XNA0118-The XNA Framework and the Game Class*

R.G. (Dick) Baldwin

This work is produced by OpenStax-CNX and licensed under the Creative Commons Attribution License 4.0

Abstract

Use a very simple XNA program to learn many of the details regarding the incorporation of the XNA framework into the object-oriented C# programming language. Also learn about constructors, the this keyword, the base keyword, and some of the differences between a Console Application and a Windows Game application.

Revised: Sat May 07 18:29:12 CDT 2016

This page is part of a Book titled XNA Game Studio

1 Table of Contents

- Table of Contents (p. 1)
- Preface (p. 4)
  - Moving from C# to XNA (p. 4)
    * My objective (p. 4)
    * Fasten your seatbelt (p. 4)
  - Viewing tip (p. 4)
    * Figures (p. 4)
    * Listings (p. 5)
- General background information (p. 5)
  - A software framework (p. 5)
    * What is a software framework? (p. 5)
  - New project choices in Visual C# (p. 6)
  - A Console Application (p. 6)
    * Start debugging (p. 6)
    * The Main method (p. 6)
    * The starting point for a Console Application (p. 6)
  - A Windows Game Application (p. 7)
    * Source code files (p. 7)

*Version 1.2: May 7, 2016 6:30 pm -0500
1http://creativecommons.org/licenses/by/4.0/
1http://cnx.org/contents/GY804-eY
* Build and debug (p. 7)
  * The file named Program.cs (p. 8)
    * Instantiate a Game1 object and call the Run method (p. 8)
    * What do we know about the class named Game1? (p. 8)
    * The using directive (p. 8)
    * Memory management (p. 9)
    * Don’t modify the file named Program.cs (p. 9)
  * The Game1 class (p. 9)
  * How to create a game (p. 11)
    * Game1 extends Game (p. 11)
    * Main method calls the Run method (p. 11)
    * The Run method (p. 11)
    * Override five methods (p. 11)
    * Initialization (p. 11)
    * The game loop (p. 12)
    * Override Update for game logic (p. 12)
    * Override the Draw method (p. 12)

• Preview (p. 12)
  * Program output (p. 12)
  * What if you don’t honor the transparent background? (p. 13)
  * Not a very exciting program (p. 14)

• Discussion and sample code (p. 14)
  * Creating a new Windows Game project (p. 14)
    * Step 1: Create a new Windows Game project named XNA0118Proj (p. 14)
      * Select a Windows Game project (p. 15)
    * Step 2: Add your image file to the Content folder (p. 17)
      * Add your image to the Content folder (p. 19)
      * The Asset Name (p. 19)
    * Steps 3, 4, and 5: Write code (p. 19)
      * Modify two overridden methods (p. 19)
      * Will discuss in fragments (p. 19)
  * Beginning of the class named Game1 (p. 19)
    * The namespace (p. 19)
  * General information (p. 20)
    * The superclass named Game (p. 20)
    * Overridden methods (p. 20)
    * The game loop (p. 20)
    * The big picture (p. 20)
  * The constructor for the Game1 class (p. 21)
    * What is a constructor? (p. 21)
    * A new GraphicsDeviceManager object (p. 21)
    * Passing the this keyword as a parameter (p. 22)
    * The ContentManager (p. 22)
    * When the constructor in Listing 5 terminates... (p. 22)
  * The Run method of the Game class (p. 22)
* The BeginRun method (p. 23)
  - The game loop (p. 23)
  - The overridden LoadContent method (p. 23)
    * Two new instance variables (p. 23)
    * The Texture2D class (p. 24)
    * The SurfaceFormat enumeration (p. 24)
    * The Texture Format of my image (p. 24)
  - A new SpriteBatch object (p. 24)
    * The SpriteBatch class (p. 25)
      - This can be confusing (p. 25)
      - The inherited property (p. 25)
    * The new code (p. 25)
    * Generic methods (p. 25)
      - Required syntax for the Load method (p. 25)
      - What do the angle brackets mean? (p. 25)
      - Calling the Load method of the current ContentManager (p. 26)
      - Populate the variable named myTexture (p. 26)
    * The Vector2 structure (p. 26)
  - The overridden Game.Draw method (p. 26)
    * Game loop timing (p. 26)
  - More general information (p. 27)
    * What is a sprite? (p. 27)
    * What is a bitmap? (p. 27)
    * What is GDI+? (p. 27)
    * What about our image? (p. 27)
  - Beginning of the Game.Draw method (p. 27)
    * GameTime information (p. 28)
    * The call to the GraphicsDevice.Clear method (p. 28)
    * The Color class (p. 28)
      - Type ARGB (p. 28)
      - Constructors, methods, and constants (p. 28)
    * Code to draw the sprite (p. 29)
      - Honor the alpha values (p. 29)
      - Ignore the alpha values (p. 29)
      - Drawing the sprite(s) (p. 29)
    * Overloaded Draw methods (p. 29)
    * The SpriteBatch.End method (p. 30)
  - Call Game.Draw on the superclass (p. 30)
    * Execute both versions of the overridden method (p. 30)
    * A required statement (p. 30)
  - The end of the program (p. 31)

- Run the program (p. 31)
- Run my program (p. 31)
- Summary (p. 31)
- Miscellaneous (p. 31)
- Complete program listing (p. 32)
2 Preface

This module is one in a collection of modules designed primarily for teaching GAME 1343 Game and Simulation Programming I at Austin Community College in Austin, TX. These modules are intended to supplement and not to replace the textbook.

The modules were originally published for use with XNA 3.1 and have been upgraded for XNA 4.0. This upgrade had very little impact on earlier modules in this collection. However, beginning with this module, we begin to see major differences between version 3.1 and version 4.0 of XNA. In May of 2016, the modules are being updated for use with version 4.0 Refresh. This will have very little impact on the modules.

Here is what Microsoft has to say about the newer product:

Microsoft XNA Game Studio 4.0 Refresh updates XNA Game Studio 4.0 to fix bugs and add support for developing games that target Windows Phone OS 7.1 and developing games in Visual Basic.

This course doesn’t address Visual Basic or Windows Phone, but bug fixes are always welcome.

An earlier module titled Getting Started provided information on how to get started programming with Microsoft’s XNA Game Studio.

2.1 Moving from C# to XNA

All of the modules prior to this one have been preparing you for this module. This is the module where we will begin applying what you have learned about C# and OOP to the XNA framework.

2.1.1 My objective

My objective is that you fully understand what you are doing when you write C# programs using the XNA framework. I don’t want you to simply be filling in the blanks and hacking something together in hopes that you can find a combination of statements that seems to work. If we can accomplish that, you will be prepared to go much further into the sophisticated use of the XNA framework on your own after you complete the course.

2.1.2 Fasten your seatbelt

This module may be a rough ride from a technical viewpoint, so fasten your seatbelt, arm yourself with a pot of coffee, and let’s go.

2.2 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

2.2.1 Figures

- Figure 1 (p. 7). Initial game window.
- Figure 2 (p. 12). Raw image with an (almost) transparent background.
- Figure 3 (p. 12). Cropped upper-left corner of the game window
- Figure 4 (p. 13). Cropped upper-left corner of the game window without honoring alpha transparency.
- Figure 5 (p. 14). Select New Project on the Visual C# File menu.
- Figure 6 (p. 15). Select a Windows Game project.
- Figure 7 (p. 17). Solutions explorer and properties window exposed.

http://cnx.org/content/m49509/1.2/
• Figure 8 (p. 25). The Load method of the ContentManager class.

2.2.2 Listings

• Listing 1 (p. 6). Initial contents of the file named Program.cs for a Console Application.
• Listing 2 (p. 8). The file named Program.cs for a Windows Game project.
• Listing 3 (p. 9). Initial contents of the file named Game1.cs.
• Listing 4 (p. 19). Beginning of the class named Game1.
• Listing 5 (p. 21). Constructor for the Game1 class.
• Listing 6 (p. 23). The overridden LoadContent method.
• Listing 7 (p. 27). Beginning of the Game.Draw method.
• Listing 8 (p. 29). Draw the sprite.
• Listing 9 (p. 30). Call Game.Draw on the superclass.
• Listing 10 (p. 32). The Game1 class for the project named XNA0118Proj.

3 General background information

3.1 A software framework

XNA is a very sophisticated C# application. It isn’t simply a program in the sense of the programs that you have seen so far in this course or a word processor or a spread sheet. Instead, it is a software framework designed specifically to make it easier for you to create computer games using the C# programming language.

3.1.1 What is a software framework?

Here is part of what Wikipedia has to say about a software framework:

A software framework, in computer programming, is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality. Frameworks are a special case of software libraries in that they are reusable abstractions of code wrapped in a well-defined API, yet they contain some key distinguishing features that separate them from normal libraries.

Software frameworks have these distinguishing features that separate them from libraries or normal user applications:

1. **inversion of control** - In a framework, unlike in libraries or normal user applications, the overall program’s flow of control is not dictated by the caller, but by the framework.
2. **default behavior** - A framework has a default behavior. This default behavior must actually be some useful behavior and not a series of no-ops.
3. **extensibility** - A framework can be extended by the user usually by selective overriding or specialized by user code providing specific functionality.
4. **non-modifiable framework code** - The framework code, in general, is not allowed to be modified. Users can extend the framework, but not modify its code.

In short, a framework is a computer program that helps you to write computer programs. The description given above is a good match for the XNA framework.

\(^3\)http://en.wikipedia.org/wiki/Software_Framework

http://cnx.org/content/m49509/1.2/
3.2 New project choices in Visual C#

Up to this point in this collection of modules, all of the programs that I have explained have been Visual C# Console Applications.

Assuming that you have Visual C# 2010 and XNA Game Studio 4.0 Refresh installed on your computer, if you start Visual C# and select New Project from the File menu, you have a choice of about a dozen different kinds of projects that you can create within Visual C#. One of those choices is to create a Console Application.

3.3 A Console Application

When you create a Console Application, a project tree is created on your disk containing numerous folders and files. One of those files is a C# source code file named Program.cs. This file, which is the skeleton of a new class named Program, is opened in the editor pane in Visual C#.

The skeleton code in the file looks something like Listing 1 (p. 6) when it first opens.

Listing 1. Initial contents of the file named Program.cs for a Console Application.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

3.3.1 Start debugging

If you select Start Debugging on the Debug menu, the program will run and terminate almost immediately. At that point, the project tree will have expanded to contain more files and folders even though the program hasn't yet done anything useful.

3.3.2 The Main method

As you have learned in earlier modules, every C# program must have a Main method. Program execution begins and ends in the Main method. When control is in the Main method and it no longer has any code to execute, the program terminates. The Main method in Listing 1 (p. 6) is empty, which explains why the program runs and terminates almost immediately.

3.3.3 The starting point for a Console Application

This is the starting point for creating a Console Application in Visual C#. To cause your program to exhibit some useful behavior, you may add code inside the Main method, add new methods to the Program class, define new classes, instantiate objects, call methods on those objects etc.

If you have studied the earlier modules in this collection, this will not be new information for you.

http://cnx.org/content/m49509/1.2/
3.4 A Windows Game Application

Another choice that you have when creating a new project is a **Windows Game (4.0)** application.

Once again, a project tree is created on your disk but it contains more folders and files than the tree that is created for a console application. (Some of the files are initially empty.)

3.4.1 Source code files

As before, there is a C# source code file named `Program.cs` along with another C# source code file named `Game1.cs`. Unlike before, however, the file named `Program.cs` is not opened in the editor pane of Visual C#. Instead, the file named `Game1.cs` is opened in the editor pane.

3.4.2 Build and debug

If you debug this program at this point, it does not run and terminate immediately like a console application. Instead it runs and displays a game window like the one shown in Figure 1 (p. 7) (except that it is quite a bit larger).

**Figure 1**. Initial game window.

The game window will remain on your computer screen until you terminate the program by clicking the X in the upper-right corner or terminate it in some other way.
3.4.3 The file named Program.cs

The file named Program.cs doesn’t automatically open in the editor for one simple reason. The creators of XNA didn’t intend for us to modify it. However, it will be instructive for us to take a look at it anyway.

The source code contained in the file named Program.cs is shown in Listing 2 (p. 8).

Listing 2. The file named Program.cs for a Windows Game project.

```csharp
using System;

namespace WindowsGame2
{
    #if WINDOWS || XBOX
        static class Program
        {
            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            static void Main(string[] args)
            {
                using (Game1 game = new Game1())
                {
                    game.Run();
                }
            }
        }#endif
    }
}
```

3.4.3.1 Instantiate a Game1 object and call the Run method

The code in Listing 2 (p. 8):

- Instantiates a new object of a class named Game1.
- Saves the object’s reference in a reference variable named game.
- Calls the Run method on the reference to the Game1 object.

This code is inside the Main method in the file named Program.cs. The Main method runs when the program is started. The call to the Run method starts the game portion of the program running.

3.4.3.2 What do we know about the class named Game1?

We don’t know anything about the class named Game1 yet, but we will learn about it shortly. Before we get to that, however, I need to explain some unusual syntax in Listing 2 (p. 8).

3.4.3.3 The using directive

You learned earlier that one of the benefits of the using directive is to eliminate the requirement to always type the namespace (such as System) when referring to a class that belongs to that namespace. That is the purpose of the using directive at the top of Listing 2 (p. 8).

However, there is another benefit that derives from the using directive that may be more important.

http://cnx.org/content/m40500/1.2/
3.4.3.4 Memory management

One of the big issues in game programming (or any kind of programming that makes use of graphics files, sound files, or other large resource files) is to make certain that the memory occupied by those resources is freed up as soon as the resource is no longer needed.

Without going into a lot of detail, the use of the using keyword inside the Main method in Listing 2 (p. 8) will assure that the Dispose method is called to free up all of the memory occupied by the Game1 object when control reaches the closing curly brace following the using keyword.

3.4.3.5 Don't modify the file named Program.cs

As an XNA game programmer, you shouldn't normally have any reason to modify the contents of the file named Program.cs. We need to leave it alone and modify the file named Game1.cs instead.

3.4.4 The Game1 class

I told you earlier that when you create a new Windows Game project, the file named Game1.cs is opened in the editor pane. Listing 3 (p. 9) shows the contents of that file.

Listing 3. Initial contents of the file named Game1.cs.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace WindowsGame2
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        /// <summary>
        /// Allows the game to perform any initialization
        /// it needs to before starting to run.
        ```
/// This is where it can query for any required
/// services and load any non-graphic
/// related content. Calling base.Initialize
/// will enumerate through any components
/// and initialize them as well.
/// </summary>
protected override void Initialize()
{
    // TODO: Add your initialization logic here

    base.Initialize();
}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // TODO: use this.Content to load your game content here
}

/// <summary>
/// UnloadContent will be called once per game and is the place to unload
/// all content.
/// </summary>
protected override void UnloadContent()
{
    // TODO: Unload any non ContentManager content here
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
        this.Exit();

    // TODO: Add your update logic here

    base.Update(gameTime);
}
```csharp
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    // TODO: Add your drawing code here
    base.Draw(gameTime);
}
```
3.4.5.6 The game loop

Once the game is initialized, the Run method, or some method called by the Run method ping-pongs back and forth between calls to the overridden Update method and the overridden Draw method. (Note, however that the two methods don’t necessarily take turns executing.)

3.4.5.6.1 Override Update for game logic

You override the Update method to create the program logic associated with game play. To accomplish this, you will likely need to define other methods, define other classes, instantiate objects of other classes, call methods on those objects, test the keyboard, test the mouse, etc. In other words, at this point you need to know how to program in C#.

3.4.5.6.2 Override the Draw method

You override the Draw method to cause the various graphics objects in your game to be rendered in the game window shown in Figure 1 (p. 7).

This module includes an explanation of a very simple program that displays a green arrow sprite near the upper-left corner of the game window (see Figure 3 (p. 12)).

4 Preview

I will create a simple Windows game application that imports the image shown in Figure 2 (p. 12). Note that this is a rectangular image with an (almost) transparent background. (The values of the alpha bytes outside the blue elliptical shape are about 5.)

Figure 2. Raw image with an (almost) transparent background.

If you would like to replicate my program using this image, you should be able to right-click on the image in Figure 2 (p. 12), download it, and save it on your computer. You can save it under any name you choose but the file name extension should be .png.

4.1 Program output

The program displays the image near the upper-left corner of the game window and honors the transparency of the background as shown in Figure 3 (p. 12).

Figure 3. Cropped upper-left corner of the game window.

http://cnx.org/content/m49509/1.2/
4.2 What if you don’t honor the transparent background?

Figure 4 (p. 13) shows the result of causing the alpha transparency value to be ignored and allowing the pixels that are almost transparent in Figure 3 (p. 12) to be opaque.

Figure 4. Cropped upper-left corner of the game window without honoring alpha transparency.

Honoring alpha transparency is the default in XNA 4.0. Figure 4 (p. 13) was created by setting the Premultiply Alpha property (see Figure 7 (p. 17) ) of the image named gorightarrow.png to a value of False and then re-running the program.
4.3 Not a very exciting program

This program isn’t very exciting because there is no motion and no sound. The program simply draws the same image in the same position during every iteration of the game loop. Despite that, this program will give us the opportunity to drill down into the technical aspects of several areas of the XNA framework.

5 Discussion and sample code

5.1 Creating a new Windows Game project

Before getting into the details of the code, I’m going to walk you through the steps involved in creating this Windows Game project using XNA.

5.1.1 Step 1: Create a new Windows Game project named XNA0118Proj

Pull down the Visual C# File menu and select New Project as shown in Figure 5 (p. 14).

![Figure 5](http://cnx.org/content/m49509/1.2/)
5.1.1.1 Select a Windows Game project

Select XNA Game Studio 4.0 in the left pane of the New Project dialog. Select the Windows Game (4.0) icon and enter the name of your project in the Name field. Enter the storage location in the Location field and click the OK button that is off screen further to the right in Figure 6 (p. 15).

Figure 6. Select a Windows Game project.
5.1.2 Step 2: Add your image file to the Content folder

If necessary, pull down the View menu and select Other Windows. Then select Solution Explorer and/or Properties Window so that they are exposed on the right side of the IDE as shown in Figure 7 (p. 17). (Note that the initial view of the Properties window is different from that shown in Figure 7 (p. 17).)

Figure 7. Solutions explorer and properties window exposed.
5.1.2.1 Add your image to the Content folder

Assuming that your project is named XNA0118proj, right click on the XNA0118projContent (Content) folder in the Solution Explorer. Select Add/Existing Item in the dialog that follows. Browse to the image file that you are going to use and click the Add button. A copy of the image file should appear in the Content folder.

5.1.2.2 The Asset Name

Click the image file name in the Content folder and the information in the Properties Window should change to describe that file. Note the value of the Asset Name property in the Properties Window. You will need it later. (In this example, the value of the Asset Name in Figure 7 (p. 17) is gorightarrow.)

5.1.3 Steps 3, 4, and 5: Write code

The next three steps involve writing code to upgrade the skeleton version of the class definition of the Game1 class. I will explain that new code later. For now, the three steps for this example program are:

- Declare two instance variables named myTexture and spritePosition.
- Add a statement to the LoadContent method to load the image.
- Add statements to the Draw method to cause the image to be drawn in the game window.

5.1.3.1 Modify two overridden methods

As you saw in Listing 3 (p. 9), there are five overridden methods in the class definition for the Game1 class that you can modify to customize the class for your game. This program modifies only two of those methods:

- LoadContent
- Draw

I will discuss those two methods along with some other material in this module. I will defer a detailed discussion of the other three methods until a future module when I write a program that modifies them.

5.1.3.2 Will discuss in fragments

A complete listing of the modified definition of the Game1 class is provided in Listing 10 (p. 32) near the end of the module. I will explain selected code fragments in the following paragraphs.

5.2 Beginning of the class named Game1

The beginning of the class definition for the class named Game1 is shown in Listing 4 (p. 19).

Listing 4. Beginning of the class named Game1.

    namespace XNA0118Proj{
        public class Game1 : Microsoft.Xna.Framework.Game{

5.2.1 The namespace

This class definition belongs to the XNA0118Proj namespace. This is the name of the folder containing all of the other folders and files in the project tree as shown in the Solution Explorer in Figure 7 (p. 17).
5.3 General information

5.3.1 The superclass named Game

The new class named Game1 extends the existing class named Game. You will find the documentation for the class named Game here. The description of this class in the documentation is fairly modest. It says simply

"Provides basic graphics device initialization, game logic, and rendering code."

5.3.2 Overridden methods

As I explained earlier, in order to write a program that runs under the XNA framework, you need to override some or all of five methods that are inherited into the Game1 class from the Game class. Let’s see some of what the documentation for the Game class has to say about these methods.

- **Initialize** - Called after the Game and GraphicsDevice are created, but before LoadContent. Override this method to query for any required services, and load any non-graphics resources. Use LoadContent to load graphics resources.
- **LoadContent** - Called when graphics resources need to be loaded. Override this method to load any game-specific graphics resources. This method is called by Initialize. Also, it is called any time the game content needs to be reloaded, such as when the DeviceReset event occurs.
- **UnloadContent** - Called when graphics resources need to be unloaded. Override this method to unload any game-specific graphics resources.
- **Update** - Called when the game has determined that game logic needs to be processed. This might include the management of the game state, the processing of user input, or the updating of simulation data. Override this method with game-specific logic.
- **Draw** - Called when the game determines it is time to draw a frame. Override this method with game-specific rendering code.

5.3.3 The game loop

According to the documentation for the Game class,

"Update and Draw are called at different rates depending on whether IsFixedTimeStep is true or false.

If IsFixedTimeStep is false, Update and Draw will be called in a continuous loop.

If IsFixedTimeStep is true, Update will be called at the interval specified in TargetElapsedTime, while Draw will only be called if an Update is not due.

If Draw is not called, IsRunningSlowly will be set to true.

For more information on fixed-step and variable-step game loops, see Application Model Overview."  

5.3.4 The big picture

Sifting through all of this detail in an attempt to get a big picture view, we see that we should:

- Override Initialize for any special initialization and for loading any non-graphic resources. For example, sound files are non-graphic resources.
- Override LoadContent to load all graphic resources.
- Override UnloadContent if any graphic resources need to be unloaded.
- Override Update to implement all of the game logic.

---

• Override `Draw` to draw an individual frame based on values created and stored by the overridden `Update` method, such as the current position of a sprite.

• Be aware that there are two different approaches to controlling the timing of the game loop, depending on whether the `IsFixedTimeStep` property of the `Game` object is true or false. The default value is true, meaning that the game will attempt to call the `Update` method on a fixed time interval even if that means that the `Draw` method doesn't get called during some iterations of the game loop.

5.4 The constructor for the Game1 class

Listing 5 (p. 21) shows the declaration of two instance variables followed by the constructor for the `Game1` class. One of the instance variables is used in the constructor and the other is used later in the program.

Listing 5 . Constructor for the Game1 class.

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;

public Game1() {
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
} // end constructor
```

5.4.1 What is a constructor?

A constructor is a special method-like structure that is executed once and only once during the instantiation of an object.

The first statement in the `Main` method in Listing 2 (p. 8) uses the `new` operator to cause the constructor to be executed. When the constructor completes its task, it returns a reference to the object just constructed. That reference is stored in the local reference variable of type `Game1` named `game` in Listing 2 (p. 8).

5.4.2 A new GraphicsDeviceManager object

The first statement in the constructor in Listing 5 (p. 21) instantiates a new object of the class `GraphicsDeviceManager` and stores that object’s reference in the instance variable named `graphics`.

The documentation for `GraphicsDeviceManager` isn’t very descriptive. Here is some of what Aaron Reed (the author of the `Learning XNA` books from O’Reilly) has to say on the topic.

"This (GraphicsDeviceManager) is a very important object because it provides you, as a developer, with a way to access the graphics device on your ... The GraphicsDeviceManager object has a property called GraphicsDevice that represents the actual graphics device on your machine."

He goes on to explain how the `GraphicsDevice` object acts as a conduit between your XNA program and the physical graphics device on your machine.

---


http://cnx.org/content/m49509/1.2/
5.4.3 Passing the \this\ keyword as a parameter

Note the parameter that is passed to the \GraphicsDeviceManager\ constructor in Listing 5 (p. 21). The documentation tells us that the parameter must be of type \Game\ and is the \Game\ that the \GraphicsDeviceManager\ should be associated with.

I don’t recall having discussed the keyword \this\ earlier in this collection of modules. According to Jesse Liberty (Programming C# 9 from O’Reilly)

the keyword \this\ is a variable that references the current instance of a class or struct.

Any time that the code in an instance method needs a reference to the object to which the method belongs, \this\ is available as a reference to that object.

In this case, the code in Listing 5 (p. 21) says to instantiate a new \GraphicsDeviceManager\ object and to associate it with \this\ \Game\ object.

5.4.4 The \ContentManager\  

A \Game\ object has a property named \Content\ that references an object of the class \ContentManager\ 10.

According to the documentation, the \ContentManager\:

"is the run-time component which loads managed objects from the binary files produced by the design time content pipeline. It also manages the lifespan of the loaded objects..."

The second statement in the constructor in Listing 5 (p. 21) notifies the \ContentManager\ that the folder named "\Content" is the root of a directory tree in which content for the game will be stored. In this program, we only have one item of content and it is the image file that was added to the \Content\ folder earlier.

5.4.5 When the constructor in Listing 5 terminates...

When the constructor terminates, the new \Game1\ object occupies memory and a reference to the object is stored in the variable named \game\ in Listing 2 (p. 8). The code in the \Main\ method in Listing 2 (p. 8) immediately calls the \Run\ method on the \Game1\ object’s reference.

The \Game1\ class neither defines nor overrides a method named \Run\. However, it does inherit a method named \Run\ 11 from the \Game\ class. Therefore, the method named \Run\ that is defined in the \Game\ class is executed.

5.5 The \Run\ method of the \Game\ class

Here is what the documentation has to say about this method.

"Call this method to initialize the game, begin running the game loop, and start processing events for the game.

This method calls the game \Initialize\ and \BeginRun\ methods before it begins the game loop and starts processing events for the game."

9\http://oreilly.com/catalog/9780596001179
5.5.1 The BeginRun method

We already know about the Initialize method. Here is what the documentation has to say about the BeginRun method.

"Called after all components are initialized but before the first update in the game loop."

5.6 The game loop

At this point, after the Initialize and the LoadContent methods have been called, either the Run method or the BeginRun method, or perhaps some other method that is called by one of those methods goes into a loop calling the Update method and the Draw method.

The timing and the order in which the two methods are called is determined by the value of IsFixedTimeStep as explained earlier.

But, we're getting ahead of ourselves. We need to slow down and discuss the overridden LoadContent method.

5.7 The overridden LoadContent method

The overridden LoadContent method is shown in its entirety in Listing 6 (p. 23) along with the declaration of two instance variables.

Listing 6 . The overridden LoadContent method.

```csharp
//Declare two variables
Texture2D myTexture;
Vector2 spritePosition = new Vector2(10.0f,15.0f);

protected override void LoadContent() {
    // Create a new SpriteBatch, which can be used
    // to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);
    //Load the image
    myTexture = Content.Load<Texture2D>("gorightarrow");
} //end LoadContent
```

5.7.1 Two new instance variables

Listing 6 (p. 23) begins by declaring two new instance variables of types Texture2D and Vector2 named myTexture and spritePosition respectively.

The variable named myTexture will be used in the LoadContent method of Listing 6 (p. 23) to store a reference to a Texture2D object created from the image file with the Asset Name of gorightarrow (see Figure 7 (p. 17)). It will also be used later in the overridden Draw method where it is the sprite being drawn.

The variable named spritePosition will be used later in the overridden Draw method to specify the location to draw the sprite. I will have more to say about this variable later.

---

[^13]: [http://cnx.org/content/m49509/latest/XNA0118revised.html#Update](http://cnx.org/content/m49509/latest/XNA0118revised.html#Update)
5.7.2 The Texture2D class

Here is some of what the documentation\(^\text{14}\) has to say about the Texture\(_{2D}\) class.

"Represents a 2D grid of texels ."

A texel represents the smallest unit of a texture that can be read from or written to by the GPU (Graphics Processing Unit). A texel is composed of 1 to 4 components. Specifically, a texel may be any one of the available texture formats represented in the SurfaceFormat enumeration.

A Texture\(_{2D}\) resource contains a 2D grid of texels. Each texel is addressable by a u, v vector. Since it is a texture resource, it may contain mipmap levels."

You can view a diagram of a texture resource containing a single 3x5 texture with three mipmap levels here\(^\text{15}\). You can read what Wikipedia has to say about mipmaps here\(^\text{16}\). The image used in this program doesn't have any mipmaps.

5.7.3 The SurfaceFormat enumeration

The SurfaceFormat enumeration defines numeric values representing about 50 different types of surface formats such as Rgba\(_{32}\), which is defined as

"(Unsigned format) 32-bit RGBA pixel format with alpha, using 8 bits per channel."

5.7.4 The Texture Format of my image

For example, the Properties Window in Figure 7 (p. 17) shows the image that I used for my program to have a Texture Format property value of Color . The definition of the SurfaceFormat enumeration for Color is

"(Unsigned format) 32-bit ARGB pixel format with alpha, using 8 bits per channel."

Note that this is similar to Rgba\(_{32}\) except that the position of the alpha byte relative to the other three bytes is different.

5.8 A new SpriteBatch object

The code in the overridden LoadContent method of Listing 6 (p. 23) begins by instantiating a new SpriteBatch object and saving its reference in the reference variable named spriteBatch . (That variable is declared at the top of Listing 3 (p. 9).)

Note that the statement that instantiates the SpriteBatch object in Listing 6 (p. 23) is already in the skeleton of the Game1 class when it first appears in the edit window of the Visual C# IDE. (See Listing 3 (p. 9).)

\(^{16}\)http://en.wikipedia.org/wiki/Mipmap
5.8.1 The SpriteBatch class

According to the documentation, an object of the SpriteBatch class "Enables a group of sprites to be drawn using the same settings."

The constructor for an object of the SpriteBatch class requires an incoming parameter that is a reference to the graphicsDevice of the current platform as type GraphicsDevice.

5.8.1.1 This can be confusing

GraphicsDevice is the name of an XNA class. It is also the name of a property of the Game class that is inherited into the Game1 class. The parameter that is passed to the constructor for the SpriteBatch object in Listing 6 (p. 23) is the inherited property.

5.8.1.2 The inherited property

The inherited property contains a reference to an object of the class GraphicsDevice, which is apparently populated in conjunction with the instantiation of the GraphicsDeviceManager object in the constructor of Listing 5 (p. 21). However, it gets populated, it is a reference to the graphicsDevice on the current platform. This causes the new SpriteBatch object to be aware of the graphicsDevice on the current platform. It will be used in the Draw method later to draw the sprite.

5.8.2 The new code

The last statement in Listing 6 (p. 23) is the new code that I wrote into the overridden LoadContent method. The Game1 class inherits a property of the Game class named Content. This property contains a reference to the current ContentManager object.

Therefore, the new code in Listing 6 (p. 23) calls the Load method on the current ContentManager object.

5.8.3 Generic methods

Some methods in C# are known as generic methods, and the Load method of the ContentManager class is one of them. The documentation describes the Load method as follows:

"Loads an asset that has been processed by the Content Pipeline."

5.8.3.1 Required syntax for the Load method

Figure 8 (p. 25) shows the syntax required for calling the Load method. This syntax was taken from the documentation.

public virtual T Load<T> (string assetName)

5.8.3.2 What do the angle brackets mean?

To call this method, you must replace the T between the angle brackets in Figure 8 (p. 25) with the type of asset to be loaded. According to the documentation,

"Model, Effect, SpriteFont, Texture, Texture2D, Texture3D and TextureCube
are all supported by default by the standard Content Pipeline processor, but additional types may
be loaded by extending the processor."

### 5.8.3.3 Calling the Load method of the current ContentManager

Listing 6 (p. 23) calls the **Load** method, specifying an asset type of **Texture2D**, for the purpose of
loading the content identified in Figure 7 (p. 17) with an **Asset Name** property value of **gorightarrow**.

You will recall that this is the value given to the **Asset Name** property of the image file named
**gorightarrow.png** when it was added to the Content folder earlier in this module.

### 5.8.3.4 Populate the variable named myTexture

The value returned from the **Load** method is assigned to the variable named **myTexture** in Listing 6
(p. 23). It will be used later in the **Draw** method to draw the sprite in the game window as shown in
Figure 3 (p. 12) and Figure 4 (p. 13).

That completes the definition of the overridden **LoadContent** method.

### 5.8.4 The Vector2 structure

Returning to the variable declarations in Listing 6 (p. 23), **Vector2** is a structure (similar to a class with
no inheritance capability) containing two components of type **float** named **X** and **Y**.

In this program, the structure referred to by **spritePosition** in Listing 6 (p. 23) is used to encapsulate
the coordinates of the upper-left corner of the sprite (10.0f, 15.0f) when the sprite is drawn in the game
window as shown in Figure 3 (p. 12) and more obviously in Figure 4 (p. 13).

This variable will also be used later in the overridden **Draw** method.

### 5.9 The overridden Game.Draw method

That brings us to the **Draw** method inherited from the **Game** class, shown near the bottom of Listing
3 (p. 9). According to the documentation,

"Called when the game determines it is time to draw a frame. Override this method with game-
specific rendering code."

Note that significant changes were made to the required contents of the **Game.Draw** method in XNA 4.0
as compared to XNA 3.1.

#### 5.9.1 Game loop timing

As you learned earlier, **Update** and **Draw** are called at different rates depending on whether **IsFixed-TimeStep** is true or false.

If **IsFixedTimeStep** is false, **Update** and **Draw** will be called sequentially as often as possible.

If **IsFixedTimeStep** is true, **Update** will be called at the interval specified in **TargetElapsedTime**,
while **Draw** will continue to be called as often as possible. For more information on fixed-step and
variable-step game loops, see Application Model Overview.

Because this program doesn’t override the **Update** method, it doesn’t matter how often the **Draw**
method is called. Each time it is drawn, the sprite is drawn in the same position as shown in Figure 3 (p.
12) and Figure 4 (p. 13).
5.10 More general information

5.10.1 What is a sprite?
According to the 2D Graphics Overview \(^{25}\),

"Sprites are 2D bitmaps drawn directly on the screen, as opposed to being drawn in 3D space. Sprites are commonly used to display information such as health bars, number of lives, or text such as scores. Some games, especially older games, are composed entirely of sprites."

5.10.2 What is a bitmap?
According to the documentation for the Bitmap \(^{26}\) class,

"A bitmap consists of the pixel data for a graphics image and its attributes. There are many standard formats for saving a bitmap to a file. GDI+ supports the following file formats: BMP, GIF, EXIF, JPG, PNG and TIFF."

5.10.3 What is GDI+?
According to the documentation \(^{27}\),

"Microsoft Windows GDI+ is a class-based API for C/C++ programmers. It enables applications to use graphics and formatted text on both the video display and the printer. Applications based on the Microsoft Win32 API do not access graphics hardware directly. Instead, GDI+ interacts with device drivers on behalf of applications."

5.10.4 What about our image?
Working backwards through the above information, we started with an image file named gorightarrow.png. We manually added the file to the Content folder producing a game asset with an Asset Name of gorightarrow (see Figure 7 (p. 17)).

Then we called the Load method in Listing 6 (p. 23) to load the contents of the file into an object of type Texture2D and saved that object’s reference in the instance variable named myTexture. At that point in the process, we had converted the contents of our image file into a format that can be thought of as a sprite. The variable named myTexture contains a reference to our sprite.

5.11 Beginning of the Game.Draw method

The Game.Draw method begins in Listing 7 (p. 27). I am referring to the method here as Game.Draw to distinguish it from the method named SpriteBatch.Draw, which we will encounter shortly.


```csharp
protected override void Draw(GameTime gameTime) {
    GraphicsDevice.Clear(Color.CornflowerBlue);
```

5.11.1 GameTime information
Each time the Game.Draw method is executed, the incoming parameter contains time information encapsulated in an object of type GameTime. According to the documentation, the GameTime object provides a

"Snapshot of the game timing state expressed in values that can be used by variable-step (real time) or fixed-step (game time) games."

We won't be using this information in this module, so I won't pursue it further here. However, we will need the information in future modules when we write code to cause a sprite to be moved and/or animated.

5.11.2 The call to the GraphicsDevice.Clear method
The call to the GraphicsDevice.Clear method in Listing 7 (p. 27) is contained in the skeleton code for the Game1 class as shown in Listing 3 (p. 9).

The GraphicsDevice class provides overloaded versions of the Clear method. According to the documentation, the version shown in Listing 7 (p. 27)

"Clears the viewport to a specified color."

This version of the Clear method requires a single incoming parameter of type Color.

5.11.3 The Color class
The documentation describes an object of the Color class as follows:

"A Color object stores a 32-bit value that represents a color. The color value contains four, 8-bit components: alpha, red, green, and blue. The first 8 bits (the most significant) contain the alpha component, the next 8 bits contain the red component, the next 8 bits contain the green component, and the next 8 bits (the least significant) contain the blue component. The 32-bit value is stored in a variable of type ARGB."

5.11.3.1 Type ARGB
We learned about the ARGB texture format earlier. Although ARGB is referred to as a type in the above quotation, it is not a class. Rather, it is a type established using a C-style typedef.

5.11.3.2 Constructors, methods, and constants
The Color class provides several overloaded constructors and numerous methods that allow you to perform various operations on a Color object.

One of the constructors allows you to create a Color object that represents the color of your choice by specifying the individual values of the alpha, red, green, and blue color components.

In addition, the class provides many constants that represent different colors, one of which is named CornflowerBlue. This is the background color of the game window shown in Figure 1 (p. 7).

You can create Color objects representing those colors simply by calling out the name of the class and the name of the color as shown by the code in Listing 7 (p. 27).

http://cnx.org/content/m49509/latest/XNA0118revised.html#Text
5.11.4 Code to draw the sprite

Three statements are required to draw one sprite and twelve statements are required to draw ten sprites with the same settings. The sequence consists of a Begin statement, one or more SpriteBatch.Draw statements, and one End statement.

Listing 8 (p. 29) shows the code that is used to draw our sprite once each time the Game.Draw method is called. Note that the SpriteBatch.Draw method is called inside the Game.Draw method.

Listing 8. Draw the sprite.

```csharp
spriteBatch.Begin();
spriteBatch.Draw(
    myTexture, spritePosition, Color.White);
spriteBatch.End();
```

The image that we used to create the sprite is shown in raw form in Figure 2 (p. 12). This is a rectangular image with the pixels outside the blue area having an alpha value of about 0.5.

5.11.4.1 Honor the alpha values

As mentioned earlier, the default case is to honor the alpha values in XNA 4.0. This produces the output image shown in Figure 3 (p. 12).

Setting the Premultiply Alpha property value to False in Figure 7 (p. 17) will cause the alpha value to be ignored. This will produce the output image shown in Figure 4 (p. 13).

5.11.4.2 Ignore the alpha values

As explained earlier, honoring alpha transparency is the default case in XNA 4.0. Figure 4 (p. 13) was created by setting the Premultiply Alpha property (see Figure 7 (p. 17) ) of the image named gorihtarrow.png to a value of False and then re-running the program. This causes even the pixels with the very low alpha values to be opaque as shown in Figure 4 (p. 13).

5.11.4.3 Drawing the sprite(s)

You can draw as many sprites as you need following the call to the Begin method in Listing 8 (p. 29).

Each sprite drawn will be drawn according to the parameters passed to the Begin method.

If you need to draw some sprites with different parameters, call the SpriteBatch.End method and start the sequence over with a new call to the SpriteBatch.Begin method and new parameters.

In this case we only have one sprite to draw. Listing 8 (p. 29) calls the SpriteBatch.Draw method to draw that sprite and then calls the SpriteBatch.End method to end the drawing sequence.

5.11.5 Overloaded Draw methods

There are several overloaded versions of the SpriteBatch.Draw method. According to the documentation

"Adds a sprite to the batch of sprites to be rendered, specifying the texture, screen position, and color tint. Before any calls to Draw, you must call Begin. Once all calls to Draw are complete, call End."

The code in Listing 8 (p. 29) passes three parameters to the Draw method:

• **myTexture** - The sprite texture. (See Listing 6 (p. 23).)
• **spritePosition** - The location, in screen coordinates, where the sprite will be drawn. (See Listing 6 (p. 23).)
• **Color.White** - The color channel modulation to use. (Use Color.White for full color with no tinting.)

### 5.11.6 The SpriteBatch.End method

According to the documentation, this method

"Flushes the sprite batch and restores the device state to how it was before Begin was called. Call End after all calls to Draw are complete."

### 5.12 Call Game.Draw on the superclass

When you instantiate an object from a class that extends another class and overrides a method from the superclass, the new object contains both the original version and the overridden version of the method.

#### 5.12.1 Execute both versions of the overridden method

Often it is desirable or necessary to cause both versions to be executed. The code in Listing 9 (p. 30) shows the syntax used to cause an overridden method to call the original version of the method using the keyword **base**. The keyword **base** is a reference to that portion of the object that represents the properties, events, and methods of the superclass.

**Listing 9**. Call Game.Draw on the superclass.

```csharp
    base.Draw(gameTime);
} //end Draw method
} //End class
} //End namespace
```

#### 5.12.2 A required statement

The statement shown in Listing 9 (p. 30) is already contained in the skeleton code produced by Visual C# (see Listing 3 (p. 9)).

The documentation for the **Game.Draw** method contains the following:

"In classes that derive from **Game**, it is necessary to make these calls:

Call **base.Draw** in **Draw** to enumerate through any graphics components that have been added to **Components**. This method will automatically call the **Initialize** method for every component that has been added to the collection."

We won’t worry about the reason why we must do this at this point. We will simply follow the instructions and make the call.

---


http://cnx.org/content/m49509/1.2/
5.13 The end of the program

That completes the explanation for this program. Because of the simplicity of the program, we had no need to override the following methods (see Listing 3 (p. 9)):

- Initialize
- UnloadContent
- Update

We will develop more complicated programs in future modules and will have a need to override one of more of these methods. I will explain them at that time.

6 Run the program

I encourage you to copy the code from Listing 10 (p. 32). Use that code to create an XNA project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

7 Run my program

Click here[^36] to download a zip file containing my version of the program. Extract the folder named XNA0118Proj from the zip file and save it somewhere on your disk. Start Visual C# 2010 Express and select Open Project... from the File menu. Navigate to the project folder and select the file with the extension of .sln. This should cause the project to open and be ready to run or debug as described in the earlier module titled Getting Started[^37].

8 Summary

In this module, I used a very simple XNA program to teach you many of the details regarding the incorporation of the XNA framework into the object-oriented C# programming language. I also taught you about constructors, the this keyword, the base keyword, and some of the differences between a Console Application and a Windows Game application.

9 Miscellaneous

This section contains a variety of miscellaneous information.

**Housekeeping material**

- Module name: Xna0118-The XNA Framework and the Game Class
- File: Xna0118.htm
- Published: 02/28/14

**Disclaimers:** **Financial**: Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

[^36]: http://cnx.org/content/m49509/latest/XNA0118Proj.zip
[^37]: http://cnx.org/contents/GY804-eYg-SoRzQu
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

10 Complete program listing

A complete listing of the XNA program discussed in this module is provided in Listing 10 (p. 32).

Listing 10. The Game1 class for the project named XNA0118Proj.

```csharp
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace XNA0118Proj{
    public class Game1 : Microsoft.Xna.Framework.Game{
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        protected override void Initialize()
        {
            // TODO: Add your initialization logic here
            base.Initialize();
        }

        // Declare two variables
        Texture2D myTexture;
        Vector2 spritePosition = new Vector2(10.0f, 15.0f);

        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used
```
// to draw textures.
spriteBatch = new SpriteBatch(GraphicsDevice);

// Load the image
myTexture =
    Content.Load<Texture2D>("gorightarrow");
}//end LoadContent

protected override void UnloadContent()
{
    // TODO: Unload any non ContentManager content here
}

protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
        this.Exit();

    // TODO: Add your update logic here
    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();
    spriteBatch.Draw(myTexture, spritePosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}//end Draw method
}//End class
}//End namespace

-http://cnx.org/content/m49509/1.2/-