C# Programming Tutorial *with Alien Invaders!*

Welcome to assignment #3.

As usual, I’m assuming you completed assignment #2, and haven’t changed it since then.

This time we’re going to see aliens!

Enjoy!

# Assignment #3 – Add aliens!

In this assignment, you will modify the AlienInvaders code as follows:

* You will create a new class called Sprite that will allow you to show aliens (and other things, eventually) on the screen.
* You will write a lot of support code to allow for animating the sprites.

As before, I will be covering a lot of new ideas, so hold onto your hat. Here we go!

## Step 1 – Create the Sprite class

In this step we’ll create a class called *Sprite*, and add a few things to it.

A lot of the old arcade games I grew up playing involved little pictures that moved around on the screen. The pictures themselves were rather static – usually alternating among 2 or 3 different frames to give the illusion of running or flashing or spinning or whatever – but they moved around smoothly on the screen. There were games like Galaga, Dig-Dug, the original Mario Bros., Defender, Asteroids, and of course, Space Invaders. Newer games like Warcraft and StarCraft are also like this (but they’re bit more advanced than what we’re going to do).

Among programmers, these little pictures that you animate by moving them around on the screen are called “sprites”. In this step, we’re going to create a special class to represent a sprite object on the screen.

To start, do the following:

1. In the Solution Explorer, right-click on the AlienInvaders project (not the solution) and select *Add* -> *Class…*
2. In the *Add New Item* dialog, type “Sprite” (without the quotes) and press the *Add* button (or hit Enter).
3. You should now be looking at this code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace AlienInvaders {

 class Sprite {

 }

}

You probably see stuff there that you recognize. We have our namespace – everything we write will go in that namespace. Then we have a class, called Sprite, and it’s completely empty. No constructor, no data or properties or methods. Pretty simple, right?

Of course it won’t stay that way.

The first thing I want to do is add a comment about the Sprite class. Just because I like to comment my code.

So add a nice comment like this:

// The Sprite class describes the graphical elements on the screen.

// Each and every graphical element (like an alien, the player's ship, and every other

// thing on the screen) will be represented by an instance of the Sprite class.

class Sprite {

}

Remember that a sprite is basically a little picture that moves around on the screen. So one of the main things that our Sprite class will need is an image – the image that represents the sprite.

So, add the following line to the Sprite class:

class Sprite {

 // This is the image that represents the sprite on the screen.

 private Bitmap \_bitmap;

}

*\_bitmap* will be where the Sprite class will store the image that shows on the screen.

The next thing a sprite needs is a location. Its current location on the screen.

As you probably already know, the location of anything on a 2-dimensional surface (like the surface of your computer screen) is defined in terms of X and Y. X is its horizontal position, and Y is its vertical position.

What you may not know is that in the computer world, the (0, 0) location is at the top-left of the screen.

This means that as X gets bigger, things move across the screen to the right, and as Y gets bigger, things move *down* the screen. This is kind of confusing to some people because in math or geometry class at school, increasing values of Y usually go upward, not downward.

Here’s a picture that will help illustrate this. The big rectangle is your computer screen.

 

To define a sprite’s location on the screen, we’re going to use a nice little datatype that .NET provides, called *PointF*. Remember *RectangleF* from the last assignment? *PointF* is like that. .NET provides a datatype called *Point*, in which *X* and *Y* are integers, but we’ll use *PointF*, in which *X* and *Y* are floating-point numbers. That will give us a lot more precise control over where things are on the screen.

Here’s the code to add to sprite.cs:

class Sprite {

 // The current location of the sprite on the screen.

 public PointF Location { get; set; }

 // This is the image that represents the sprite on the screen.

 private Bitmap \_bitmap;

}

(Remember that the gray stuff is just there to tell you where to put the new stuff)

Notice something different about that?

That’s called an “auto-property”. It’s something programmers call “syntactic sugar”.

Let me try to explain.

Remember how I said that all data members within a class should be *private*? And remember I also said that if you want to make some of your private data visible to the public, you should do so through *properties*? Remember my illustration about your nose and knees – how they’re visible to the public, but still private?

Well, a lot of programmers think this way – about data always being private, and “exposed” to the outside through properties only when you want to expose them.

Because of that, back in the old days, people would write code like this:

public int Data {

 get { return \_data; }

 set { \_data = value; }

}

private int \_data;

Now, let’s look at that code right there. What we have there is a private data member called *\_data*, like this:

private int \_data;

It’s private, which means it’s only visible – that is, it can only be used – inside of whatever class it’s part of. Outside the class it’s not visible at all.

The programmer wanted to give the outside world access to this piece of data, but he didn’t want to break his all-important rule about keeping data private, so he created a property called *Data*, where the “getter” just returns *\_data*, and does nothing else, and the “setter” just assigns a value to *\_data*, and does nothing else.

It seems kind of silly. Why not just make the data member public? Well, the reason for the data-members-are-always-private rule is hard to explain… it’s something you learn over years of programming. Anyway, it’s very common.

In fact, code like this was so common that the people who designed C# gave us a shortcut for doing this, called an auto-property, so that we could just write this:

public int Data { get; set; }

And that’s *exactly* the same as doing the more complicated code above, where we had the private data member *\_data* and the public property *Data*.

It’s just a shortcut for programmers. It’s not better or worse, it’s exactly (ok, not really exactly, but very, very close to exactly) the same whether you do it the old way with a private data member and a public property, or the new way with an auto-property.

This is the kind of thing that programmers call “syntactic sugar”. They call it that because it doesn’t really make your code any better – not in any measurable sense like faster or more efficient or anything like that. It’s just easier to type and uses fewer lines on your screen when you’re looking at code. So it’s like extra sugar on your donuts. It’s just a nice thing to have. And… it’s kind of neat.

There are many shortcuts like that in C#. Things that can be done in a longer way, but there’s a shortcut you can use if you want. I’ll point those out when we see them.

Anyway, that’s enough about auto-properties for now. There are a few other tricks you can do with them. We’ll see them again soon.

*Oops… you may be seeing errors!*

You might be seeing red squiggles underneath *PointF* and *Bitmap* right now in your code. If so, you need to add this line up at the top of the file, along with all the other “using” lines up there:

using System.Drawing;

## Step 2 – Adding more stuff to the Sprite class

Right now we have a Sprite class that has two things in it: an image, and a location.

Now we need to think about the fact that the window in which our game is played can be resized.

Remember that we set the initial size of our window to 600 by 600 pixels? But what if the user wants to maximize the screen, or make it smaller? If they do that, we will need our sprites to get bigger or smaller too.

In other words, we want our sprites to *scale* along with the window. Like when you watch a movie on a small TV screen, or a cell-phone screen. Everything scales down together, so that it looks the same on any size screen. We want our game to work that way.

To do this, we will add a *ScaleFactor* property to the class like this:

1. Add a private data member for the scale factor (just add it below *\_bitmap* in the class):

// Represents how much the sprite image needs to be scaled, both in height, and width.

private SizeF \_scaleFactor;

Remember *RectangleF* and *PointF*? Well, this time we’re using *SizeF*, which is a version of the *Size* datatype that uses floating-point numbers for *Height* and *Width*. They represent the scaling factors in the horizontal (*Width*) and vertical (*Height*) directions.

Suppose the user makes the game window smaller so that instead of 600 by 600, it’s 300 by 300. That’s half as big as its normal size. In that case we’d want all our sprites to be half as big too. So if the screen is half-size, we’d set the “scale factor” of the sprites to (Height=0.5, Width=0.5). (0.5 is the floating-point number that’s equal to ½ -- you probably already knew that).

1. Add an auto-property called *ScaledBitmap*.

Here’s the code to add. Put it below the Location property in the code:

// The sprite's image, scaled according to the current scale factor.

public Bitmap ScaledBitmap { get; private set; }

Notice that the “setter” is private. That means that the outside world (outside the Sprite class) can *see* the *ScaledBitmap* property (the AlienInvadersForm class will need to be able to get the scaled bitmap so that it can draw it on the screen), but it will not be possible to *set* the value of this property outside the class. The reason for this is that *ScaledBitmap* will be generated internally by the class whenever the scale factor is changed.

1. Add the *ScaleFactor* property to the class.

Now this is going to be a bit complicated. Here’s the code for this property (put it below *ScaledBitmap*):

// This property represents how much the sprite's image needs to be scaled, both in height

// and in width.

// When the property is set, the ScaledBitmap is generated, and the Location is modified

// accordingly.

public SizeF ScaleFactor {

 // For the getter, just return the value of \_scaleFactor.

 get { return \_scaleFactor; }

 // For the setter, we set the value of \_scaleFactor, and we also generate the new

 // ScaledBitmap and modify the sprite's location accordingly.

 set {

PointF unscaledLocation = new PointF(Location.X / \_scaleFactor.Width,

 Location.Y / \_scaleFactor.Height);

\_scaleFactor = value;

Location = new PointF(unscaledLocation.X \* value.Width,

 unscaledLocation.Y \* value.Height);

var scaledSize = new SizeF(\_bitmap.Width \* value.Width,

 \_bitmap.Height \* value.Height);

ScaledBitmap = new Bitmap(\_bitmap,

 new Size((int)Math.Round(scaledSize.Width),

 (int)Math.Round(scaledSize.Height)));

 }

}

Ok, this may be getting over your head. That’s a chunk of complicated code right there.

Nevertheless, I’ll try to explain some of it.

For the “getter”, we’re just returning the value of *\_scaleFactor*. This is the kind of thing where I might have used the auto-property shortcut. But I can’t do that because the “setter” is more complicated than just setting the value of a private data member. You can only use auto-properties in the super-simple situation I described earlier.

So what are we doing in that setter?

Basically, what the setter is doing is:

1. Update the sprite’s location so that it corresponds with the scale factor.

Again, think about what happens when the window is shrunk from 600 by 600 to 300 by 300. Suppose a sprite was at the location (300, 300) – right in the middle of the 600 by 600 screen. When the window is shrunk by half, the sprite’s size will have to also shrink by half, *and* its location will have to shrink by half too. With the smaller screen, the center is now (150, 150) . So that’s part of what the code is doing.

1. Generate a scaled image and store it in *ScaledBitmap*.
2. Set the *\_scaleFactor* data member to the new value.

Exactly how it’s doing those things involves some pretty complicated math. It’s not super important that you understand that now. If you want me to explain it in more detail, you can write to me and ask.

Ooookay… so here we go. We’re going to dive in. In this step I’m going to explain the code we already have, that was written for us by the designer tool.

I’m going to introduce a lot of new ideas, so this might get long.

First of all, go to the Solution Explorer and double-click on the file AlienInvadersForm.cs. You should now be looking at the empty window with the title “Alien Invaders”, like this:



This is what’s called “design mode”. That is, you are looking at the AlienInvadersForm class in a designer tool that lets you see what the form will look like when the program is run.

To get to the actual C# code that defines this form (by the way, the word “form” is pretty much synonymous with the word “window” for this project), right-click on the form you see now and select *View Code*. This will show you the source code that defines the form. It should 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;

namespace AlienInvaders {

 public partial class AlienInvadersForm : Form {

 public AlienInvadersForm() {

 InitializeComponent();

 }

 }

}

Again, the colors may be different, and there may be some formatting differences, but this ought to be what your code looks like now.

1. Namespaces

For a moment, ignore all the lines at the top of the file that start with the word “using”. We’ll get to those later. The next line is:

namespace AlienInvaders {

What’s a “namespace”, you might ask?

Well, suppose we wanted to name something “X”. Maybe we have a variable and we want to give it the name “X”, like this:

int X = 1;

Now, a problem would arise if there’s already something called “X” somewhere. Maybe down in the depths of all the .NET stuff (remember .NET contains a lot of stuff!) there’s already a variable called “X”. This would break our code because you can’t have two things with exactly the same name.

That’s where namespaces come in. Namespaces allow us to subdivide our code so that names don’t overlap. If we define a variable called “X” inside our “AlienInvaders” namespace, the full name of that variable will be “AlienInvaders.X” (which is pronounced “alien invaders *dot* x”), not just “X”. All of the stuff in .NET is also divided up into different namespaces, so unless there’s an “AlienInvaders” namespace in .NET (not likely), there’s no way any of our names will collide with anything in there.

This is important because .NET contains literally thousands of things, many hundreds of thousands of lines of code, and without namespaces we’d surely be bumping into their names all the time, and it would cause us no end of problems.

1. Classes

The next line looks like this:

public partial class AlienInvadersForm : Form {

“public” simply means that the AlienInvadersForm class is available to be used by anyone. If you were to post this project online, “public” says that anyone who downloads your project can use this class.

“partial” means that part of this class defined elsewhere. Don’t worry about that for now, we’ll talk about it later.

“class” means that you’re defining a new class.

What’s a “class”, you say?

A class is the code that defines some kind of thing. Down in the bowels of Windows, there are classes that define a window, a button, the mouse pointer, the keyboard, a printer, the screen, and a thousand other things. In this case, we are creating a class that will define the Alien Invaders game window, and we’ve called it “AlienInvadersForm”.

Inside of any class you have three main things: you have “properties”, and “methods”, and you have “data”.

Think about it this way:

Suppose I were to write a class called “Pencil”.

My Pencil class might define different *properties*, such as the pencil’s color, its length, whether it’s sharp or dull, how much eraser it has left, etc. Those are all *properties* of the pencil.

Then I might define some *methods*, like *Write*, *Erase*, and *Sharpen*. Those are all things that a pencil can *do*, or things that can *be done* with a pencil. The code for *Write* would instruct you to scratch the tip of the pencil across the paper. The code for *Erase* would instruct you to flip the pencil over and rub the soft, pink end across the paper. The code for *Sharpen* would instruct you to insert the tip into a sharpener and twist the pencil or turn a crank or something.

Generally speaking, properties are nouns, and methods are verbs (but this is not absolutely always the case, as we will see later on).

*Data* for the pencil is all the private stuff internal to the pencil. It might include its current color (probably yellow, and made available to the user through the color property), its current length (also probably visible to the outside through a length property), its composition (whether it’s made of wood or some artificial stuff), the composition of its writing material (which might be lead or graphite), and many other things – many of them purely internal to the pencil, such as its molecular properties, the tiny imperfections or cracks that exist inside it, etc.

Generally speaking, properties and methods are public things – visible to the outside world, while data elements are private things – things only known by the class itself. This will probably become clearer as we go on.

1. Inheritance

Look at that line again:

public partial class AlienInvadersForm : Form {

That little part at the end where it says “: Form” means “inherits from Form”.

“Form” is a class defined by .NET. It’s the most basic kind of window – just an empty window with a frame around it. What this is saying is that we want our AlienInvadersForm to be everything that the Form class is, do everything the Form class does (in other words, include all the properties, methods and data that a Form has), plus whatever additional stuff we write.

In programming lingo, this means that AlienInvadersForms is a *child* of the Form class, or that it is *derived from* the Form class. It also means that Form is the *base class*, or *parent class* of AlienInvadersForm. These are all terms you might see me using later.

1. Constructors

Next there’s this:

public AlienInvadersForm() {

 InitializeComponent();

 }

That’s a *constructor*. Whenever you have a method inside a class that has the same name as the class, you know it’s a constructor.

A constructor is where you put code to build your class. In this case, we’re just calling a method called InitializeComponent (we’ll talk about that later – it’s automatically generated by the design tool).

Think of a constructor like a factory. If you go to the car dealership and say “I want a brand new Cadillac with a bright red roof and yellow doors”, they’re probably not going to have that on their lot, so they’ll have to call the Cadillac factory and say “this customer wants one with a bright red roof and yellow doors”. The factory will then build one and send it to the dealership, where you’ll have to pay a lot of money for it.

A constructor is like that factory. Whenever you write some code like this:

var myform = new AlienInvadersForm();

You’re like the guy asking for a new Cadillac. This is saying “call the constructor for the AlienInvadersForm and create a new instance of that class”.

Remember that a “class” is just the code that defines something – an “instance” of a class is an actual, physical thing.

A class is like the blueprints for a house, while the actual house is an instance of the class. You could then go on to build several more houses using the same blueprints, and have several instances of the “house” class that all look the same. The blueprints are not the house itself – they define what the house will be.

## Step 2 – Override the BackgroundImage property.

Like I said, that was long. If you don’t completely understand it all right now, don’t worry. It will (hopefully) get clearer as we go.

Remember how our AlienInvadersForm is *derived from* the .NET class called Form? Well, the Form class has a property called BackgroundImage. This property allows you to set an image to be drawn as the background of the form, instead of a solid color.

However, there’s a problem with using this, if you want something to be animated on your form. Every time your form (or some part of your form) gets repainted on the screen, Windows will first draw the background image, and then draw your animated things on top of that. The result will be a constant flickering to your animation, which looks bad.

In addition, Windows will be always re-scaling your image to fit your window. This can take valuable time away from the job of animating your screen, which might cause your animation to be jerky. We want to take control of both the drawing of things on our window, and everything else that might take time away from the task of animating our game.

In general, whenever you’re animating things on the screen, you want the movement to be smooth, with no flickering or jerky motion -- at least as smooth and non-jerky as possible. This means you don’t want Windows wasting time drawing things on the screen that it doesn’t need to. You want it to only draw the things that are changing.

We’ll have to do some things to accomplish this. It will take a bit of work. The first things are:

1. We will manage for ourselves which parts of the form need to be redrawn at any time (that prevents Windows from spending time redrawing parts of the form that haven’t changed).
2. We will override the BackgroundImage property to take control of how this property works.

“Overriding” something simply means that for our class, we want a certain property or method that is defined by the parent class to be different for our class.

Think about it this way: you inherited many properties from your parents, such as your skin color, your eye color, and your hair color. But suppose you decide you want **blue** hair. So you go to the store and buy some hair color and apply it. In a sense, what you’re doing is *overriding* the hair color *property* that you inherited from your parents. You’re saying “yeah, I’m fine with inheriting a lot of things from my parents, but for my hair – I want it to be different than my parents’ hair. I want it to be blue.”

That’s what we’re going to say about the BackgroundImage property. “In general, I want my new Form to inherit everything from the Form class, but for the BackgroundImage property, I want it to be different.”

So, here’s what to do:

1. Create some data members.
Remember that *data members* are the *private* stuff only known inside the class. We are going to create two private data members: one is the background image scaled to fit inside our form, and the second is a rectangle that defines what part of the form has changed since the last time it was drawn on the screen.

Put this code at the bottom of the class, just below the constructor, like this:

public partial class AlienInvadersForm : Form {

 public AlienInvadersForm() {

 InitializeComponent();

 }

 // This is the rectangle that covers everything that's changed since the last time the screen

 // was drawn.

 private RectangleF \_rectToRepaint;

 // This is a version of the background image scaled to the current size of the form.

 private Bitmap \_scaledBackground;

}

In the above code, the stuff in gray is code you already have. The colorful stuff is the new code. I’ve done it that way to help you understand where to put the new code. Just copy the new parts and add them to AlienInvadersForm.cs in the proper spot.

Some things about those two data members:

* Data members are “private”.

*This is important. Remember it.*

Why wear a bathing suit in the pool? Because you’ve got some *privates* that you don’t want everyone looking at all the time!

But think again. How about your nose? Is it private? What about your knees? Suppose you were in a hotel swimming pool, and some stranger came up and started touching your nose. Then he starts feeling your knees. Would that be ok with you? Probably not, because your nose and your knees *are* *private too*. Yeah, they’re visible to the public – you don’t hide them under a swimming suit, but that doesn’t mean you want strangers doing whatever they want with your nose and knees.

In the same way, *data members of a class ought to be private*. If you want them visible to the public, fine – you do that by creating a *property* that makes them available (we’ll be looking at how to do that in a moment), but your data members are private.

In C#, you make something private by putting the keyword “private” in front.

For example, suppose I were to write a class like this (don’t put this in your code, it’s just an example):

 public class Something {

 public int abc;

 private int xyz;

 }

Now, if I were to create an instance of the *Something* class in some other part of my code, like this:

var s = new Something();

I could access the “abc” data member, because it’s public, but I could not access the “xyz” data member, because it’s private to the Something class. In other words, I could do this:

s.abc = 3; // This is ok.

But not this:

s.xyz = 5; // This is not ok. This will cause a compiler error.

Note that all this about making data members private is just good advice – it’s not required. You can make data public if you want. Police will not break down your door if you make data members public. But, like brushing your teeth, making data members private – and only making them available via properties when you want to – is a good habit to acquire as a programmer.

* Private data member names start with underscores.

Notice how the names *\_rectToRepaint* and *\_scaledBackground* start with underscores? This is to help later on when you’re looking at a lot of code, and it can be hard to tell what’s what. The underscores at the start of the names help to distinguish them from other variables that you might have. It’s not required, but many programmers do this as a good habit – again, like brushing your teeth. Police won’t arrest you if you don’t brush your teeth, but it’s a good thing to do that will help you later on. There are a lot of good habits of programming that I’ll be telling you about – things that are not required, but are a good idea.

* RectangleF

Notice that the datatype of *\_rectToRepaint* is “RectangleF”, not just “Rectangle”.

Both *Rectangle* and *RectangleF* are datatypes defined by .NET. In a normal *Rectangle*, the location (X, Y) and size (Height, Width) of the rectangle are whole numbers (integers like 1, 2, 3, etc.).

The “F” at the end of *RectangleF* means that the location and size of the rectangle are floating-point numbers, rather than integers. They can have values like 1.01, or 13.8. Using the “F” version will allow us to have greater accuracy when dealing with things on the screen.

* Bitmap

“Bitmap” is the datatype of \_scaledBackground. You probably already know what a bitmap is – bitmaps are how images are stored in computers. The *Bitmap* datatype is defined for us in .NET.

1. Override the BackgroundImage property.

Now we finally get to doing this. Here is the code:

public AlienInvadersForm() {

 InitializeComponent();

}

 // Override the BackgroundImage property so that we can scale the image ourselves, and maintain

 // what parts of the image need to be redrawn.

 // When the BackgroundImage property is changed, the whole form needs to be redrawn.

 public override Image BackgroundImage {

 // The "getter" returns the current value of the property.

 // Here we just return the base class's BackgroundImage property value.

 get { return base.BackgroundImage; }

 // The setter sets the current value of the property.

 // When the background image is changed, we need to:

 // 1. Change the value of this property in the base class.

 // 2. Create our scaled version of the image.

 // 3. Set \_rectToRepaint to say that the whole form needs to be redrawn.

 // 4. Call Invalidate to tell Windows to redraw our form.

 set {

 // Change the BackgroundImage value in the base class (Form).

 base.BackgroundImage = value;

 // Create a copy of the image, scaled to the size of our form.

 \_scaledBackground = new Bitmap(value, ClientRectangle.Size);

 // Set \_rectToRepaint to the entire form area.

 \_rectToRepaint = ClientRectangle;

 // Tell Windows to redraw our form.

 Invalidate();

 }

 }

 // This is the rectangle that covers everything that's changed since the last time the screen

// was drawn.

 private RectangleF \_rectToRepaint;

 // This is a version of the background image scaled to the current size of the form.

 private Bitmap \_scaledBackground;

Here again I’ve tried to make it clear where the new code goes by putting the gray code around it. Only the new code needs to be added, of course. Just copy it from here and paste it into the source code (AlienInvadersForm.cs) at the spot indicated.

Now, here are some things to learn from this code:

* Properties have getters and setters.

Look at the property code. Notice that there’s a “get” section, and a “set” section. The “get” section is called the “getter” (or sometimes the “get accessor”, but I don’t normally use that term), and the “set” section is called the “setter” (or “set accessor”).

The “getter” is where we put code that returns (or “gets”) the property’s value. The “setter” is where we put code that changes (or “sets”) the property’s value.

In addition to just setting or getting the value, this allows us to do other stuff when a property is accessed, like the stuff we’re doing in the setter for BackgroundImage.

* The “value” keyword.

Within a “setter”, the keyword *value* refers to whatever value the property is being set to. You can see it being used twice in the code for BackgroundImage’s setter. First when we set the base class’s value:

base.BackgroundImage = value;

And second when we create the scaled version of the image:

\_scaledBackground = new Bitmap(value, ClientRectangle.Size);

* ClientRectangle

This is a property of Form which simply gives a rectangle the size of the whole area of the form that we can draw on. This line:

\_rectToRepaint = ClientRectangle;

sets *\_rectToRepaint* to be the whole rectangle that covers the form’s area. We do this because whenever the background image is changed, we will have to repaint the whole form. Of course, the background image will not be changing in our game, but it will change one time – at the very start of the program when it’s set for the first time. At that point, we will need Windows to paint the whole form.

* So many comments!

You may have noticed that there are a lot of comments in that code. Comments are lines that start with two slashes (“//”) and usually show up in green in Visual Studio.

Putting comments in your code is a very good idea in general. You might not have time to work on this project for a few days, and when you get back to it, it might be hard to remember what you were trying to do. Later on, when this project becomes quite a bit bigger, the comments will make it easier to read the code.

Most of the time, I do not put *that* many comments in my code. I’m doing it now to help you understand what’s going on. In the future I may not put quite so many comments in the code. However, this is one of those “good ideas” that aren’t required. Good programmers put a lot of comments in their code, explaining what the code is trying to do.

## Step 3 - Set the BackgroundImage property of AlienInvadersForm.

Now we’ll set the BackgroundImage property to the background we want our game to have. Our game takes place in outer space, of course, so we need to set the background to a nice starry outer-space scene.

Here’s how:

1. Get the background image from my website.

Using an internet browser, go to this location:

<http://www.timblaisdell.com/alieninvaders/assignment2/>

There you will find a file called “stars background.jpg”.

Right-click on the link for that file and select “save as” (depending on your browser, it might be “save target as” or “save link as” or something similar). When the “save as” dialog comes up, don’t change the name, just save the image to your “Pictures” folder.

1. Add the background image to your project’s resources.

Close the browser and go back to Visual Studio.

* Go to the Solution Explorer.
* Right-click on the AlienInvaders project (the project, not the solution, which is also called AlienInvaders – the project will be in **boldface**) and select *Properties*. This will bring up the properties editor for the project.
* Down the left-hand side of the properties editor is a list of tabs you can select. At the top are *Application*, *Build*, *Build Events*, etc. Click on *Resources*.
* Now, look at the top and find the *Add Resource* button. It has a little down-arrow thingy to its right. Like this:



Click on the little down-arrow and select *Add Existing File…*

* In the dialog that comes up, navigate to your Pictures folder where you saved *stars background.jpg* and select it. Then hit *Ok*.
* You should now see that the starry image has been added to your resources. It should have the name “stars\_background” underneath it, like this:



1. Set the BackgroundImage property of AlienInvadersForm to the starry image.
* Go back to the designer for AlienInvadersForm.cs.

You might already have it open and can just click on the tab for it at the top of the screen (which will say “AlienInvadersForm.cs [Design]”), but if you don’t have it open, double-click on the file in the Solution Explorer.

* Right-click on the form and select *Properties*.
* Find BackgroundImage in the properties list. Its current value should be “(none)”. Once you’ve selected it, it should look like this:



* Click on the little “…” button on the far right of the property listing. This will bring the “Select Resource” dialog up.
* There should be a list box in there with the *stars\_background* image listed in it. Select it and press *OK*.

## Step 4 – Test it.

Now you should just be able to hit the Start button again (like in the test phase of Assignment #1) and see this:

0

That’s not quite how we want our background to look. The stars are too big, because the original image (not the scaled one) is currently being drawn. We’ll fix that, and do a bunch of other things, in the next assignment.

If any of this is confusing or if you are unable to get the program to work for some reason, please contact me and ask questions. It may be that I’ve said something wrong in here that I need to change.

In the next assignment, we’ll be overriding more methods, and end up with an alien on the screen! Woo hoo!