, and then change some of its properties.

Add one Label control

Use these values for it:

Label control

Text	Location (X)	Location (Y)
Select action	617	531

Add two ComboBox controls

Use these values for them:

Combobox controls

Name	Location (X)	Location (Y)
cboWeapons	369	559
cboPotions	369	593

Add six Button controls

Use these values for them:

Button controls

Name	Text	Location (X)	Location (Y)
btnUseWeapon	Use	620	559
btnUsePotion	Use	620	593
btnNorth	North	493	433
btnEast	East	573	457
btnSouth	South	493	487
btnWest	West	412	457

Add two RichTextBox controls

Use these values for them:

RichTextBox controls

Name	Location (X)	Location (Y)	Size (Width)	Size (Height)	ReadOnly
rtbLocation	347	19	360	105	True
rtbMessages	347	130	360	286	True

Add two DataGridView controls – from the *Data* group of the Toolbox

When you add the DataGridViews, you'll see a popup box with some options. Uncheck the boxes for *Enable Adding, Enable Editing,* and *Enable Deleting.*

Use these values for them:

DataGridView controls

Name	Location (X)	Location (Y)	Size (Width)	Size (Height)
dgvInventory	16	130	312	309
dgvQuests	16	446	312	189

For both the DataGridViews, you also need to set these properties:

Property	Value
ReadOnly	True
EditMode	EditProgramatically
AllowUsersToAddRows	False
AllowUsersToDeleteRows	False
AllowUsersToResizeRows	False
Enabled	False
MultiSelect	False
RowHeadersVisible	False

Summary

We've added some new control types to the game.

The ComboBoxes will let the player select which weapon, or potion, they want to use during a battle. The RichTextBoxes will display status information about where the player is at and what is happening to them. And the DataGridViews will show lists of what the player has in their inventory and what quests they have.

13 Functions, procedures, and methods

Lesson objectives

At the end of this lesson, you will know...

- ➔ What functions, procedures, and methods are
- ➔ How functions can call other functions

What are functions, procedures, and methods? These are just the names we use to call about a small piece of the program that does a particular task.

That task may be to perform a mathematical calculation on some numbers, or read some data from a file, or place an order on a website. They can perform huge tasks or small ones. However, if the task/function is large, we usually break it down into smaller tasks/functions. This makes them easier to read.

Personally, I try to keep my functions no longer than 15 – 20 lines of code – at the max. Sometimes, it makes sense to have larger ones. But it's easier to figure out what a function is doing if you can see all of it on one screen, without constantly scrolling up and down.

Nowadays, most people use these names interchangeably. In the past, programmers would often use **function** for a piece of the program that did something and returned a value. Whereas a **method** did something, but didn't return any value.

If you look at the World.cs class we created, it has eight methods – if you don't count the constructor method.

Methods with no return values

The first four are the ones that start with *Populate*. Let's take a look at the *PopulateItems()* method.

```
private static void PopulateItems()
{
    ...
}
```

First, we have the scope *private*.

```
private static void PopulateItems()
```

Private means that this method can only be run by other code inside this class. If it was **public,** then it could be run from other classes. But we don't need to allow that for this method, since it's only ever run once, from this class' constructor.

Next, we have *static*.

```
private static void PopulateItems()
```

Normally, when you want to run a method, you do it *on an object*. For example, on a *monster* object, you might have a ReceiveDamage method that gets the amount of damage that was done to the monster, and subtracts that from its CurrentHitPoints. The method is doing something *to,* or *with,* the object.

If you want to run a method without instantiating an object from the class, then the method *needs* to be static. Since this method is in a static class, which can never be instantiated, we need to include static here.

Then, there is *void*.

```
private static void PopulateItems()
```

Void means that this method is not going to return a value. It's just going to do some work. In this case, the work is to populate the Items list with the items in our game world.

Finally, we have the method name *PopulateItems*.

```
private static void PopulateItems()
```

Give your methods names that are easy to understand. Then, when you, or another programmer, want to make changes to the program in the future, it will be easier to figure out what the method is supposed to be doing.

It's also common to name methods in a **verb-noun format** – what are you doing, and what are you doing it to?

Methods that return a value

Now, let's take a look at a method that returns something, the *ItemByID()* method.

```
public static Item ItemByID(int id)
{
    ...
}
```

This time, the scope of the method is *public*.

```
public static Item ItemByID(int id)
```

We're going to need to *call* (a common way to say *run* or *execute* for a method) this method from other parts of the game. So, it needs to be public.

We still need *static,* since this class is never an object.

```
public static Item ItemByID(int id)
```

Now, instead of *void,* we have *Item.*

```
public static Item ItemByID(int id)
```

That's because this method is going to *return* a value, and the datatype of that value is going to be *Item.* So, the user knows that when they call this method, they should get an Item back.

This method also accepts parameters.

```
public static Item ItemByID(int id)
```

This is just like we had in the constructors of our other classes. We'll talk later about exactly what is happening inside the method. But, for now, you can probably guess that the method accepts an ID number and returns the item with the matching ID number.

Methods that call other methods

Looking at the code in the constructor of this class, we see that all it has in it is the name of the Populate methods in the class. That's because all this method does is run those other methods.

```
static World()
{
    PopulateItems();
    PopulateMonsters();
    PopulateQuests();
    PopulateLocations();
}
```

When the constructor is called, it *runs* the PopulateItems() method, then the PopulateMonsters(), PopulateQuests(), and PopulateLocations() methods. When you run your program, the computer is basically performing one line of the program at a time – at least, until you get into multi-threaded programming. So picture the computer working like this:

- It starts to run the World constructor
- It sees the line *PopulateItems();* and goes to the PopulateItems() method
- It goes through each line in the PopulateItems() method, adding items to the list
- When it reaches the end of the PopulateItems() method, it returns back to where it was last in the World constructor
- The next line is *PopulateMonsters();* , so the computer goes to the PopulateMonsters() method and performs each line in there
- At the end of the PopulateMonsters() method, it returns to the World method and performs the next line *PopulateQuests();*
- Etc.

So, a line of code in one method can call another method, which can contain a line of code that calls another method, which contains a line of code that calls another method, and so on…

In large programs, these method calls can get 20 layers deep, or deeper. That can make it difficult to find bugs, since you aren't really sure what happened in each layer. So, remember to keep your programs simple. Don't add another layer, just because you can.

Summary

Know you understand the basics of functions, procedures, and methods.

These are important, because you need to create them to start doing things in the program. We'll have some that handle user input, some that do let the player move around, and some that handle our game's battles. These will be the brains of our program.

13.1 Creating functions to handle user input

Lesson objectives

At the end of this lesson, you will know...

➔ How to create functions to handle button clicks in the UI

Creating functions to handle the user's input

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Right-click on the SuperAdventure.cs form, in the SuperAdventure project, then select *Open* or *View Designer*.

STEP 3 Double-click on each of the buttons on the form – the four direction buttons (North, South, East, and West) and the two *Use* buttons that will be used for combat. It doesn't matter what order you do these in.

After you double-click each button, you'll be taken to the *code* part of the screen and be placed where the new function was created. This is where you can write the code that will run when the user presses the connected button. So, when the user clicks the *North* button, this will be the code to *move* the player to the location to the North.

Something you don't see when you do this is the code that gets created to connect the buttons to the new functions. You might have noticed that the code for the UI screen says *public partial class SuperAdventure : Form*. The word **partial** is important here. There is another part of this class that you don't normally see. This is the part that connects the controls (buttons, comboboxes, etc) in the UI to the code we've been looking at.

We haven't looked at the other part of this class, because we're not focusing on how to build the UI in these guides. However, it's something you need to be aware of if you're creating Windows Form programs.

There are also other things you can handle from the UI. These are called **events.** In this case, the only event we care about is when the button click. However, there are dozens of other events that can be handled. For example, many controls have an *OnMouseOver* event – when the user moves the mouse over the control. For that event, you might do something such as change the background color of the control – so the user knows where the focus is.

Summary

Now you can create functions to handle the common user actions in the UI – in this case, when the user clicks on a button. Right now, the functions don't do anything. So our next lesson will be to start writing our functions, to let the player play the game.

14 Variables

Lesson objectives

At the end of this lesson, you will know...

- ➔ How to create/declare variables
- ➔ Where variables are visible

Before we start adding the game logic to the program, we should spend a little time going over some of the things you'll see in the code. The first thing we'll talk about is variables.

We've already done a little bit with them, but there are some important things you need to be aware of, when creating and using variables.

Here's the code for this lesson. This class doesn't really do anything useful, but it shows how to use variables.

```
 1  namespace Workbench
 2  {
 3      public class Variables
 4      {
 5          // Class-level variables
 6          public int classVariable1 = 3;  // Visible to other classes
 7          private int classVariable2 = 7; // Only visible to this class
 8          private int classVariable3; // Declaring the variable, without
                                        setting a value
 9
10          public void Function1()
11          {
12              int functionVariable1;
13
14              if(classVariable1 < 5)
15              {
16                  functionVariable1 = 1;
17
18                  int innerVariable1;
19              }
20
21              if(classVariable1 >= 5)
22              {
23                  functionVariable1 = 2;
24
25                  int innerVariable1;
26              }
27          }
28
```

```
29          public void Fuction2()
30          {
31              int functionVariable1;
32
33              if(classVariable2 < 5)
34              {
35                  functionVariable1 = 3;
36
37                  int innerVariable1;
38              }
39
40              if(classVariable2 >= 5)
41              {
42                  functionVariable1 = 4;
43
44                  int innerVariable1;
45              }
46          }
47      }
48  }
```

One trick you can use to figure out where a variable is visible is to look at the curly braces it is between, when it's *declared*. Generally, the variable is visible anywhere between those curly braces.

Lines 6-8: These are *class-level variables*. They can be used anywhere in this class.

The variables on line 6 through 8 are declared inside the curly braces for the class (lines 4 and 47). So, they're visible everywhere between lines 4 and 47 – the whole class.

When we declare the variables on lines 6 and 7, we also set them to a value. We don't do that on line 8. So, classVariable3 has the default value – which for *int*s is 0.

Different datatypes have different default values. And if you create a variable whose datatype is one of your classes, it has a value of *null* – nothing/empty.

classVariable1 (Line 6) can also be used outside of this class, since it is declared public. So, if another class makes an object of the Variables class, it will be able to change the value of classVariable1.

Since classVariable2 and classVariable3 are private, the other class wouldn't be able to directly change the values of those variables. However, the other class could call one of the public functions (Function1 or Function2). Since those functions are inside the class, they can modify the private class-level variables (see lines 16, 23, 35, and 42).

Lines 10-27: *functionVariable1,* on line 12, is declared inside the curly braces for Function1() (lines 11 and 27). So, it's visible everywhere inside that function.

innerVariable1 (line 18) is declared between the curly braces on lines 15 and 19. So it's only visible in between those lines. That's why we can declare another variable with the same name on line 25, inside another completely separate set of curly braces.

We couldn't declare another *functionVarable1* on line 18, since we have a functionVariable1 that is already visible at that level – the one declared on line 12.

Lines 29-46: Notice that we are able to declare a second *functionVariable1* variable on line 31, even though it has the same name as the variable declared on line 12.

We can do that because the *functionVariable1* on line 12 only exists between lines 11 and 27.

General variable declaration rules

It's a good habit to declare your variables at the *lowest level* possible. You could declare all your variables as class-level variables, and then change them in any function of your class. However, that can make it difficult to track down any problems you might have in the future. Since your variable can be changed in lots of places, you won't necessarily know where it was changed.

Another extremely important rule with variables is to give them *meaningful names.* Many programs have long lives. If you come back in six months, or six years, you want to be able to understand what your program is doing. When your class, functions, properties, and variables have clear names, it's much easier to understand why they exist, and what they're supposed to do.

Summary

Now you know where to declare your variables, and where your variables will be visible.

14.1 IF, ELSE, ELSE IF statements

Lesson objectives

At the end of this lesson, you will know...

- ➔ How to use IF, ELSE and ELSE IF statements
- ➔ How to use AND (&&) and OR (||) operators

One of the most common things you'll do in a program is use an **IF statement.** If some condition is true, then you want a certain section of code to run. If it isn't true, you don't want that section of code to run.

For example, during a battle, if the player's hit points drops to zero, you'll want to end the battle. If the player has more than zero hit points, you'll want them to be able to continue fighting. You can also create IF statements that are more complex than just checking one value. Or, you may want to be able to handle more than two conditions for a value.

Here are code samples, and description for four typical ways you'll use an IF statement.

The simple IF statement

```
public bool IsPlayerStillAlive1(int currentHitPoints)
{
    if(currentHitPoints > 0)
    {
        return true;
    }

    return false;
}
```

Here, we check if the passed-in *currentHitPoints* is greater than zero. If it is, the computer will run the code between the curly braces after the IF – in this case, lines 4 through 6. When it reaches line 5, it will stop processing the rest of the code in this function and return a value of *true.*

If *currentHitPoints* is equal, or less than, zero, the computer will continue going through this function. In this case, it will get to line 8 and return *false* to whatever called this function.

IF with an ELSE

```
public bool IsPlayerStillAlive2(int currentHitPoints)
{
    if(currentHitPoints > 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

This function does the same thing as the previous one. However, we explicitly have an ELSE statement. So, if the expression between the parentheses of the IF statement is true, the *return true;* code will be run. Else, the code between the next set of curly braces will be run, *return false;* .

This is a simple function, with just one line in each branch of the IF statement (the *true* branch and the *false* branch). However, you could have more lines of code between the curly braces – as you'll see in the lessons when we add the code to play the game.

Complex IF conditions

NOTE:
With the IF condition on the right, I included lots of parentheses, so you'd know exactly how the comparison logic should be calculated. Be nice to yourself, or the next programmer who will look at your code, and do the same.

```
public bool IsPlayerStillAlive3(int currentHitPoints,
    bool hasResurrectionRing)
{
    if((currentHitPoints > 0) || ((currentHitPoints == 0) &&
        hasResurrectionRing))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

You can check more than one condition in an IF statement. You can also make it so all the conditions must be true (using ANDs), or, at least one of the conditions must be true (using ORs). We'll talk more about AND and OR in a moment.

This is a place where some people get a little confused – especially when the code has many conditions that it's checking. An important thing to remember is to look for parentheses around the conditions, and work your way from the inner-most sets of parentheses, to the outermost – just like in Algebra class (if you remember that).

Let's say that the function on the previous page is called with these parameters:

currentHitPoints = 5
hasResurrectionRing = false

If we replace the variables in the parentheses with the values, we end up with this:

```
if((5 > 0) || ((5 == 0) && false))
```

Which becomes:

```
if((true) || ((false) && false))
```

Now we can deal with the new symbols `&&` and `||`. These are used when you want to work with Boolean values (true, false):

- && means AND
- || means OR

With `&&`, the value on both sides of it needs to be true, for the whole comparison to be true.

Now we can reduce the `((false) && false))` to `(false)`, since the values on both sides are not equal to true. That gives us this:

```
if((true) || (false))
```

With `||`, if the value on either side of it is true, then the whole statement is true. If you look at the original IF statement in English, you'd say it like this:

"IF the player's current hit points is greater than zero, OR the player's current hit points is zero AND they have a resurrection ring, then do this…"

Since we have a true on one side of the `||` (OR), the whole thing evaluates to true. So, for these parameter values, the code between the first set of curly braces would be run.

If you passed in these parameters, the program would follow the steps below to eventually figure out to run the ELSE code:

currentHitPoints = 0
hasResurrectionRing = false

Which make the IF look like this:

```
if((0 > 0) || ((0 == 0) && false))
```

To:

```
if((false) || ((true) && false))
```

To:

```
if((false) || (false))
```

To:

```
if(false)
```

IF with ELSE IF

Sometimes you want to do multiple checks against something.

Here's one way you might look at the player's current experience points to determine their level:

```
public int ComputePlayerLevel(int experiencePoints)
{
    if(experiencePoints < 100)
    {
        return 1; // Player is level 1
    }
    else if(experiencePoints < 250)
    {
        return 2; // Player is level 2
    }
    else if(experiencePoints < 500)
    {
        return 3; // Player is level 3
    }
    else if(experiencePoints < 1000)
    {
        return 4; // Player is level 4
    }

    return 5; // The maximum level is 5, in this sample
}
```

In this case, if the experiencePoints parameter is less than 100, the function will return a value of one. However, if it's 100 or more, we still want to do some more comparisons in the ELSE. So, we use an ELSE IF to connect our next IF statement.

In this function, if the player had 672 experiencePoints, it would be false for the first IF (where it checks to see if the value is less than 100), and go to the ELSE IF statement. Since 672 is not less than 250, it would go to the next ELSE IF. There, it's also not less than 500, so it goes to the next ELSE IF. Finally, 672 is less than 1000, so the program runs the code to return a value of 4.

In case none of the IF conditions were true, the program would have continued goi ng through the function until it hit the line to return 5.

Summary

Now you see how you can run, or not run, sections of code based on conditions in the program.

When we add in the source code for the game logic (moving the player around to different locations, battling monsters, completing quests, etc.) you'll see several IF statements in action.

14.2 FOREACH loops

Lesson objectives

At the end of this lesson, you will know...

➔ How to use a FOREACH loop to work with the objects inside lists

Another common thing you'll need to do is look at the objects contained in a list. Maybe you'll need to total up some values, or see if you have a certain object in the list. This is where you can use a FOREACH loop. Think of it as, *"I want to do 'something' for each object in this list."*

Using a FOREACH to see if an item exists in a list

In the World class, we already have some functions with a FOREACH.

Here's the code for the ItemByID() function that finds an item, based on its ID:

```
public static Item ItemByID(int id)
{
    foreach(Item item in Items)
    {
        if(item.ID == id)
        {
            return item;
        }
    }

    return null;
}
```

We already have the Items list created as a class-level variable and populated from the constructor. This function will find the item with the same ID as the parameter that was passed in, and return it to the part of the program that called this function.

Here is what's happening in the parentheses after the *foreach*:

- The *in Items,* at the end, is where you say which list you want to look through. In this case, it's the *Items* list.
- The *Item* is the datatype of the objects in this list.
- And the *item* is the variable that will hold each object from the list, when the code between the curly braces is run.

So, the general pattern for a FOREACH is:

```
foreach(DataType variableName in listName)
```

When this code runs, it will take the first object from the *Items* list, and assign it to the *item* variable.

Then it will check to see if the *ID* property of the current *item* variable matches the *id* that was passed in as a parameter to the function. If it does match, the function returns that item object. If it doesn't match, the foreach will get the next object from the list, assign it to the *item* variable, and then check it.

NOTE:
Returning null is generally not a good idea. The code that called this function is expecting an *Item* object returned, and now it also needs to handle receiving a null.

This is one of those things we're doing to keep it simple for these tutorials that you probably wouldn't do in a *real* program.

If the function goes through all the item objects in the Items list, and none of them match, the FOREACH loop will finish and the rest of the code in the function will be run.

In this case, the only code in the rest of the function is *return null* (nothing/empty).

Briefly about LINQ

.Net Framework has something called **LINQ** that also works with lists. It lets you perform the same logic, but often in a single line of code. However, for some people, it takes a little time to figure out how to work with it.

Here's a function to show you how LINQ can be much shorter than using a foreach.

```
public void CalculateAverage()
{
    // Create and populate the list of numbers
    List<double> values = new List<double>();

    values.Add(1);
    values.Add(5);
    values.Add(21);

    // Calculating the average, using a foreach
    double total = 0;
    double counter = 0;

    foreach(double value in values)
    {
        total = (total + value);
        counter = (counter + 1);
    }

    double average = (total / counter);

    // Calculating the average, using LINQ
    double linqAverage = values.Average();
}
```

In the first few lines, we create a list of **doubles** (double-precision numbers, that can have decimal values) and add values to it.

Next, we calculate the average value with a FOREACH.

To do this, we need to have some variables to hold the total of the values in the list and to see how many items are in the list. Then we calculate the average by dividing the total by the number of values (for this sample, I'm not worrying about the list being empty, which would result in a **divide by zero** error).

That code takes up 10 lines in the function.

The final line is how you can get the average value from a list, using LINQ. It's one line.

This is a simple demo of one LINQ function. They can get more complex, which is why I'm not covering them in this tutorial.

Summary

Now you can work with the lists we have in the game – the player's inventory and quest lists, and the list of locations, items, monsters, and quests in the World class.

15 Getting random numbers for the game

Lesson objectives

At the end of this lesson, you will know...

➔ How to create random numbers – the easy way, and the good way

This game, like many games, needs to get random numbers. We'll use random numbers to determine how much damage a player does to a monster, how much damage the monster does to the player, and which items from the monster loot table the player will receive after defeating a monster.

However, there is something important that you need to consider when using random numbers.

The .Net framework has a built-in random number generating class, called *Random*. Unfortunately, it doesn't produce numbers that are really random. In a simple game like this, that's not a big problem. But if you were serious about building a good game, you'd want to use a better technique.

The problem with better technique is that it uses some complex objects and logic to generate the random numbers. These are way more complex than I can describe in a tutorial for beginners. In fact, some of the things in it are difficult for advanced programmers to understand. So, if you want to use the better version, you'll just have to take my word that it works.

Both versions are on the next two pages. You can select either one (but just one!) to use in your version of the game.

Creating a random number generator

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Right-click on the Engine project and add a new class named RandomNumberGenerator.cs.

STEP 3 Decide which method you want to use, and copy it into the RandomNumberGenerator class.

Summary

Now you can create random numbers in the game.

We only use random numbers in a few places, in this simple version of the game. However, if you want to add more features, you may find this useful.

RandomNumberGenerator.cs – The code for the easy way

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Linq;
 4  using System.Text;
 5  using System.Threading.Tasks;
 6
 7  namespace Engine
 8  {
 9      public static class RandomNumberGenerator
10      {
11          private static Random _generator = new Random();
12
13          public static int NumberBetween(int minimumValue, int maximumValue)
14          {
15              return _generator.Next(minimumValue, maximumValue + 1);
16          }
17      }
18  }
```

In the simple version, we create a Random object (_generator) and use its Next() method to get a random value between the minimum and maximum values passed in as parameter.

continued on next page

RandomNumberGenerator.cs – The code for the *more* random way

```
 1  using System;
 2  using System.Security.Cryptography;
 3
 4  namespace Engine
 5  {
 6      // This is the more complex version
 7      public static class RandomNumberGenerator
 8      {
 9          private static readonly RNGCryptoServiceProvider _generator =
                new RNGCryptoServiceProvider();
10
11          public static int NumberBetween(int minimumValue, int maximumValue)
12          {
13              byte[] randomNumber = new byte[1];
14
15              _generator.GetBytes(randomNumber);
16
17              double asciiValueOfRandomCharacter = Convert.ToDouble(randomNumber[0]);
18
19              // We are using Math.Max, and substracting 0.00000000001,
20              // to ensure "multiplier" will always be between 0.0 and .99999999999
21              // Otherwise, it's possible for it to be "1", which causes problems in our
                   rounding.
22              double multiplier =
                    Math.Max(0, (asciiValueOfRandomCharacter / 255d) - 0.00000000001d);
23
24              // We need to add one to the range, to allow for the rounding done with
                   Math.Floor
25              int range = maximumValue - minimumValue + 1;
26
27              double randomValueInRange = Math.Floor(multiplier * range);
28
29              return (int)(minimumValue + randomValueInRange);
30          }
31      }
32  }
```

In this version, we use an instance of an encryption class RNGCryptoServiceProvider. This class is better at not following a pattern when it creates random numbers. But we need to do some more math to get a value between the passed in parameters.

16 Writing the function to move the player

Lesson objectives

At the end of this lesson, you will know...

- ➔ How to plan a function
- ➔ How to write the code/logic for a function
- ➔ Some of the most common C# commands

Ok, we've spent all this time building up the pieces we need for our game. Now it's time to actually get the game to do something. The first thing we're going to add the code to handle when the player moves to a new location.

By the way, this is a long lesson, and covers a lot of new things. So make sure you have some time to complete it.

Outline of the function

The first thing we need to do is figure out what's going to happen during a move.

When I write a function that needs to do a lot of things, like this one, I like to do some planning first – so I don't miss anything.

When the player moves to a new location, we'll completely heal them, we'll check to see if the location has any quests (and if the player can complete them), and if there are any monsters to fight there.

Here is an outline of the logic for a player move function. Something like this is often called **pseudo-code.** It isn't C# code, but it represents what the code will do. Each indentation level is where we handle a different condition – for example, the steps to follow if the location has a monster, and the steps to follow if the location doesn't have a monster.

- If the location has an item required to enter it
  - If the player does not have the item
    - Display message
    - Don't let the player move here (stop processing the move)
- Update the player's current location
  - Display location name and description
  - Show/hide the available movement buttons
- Completely heal the player (we assume they rested/healed while moving)
  - Update hit points display in UI
- Does the location have a quest?
  - If so, does the player already have the quest?
    - If so, is the quest already completed?
      - If not, does the player have the items to complete the quest?
        - If so, complete the quest
          - Display messages
          - Remove quest completion items from inventory
          - Give quest rewards
          - Mark player's quest as completed
    - If not, give the player the quest
      - Display message
      - Add quest to player quest list
- Is there a monster at the location?
  - If so,
    - Display message
    - Spawn new monster to fight
    - Display combat comboboxes and buttons
      - Repopulate comboboxes, in case inventory changed
  - If not
    - Hide combat comboboxes and buttons
- Refresh the player's inventory in the UI – in case it changed
- Refresh the player's quest list in the UI – in case it changed
- Refresh the cboWeapons ComboBox in the UI
- Refresh the cboPotions ComboBox in the UI

Creating a shared function

We have four different functions for movement, one for each direction. And we need to do the steps in the pseudo-code for moving in each direction. We could do that by writing the same code in each function. But then, if we ever want to change that logic, we'd need to make the change in four places – and that often leads to mistakes.

So, we're going to create a new shared *MoveTo()* function to handle movement to any location. Each of the four movement functions will call that one new function.

Storing the player's location

We also need a place to save the player's current location. Since this value will change, we need to store it in either a variable or a property. In this case, we'll make a property in the Player class. It makes sense to do this because a player's location is a *property* (in the general sense) of the player.

Storing the current monster

We also need a place to store the current monster that the player is fighting, in case they move to a location that has a monster there.

In the World class, we have a list of all the monsters in the game. However, we can't use the monsters from there to fight against. We only have one *instance* of each monster. So, if we fought against the rat in the World.Monsters list, and killed it, the next time we fight against it, it would already be dead.

When we move to a new location, if it has a monster there, we'll create a new instance of that type of monster and save it to a variable. Then, the player will fight against that instance of the monster.

Now that we know what we want to accomplish, we're ready to add the code.

Creating the functions to move the player

STEP 1 Start Visual Studio, and open the solution.

STEP 2 First, let's create the new property to store the player's current location.

NOTE:
You don't have to put the properties in any specific order. I just like to like to keep my List properties at the end of the properties. I find it a little easier to read when they're grouped in the same place in every class.

Double-click on the Player class, in the Engine project. Add a new property named CurrentLocation, with a datatype of Location.

```
public Location CurrentLocation { get; set; }
```

STEP 3 Right-click on the SuperAdventure.cs form, in the SuperAdventure project, then select *View Code*. Overwrite your existing code with the code on the next nine pages.

We have a lot of code to add, so be careful if you're typing it in (Visual Studio's **IntelliSense**/Autocomplete can be of great help), or when copy-pasting. Red squiggly lines usually mean there's an error present. But don't start solving those until you've entered all the code. We'll talk about what it does after.

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.ComponentModel;
 4  using System.Data;
 5  using System.Drawing;
 6  using System.Linq;
 7  using System.Text;
 8  using System.Threading.Tasks;
 9  using System.Windows.Forms;
10
11  using Engine;
12
13  namespace SuperAdventure
14  {
15      public partial class SuperAdventure : Form
16      {
17          private Player _player;
18          private Monster _currentMonster;
19
20          public SuperAdventure()
21          {
22              InitializeComponent();
23
24              _player = new Player(10, 10, 20, 0, 1);
25              MoveTo(World.LocationByID(World.LOCATION_ID_HOME));
26              _player.Inventory.Add(new InventoryItem(
                    World.ItemByID(World.ITEM_ID_RUSTY_SWORD), 1));
27
28              lblHitPoints.Text = _player.CurrentHitPoints.ToString();
29              lblGold.Text = _player.Gold.ToString();
30              lblExperience.Text = _player.ExperiencePoints.ToString();
31              lblLevel.Text = _player.Level.ToString();
32          }
33
34          private void btnNorth_Click(object sender, EventArgs e)
35          {
36              MoveTo(_player.CurrentLocation.LocationToNorth);
37          }
38
39          private void btnEast_Click(object sender, EventArgs e)
40          {
41              MoveTo(_player.CurrentLocation.LocationToEast);
42          }
43
44          private void btnSouth_Click(object sender, EventArgs e)
45          {
46              MoveTo(_player.CurrentLocation.LocationToSouth);
47          }
48
49          private void btnWest_Click(object sender, EventArgs e)
50          {
51              MoveTo(_player.CurrentLocation.LocationToWest);
52          }
53
```

continued on next page

```
54          private void MoveTo(Location newLocation)
55          {
56              //Does the location have any required items
57              if(newLocation.ItemRequiredToEnter != null)
58              {
59                  // See if the player has the required item in their inventory
60                  bool playerHasRequiredItem = false;
61
62                  foreach(InventoryItem ii in _player.Inventory)
63                  {
64                      if(ii.Details.ID == newLocation.ItemRequiredToEnter.ID)
65                      {
66                          // We found the required item
67                          playerHasRequiredItem = true;
68                          break; // Exit out of the foreach loop
69                      }
70                  }
71
72                  if(!playerHasRequiredItem)
73                  {
74                      // We didn't find the required item in their inventory,
                           so display a message and stop trying to move
75                      rtbMessages.Text += "You must have a " +
                        newLocation.ItemRequiredToEnter.Name +
                        " to enter this location." + Environment.NewLine;
76                      return;
77                  }
78              }
79
80              // Update the player's current location
81              _player.CurrentLocation = newLocation;
82
83              // Show/hide available movement buttons
84              btnNorth.Visible = (newLocation.LocationToNorth != null);
85              btnEast.Visible = (newLocation.LocationToEast != null);
86              btnSouth.Visible = (newLocation.LocationToSouth != null);
87              btnWest.Visible = (newLocation.LocationToWest != null);
88
89              // Display current location name and description
90              rtbLocation.Text = newLocation.Name + Environment.NewLine;
91              rtbLocation.Text += newLocation.Description + Environment.NewLine;
92
93              // Completely heal the player
94              _player.CurrentHitPoints = _player.MaximumHitPoints;
95
96              // Update Hit Points in UI
97              lblHitPoints.Text = _player.CurrentHitPoints.ToString();
98
```

continued on next page

```
 99              // Does the location have a quest?
100              if(newLocation.QuestAvailableHere != null)
101              {
102                  // See if the player already has the quest, and if they've completed it
103                  bool playerAlreadyHasQuest = false;
104                  bool playerAlreadyCompletedQuest = false;
105
106                  foreach(PlayerQuest playerQuest in _player.Quests)
107                  {
108                      if(playerQuest.Details.ID == newLocation.QuestAvailableHere.ID)
109                      {
110                          playerAlreadyHasQuest = true;
111
112                          if(playerQuest.IsCompleted)
113                          {
114                              playerAlreadyCompletedQuest = true;
115                          }
116                      }
117                  }
118
119                  // See if the player already has the quest
120                  if(playerAlreadyHasQuest)
121                  {
122                      // If the player has not completed the quest yet
123                      if(!playerAlreadyCompletedQuest)
124                      {
125                          // See if the player has all the items needed to complete the quest
126                          bool playerHasAllItemsToCompleteQuest = true;
127
128                          foreach(QuestCompletionItem qci in
                                newLocation.QuestAvailableHere.QuestCompletionItems)
129                          {
130                              bool foundItemInPlayersInventory = false;
131
132                              // Check each item in the player's inventory, to see
                                 if they have it, and enough of it
133                              foreach(InventoryItem ii in _player.Inventory)
134                              {
135                                  // The player has this item in their inventory
136                                  if(ii.Details.ID == qci.Details.ID)
137                                  {
138                                      foundItemInPlayersInventory = true;
139
```

continued on next page

```
140                                 if(ii.Quantity < qci.Quantity)
141                                 {
142                                     // The player does not have enough of this item
                                           to complete the quest
143                                     playerHasAllItemsToCompleteQuest = false;
144
145                                     // There is no reason to continue checking
                                           for the other quest completion items
146                                     break;
147                                 }
148
149                                 // We found the item, so don't check
                                       the rest of the player's inventory
150                                 break;
151                             }
152                         }
153
154                         // If we didn't find the required item, set our variable and
                               stop looking for other items
155                         if(!foundItemInPlayersInventory)
156                         {
157                             // The player does not have this item in their inventory
158                             playerHasAllItemsToCompleteQuest = false;
159
160                             // There is no reason to continue checking for the other
                                   quest completion items
161                             break;
162                         }
163                     }
164
165                     // The player has all items required to complete the quest
166                     if(playerHasAllItemsToCompleteQuest)
167                     {
168                         // Display message
169                         rtbMessages.Text += Environment.NewLine;
170                         rtbMessages.Text += "You complete the " +
                                newLocation.QuestAvailableHere.Name +
                                    " quest." + Environment.NewLine;
171
172                         // Remove quest items from inventory
173                         foreach(QuestCompletionItem qci in
                                newLocation.QuestAvailableHere.QuestCompletionItems)
174                         {
175                             foreach(InventoryItem ii in _player.Inventory)
176                             {
177                                 if(ii.Details.ID == qci.Details.ID)
178                                 {
179                                     // Subtract the quantity from the player's
                                           inventory that was needed to complete the quest
180                                     ii.Quantity -= qci.Quantity;
181                                     break;
182                                 }
183                             }
184                         }
```

continued on next page

```
185
186                         // Give quest rewards
187                         rtbMessages.Text += "You receive: " + Environment.NewLine;
188                         rtbMessages.Text +=
                        newLocation.QuestAvailableHere.RewardExperiencePoints.ToString() +
                            " experience points" + Environment.NewLine;
189                         rtbMessages.Text +=
                                newLocation.QuestAvailableHere.RewardGold.ToString() +
                                " gold" + Environment.NewLine;
190                         rtbMessages.Text +=
                                newLocation.QuestAvailableHere.RewardItem.Name +
                                    Environment.NewLine;
191                         rtbMessages.Text += Environment.NewLine;
192
193                         _player.ExperiencePoints +=
                                newLocation.QuestAvailableHere.RewardExperiencePoints;
194                         _player.Gold += newLocation.QuestAvailableHere.RewardGold;
195
196                         // Add the reward item to the player's inventory
197                         bool addedItemToPlayerInventory = false;
198
199                         foreach(InventoryItem ii in _player.Inventory)
200                         {
201                             if(ii.Details.ID ==
                                    newLocation.QuestAvailableHere.RewardItem.ID)
202                             {
203                                 // They have the item in their inventory,
                                       so increase the quantity by one
204                                 ii.Quantity++;
205
206                                 addedItemToPlayerInventory = true;
207
208                                 break;
209                             }
210                         }
211
212                         // They didn't have the item, so add it to their inventory,
                            with a quantity of 1
213                         if(!addedItemToPlayerInventory)
214                         {
215                             _player.Inventory.Add(new InventoryItem(
                                    newLocation.QuestAvailableHere.RewardItem, 1));
216                         }
217
```

continued on next page

```
218                             // Mark the quest as completed
219                             // Find the quest in the player's quest list
220                             foreach(PlayerQuest pq in _player.Quests)
221                             {
222                                 if(pq.Details.ID == newLocation.QuestAvailableHere.ID)
223                                 {
224                                     // Mark it as completed
225                                     pq.IsCompleted = true;
226 
227                                     break;
228                                 }
229                             }
230                         }
231                     }
232                 }
233                 else
234                 {
235                     // The player does not already have the quest
236 
237                     // Display the messages
238                     rtbMessages.Text += "You receive the " +
                            newLocation.QuestAvailableHere.Name +
                            " quest." + Environment.NewLine;
239                     rtbMessages.Text += newLocation.QuestAvailableHere.Description +
                            Environment.NewLine;
240                     rtbMessages.Text += "To complete it, return with:" +
                            Environment.NewLine;
241                     foreach(QuestCompletionItem qci in
                            newLocation.QuestAvailableHere.QuestCompletionItems)
242                     {
243                         if(qci.Quantity == 1)
244                         {
245                             rtbMessages.Text += qci.Quantity.ToString() + " " +
                                    qci.Details.Name + Environment.NewLine;
246                         }
247                         else
248                         {
249                             rtbMessages.Text += qci.Quantity.ToString() + " " +
                                    qci.Details.NamePlural + Environment.NewLine;
250                         }
251                     }
252                     rtbMessages.Text += Environment.NewLine;
253 
254                     // Add the quest to the player's quest list
255                     _player.Quests.Add(new PlayerQuest(newLocation.QuestAvailableHere));
256                 }
257             }
258 
```

continued on next page

```
259             // Does the location have a monster?
260             if(newLocation.MonsterLivingHere != null)
261             {
262                 rtbMessages.Text += "You see a " + newLocation.MonsterLivingHere.Name +
                        Environment.NewLine;
263
264                 // Make a new monster, using the values from the standard monster
                       in the World.Monster list
265                 Monster standardMonster = World.MonsterByID(
                        newLocation.MonsterLivingHere.ID);
266
267                 _currentMonster = new Monster(standardMonster.ID, standardMonster.Name,
268                     standardMonster.MaximumDamage, standardMonster.RewardExperiencePoints,
                            standardMonster.RewardGold, standardMonster.CurrentHitPoints,
                                standardMonster.MaximumHitPoints);
269
270                 foreach(LootItem lootItem in standardMonster.LootTable)
271                 {
272                     _currentMonster.LootTable.Add(lootItem);
273                 }
274
275                 cboWeapons.Visible = true;
276                 cboPotions.Visible = true;
277                 btnUseWeapon.Visible = true;
278                 btnUsePotion.Visible = true;
279             }
280             else
281             {
282                 _currentMonster = null;
283
284                 cboWeapons.Visible = false;
285                 cboPotions.Visible = false;
286                 btnUseWeapon.Visible = false;
287                 btnUsePotion.Visible = false;
288             }
289
290             // Refresh player's inventory list
291             dgvInventory.RowHeadersVisible = false;
292
293             dgvInventory.ColumnCount = 2;
294             dgvInventory.Columns[0].Name = "Name";
295             dgvInventory.Columns[0].Width = 197;
296             dgvInventory.Columns[1].Name = "Quantity";
297
298             dgvInventory.Rows.Clear();
299
300             foreach(InventoryItem inventoryItem in _player.Inventory)
301             {
302                 if(inventoryItem.Quantity > 0)
303                 {
304                     dgvInventory.Rows.Add(new[] { inventoryItem.Details.Name,
                            inventoryItem.Quantity.ToString() });
305                 }
306             }
307
```

continued on next page

```
308             // Refresh player's quest list
309             dgvQuests.RowHeadersVisible = false;
310
311             dgvQuests.ColumnCount = 2;
312             dgvQuests.Columns[0].Name = "Name";
313             dgvQuests.Columns[0].Width = 197;
314             dgvQuests.Columns[1].Name = "Done?";
315
316             dgvQuests.Rows.Clear();
317
318             foreach(PlayerQuest playerQuest in _player.Quests)
319             {
320                 dgvQuests.Rows.Add(new[] { playerQuest.Details.Name,
                        playerQuest.IsCompleted.ToString() });
321             }
322
323             // Refresh player's weapons combobox
324             List<Weapon> weapons = new List<Weapon>();
325
326             foreach(InventoryItem inventoryItem in _player.Inventory)
327             {
328                 if(inventoryItem.Details is Weapon)
329                 {
330                     if(inventoryItem.Quantity > 0)
331                     {
332                         weapons.Add((Weapon)inventoryItem.Details);
333                     }
334                 }
335             }
336
337             if(weapons.Count == 0)
338             {
339                 // The player doesn't have any weapons,
                       so hide the weapon combobox and the "Use" button
340                 cboWeapons.Visible = false;
341                 btnUseWeapon.Visible = false;
342             }
343             else
344             {
345                 cboWeapons.DataSource = weapons;
346                 cboWeapons.DisplayMember = "Name";
347                 cboWeapons.ValueMember = "ID";
348
349                 cboWeapons.SelectedIndex = 0;
350             }
351
```

continued on next page

```
352             // Refresh player's potions combobox
353             List<HealingPotion> healingPotions = new List<HealingPotion>();
354
355             foreach(InventoryItem inventoryItem in _player.Inventory)
356             {
357                 if(inventoryItem.Details is HealingPotion)
358                 {
359                     if(inventoryItem.Quantity > 0)
360                     {
361                         healingPotions.Add((HealingPotion)inventoryItem.Details);
362                     }
363                 }
364             }
365
366             if(healingPotions.Count == 0)
367             {
368                 // The player doesn't have any potions, so hide the potion combobox and
                       the "Use" button
369                 cboPotions.Visible = false;
370                 btnUsePotion.Visible = false;
371             }
372             else
373             {
374                 cboPotions.DataSource = healingPotions;
375                 cboPotions.DisplayMember = "Name";
376                 cboPotions.ValueMember = "ID";
377
378                 cboPotions.SelectedIndex = 0;
379             }
380         }
381
382         private void btnUseWeapon_Click(object sender, EventArgs e)
383         {
384
385         }
386
387         private void btnUsePotion_Click(object sender, EventArgs e)
388         {
389
390         }
391     }
392 }
```

What's in the code you just added?

On **line 18,** we added a new variable to hold the monster that the player is fighting at the current location.

In the form's constructor, we do a couple things to start the game.

On **line 25,** we *move* the player to their home. Since the MoveTo() function expects a location as the parameter, we need to use the World.GetLocationByID() function to get the correct location. This is where we use the constant World.LOCATION_ID_HOME, instead of using the value 1. It's much easier to look at the code with the constant and know what it's supposed to be doing when we use this clearly-named constant.

On **line 26,** we add an item to the player's inventory – a rusty sword. They'll need something to fight with when they encounter their first monster.

We added the new MoveTo() function to handle all player movement.

We've also gone into each of the four functions that handle moving in a different direction and had them call the MoveTo() function.

The MoveTo() function

The first thing you may notice in this function are the lines that have a *double forward slash* (//) in them. These are **comments.** Comments are ignored by the computer. They only exist for programmers to read, to know what the program is supposed to be doing. Everything after the double-slashes, until the end of the line, is ignored by the computer.

I used a lot more comments in this function than I normally would. That's to make it easier for you to follow along with what is happening, and see how the code ties back to the pseudo-code we have above.

The second thing you may have noticed is that this function is long – over 300 lines long. That's really too long for a function.

When a function is that long, it's difficult to keep track of what exactly it's supposed to be doing. But we're going to clean this up in the next lesson. Personally, I like to keep my functions around 10 to 40 lines long.

We'll break this function into smaller functions in the next lesson.

What's happening in the MoveTo() function?

On **line 57**, we have our first IF statement.

In this case, we check if the new location has any items required to enter it.

The != is how C# says *not equal to*. The exclamation point, when doing any sort of comparison in C#, means *not*. And *null* means nothing/empty.

So, if the ItemRequiredToEnter property of the location is not empty, we need to check if the player has the required item in their inventory. If it is empty, we don't need to do anything – there is no required item, so the player can always move to the new location.

On **line 72** we see if we found the required item in the player's inventory. If we didn't find an item with a matching ID, the *playerHasRequiredItem* variable will still be *false*.

Notice the exclamation point (!) in front of playerHasRequiredItem.

Let's assume the player does not have the required item in their inventory. The *playerHasRequiredItem* variable will have a value of *false*. Doing a *not* on a Boolean variable, reverses its value: *!true* equals *false,* and *!false* equals *true*.

Thinking *if not false* is not as clear as thinking *if true*. But they both mean the same thing.

On **line 75**, we display the message that the player is missing the item required to enter this location. This line has a new += symbol – the **addition assignment operator.**

+= means, take the value from the variable/property on the left, add the value on the right to it, and assign the results back into the variable/property on the left.

When you use += with a string, it means, *"add the string value on the right to the end of the existing string"*. When you use it with a number, it means, *"add the value on the right to the value on the left"*.

Here, we take the text in the *rtbMessages* RichTextBox, and add our new message to the end of it. That way, the player can still see the old messages. If we used the = sign instead, it would replace the existing *Text* value with our new message.

We also have *Environment.NewLine*. This adds an *Enter* to the text, so the next thing we add to it will be displayed on the next line, instead of the end of the current line.

Line 76 has *return*. This means *"exit out of this function"*.

Since this function is a *void* function (see line 54), it doesn't return a value. We can *return* here and not do the rest of the function. We want to do that in this case, because the player does not have the item required to enter the location. So, we don't want to do the rest of the function, which would actually move them to the location.

On **lines 84 through 87**, we make the movement buttons visible, or not, based on whether or not the new location has a place to move to in each direction. We do this by checking if the property for the location is empty or not.

The *Visible* property of the buttons expects a Boolean value: true or false.

So, on **line 84**, if the LocationToNorth property is not empty, the value to the right of the equal sign will evaluate to *true,* and the button will be visible. If the LocationToNorth property is empty, this will evaluate to *false,* and the button will not be visible.

On **line 100**, we check if there is a quest at this location. If so, we need to do some more work.

Lines 106 through 117 are where we look through the player's quest list, to see if they already have the quest at this location and if they already completed it.

Lines 128 through 163 looks at each item required to complete the quest, then checks each item in the player's inventory, to see if they have it, and have enough of it, to complete the quest.

There are some *break* statements in this FOREACH. Those are used to stop looping through the items and exit the FOREACH loop. If we discover that the player doesn't have one item, or enough of it, to complete the quest, we can stop checking for any other items.

Line 180 has a `-=` symbol – the **subtraction assignment operator.** The `+=` means, *"add the value on the right to the variable/property on the left"*. So, a `-=` means, *"subtract the value on the right from the variable/property on the left"*. You can only use this with numbers, and not strings, unlike the `+=` .

In this case, we are using it to remove items from the player's inventory that they turn in to complete the quest.

On **line 204**, there is a `++` symbol – the **increment operator**. When you have this after a variable or property, it means *"add 1 to this variable or property"*. There is also a `--` , for when you want to subtract 1 from a variable or property.

At **lines 264 through 273**, we create the new monster to fight, by making a new monster object, using the values from the standard monster in our World class.

Updating the DataGridViews in the UI

From **lines 290 through 321** we update the DataGridView controls in the UI.

NOTE:

There are better ways to connect your list properties to DataGridViews and ComboBoxes. But we're just concentrating on the basics in these tutorials.

The player's inventory will have changed if they completed a quest. The items needed to turn in were removed from their inventory. The reward item was added to their inventory. So, we need to update the UI with their current inventory.

Also, if they received a new quest, or completed an existing one, their quest list would change.

Updating the ComboBoxes in the UI

For the ComboBoxes in the UI, we create new lists to hold the specific datatype of items we want to show in the list **(**lines 324 and 353**).** Next, we go through all the items in the player's inventory and add them to these lists, if they are the correct datatype (lines 326-335 and 355-364).

On **lines 328 and 357**, there is a new comparison: *is* . This is used to see if an object is a specific datatype.

Remember how we created the Weapon and HealingPotion sub-classes of the Item class? When you create a Weapon object, its datatype is both Weapon and Item. When you create a HealingPotion object, its datatype is both HealingPotion and Item.

If the lists are empty (weapons.Count or healingPotions.Count are equal to 0), we hide the ComboBox and *use* buttons, since there is no weapon or potion for the player to use.

If the lists have items in them, we *bind* the list to the comboboxes **(**lines 345-349 and 374-378). The *DisplayMember* determines what property will be displayed in the comboboxes. In this case, we want to display the value of the Name property. The *ValueMember* is the behind-the-scenes value we'll use a little later, to know which item was selected.

Hopefully you noticed something about the variable names. They are generally long and descriptive. That makes it easy to understand what values they are supposed to hold. By making your variable names descriptive, it will be easier to work with your program – especially when fixing bugs or making changes in the future.

There are a couple cases in the FOREACHs where I use short variables like *qci* and *ii*.

Since the foreach loop is only a few lines long, I sometimes use a shorter variable name. The variable has a very short life – it only exists within those few lines of the loop. You can display the whole loop on your screen, without scrolling. So, it's easy to keep track of where the variable is populated and where it is used. If the loop was longer, I'd use a longer, more descriptive name.

Summary

Now you've seen how to plan out your logic in pseudo-code, and then create a function in the program to perform that logic.

You've seen how IFs, ELSEs, and FOREACHs are used in a function. You've also seen what a huge function looks like, and you probably have an idea how difficult it would be to work with it. The next lesson will cover how to clean up that function and make it easier to understand.

We covered a lot in this lesson. If there was anything that wasn't clear, please leave a comment at scottlilly.com.

16.1 Refactoring the player movement function

Lesson objectives

At the end of this lesson, you will know...

- ➔ How to break a large function into smaller, easier-to-understand functions
- ➔ How to move functions from the UI to the *business classes*

In the last lesson, we created a huge function to move a player to a new location. However, it was too big to easily maintain. Now, we're going to clean it up and move things around so it is easier to read and understand. This is often called **refactoring.**

Refactoring is a large subject, and there are many techniques you can use to do it. We'll focus on a couple of the common, simple, techniques that have the biggest benefits. As you continue writing programs, you'll learn more refactoring techniques.

Steps for refactoring

Refactoring is just rearranging your code so it is easier to work with. You're not looking to add any new features, fixing any bugs, or improving the performance. There's a lot you can do for refactoring, but we'll focus on a couple of the most common techniques.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Right-click on the SuperAdventure.cs form, in the SuperAdventure project, then select *View Code*.

STEP 3 Look for **duplicated code.**

If you have the exact same code in more than one place, then you can usually move that code to its own function. Then, in the places that used to have that code, change it to call the new function. This is especially important when you might make changes in the future.

When we created the MoveTo() function, we could have put all that code in the four functions that moved the player in each direction. However, if we ever decided to change how the movement logic worked, we'd need to remember to make the change in four places. If we weren't paying attention, we might only change three of the four. Then, the game would suddenly start acting strangely every time the player moved in the one direction that didn't have the change added.

We don't have duplicated code in this function. We have a few places that look close to each other, but no exact duplicates. So, we'll look for other ways to refactor this function.

STEP 4 Look for **code that has one distinct purpose,** and move it to its own function.

NOTE:
When I'm referring to line numbers on these pages, I'm referring to line positions as they are in lesson 16. This way, if your line numbers don't exactly match, you'll be able to find the piece of code I'm talking about.

In this huge function, we have lots of code that matches this description. For example, lines 56 through 78 checks if there is a required item for a location, and (if so), if the player has it in their inventory. We can move this to its own, smaller function.

When we move this section of code, we should think if it might belong in a better place – maybe a different class. We currently have it in the code for our user interface. However, this code is looking at the player's inventory. So it makes sense to move it to the Player class.

In your Player class, press enter twice right after the closing curly brace of the Player constructor. Then add the code below. We'll modify the SuperAdventure.cs in a moment.

```
public bool HasRequiredItemToEnterThisLocation(Location location)
{
    if(location.ItemRequiredToEnter == null)
    {
        // There is no required item for this location,
           so return "true"
        return true;
    }

    // See if the player has the required item in
       their inventory
    foreach(InventoryItem ii in Inventory)
    {
        if(ii.Details.ID == location.ItemRequiredToEnter.ID)
        {
            // We found the required item, so return "true"
            return true;
        }
    }

    // We didn't find the required item in their inventory,
       so return "false"
    return false;
}
```

In this function, we pass in the location and see if the player can move there – either because there is no required item, or because they have the required item in their inventory.

This new HasRequiredItemToMoveToThisLocation() function is 20 lines long. It does one thing, and is small enough that it's very easy to understand. If we ever want to change this logic, we'll be able to do it in this one place.

For example, you might want to change the game to also have a minimum level requirement for a player to enter certain locations. You can go to this function and easily make the change, instead of digging through the 300 line function.

Now that we have the function in the Player class, we can clean up the SuperAdventure class. Go to the SuperAdventure class and replace lines 57 through 78 with this code:

```
// Does the location have any required items
if(!_player.HasRequiredItemToEnterThisLocation(newLocation))
{
    rtbMessages.Text += "You must have a " +
        newLocation.ItemRequiredToEnter.Name +
            " to enter this location." + Environment.NewLine;
    return;
}
```

We just replaced 20 lines of code with 6, and made this function a little easier to read and understand. That's what refactoring is all about.

Now we'll move the code that checks if the player already has a quest, and if they've completed it, to the Player class. Switch back to the Player class and make some room after the HasRequiredItemToMoveToThisLocation() method's closing curly brace. Then add this code:

```
public bool HasThisQuest(Quest quest)
{
    foreach(PlayerQuest playerQuest in Quests)
    {
        if(playerQuest.Details.ID == quest.ID)
        {
            return true;
        }
    }

    return false;
}

public bool CompletedThisQuest(Quest quest)
{
    foreach(PlayerQuest playerQuest in Quests)
    {
        if(playerQuest.Details.ID == quest.ID)
        {
            return playerQuest.IsCompleted;
        }
    }

    return false;
}
```

In the SuperAdventure.cs class, delete lines 102 through 117 and replace them with these calls to the player object. We are calling the last two methods we've just added to the Player class.

```
// See if the player already has the quest, and if they've
   completed it
bool playerAlreadyHasQuest =
    _player.HasThisQuest(newLocation.QuestAvailableHere);
bool playerAlreadyCompletedQuest =
    _player.CompletedThisQuest(newLocation.QuestAvailableHere);
```

Now, let's move the code that checks if the player has all the quest completion items in their inventory, and enough of them. Continue in the Player class and add this code:

```
public bool HasAllQuestCompletionItems(Quest quest)
{
    // See if the player has all the items needed
       to complete the quest here
    foreach(QuestCompletionItem qci in quest.QuestCompletionItems)
    {
        bool foundItemInPlayersInventory = false;

        // Check each item in the player's inventory,
           to see if they have it, and enough of it
        foreach(InventoryItem ii in Inventory)
        {
            // The player has the item in their inventory
            if(ii.Details.ID == qci.Details.ID)
            {
                foundItemInPlayersInventory = true;
                // The player does not have enough of this item
                   to complete the quest
                if(ii.Quantity < qci.Quantity)
                {
                    return false;
                }
            }
        }

        // The player does not have any of this quest
           completion item in their inventory
        if(!foundItemInPlayersInventory)
        {
            return false;
        }
    }

    // If we got here, then the player must have all the required
       items, and enough of them, to complete the quest.
    return true;
}
```

Go back to the SuperAdventure.cs and delete lines 125 through 163. In their place, add this code:

```
// See if the player has all the items needed to complete the quest
bool playerHasAllItemsToCompleteQuest =
    _player.HasAllQuestCompletionItems(newLocation.QuestAvailableHere);
```

We can also move the code that removes the quest completion items from the player's inventory into the Player class. We continue in the Player class:

```
public void RemoveQuestCompletionItems(Quest quest)
{
    foreach(QuestCompletionItem qci in quest.QuestCompletionItems)
    {
        foreach(InventoryItem ii in Inventory)
        {
            if(ii.Details.ID == qci.Details.ID)
            {
                // Subtract the quantity from the player's
                   inventory that was needed to complete the quest
                ii.Quantity -= qci.Quantity;
                break;
            }
        }
    }
}
```

Then go back to SuperAdventure.cs and remove lines 172 through 184, and replace them with this:

```
// Remove quest items from inventory
_player.RemoveQuestCompletionItems(newLocation.QuestAvailableHere);
```

We can also move the code that adds the reward item to the player's inventory into the Player class:

```
public void AddItemToInventory(Item itemToAdd)
{
    foreach(InventoryItem ii in Inventory)
    {
        if(ii.Details.ID == itemToAdd.ID)
        {
            // They have the item in their inventory, so increase
               the quantity by one
            ii.Quantity++;

            return; // We added the item, and are done, so get out
                       of this function
        }
    }
    // They didn't have the item, so add it to their inventory,
       with a quantity of 1
    Inventory.Add(new InventoryItem(itemToAdd, 1));
}
```

In SuperAdventure.cs, replace lines 196 through 216 with this:

```
// Add the reward item to the player's inventory
_player.AddItemToInventory(newLocation.QuestAvailableHere.RewardItem);
```

The last thing we'll move into the Player class, is the code that marks a quest as completed. Back in the Player class, add this code:

```
public void MarkQuestCompleted(Quest quest)
{
    // Find the quest in the player's quest list
    foreach(PlayerQuest pq in Quests)
    {
        if(pq.Details.ID == quest.ID)
        {
            // Mark it as completed
            pq.IsCompleted = true;

            // We found the quest, and marked it complete, so get
               out of this function
            return;
        }
    }
}
```

In SuperAdventure.cs, delete lines 218 through 229 and add this line of code:

```
// Mark the quest as completed
_player.MarkQuestCompleted(newLocation.QuestAvailableHere);
```

In the rest of the function, we have some code that updates the ComboBoxes and DataGridViews, since the player's inventory may have changed because of completing a quest.

We'll keep this code in SuperAdventure.cs, but put it in separate functions for better readability.

In SuperAdventure.cs, place your cursor behind the closing curly brace of the MoveTo() function and make some room for new code. Create these new functions:

Update inventory list in UI

```
private void UpdateInventoryListInUI()
{
    dgvInventory.RowHeadersVisible = false;

    dgvInventory.ColumnCount = 2;
    dgvInventory.Columns[0].Name = "Name";
    dgvInventory.Columns[0].Width = 197;
    dgvInventory.Columns[1].Name = "Quantity";

    dgvInventory.Rows.Clear();

    foreach(InventoryItem inventoryItem in _player.Inventory)
    {
        if(inventoryItem.Quantity > 0)
        {
            dgvInventory.Rows.Add(new[] {
                inventoryItem.Details.Name,
                inventoryItem.Quantity.ToString() });
        }
    }
}
```

Update quest list in UI

```
private void UpdateQuestListInUI()
{
    dgvQuests.RowHeadersVisible = false;

    dgvQuests.ColumnCount = 2;
    dgvQuests.Columns[0].Name = "Name";
    dgvQuests.Columns[0].Width = 197;
    dgvQuests.Columns[1].Name = "Done?";

    dgvQuests.Rows.Clear();

    foreach(PlayerQuest playerQuest in _player.Quests)
    {
        dgvQuests.Rows.Add(new[] {
            playerQuest.Details.Name,
            playerQuest.IsCompleted.ToString() });
    }
}
```

Update weapon list in UI

```
private void UpdateWeaponListInUI()
{
    List<Weapon> weapons = new List<Weapon>();

    foreach(InventoryItem inventoryItem in _player.Inventory)
    {
        if(inventoryItem.Details is Weapon)
        {
            if(inventoryItem.Quantity > 0)
            {
                weapons.Add((Weapon)inventoryItem.Details);
            }
        }
    }

    if(weapons.Count == 0)
    {
        // The player doesn't have any weapons, so hide the weapon
           combobox and "Use" button
        cboWeapons.Visible = false;
        btnUseWeapon.Visible = false;
    }
    else
    {
        cboWeapons.DataSource = weapons;
        cboWeapons.DisplayMember = "Name";
        cboWeapons.ValueMember = "ID";

        cboWeapons.SelectedIndex = 0;
    }
}
```

Update potion list in UI

```
private void UpdatePotionListInUI()
{
    List<HealingPotion> healingPotions = new List<HealingPotion>();

    foreach(InventoryItem inventoryItem in _player.Inventory)
    {
        if(inventoryItem.Details is HealingPotion)
        {
            if(inventoryItem.Quantity > 0)
            {
                healingPotions.Add(
                    (HealingPotion)inventoryItem.Details);
            }
        }
    }

    if(healingPotions.Count == 0)
    {
        // The player doesn't have any potions, so hide the potion
           combobox and "Use" button
        cboPotions.Visible = false;
        btnUsePotion.Visible = false;
    }
    else
    {
        cboPotions.DataSource = healingPotions;
        cboPotions.DisplayMember = "Name";
        cboPotions.ValueMember = "ID";

        cboPotions.SelectedIndex = 0;
    }
}
```

Our UI update code is now inside separate functions that are outside the MoveTo() function. It's time we got rid of the UI update code that is still in the MoveTo() function, and called these new functions from inside the MoveTo() function. So, go ahead and delete the code from line 290 through 379, and replace it with this (you'll notice the comments are the same):

```
// Refresh player's inventory list
UpdateInventoryListInUI();

// Refresh player's quest list
UpdateQuestListInUI();

// Refresh player's weapons combobox
UpdateWeaponListInUI();

// Refresh player's potions combobox
UpdatePotionListInUI();
```

Is the function simpler now?

Before, the MoveTo() function was over 300 lines long. Now, it's 140. It's still long, but it's much easier to read (and understand) now.

We've also moved a lot of the *logic* code out of the user interface class – which is a good thing. The code in the user interface class should only be used to handle receiving input from the user, and displaying output. It shouldn't have a lot of logic in it.

Now we have more of the game logic in the Engine project, where it belongs. We could do more refactoring on this function, but I think this is a good place to stop for now.

If you're interested in going further, I suggest you look at .Net's LINQ. You can use it to make the new functions in the Player class even smaller, and more concise. But LINQ is a whole other thing to learn, and I won't be showing it in these starter tutorials.

Summary

Refactoring doesn't change what a program does, it only cleans up the existing code, so it's simpler to understand and work with.

We often do that by finding pieces of a function that can be moved to their own function. Then we have the original function call these new, smaller functions. The smaller functions are easier to understand, since they aren't buried in a huge function. And the big function is easier to read, since it is shorter and more concise.

16.2 Functions to use weapons and potions

Lesson objectives

At the end of this lesson, you will know...

➔ Nothing new. We're just finishing the program, using the same things you learned in the previous lessons.

Now we'll write the functions the player will use when fighting monsters.

We have two things they can do (besides click on one of the direction buttons, to run away from the battle): use a weapon on the monster or use a healing potion on themselves – if they have one in their inventory. Here is the pseudo-code for the two functions:

Use a weapon function

- ❶ Get the currently selected weapon from the cboWeapons ComboBox
- ❶ Determine the amount of damage the player does to the monster
- ❶ Apply the damage to the monster's CurrentHitPoints
  - ❷ Display message
- ❶ If the monster is dead (zero hit points remaining)
  - ❷ Display a victory message
  - ❷ Give player experience points for killing the monster
    - ❸ Display message
  - ❷ Give player gold for killing the monster
    - ❸ Display message
  - ❷ Get loot items from the monster
    - ❸ Display message for each loot item
    - ❸ Add item to player's inventory
  - ❷ Refresh player data on UI
    - ❸ Gold and ExperiencePoints
    - ❸ Inventory list and ComboBoxes
  - ❷ *Move* player to current location
    - ❸ This will heal the player and create a new monster
- ❶ If the monster is still alive
  - ❷ Determine the amount of damage the monster does to the player
  - ❷ Display message
  - ❷ Subtract damage from player's CurrentHitPoints
    - ❸ Refresh player data in UI
  - ❷ If player is dead (zero hit points remaining)
    - ❸ Display message
    - ❸ Move player to *Home* location

Use a potion function

- ❶ Get currently selected potion from cboPotions ComboBox
- ❶ Add healing amount to player's CurrentHitPoints
  - ❷ CurrentHitPoints cannot exceed player's MaximumHitPoints
- ❶ Remove the potion from the player's inventory
- ❶ Display message
- ❶ Monster gets their turn to attack
  - ❷ Determine the amount of damage the monster does to the player
  - ❷ Display message
  - ❷ Subtract damage from player's CurrentHitPoints
    - ❸ Refresh player data in UI
  - ❷ If player is dead (zero hit points remaining)
    - ❸ Display message
    - ❸ Move player to *Home* location
- ❶ Refresh player data in UI

These are much simpler functions than the MoveTo() function.

We'll be able to use some of the smaller functions we created during the refactoring lesson – for example, the AddItemToInventory() function, from the Player class, if the player defeats the monster and receives loot items.

Adding functions for monster battles

STEP 1 Start Visual Studio, and open the solution.

STEP 2 Right-click on the SuperAdventure.cs form in the SuperAdventure project, to start working with the code for the UI.

STEP 3 In SuperAdventure.cs navigate to the btnUseWeapon_Click() function.

If you can't find btnUseWeapon_Click(), double click SuperAdventure.cs in the Solution Explorer to get to the form editor, and then double click the top Use button.

STEP 4 Replace the btnUseWeapon_Click() functions with the code on the next three pages.

STEP 5 Repeat steps 3 and 4 for the btnUsePotion_Click() function (the code is on the fourth page.

```
 1  private void btnUseWeapon_Click(object sender, EventArgs e)
 2  {
 3      // Get the currently selected weapon from the cboWeapons ComboBox
 4      Weapon currentWeapon = (Weapon)cboWeapons.SelectedItem;
 5
 6      // Determine the amount of damage to do to the monster
 7      int damageToMonster = RandomNumberGenerator.NumberBetween(
            currentWeapon.MinimumDamage, currentWeapon.MaximumDamage);
 8
 9      // Apply the damage to the monster's CurrentHitPoints
10      _currentMonster.CurrentHitPoints -= damageToMonster;
11
12      // Display message
13      rtbMessages.Text += "You hit the " + _currentMonster.Name + " for " +
            damageToMonster.ToString() + " points." + Environment.NewLine;
14
15      // Check if the monster is dead
16      if(_currentMonster.CurrentHitPoints <= 0)
17      {
18          // Monster is dead
19          rtbMessages.Text += Environment.NewLine;
20          rtbMessages.Text += "You defeated the " + _currentMonster.Name +
                Environment.NewLine;
21
22          // Give player experience points for killing the monster
23          _player.ExperiencePoints += _currentMonster.RewardExperiencePoints;
24          rtbMessages.Text += "You receive " +
                _currentMonster.RewardExperiencePoints.ToString() +
                    " experience points" + Environment.NewLine;
25
26          // Give player gold for killing the monster
27          _player.Gold += _currentMonster.RewardGold;
28          rtbMessages.Text += "You receive " +
                _currentMonster.RewardGold.ToString() + " gold" + Environment.NewLine;
29
30          // Get random loot items from the monster
31          List<InventoryItem> lootedItems = new List<InventoryItem>();
32
33          // Add items to the lootedItems list, comparing a random number to the drop
               percentage
34          foreach(LootItem lootItem in _currentMonster.LootTable)
35          {
36              if(RandomNumberGenerator.NumberBetween(1, 100) <= lootItem.DropPercentage)
37              {
38                  lootedItems.Add(new InventoryItem(lootItem.Details, 1));
39              }
40          }
```

continued on next page

```
41          // If no items were randomly selected, then add the default loot item(s).
42          if(lootedItems.Count == 0)
43          {
44              foreach(LootItem lootItem in _currentMonster.LootTable)
45              {
46                  if(lootItem.IsDefaultItem)
47                  {
48                      lootedItems.Add(new InventoryItem(lootItem.Details, 1));
49                  }
50              }
51          }
52
53          // Add the looted items to the player's inventory
54          foreach(InventoryItem inventoryItem in lootedItems)
55          {
56              _player.AddItemToInventory(inventoryItem.Details);
57
58              if(inventoryItem.Quantity == 1)
59              {
60                  rtbMessages.Text += "You loot " +
                        inventoryItem.Quantity.ToString() + " " +
                            inventoryItem.Details.Name + Environment.NewLine;
61              }
62              else
63              {
64                  rtbMessages.Text += "You loot " +
                        inventoryItem.Quantity.ToString() + " " +
                            inventoryItem.Details.NamePlural + Environment.NewLine;
65              }
66          }
67
68          // Refresh player information and inventory controls
69          lblHitPoints.Text = _player.CurrentHitPoints.ToString();
70          lblGold.Text = _player.Gold.ToString();
71          lblExperience.Text = _player.ExperiencePoints.ToString();
72          lblLevel.Text = _player.Level.ToString();
73
74          UpdateInventoryListInUI();
75          UpdateWeaponListInUI();
76          UpdatePotionListInUI();
77
78          // Add a blank line to the messages box, just for appearance.
79          rtbMessages.Text += Environment.NewLine;
80
81          // Move player to current location (to heal player and create a new monster
               to fight)
82          MoveTo(_player.CurrentLocation);
83      }
```

continued on next page

```
 84      else
 85      {
 86          // Monster is still alive
 87
 88          // Determine the amount of damage the monster does to the player
 89          int damageToPlayer =
 90              RandomNumberGenerator.NumberBetween(0, _currentMonster.MaximumDamage);
 91
 92          // Display message
 93          rtbMessages.Text += "The " + _currentMonster.Name + " did " +
 94              damageToPlayer.ToString() + " points of damage." + Environment.NewLine;
 95
 96          // Subtract damage from player
 97          _player.CurrentHitPoints -= damageToPlayer;
 98
 99          // Refresh player data in UI
100          lblHitPoints.Text = _player.CurrentHitPoints.ToString();
101
102          if(_player.CurrentHitPoints <= 0)
103          {
104              // Display message
105              rtbMessages.Text += "The " + _currentMonster.Name + " killed you." +
106                  Environment.NewLine;
107
108              // Move player to "Home"
109              MoveTo(World.LocationByID(World.LOCATION_ID_HOME));
110          }
111      }
112  }
```

continued on next page

btnUsePotion_Click

```
113  private void btnUsePotion_Click(object sender, EventArgs e)
114  {
115      // Get the currently selected potion from the combobox
116      HealingPotion potion = (HealingPotion)cboPotions.SelectedItem;
117
118      // Add healing amount to the player's current hit points
119      _player.CurrentHitPoints = (_player.CurrentHitPoints + potion.AmountToHeal);
120
121      // CurrentHitPoints cannot exceed player's MaximumHitPoints
122      if(_player.CurrentHitPoints > _player.MaximumHitPoints)
123      {
124          _player.CurrentHitPoints = _player.MaximumHitPoints;
125      }
126
127      // Remove the potion from the player's inventory
128      foreach(InventoryItem ii in _player.Inventory)
129      {
130          if(ii.Details.ID == potion.ID)
131          {
132              ii.Quantity--;
133              break;
134          }
135      }
136
137      // Display message
138      rtbMessages.Text += "You drink a " + potion.Name + Environment.NewLine;
139
140      // Monster gets their turn to attack
141
142      // Determine the amount of damage the monster does to the player
143      int damageToPlayer =
144          RandomNumberGenerator.NumberBetween(0, _currentMonster.MaximumDamage);
145
146      // Display message
147      rtbMessages.Text += "The " + _currentMonster.Name + " did " +
148          damageToPlayer.ToString() + " points of damage." + Environment.NewLine;
149
150      // Subtract damage from player
151      _player.CurrentHitPoints -= damageToPlayer;
152
153      if(_player.CurrentHitPoints <= 0)
154      {
155          // Display message
156          rtbMessages.Text += "The " + _currentMonster.Name + " killed you." +
157              Environment.NewLine;
158
159          // Move player to "Home"
160          MoveTo(World.LocationByID(World.LOCATION_ID_HOME));
161      }
162
163      // Refresh player data in UI
164      lblHitPoints.Text = _player.CurrentHitPoints.ToString();
165      UpdateInventoryListInUI();
166      UpdatePotionListInUI();
167  }
```

There isn't really anything new in these two functions. Just more IFs and FOREACHs to handle the player's actions in battle.

Summary

Now you have a working game. The player can move around in the world, get quests, battle monsters, receive loot, and complete quests.

These new functions could use some refactoring, since they are long and do several things. I'll leave that to you to figure out what refactoring you'd do.

17 Running the game on another computer

Lesson objectives

At the end of this lesson, you will know...

➔ How to move your game to another computer, and run it without Visual Studio

Right now, you have to start Visual Studio and load your solution before you can play the game. But if you want to share your program with someone else, you don't want to make them install Visual Studio.

So, now we'll build a version that you can copy to a CD/DVD/thumb drive/etc. and share with your friends and family. Visual Studio has a nice installation package creation project, but it gets a little complex. So, we're going to use the *quick and dirty* method, for now.

Building the program for another computer

The first thing you want to do is build the program in *Release* mode. So far, we've used *Debug* mode, which can be useful while writing the program. Debug mode lets you take a look into the program while it's running, in case you have a problem and aren't sure where it's happening.

But when you're done writing your program, and are ready to *release it,* you want to use Release mode. Then, you just need to copy the program's files to a computer that has the correct .Net Framework installed, and you can run it there.

STEP 1 Start Visual Studio, and open the solution.

STEP 2 In the menu, to the right of the *Start,* there is a dropdown that probably says *Debug* right now (that's the default value). Change it to *Release.*

STEP 3 From the menu, click on *Build,* then *Build Solution*. This will create the executable files – the ones needed to run your program.

Check the *Output* box, in the bottom middle of Visual Studio. It will tell you when it's done building the program, along with where it was built.

STEP 4 Open up Windows Explorer and get to the folder where the executable file was created. You'll see the file there, along with some other ones. You don't need all of them, but it won't hurt to have the extra files. So, we'll copy all of them to our new location.

In this case, I just created a new folder named *Games,* with a sub-folder under it named *SuperAdventure.* You can name your folder whatever you want. Then, paste the files into this folder.

STEP 5 To make this easier to run, create a desktop shortcut for the *SuperAdventure.exe* file. That's the executable, the file to run the game. Now, you can double-click on the shortcut on the desktop and start playing your game.

Summary

Now you can copy your program to another computer (assuming they've been running Windows Update, and have the same version of the .Net Framework you used when you made the program.

If you learn how to create a Windows Installer project, you can end up with a single program someone can use to install your program on their computer. If their computer is missing anything, or doesn't have the correct version of the .Net Framework, the installer program will fix that for them.

There are also some more complex installations for different types of projects – ones that use a database, or are run as a website.

But, now you know how to copy this game over to your friend or family member's computer, so they can see the program you created and play the game.

18 Future enhancements for the game

Congratulations!

You finished with the lessons, and have a working game! Plus, you've learned many of the most common things you need to write more C# programs.

There is still plenty more to learn, if you decide to get serious about programming. I've been programming for over 30 years, and learn something new every week.

Expanding the game

The easiest thing for you to do is to make a bigger world, with more locations, quests, monsters, and items.

Draw a map of your larger world, and modify the World class to include these new locations. Create more monsters and quests. Add more powerful weapons, so the player can defeat the giant spider.

Ideas for new features

This is a very simple RPG, and there is a lot you can do to expand it.

Here are a few ideas:

- Save the player's current game to disk, and re-load it later
- As the player gains experience, increase their level
  - Increase MaximumHitPoints with each new level
  - Add a minimum level requirement for some items
  - Add a minimum level requirement for some locations
- Add randomization to battles
  - Determine if the player hits the monster
  - Determine if the monster hits the player
- Add player attributes (strength, dexterity, etc.)
  - Use attributes in battle: who attacks first, amount of damage, etc.
- Add armor and jewelry
  - Makes it more difficult for the monster to hit the player
  - Has special benefits: increased chance to hit, increased damage, etc.
- Add crafting skills the player can acquire
- Add crafting recipes the player use
  - Require the appropriate skill
  - Requires components (inventory items)
- Make some quests repeatable

- Make quest chains (player must complete *Quest A* before they can receive *Quest B*)
- Add magic scrolls
- Add spells
  - Level requirements for the spells
  - Spells require components to cast (maybe?)
- Add more potions
  - More powerful healing potions
  - Potions to improve player's *to hit* chances, or damage
- Add poisons to use in battle
- Add pets
  - Help the player in battle by attacking opponents
  - Help the player in battle by healing the player
- Add stores/vendors
  - Player can sell useless items and buy new equipment, scrolls, potions, poisons, and crafting/spell components

There are also more programming techniques you can learn to make the program a little cleaner.

- LINQ, when searching lists
- Events/delegates, to handle communication between the *logic* project and the UI project – which will let you move more logic code out of the UI project
- BindingList, so you don't have to repeatedly repopulate the DataGridViews and ComboBox in the UI

Version control

If you're going to make more changes, or write more programs, you really should learn how to use a version control tool.

You can create a backup copy of your program by copying the solution folder to a new location before making your change. But version control software is a better solution.

It will let you keep track of all the changes you ever make to your program. This is extremely helpful when you make some changes that don't work, and want to go back to the old, working version.

I use **TortoiseSVN** (tortoisesvn.net) and **VisualSVN** – a Visual Studio plug-in that works with TortoiseSVN (visualsvn.com).

Git is another popular version control tool (git-scm.com). Many programmers use a web-based version of it at **GitHub** (github.com).

Version control tools usually take a little while to set up, and to figure out how to use. But once you have one in place, and you learn the basics, using it will become a habit that doesn't require any time or thought. And the first time you need to go back to a previous version, you'll thank yourself that you used version control.

Summary

Now that you have the basic game, you can expand it.

Hopefully you enjoyed these lessons, and learned some new things.

Please let me know if you have any questions about anything that wasn't clear in the lessons, if you want to see some other features in the game, or if you want to learn some other aspects of programming in C#.

19

Enhancements to the game

Scroll to the bottom of a rich text box

Use a calculated value for a property

Clean up the source code by converting FOREACH loops to LINQ

Saving and loading the player information

Changing dropdown default values

Increase maximum hit points when the player gains a level

19.1 Scroll to the bottom of a rich text box

When you play the game, you probably noticed the messages RichTextBox scrolls to the top after you add more messages to it. So, the player has to manually scroll to the bottom to see the latest message.

We want to make it easier for the player.

How to scroll to the bottom of a RichTextBox

STEP 1 Add this new ScrollToBottomOfMessages() function to the code in the SuperAdventure.cs class:

```
private void ScrollToBottomOfMessages()
{
    rtbMessages.SelectionStart = rtbMessages.Text.Length;
    rtbMessages.ScrollToCaret();
}
```

STEP 2 After you add more to *rtbMessages.Text*, call the ScrollToBottomOfMessages() function.

That's it.

19.2 Use a calculated value for a property

Lesson objectives

At the end of this lesson, you will know...

➔ How to return a value in a property, by calculating it from other properties

One thing missing from the game is updating the player's level, based on their current experience points.

We could do this by having the game re-calculate the level, and reset the value of the *Player.Level* property every time the player gains experience, but there is a much simpler way. We're going to have the property automatically calculate its value every time someone *gets* (reads) it.

How to have a property calculate its value from other properties

STEP 1 Open the SuperAdventure solution in Visual Studio and open the Player.cs class.

STEP 2 Change the line for the Level property from this:

```
public int Level { get; set; }
```

To this:

```
public int Level
{
    get { return ((ExperiencePoints / 100) + 1); }
}
```

Notice that we removed the *set;* . That's because we're never going to put a value into the Level property – which is what the *set* is for.

We also changed the *get;* . Before, since this was an *auto-property,* the *get* would get the value that it was previously *set* to. Now, the program will calculate the value for Level by dividing the ExperiencePoints by 100.

Since this property is an *int*, it will automatically round the answer down, which is why we add 1 to it – so the player will start out a level 1, and not 0.

STEP 3 Now we need to clean up the places where we were setting the Level value, since we removed the set option.

In Player.cs, remove this line from the constructor:

```
Level = level;
```

Since we don't set the Level value any more, we can also remove it as a parameter from the constructor. So, now the constructor for Player.cs shuld look like this:

```
public Player(int currentHitPoints, int maximumHitPoints,
    int gold, int experiencePoints) :
        base(currentHitPoints, maximumHitPoints)
{
    Gold = gold;
    ExperiencePoints = experiencePoints;

    Inventory = new List<InventoryItem>();
    Quests = new List<PlayerQuest>();
}
```

STEP 4 Now that we removed the Level parameter from the Player class, we need to clean up anything that called the constructor.

Open the code for the SuperAdventure.cs screen class. In the constructor for this class, we populate the *_player* variable by instantiating a new Player object.

```
_player = new Player(10, 10, 20, 0, 1);
```

Remove the value passed in for the level, so it looks like this:

```
_player = new Player(10, 10, 20, 0);
```

STEP 5 Now we need to make sure the Level is updated on the game screen, after every time the player gains experience (by killing a monster or completing a quest).

You could do this by just adding this line after every time the player's experience changes:

```
lblExperience.Text = _player.ExperiencePoints.ToString();
```

However, I decided to make this new function to update all the player's stats:

```
private void UpdatePlayerStats()
{
    // Refresh player information and inventory controls
    lblHitPoints.Text = _player.CurrentHitPoints.ToString();
    lblGold.Text = _player.Gold.ToString();
    lblExperience.Text = _player.ExperiencePoints.ToString();
    lblLevel.Text = _player.Level.ToString();
}
```

You'll find these same lines inside the constructor and the btnUseWeapon_Click() function. You can go ahead and delete them now. In their place, add a call to the UpdatePlayerStats() function. We're also going to call this function at the end of the MoveTo() function, where we update our lists in the UI.

Check your work

Build the solution and make sure there are no errors in the *Output* box at the bottom of Visual Studio. If you see any problems, double-check the changes you made in this lesson.

Summary

It may not be obvious at first, but this is a powerful programming practice.

Before, we needed to manually update both the player's ExperiencePoints property and their Level property. Why do two things (and possibly forget to do one, making the game act strangely) when you can do one thing and have it automatically update everything else that depends on it?

19.3 Clean up the source code by converting FOREACH loops to LINQ

Lesson objectives

At the end of this lesson, you will know...

➔ How to make your program easier to read by using LINQ when working with lists

We still have several places in the game that can use improvement. One way to make you code smaller, cleaner, and easier to understand is to replace some of the FOREACH loops (that usually are at least six lines long) with a LINQ statement (which is often one line long).

LINQ is short for **Language-Integrated Query.**

There are several different ways you can work with this, including one that looks similar to SQL (Structured Query Language – the language you use when you work with many databases). But we're going to use one of the other methods that I like – **lambdas**. To me, lambdas make the code very easy to read.

How to replace a FOREACH loop with a LINQ statement

STEP 1 Open the SuperAdventure solution in Visual Studio and open the Player.cs class.

STEP 2 Find the HasRequiredItemToEnterThisLocation() method. It should look like this:

```
public bool HasRequiredItemToEnterThisLocation(Location location)
{
    if(location.ItemRequiredToEnter == null)
    {
        // There is no required item for this location, so return
           "true"
        return true;
    }

    // See if the player has the required item in their inventory
    foreach(InventoryItem ii in Inventory)
    {
        if(ii.Details.ID == location.ItemRequiredToEnter.ID)
        {
            // We found the required item, so return "true"
            return true;
        }
    }

    // We didn't find the required item in their inventory,
       so return "false"
    return false;
}
```

In the first few lines, if there isn't anything required to enter the location, we return *true*, to allow the player move to the location.

For the rest of the method, if there is an item required for the player to move to the location, we have a FOREACH loop through the player's inventory, looking for the required item. The FOREACH loop, and the *return false* (if the item isn't found), take twelve lines of code.

Let's make it a little simpler.

STEP 3 In order to use LINQ, we need to have it available in the class, with a *using* statement. In this case, we already have *using System.Linq;* at the top of the class, so we're ready to make our change.

Our objective is to see if there is an item in the player's inventory with an ID that matches the ID of the item required to enter the location. With the FOREACH, we do this by looping through each item, checking it's ID.

With LINQ, we can reduce this to one line:

```
public bool HasRequiredItemToEnterThisLocation(Location location)
{
    if(location.ItemRequiredToEnter == null)
    {
        // There is no required item for this location, so return
           "true"
        return true;
    }

    // See if the player has the required item in their inventory
    return Inventory.Exists(ii => ii.Details.ID ==
        location.ItemRequiredToEnter.ID);
}
```

Let's compare the old method with the new one.

The Exists() function will check the items in the Inventory list, to see if any item matches the expression between the parentheses. If it finds an item, it returns *true*. If it doesn't, it returns *false*.

Inside the parentheses, what we see is similar to what was between the parentheses in the FOREACH and the IF statements in the old method.

To the left of the arrow (=>) is *ii* . This is the variable name the LINQ expression will use for each item in the list – just like it did with the FOREACH. To the right of the arrow is the expression that is going to be evaluated. In this case, check if the inventory item's ID matches the ID of the required item's ID.

That's how these lambda expressions work. The variable declaration for the list item is to the left of the arrow, and the expression is to the right.

STEP 4 Now we'll do the same thing to the HasThisQuest() method, and change it to this:

```
public bool HasThisQuest(Quest quest)
{
    return Quests.Exists(pq => pq.Details.ID == quest.ID);
}
```

STEP 5 Find the HasAllQuestCompletionItems() method.

This one is a little more complex. We want to see if the player has the item required to complete a quest *and* if they have enough of those items in their inventory.

So, we'll use this for our LINQ statement:

```
public bool HasAllQuestCompletionItems(Quest quest)
{
    // See if the player has all the items needed to complete
       the quest here
    foreach(QuestCompletionItem qci in quest.QuestCompletionItems)
    {
        // Check each item in the player's inventory,
           to see if they have it, and enough of it
        if(!Inventory.Exists(ii => ii.Details.ID ==
            qci.Details.ID && ii.Quantity >= qci.Quantity))
        {
            return false;
        }
    }

    // If we got here, then the player must have all the required
       items, and enough of them, to complete the quest.
    return true;
}
```

This expression will see if the item exists in the player's inventory *(ii.Details.ID == qci.Details.ID)* and if the quantity in the player's inventory is greater than, or equal to, the quantity required to complete the quest *(ii.Quantity >= qci.Quantity)*.

If the program doesn't find an item in the list that matches both conditions, we'll stop checking and return *false* for the method. If it gets through all the items required to complete the quest, the method returns *true* at the end.

We could go even further in cleaning up this method by writing a LINQ query for the remaining FOREACH in this method, but the query would be a little more complex than I want to show you right now.

STEP 6 You can also get a specific item from a list with the SingleOrDefault() method. However, you'll need to check if it returned *null*, since nothing matched the condition. SingleOrDefault() also only works if you'll only ever have one item in the list that matches the condition. You'll need to use a different LINQ method if you want to get more than one item from the list.

Here is how you can use SingleOrDefault() in the RemoveQuestCompletionItem(), AddItemToInventory(), and MarkQuestCompleted() methods:

```
public void RemoveQuestCompletionItems(Quest quest)
{
    foreach(QuestCompletionItem qci in
        quest.QuestCompletionItems)
    {
        InventoryItem item = Inventory.SingleOrDefault(
            ii => ii.Details.ID == qci.Details.ID);

        if(item != null)
        {
            // Subtract the quantity from the player's inventory
               that was needed to complete the quest
            item.Quantity -= qci.Quantity;
        }
    }
}


public void AddItemToInventory(Item itemToAdd)
{
    InventoryItem item = Inventory.SingleOrDefault(
        ii => ii.Details.ID == itemToAdd.ID);

    if(item == null)
    {
        // They didn't have the item, so add it to their
           inventory, with a quantity of 1
        Inventory.Add(new InventoryItem(itemToAdd, 1));
    }
    else
    {
        // They have the item in their inventory, so increase
           the quantity by one
        item.Quantity++;
    }
}
```

```
public void MarkQuestCompleted(Quest quest)
{
    // Find the quest in the player's quest list
    PlayerQuest playerQuest = Quests.SingleOrDefault(
        pq => pq.Details.ID == quest.ID);

    if(playerQuest != null)
    {
        playerQuest.IsCompleted = true;
    }
}
```

In this situation, it doesn't really reduce the amount of code. But you may find this useful in a future program.

Check your work

Build the solution and make sure there are no errors in the *Output* box at the bottom of Visual Studio. If you see any problems, double-check the changes you made in this lesson.

Summary

This covers just one way to use LINQ. You can also do things such as calculating the sum of a property for items in a list:

```
int sum = Inventory.Sum(ii => ii.Quantity);
```

You can build a chain of LINQ statements, like this (which will give you the sum of the Quantity of all items in the Inventory list, for items that have a Quantity greater than five):

```
int sum = Inventory.Where(
    ii => ii.Quantity > 5).Sum(ii => ii.Quantity);
```

For a good list of everything you can do with LINQ, check out dotnetperls.com.

19.4 Saving and loading the player information (XML)

Lesson objectives

At the end of this lesson, you will know...

- ➔ The basics of XML (Extensible Markup Language)
- ➔ How to save the player information to disk and reload it when the player restarts the game
- ➔ How to make prevent your program from crashing, if it reads in a bad file
- ➔ A very simple version of the *Factory* design pattern

In this lesson, we're going to look at how to save the player information for a game and reload it when they restart the game.

There are several different ways you could save data: in a database, a comma-separated value file, an XML file, etc. We're going to use an XML file. It doesn't require installing anything extra, such as a database. It's also easy to read, since an XML file includes the structure (definition) and the data.

There is a lot happening in this lesson. But that's what happens when you start to go beyond simple coding examples. In order to do one thing, you need to know how to do a couple other things. If you have problems understanding this lesson, take it one step at a time, until you are sure you know what is happening.

Defining the XML file structure

It's easier to understand XML with a sample. So, here is the format we'll use to store the game's data.

```
<Player>
    <Stats>
        <CurrentHitPoints>7</CurrentHitPoints>
        <MaximumHitPoints>10</MaximumHitPoints>
        <Gold>123</Gold>
        <ExperiencePoints>275</ExperiencePoints>
        <CurrentLocation>2</CurrentLocation>
    </Stats>
    <InventoryItems>
        <InventoryItem ID="1" Quantity="1" />
        <InventoryItem ID="2" Quantity="5" />
        <InventoryItem ID="7" Quantity="2" />
    </InventoryItems>
    <PlayerQuests>
        <PlayerQuest ID="1" IsCompleted="true" />
        <PlayerQuest ID="2" IsCompleted="false" />
    </PlayerQuests>
</Player>
```

What is XML?

Think of XML data as one long string, or file, with special formatting rules. There are three main parts to XML: **nodes, attributes,** and **values.**

Nodes, or *tags,* are the names of values, kind of like a property in a class.

In our file, the nodes are: Player, Stats, CurrentHitPoints, Gold, ExperiencePoints, Currentlocation, InventoryItems, InventoryItem, PlayerQuests, and PlayerQuest.

A few thing you need to know about XML nodes:

1. Nodes have a start and an end. Notice that there is <Player> node at the very beginning, and </Player> at the end. The forward slash (/) before the node name indicates that it is the end of the node.
2. Node start and end names *must match exactly.* They are *case-sensitive.* So, you cannot start a node with <*NAME*> and end it with <*/name*> .
3. Node names cannot contain a space. So, we can have a node named *CurrentHitPoints*, but not one named *Current Hit Points*.
4. Node starts and ends cannot cross. For example, you can have <A><B><C></C></B></A>, but you cannot have <A><B><C></B></C></A>. In the second example, the C node is started after the B node; however, the B node ends before the C node ends, which is not allowed in XML.
5. Nodes can also be *self-closing.* Notice that the InventoryItem nodes don't have a corresponding </InventoryItem>. The *end* of those nodes is done with the forward slash (/) just before the closing angle bracket (>).
6. If you start a new node before the previous node ends (like the CurrentHitPoints node, being between the start and end of the Stats node) then it is called a **child node**. CurrentHitPoints is a child node of Stats, and Stats is a child node of Player.
7. A node can have multiple child nodes, even with the same name. See how InventoryItems has multiple InventoryItem nodes.
8. You can store data as *values*, like the 7 in CurrentHitPoints, or as *attributes*, like the ID and Quantity in the InventoryItems.

Notice in the InventoryItem nodes, we store the ID and Quantity as attributes. If you wanted to, you could store them as values in child nodes.

So, instead of this:

```
<InventoryItem ID="1" Quantity="1" />
```

You could have this:

```
<InventoryItem>
    <ID>1</ID>
    <Quantity>1</Quantity>
<InventoryItem>
```

It's personal preference, which way you do it.

That's enough information about XML for us to get started. There are some more rules you need to know about, if you want to do more work with XML. Check out the Wikipedia page on XML (en.wikipedia.org/wiki/XML) to learn about them.

One thing to watch out for is if you ever output user-entered data into XML. If the user enters a less-than sign (<), XML thinks it is the beginning of a node (open angle bracket) – unless you follow some of the special techniques you'll find on the Wikipedia page (look for CDATA).

Why we are writing our own serialization code, instead of using the built-in serialization

Sometimes you need to pass objects to something outside your program. The problem is that it doesn't understand your program's object, or class. One example is the file system, which understands strings (and a few other things).

One way to do pass an object is to use *serialization*.

There are built-in functions in the .Net framework to do **serialization** (converting from an object) and **deserialization** (converting back into an object).

However, these built-in functions can be a little complex to use, especially when you have an object that contains a list of other objects. You need to modify all your classes to know how to serialize/de-serialize themselves. That's a lot of work for a small game. So, we won't be using that.

We'll write our own functions to convert the Player object information to XML, and back from XML into a Player object.

By the way, there are several types of serialization. You may have also heard of JSON (JavaScript Object Notation), which is used to transfer object information in many web sites and web apps. It uses a different format for its data, but the concept is similar to XML.

OK, enough talk. Time to write some code.

Saving and reloading player information

STEP 1 Open the SuperAdventure solution in Visual Studio and select the Player class to modify.

STEP 2 At the top of the Player class, add this line to the *using* statement section, so we have access to the XML functions:

```
using System.Xml;
```

STEP 3 Add the new ToXMLString() function to the Player class. This will take the Player information and create an XML string with all the Player's current data. You'll find the code for the function on the next two pages.

```
 1  public string ToXmlString()
 2  {
 3      XmlDocument playerData = new XmlDocument();
 4
 5      // Create the top-level XML node
 6      XmlNode player = playerData.CreateElement("Player");
 7      playerData.AppendChild(player);
 8
 9      // Create the "Stats" child node to hold the other player statistics nodes
10      XmlNode stats = playerData.CreateElement("Stats");
11      player.AppendChild(stats);
12
13      // Create the child nodes for the "Stats" node
14      XmlNode currentHitPoints = playerData.CreateElement("CurrentHitPoints");
15      currentHitPoints.AppendChild(playerData.CreateTextNode(
            this.CurrentHitPoints.ToString()));
16      stats.AppendChild(currentHitPoints);
17
18      XmlNode maximumHitPoints = playerData.CreateElement("MaximumHitPoints");
19      maximumHitPoints.AppendChild(playerData.CreateTextNode(
            this.MaximumHitPoints.ToString()));
20      stats.AppendChild(maximumHitPoints);
21
22      XmlNode gold = playerData.CreateElement("Gold");
23      gold.AppendChild(playerData.CreateTextNode(this.Gold.ToString()));
24      stats.AppendChild(gold);
25
26      XmlNode experiencePoints = playerData.CreateElement("ExperiencePoints");
27      experiencePoints.AppendChild(playerData.CreateTextNode(
            this.ExperiencePoints.ToString()));
28      stats.AppendChild(experiencePoints);
29
30      XmlNode currentLocation = playerData.CreateElement("CurrentLocation");
31      currentLocation.AppendChild(playerData.CreateTextNode(
            this.CurrentLocation.ID.ToString()));
32      stats.AppendChild(currentLocation);
33
34      // Create the "InventoryItems" child node to hold each InventoryItem node
35      XmlNode inventoryItems = playerData.CreateElement("InventoryItems");
36      player.AppendChild(inventoryItems);
37
38      // Create an "InventoryItem" node for each item in the player's inventory
39      foreach(InventoryItem item in this.Inventory)
40      {
41          XmlNode inventoryItem = playerData.CreateElement("InventoryItem");
42
43          XmlAttribute idAttribute = playerData.CreateAttribute("ID");
44          idAttribute.Value = item.Details.ID.ToString();
45          inventoryItem.Attributes.Append(idAttribute);
46
47          XmlAttribute quantityAttribute = playerData.CreateAttribute("Quantity");
48          quantityAttribute.Value = item.Quantity.ToString();
49          inventoryItem.Attributes.Append(quantityAttribute);
50
51          inventoryItems.AppendChild(inventoryItem);
52      }
```

continued on next page

```
53
54      // Create the "PlayerQuests" child node to hold each PlayerQuest node
55      XmlNode playerQuests = playerData.CreateElement("PlayerQuests");
56      player.AppendChild(playerQuests);
57
58      // Create a "PlayerQuest" node for each quest the player has acquired
59      foreach(PlayerQuest quest in this.Quests)
60      {
61          XmlNode playerQuest = playerData.CreateElement("PlayerQuest");
62
63          XmlAttribute idAttribute = playerData.CreateAttribute("ID");
64          idAttribute.Value = quest.Details.ID.ToString();
65          playerQuest.Attributes.Append(idAttribute);
66
67          XmlAttribute isCompletedAttribute = playerData.CreateAttribute("IsCompleted");
68          isCompletedAttribute.Value = quest.IsCompleted.ToString();
69          playerQuest.Attributes.Append(isCompletedAttribute);
70
71          playerQuests.AppendChild(playerQuest);
72      }
73
74      return playerData.InnerXml; // The XML document, as a string, so we can save
                                    the data to disk
75  }
```

At the start of this function, we create the *playerData* XmlDocument. This object lets us add nodes in a safe way, so we can't break the most common XML formatting rules.

Now that we have the XmlDocument, we'll populate it with our player data. First, we create the *Player* XmlNode and add it to the document.

Next, we create the *Stats* XmlNode and add it as a child node to the *Player* node. That means it will start and end *between* the start and end of the *Player* node tags <Player> and </Player>.

Then we create nodes for the CurrentHitPoints, MaximumHitPoints, Gold, ExperiencePoints, and CurrentLocation. These nodes hold data, so their only child nodes are the values added to them with the CreateTextNode() function.

Notice that we don't have a node to store the Level value. That's because Level is always calculated from the ExperiencePoints.

These nodes are all child nodes of *Stats*. So, when we finish creating them, we use the AppendChild() method on the Stats node.

Then we add the *InventoryItems* node, and *InventoryItem* nodes for all the items in the player's inventory.

Notice that we are using the CreateAttribute() method to add the values to the nodes. After creating the attribute, we set its value.

Finally, we add the *PlayerQuests* node and its *PlayerQuest* child nodes (if any exist)

Since we are writing the XML value to a file, we return the *InnerXml* property of the XmlDocument, which is all the XML as a string.

This is probably a good time to rebuild your solution, and see if there are any errors, before we go to the next step.

STEP 4 The next step is to make a new constructor for the Player class that takes the XML data and creates a new Player object, with the values from it. This is where we'll use the **Factory design pattern.**

A design pattern is basically a general method, or technique, to do something. It's good to learn the common ones, especially if you work with other programmers. Then, instead of giving a long explanation of what you're doing, you can just say, *"It's a Singleton,"* or, *"Use a Decorator."* Then the other programmer will know what you're doing.

So far, in this program, whenever we want an object, we use the constructor to create an instance of the class. We're going to change that for the Player class. We're going to use a factory to create the object for us.

Replace the constructor for the Player class with the code on the next page, and add in the two new functions, *CreateDefaultPlayer()* and *CreatePlayerFromXmlString()*.

```
 1  private Player(int currentHitPoints, int maximumHitPoints, int gold, int experiencePoints)
        : base(currentHitPoints, maximumHitPoints)
 2  {
 3      Gold = gold;
 4      ExperiencePoints = experiencePoints;
 5
 6      Inventory = new List<InventoryItem>();
 7      Quests = new List<PlayerQuest>();
 8  }
 9
10  public static Player CreateDefaultPlayer()
11  {
12      Player player = new Player(10, 10, 20, 0);
13      player.Inventory.Add(new InventoryItem(World.ItemByID(World.ITEM_ID_RUSTY_SWORD), 1));
14      player.CurrentLocation = World.LocationByID(World.LOCATION_ID_HOME);
15
16      return player;
17  }
18
19  public static Player CreatePlayerFromXmlString(string xmlPlayerData)
20  {
21      try
22      {
23          XmlDocument playerData = new XmlDocument();
24
25          playerData.LoadXml(xmlPlayerData);
26
27          int currentHitPoints = Convert.ToInt32(
                playerData.SelectSingleNode("/Player/Stats/CurrentHitPoints").InnerText);
28          int maximumHitPoints = Convert.ToInt32(
                playerData.SelectSingleNode("/Player/Stats/MaximumHitPoints").InnerText);
29          int gold = Convert.ToInt32(
                playerData.SelectSingleNode("/Player/Stats/Gold").InnerText);
30          int experiencePoints = Convert.ToInt32(
                playerData.SelectSingleNode("/Player/Stats/ExperiencePoints").InnerText);
31
32          Player player = new Player(
                currentHitPoints, maximumHitPoints, gold, experiencePoints);
33
34          int currentLocationID = Convert.ToInt32(
                playerData.SelectSingleNode("/Player/Stats/CurrentLocation").InnerText);
35          player.CurrentLocation = World.LocationByID(currentLocationID);
36
37          foreach(XmlNode node in playerData.SelectNodes(
                "/Player/InventoryItems/InventoryItem"))
38          {
39              int id = Convert.ToInt32(node.Attributes["ID"].Value);
40              int quantity = Convert.ToInt32(node.Attributes["Quantity"].Value);
41
42              for(int i = 0; i < quantity; i++)
43              {
44                  player.AddItemToInventory(World.ItemByID(id));
45              }
46          }
47
```

continued on next page

```
48          foreach(XmlNode node in playerData.SelectNodes("/Player/PlayerQuests/PlayerQuest"))
49          {
50              int id = Convert.ToInt32(node.Attributes["ID"].Value);
51              bool isCompleted = Convert.ToBoolean(node.Attributes["IsCompleted"].Value);
52
53              PlayerQuest playerQuest = new PlayerQuest(World.QuestByID(id));
54              playerQuest.IsCompleted = isCompleted;
55
56              player.Quests.Add(playerQuest);
57          }
58
59          return player;
60      }
61      catch
62      {
63          // If there was an error with the XML data, return a default player object
64          return Player.CreateDefaultPlayer();
65      }
66  }
```

Notice that the constructor is now private. That means it can only be called by another function inside the Player class. We didn't need to do this. However, since we are only going to use the other two methods to create a Player object, I made it private.

So now, if you want to create a new Player object, you need to either call the CreateDefaultPlayer() function or the CreatePlayerFromXmlString() function. These methods are public. So they can be accessed by the rest of the solution. Since they are static, you can call them directly from the class, without an object (you'll see to how to do that in a minute).

The CreateDefaultPlayer() function should look familiar. It's basically the same thing that the SuperAdventure page does when it creates a new player.

The CreatePlayerFromXmlString() function is where we take the XML string with the game's data, read the values in it, and create a player object with the values from the saved game.

First, we load the string into an XmlDocument object.

To get the data that only has a single value, we use the SelectSingleNode() function on the XmlDocument. We give it the XPath of the data (like */Player/Stats/CurrentHitPoints*). The *InnerText* says to get the value. InnerText always returns a string, so we need to wrap all that with *Convert.ToInt32*, to get the number.

Think of XPath, and child nodes, kind of like directories/folders, with sub-directories/folders, on your hard disk.

For the nodes that can have multiple items (inventory items and player quests), we use the SelectNodes() function on the XmlDocument. This gets every node that matches the XPath. For each node, we create an InventoryItem, or PlayerQuest object, set its values, and add it to the player object.

Notice that this function has a *try* and a *catch* section. This is known as a **try-catch block.** What happens here is that the program tries to run everything in the *try* section. If there is an error, it does whatever is in the *catch* section.

We have this here in case there is ever a problem with the file (like, if someone tries to modify it, and messes it up). Instead of the user seeing an error, the program will execute the *catch* block and create the default player for the game.

Try-catch blocks are a very powerful tool in professional programming. Instead of having your program crash, you can have it deal with problems more gracefully.

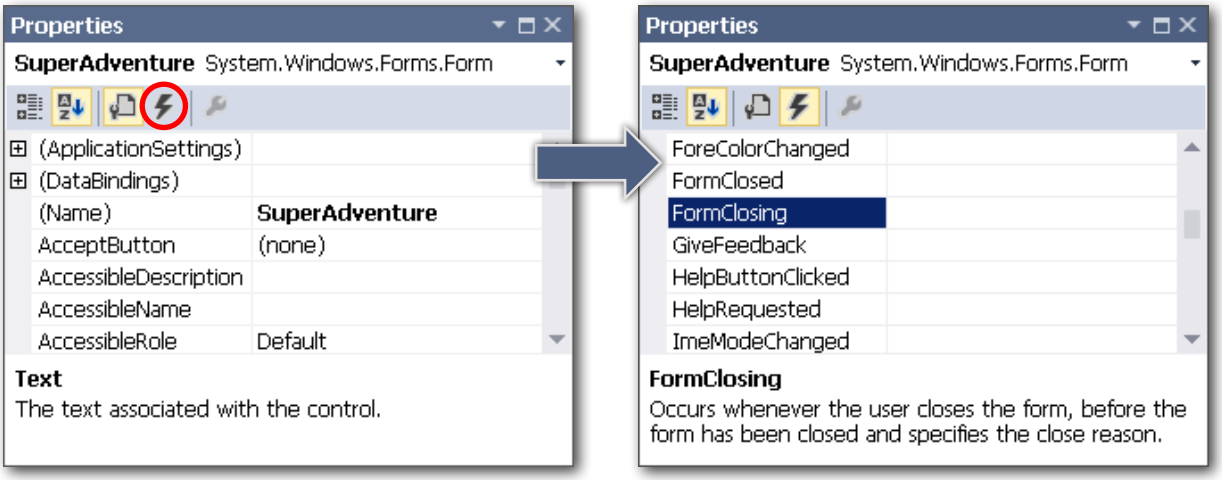
Don't try building your program now. Since we made the Player constructor private, we need to make a couple changes before we can build without errors.

STEP 5 Next, we need to have the program write the player's information to disk when they exit the program.

We're going to do this by adding code to the *FormClosing* event of the program. Whenever the program is being closed (stopped), this function will run. It will take the player's current information, using the new ToXMLString() method of the Player class, and write it to the disk, in a file named *PlayerData.XML*.

Select the SuperAdventure.cs file, and open it in design mode so you see the screen (default shortcut is Shift+F7). In the lower-right corner of Visual Studio, in the Properties box, click on the lightning symbol.

Now you see all the events that this form has. Scroll to the *FormClosing* event, and double-click on it.



That should have taken you to the code for the SuperAdventure form and created a function named *SuperAdventure_FormClosing()*. This is the function that will be run when you close the program.

It is important to create the function this way. This method adds a line of code to another file that connects the function to the FormClosing event. If you only copy/paste the new SuperAdventure.cs code into your program, that line will not be created, and this function will not ever run.

Before we add any code to the SuperAdventure_FormClosing() function, we need to include a reference to the library that lets us write to, and read from, the disk. Go to the top of the page and add this line to the section with the other *using* statements:

```
using System.IO;
```

Scroll down to the section where we define our class-level variables *_player* and *_current-Monster*. Add this new line as class-level constant:

```
private const string PLAYER_DATA_FILE_NAME = "PlayerData.xml";
```

That is the file name where we will save the player's data. We made it a constant, because it doesn't change. And it's a class-level variable, so we can use it in the function that creates the file and the function that loads the file. We need it visible in both places.

Go back to the SuperAdventure_FormClosing() function and change it to this:

```
private void SuperAdventure_FormClosing(
    object sender, FormClosingEventArgs e)
{
    File.WriteAllText(
        PLAYER_DATA_FILE_NAME, _player.ToXmlString());
}
```

Now, when the program is closing, it will write the player's data, in our XML format, to the PlayerData.xml file.

STEP 6 Finally, when the game starts, we want to look for the PlayerData.xml file. If it exists, we'll read it and create the player object from its values. If it doesn't exist, we'll assume we are starting a new game and use the normal Player constructor to create a new Player object.

Change the SuperAdventure() function to this:

```
public SuperAdventure()
{
    InitializeComponent();

    if(File.Exists(PLAYER_DATA_FILE_NAME))
    {
        _player = Player.CreatePlayerFromXmlString(
            File.ReadAllText(PLAYER_DATA_FILE_NAME));
    }
    else
    {
        _player = Player.CreateDefaultPlayer();
    }

    MoveTo(_player.CurrentLocation);
    UpdatePlayerStats();
}
```

When the game starts, it will look for the PlayerData.xml file. If the file exists, it will read it and create the Player object with the values in the file. If it doesn't find the file, it will create the default Player object.

Now you can build the solution.

Try moving the player to a new location, then exit the game and restart it. You should see that the game starts with the player in the new location.

If you ever want to have the player start over, you can manually delete the PlayerData.xml file (it will be in the same directory as SuperAdventure.exe). If you want to get fancy, you could add a menu to the game and let the play save and restore multiple games, with different file names.

But, I'm going to stop for now. We're already way over 2000 words in this lesson.

Check your work

Build the solution and make sure there are no errors in the *Output* box at the bottom of Visual Studio. If you see any problems, double-check the changes you made in this lesson.

Summary

This was a big lesson, with some new concepts.

If you want to become a professional programmer, read about design patterns and learn about good practices to use with *try-catch* blocks.

19.5 Changing dropdown default values

Lesson objectives

At the end of this lesson, you will know how to…

- ➔ Set the default selected item in a dropdown
- ➔ Prepare to make a change to an existing program

A reader mentioned a problem they saw when the player has more than one weapon. When the player moves to a new location, the dropdown for weapons always selects the first weapon in the list.

If the player wants to fight with a different weapon, the reader wanted to have the dropdown always select that weapon – unless the player manually changed weapons again.

Here is how to do that.

Planning

Before you change an existing program, it's good to think about everything it will affect. This seems like a simple change. However, we should make some additional changes when we do this. For example, store the currently selected weapon in the save game file. This way, when the player restarts the game, they will have their last weapon selected.

If we think a little more, we realize that the player may already have a saved game file, without a value for the currently selected weapon in it. So, in the code that reads the currently selected weapon from the save game file, we need to deal with a file that does not have that value in it.

This is the type of thing to watch out for when you modify a program.

Saving and reloading player information

STEP 1 Open the SuperAdventure solution in Visual Studio and select the Player class to modify.

Add a new property to store the player's current weapon. I added this under the *CurrentLocation* property (although you could add it anywhere outside of a function):

```
public Weapon CurrentWeapon { get; set; }
```

STEP 2 Open the code file for the SuperAdventure.cs form.

In the lesson where we save and restore the game state in a file, we connected a function to the *FormClosing* event. This function is an **event handler,** it *handles* the closing *event*. In that lesson, we let Visual Studio connect the event to the function automatically. For this lesson, we need to connect the function manually. You'll see why in a minute.

Add this function to SuperAdventure.cs:

```
private void cboWeapons_SelectedIndexChanged(
    object sender, EventArgs e)
{
    _player.CurrentWeapon = (Weapon)cboWeapons.SelectedItem;
}
```

This gets the selected item from the cboWeapons dropdown and saves it in the player's CurrentWeapon property.

Notice that we have *(Weapon)* in front of *cboWeapons.SelectedItem*. That's because a dropdown can hold different types of objects, and *cboWeapons.SelectedItem* could be any datatype. By adding *(Weapon)* in front of it, we are **casting** it to the Weapon datatype.

We can do this because the items in the dropdown are Weapon objects, so the casting will work. We need to do this, because the CurrentWeapon property only holds Weapon objects. It can't take a generic SelectedItem object.

STEP 3 Now we need to change the code that populates the cboWeapons dropdown. We want it to select the player's CurrentWeapon (if they have one) and connect to the cboWeapons_SelectedIndexChanged() function (for when the player changes the value in the dropdown list).

Find the *UpdateWeaponsListInUI()* function and change it to the code below:

```
private void UpdateWeaponListInUI()
{
    List<Weapon> weapons = new List<Weapon>();

    foreach(InventoryItem inventoryItem in _player.Inventory)
    {
        if(inventoryItem.Details is Weapon)
        {
            if(inventoryItem.Quantity > 0)
            {
                weapons.Add((Weapon)inventoryItem.Details);
            }
        }
    }

    if(weapons.Count == 0)
    {
        // The player doesn't have any weapons, so hide the
           weapon combobox and "Use" button
        cboWeapons.Visible = false;
        btnUseWeapon.Visible = false;
    }
    else
    {
        cboWeapons.SelectedIndexChanged -=
            cboWeapons_SelectedIndexChanged;
        cboWeapons.DataSource = weapons;
        cboWeapons.SelectedIndexChanged +=
            cboWeapons_SelectedIndexChanged;
        cboWeapons.DisplayMember = "Name";
        cboWeapons.ValueMember = "ID";

        if(_player.CurrentWeapon != null)
        {
           cboWeapons.SelectedItem = _player.CurrentWeapon;
        }
        else
        {
            cboWeapons.SelectedIndex = 0;
        }
    }
}
```

Look at the lines before and after where we set the *cboWeapons.DataSource*. These lines connect, or disconnect, a function to the *SelectedIndexChanged* event of cboWeapons.

In the line before setting the datasource, we remove the function connected to the dropdown's *SelectedIndexChanged* event (the line with the `-=` operator). That's because when you set the DataSource property of a dropdown, it automatically calls the function connected to the *SelectedIndexChanged* event. We don't want that to happen. We only want that event called when the player manually changes the value.

After the DataSource is set, we add the event handler function back to the *SelectedIndexChanged* event (with the `+=` operator). This way, the function will run when the player changes the value.

At the end of this function, we have code to check if the player has a value in their CurrentWeapon property. If they do, we set the dropdown's SelectedItem to that weapon. If the player doesn't have a CurrentWeapon, we default to the first one.

STEP 4 Now we need to update the save game and load game functions. Go back to editing the Player.cs file. Find the *ToXmlString()* function and add this code after the section for saving the current location:

```
if(CurrentWeapon != null)
{
    XmlNode currentWeapon =
        playerData.CreateElement("CurrentWeapon");
    currentWeapon.AppendChild(
        playerData.CreateTextNode(this.CurrentWeapon.ID.ToString()));
    stats.AppendChild(currentWeapon);
}
```

This will save the ID of the player's currently selected weapon (if they have one).

Next, find the *CreatePlayerFromXmlString()* function and add this code after the lines to set the CurrentLocation:

```
if(playerData.SelectSingleNode(
        "/Player/Stats/CurrentWeapon") != null)
{
    int currentWeaponID = Convert.ToInt32(
        playerData.SelectSingleNode(
            "/Player/Stats/CurrentWeapon").InnerText);
    player.CurrentWeapon = (Weapon)World.ItemByID(currentWeaponID);
}
```

This will load the player's currently selected weapon, if one exists in the save game file.

Now, we're finished. When the player changes their active weapon, that value will stay set when they move to a new location, or if they exit and resume the game. It makes the game a little easier for the player, which is always a good thing to do with your programs.

Check your work

Build the solution and make sure there are no errors in the *Output* box at the bottom of Visual Studio. If you see any problems, double-check the changes you made in this lesson.

Summary

The important thing from this lesson is that you need to think before making a change to an existing program. What else could your change affect?

This is why some programmers create unit tests and use continuous integration. With those, your can make a change and quickly know if it broke anything else.

19.6 Increase maximum hit points when the player gains a level

Lesson objectives

At the end of this lesson, you will know…

➔ How to ensure all changes for a property go through a shared method of a class

One of the most common requests from readers has been to show how to increase the player's MaximumHitPoints when they gain a level.

We could take the easy way, and add come code to recalculate the player's MaximumHit-Points after every line where we add to the player's experience points. However, that could lead to problems.

We'd need to find every place where we add to the player's experience points. In a small program like this, that isn't too difficult. In a larger program, that can be a nightmare. Plus, if we ever decide to change the increase for each level (for example, increase by 5 hit points per level, instead of 10), we'd have to go back through all the code again, and make multiple changes.

If you do this in big programs, you are almost guaranteed to miss one place. Then, you'll spend hours (or days) trying to track down why your program isn't working the way you expect.

To make this easy to maintain, we're going to add the code in one place, and make sure the rest of the program has to go through that one place.

How to ensure all changes for a property go through a common location

STEP 1 Open the SuperAdventure solution in Visual Studio and open the Player.cs class. Insert this new method after the CreateDefaultPlayer() method:

```
public void AddExperiencePoints(int experiencePointsToAdd)
{
    ExperiencePoints += experiencePointsToAdd;
    MaximumHitPoints = (Level * 10);
}
```

STEP 2 To make sure nowhere else in the program can modify the player's experience points, we need to make its *set* **accessor** private. This way, the value can only be set by other methods in the Player class. Go up to the *ExperiencePoints* property in the Player.cs class and change it to this:

```
public int ExperiencePoints { get; private set; }
```

STEP 3 Find all the places in the program that used to directly update the player's *ExperiencePoints* property and change them to use the new AddExperiencePoints() method.

Since we have a small program, we can cheat a little and build the solution. We'll get an error for every line that is currently trying to add to ExperiencePoints. If you haven't modified the program, this should be in two places in SuperAdventure.cs: line 122, when the player gets experience for completing a quest, and line 347, when the player gets experience for killing a monster.

Make these changes:

From:

```
_player.ExperiencePoints +=
    newLocation.QuestAvailableHere.RewardExperiencePoints;
```

To:

```
_player.AddExperiencePoints(
    newLocation.QuestAvailableHere.RewardExperiencePoints);
```

And from:

```
_player.ExperiencePoints += _currentMonster.RewardExperiencePoints;
```

To:

```
_player.AddExperiencePoints(_currentMonster.RewardExperiencePoints);
```

Check your work

Build the solution and make sure there are no errors in the *output* box at the bottom of Visual Studio. If you see any problems, double-check the changes you made in this lesson.

Summary

There is a programming principle called DRY – Don't Repeat Yourself.

You generally don't want to have code that does the same thing in multiple places. With repeated code, it takes longer to make changes, and you'll often end up with bugs, because you missed the change in one place. Do something like this to ensure all your program uses the same logic.

20

Improving SuperAdventure's code quality by refactoring

Binding a custom object's properties to UI controls

Binding list properties to datagridviews

Binding child list properties to a combobox

Moving the game logic functions from the UI project to the Engine project

20 Refactoring the SuperAdventure program

Right now, SuperAdventure works the way it should. But the code is a little sloppy in some places. That's because I didn't want to get into the more complex parts of programming for this beginner's guide.

However, if you've come this far, you're ready for more advanced techniques.

As you do more programming, and write larger programs, you'll want to keep your solutions, projects, and classes well-organized. You'll also want to keep the logic out of the user interface code – something we didn't do in the current project.

We're going to clean up SuperAdventure by refactoring – making changes to a program, without changing the way it works (adding new features, fixing bugs, or making it faster). These changes are only for us – so it will be easier to maintain the program and add more features in the future.

Ideally, your UI code should only do things related to receiving input and displaying output. In the current program, the UI code handles some of the game logic. It should really only handle receiving input from the player, and displaying output from the classes in the *Engine* project.

We'll continue what we started in Lesson 16.1 – refactoring the player movement function. We'll move more of the logic to the Player class, into smaller functions. We'll also **decouple** the UI code from the logic code.

20.1 Binding a custom object's properties to UI controls

Lesson objectives

At the end of this lesson, you will know…

➔ How to use databinding to automatically update an object's properties' values to UI controls (labels)

Using databinding to connect values of an object's properties to the UI

Right now, we have a lot of code in SuperAdventure.cs that reads from the Player object and modifies the labels, buttons, and datagrids in the UI. If we want to add more features to the program, we'll need to copy that code in more places. But, duplicating code is a bad programming habit.

If you have a lot of duplicated code, and you ever need to make a change, there's a good chance you'll miss one (or more) place. So we'll use a technique that automatically handles updates, after writing a few lines of code – in one place. We'll do this with **databinding.**

Whenever a property is changed in the player object, the UI will be notified, and the UI will update the appropriate control (labels, for this lesson). Think of this as a **publish/subscribe** technique. The UI will *subscribe* to any changes of the properties, and the Player object will *publish* a notification/event when one of the values changes.

For this lesson, we'll work with the integer properties: CurrentHitPoints, Gold, Experience-Points, and Level.

STEP 1 Open the SuperAdventure solution in Visual Studio.

STEP 2 For the UI to know when a value has changed, we need to send a notification from the player object.

Open the LivingCreature.cs class (since it's the base class for Player, and holds the Current-HitPoints property). Add this *using* statement, at the top of the file:

```
using System.ComponentModel;
```

Then, change this line:

```
public class LivingCreature
```

To:

```
public class LivingCreature : INotifyPropertyChanged
```

This change means that the LivingCreature class needs to implement the *INotifyProper-tyChanged* **interface.** The INotifyPropertyChanged interface is what the .NET Framework uses for databinding notifications, when a property value changes.

Add this code to the LivingCreature class, to do the notification:

```
public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged(string name)
{
    if(PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(name));
    }
}
```

The *PropertyChanged* event is what the UI will *subscribe* to.

The OnPropertyChanged() function checks if anything is subscribed to the event. If nothing is subscribed, then *PropertyChanged* will be null. If PropertyChanged is not null, then another class wants to be notified of changes, so the next line will run, and a PropertyChanged event will be *raised* (the notification will be sent out).

The final step in this class is to *raise the event*, when the CurrentHitPoints value changes. In order to do this, we need to call the OnPropertyChanged() function whenever the CurrentHitPoints value is set.

However, since CurrentHitPoints is an auto-property, we don't have any place where we can stick in the code to call that function. So, we need to change it to a property with a backing variable.

Change this:

```
public int CurrentHitPoints { get; set; }
```

To:

```
private int _currentHitPoints;

public int CurrentHitPoints
{
    get { return _currentHitPoints; }
    set
    {
        _currentHitPoints = value;
        OnPropertyChanged("CurrentHitPoints");
    }
}
```

Now, the CurrentHitPoints property is a **wrapper** for the private variable *_currentHitPoints*. When the property is set to a value, the OnPropertyChanged() function will be run, the PropertyChanged event will be raised, and the UI will update the label.

STEP 3 Now we need to make the changes to the other properties that will use databinding – the ones in the Player class. Open Player.cs and change the Gold and ExperiencePoints auto-properties to this:

```
private int _gold;
private int _experiencePoints;

public int Gold
{
    get { return _gold; }
    set
    {
        _gold = value;
        OnPropertyChanged("Gold");
    }
}

public int ExperiencePoints
{
    get { return _experiencePoints; }
    private set
    {
        _experiencePoints = value;
        OnPropertyChanged("ExperiencePoints");
        OnPropertyChanged("Level");
    }
}
```

Notice that the ExperiencePoints property raises a PropertyChanged event for both ExperiencePoints and Level. That's because we never set the Level value. It's always calculated from ExperiencePoints.

So, every time the ExperiencePoints property value changes, we'll also send a notification to update the Level. We could change this to only send a notification when the Level changes, but this extra notification won't hurt us in a small program.

Also, the Player class is using the OnPropertyChanged() function in LivingCreature. That's because Player inherits from the LivingCreature class. So, it can use any functions (and properties, and events) from LivingCreature – as long as they are scoped (visible) as *public*, *internal*, or *protected*.

STEP 4 Open SuperAdventure.cs and change the constructor function to this:

```
public SuperAdventure()
{
    InitializeComponent();

    if(File.Exists(PLAYER_DATA_FILE_NAME))
    {
        _player = Player.CreatePlayerFromXmlString(
            File.ReadAllText(PLAYER_DATA_FILE_NAME));
    }
    else
    {
        _player = Player.CreateDefaultPlayer();
    }

    lblHitPoints.DataBindings.Add(
        "Text", _player, "CurrentHitPoints");
    lblGold.DataBindings.Add(
        "Text", _player, "Gold");
    lblExperience.DataBindings.Add(
        "Text", _player, "ExperiencePoints");
    lblLevel.DataBindings.Add(
        "Text", _player, "Level");

    MoveTo(_player.CurrentLocation);
}
```

The four new lines bind the labels to the properties.

To break down what's happening, the first new line means this: For the lblHitPoints control, add a databinding – a *subscription* to a property's *notifications*. The databinding will connect to the *Text* property of lblHitPoints to the *CurrentHitPoints* property of the *_player* object.

Notice that we added the databindings *after* we created the object. The object needs to be instantiated before you can bind to it. We also removed the line that called the UpdatePlayerStats() function. We don't need to call that method anymore. The databinding will automatically do that for us.

Remove the other lines where we call the UpdatePlayerStats() function. You can find all those places by doing a search (Ctrl + F on the keyboard). Also remove any place where we set the *lblHitPoints.Text* property. We don't need to manually set that value any more.

STEP 5 Compile your program, run it, and make sure the labels in the UI update when you fight monsters.

Summary

Before, the UI would *push* values to the player object's properties, then it would *pull* values from it, to update the UI. Now, it still pushes values to the player object, but the player object *pushes* notifications of changes to the UI (and anything else that might be listening).

This is one way of *decoupling* – making your program so the classes are more independent. When your classes are more independent, it's easier to make changes. You usually need to make fewer changes, in fewer places, when your code is not highly-coupled.

20.2 Binding list properties to datagridviews

Lesson objectives

At the end of this lesson, you will know…

➔ How to bind list properties of custom classes, to automatically update in datagrid-views in the UI

Databinding list properties to the UI

In the last lesson we used databinding to connect the integer properties of the Player object to label controls in the UI (CurrentHitPoints, Level, etc.). In this lesson, we'll bind the list properties for the player's inventory and quests.

The basic principles are the same as binding an integer property; however, we need to do a few things differently.

STEP 1 Open the InventoryItem.cs class, in the Engine project.

Just like with the Player class, we need to say that this class implements the *INotifyPropertyChanged* interface. So, change this:

```
public class InventoryItem
```

To:

```
public class InventoryItem : INotifyPropertyChanged
```

While adding this *using* statement at the top of the file:

```
using System.ComponentModel;
```

Then, add the PropertyChanged event and OnPropertyChanged() function.

```
public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged(string name)
{
    if(PropertyChanged != null)
    {
        PropertyChanged(
            this, new PropertyChangedEventArgs(name));
    }
}
```

Next, change the Details and Quantity auto-properties to use backing variables and call the OnPropertyChanged() function.

```
private Item _details;
private int _quantity;

public Item Details
{
    get { return _details; }
    set
    {
        _details = value;
        OnPropertyChanged("Details");
    }
}

public int Quantity
{
    get { return _quantity; }
    set
    {
        _quantity = value;
        OnPropertyChanged("Quantity");
        OnPropertyChanged("Description");
    }
}
```

STEP 2 We want the datagridview to display a property of InventoryItem's Details property. So, we need to do one final thing to the InventoryItem class.

Add this new read-only property, that gets the item's name from the Item object. If Quantity is greater than 1, we return *NamePlural*, if it's not, we return *Name*:

```
public string Description
{
    get
    {
        return Quantity > 1 ? Details.NamePlural :
            Details.Name;
    }
}
```

This way, the InventoryItem object has a Description property we can bind to in the datagrid. We can't databind to a property of a property, without doing a lot of extra work. So, we'll use this technique.

STEP 3 Open the Player.cs class.

To bind a list property, you need to change its datatype to either **BindingList** or **ObservableCollection.** BindingList gives more options than ObservableCollection – like searching and sorting. So, we'll use that.

Add this to the *using* statements, at the top of the file:

```
using System.ComponentModel;
```

Then change the Inventory property from this:

```
public List<InventoryItem> Inventory { get; set; }
```

To this:

```
public BindingList<InventoryItem> Inventory { get; set; }
```

And change the Player.cs constructor to this, to match the change to the Inventory property's datatype:

```
private Player(int currentHitPoints, int maximumHitPoints,
    int gold, int experiencePoints) :
        base(currentHitPoints, maximumHitPoints)
{
    Gold = gold;
    ExperiencePoints = experiencePoints;

    Inventory = new BindingList<InventoryItem>();
    Quests = new List<PlayerQuest>();
}
```

One thing I don't like in the .NET framework is that the different list/collection datatypes have different ways of working with them. Because we switched from a List datatype, to a BindingList, we need to make a couple other changes.

In some of the functions of the Player class, we check if an item exists in the player's inventory – for example, to see if they have all the items needed to complete a quest. We used the Exists() method on the List properties to do this check. However, Exists() is not available with BindlingLists.

So, in the Player class, use Ctrl + F to search for *Inventory.Exists* and change them to *Inventory.Any*. The *Any()* method will return a *true*, if any of the items in the BindingList match the criteria we are looking for, just like Exists() does for a List property or variable.

You should find this in two places: the HasRequiredItemToEnterThisLocation() function and the HasAllQuestCompletionItems() function.

STEP 4 Now that the business objects are ready, we can bind them to the UI.

Open SuperAdventure.cs.

We currently call UpdateInventoryListInUI(), to update the inventory. Just like with the databinding for the integer properties, we're going to perform the databinding in SuperAdenture's constructor method, delete the old function, and delete any lines where we call the old function.

In SuperAdventure's constructor, add these lines after the databinding for the hit points, gold, etc.:

```
dgvInventory.RowHeadersVisible = false;
dgvInventory.AutoGenerateColumns = false;

dgvInventory.DataSource = _player.Inventory;

dgvInventory.Columns.Add(new DataGridViewTextBoxColumn
    {
        HeaderText = "Name",
        Width = 197,
        DataPropertyName = "Description"
    });

dgvInventory.Columns.Add(new DataGridViewTextBoxColumn
    {
        HeaderText = "Quantity",
        DataPropertyName = "Quantity"
    });
```

These lines configure the dgvInventory datagrid view.

We say that we don't want to show row headers – the blank squares to the left of each row.

We also don't want the binding to automatically generate the data grid's columns. If this value was set to *true*, the datagrid would create a column for each property of InventoryItem. We want to manually configure the columns, so we set *AutoGenerateColumns* to *false*.

Next, we say that the *DataSource* for the datagridview is going to be the player's Inventory property. This is where the data *binds* to the UI.

The next two parts are where we configure the columns of the datagridview. We add *new DataGridViewTextBoxColumns*, since we want to display text in them. There are different column types you can use if you want to display buttons – for example.

The DataPropertyName value is the property of InventoryItem that we want to display in the column. This is why we created the Description property in InventoryItem. We couldn't use *Details.Name* here. We needed a property of InventoryItem for the databinding.

STEP 5 Now we have the databinding in place, and we can get rid of the old code we used to refresh the Inventory datagridview. Delete the UpdateInventoryListInUI() function from SuperAdventure.cs. It should be around line 223.

Use Ctrl + F to search for the places in SuperAdventure.cs where we call UpdateInventory-ListInUI() and delete those lines.

STEP 6 Run the program to make sure everything works.

STEP 7 We need to repeat these steps to bind the player's Quests list property to the UI.

Change PlayerQuest.cs to the code that's on the next page.

PlayerQuest.cs

```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.ComponentModel;
 4  using System.Linq;
 5  using System.Text;
 6  using System.Threading.Tasks;
 7
 8  namespace Engine
 9  {
10      public class PlayerQuest : INotifyPropertyChanged
11      {
12          private Quest _details;
13          private bool _isCompleted;
14
15          public Quest Details
16          {
17              get { return _details; }
18              set
19              {
20                  _details = value;
21                  OnPropertyChanged("Details");
22              }
23          }
24
25          public bool IsCompleted
26          {
27              get { return _isCompleted; }
28              set
29              {
30                  _isCompleted = value;
31                  OnPropertyChanged("IsCompleted");
32                  OnPropertyChanged("Name");
33              }
34          }
35
36          public string Name
37          {
38              get { return Details.Name; }
39          }
40
41          public PlayerQuest(Quest details)
42          {
43              Details = details;
44              IsCompleted = false;
45          }
46
47          public event PropertyChangedEventHandler PropertyChanged;
48
49          protected void OnPropertyChanged(string name)
50          {
51              if(PropertyChanged != null)
52              {
53                  PropertyChanged(this, new PropertyChangedEventArgs(name));
54              }
55          }
56      }
57  }
```

STEP 8 Modify Player.cs .

Change the Quests property to a BindingList:

```
public BindingList<PlayerQuest> Quests { get; set; }
```

In the Player constructor, change the Quests to be a BindingList also:

```
Quests = new BindingList<PlayerQuest>();
```

Find the HasThisQuest() function, and change *Quests.Exists* to *Quests.Any*.

STEP 9 Modify SuperAdventure.cs .

Add this code in the constructor, to configure the datagridview and bind the player's quest list to the UI:

```
dgvQuests.RowHeadersVisible = false;
dgvQuests.AutoGenerateColumns = false;

dgvQuests.DataSource = _player.Quests;

dgvQuests.Columns.Add(new DataGridViewTextBoxColumn
    {
        HeaderText = "Name",
        Width = 197,
        DataPropertyName = "Name"
    });

dgvQuests.Columns.Add(new DataGridViewTextBoxColumn
    {
        HeaderText = "Done?",
        DataPropertyName = "IsCompleted"
    });
```

Then, delete the UpdateQuestListInUI() function and delete where it was called in the MoveTo() function.

STEP 10 Run your program, and make sure there aren't any errors.

Summary

This may not seem like the most exciting work – making all these changes, without adding any new features to the program. However, refactoring will make it much easier for you to expand the program later.

Plus, now that you know how to do databinding to the UI, you can start out using it the next time – instead of doing it the way we originally did, then making the change to this better method.

20.3 Binding child list properties to a combobox

Lesson objectives

At the end of this lesson, you will know…

➔ How to manually bind list properties of custom classes, to automatically update comboboxes in the UI

Manually binding list properties to a combobox

It seems like we could do the same thing to the comboboxes that we did to bind the Inventory to a datagridview. However, there are a couple of problems.

First, the Player.Inventory property contains a list of InventoryItem objects, and we only want a list of the Details property for each InventoryItem object. There isn't an automatic way to get those values as a bindable list.

Second, we want to apply a filter, so we only include items that are weapons (for the cboWeapons combobox) or potions (for the cboPotions combobox).

So, we need to use a different technique.

When we make this change, we'll also create a function in the Player class to remove items from inventory. Then, we will change SuperAdventure.cs to use the new function. This is another part of moving our logic code out of the UI code.

STEP 1 Open the Player.cs class in Visual Studio.

The first thing we need to do is create properties for the lists of our weapons and potions – to bind to the comboboxes.

It would be nice if we could bind the combobox to the Inventory property, and filter it to only include items that are weapons. However, there are limits to what you can do with the different types of collections and lists in .NET. So, we need to create new properties on the Player class, and do our binding manually.

Create these new properties in Player.cs:

```
public List<Weapon> Weapons
{
    get { return Inventory.Where(
        x => x.Details is Weapon).Select(
            x => x.Details as Weapon).ToList(); }
}

public List<HealingPotion> Potions
{
    get { return Inventory.Where(
        x => x.Details is HealingPotion).Select(
            x => x.Details as HealingPotion).ToList(); }
}
```

These use LINQ to create a new list of Inventory items where the Details property is the datatype Weapon (or HealingPotion). The *is* is used to check the datatype of an object. Remember that the Details property, of the InventoryItem class, holds objects whose datatype is *Item*. The Weapon class inherits from the Item class. So, Weapon objects have a datatype of *both* Weapon and Item.

Think of it like the base class is *Animal*, and the child class is *Dog*. A poodle is both an animal and a dog. So a poodle can fit in a list of Animal objects, and a list of Dog objects.

The *Select(x => x.Details)* returns only the Details property of the InventoryItem object. We don't use InventoryItem's Quantity in the comboboxes, so we only take the Details value.

The *ToList()* function converts the results of the LINQ query into a new list. We will bind these lists to the comboboxes.

STEP 2 Add this new function to Player.cs. We will use this to notify the UI when the inventory changes.

```
private void RaiseInventoryChangedEvent(Item item)
{
    if(item is Weapon)
    {
        OnPropertyChanged("Weapons");
    }

    if(item is HealingPotion)
    {
        OnPropertyChanged("Potions");
    }
}
```

When we add (or remove) anything in the Inventory list, we will call this method. If the item was a weapon, it will raise an event saying that the *Weapons* property has been changed. It will do the same for the *Potions* property, when we add or remove potions.

STEP 3 Add this new function to Player.cs. From now on, we will call this when we want to remove an item from the player's inventory.

```
public void RemoveItemFromInventory(
    Item itemToRemove, int quantity = 1)
{
    InventoryItem item = Inventory.SingleOrDefault(
        ii => ii.Details.ID == itemToRemove.ID);

    if(item == null)
    {
        // The item is not in the player's inventory,
           so ignore it.

        // We might want to raise an error for this situation
    }
    else
    {
        // They have the item in their inventory,
           so decrease the quantity
        item.Quantity -= quantity;

        // Don't allow negative quantities.
           We might want to raise an error for this situation
        if(item.Quantity < 0)
        {
            item.Quantity = 0;
        }

        // If the quantity is zero, remove the item from
           the list
        if(item.Quantity == 0)
        {
            Inventory.Remove(item);
        }

        // Notify the UI that the inventory has changed
        RaiseInventoryChangedEvent(itemToRemove);
    }
}
```

When we call this, we pass the item we want to remove, and the quantity to remove.

Notice that the quantity parameter has an *= 1* after it. That makes it an optional parameter. If we call this function with only an item, it will assume we want to remove one of it. If we want to delete more than one of that item, we can pass in the number to remove for this parameter.

There are two comments in this code about raising an error – when you try to remove an item from inventory that doesn't exist in the player's inventory, and when you try to remove a quantity larger than what they have in their inventory.

If the remaining quantity of the item is zero, we completely remove that item from the Inventory list.

In the last line, we call the function to send the UI a property change notification.

STEP 4 Now we need to change the Player class functions to use the new RemoveItemFromInventory() function – to raise the notification events when the inventory changes.

Replace the existing RemoveQuestCompletionItems() and AddItemToInventory() functions with this code (continued on next page):

```
public void RemoveQuestCompletionItems(Quest quest)
{
    foreach(QuestCompletionItem qci in
        quest.QuestCompletionItems)
    {
        // Subtract the quantity from the player's inventory
           that was needed to complete the quest
        InventoryItem item = Inventory.SingleOrDefault(
            ii => ii.Details.ID == qci.Details.ID);

        if(item != null)
        {
            RemoveItemFromInventory(
                item.Details, qci.Quantity);
        }
    }
}
```

```
public void AddItemToInventory(
    Item itemToAdd, int quantity = 1)
{
    InventoryItem item = Inventory.SingleOrDefault(
        ii => ii.Details.ID == itemToAdd.ID);

    if(item == null)
    {
        // They didn't have the item, so add it to
           their inventory
        Inventory.Add(new InventoryItem(
            itemToAdd, quantity));
    }
    else
    {
        // They have the item in their inventory,
           so increase the quantity
        item.Quantity += quantity;
    }

    RaiseInventoryChangedEvent(itemToAdd);
}
```

We want to use these functions every time we add or remove items in the player's inventory. There are not many places right now. However, if we add vendors to the game, and let the player buy and sell items, we will use these two functions. That way, we know the event notification will always be raised for the UI.

STEP 5 Now we need to modify SuperAdventure.cs. The first thing we'll do is set up the comboboxes to bind to the new Player properties.

In the constructor for SuperAdventure.cs, add these lines before the line *MoveTo(_player.CurrentLocation)*; :

```
cboWeapons.DataSource = _player.Weapons;
cboWeapons.DisplayMember = "Name";
cboWeapons.ValueMember = "Id";

if(_player.CurrentWeapon != null)
{
    cboWeapons.SelectedItem = _player.CurrentWeapon;
}

cboWeapons.SelectedIndexChanged += cboWeapons_SelectedIndexChanged;

cboPotions.DataSource = _player.Potions;
cboPotions.DisplayMember = "Name";
cboPotions.ValueMember = "Id";

_player.PropertyChanged += PlayerOnPropertyChanged;
```

These lines say to use the Weapons and Potions properties as the datasources for the comboboxes. We say what property to display in the combobox (the *DisplayMember*) and what property to use as the value (the *ValueMember*) when we check for the currently selected item.

We also set the SelectedItem to the player's CurrentWeapon – if they have one.

There are two connections to event handlers:

1. *cboWeapons_SelectedIndexChanged()*, the existing function for when the player chooses a new weapon in the combobox.
2. *PlayerOnPropertyChanged()* – a new function to update the combobox data when the player's inventory changes.

STEP 6 Add the PlayerOnPropertyChanged() function to SuperAdventure.cs. This will update the combobox data when the player's inventory changes.

```
        private void PlayerOnPropertyChanged(object sender,
            PropertyChangedEventArgs propertyChangedEventArgs)
        {
            if(propertyChangedEventArgs.PropertyName == "Weapons")
            {
                cboWeapons.DataSource = _player.Weapons;

                if(!_player.Weapons.Any())
                {
                    cboWeapons.Visible = false;
                    btnUseWeapon.Visible = false;
                }
            }

            if(propertyChangedEventArgs.PropertyName == "Potions")
            {
                cboPotions.DataSource = _player.Potions;

                if(!_player.Potions.Any())
                {
                    cboPotions.Visible = false;
                    btnUsePotion.Visible = false;
                }
            }
        }
```

The *propertyChangedEventArgs.PropertyName* tells us which property was changed on the Player object. This value comes from the *Player.RaiseInventoryChangedEvent* function, where it says *OnPropertyChanged("Weapons")*, or *OnPropertyChanged("Potions")*.

We re-bind the combobox to the Weapons (or Potions) DataSource property, to refresh it with the current items. Then, we see if the lists are empty, by using *!_player.Weapons.Any()*. Remember that Any() tells us if there are any items in the list: *true* if there are, *false* if there are not. So, we are saying, *"if there are not any items in the list, set the visibility of the combobox and 'Use' button to false (not visible)"*.

This is in case we use our last potion in the middle of a fight. Since the player's Potions property will be an empty list, it will hide the potions combobox and *Use* button.

STEP 7 With the binding in place, we can remove the old code we used to manually refresh the comboboxes, from SuperAdventure.cs .

Look for the UpdateWeaponListInUI() function and the UpdatePotionListInUI() function, and delete both of them. Use Ctrl + F to find where we called those two functions and delete those lines. They are at the end of the MoveTo() function, in the bntUseWeapon_Click() function, and in the btnUsePotion_Click() function.

STEP 8 The final change is to make the btnUsePotion_Click() function use our new RemoveItemFromInventory() function, when the player uses a potion.

Change this code:

```
// Remove the potion from the player's inventory
foreach(InventoryItem ii in _player.Inventory)
{
    if(ii.Details.ID == potion.ID)
    {
        ii.Quantity--;
        break;
    }
}
```

To this:

```
// Remove the potion from the player's inventory
_player.RemoveItemFromInventory(potion, 1);
```

Now we aren't directly changing the player's inventory from the UI. It has to pass through our function in the Player object, which will send up a notification that the player's inventory has changed.

STEP 9 We'll also want to use the new Weapons and Potions properties to show, or hide, the comboboxes and buttons when the player moves to a new location and encounters a monster.

In SuperAdventure.cs, change lines:

```
cboWeapons.Visible = true;
cboPotions.Visible = true;
btnUseWeapon.Visible = true;
btnUsePotion.Visible = true;
```

To this:

```
cboWeapons.Visible = _player.Weapons.Any();
cboPotions.Visible = _player.Potions.Any();
btnUseWeapon.Visible = _player.Weapons.Any();
btnUsePotion.Visible = _player.Potions.Any();
```

STEP 10 Run the program, and make sure it works.

Summary

We've moved more logic code where it belongs – in the classes of the Engine project. We've also provided a central place to remove items from the player's inventory.

Players won't see a change in how the game works. However, we are making it much easier for us to work with in the future.

20.4 Moving the game logic functions from the UI project to the Engine project

The main goal of this refactoring was to move these functions from SuperAdventure.cs to Player.cs.

- MoveTo() – the function that moves the player to a new location
- btnUseWeapon_click() – to attack a monster
- btnUsePotion_Click() – to use a potion during battle

However, this change could not be done with a few cut-and-pastes. Those functions contained code for both the game logic and updating the UI. When the code moved to the Player class, it would not have access to the UI controls.

Completing this refactoring required a lot of changes, in a lot of places. This lesson became *very* long, and *very* complicated. It was going to take around 30 steps, or more, to include the details for each change. So, I decided to summarize the changes I made in this lesson.

I suggest you follow this lesson by opening SuperAdventure.cs and Player.cs in Visual Studio, while viewing the updated classes for this lesson (found at scottlilly.com).

At the end, you can update your solution with the source code from the updated classes.

Moving the code to handle button clicks from the UI to the engine

First, I cut-and-pasted the MoveTo() function from SuperAdventure.cs into Player.cs. That broke the program in about 60 places.

Now, the UI button click functions need to call the MoveTo() method in the Player class. I could have made MoveTo() public. Instead, I decided to create these new functions in Player.cs, with clearer names.

```
public void MoveNorth()
{
    if(CurrentLocation.LocationToNorth != null)
    {
        MoveTo(CurrentLocation.LocationToNorth);
    }
}

public void MoveEast()
{
    if(CurrentLocation.LocationToEast != null)
    {
        MoveTo(CurrentLocation.LocationToEast);
    }
}

public void MoveSouth()
{
    if(CurrentLocation.LocationToSouth != null)
    {
        MoveTo(CurrentLocation.LocationToSouth);
    }
}

public void MoveWest()
{
    if(CurrentLocation.LocationToWest != null)
    {
        MoveTo(CurrentLocation.LocationToWest);
    }
}
```

Then, I changed the SuperAdventure functions to call the new functions in the Player class.

```
private void btnNorth_Click(object sender, EventArgs e)
{
    _player.MoveNorth();
}

private void btnEast_Click(object sender, EventArgs e)
{
    _player.MoveEast();
}

private void btnSouth_Click(object sender, EventArgs e)
{
    _player.MoveSouth();
}

private void btnWest_Click(object sender, EventArgs e)
{
    _player.MoveWest();
}
```

For the UseWeapon and UsePotion button click handlers, I created new functions in the Player class (see the next three pages). We still need functions in SuperAdventure.cs, to handle the click events. But now, all they will do is get the current weapon or potion from the dropdown and call the new function in the Player class.

This is what they look like now, in SuperAdventure.cs:

```
private void btnUseWeapon_Click(object sender, EventArgs e)
{
    // Get the currently selected weapon from the
    // cboWeapons ComboBox
    Weapon currentWeapon = (Weapon)cboWeapons.SelectedItem;

    _player.UseWeapon(currentWeapon);
}

private void btnUsePotion_Click(object sender, EventArgs e)
{
    // Get the currently selected potion from the combobox
    HealingPotion potion = (HealingPotion)cboPotions.SelectedItem;

    _player.UsePotion(potion);
}
```

The new Player functions accept the weapon, or potion, as a parameter, instead of reading it from the combobox – because the Player class cannot read the combobox controls of SuperAdventure.cs.

```
 1  public void UseWeapon(Weapon weapon)
 2  {
 3      // Determine the amount of damage to do to the monster
 4      int damageToMonster = RandomNumberGenerator.NumberBetween(
            weapon.MinimumDamage, weapon.MaximumDamage);
 5
 6      // Apply the damage to the monster's CurrentHitPoints
 7      _currentMonster.CurrentHitPoints -= damageToMonster;
 8
 9      // Display message
10      RaiseMessage("You hit the " + _currentMonster.Name +
            " for " + damageToMonster + " points.");
11
12      // Check if the monster is dead
13      if(_currentMonster.CurrentHitPoints <= 0)
14      {
15          // Monster is dead
16          RaiseMessage("");
17          RaiseMessage("You defeated the " + _currentMonster.Name);
18
19          // Give player experience points for killing the monster
20          AddExperiencePoints(_currentMonster.RewardExperiencePoints);
21          RaiseMessage("You receive " + _currentMonster.RewardExperiencePoints +
                " experience points");
22
23          // Give player gold for killing the monster
24          Gold += _currentMonster.RewardGold;
25          RaiseMessage("You receive " + _currentMonster.RewardGold + " gold");
26
27          // Get random loot items from the monster
28          List<InventoryItem> lootedItems = new List<InventoryItem>();
29
30          // Add items to the lootedItems list, comparing a random number
               to the drop percentage
31          foreach(LootItem lootItem in _currentMonster.LootTable)
32          {
33              if(RandomNumberGenerator.NumberBetween(1, 100) <= lootItem.DropPercentage)
34              {
35                  lootedItems.Add(new InventoryItem(lootItem.Details, 1));
36              }
37          }
38
39          // If no items were randomly selected, then add the default loot item(s).
40          if(lootedItems.Count == 0)
41          {
42              foreach(LootItem lootItem in _currentMonster.LootTable)
43              {
44                  if(lootItem.IsDefaultItem)
45                  {
46                      lootedItems.Add(new InventoryItem(lootItem.Details, 1));
47                  }
48              }
49          }
50
```

continued on next page

```
51          // Add the looted items to the player's inventory
52          foreach(InventoryItem inventoryItem in lootedItems)
53          {
54              AddItemToInventory(inventoryItem.Details);
55
56              if(inventoryItem.Quantity == 1)
57              {
58                  RaiseMessage("You loot " +
                        inventoryItem.Quantity + " " + inventoryItem.Details.Name);
59              }
60              else
61              {
62                  RaiseMessage("You loot " + inventoryItem.Quantity +
                        " " + inventoryItem.Details.NamePlural);
63              }
64          }
65
66          // Add a blank line to the messages box, just for appearance.
67          RaiseMessage("");
68
69          // Move player to current location (to heal player and create
               a new monster to fight)
70          MoveTo(CurrentLocation);
71      }
72      else
73      {
74          // Monster is still alive
75
76          // Determine the amount of damage the monster does to the player
77          int damageToPlayer = RandomNumberGenerator.NumberBetween(
                0, _currentMonster.MaximumDamage);
78
79          // Display message
80          RaiseMessage("The " + _currentMonster.Name + " did " +
                damageToPlayer + " points of damage.");
81
82          // Subtract damage from player
83          CurrentHitPoints -= damageToPlayer;
84
85          if(CurrentHitPoints <= 0)
86          {
87              // Display message
88              RaiseMessage("The " + _currentMonster.Name + " killed you.");
89
90              // Move player to "Home"
91              MoveHome();
92          }
93      }
94  }
```

continued on next page

UsePotion(), MoveHome()

```
public void UsePotion(HealingPotion potion)
{
    // Add healing amount to the player's current hit points
    CurrentHitPoints = (CurrentHitPoints + potion.AmountToHeal);

    // CurrentHitPoints cannot exceed player's MaximumHitPoints
    if(CurrentHitPoints > MaximumHitPoints)
    {
        CurrentHitPoints = MaximumHitPoints;
    }

    // Remove the potion from the player's inventory
    RemoveItemFromInventory(potion, 1);

    // Display message
    RaiseMessage("You drink a " + potion.Name);

    // Monster gets their turn to attack

    // Determine the amount of damage the monster does to the player
    int damageToPlayer = RandomNumberGenerator.NumberBetween(
        0, _currentMonster.MaximumDamage);

    // Display message
    RaiseMessage("The " + _currentMonster.Name +
        " did " + damageToPlayer + " points of damage.");

    // Subtract damage from player
    CurrentHitPoints -= damageToPlayer;

    if(CurrentHitPoints <= 0)
    {
        // Display message
        RaiseMessage("The " + _currentMonster.Name + " killed you.");

        // Move player to "Home"
        MoveHome();
    }
}

private void MoveHome()
{
    MoveTo(World.LocationByID(World.LOCATION_ID_HOME));
}
```

Fixing the class-level variables

SuperAdventure.cs holds the Player object in the class-level variable *_player*. The MoveTo(), UseWeapon(), and UsePotion() functions all used the methods and properties of the _player object. Now that the code is inside the Player class, we don't need to do that anymore.

For example, in the old MoveTo() function, we had lines like this:

```
if(!_player.HasRequiredItemToEnterThisLocation(newLocation))
```

That line is saying, *"for the Player object that we are storing in the _player variable, call the HasRequiredItemToEnterThisLocation() function"*.

After moving the MoveTo() function inside the Player class, we change it to this:

```
if(!HasRequiredItemToEnterThisLocation(newLocation))
```

The function knows to look for HasRequiredItemToEnterThisLocation() on itself, the Player object it is *inside*. So, we need to remove all the references to *_player,* from the code we added into the Player class. When I did that, it eliminated many errors.

We also need to handle the _currentMonster variable that was a class-level variable in SuperAdventure.cs. We stored the current monster, from the current location, in that variable so we could use it during combat (in the UseWeapon() function). But now that the UseWeapon() function is in the Player class, we need to make the _currentMonster variable a class-level variable in the Player class.

So, I deleted the _currentMonster declaration that was in SuperAdventure.cs, and added it to Player.cs. That eliminated a few more errors.

Notifying the UI of changes

Just like with the previous refactoring lessons, we're using events to communicate between the Engine classes and the UI. We now raise a PropertyChanged event when the player's CurrentLocation changes, just like we did with CurrentHitPoints.

I changed the CurrentLocation auto-property in the Player class, to use a backing variable, and raise an event when it is changed.

```
private Location _currentLocation;

public Location CurrentLocation
{
    get { return _currentLocation; }
    set
    {
        _currentLocation = value;
        OnPropertyChanged("CurrentLocation");
    }
}
```

The UI sees the event and updates the text showing the current location's name and description and makes the weapon and potion drop-downs and buttons *visible* (if there is a monster at the location, and the player has weapons or potions). This is done by adding this code to the PlayerOnPropertyChanged() function in SuperAdventure.cs.

```
if(propertyChangedEventArgs.PropertyName ==
    "CurrentLocation")
{
    // Show/hide available movement buttons
    btnNorth.Visible = (
        _player.CurrentLocation.LocationToNorth != null);
    btnEast.Visible = (
        _player.CurrentLocation.LocationToEast != null);
    btnSouth.Visible = (
        _player.CurrentLocation.LocationToSouth != null);
    btnWest.Visible = (
        _player.CurrentLocation.LocationToWest != null);

    // Display current location name and description
    rtbLocation.Text = _player.CurrentLocation.Name +
        Environment.NewLine;
    rtbLocation.Text += _player.CurrentLocation.Description +
        Environment.NewLine;

    if(_player.CurrentLocation.MonsterLivingHere == null)
    {
        cboWeapons.Visible = false;
        cboPotions.Visible = false;
        btnUseWeapon.Visible = false;
        btnUsePotion.Visible = false;
    }
    else
    {
        cboWeapons.Visible = _player.Weapons.Any();
        cboPotions.Visible = _player.Potions.Any();
        btnUseWeapon.Visible = _player.Weapons.Any();
        btnUsePotion.Visible = _player.Potions.Any();
    }
}
```

That lets us delete any lines in the Player functions that tried to write to rtbLocation, or change the visibility of the buttons. More errors eliminated.

Creating a custom event argument

We also need to use a new notification technique. This one will handle everything we display in rtbMessages. With the previous notifications, the Player class raised an event, and the UI read the new values from the _player object's properties. We can't do that with the messages – there is no *message* property in the Player class, with a value for the UI to read.

For the message events, we want to send the text of the message *along with* the event notification. To do this, we will use a custom event argument class. This is like saying, *"This event happened, and here's all the information you need to know about it."*

Here's the new class we need to add to the Engine project, MessageEventArgs.cs:

```
using System;

namespace Engine
{
    public class MessageEventArgs : EventArgs
    {
        public string Message { get; private set; }
        public bool AddExtraNewLine { get; private set; }

        public MessageEventArgs(string message, bool addExtraNewLine)
        {
            Message = message;
            AddExtraNewLine = addExtraNewLine;
        }
    }
}
```

The *: EventArgs* is for this class to inherit from the base EventArgs class, a built-in class for event notifications. All custom event argument classes need to inherit from EventArgs.

There are two auto-properties in the class. They hold the Message, and a Boolean for if we want to add a blank line after the message, for spacing.

In the Player class, our eventhandler code will be a little different from what we used before (see LivingCreature.cs). It looks like this:

```
public event EventHandler<MessageEventArgs> OnMessage;
```

The *EventHandler<MessageEventArgs>* signifies that the Player class will send an event notification with a MessageEventArgs object – the object with the message text we want to display.

Then, we add this function in the Player class, to raise the event:

```
private void RaiseMessage(
    string message, bool addExtraNewLine = false)
{
    if(OnMessage != null)
    {
        OnMessage(this, new MessageEventArgs(
            message, addExtraNewLine));
    }
}
```

When this function raises the event, it passes a MessageEventArgs(), with the values the UI code needs. Inside SuperAdventure.cs, we need to handle these events.

This line is added to SuperAdventure.cs's constructor, to watch for these events:

```
_player.OnMessage += DisplayMessage;
```

And we add this DisplayMessage() function to run when the OnMessage event is raised, and we want to add the new message to the UI:

```
private void DisplayMessage(
    object sender, MessageEventArgs messageEventArgs)
{
    rtbMessages.Text +=
        messageEventArgs.Message + Environment.NewLine;

    if(messageEventArgs.AddExtraNewLine)
    {
        rtbMessages.Text += Environment.NewLine;
    }

    rtbMessages.SelectionStart = rtbMessages.Text.Length;
    rtbMessages.ScrollToCaret();
}
```

Notice that I also moved the old ScrollToBottom() code into this function. When the UI receives a message event, it will display the text and automatically scroll to the bottom of the rich text box.

Finally, I changed the code in the MoveTo(), UseWeapon(), and UsePotion() functions to raise this event, instead of trying to write directly to rtbMessage.

So, if we previously wrote a line like this, to send a message to rtbMessage:

```
rtbMessages.Text += "You receive: " + Environment.NewLine;
```

We'll now write:

```
RaiseMessage("You receive: ");
```

Making the changes in your solution

At this point, everything was working again. I ran the program, and ensured it still worked the same – and it did. The refactoring was done, and it was time to check these changes into source control.

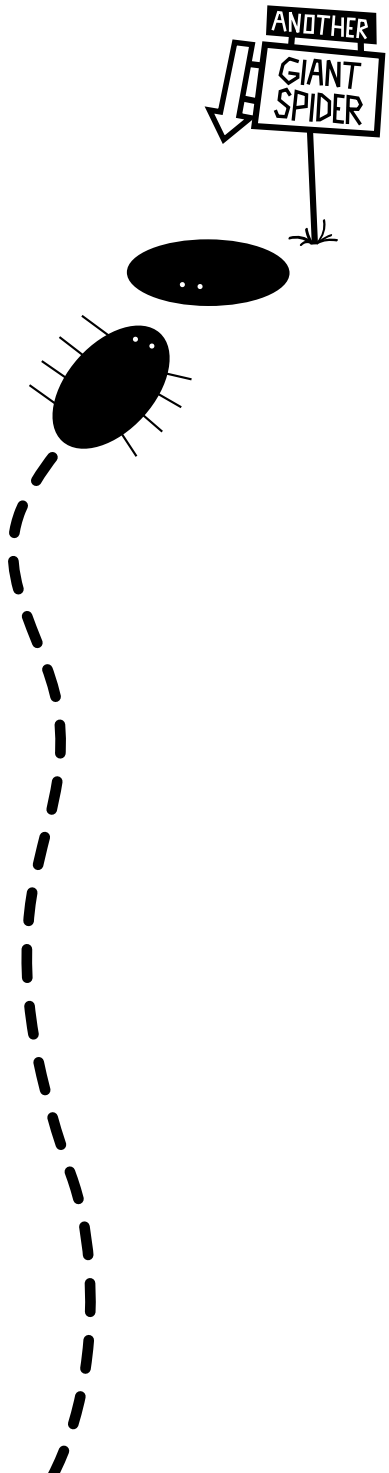
In your version of SuperAdventure, add the new MessageEventArgs class to the Engine project, and copy-paste the refactored code into SuperAdventure.cs and Player.cs. Then run your game, to ensure it works.

Summary

Sometimes, when you start refactoring, you need to make a lot of changes to get the program running again. But, if you're doing refactoring correctly, you end up with much better code.

For example, after these changes, we could probably write a WPF or ASP.Net (web-based) version of the game very quickly. You would only need to create a new UI project (a WPF app, or an ASP.Net web app), add a reference to Engine, and create a simple UI page that does the same type of binding that SuperAdventure.cs does.

Also, by moving all the logic into a class library, we could easily create unit tests to ensure that our classes work correctly.


ANOTHER
GIANT
SPIDER

21

Adding a vendor to locations

(with buying and selling items)

Plans for adding a vendor to locations

Adding a price to game items

Create the vendor class and add it to locations

Add a button and create its eventhandler in code, without the UI design

21 Plans for adding a vendor to locations

Many people have asked about adding a vendor to the game. So, that's what we will do in the next few lessons.

The first thing to do, when making any big change to a program, is to create a plan. A good plan will let you make your change faster. It will also prevent you from finishing half your changes, and realizing you need to delete all those changes and try a different method.

So, let's figure out what we need to do.

Features for adding a vendor

Here are the changes we need to make:

1. Add a price to items
2. Create a new Vendor class. This class will have properties for the vendor's name and their inventory.
3. Change the Location class, to hold an (optional) Vendor at that location
4. Change the UI to let the player buy and sell items with the vendor

Plan how to implement the changes

The first three changes don't require any new skills. We've already created classes and added properties. So those should be simple to add.

The fourth change (the UI to buy and sell) could be done a few different ways. The way I'm going to do it is with a new UI form. When the player moves to a location with a vendor, we will display a button to *open the store*. This will open a new form, which will show a list of the player's inventory and the vendor's inventory. The inventory lists will have buttons to *buy* (from the vendor) or *sell* (to the vendor).

We could keep this all on the SuperAdventure.cs UI form, by making it bigger. But, because we'll use a new form, you'll see how to pass objects between forms – which is a useful thing to learn.

21.1 Adding a price to game items

If we want the player to be able to buy and sell items, we need to add prices to each Item object.

STEP 1 In the Engine project, edit the Item.cs class.

Add an integer *Price* auto-property to the class, add a price parameter to the constructor, and set the Price property's value in the constructor.

The Item class should look like this now:

```
namespace Engine
{
    public class Item
    {
        public int ID { get; set; }
        public string Name { get; set; }
        public string NamePlural { get; set; }
        public int Price { get; set; }

        public Item(int id, string name,
            string namePlural, int price)
        {
            ID = id;
            Name = name;
            NamePlural = namePlural;
            Price = price;
        }
    }
}
```

STEP 2 Now that the Item constructor requires a price parameter, we need to change the classes that use Item as their base class – Weapon and HealingPotion.

We will change their constructors to accept a *price* parameter, and pass it to the base (Item) class constructor. This is how they should look now:

Weapon.cs

```
namespace Engine
{
    public class Weapon : Item
    {
        public int MinimumDamage { get; set; }
        public int MaximumDamage { get; set; }

        public Weapon(int id, string name, string namePlural,
            int minimumDamage, int maximumDamage, int price) :
               base(id, name, namePlural, price)
        {
            MinimumDamage = minimumDamage;
            MaximumDamage = maximumDamage;
        }
    }
}
```

HealingPotion.cs

```
namespace Engine
{
    public class HealingPotion : Item
    {
        public int AmountToHeal { get; set; }

        public HealingPotion(int id, string name, string
            namePlural, int amountToHeal, int price) :
                base(id, name, namePlural, price)
        {
            AmountToHeal = amountToHeal;
        }
    }
}
```

STEP 3 The next change will be to the World class, where we create instances of the items. Change the PopulateItems() function to this (you can use your own prices for the items):

```
private static void PopulateItems()
{
    Items.Add(new Weapon(ITEM_ID_RUSTY_SWORD,
        "Rusty sword", "Rusty swords", 0, 5, 5));
    Items.Add(new Item(ITEM_ID_RAT_TAIL,
        "Rat tail", "Rat tails", 1));
    Items.Add(new Item(ITEM_ID_PIECE_OF_FUR,
        "Piece of fur", "Pieces of fur", 1));
    Items.Add(new Item(ITEM_ID_SNAKE_FANG, "Snake fang",
        "Snake fangs", 1));
    Items.Add(new Item(ITEM_ID_SNAKESKIN,
        "Snakeskin", "Snakeskins", 2));
    Items.Add(new Weapon(ITEM_ID_CLUB,
        "Club", "Clubs", 3, 10, 8));
    Items.Add(new HealingPotion(ITEM_ID_HEALING_POTION,
        "Healing potion", "Healing potions", 5, 3));
    Items.Add(new Item(ITEM_ID_SPIDER_FANG,
        "Spider fang", "Spider fangs", 1));
    Items.Add(new Item(ITEM_ID_SPIDER_SILK,
        "Spider silk", "Spider silks", 1));
    Items.Add(new Item(ITEM_ID_ADVENTURER_PASS,
        "Adventurer pass", "Adventurer passes", -1));
}
```

NOTE:
For the Adventurer pass, we set the item's value to -1. We're going to use that as a **flag** (indicator) value. In our code to display the player's items, we won't include any items that have a value of -1. Those will be unsellable items.

STEP 4 So you can see how I think when I'm programming, I left the *Note* in step 3.

However, if I need to add a note for the lesson, that might also mean something is not clear in the code. I don't want to make a change in the future, and forget why some items have a value of -1. This is a **code smell** – a sign that the source code may have a problem.

Let's make this easier to understand.

In the World class, add this new class-level, public constant, near the other constants at the top of the class:

```
public const int UNSELLABLE_ITEM_PRICE = -1;
```

Then, change the Adventurer pass line in PopulateItems() to this:

```
Items.Add(new Item(ITEM_ID_ADVENTURER_PASS,
    "Adventurer pass", "Adventurer passes",
        UNSELLABLE_ITEM_PRICE));
```

Now, it is very clear that we don't want the player to be able to sell adventurer passes. We won't ever be confused by a **magic number** – a hard-coded value that has a special meaning.

STEP 5 Run your program, and make sure it still works. Right now, you won't see the price anywhere. But, it's good to check your program after each change.

21.2 Create the vendor class and add it to locations

For the vendors, we will use UI binding to their inventory, the same way we do with the Player class. But vendors will not have hit points, experience, etc. So, the Vendor class will be similar to the Player class, but with less properties and functions.

STEP 1 In the Engine project, create a new class – Vendor.cs. You'll find the code for the class on the next two pages.

We're only going to have the vendor's name and inventory as properties in this class. We will add, and remove items, for the vendor's inventory with functions that will raise the property changed notification.

STEP 2 Modify Location.cs. Add a new property to hold the Vendor.

```
public Vendor VendorWorkingHere { get; set; }
```

STEP 3 Modify World.cs.

In the PopulateLocations() function, create a new vendor object, and give it some inventory, for any locations where you want a vendor. I decided to add a vendor to the town square. Now, that part of PopulateLocations looks like this:

```
Location townSquare = new Location(LOCATION_ID_TOWN_SQUARE,
    "Town square", "You see a fountain.");

Vendor bobTheRatCatcher = new Vendor("Bob the Rat-Catcher");

bobTheRatCatcher.AddItemToInventory(
    ItemByID(ITEM_ID_PIECE_OF_FUR), 5);
bobTheRatCatcher.AddItemToInventory(
    ItemByID(ITEM_ID_RAT_TAIL), 3);

townSquare.VendorWorkingHere = bobTheRatCatcher;
```

STEP 4 Run your program, and make sure it still works. We still won't see anything new in the UI with these changes. That will happen in the next lesson.

```
 1  using System.Collections.Generic;
 2  using System.ComponentModel;
 3  using System.Linq;
 4
 5  namespace Engine
 6  {
 7      public class Vendor : INotifyPropertyChanged
 8      {
 9          public string Name { get; set; }
10          public BindingList<InventoryItem> Inventory { get; private set; }
11
12          public Vendor(string name)
13          {
14              Name = name;
15              Inventory = new BindingList<InventoryItem>();
16          }
17
18          public void AddItemToInventory(Item itemToAdd, int quantity = 1)
19          {
20              InventoryItem item = Inventory.SingleOrDefault(
                    ii => ii.Details.ID == itemToAdd.ID);
21
22              if(item == null)
23              {
24                  // They didn't have the item, so add it to their inventory
25                  Inventory.Add(new InventoryItem(itemToAdd, quantity));
26              }
27              else
28              {
29                  // They have the item in their inventory, so increase the quantity
30                  item.Quantity += quantity;
31              }
32
33              OnPropertyChanged("Inventory");
34          }
35
```

continued on next page

```
36          public void RemoveItemFromInventory(Item itemToRemove, int quantity = 1)
37          {
38              InventoryItem item = Inventory.SingleOrDefault(
                    ii => ii.Details.ID == itemToRemove.ID);
39
40              if(item == null)
41              {
42                  // The item is not in the player's inventory, so ignore it.
43                  // We might want to raise an error for this situation
44              }
45              else
46              {
47                  // They have the item in their inventory, so decrease the quantity
48                  item.Quantity -= quantity;
49
50                  // Don't allow negative quantities.
51                  // We might want to raise an error for this situation
52                  if(item.Quantity < 0)
53                  {
54                      item.Quantity = 0;
55                  }
56
57                  // If the quantity is zero, remove the item from the list
58                  if(item.Quantity == 0)
59                  {
60                      Inventory.Remove(item);
61                  }
62
63                  // Notify the UI that the inventory has changed
64                  OnPropertyChanged("Inventory");
65              }
66          }
67
68          public event PropertyChangedEventHandler PropertyChanged;
69
70          private void OnPropertyChanged(string name)
71          {
72              if(PropertyChanged != null)
73              {
74                  PropertyChanged(this, new PropertyChangedEventArgs(name));
75              }
76          }
77      }
78  }
```

21.3 Add a button and create its eventhandler in code, without the UI design screen

Lesson objectives

At the end of this lesson, you will know…

- ➔ How the Designer.cs file holds information about the controls (buttons, labels, etc.) on a form
- ➔ How to add a button in code and its event handler (the button *click* event) connection, without using the drag-and-drop UI

We're going to add a *Trade* button that will be visible when the current location has a vendor. However, instead of using the *Design* mode, we're going to create it with code. We will do this in the SuperAdventure form's Designer.cs file.

Knowing how these designer forms work will also help you if you accidentally created an event on your form.

STEP 1 Open the SuperAdventure solution in Visual Studio.

Inside the SuperAdventure project, double-click on the file *SuperAdventure.Designer.cs,* to open it for editing. If you cannot see this file, you may need to click on the triangle on the left side of the SuperAdventure.cs file. That will show you all of SuperAdventure.cs's files.

Look near the top lines of SuperAdventure.Designer.cs. You will see this line:

```
partial class SuperAdventure
```

If you view the code for SuperAdventure.cs, you will see this similar line:

```
public partial class SuperAdventure : Form
```

Both these files say they are a *partial class* for SuperAdventure.

A partial class is when you have some of the code for a class in one file, and more code for that class in another file. Together, these files have *all* the code needed for the SuperAdventure class.

Visual Studio creates the designer classes to keep the control information (name, location, size, etc.) out of the *logic* part of the code. It's a little easier to work with all your UI configuration code in one file, and your logic code in a different file.

If you scroll through the Designer.cs file, you will see the information for all your labels, buttons, comboboxes, etc.

STEP 2 For this step, your line numbers might be slightly different than mine. If they are, that's OK. Just put the code in the areas with the same type of code.

Inside SuperAdventure.Designer.cs, find the InitializeComponent() function. This is where you create the object on the form.

Add this as line 52 (or wherever this section is in your file), in the function:

```
this.btnTrade = new System.Windows.Forms.Button();
```

Scroll down to line 254 (after the settings for the dgvQuests datagrid) and add these lines:

```
//
// btnTrade
//
this.btnTrade.Location = new System.Drawing.Point(493, 620);
this.btnTrade.Name = "btnTrade";
this.btnTrade.Size = new System.Drawing.Size(75, 23);
this.btnTrade.TabIndex = 21;
this.btnTrade.Text = "Trade";
this.btnTrade.UseVisualStyleBackColor = true;
this.btnTrade.Click +=
    new System.EventHandler(this.btnTrade_Click);
```

As you can probably guess, this is where you add more information about how to define *btnTrade*, the new button we want on the screen.

The line with *this.btnTrade.Click* is the eventhandler code. It says, *"When the user clicks this button, run the btnTrade_Click function in this class"*. We will add that function in a minute.

Scroll to line 270, and add this line:

```
this.Controls.Add(this.btnTrade);
```

That line adds btnTrade to the controls on this screen.

Finally, scroll to line 325, and add this line:

```
private System.Windows.Forms.Button btnTrade;
```

This makes the button a class-level variable, so it can be used by any functions in the class – including functions in SuperAdventure.cs, since it is part of the SuperAdventure class (the other *partial* class file).

STEP 3 Edit SuperAdventure.cs.

Add this eventhandler function. It does not do anything. However, since we declared this eventhandler in SuperAdventure.Designer.cs, we need to have it in the class.

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

STEP 4 Compile and run your program. Right now, you should see the *Trade* button on every location, although it doesn't do anything when you click it.

Summary

This should give you an idea how the drag-and-drop designer works. When you drag a button on to a form, Visual Studio creates all this code for you. If you double-click a button, it creates the eventhandler, and the line in the Designer.cs file that connects the button to it.

This is also where you can look if you accidentally created an eventhandler.

Several people have accidentally double-clicked on a datagrid (or some other UI control) on the graphic (Design) screen for SuperAdventure.cs. This created an eventhandler in SuperAdventure.Designer.cs, and the event-handling function in SuperAdventure.cs.

When they copy-pasted the code from the lesson into SuperAdventure.cs, it didn't have the accidental event-handling function. However, *SuperAdventure.Designer.cs* still had the eventhandler in it. That caused an error, since the eventhandler is trying to reference a function that does not exist any more.

21.4 Completing the trading screen

Lesson objectives

At the end of this lesson, you will know…

- ➔ How to open a new form with a button click on a different form
- ➔ How to pass variables to a different form
- ➔ The difference between passing a variable *by value,* and *by reference*

There are several steps needed, to finish adding the trading screen. When you finish each step, run the program, to be sure the change works. If there is a problem, it will be much easier to find out where it is if you have only made one change.

STEP 1 In the SuperAdventure project, create a new Windows form named *TradingScreen.cs*. Do this by, right-clicking on the SuperAdventure (UI) project, and selecting Add ➡ Windows Form.

In Visual Studio's Solution Explorer, select the TradingScreen file, to edit in design (graphic) mode. Set the form's properties to these values, using the *Properties* section in the lower-right corner of Visual Studio:

Property	Value
Text	Trade
Size (Width)	544
Size (Height)	349

Next, add these controls to the screen:

Labels

Name	Text	Location (X)	Location (Y)
lblMyInventory	My Inventory	99	13
lblVendorInventory	Vendor's Inventory	349	13

DataGridViews

Name	Location (X)	Location (Y)	Size (Width)	Size (Height)
dgvMyItems	13	43	240	216
dgvVendorItems	276	43	240	216

Create one button, and set its properties to these values:

Button

Property	Value
Text	Close
Name	btnClose
Location (X)	441
Location (Y)	274
Size (Width)	75
Size (Height)	23

Then, create an eventhandler for the button's Click event. You can do this by:

1. Double-click on the btnClose button, in the Design (UI) editor.
2. Or, select the button in the Design editor, click on the lightning bolt symbol (in the properties area of Visual Studio), and type in the value *btnClose_Click* for the Click event.

Both of these methods will create the eventhandler in TradingScreen.Designer.cs, and the empty function in TradingScreen.cs.

Edit TradingScreen.cs, so the btnClose_Click function looks like this:

```
private void btnClose_Click(object sender, EventArgs e)
{
   Close();
}
```

Now, when the user clicks the *Close* button, the TradingScreen form will close (not be visible anymore).

STEP 2 Now, let's open up the trading screen when the user clicks on the *Trade* button.

Edit SuperAdventure.cs, and change the btnTrade_Click function to this:

```
private void btnTrade_Click(object sender, EventArgs e)
{
   TradingScreen tradingScreen = new TradingScreen();
   tradingScreen.StartPosition = FormStartPosition.CenterParent;
   tradingScreen.ShowDialog(this);
}
```

When the player clicks the *Trade* button, this btnTrade_Click function will run.

It will create a new object/instance of the TradingScreen form, and set its position to the center of its parent. The SuperAdventure form is its parent, because that is where we created the object.

The ShowDialog() function is what makes the TradingScreen form display itself.

Build and run the game, to make sure these changes work.

You should be able to click on the *Trade* button, see the new trading screen appear, and click the *Close* button to remove it.

STEP 3 When we display this form, we want to populate the datagridviews with the inventories – ours and the vendor's. So, we need to get that information to the form. I'll show you two ways to do that.

Method 1 – Set a property on the form

The first way is to create a public property on the new form.

Edit TradingScreen.cs, and add this property:

```
public Player CurrentPlayer { get; set; }
```

You also need to add this *using* statement, at the top of TradingScreen.cs, for the form to know where the Player class is:

```
using Engine;
```

Now that the form has this public property, change the btnTrade_Click function, in SuperAdventure.cs, to this:

```
private void btnTrade_Click(object sender, EventArgs e)
{
   TradingScreen tradingScreen = new TradingScreen();
   tradingScreen.StartPosition = FormStartPosition.CenterParent;
   tradingScreen.CurrentPlayer = _player;
   tradingScreen.ShowDialog(this);
}
```

After we instantiate/create the form object, we set its CurrentPlayer property to the _player object in SuperAdventure.cs. So, the functions inside TradingScreen.cs will be able to work with our _player object.

NOTE:
When we set CurrentPlayer to the _player object, the game *does not* create a copy of our _player object, and make a second variable inside TradingScreen.cs. It points to the exact same variable/object that is already in SuperAdventure.

So, if we do something to TradingScreen.CurrentPlayer (such as, sell an item and remove it from CurrentPlayer's inventory), we will see that item is gone when we look at _player's inventory.

This is called using/**passing** a variable **by reference**. There is not a second copy of the variable, only a *reference* to the original variable. For more details, and samples, of how this works, you can read my post *C# – Difference between passing variables by reference and by value* at scottlilly.com.

Method 2 – Pass the variable in the constructor

The second way to pass the _player object to another form is to make it a parameter in the constructor. Forms are classes, and have constructors, just like any other class. So, you can also use parameters with them.

To pass the _player object as a parameter, you would make these changes:

In *TradingScreen.cs,* add this line, inside the class, but outside of any functions, to create a private *class-level* variable.

```
private Player _currentPlayer;
```

Change the constructor to accept a Player parameter, and set the private variable to that value:

```
public TradingScreen(Player player)
{
   _currentPlayer = player;
   InitializeComponent();
}
```

In *SuperAdventure.cs,* change the btnTrade_Click function to this:

```
private void btnTrade_Click(object sender, EventArgs e)
{
   TradingScreen tradingScreen = new TradingScreen(_player);
   tradingScreen.StartPosition = FormStartPosition.CenterParent;
   tradingScreen.ShowDialog(this);
}
```

Now, when we create the new TradingScreen object, we pass in the _player object. When passing a variable as a parameter, it's still used *by reference*. So, any changes we do to _currentPlayer, in TradingScreen.cs, will also be seen/done in _player, in SuperAdventure.cs.

I'm going to use the version where we pass the _player object as a parameter in the constructor, and use the _currentPlayer variable in the functions in TradingScreen.cs. If you want to use the property, you will need to change the rest of the code in this lesson, so it uses *CurrentPlayer,* instead of *_currentPlayer.*

After you finish this step, build and run the program. Check that the trading screen displays when you click the *Trade* button.

STEP 4 The next step is to only show the *Trade* button when there is a vendor at the location. If we don't do this, we will see an error when we try to bind the non-existent vendor's inventory to the datagridview.

In SuperAdventure.cs's PlayerOnPropertyChanged() function, find the IF statement where we make the changes when the *PropertyName* == "*CurrentLocation*".

Add this line inside that IF:

```
btnTrade.Visible =
    (_player.CurrentLocation.VendorWorkingHere != null);
```

STEP 5 For the datagridviews on TradingScreen, we need to include the inventory item's ID and price. To do this, we need to make a change similar to what we did to display its description.

Edit InventoryItem.cs, and add these new properties:

```
public int ItemID
{
   get { return Details.ID; }
}

public int Price
{
   get { return Details.Price; }
}
```

Now, we can bind to these properties on InventoryItem, and they will show the values from the properties of the Details (Item) object.

STEP 6 We're finally ready to display the inventories in the datagridviews.

This will be similar to the way we used databinding for dgvInventory, in the constructor of SuperAdventure.cs. The big difference is a new column type that displays a button in each row, to buy or sell that item.

Edit TradingScreen.cs. Change the constructor code to the code on the next two pages, and add these new functions: *dgvMyItems_CellClick()* and *dgvVendorItems_CellClick()*.

```
public TradingScreen(Player player)
{
    _currentPlayer = player;

    InitializeComponent();

    // Style, to display numeric column values
    DataGridViewCellStyle rightAlignedCellStyle = new DataGridViewCellStyle();
    rightAlignedCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;

    // Populate the datagrid for the player's inventory
    dgvMyItems.RowHeadersVisible = false;
    dgvMyItems.AutoGenerateColumns = false;

    // This hidden column holds the item ID, so we know which item to sell
    dgvMyItems.Columns.Add(new DataGridViewTextBoxColumn
    {
        DataPropertyName = "ItemID",
        Visible = false
    });

    dgvMyItems.Columns.Add(new DataGridViewTextBoxColumn
    {
        HeaderText = "Name",
        Width = 100,
        DataPropertyName = "Description"
    });

    dgvMyItems.Columns.Add(new DataGridViewTextBoxColumn
    {
        HeaderText = "Qty",
        Width = 30,
        DefaultCellStyle = rightAlignedCellStyle,
        DataPropertyName = "Quantity"
    });

    dgvMyItems.Columns.Add(new DataGridViewTextBoxColumn
    {
        HeaderText = "Price",
        Width = 35,
        DefaultCellStyle = rightAlignedCellStyle,
        DataPropertyName = "Price"
    });

    dgvMyItems.Columns.Add(new DataGridViewButtonColumn
    {
        Text = "Sell 1",
        UseColumnTextForButtonValue = true,
        Width = 50,
        DataPropertyName = "ItemID"
    });
```

continued on next page

```
52      // Bind the player's inventory to the datagridview
53      dgvMyItems.DataSource = _currentPlayer.Inventory;
54
55      // When the user clicks on a row, call this function
56      dgvMyItems.CellClick += dgvMyItems_CellClick;
57
58
59      // Populate the datagrid for the vendor's inventory
60      dgvVendorItems.RowHeadersVisible = false;
61      dgvVendorItems.AutoGenerateColumns = false;
62
63      // This hidden column holds the item ID, so we know which item to sell
64      dgvVendorItems.Columns.Add(new DataGridViewTextBoxColumn
65      {
66          DataPropertyName = "ItemID",
67          Visible = false
68      });
69
70      dgvVendorItems.Columns.Add(new DataGridViewTextBoxColumn
71      {
72          HeaderText = "Name",
73          Width = 100,
74          DataPropertyName = "Description"
75      });
76
77      dgvVendorItems.Columns.Add(new DataGridViewTextBoxColumn
78      {
79          HeaderText = "Price",
80          Width = 35,
81          DefaultCellStyle = rightAlignedCellStyle,
82          DataPropertyName = "Price"
83      });
84
85      dgvVendorItems.Columns.Add(new DataGridViewButtonColumn
86      {
87          Text = "Buy 1",
88          UseColumnTextForButtonValue = true,
89          Width = 50,
90          DataPropertyName = "ItemID"
91      });
92
93      // Bind the vendor's inventory to the datagridview
94      dgvVendorItems.DataSource = _currentPlayer.CurrentLocation.VendorWorkingHere.Inventory;
95
96      // When the user clicks on a row, call this function
97      dgvVendorItems.CellClick += dgvVendorItems_CellClick;
98  }
99
100 private void dgvMyItems_CellClick(object sender, DataGridViewCellEventArgs e)
101 {
102 }
103
104 private void dgvVendorItems_CellClick(object sender, DataGridViewCellEventArgs e)
105 {
106 }
```

Much of the code is the same as when we bound the player's inventory to the datagridview on the SuperAdventure screen. But there are a few new things.

First, we created a DataGridViewCellStyle object. You can use these objects to define special formatting for a data grid's columns. This code creates a style to align the text to the right, instead of the default alignment to the left. We will use it for our numeric columns (quantity and price):

```
// Style, to display numeric column values
DataGridViewCellStyle rightAlignedCellStyle =
    new DataGridViewCellStyle();
rightAlignedCellStyle.Alignment =
    DataGridViewContentAlignment.MiddleRight;
```

We use this code to hide the ItemID columns, by setting their *Visible* status to false:

```
// This hidden column holds the item ID, so we know which
   item to sell
dgvMyItems.Columns.Add(new DataGridViewTextBoxColumn
{
    DataPropertyName = "ItemID",
    Visible = false
});
```

We need to include the ItemID column, so we know which item to buy or sell. However, we don't want to display it – that number won't mean anything to the player.

For the numeric columns that we want to be right-aligned, we set their DefaultCellStyle property to the rightAlignCellStyle object that we created earlier:

```
dgvMyItems.Columns.Add(new DataGridViewTextBoxColumn
{
    HeaderText = "Qty",
    Width = 30,
    DefaultCellStyle = rightAlignedCellStyle,
    DataPropertyName = "Quantity"
});
```

To connect clicking the buy/sell buttons with the functions to perform the buying and selling, we add these lines:

```
// When the user clicks on a row, call this function
dgvMyItems.CellClick += dgvMyItems_CellClick;

// When the user clicks on a row, call this function
dgvVendorItems.CellClick += dgvVendorItems_CellClick;
```

When the user clicks on the *Sell 1* button, in their inventory, the program will call the dgvMyItems_CellClick() function, and sell one of that item. When they click on the *Buy 1* button, in the vendor's inventory, the program will call the dgvVendorItems_CellClick() function, and try to buy one of those items.

We've added a lot of code, so run your program, move the player to the Town Square (the only location with a vendor), and click on the *Trade* button. You should see items in the inventory datagridviews. If you click on the *Sell 1* or *Buy 1* buttons, nothing will happen yet – that's the next step.

STEP 7 Almost there!

Now we will add the logic to buy and sell items. These functions will increase, or decrease, the player's inventory and gold. We also need to add two tests: check if the player has enough gold to buy an item, and don't let the player sell items where the price is the *unsellable item flag price* we created in Lesson 21.1.

In TradingScreen.cs, change the two button click functions, dgvMyItems_CellClick() and dgvVendorItems_CellClick(), to the code on the next two pages.

For both of these functions, the *e* parameter gives us information about what cell (row/column) was clicked. That is an automatic parameter that is created, and passed, when the *CellClick* event happens.

We will use the column to make sure the user clicked on a button column – and not one of the other columns. The row will let us determine which item they want to buy or sell.

The functions check the price of the item. If the player tries to sell an item with an unsellable price, the game displays an error message. It also displays an error message if the player tries to buy an item, but doesn't have enough money for it.

If there is not a problem, the functions increase, or decrease, the player's gold and inventory. Because the player's inventory and gold raise events when they are updated, you will automatically see these changes on the SuperAdventure form.

dgvMyItems_CellClick()

```
 1  private void dgvMyItems_CellClick(object sender, DataGridViewCellEventArgs e)
 2  {
 3      // The first column of a datagridview has a ColumnIndex = 0
 4      // This is known as a "zero-based" array/collection/list.
 5      // You start counting with 0.
 6      //
 7      // The 5th column (ColumnIndex = 4) is the column with the button.
 8      // So, if the player clicked the button column, we will sell an item from that row.
 9      if(e.ColumnIndex == 4)
10      {
11          // This gets the ID value of the item, from the hidden 1st column
12          // Remember, ColumnIndex = 0, for the first column
13          var itemID = dgvMyItems.Rows[e.RowIndex].Cells[0].Value;
14
15          // Get the Item object for the selected item row
16          Item itemBeingSold = World.ItemByID(Convert.ToInt32(itemID));
17
18          if(itemBeingSold.Price == World.UNSELLABLE_ITEM_PRICE)
19          {
20              MessageBox.Show("You cannot sell the " + itemBeingSold.Name);
21          }
22          else
23          {
24              // Remove one of these items from the player's inventory
25              _currentPlayer.RemoveItemFromInventory(itemBeingSold);
26
27              // Give the player the gold for the item being sold.
28              _currentPlayer.Gold += itemBeingSold.Price;
29          }
30      }
31  }
```

continued on next page

dgvVendorItems_CellClick()

```
 1  private void dgvVendorItems_CellClick(object sender, DataGridViewCellEventArgs e)
 2  {
 3      // The 4th column (ColumnIndex = 3) has the "Buy 1" button.
 4      if(e.ColumnIndex == 3)
 5      {
 6          // This gets the ID value of the item, from the hidden 1st column
 7          var itemID = dgvVendorItems.Rows[e.RowIndex].Cells[0].Value;
 8
 9          // Get the Item object for the selected item row
10          Item itemBeingBought = World.ItemByID(Convert.ToInt32(itemID));
11
12          // Check if the player has enough gold to buy the item
13          if(_currentPlayer.Gold >= itemBeingBought.Price)
14          {
15              // Add one of the items to the player's inventory
16              _currentPlayer.AddItemToInventory(itemBeingBought);
17
18              // Remove the gold to pay for the item
19              _currentPlayer.Gold -= itemBeingBought.Price;
20          }
21          else
22          {
23              MessageBox.Show("You do not have enough gold to buy the " +
                    itemBeingBought.Name);
24          }
25      }
26  }
```

STEP 8 Compile and run your program. Move the player to the town square location, click on the *Trade* button, and try to buy and sell items.

When the player sells items, we do not add them to the vendor's inventory. We could, but then we would probably want to add the ability to save, and read, the inventories for all vendors, when the player closes and restarts the game. You could do that, if you want to expand the game.

Summary

Once you know how to pass variables to other forms, and how some variables are passed *by reference,* you can make larger programs, with more screens.

22

Use SQL to save and restore player's game data

Installing MS SQL Server on your computer

Creating database tables from classes

Creating the SQL to save and load the saved game data

22 Installing MS SQL Server on your computer

Adding SQL to SuperAdventure

When we add the SQL Server code to SuperAdventure, we will check to see if the database exists. If it doesn't, the program will continue to use the XML file to save the player's game data. This way, people can still use the program if they don't want to install SQL Server.

Which SQL database engine?

For these SQL lessons, I will use Microsoft SQL Server 2014. If you do not have, or cannot install, Microsoft SQL Server, you can use one of the other SQL database engines available. Two other popular SQL database engines are **PostgreSQL** (postgresql.org) and **MySQL** (mysql.com).

If you use one of the other SQL database engines, there will be a few small differences. I will try to mention the areas that work different from Microsoft SQL Server. If you see a problem, get in touch with me at scottlilly.com and I will try to find an answer to get your SQL engine working.

Preparing to install Microsoft SQL Server

It's best to install SQL Server before installing Visual Studio. However, if you are at this point in the lessons, you already Visual Studio installed. So, we will install SQL Server now.

Before installing Microsoft SQL Server, you need to install version 3.5 of the .NET Framework. You even need to do it if you have a newer version installed. SQL Server has some specific things it needs from version 3.5.

Or, you can open your Control Panel, go to Programs and Features, select *Turn Windows features on or off,* and check the box for *.NET Framework 3.5 (includes .NET 2.0 and 3.0)*. This will try to install the 3.5 Framework from your Windows installation disks, or from the Internet.

Parts of SQL Server

There are usually two things you install, when you install any SQL program – the *database engine* and the *management tool*. The database engine is the program that runs in the background. It's always running, but it does not have a user interface. This is what your program uses to save and retrieve data.

The management tool has a user interface. It lets you easily manage your databases. Microsoft named theirs *Microsoft SQL Server Management Studio*. This is the program we will use to create the tables in our database. If you use a different SQL engine, they will probably have their own management program.

22.1 Creating database tables from classes

Lesson objectives

At the end of this lesson, you will know…

- ➔ How to design database tables from custom classes
- ➔ How to create a SQL database, using the SQL Server Management Studio
- ➔ How to create tables, using DDL (Data Definition Language SQL statements)
- ➔ The difference between DDL (Data Definition Language) and DML (Data Manipulation Language) SQL statements

Now, we will design and build the database, with its tables. We'll use the structure of the PlayerData.xml file to design our tables.

A typical PlayerData.XMl file looks like this:

```
<Player>
    <Stats>
        <CurrentHitPoints>10</CurrentHitPoints>
        <MaximumHitPoints>10</MaximumHitPoints>
        <Gold>170</Gold>
        <ExperiencePoints>79</ExperiencePoints>
        <CurrentLocation>2</CurrentLocation>
    </Stats>
    <InventoryItems>
        <InventoryItem ID="1" Quantity="1" />
        <InventoryItem ID="3" Quantity="21" />
        <InventoryItem ID="7" Quantity="1" />
        <InventoryItem ID="4" Quantity="4" />
        <InventoryItem ID="5" Quantity="3" />
        <InventoryItem ID="10" Quantity="1" />
        <InventoryItem ID="2" Quantity="6" />
    </InventoryItems>
    <PlayerQuests>
        <PlayerQuest ID="1" IsCompleted="true" />
        <PlayerQuest ID="2" IsCompleted="true" />
    </PlayerQuests>
</Player>
```

We can have an unlimited number of InventoryItem nodes. The same with PlayerQuest nodes. This is called a **one-to-many** relationship. One Player can have many InventoryItem/PlayerQuest objects. (Technically, this is a one-to-zero-to-many relationship, since it is possible to have zero items in the list properties.)

So, when we design our tables, we will have one table hold the Player stats (CurrentHitPoints, etc.). Then, we will create two more tables: one to hold the player's inventory list, and the second to hold their quest list.

If you are not using Microsoft SQL Server, reading this steps should be useful (and *might* work). There are some additional comments at the end of the lesson that will summarize what you need to do for your environment.

STEP 1 Start Microsoft SQL Server Management Studio.

NOTE:
If you don't know your computer name, open the Control Panel, select Administrative Tools ➡ System Information ➡ and get the value from *System Name.*

You will see a popup screen, asking you to connect to the server. The server name should be filled in with your computer's name. If it isn't, you can fill in your computer's name. You can also use a period/full stop (.), or *(local)*, to signify *the SQL engine on the current/local computer.*

Now you are connected to your SQL Server engine.

STEP 2 In the left-most section, is the Object Explorer. This is similar to the Solution Explorer in Visual Studio, except it holds all the objects (databases, users, etc.) of the SQL Server.

Click the plus sign (+) next to *Databases.* This shows you the databases that are currently running with this SQL engine. If this is a new installation of SQL Server, you will only see *System Databases* and *Database Snapshots.* When you add your own databases, you will see them here.

Normally, you create one database per program. That database will hold all the information that program needs. For larger programs, or if you are going to store a huge amount of data, you might have multiple databases. But for SuperAdventure, we only need one.

You can create the database for SuperAdventure by using the UI, or by using a **script.** We will use the UI.

A script is a set of SQL statements you run on a SQL Server. We will use them to create the tables, so you'll see how they work. However, the script to create the database needs some information that is specific to your computer. So, we won't use one for this step – because I don't know exactly how your computer is set up.

Creating a new database with the UI

Right-click on the *Databases* folder icon and select *New Database…* . A screen will pop up.

Enter *SuperAdventure* in the Database Name textbox, and click the *OK* button. There are some more advanced options you can change before clicking *OK,* but we don't need them for SuperAdventure. Usually, they will only be used with larger databases.

After you click the *OK* button, you should see the SuperAdventure database listed under the *Databases* folder.

STEP 3 Click the + , next to the SuperAdventure database, then click the + next to *Tables.*

The tables are where you store data in a SQL database. Right now, there are only the built-in system tables. We need to add our custom tables. This is similar to how we created the classes in the Engine project.

To create a new table, you could right-click on *Tables,* select *New,* then *Table.* This would give you a way to create the database in the UI. But we will create the table using SQL statements/commands.

There are two types of SQL statements: **DDL** and **DML.** DDL stands for **Data Definition Language.** These are the SQL commands you use to create, and modify, tables and their definitions (the structure of your database). DML stands for **Data Manipulation Language.** These are the commands you use to add, read, delete, and modify the data inside your database tables. Right now, we are using DDL statements, to create the structure of our database.

In the menu at the top, click on *New Query*. This will create a place for us to run our DDL SQL statements.

Inside the query section (the middle of the screen), paste in this SQL code:

```
USE [SuperAdventure]
GO

/****** Object:  Table [dbo].[SavedGame]
    Script Date: 2/2/2016 6:21:08 PM ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[SavedGame](
       [CurrentHitPoints] [int] NOT NULL,
       [MaximumHitPoints] [int] NOT NULL,
       [Gold] [int] NOT NULL,
       [ExperiencePoints] [int] NOT NULL,
       [CurrentLocationID] [int] NOT NULL
) ON [PRIMARY]

GO
```

This is what the different parts of the SQL statements mean:

```
USE [SuperAdventure]
GO
```

This lets SQL Server Management Studio know we want to work in the SuperAdventure database.

The ANSI_NULLS and QUOTED_IDENTIFIER lines deal with how SQL treats some of your commands. These are common settings that you probably won't need to change, unless you do something special in your database.

The CREATE TABLE part is the important part. This is where we define the structure of the database.

Think of a database table as a spreadsheet. If you created a spreadsheet to store your expenses, you would probably have a column for the date you spent the money, another column for a description of what you bought, and a third column for the amount of money you spent. Each row on your spreadsheet would be an expense.

A database table works similarly.

In the CREATE TABLE command, we define the columns we want to store in this table.

Our table will be named *SavedGame*. The *[dbo]* means the table belongs to the *dbo* schema. A schema is a way to group your tables together, and *dbo* is the default DataBase Owner. Having more than one schema is something you will probably only do in big applications.

This table will have the five columns that hold the player's stats (CurrentHitPoints, MaximumHitPoints, etc.). Each of these columns will hold integer values (the *[int]* part is where we define the column's datatype). The NOT NULL means that we do not want to allow null/empty values in here. If we ever tried to set one of those values to NULL, SQL Server would raise an error.

The ON [PRIMARY] part means we want to store this table's data in the primary file group. We only have one file group set up (the default when you create a database). You will probably only have more than one file group if you work with much larger databases.

Finally, click on the *! Execute,* in the menu, to run this set of SQL statements. You should see a message that says *Command(s) completed successfully.* If you right-click on the *Tables* folder (in the Object Explorer), and select *Refresh,* you should see the SavedGame table.

Click on *New Query,* paste in this SQL code, and execute this query to create the Inventory table:

```
USE [SuperAdventure]
GO

/****** Object:  Table [dbo].[Inventory]
    Script Date: 2/2/2016 6:20:57 PM ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Inventory](
       [InventoryItemID] [int] NOT NULL,
       [Quantity] [int] NOT NULL
) ON [PRIMARY]

GO
```

Then, click *New Query* again, paste in this query, and execute it to create the Quest table:

```
USE [SuperAdventure]
GO

/****** Object: Table [dbo].[Quest]
    Script Date: 2/2/2016 6:21:03 PM ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Quest](
       [QuestID] [int] NOT NULL,
       [IsCompleted] [bit] NOT NULL
) ON [PRIMARY]

GO
```

If you refresh the *Tables* folder again, you should see the three tables we will use.

If you are not using Microsoft SQL Server

At the end of this lesson, you should have an empty database with the tables you will need for the next lesson. If you have a different SQL engine, you need to create a database named *SuperAdventure,* with these tables:

SavedGame table

Column Name	Data Type	Allow Nulls
CurrentHitPoints	int	no
MaximumHitPoints	int	no
Gold	int	no
ExperiencePoints	int	no
CurrentLocationID	int	no

Inventory table

Column Name	Data Type	Allow Nulls
InventoryItemID	int	no
Quantity	int	no

Quest table

Column Name	Data Type	Allow Nulls
QuestID	int	no
IsCompleted	*bit*	no

If your database engine does not have the ability to specify if the column should allow nulls, don't worry – you don't really need that. Our code will prevent that situation from happening, so it isn't really necessary.

Also, any numeric datatype should be OK, if your database does not have an integer datatype (although, most of them should have it). In the Quest table, the IsCompleted is a boolean value, which is saved as a *bit* datatype in MS SQL Server. Your database may call their datatype something like *boolean,* or *true/false*.

Summary

For these tables, we will only ever have one row/record in the SavedGame table. When the player exits their game, we will overwrite the old SavedGame row/record.

The Inventory and Quest tables will have one row per inventory item/quest that the player has. That might be zero rows, or it could be dozens (if you modified the game to have many more quests and items).

For these tables, all our columns have an *int* datatype. There are many other datatypes SQL can store, but we don't need to use them right now. If we need to store string values, we would create *varchar* columns. Boolean properties are stored in *bit* columns. .NET Datetime are stored in SQL Datetime datatype columns.

You can find a more comprehensive list of SQL datatypes at msdn.microsoft.com, to give you an idea of what else is available.

22.2 Creating the SQL to save and load the saved game data

Lesson objectives

At the end of this lesson, you will know…

- ➔ How to connect your program to a SQL database
- ➔ How to create SQL statements to add, read, delete, and update data in SQL database tables

Now, we are going to use the SQL database you created, to store, update, and retrieve the player's game data.

This lesson is a long one. We need to add the code to save the game data to the database, update the database for each new save, and read the saved data. We also need to change some existing classes, that let you create the Player object from the SQL data.

How to connect your C# program to a SQL Server database

To connect to the database, we need a *connection string.*

A connection string tells our program where the database server is located, which database we want to use, and the database's user ID and password – if you configure your database to use them (which we did not do for this database).

If you are using MS SQL Server, you will use this connection string:

```
Server=(local);Database=SuperAdventure;Trusted_Connection=True;
```

This tells SuperAdventure that the SQL engine is on the local computer (the same one we are running the program from), the database name is SuperAdventure, and we are not using a user ID and password to connect to the database. If you had your database on a different computer, you would use that computer's network name, instead of *(local).*

Every SQL engine has a slightly different format for its connection string. Fortunately, connectionstrings.com has every connection string you will probably ever need. You can go there, select the SQL engine you are using, and it will tell you the format to use.

STEP 1 Open the solution in Visual Studio, and edit the Player.cs.

```
public static Player CreatePlayerFromDatabase(
    int currentHitPoints, int maximumHitPoints, int gold,
        int experiencePoints, int currentLocationID)
{
    Player player = new Player(currentHitPoints,
        maximumHitPoints, gold, experiencePoints);

    player.MoveTo(World.LocationByID(currentLocationID));

    return player;
}
```

This is similar to the function we use to create the player from the XML data, except it uses the parameters we will read from the columns in the SavedGame table.

STEP 2 In the Engine project, create a new class named *PlayerDataMapper*. This is where we will put our database functions. Add the code on the next six pages to the PlayerDataMapper class. I'll explain what it is doing after.

```
 1  using System;
 2  using System.Data;
 3  using System.Data.SqlClient;
 4
 5  namespace Engine
 6  {
 7      public static class PlayerDataMapper
 8      {
 9          private static readonly string _connectionString =
                "Data Source=(local);Initial Catalog=SuperAdventure;Integrated Security=True";
10
11          public static Player CreateFromDatabase()
12          {
13              try
14              {
15                  // This is our connection to the database
16                  using(SqlConnection connection = new SqlConnection(_connectionString))
17                  {
18                      // Open the connection, so we can perform SQL commands
19                      connection.Open();
20
21                      Player player;
22
23                      // Create a SQL command object, that uses the connection to
24                      // our database. The SqlCommand object is where we create our
                           SQL statement.
25                      using(SqlCommand savedGameCommand = connection.CreateCommand())
26                      {
27                          savedGameCommand.CommandType = CommandType.Text;
28                          // This SQL statement reads the first rows in the SavedGame table.
29                          // For this program, we should only ever have one row,
30                          // but this will ensure we only get one record in our
                               SQL query results.
31                          savedGameCommand.CommandText = "SELECT TOP 1 * FROM SavedGame";
32
33                          // Use ExecuteReader when you expect the query to return a row,
                               or rows
34                          SqlDataReader reader = savedGameCommand.ExecuteReader();
35
36                          // Check if the query did not return a row/record of data
37                          if(!reader.HasRows)
38                          {
39                              // There is no data in the SavedGame table,
40                              // so return null (no saved player data)
41                              return null;
42                          }
43
44                          // Get the row/record from the data reader
45                          reader.Read();
46
47                          // Get the column values for the row/record
48                          int currentHitPoints = (int)reader["CurrentHitPoints"];
49                          int maximumHitPoints = (int)reader["MaximumHitPoints"];
50                          int gold = (int)reader["Gold"];
51                          int experiencePoints = (int)reader["ExperiencePoints"];
52                          int currentLocationID = (int)reader["CurrentLocationID"];
```

continued on next page

```
53
54                      // Create the Player object, with the saved game values
55                      player = Player.CreatePlayerFromDatabase(
                            currentHitPoints, maximumHitPoints, gold,
56                              experiencePoints, currentLocationID);
57                  }
58
59                  // Read the rows/records from the Quest table, and add them
                       to the player
60                  using(SqlCommand questCommand = connection.CreateCommand())
61                  {
62                      questCommand.CommandType = CommandType.Text;
63                      questCommand.CommandText = "SELECT * FROM Quest";
64
65                      SqlDataReader reader = questCommand.ExecuteReader();
66
67                      if(reader.HasRows)
68                      {
69                          while(reader.Read())
70                          {
71                              int questID = (int)reader["QuestID"];
72                              bool isCompleted = (bool)reader["IsCompleted"];
73
74                              // Build the PlayerQuest item, for this row
75                              PlayerQuest playerQuest =
                                    new PlayerQuest(World.QuestByID(questID));
76                              playerQuest.IsCompleted = isCompleted;
77
78                              // Add the PlayerQuest to the player's property
79                              player.Quests.Add(playerQuest);
80                          }
81                      }
82                  }
83
84                  // Read the rows/records from the Inventory table, and add them
                       to the player
85                  using(SqlCommand inventoryCommand = connection.CreateCommand())
86                  {
87                      inventoryCommand.CommandType = CommandType.Text;
88                      inventoryCommand.CommandText = "SELECT * FROM Inventory";
89
90                      SqlDataReader reader = inventoryCommand.ExecuteReader();
91
92                      if(reader.HasRows)
93                      {
94                          while(reader.Read())
95                          {
96                              int inventoryItemID = (int)reader["InventoryItemID"];
97                              int quantity = (int)reader["Quantity"];
98
99                              // Add the item to the player's inventory
100                             player.AddItemToInventory(
                                    World.ItemByID(inventoryItemID), quantity);
101                         }
102                     }
103                 }
```

continued on next page

```
104
105                     // Now that the player has been built from the database, return it.
106                     return player;
107                 }
108             }
109             catch(Exception ex)
110             {
111                 // Ignore errors. If there is an error, this function will return
                    a "null" player.
112             }
113
114             return null;
115         }
116
117         public static void SaveToDatabase(Player player)
118         {
119             try
120             {
121                 using(SqlConnection connection = new SqlConnection(_connectionString))
122                 {
123                     // Open the connection, so we can perform SQL commands
124                     connection.Open();
125
126                     // Insert/Update data in SavedGame table
127                     using(SqlCommand existingRowCountCommand = connection.CreateCommand())
128                     {
129                         existingRowCountCommand.CommandType = CommandType.Text;
130                         existingRowCountCommand.CommandText =
                                "SELECT count(*) FROM SavedGame";
131
132                         // Use ExecuteScalar when your query will return one value
133                         int existingRowCount =
                                (int)existingRowCountCommand.ExecuteScalar();
134
135                         if(existingRowCount == 0)
136                         {
137                             // There is no existing row, so do an INSERT
138                             using(SqlCommand insertSavedGame = connection.CreateCommand())
139                             {
140                                 insertSavedGame.CommandType = CommandType.Text;
141                                 insertSavedGame.CommandText =
142                                     "INSERT INTO SavedGame " +
143                                     "(CurrentHitPoints, MaximumHitPoints, Gold,
                                        ExperiencePoints, CurrentLocationID) " +
144                                     "VALUES " +
145                                     "(@CurrentHitPoints, @MaximumHitPoints, @Gold,
                                        @ExperiencePoints, @CurrentLocationID)";
146
```

continued on next page

```
147                             // Pass the values from the player object, to the SQL
                                   query, using parameters
148                             insertSavedGame.Parameters.Add(
149                                 "@CurrentHitPoints", SqlDbType.Int);
                                insertSavedGame.Parameters["@CurrentHitPoints"].Value =
                                    player.CurrentHitPoints;
150                             insertSavedGame.Parameters.Add(
                                    "@MaximumHitPoints", SqlDbType.Int);
151                             insertSavedGame.Parameters["@MaximumHitPoints"].Value =
                                    player.MaximumHitPoints;
152                             insertSavedGame.Parameters.Add(
                                    "@Gold", SqlDbType.Int);
153                             insertSavedGame.Parameters["@Gold"].Value =
                                    player.Gold;
154                             insertSavedGame.Parameters.Add(
                                    "@ExperiencePoints", SqlDbType.Int);
155                             insertSavedGame.Parameters["@ExperiencePoints"].Value =
                                    player.ExperiencePoints;
156                             insertSavedGame.Parameters.Add(
                                    "@CurrentLocationID", SqlDbType.Int);
157                             insertSavedGame.Parameters["@CurrentLocationID"].Value =
                                    player.CurrentLocation.ID;
158
159                             // Perform the SQL command.
160                             // Use ExecuteNonQuery, because this query does
                                   not return any results.
161                             insertSavedGame.ExecuteNonQuery();
162                         }
163                     }
164                     else
165                     {
166                         // There is an existing row, so do an UPDATE
167                         using(SqlCommand updateSavedGame = connection.CreateCommand())
168                         {
169                             updateSavedGame.CommandType = CommandType.Text;
170                             updateSavedGame.CommandText =
171                                 "UPDATE SavedGame " +
172                                 "SET CurrentHitPoints = @CurrentHitPoints, " +
173                                 "MaximumHitPoints = @MaximumHitPoints, " +
174                                 "Gold = @Gold, " +
175                                 "ExperiencePoints = @ExperiencePoints, " +
176                                 "CurrentLocationID = @CurrentLocationID";
177
```

continued on next page

```
178                             // Pass the values from the player object, to the SQL
                                   query, using parameters
179                             // Using parameters helps make your program more secure.
180                             // It will prevent SQL injection attacks.
181                             updateSavedGame.Parameters.Add(
                                    "@CurrentHitPoints", SqlDbType.Int);
182                             updateSavedGame.Parameters["@CurrentHitPoints"].Value =
                                    player.CurrentHitPoints;
183                             updateSavedGame.Parameters.Add(
                                    "@MaximumHitPoints", SqlDbType.Int);
184                             updateSavedGame.Parameters["@MaximumHitPoints"].Value =
                                    player.MaximumHitPoints;
185                             updateSavedGame.Parameters.Add(
                                    "@Gold", SqlDbType.Int);
186                             updateSavedGame.Parameters["@Gold"].Value =
                                    player.Gold;
187                             updateSavedGame.Parameters.Add(
                                    "@ExperiencePoints", SqlDbType.Int);
188                             updateSavedGame.Parameters["@ExperiencePoints"].Value =
                                    player.ExperiencePoints;
189                             updateSavedGame.Parameters.Add(
                                    "@CurrentLocationID", SqlDbType.Int);
190                             updateSavedGame.Parameters["@CurrentLocationID"].Value =
                                    player.CurrentLocation.ID;
191
192                             // Perform the SQL command.
193                             // Use ExecuteNonQuery, because this query does not
                                   return any results.
194                             updateSavedGame.ExecuteNonQuery();
195                         }
196                     }
197                 }
198
199                 // The Quest and Inventory tables might have more, or less, rows in
200                 // the database than what the player has in their properties.
201                 // So, when we save the player's game, we will delete all the old rows
202                 // and add in all new rows.
203                 // This is easier than trying to add/delete/update each individual rows
204
205                 // Delete existing Quest rows
206                 using(SqlCommand deleteQuestsCommand = connection.CreateCommand())
207                 {
208                     deleteQuestsCommand.CommandType = CommandType.Text;
209                     deleteQuestsCommand.CommandText = "DELETE FROM Quest";
210
211                     deleteQuestsCommand.ExecuteNonQuery();
212                 }
213
214                 // Insert Quest rows, from the player object
215                 foreach(PlayerQuest playerQuest in player.Quests)
216                 {
217                     using(SqlCommand insertQuestCommand = connection.CreateCommand())
218                     {
```

continued on next page

```
219                             insertQuestCommand.CommandType = CommandType.Text;
220                             insertQuestCommand.CommandText =
                                    "INSERT INTO Quest (QuestID, IsCompleted)
                                        VALUES (@QuestID, @IsCompleted)";
221
222                             insertQuestCommand.Parameters.Add(
                                    "@QuestID", SqlDbType.Int);
223                             insertQuestCommand.Parameters["@QuestID"].Value =
                                    playerQuest.Details.ID;
224                             insertQuestCommand.Parameters.Add(
                                    "@IsCompleted", SqlDbType.Bit);
225                             insertQuestCommand.Parameters["@IsCompleted"].Value =
                                    playerQuest.IsCompleted;
226
227                             insertQuestCommand.ExecuteNonQuery();
228                         }
229                     }
230
231                     // Delete existing Inventory rows
232                     using(SqlCommand deleteInventoryCommand = connection.CreateCommand())
233                     {
234                         deleteInventoryCommand.CommandType = CommandType.Text;
235                         deleteInventoryCommand.CommandText = "DELETE FROM Inventory";
236
237                         deleteInventoryCommand.ExecuteNonQuery();
238                     }
239
240                     // Insert Inventory rows, from the player object
241                     foreach(InventoryItem inventoryItem in player.Inventory)
242                     {
243                         using(SqlCommand insertInventoryCommand =
                                    connection.CreateCommand())
244                         {
245                             insertInventoryCommand.CommandType = CommandType.Text;
246                             insertInventoryCommand.CommandText =
                                    "INSERT INTO Inventory (InventoryItemID, Quantity)
                                        VALUES (@InventoryItemID, @Quantity)";
247                             insertInventoryCommand.Parameters.Add(
                                    "@InventoryItemID", SqlDbType.Int);
248                             insertInventoryCommand.Parameters[
                                    "@InventoryItemID"].Value = inventoryItem.Details.ID;
249                             insertInventoryCommand.Parameters.Add(
                                    "@Quantity", SqlDbType.Int);
250                             insertInventoryCommand.Parameters["@Quantity"].Value =
                                    inventoryItem.Quantity;
251                             insertInventoryCommand.ExecuteNonQuery();
252                         }
253                     }
254                 }
255             }
256             catch(Exception ex)
257             {
258                 // We are going to ignore erros, for now.
259             }
260         }
261     }
262 }
```

Explanation of the PlayerDataMapper class

In order to use the .NET SQL objects, we need these *using* statements:

```
using System.Data;
using System.Data.SqlClient;
```

These namespaces are where the *SqlConnection*, *SqlCommand*, and *SqlDataReader* classes exist. Those are the classes we will use to access the SQL server.

On **line 9,** we have the connection string.

```
private static readonly string _connectionString =
    "Data Source=(local);Initial Catalog=SuperAdventure;
        Integrated Security=True";
```

If you aren't using SQL Server, you will need to change this to the correct one for your database. This is a static variable because we are using it in the static methods of this static class. So, any class-level variables we use in them must also be static.

We made the class a *public static class* because we are not going to create an instance of it. We are only going to call its static functions to manage our data access – similar to the World class.

CreateFromDatabase() function description

This function will be called when the player starts the game. We will see if there is saved game data in the database. If there is, we will create a Player object, and return it to SuperAdventure.cs. If nothing is in the database, we will return null, and SuperAdventure.cs will create the player from the XML file, or create a new player.

There is a *try/catch* in here **(lines 13 and 109)** because you might have problems with this code (the database might not be running, the connection string might not be correct, etc.) The function will *try* to execute the code in the *try* section. If there is an error, it will run the *catch* section.

If you have a problem running this, you can set a breakpoint in the *catch* and examine the error (the *ex* variable). That may help you (and me) discover the source of the problem.

Starting on **line 16,** we have something new, a *using* inside a function:

```
using(SqlConnection connection =
    new SqlConnection(_connectionString))
{
    ...
}
```

This *using* is different from the *using* lines we add at the top of a class. Notice that it has opening and closing curly braces after it, similar to an IF statement.

Inside the parentheses we are instantiating a SqlConnection object, using our connection string. This is the object that connects to the SQL database. All our commands will use it.

A SqlConnection object is a special type of object. It is a disposable object.

Usually, when you create an object, .NET knows when you don't need it any more. If you instantiate a variable inside a function, .NET knows it is not needed after the function is done. It can re-use that space in memory. However, objects such as a SqlConnection object connect to resources outside your program. .Net is not sure when it can get rid of that object and re-use the memory (or other resources the object used). So, you need to tell .NET when you are done with disposable objects.

We could do it like this:

```
SqlConnection connection =
    new SqlConnection(_connectionString);

...

connection.Dispose();
```

In this example, we instantiate the connection object without a *using,* and we call the Dispose() method on it to tell .NET we are done with it.

However, the *using,* with the curly braces, will automatically do that for us. It sets the *scope* of the connection variable (where it exists) to the inside of the curly braces. When it reaches the closing curly brace, it automatically calls the Dispose() on the connection object.

On **line 19,** we open the connection to the database, so we can start communicating with it.

Line 21 creates the variable we will use to hold our Player object, if we find data in the database. If the database is empty, this function will return this object, which will be *null* (no player found).

Line 25 creates our SqlCommand object, from the SqlConnection object – so it will communicate with the SQL Server the SqlConnection object is talking with.

The SqlCommand object is what we will use to create our SQL statement that queries the database. Notice that it also is disposable, and we create it with a *using,* like the SqlConnection object.

Lines 27-31, we give more details of what the SqlCommand will be. It will be a Text query (we will write our own SQL statement), and the statement is:

```
SELECT TOP 1 * FROM SavedGame
```

A SELECT statement means *"get some rows/records from the database"*. In this case, we want to get the first row TOP 1. The asterisk (*) means that we want all the columns. You can write SQL queries that only return some of the columns. For that, you would include a list of the columns you want, like:

```
SELECT CurrentHitPoints, MaximumHitPoints FROM SavedGame
```

The FROM SavedGame says what table has the data we are looking for. You can write more complex SQL queries that access more than one table. But, describing everything you can do in SQL would take dozens more lessons. So, we'll just cover the basics here.

On **line 34,** we instantiate out SqlDataReader object for our SqlCommand. This will hold the results we get back from the database, after we run the query.

We run the query when we call savedGameCommand.ExecuteReader(). This runs our query and starts streaming the result back to our SqlDataReader. The SqlDataReader does not hold the results right now. Instead, we will tell it to read the rows (although, in this example, we will only ever have one row at the most).

Lines 37-42, we check if the reader sees any rows. If there are no rows (there is no saved game data), we return null from this function. If the reader has rows, we read the first one on **line 45.**

In **lines 48-52,** we get the values from the columns that the reader has, convert them to integers, and store them in variables.

Finally, on **line 55,** we create the Player object from the values we read from the database.

The next two sections **(lines 59-82** and **84-103)** are where we read the player's quests and inventories from their tables.

These parts of the function have something new in them, because they might have more than one row in their query results. The *while(reader.Read())* is a loop. If the reader can read a row, this value is *true,* so the code inside will run. This code will create a new PlayerQuest, or Inventory object, and add it to the player variable. When the reader cannot read any more rows, *reader.Read()* will return *false,* and the loop will stop running.

On **line 106,** after the player object has been created, and given all its inventory and quests, it is returned from the function.

SaveToDatabase() function description

This function will be called when the player closes the game. It will save their data to the database, like it currently does to the XML file.

Much of this is similar to the CreateFromDatabase function. It uses a SqlConnection object, and SqlCommand objects that perform queries. So, I'll only describe what it does differently here.

On **line 130,** the query is:

```
SELECT count(*) FROM SavedGame
```

The *count(*)* will count the number of rows/records in the SavedGame table and return that value. So, when we run this query, we won't get a row of data. Instead, we will get a single number. SQL has many different built-in functions similar to *count*. It can do averages, sums, minimum value, maximum value, etc.

On **line 133**, we execute the SqlCommand. Notice that this uses *ExecuteScalar(),* instead of *ExecuteReader()*. This is because our SQL statement only returns one value (that's what *scalar* means, with SQL). We are not going to be reading rows of data, so we do not need a reader.

On **line 135,** we check the count value. If it was *0,* that means there is no existing row/record in the SavedGame table. So, we will need to do an INSERT, to add a new row. If there is an existing row, the function goes to the ELSE statement on **line 164.** That is where we will do an UPDATE to the values for the existing record.

Our SQL command is a little different on **lines 142-145.** For an INSERT command, we tell it which table we want to insert into (SavedGame), the columns we want to add values to (the list on line 143), and the values we want to add (lines 144-145).

Notice that the values on line 145 have an *at* sign (@) in front of them. This is how you can signify that value is a parameter – a value we are going to pass into the SQL statement.

When you build your SQL statement, use parameters to add any variable values – especially if they are values you receive from user input. If you build your CommandText by concatenating a string, it's possible for someone to pass in special values that cause your SQL statement to do something dangerous. This is known as a **SQL injection attack.** You can find more information on the subject, and how to prevent a SQL injection attack, in my post *Common C# Vulnerabilities – SQL Injection* at scottlilly.com.

In **lines 148-157,** we add the parameters to the SqlCommand, using their name, datatype, and value. Then, on **line 161,** we run the SQL statement. Notice that this uses an *ExecuteNonQuery()*. This is because we do not expect SQL to return any results to us. We only want to insert some data into the database.

In **lines 170-176,** we update the existing record in the SavedGame table (if it already had a record, when we checked the table on line 133).

The format of the SQL statement is a little different. We tell it the table to update (SavedGame), then tell it to set each column value to our parameter value. We populate the parameters the same way as we did for the INSERT statement, by adding parameters on **lines 181-190.**

For the Quests and Inventory items, we run several SQL queries. At first, we run a DELETE query, to delete everything from the table. Then, we do an insert for each quest or inventory item.

We do this because there are three possible situations for each quest and inventory item.

1. It can exist in the Player object, but not have a record in the table (it's new, since the last time the game was saved).
2. It can have a record in the table, but not in the Player object (it existed the last time the game was saved, but doesn't any more – for example, the player sold all of the item from their inventory).
3. It exists in the Player object, and exists in the table, but might have different values (the player may have completed the quest since the last saved game).

To cover all those situations, we would need to check each item, in the database and in the Player object. Then, we would do an INSERT, DELETE, or UPDATE, depending on the situation. It's simpler to delete everything, then re-populate the table with all the items/quests that the player currently has.

So, in lines **206-212,** we create and run a query to delete all the rows from the Quest table. We do the same thing for the Inventory table in **lines 233-238.** After we delete the old rows, we loop through the Player's PlayerQuest and Inventory properties, and insert each object into the table.

STEP 3 Modify SuperAdventure.cs, to use the new functions that save and load the game with the SQL database.

Change the part of the constructor code that loads the player data from the XML file to this:

```
_player = PlayerDataMapper.CreateFromDatabase();

if(_player == null)
{
   if(File.Exists(PLAYER_DATA_FILE_NAME))
   {
       _player = Player.CreatePlayerFromXmlString(
           File.ReadAllText(PLAYER_DATA_FILE_NAME));
   }
   else
   {
       _player = Player.CreateDefaultPlayer();
   }
}
```

This will try to load the player from the PlayerDataMapper class. If there is an error, or if the SavedGame table is empty, PlayerDataMapper.CreateFromDatabase() will return a null. So, we know we need to check the XML file for the saved game. If we don't see the XML file, we will create a new player.

This adds a little extra safety. If the database doesn't work, we will still have the XML file. If you want, after you know the database is working, you could remove the code to use the XML file for the saved game data.

You also need to change the function that saves the game when the player exits the program. Modify SuperAdventure_FormClosing to this:

```
private void SuperAdventure_FormClosing(
    object sender, FormClosingEventArgs e)
{
    File.WriteAllText(
        PLAYER_DATA_FILE_NAME, _player.ToXmlString());

   PlayerDataMapper.SaveToDatabase(_player);
}
```

Now, when the player exits the game, it will save the data to the XML file and the database.

Check that your program works

Now, run SuperAdventure, and see if it still works.

Because this is a big change, there might be a problem. The most likely thing is that the connection string might not be correct for your database. Put a breakpoint at line 16 of PlayerDataMapper, and see if you can find where the program has an error.

If you do have a problem, leave a comment at scottlilly.com. Let me know what SQL engine you are using, if it is not Microsoft SQL Server.

If everything works, you should be able to use your SQL management program to look at the data in your tables. You might even be able to modify the values, like you would with a spreadsheet. You can even change a value in the database and give yourself a million rat tails, if that makes you happy.

Summary

This is the basics of working with SQL. When you work with larger programs, your databases will be more complex, and your SQL statements will be more complex. For example, if you had a website that sold shirts, you could create a query to tell you your most popular shirt, by number of sales, for each month of the year, for each country you sell to.

You will also want to look at things like SQL transactions. In a SQL transaction, you group SQL statements together. If one of them fails, all the changes from the other statements are reverted/removed.

If our game was a more professional game, we would probably include all the queries in SaveToDatabase() inside one transaction. We would at least create transactions for the places where we delete the records, the insert new ones from the Player object. That way, if the inserts failed, the delete would be undone.

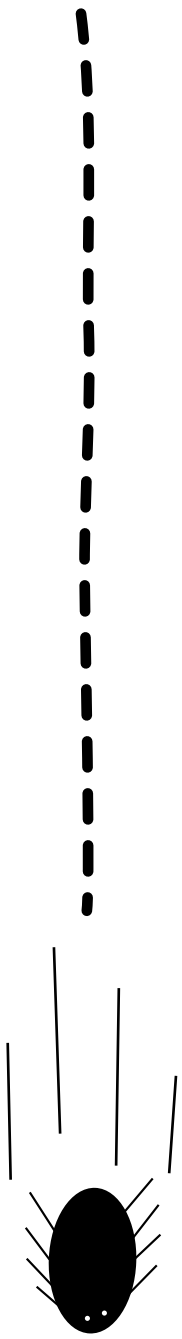
This is also what you would use if you write a program for a bank, to transfer money between accounts. To do that, you would probably write two queries – one to remove the money from the first account, another query to add it to the receiving account. If you didn't use a transaction, and there was an error between the two queries, you would have very unhappy customers – with money missing from their accounts.

Basics of performing a SQL statement in .NET

1. Get your connection string
2. Create a SqlConnection object, using your connection string
3. Create a SqlCommand object, using your SqlConnection
4. Add your SQL statement to your SqlCOmmand object
5. Add your parameter values to your SqlCommand object, if needed
6. Execute your SqlCommand, using the appropriate type of Execute function
7. Read your results, if any

The three ways to execute a SqlCommand

1. *ExecuteReader(),* when you expect to receive a row, or many rows, of data.
2. *ExecuteScalar(),* when you expect to receive a single value.
3. *ExecuteNonQuery(),* when you do not expect to receive any data from your SQL statement (for example, insert, update, or delete data).



23

Creating a console UI for SuperAdventure

23 Creating a console front-end for the game

Lesson objectives

At the end of this lesson, you will know…

- ➔ How to create a console (text) application, using the existing Engine project
- ➔ How to do simple parsing of user input

In this lesson, we will create a new front-end for the SuperAdventure program. This one will be a **console application** – a program that only uses text in the user interface. This will let us play SuperAdventure the same way as the classic adventure game, *Hunt the Wumpus.*

The great thing about creating this second interface is that we can re-use a lot of the code we already have, especially since most of the game logic is in the Engine project (and not the UI code).

STEP 1 Add the console project.

Open the SuperAdventure solution in Visual Studio. In the Solution Explorer, right-click on the SuperAdventure *solution,* and select Add ➡ New Project… . Choose *Console Application,* in the Visual C# section, and name it *SuperAdventureConsole*. This will create the new UI project.

Right-click on the SuperAdventureConsole project, and select *Set as Startup Project*. After you do this, when you run your solution, it will run the console UI. If you want to run the Windows Form UI again, right-click on the SuperAdventure project, and select *Set as Startup Project*. If you don't want to switch the startup project for the solution, you could also right-click on the project you want to run (SuperAdventure or SuperAdventureConsole), and select *Debug* ➡ *Start new instance*. That will run the project you clicked on.

Right-click on *References,* in the SuperAdventureConsole project, and add the Engine project. That will let us use the game objects in this version of the game.

STEP 2 Edit Program.cs, in the SuperAdventureConsole project.

This class has a static function named *Main()*. This is the function, and class, that is executed when you run the console program. We are going to put an **infinite loop** in this function.

Normally, an infinite loop is a bad thing in a program. It means your program is stuck doing something, and it will never end. However, we are going to add a way for the player to exit the loop – by typing the command *Exit*. If the user types anything else, the program will continue running, and the player can continue playing.

NOTE:
A quick way to write *Console.WriteLine()* is to write the letters *cw* and press the TAB key twice.

Inside our loop, we will wait for the user to type something and press the Enter key. When they press Enter, we will try to determine what the game should do, based on the user's input.

STEP 3 Add the code on the next nine pages to Program.cs.

```
using System;
using System.ComponentModel;
using System.IO;
using System.Linq;
using Engine;

namespace SuperAdventureConsole
{
    public class Program
    {
        private const string PLAYER_DATA_FILE_NAME = "PlayerData.xml";

        private static Player _player;

        private static void Main(string[] args)
        {
            // Load the player
            LoadGameData();

            Console.WriteLine("Type 'Help' to see a list of commands");
            Console.WriteLine("");

            DisplayCurrentLocation();

            // Connect player events to functions that will display in the UI
            _player.PropertyChanged += Player_OnPropertyChanged;
            _player.OnMessage += Player_OnMessage;

            // Infinite loop, until the user types "exit"
            while(true)
            {
                // Display a prompt, so the user knows to type something
                Console.Write(">");

                // Wait for the user to type something, and press the <Enter> key
                string userInput = Console.ReadLine();

                // If they typed a blank line, loop back and wait for input again
                if(string.IsNullOrWhiteSpace(userInput))
                {
                    continue;
                }

                // Convert to lower-case, to make comparisons easier
                string cleanedInput = userInput.ToLower();

                // Save the current game data, and break out of the "while(true)" loop
                if(cleanedInput == "exit")
                {
                    SaveGameData();

                    break;
                }

                // If the user typed something, try to determine what to do
                ParseInput(cleanedInput);
            }
        }
```

continued on next page

```
59
60          private static void Player_OnPropertyChanged(
                object sender, PropertyChangedEventArgs e)
61          {
62              if(e.PropertyName == "CurrentLocation")
63              {
64                  DisplayCurrentLocation();
65
66                  if(_player.CurrentLocation.VendorWorkingHere != null)
67                  {
68                      Console.WriteLine("You see a vendor here: {0}",
                            _player.CurrentLocation.VendorWorkingHere.Name);
69                  }
70              }
71          }
72
73          private static void Player_OnMessage(object sender, MessageEventArgs e)
74          {
75              Console.WriteLine(e.Message);
76
77              if(e.AddExtraNewLine)
78              {
79                  Console.WriteLine("");
80              }
81          }
82
83          private static void ParseInput(string input)
84          {
85              if(input.Contains("help") || input == "?")
86              {
87                  Console.WriteLine("Available commands");
88                  Console.WriteLine("====================================");
89                  Console.WriteLine("Stats - Display player information");
90                  Console.WriteLine("Look - Get the description of your location");
91                  Console.WriteLine("Inventory - Display your inventory");
92                  Console.WriteLine("Quests - Display your quests");
93                  Console.WriteLine("Attack - Fight the monster");
94                  Console.WriteLine("Equip <weapon name> - Set your current weapon");
95                  Console.WriteLine("Drink <potion name> - Drink a potion");
96                  Console.WriteLine("Trade - display your inventory and vendor's inventory");
97                  Console.WriteLine("Buy <item name> - Buy an item from a vendor");
98                  Console.WriteLine("Sell <item name> - Sell an item to a vendor");
99                  Console.WriteLine("North - Move North");
100                 Console.WriteLine("South - Move South");
101                 Console.WriteLine("East - Move East");
102                 Console.WriteLine("West - Move West");
103                 Console.WriteLine("Exit - Save the game and exit");
104             }
105             else if(input == "stats")
106             {
107                 Console.WriteLine("Current hit points: {0}", _player.CurrentHitPoints);
108                 Console.WriteLine("Maximum hit points: {0}", _player.MaximumHitPoints);
109                 Console.WriteLine("Experience Points: {0}", _player.ExperiencePoints);
110                 Console.WriteLine("Level: {0}", _player.Level);
111                 Console.WriteLine("Gold: {0}", _player.Gold);
112             }
```

continued on next page

```
113             else if(input == "look")
114             {
115                 DisplayCurrentLocation();
116             }
117             else if(input.Contains("north"))
118             {
119                 if(_player.CurrentLocation.LocationToNorth == null)
120                 {
121                     Console.WriteLine("You cannot move North");
122                 }
123                 else
124                 {
125                     _player.MoveNorth();
126                 }
127             }
128             else if(input.Contains("east"))
129             {
130                 if(_player.CurrentLocation.LocationToEast == null)
131                 {
132                     Console.WriteLine("You cannot move East");
133                 }
134                 else
135                 {
136                     _player.MoveEast();
137                 }
138             }
139             else if(input.Contains("south"))
140             {
141                 if(_player.CurrentLocation.LocationToSouth == null)
142                 {
143                     Console.WriteLine("You cannot move South");
144                 }
145                 else
146                 {
147                     _player.MoveSouth();
148                 }
149             }
150             else if(input.Contains("west"))
151             {
152                 if(_player.CurrentLocation.LocationToWest == null)
153                 {
154                     Console.WriteLine("You cannot move West");
155                 }
156                 else
157                 {
158                     _player.MoveWest();
159                 }
160             }
161             else if(input == "inventory")
162             {
163                 foreach(InventoryItem inventoryItem in _player.Inventory)
164                 {
165                     Console.WriteLine("{0}: {1}",
                            inventoryItem.Description, inventoryItem.Quantity);
166                 }
167             }
```

continued on next page

```
168             else if(input == "quests")
169             {
170                 if(_player.Quests.Count == 0)
171                 {
172                     Console.WriteLine("You do not have any quests");
173                 }
174                 else
175                 {
176                     foreach(PlayerQuest playerQuest in _player.Quests)
177                     {
178                         Console.WriteLine("{0}: {1}", playerQuest.Name,
179                             playerQuest.IsCompleted ? "Completed" : "Incomplete");
180                     }
181                 }
182             }
183             else if(input.Contains("attack"))
184             {
185                 if(_player.CurrentLocation.MonsterLivingHere == null)
186                 {
187                     Console.WriteLine("There is nothing here to attack");
188                 }
189                 else
190                 {
191                     if(_player.CurrentWeapon == null)
192                     {
193                         // Select the first weapon in the player's inventory
194                         // (or 'null', if they do not have any weapons)
195                         _player.CurrentWeapon = _player.Weapons.FirstOrDefault();
196                     }
197 
198                     if(_player.CurrentWeapon == null)
199                     {
200                         Console.WriteLine("You do not have any weapons");
201                     }
202                     else
203                     {
204                         _player.UseWeapon(_player.CurrentWeapon);
205                     }
206                 }
207             }
208             else if(input.StartsWith("equip "))
209             {
210                 string inputWeaponName = input.Substring(6).Trim();
211 
212                 if(string.IsNullOrEmpty(inputWeaponName))
213                 {
214                     Console.WriteLine("You must enter the name of the weapon to equip");
215                 }
```

continued on next page

```
216                 else
217                 {
218                     Weapon weaponToEquip =
219                         _player.Weapons.SingleOrDefault(
220                             x => x.Name.ToLower() ==
                                 inputWeaponName || x.NamePlural.ToLower() ==
                                     inputWeaponName);
221
222                     if(weaponToEquip == null)
223                     {
224                         Console.WriteLine("You do not have the weapon: {0}",
                             inputWeaponName);
225                     }
226                     else
227                     {
228                         _player.CurrentWeapon = weaponToEquip;
229
230                         Console.WriteLine("You equip your {0}",
                             _player.CurrentWeapon.Name);
231                     }
232                 }
233             }
234             else if(input.StartsWith("drink "))
235             {
236                 string inputPotionName = input.Substring(6).Trim();
237
238                 if(string.IsNullOrEmpty(inputPotionName))
239                 {
240                     Console.WriteLine("You must enter the name of the potion to drink");
241                 }
242                 else
243                 {
244                     HealingPotion potionToDrink =
245                         _player.Potions.SingleOrDefault(
246                             x => x.Name.ToLower() ==
                                 inputPotionName || x.NamePlural.ToLower() ==
                                     inputPotionName);
247
248                     if(potionToDrink == null)
249                     {
250                         Console.WriteLine("You do not have the potion: {0}",
                             inputPotionName);
251                     }
252                     else
253                     {
254                         _player.UsePotion(potionToDrink);
255                     }
256                 }
257             }
258             else if(input == "trade")
259             {
260                 if(_player.CurrentLocation.VendorWorkingHere == null)
261                 {
262                     Console.WriteLine("There is no vendor here");
263                 }
```

continued on next page

```
264                 else
265                 {
266                     Console.WriteLine("PLAYER INVENTORY");
267                     Console.WriteLine("================");
268 
269                     if(_player.Inventory.Count(
                            x => x.Price != World.UNSELLABLE_ITEM_PRICE) == 0)
270                     {
271                         Console.WriteLine("You do not have any inventory");
272                     }
273                     else
274                     {
275                         foreach(
276                             InventoryItem inventoryItem in _player.Inventory.Where(
                                    x => x.Price != World.UNSELLABLE_ITEM_PRICE))
277                         {
278                             Console.WriteLine("{0} {1} Price: {2}",
                                    inventoryItem.Quantity, inventoryItem.Description,
279                                         inventoryItem.Price);
280                         }
281                     }
282 
283                     Console.WriteLine("");
284                     Console.WriteLine("VENDOR INVENTORY");
285                     Console.WriteLine("================");
286 
287                     if(_player.CurrentLocation.VendorWorkingHere.Inventory.Count == 0)
288                     {
289                         Console.WriteLine("The vendor does not have any inventory");
290                     }
291                     else
292                     {
293                         foreach(InventoryItem inventoryItem in
                                _player.CurrentLocation.VendorWorkingHere.Inventory)
294                         {
295                             Console.WriteLine("{0} {1} Price: {2}",
                                    inventoryItem.Quantity, inventoryItem.Description,
296                                         inventoryItem.Price);
297                         }
298                     }
299                 }
300             }
301             else if(input.StartsWith("buy "))
302             {
303                 if(_player.CurrentLocation.VendorWorkingHere == null)
304                 {
305                     Console.WriteLine("There is no vendor at this location");
306                 }
307                 else
308                 {
309                     string itemName = input.Substring(4).Trim();
310 
311                     if(string.IsNullOrEmpty(itemName))
312                     {
313                         Console.WriteLine("You must enter the name of the item to buy");
314                     }
```

continued on next page

```
315                     else
316                     {
317                         // Get the InventoryItem from the trader's inventory
318                         InventoryItem itemToBuy =
319                             _player.CurrentLocation.VendorWorkingHere.
                                    Inventory.SingleOrDefault(
320                                     x => x.Details.Name.ToLower() == itemName);
321
322                         // Check if the vendor has the item
323                         if(itemToBuy == null)
324                         {
325                             Console.WriteLine("The vendor does not have any {0}",
                                    itemName);
326                         }
327                         else
328                         {
329                             // Check if the player has enough gold to buy the item
330                             if(_player.Gold < itemToBuy.Price)
331                             {
332                                 Console.WriteLine(
                                        "You do not have enough gold to buy a {0}",
                                            itemToBuy.Description);
333                             }
334                             else
335                             {
336                                 // Success! Buy the item
337                                 _player.AddItemToInventory(itemToBuy.Details);
338                                 _player.Gold -= itemToBuy.Price;
339
340                                 Console.WriteLine("You bought one {0} for {1} gold",
                                        itemToBuy.Details.Name, itemToBuy.Price);
341                             }
342                         }
343                     }
344                 }
345             }
346             else if(input.StartsWith("sell "))
347             {
348                 if(_player.CurrentLocation.VendorWorkingHere == null)
349                 {
350                     Console.WriteLine("There is no vendor at this location");
351                 }
352                 else
353                 {
354                     string itemName = input.Substring(5).Trim();
355
356                     if(string.IsNullOrEmpty(itemName))
357                     {
358                         Console.WriteLine("You must enter the name of the item to sell");
359                     }
```

continued on next page

```
360                     else
361                     {
362                         // Get the InventoryItem from the player's inventory
363                         InventoryItem itemToSell =
364                             _player.Inventory.SingleOrDefault(
                                    x => x.Details.Name.ToLower() == itemName &&
365                                 x.Quantity > 0 &&
366                                 x.Price != World.UNSELLABLE_ITEM_PRICE);
367
368                         // Check if the player has the item entered
369                         if(itemToSell == null)
370                         {
371                             Console.WriteLine("The player cannot sell any {0}", itemName);
372                         }
373                         else
374                         {
375                             // Sell the item
376                             _player.RemoveItemFromInventory(itemToSell.Details);
377                             _player.Gold += itemToSell.Price;
378
379                             Console.WriteLine("You receive {0} gold for your {1}",
                                    itemToSell.Price, itemToSell.Details.Name);
380                         }
381                     }
382                 }
383             }
384             else
385             {
386                 Console.WriteLine("I do not understand");
387                 Console.WriteLine("Type 'Help' to see a list of available commands");
388             }
389
390             // Write a blank line, to keep the UI a little cleaner
391             Console.WriteLine("");
392         }
393
394         private static void DisplayCurrentLocation()
395         {
396             Console.WriteLine("You are at: {0}", _player.CurrentLocation.Name);
397
398             if(_player.CurrentLocation.Description != "")
399             {
400                 Console.WriteLine(_player.CurrentLocation.Description);
401             }
402         }
403
```

continued on next page

```
404         private static void LoadGameData()
405         {
406             _player = PlayerDataMapper.CreateFromDatabase();
407 
408             if(_player == null)
409             {
410                 if(File.Exists(PLAYER_DATA_FILE_NAME))
411                 {
412                     _player = Player.CreatePlayerFromXmlString(
                            File.ReadAllText(PLAYER_DATA_FILE_NAME));
413                 }
414                 else
415                 {
416                     _player = Player.CreateDefaultPlayer();
417                 }
418             }
419         }
420 
421         private static void SaveGameData()
422         {
423             File.WriteAllText(PLAYER_DATA_FILE_NAME, _player.ToXmlString());
424 
425             PlayerDataMapper.SaveToDatabase(_player);
426         }
427     }
428 }
```

This is everything we need to run SuperAdventure as a console application. You will probably notice that some of the functionality is from the Windows Form project.

Line 11-13: We set the variable for the saved game file, and the player. Because the Main function is a static function, we need to make the _player variable static. Also, the functions that Main calls will need to be static too. These are basically the same as lines 18-20, in SuperAdventure.cs, in the Windows Form project.

Line 18: We call the LoadGameData() function. That function looks for a previously-saved game, or creates a new player. This code is the same as lines 26-38, in SuperAdventure.cs.

Lines 20-21: *Console.WriteLine()* is a function you use to display text in the console UI. This displays the text you pass in the parameter, and does a line-feed (moves to the next line down). If you don't want to move to the next line, you would use:

```
Console.Write("your text here");
```

Line 23: Calls a function that display's the player's current location. We put this in a function, so we can display the same text when the player moves to a new location.

Lines 25-27: We connect the events from the Player class to the functions that will handle them for the UI (similar to lines 96-97, of SuperAdventure.cs).

Line 30: This is our game loop. It will continuously run, waiting for the user's input, until the user types *Exit*. A **WHILE loop** evaluates the equation inside the parentheses, and runs until that equation is false.

However, we don't have an equation in there. We only have *true,* which will always be true, which means the WHILE will never end – until the player types the word *Exit,* and the code in the IF statement on line 48 runs. The *break;* will exit the loop it is in, and finish running the rest of the code in the function. But, there is no code in this function, so the function will end – and the program will stop running.

Line 33: Display the right angle bracket (>), as a prompt for the user to enter their command. Because we use *Console.Write(),* the cursor will stay on the same line as the prompt.

Line 36: *Console.ReadLine()* is the function to read the user's input, after they press the Enter key. There is a *Console.Read()* that you could use to read every key pressed, one character at a time. But we are going to wait for the user to press Enter. Whatever the user types will be assigned to the *userInput* variable.

Lines 39-42: If the user only hit the Enter key, and didn't type anything else, the *userInput* will be null. The *continue;* commands tells the program to go back up to the *while* line and keep running. It does not execute the lines after it (45-56).

Line 45: We convert the user's command to all lower-case letters. So, if the user types *EXIT, Exit,* or *exit,* the value in *cleanedInput* will be *exit*. This will make it easier for us when we try to parse what the user typed, to determine the action to perform.

Lines 48-53: If the user typed *exit,* call the SaveGameData() function and break out of the loop – ending the game.

Line 56: If we got here, the user typed a non-blank line, that was not the word *exit*. Now we will call the ParseInput() function, and try to do what the user wants. This will be a **bug function**, since we need to manage every action that we had buttons for in the Windows Form UI.

Lines 60-71: This is the function we call when the player's PropertyChanged event is fired. It is similar to PlayerOnPropertyChanged(), in SuperAdventure.cs. However, since we don't have buttons to hide and show, or datagrids to update, it is much smaller.

The only thing we care about is if the player is at a new location. Then, we display the location's information, and the vendor information (if there is a vendor at this location).

Lines 73-81: Display messages from the player's events, like DisplayMessage() in SueprAdventure.cs.

Lines 83-392: This is where we look at the user's input, and try to do what the action they want to perform. It's a long series of IFs and ELSE IFs. If we cannot determine what the user wants to do, we eventually reach the ELSE at line 384 and give the user a message that we don't understand their input, and they can type *Help* to see a list of the valid commands.

You should be familiar with much of the logic in this function, but I will explain the new things.

Line 85: The Contains() is a new function. You can use it to see if a string (*input,* on this line) contains the string used as the parameter (*help,* in this code).

We use this, because we want this IF to run when the user types the word *help* anywhere in their input. So, if they type, *"I need help",* or *"HELP ME!!!!",* we will display the list of valid commands. We also check if they typed a question mark (?) , a common way used in console app games to display the help information.

Line 107: If you want to create a string, with variable values inside it, you could do it with string concatenation – adding pieces of the string together, like this:

```
string myMessage = "Current Hit Points: " +
    _player.CurrentHitPoints.ToString();
```

However, Console.WriteLine() and string.Format() have a little bit cleaner way to do this. You have the string you want, with *{0}* in the location where you want to include a variable value. After the string, you have the variable you want to insert into the string. The function is smart enough to automatically do a *ToString(),* if it is needed.

If you want to put several variables in your string, you only need to add more sets of curly braces, and more variables to your list. Just remember that the position is important. The first variable in your list will go where {0} is. The second will go where {1} is, and so on. So, you could have something like this:

```
String myMessage =
    string.Format("X={0}, Y={1}, Z={2}", x, y, z);
```

Lines 117-160: In the Windows Form version, we hide the movement buttons for directions where there is no location. We can't do that in the console version, so we need to check if there is a valid location, before trying to move the player there. To be nice, we display a message if the location does not exist.

Lines 183-207: Similar to the movement validations, we need to check if there is a monster at the location, before we try to allow the player to *attack*. Because we don't have a combobox, with the player's weapons listed in it, we do a few checks, and try to pick a default weapon, when the player tries to attack the monster.

Lines 208-233: This is a more complex command. We are looking for the word *equip,* but we also need to know what weapon the player is trying to equip (set as their default weapon).

NOTE:
For functions like Substring(), the first character of a string is at position 0, not position 1.

The Trim() function removes any leading or trailing spaces from a string. So, if the user types *Equip rusty sword,* inputWeaponName will be *rusty sword.*

On line 208, we check if the player's input had *equip* – notice the space at the end. Then, we take the rest of the player's input string, starting at the sixth position.

On line 218, we look through the player's inventory and try to find an item with the same name as what the player is trying to equip. If it finds something, it will assign it to the *weaponToEquip* variable. If it does not find a matching weapon, that variable will be null (the default value).

If we don't find the weapon, we display a failure message to the player. If we do find it, we set it to the player's current weapon, and display a success message.

Lines 234-257: We do the same thing here, as we did for the equip weapon section, except we look for a potion to drink.

Lines 258-300: When the player inputs *trade,* we will show the player's inventory and the vendor's inventory, if there is a vendor at the player's current location.

On line 260, we check how many items the player has that they can sell (the price does not match our *flag* price that indicates the item cannot be sold). If the player does not have any sellable items, we display a message. If they do, we go to line 275, loop through their inventory, and display each item with its price.

In lines 283-298, we do the same for the vendor's inventory – except we don't need to check for unsellable items.

Lines 301-345: This is where we try to handle the player buying an item from a vendor.

After making sure there is a vendor at the player's current location, we do the same type of substring function that we did for equipping a weapon and drinking a potion. Then, we check if the vendor has that item, and if the player has enough gold to buy it. If so, we purchase it on lines 337-338, the same way we do on lines 167-170 of TradingScreen.cs.

Lines 346-383: Try to sell an item, using very similar parsing and logic as we did to buy an item.

Line 384: If the user's input did not match any of our previous checks, we reach the final ELSE and display our error message.

We have the DisplayCurrentLocation() function in order to have a consistent format. We want to display this information from a couple different places. So, instead of duplicating code, we have a single function.

NOTE:
If you did not do the SQL lessons, you can delete, or comment out, lines 406 (where we try to load the saved game from the database) and 425 (where we try to save it to the database).

The LoadGameData() function is the same as what we have in lines 26-38 of SuperAdventure.cs. The SaveGameData() function is the same as the SuperAdventure_FormClosing() function in SuperAdventure.cs.

Check that your program works

Run the program and try typing in some commands. Type *Help,* if you forget the game's commands.

Summary

Now, you have the same game, with two different front-ends.

The nice thing about having most of our game logic in the Engine class is that we can create this new front-end very quickly. I wrote all the code in Program.cs in less than two hours. In fact, it took me longer to write this lesson, than to write the code.

If you know XAML, you could probably create a WPF/XAML front-end project even faster. You would have similar UI controls (buttons, datagrids, etc.), and not need to figure out the user input parsing logic.

You could also create a unit test project, to do some automated testing of the game logic.

24

Final refactoring (cleanup) of the SuperAdventure source code

24 Make the SuperAdventure source code easier to understand and modify

Lesson objectives

At the end of this lesson, you will know…

- ➔ How to use Visual Studio, to help you clean up source code
- ➔ How to make your source code easier to read, which makes it easier to find bugs and make changes

There are thousands of different ways to write any program. Many of them will work as well as any other. However, you need to remember that some other programmer may need to make changes to it in the future. And, that future programmer may be you (in six months, when you've forgotten exactly why you wrote your code the way you did).

In this lesson, I'll show you some common techniques to make your code easier to understand – and how to use Visual Studio to help you do this. I'll be using Visual Studio Community 2015. Older versions may not have the same refactoring tools, or they may work differently.

None of the changes involve any new skill. They are only creating new properties, methods, and renaming a few things. However, I believe you'll see a version of SuperAdventure that is much easier to work with.

This refactoring changes 13 files. If you don't want to make all these changes to your SuperAdventure program, I suggest that you still read through the lesson and compare your code with the refactored code for this lesson (found at scottlilly.com). This will give you an idea of how your future programs should look – with smaller functions, and improved names.

Use source control, when refactoring code

When you refactor your program, or make any large group of changes, it's very helpful to use source control.

Before you make changes, check your code into your source code repository. After you make your changes, test your program. If it still works, check in the new version of your source code. If the changes don't work, and you can't fix them, you can *revert/rollback* your code to what it was before you started the changes.

If you are not familiar with source control, you can watch my video on *Installing TortoiseSVN (Subversion) and VisualSVN for Visual Studio 2015 on Windows 10,* at scottlilly.com. These are the same source control tools I use at home, and at the office.

STEP 1 At the top of many of our classes, we have several *using* statements. Some of them are added when Visual Studio creates the class file. However, we usually don't need all of them. We will clean up our classes by removing those excess lines.

Start Visual Studio, open Player.cs, and look at the *using* statements. You should see that these two lines are light grey:

```
using System.Text;
using System.Threading.Tasks;
```

They are light grey because the Player class does not use any classes from those namespaces. So, we can remove them. If you look at most of the other classes, you'll notice different *using* statements that are light grey, and can also be removed.

You could go through each file, and manually remove all these lines, but there is an easier way. In Visual Studio's menu, click on Edit ➡ IntelliSense ➡ Organize Usings ➡ Remove and Sort Usings. This will remove all the un-needed *using* lines, and sort the remaining ones in alphabetical order. You don't need to do the sort, but I like to keep them organized.

After doing this, I click the *Save All* button in Visual Studio's menu, and run the game, to make sure it still works.

It all worked, so I committed these changes to the source code repository. Remember to check in your changes after each successful refactoring change.

STEP 2 Make your comparisons sound more like a natural language.

Sometimes, IF conditions can be very complex. They might check several different values, and use complex combinations of *IF, AND (&&)*, and *NOT (!)*. This can also happen when a comparison contains negatives – some value *does not* equals another value. To make these easier to understand, you can change the variable, property, function, or series of comparisons to sound like how you would naturally speak.

For these situations, *wrap* the unnatural-sounding comparison behind a new function, or property, that has a clearer name. For example, in the Player MoveTo() function, we have this line, to see if a location has a quest:

```
// Does the location have a quest
if(newLocation.QuestAvailableHere != null)
```

The IF statement doesn't sound very natural when you read it aloud – at least, not to me.

So, I created a new property in the Location class that wraps that comparison. It does the exact same logic, but with a more natural name. Here's the code for that function:

```
public bool HasAQuest
{
    get
    {
        return QuestAvailableHere != null;
    }
}
```

Now, we can change the code in MoveTo() to this:

```
if(newLocation.HasAQuest)
```

This line sounds much more natural, and easier to understand. It's so clear, I removed the comment from the line above it. This is what **self-documented code** means. The source code is so easy to read, and understand, that you need very few comments.

I made several more similar changes to SuperAdventure, creating new properties and functions, with names that sound more natural.

STEP 3 *Inline* variables that are only used once, especially if they are used almost immediately after they are created.

Sometimes, we create variables that are only used once. They may serve the same purpose as the *wrapper* property we created in the previous step – to put a complex evaluation in an easy-to-understand location.

But, if we have well-named variables, properties, and functions, and we only use that variable in a few places, we can eliminate that variable (and that line of code) by placing its value *inline*.

In the Player class, we have:

```
bool playerAlreadyHasQuest =
    HasThisQuest(newLocation.QuestAvailableHere);
```

We only use that variable in one place, three lines later.

```
if(playerAlreadyHasQuest)
```

To eliminate the temporary variable, click on it and press Ctrl + . (the period/full stop key). This pulls up Visual Studio's refactoring tool. Choose *Inline temporary variable.* Visual Studio will show you where it will make the change, and you can click the *Apply* button to complete the change. Another line of code is eliminated.

I did this for several more variables that were only used once (or a few times), and were used almost immediately after they were created.

Sometimes, you may want to keep a single-use variable in your code, and not *inline* it. If you debug your program, you can set a breakpoint to see what happens when the variable is populated. That may be a little easier to follow than setting the breakpoint on the IF statement. It's one less thing happening on the line of code where you set the breakpoint.

STEP 4 Give variables, properties, and functions more descriptive names.

If you have unclear names for properties, functions, or variables, and they don't need to be *wrapped,* you can easily rename them in one place – and Visual Studio can update them every place they are used.

Left-click on a variable, property, or function name. It should have a light grey background.

Now, press the *F2* key. The background should turn green, and a box will pop up in the upper-right corner of the editing screen. Start typing, and Visual Studio will replace the name of your variable/property/function – every place it is used in your solution.

You can check the options in the pop-up box if you also want Visual Studio to change the name if it exists in comments or strings in your solution. If you check the *Preview Changes* box, Visual Studio will show you every place it will make the change, when you click the *Apply* button. In the preview, you can uncheck places where you don't want the change made, before Visual Studio does the replacement.

STEP 5 Break large functions into smaller functions that only do one thing.

The MoveTo() function is large, does a lot of things, and difficult to understand.

When you see a function like this, look for the parts that have one purpose. Move those lines of code to their own function, and have the large function call that new, smaller function. The smaller function will be easier to understand. It only has one purpose and has fewer lines and variables. Plus, the larger function will now have one line, where it used to have 15-30 lines. That will make the larger function easier to read.

When you see lines of code that you would like to move to a separate function, highlight them, click Ctrl + . (to bring up the refactoring menu), and click on *Extract Method.* Give the new method a name, and Visual Studio will create the function. It will also replace the lines you highlighted with a call to the new function.

Here is what the Player MoveTo() function looks like, before and after all the refactoring changes (see the next four pages).

```
 1  public void MoveTo(Location newLocation)
 2  {
 3      // Does the location have any required items
 4      if(!HasRequiredItemToEnterThisLocation(newLocation))
 5      {
 6          RaiseMessage("You must have a " +
                newLocation.ItemRequiredToEnter.Name + " to enter this location.");
 7          return;
 8      }
 9
10      // Update the player's current location
11      CurrentLocation = newLocation;
12
13      // Completely heal the player
14      CurrentHitPoints = MaximumHitPoints;
15
16      // Does the location have a quest?
17      if(newLocation.QuestAvailableHere != null)
18      {
19          // See if the player already has the quest, and if they've completed it
20          bool playerAlreadyHasQuest = HasThisQuest(newLocation.QuestAvailableHere);
21          bool playerAlreadyCompletedQuest = CompletedThisQuest(
                newLocation.QuestAvailableHere);
22
23          // See if the player already has the quest
24          if(playerAlreadyHasQuest)
25          {
26              // If the player has not completed the quest yet
27              if(!playerAlreadyCompletedQuest)
28              {
29                  // See if the player has all the items needed to complete the quest
30                  bool playerHasAllItemsToCompleteQuest =
                        HasAllQuestCompletionItems(newLocation.QuestAvailableHere);
31
32                  // The player has all items required to complete the quest
33                  if(playerHasAllItemsToCompleteQuest)
34                  {
35                      // Display message
36                      RaiseMessage("");
37                      RaiseMessage("You complete the " +
                            newLocation.QuestAvailableHere.Name + " quest.");
38
39                      // Remove quest items from inventory
40                      RemoveQuestCompletionItems(newLocation.QuestAvailableHere);
41
```

continued on next page

```
42                      // Give quest rewards
43                      RaiseMessage("You receive: ");
44                      RaiseMessage(newLocation.QuestAvailableHere.RewardExperiencePoints +
                            " experience points");
45                      RaiseMessage(newLocation.QuestAvailableHere.RewardGold + " gold");
46                      RaiseMessage(newLocation.QuestAvailableHere.RewardItem.Name, true);
47
48                      AddExperiencePoints(
                            newLocation.QuestAvailableHere.RewardExperiencePoints);
49                      Gold += newLocation.QuestAvailableHere.RewardGold;
50
51                      // Add the reward item to the player's inventory
52                      AddItemToInventory(newLocation.QuestAvailableHere.RewardItem);
53
54                      // Mark the quest as completed
55                      MarkQuestCompleted(newLocation.QuestAvailableHere);
56                  }
57              }
58          }
59          else
60          {
61              // The player does not already have the quest
62
63              // Display the messages
64              RaiseMessage("You receive the " + newLocation.QuestAvailableHere.Name +
                    " quest.");
65              RaiseMessage(newLocation.QuestAvailableHere.Description);
66              RaiseMessage("To complete it, return with:");
67              foreach(QuestCompletionItem qci in
                    newLocation.QuestAvailableHere.QuestCompletionItems)
68              {
69                  if(qci.Quantity == 1)
70                  {
71                      RaiseMessage(qci.Quantity + " " + qci.Details.Name);
72                  }
73                  else
74                  {
75                      RaiseMessage(qci.Quantity + " " + qci.Details.NamePlural);
76                  }
77              }
78              RaiseMessage("");
79
80              // Add the quest to the player's quest list
81              Quests.Add(new PlayerQuest(newLocation.QuestAvailableHere));
82          }
83      }
```

continued on next page

```
84      // Does the location have a monster?
85      if(newLocation.MonsterLivingHere != null)
86      {
87          RaiseMessage("You see a " + newLocation.MonsterLivingHere.Name);
88
89          // Make a new monster, using the values from the
               standard monster in the World.Monster list
90          Monster standardMonster = World.MonsterByID(newLocation.MonsterLivingHere.ID);
91
92          _currentMonster = new Monster(standardMonster.ID, standardMonster.Name,
                standardMonster.MaximumDamage, standardMonster.RewardExperiencePoints,
93                  standardMonster.RewardGold, standardMonster.CurrentHitPoints,
                        standardMonster.MaximumHitPoints);
94
95          foreach(LootItem lootItem in standardMonster.LootTable)
96          {
97              _currentMonster.LootTable.Add(lootItem);
98          }
99      }
100     else
101     {
102         _currentMonster = null;
103     }
104 }
```

continued on next page

MoveTo() AFTER

```
public void MoveTo(Location location)
{
    if(PlayerDoesNotHaveTheRequiredItemToEnter(location))
    {
        RaiseMessage("You must have a " + location.ItemRequiredToEnter.Name +
            " to enter this location.");

        return;
    }

    // The player can enter this location
    CurrentLocation = location;

    CompletelyHeal();

    if(location.HasAQuest)
    {
        if(PlayerDoesNotHaveThisQuest(location.QuestAvailableHere))
        {
            GiveQuestToPlayer(location.QuestAvailableHere);
        }
        else
        {
            if(PlayerHasNotCompleted(location.QuestAvailableHere) &&
                PlayerHasAllQuestCompletionItemsFor(location.QuestAvailableHere))
            {
                GivePlayerQuestRewards(location.QuestAvailableHere);
            }
        }
    }

    SetTheCurrentMonsterForTheCurrentLocation(location);
}
```

The MoveTo() function has gone from over 100 lines, to only 31 lines. The other lines are all in small functions, usually around 20 lines each.

If you want to change the SuperAdventure game – maybe add new features – it will be much easier to do with this smaller MoveTo function.

STEP 6 Repeat, until all the code is easy to understand.

When you program, it's usually a good idea to refactor often. After each change, look for anything you can clean up. Think of it like changing the oil in a car. If you always drive (add code), and never change your oil (refactor), eventually, your engine (and program) will break down. Then, it will take much longer time to repair.

I've worked on programs that have been in development for years, with dozens of programmers making changes and additions. Many times, management will say that we don't have time to clean up the code – we need to add in new features, *"Now!"*.

But, eventually, the code gets so bad that every change causes something new to break. When you keep your code clean, you reduce these problems.

Summary

When you create large programs, or programs that are modified long after they are written, it helps to have the code easy to read and understand.

I've worked on several projects that are millions of lines long, have been in development for years, and have had more than a dozen programmers work on them. When the code is not easy to understand, changes take longer, and are more likely to break something.

If you frequently do these refactoring techniques, your code will be *much* easier to work with.


PLAIN
FOREST
VILLAGE
LOCAL
WITCH
CASTLE
THE THREE
LAKES