C# Programming Tutorial *with Alien Invaders!*

Welcome to assignment #2.

If you’ve gotten this far, it means you’ve completed assignment #1. I am going to assume that you’ve not changed the AlienInvaders solution or project from what it was at the end of assignment #1.

If this is not the case, let me know and I’ll send you a fresh copy of assignment #1, so you can start from there when doing this assignment.

Now onward…

# Assignment #2 – Add a starry background and start preparing to animate

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

* You will begin to get the project ready to support smooth animation.
* You will add a background showing an outer-space star field.

In the process of doing those two things, I will be introducing several new programming ideas. So get ready to learn new things.

## Step 1 – About the code generated by the designer.

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 the “form designer”. 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 a lot 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 thousands of things, and without namespaces we’d surely be bumping into their names all the time, and it would cause us no end of problems.

###  2. 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 “data”.

Think about it this way:

Suppose I was 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 color and length. A pencil’s color and length are not things you can change, except by doing something to or with the pencil. For example, if you sharpen the pencil, it’s length will change, but you can’t just tell the pencil to be shorter or longer without doing something with it that affects the length.

Other data for a pencil might include 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.

### 3. 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.

Back to my Pencil class. Suppose I’d already written this Pencil class some time ago. I worked long and hard on it, and I know it works well. Then one day I’m working on a new project and need a colored pencil – that is, a pencil that writes in color.

My original Pencil class didn’t include any concept of colored writing. It had a “color” property, but that was about the color of the pencil itself (e.g. yellow), not the color of the lead (or whatever) inside it. I need a new class that’s exactly like “Pencil” in every respect, except that it also includes a new property – the color of the lead inside.

This is where inheritance comes in. I don’t have to re-write my whole Pencil class, I just derive a new class from Pencil – maybe I call it ColoredPencil. Then all I have to do is write the new stuff – the stuff that’s different for a ColoredPencil. My code might look like this:

public class ColoredPencil : Pencil {

### 4. 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. You might write some code like this:

var myform = new AlienInvadersForm();

That’s like the guy asking for a new Cadillac. It’s saying “call the constructor for the AlienInvadersForm and create a new instance of that class”.

In programming, a “class” is the code that defines something – an “instance” of a class is the actual thing that the class defines.

With my Pencil class – if I say “Pencil”, a lot of things come to your mind. A pencil is a long, thin thing with some lead or graphite running though the middle. You write by scratching the pointy end along some paper, and there’s an eraser on the other end. All that information is the class “Pencil” – but it’s not a pencil. You can’t write with just the concept of a pencil. When you actually hold one in your hand, that thing you’re holding is an “instance” of the class.

Another way to think about it is that a class is like the blueprints for a house, while the actual house is an instance of the class. After building a house based on the blueprints, you could then go on to build several more houses using the same blueprints, and have several instances of the “house” class that the blueprints define. 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 other stuff 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. 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.

Some things about those two data members:

* They are “private”.

Generally speaking, a class’s *data* is the stuff it uses internally, but can’t been accessed by the outside world. *Properties* are public, *data* is private.

Back to my pencil class: “Color” (that is, the external color of the pencil) is a property of the pencil. It’s public in the sense that everyone can see it – everyone has *read-only* access to it. They can see that it’s yellow, but they can’t make it green or blue unless they do something to it (like paint it, or remove the paint that’s on it).

So the “Color” property provides the outside world with read-only access to a private data member we might call “\_color”. Internally within the Pencil class, we might have a “Paint” method that changes the color. The outside world can see that the color is yellow, but they can’t change the color without invoking the “Paint” method (that is, painting it).

If this is a bit confusing, don’t worry. All I’m saying is that when I talk about “data”, I’m usually talking about “private” data, and “properties” are generally things that give access (maybe limited access, like read-only) to private data.

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.

That 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 (not all) 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. With RectangleF the location and size of the rectangle are floating-point numbers, rather than integers. The .NET graphics functions allow you refer to locations like (1.3, 2.4) or whatever. Even though a computer screen is made up of pixels that can’t be subdivided, Windows will use anti-aliasing (google it if you don’t know what it is) to give the illusion that (1.3, 2.4) is a tiny bit to the right and down from plain old (1, 2).

* 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.

### 2. 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.

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 getting or setting the value, getters and setters allow 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.

*Note: Some people say that code ought to be “self-documenting”, meaning that it ought to be written and formatted in such a way that it’s easy to tell what the code is doing just by looking at it. While I agree that “self-documenting” code is a good thing to shoot for, I’m putting a lot of comments in here to make it easier (hopefully) for an inexperienced programmer, who maybe can’t read code like it’s his or her native tongue, to understand what it’s doing.*

## 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. 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 on the right side of the button 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!