--=====================================

-- Developing apps for Windows Phone or Xbox

http://create.msdn.com/home/getting\_started

GDD --> Game Design and Development

OO --> Object Oriented

XNA -- is a game framework that Microsoft has provided. It's good to build games for Xbox 360 and windows phone 7

XNA Game Studio 4.0 Refresh

Several years ago, Microsoft released XNA Game Studio Express, a game development framework to make it easier for students and hobbyists to develop games.

Even better, with a membership in App Hub games developed using the framework can be uploaded, played, and even sold on the Xbox 360!

DLL -- dynamic link libraries

--- Microsoft will stop (or stopped) supporting XNA in April 2014 ---

Monogame (Andriod, iOS) framework uses XNA ideas

The .NET Common Language Runtime (CLR) runs all Common Intermediate Language

n = 2^b --> we need "b" bits to represent n different things (to represent 4 outcomes we need 2 bits: 00, 01, 10, 11)

b = log2n

--===========================================

F6 -- to build a code in Visual Studio

F5 -- start application (stops on break points)

Ctrl + F5 -- start an application in the debugging model (ignores all the break points)

File --> Save ALL (Ctrl + Shift + S) to save the solution in the desired location

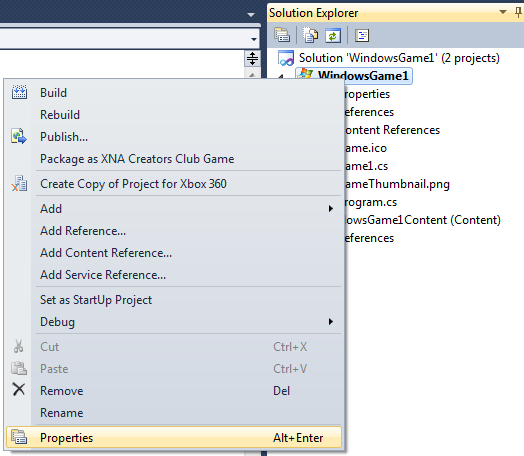
F10 -- debugging with the step over

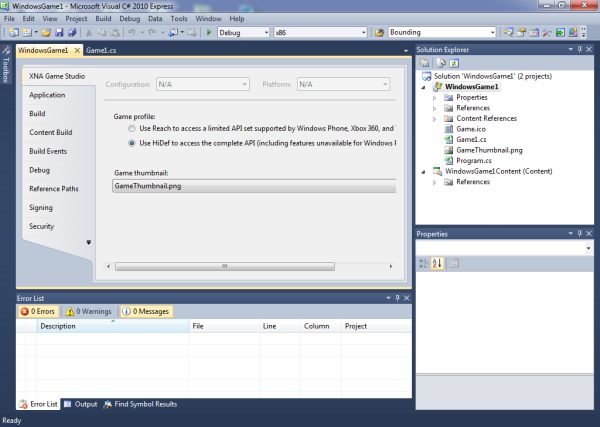
F11 -- debugging with going inside of the module

**--=========================================================================**

When you try to run the program, you may get an error dialog that says you don’t have a suitable graphics card. XNA Game Studio has two different graphics profiles: Reach and HiDef. The Reach profile is designed to reach (get it?) a large number of devices, while the HiDef profile requires a better graphics card and later versions of DirectX. If you got the error dialog, it means your graphics card doesn’t meet the HiDef profile.

The good news is that it’s easy to change this in your XNA solution. Right click the WindowsGame1 project (the top one, not the WindowsGame1Content project on the bottom) in the solution and select Properties at the bottom of the dropdown list. You’ll get the following display (if it doesn’t look like this, click the XNA Game Studio tab at the top left of the pane):





Just click the radio button next to the Use Reach … option, Ctrl-S to save, and F5 to run again. Everything should work fine now. If you have to use Reach here, you’ll need to use that graphics profile for all your XNA games (on that computer). That’s okay, though, because all the games we build together in this book work fine using the Reach graphics profile.

# **WEEK 1**

Everything is stored as 0 and 1

We can toss a coin 2 times and represent 4 different outcomes:

00, 01, 10, 11

N = 2^b

To represent N objects we need 2^b bits

Or Log 2N = b

**Declare a variable = to allocate a memory address for it**

We can optionally assign a value to that variable

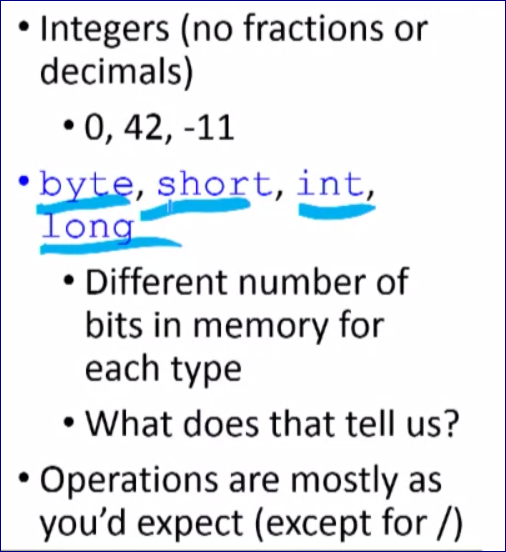
We have to assign a datatype so the Interpreter know how to interpret the bits (whether those 0 and 1s are 65 or A)

**3.3 Integer Data Types (13:34)**

**Integers**

We have 4 data types to represent an integer:

* byte = 1 byte (8 bits) and can represent 2^8 = 256 unique values
* short = 16 bits
* int = 32 bits
* long = 64 bits



If you are in a memory constraint environment and you deal with a small numbers you can choose a smaller datatype (short or byte).

In C#, if we add 1 to an int variable that currently has a value of 2,147,483,647, the new value of the variable is

Answer = -2,147,483,648

2^31 = 2,147,483,648  
this is the largest number that can be represented by the “int” datatype.

When we add 1 it turns to be the highest negative value.

// Declare an integer variable and constant

int totalSecondsPlayed = 100;

const int SECONDS\_PER\_MINUTE = 60;

// Claculate minutes and seconds

int minutes = totalSecondsPlayed / SECONDS\_PER\_MINUTE;

int seconds = totalSecondsPlayed % SECONDS\_PER\_MINUTE; // print a remainder

// Print results

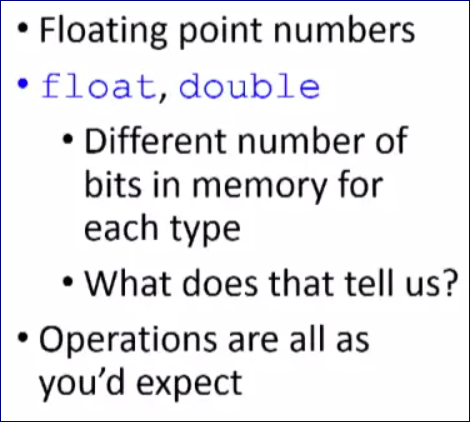
Console.WriteLine("Minutes: " + minutes);

Console.WriteLine("Seconds: " + seconds);

**3.4 Real Number Data Types (11:58)**

float – represents smaller range of number

double – represent bigger range of numbers



int score = 1356;

int totalSecondsPlayed = 10000;

// Calculate and display results

// Temporarily convert "score" variable to float data type

float pointsPerSecond = (float)score / totalSecondsPlayed;

Console.WriteLine("Points Per Second " + pointsPerSecond);

Console.WriteLine();

**DECIMAL**

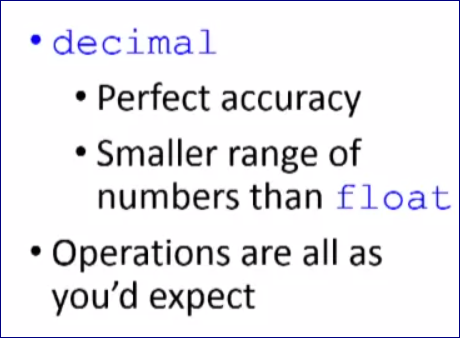
Float does not provide a perfect accuracy.

We have a capability to reach a PERFECT accuracy for float numbers

We just reduce the range of the numbers we can represent.

So we can represent with a perfect precision.

At the cost of the reduced range (though up to 28 decimal places)



**3.5 Other Value Types (5:41)**

**Characters**

**Char --** 16 bits to hold Unicode representation of the character

ASCII has 8 bit representation

**Now we are moving on from ASCII to Unicode.**

Operations as we need them

**Char datatype allows you to store a single character.**

**Booleans**

**bool**

Only two possible values: **true, false**

Operations may be new to you:

* And: &&
* Or: ||
* Not: !

Booleans are very helpful when we make decisions in our code

# **WEEK 2**

**Module 4 -- Classes and Objects**

**The big idea: software can be implemented as a set of interacting objects.**

**Class**

• Template for creating objects (“cookie cutter”)

• Defines the fields, properties, and behavior of every object of the class

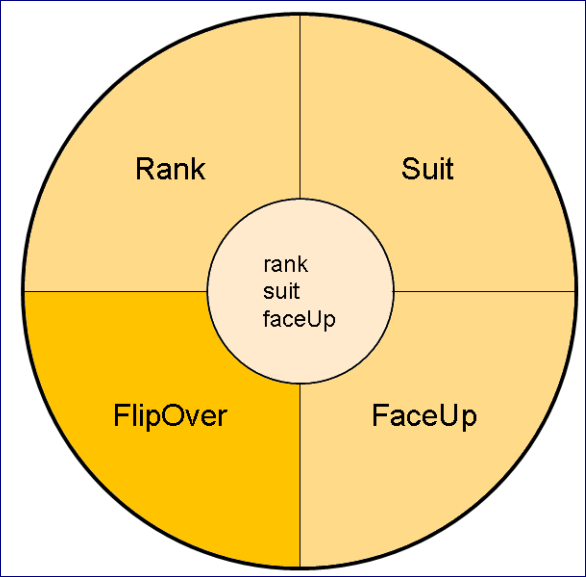
**Object**

• Actual instance of the class in memory

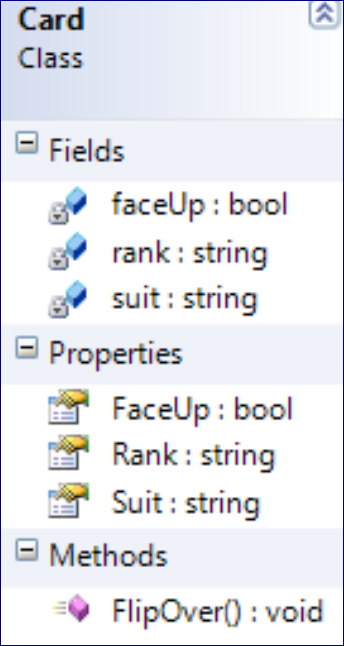
• Each object stores its own state (Different card objects have different ranks and suits, for example)

**Objects have:**

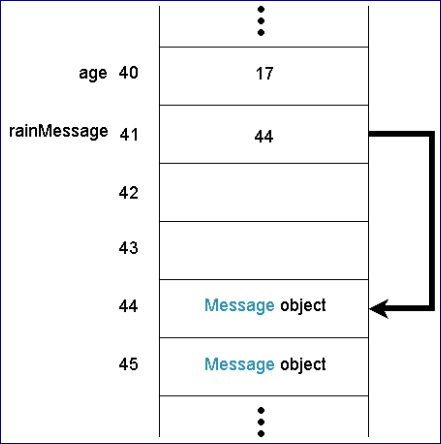
* State -- Characteristics of the object
* Behavior – (1) What we can do to the object, (2) What we can tell the object to do to itself
* Identity – (1) So we can distinguish one object from another, (2) Memory address



**UML – Unified Modeling Language**



Classes are reference types, not value types. The ones and zeros at the memory location allocated to the variable aren’t the value of the variable, they’re a reference to the location in memory where the actual object is.



**Recap**

OO provides classes (reference types in C#) we can use to create objects.

Objects interact with each other to implement our game.

Classes are basically their own datatypes.

To create a new object of the class we need to use a “Constructor”

For example, we have a Class: **Deck()**

**To create an object in the C# we need to do the same as we do creating a new variable:**

1. Identify a data type
2. Give a variable a name
3. (optionally) assign a value to the variable

// Let's say we have a class Deck()

// To create a new object of this class

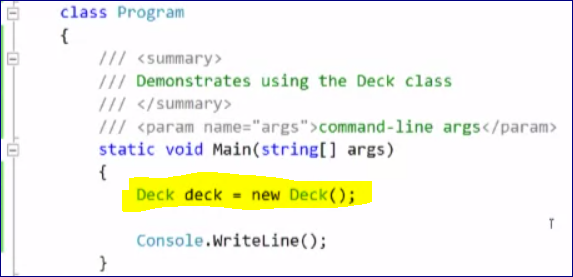
Deck myDeck1 = new Deck();

1. Identify a data type (Deck)
2. Give a variable a name (myDeck1)
3. (optionally) assign a value to the variable (new Deck())

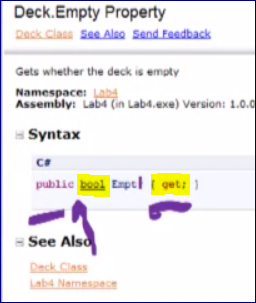
new - is a special command that runs a constructor

The **constructor** will always be the class name

Deck() – class Deck doesn’t require input parameters



Properties of the object



// Let's say we have a class Deck()

// To create a new object of this class

Deck myDeck1 = new Deck();

// Print information about the deck status (full/ empty)

// "Empty" is a property of the Class Deck()

bool isDeckEmpty = myDeck1.Empty;

Console.WriteLine("Empty: " + isDeckEmpty);

// OR we can access the property directly without assigning it to a variable

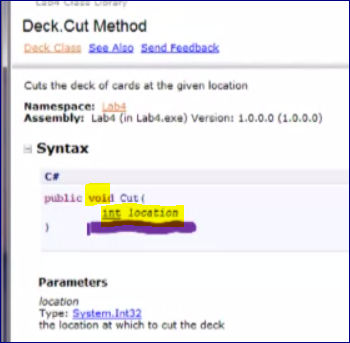
Console.WriteLine("Empty: " + isDeckEmpty.Empty);

**// when accessing a property we NEVER put () parenthesis**

**4.3 Using a Class Methods (15:04)**

// tell the deck to shuffle itself

mydeck1.Shuffle(); // when accessing a method we always put () parenthesis



Void – means that the method doesn’t return anything to us.

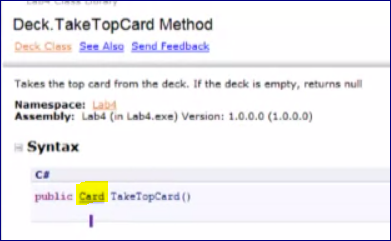
**When we provide information to a method by calling the method, the information between the parentheses in the method call are ARGUMENT.**

We see that an Argument of the Cut() method must be **int**.

// cut the deck

myDeck1.Cut(26);

**Another method 🡪 TakeTopCard()**



This method returns a Card object.

So, we can assign it to a Card object or just run this method without assigning (even though it’s rare).

Card myCard1 = myDeck1.TakeTopCard();

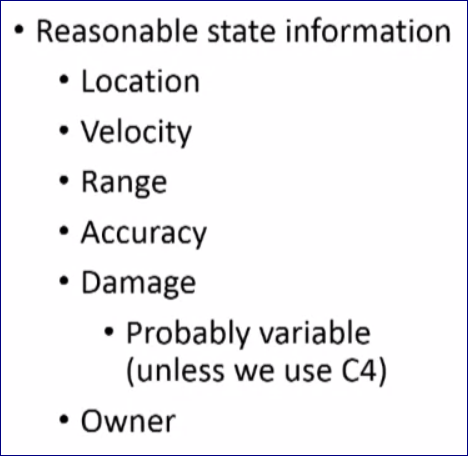
**4.4 Designing a Class (9:03)**

Create a RubberChicken class – it will be our weapon

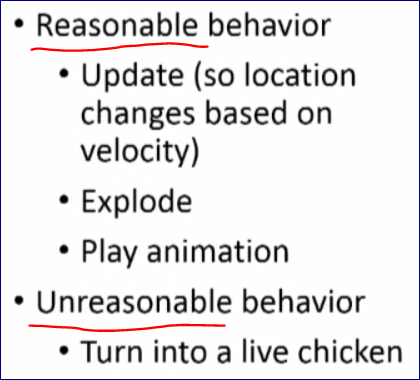
Where it exists

Owner – to track to whom allocate benefits

**State:**



**Behavior:**



**5.1 Intro to XNA (12:02)**

**General things we do in games:**

-- Initialize the game

-- Load content for the game

-- Run the game loop:

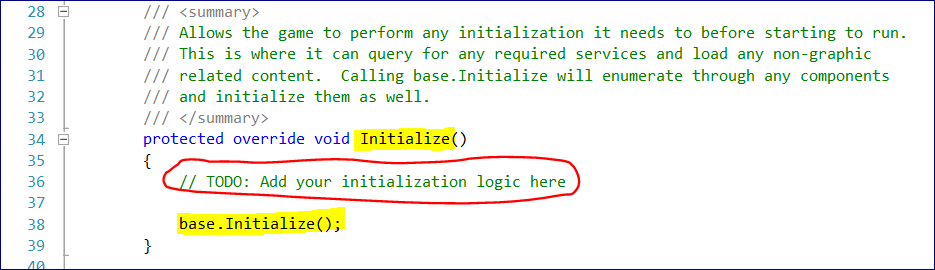
-- Update game world

-- Render (draw) visible part of game world

**Construct Method**

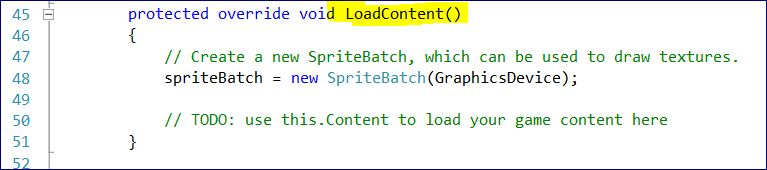


**Initialize Method**



**LoadContent Method**

Content loaded in the **LoadContent** method can include **graphical assets**

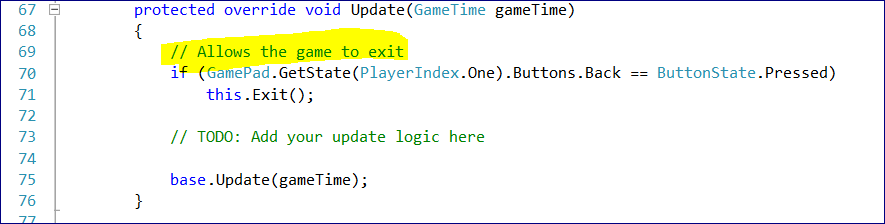


And then XNA enters the game loop

**Then UPDATE method**

This method is called every frame in the game -- *The default frame rate for an XNA game is 60fps.*

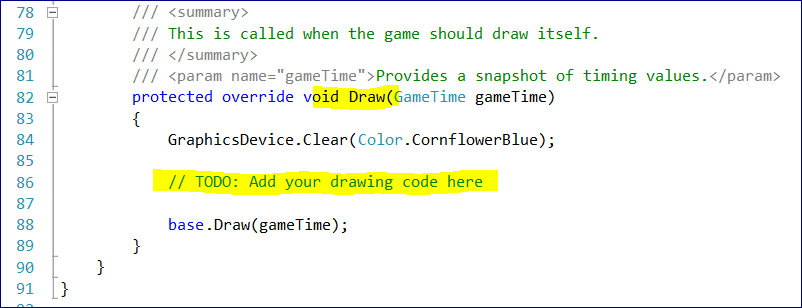
**It also allows to exit the game**



**DRAW method**

This is here we draw our game

It will draw all visible entities in our game



**2D (2 Dimensional) Sprites have 4 components**

1. R-component
2. G-component
3. B-component
4. Alpha-component (opacity)

**5.2 Your First XNA Game Resolution and Loading Content (11:06)**

**MyCode:**

C:\Dim\Google Drive\Coursera -- Beginning Game Programming with C#\week 02\FirstXnaGame

**Load and Draw some sprites**

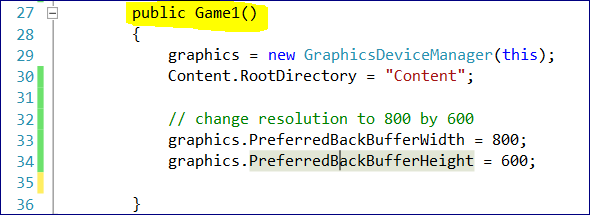
First we set up the resolution of the screen 800x600

We do it after the Game Object was initialized

// change resolution to 800 by 600

graphics.PreferredBackBufferWidth = 800;

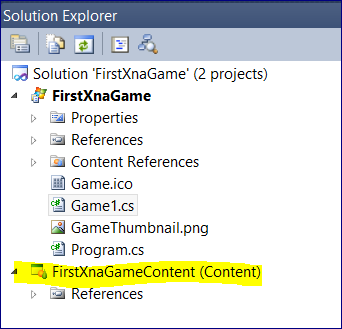
graphics.PreferredBackBufferHeight = 600;



Then create the **Texture2D** objects and **Rectangle** to hold the objects



All the content such as picture is stored in the Content “project”

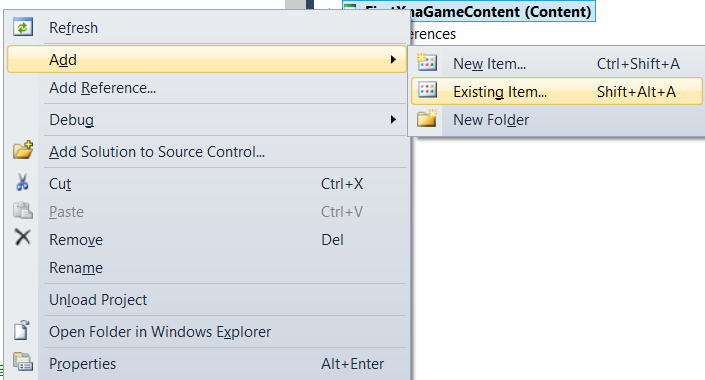


We have .png pictures on the hard drive on

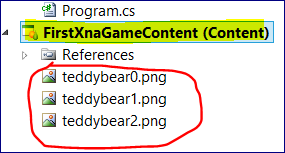
C:\Dim\Google Drive\Coursera -- Beginning Game Programming with C#\week 02\Teddy Bear Pictures PNG



To upload them to the project Right Click on the **FirstXnaGameContent**  🡪 Add 🡪 Existing Item



Now we have our pictures uploaded to the project (Content)



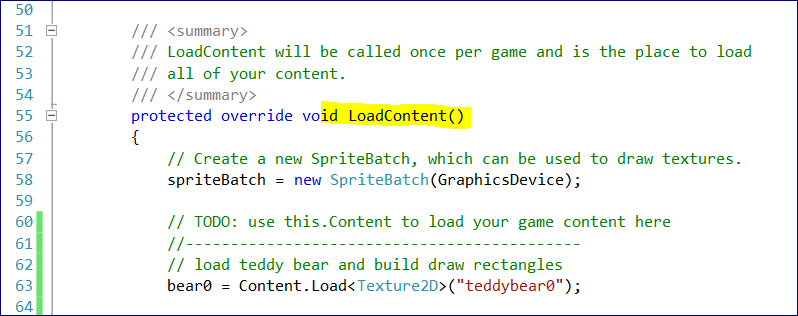
**Now in the LoadContent Method**

// load teddy bear and build draw rectangles

bear0 = Content.Load<Texture2D>("teddybear0");

So we load a picture with the name “teddybear0” to the object “bear0”.

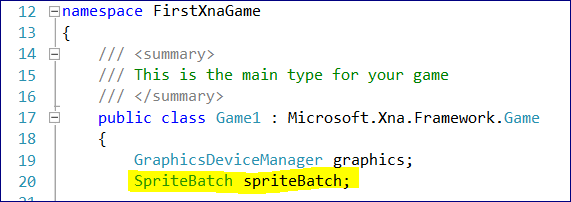
In <> we specify what kind of content we want to load (Texture2D)

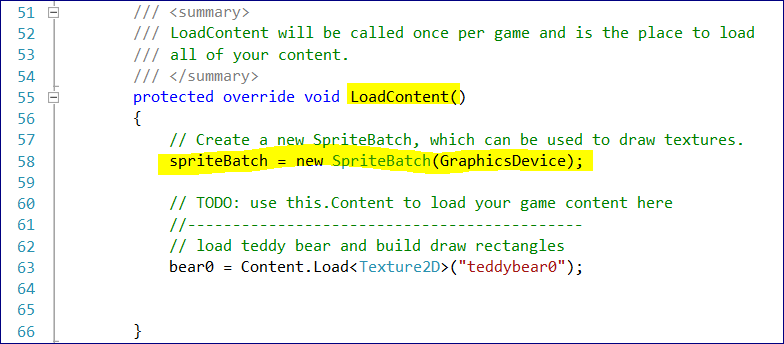


**5.3 Your First XNA Game Drawing (11:28)**

“SpriteBatch” is used to draw stuff.

We initialize the spriteBatch object

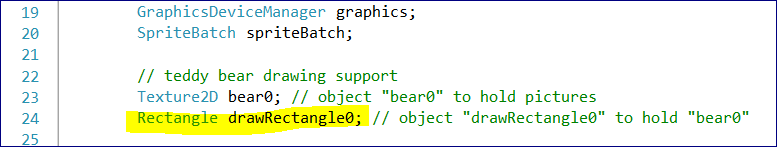




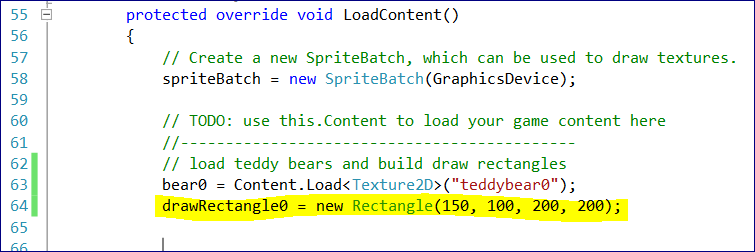
**Sprites are drawn to the screen when the END method of SpriteBatch is called.**

First we need to build a draw rectangle object which is used to draw a picture on the screen.

Earlier we defined a variable drawRectangle0 but we did not initialize it.



Let’s initialize the **drawRectangle0**



We have to provide 4 parameters to create a Rectangle structure:

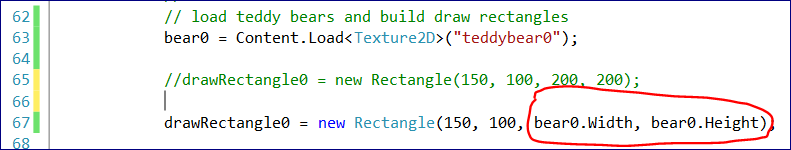
* X
* Y
* Width
* Height

200 by 200 gives us a too pixelated picture.



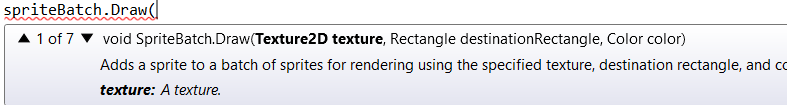
To use the original image size implement

drawRectangle0 = new Rectangle(150, 100, bear0.Width, bear0.Height);



Then we draw the object on the screen using the DRAW method which takes:

* Texture2D object
* Rectangle object
* color



**Summary**

**Define (Initialize) Bears and Rectangles, SpriteBatch, and GraphicDeviceManager**

Define constants for the HEIGHT and WIDTH

namespace FirstXnaGame

{

/// <summary>

/// This is the main type for your game

/// </summary>

public class Game1 : Microsoft.Xna.Framework.Game

{

**GraphicsDeviceManager graphics;**

**SpriteBatch spriteBatch;**

// teddy bear drawing support

Texture2D bear0; // object "bear0" to hold pictures

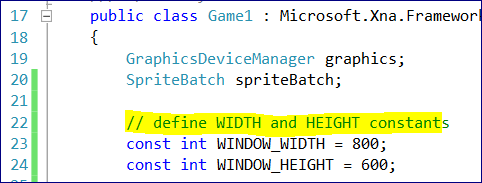
Rectangle drawRectangle0; // object "drawRectangle0" to hold "bear0"

Texture2D bear1;

Rectangle drawRectangle1;

Texture2D bear2;

Rectangle drawRectangle2;



**Set up Screen Resolution**

In the Game1() method

public Game1()

{

graphics = new GraphicsDeviceManager(this);

Content.RootDirectory = "Content";

// change resolution to 800 by 600

graphics.PreferredBackBufferWidth = 800;

graphics.PreferredBackBufferHeight = 600;

}

**LOAD CONTENT AND BUILD RECTANGLES**

**Create a new SpriteBatch, which can be used to draw textures.**

**Load pictures (and other content)**

**Build Rectangles**

protected override void LoadContent()

{

spriteBatch = new SpriteBatch(GraphicsDevice);

bear0 = Content.Load<Texture2D>("teddybear0");

drawRectangle0 = new Rectangle(150, 100, bear0.Width, bear0.Height);

bear1 = Content.Load<Texture2D>("teddybear1");

drawRectangle1 = new Rectangle(300, 100, bear1.Width, bear1.Height);

bear2 = Content.Load<Texture2D>("teddybear2");

drawRectangle2 = new Rectangle(450, 100, bear2.Width, bear2.Height);

}

**DRAW CONTENT**

protected override void Draw(GameTime gameTime)

{

GraphicsDevice.Clear(Color.CornflowerBlue);

// TODO: Add your drawing code here

// draw teddy bears

**spriteBatch.Begin();**

spriteBatch.Draw(bear0, drawRectangle0, Color.White);

spriteBatch.Draw(bear1, drawRectangle1, Color.White);

spriteBatch.Draw(bear2, drawRectangle2, Color.White);

**spriteBatch.End();**

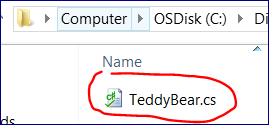
base.Draw(gameTime);

}

**5.4 More XNA Practice Code Zoomed (15:09)**

Let’s add a prebuilt TeddyBears Class

This is a TeddtBears.cs file



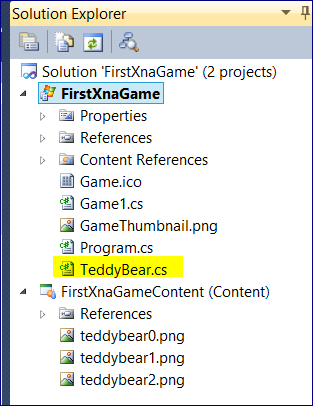
**In the location:**

C:\Dim\Google Drive\Coursera -- Beginning Game Programming with C#\Teddy Bear Class

Right Click on project name (FirstXnaGame) 🡪 Add 🡪 Existing item

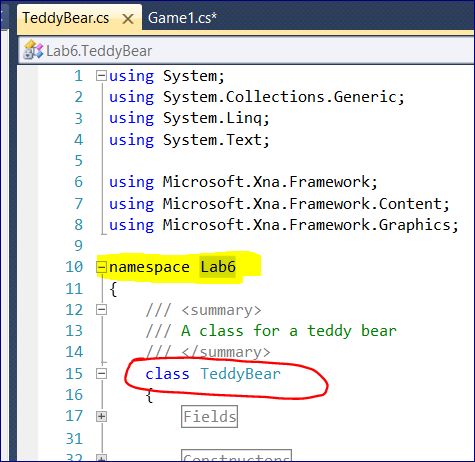


We have added a TeddyBear.cs file to the project.

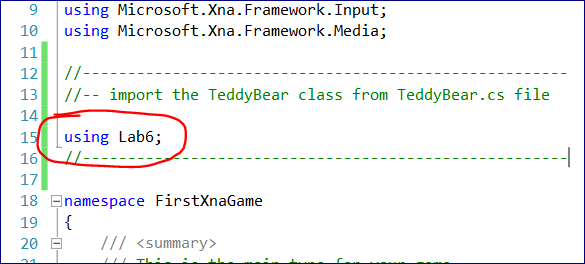


The file TeddyBear.cs contains the class TeddyBear that we are going to use.

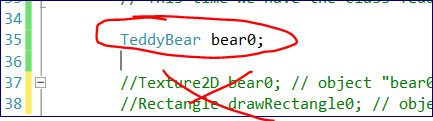
**The namespace for this class is “Lab6”**



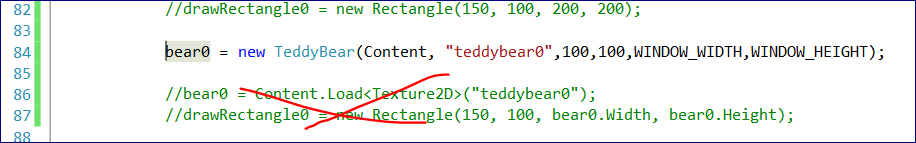
**So, in our program we need to import this module:**



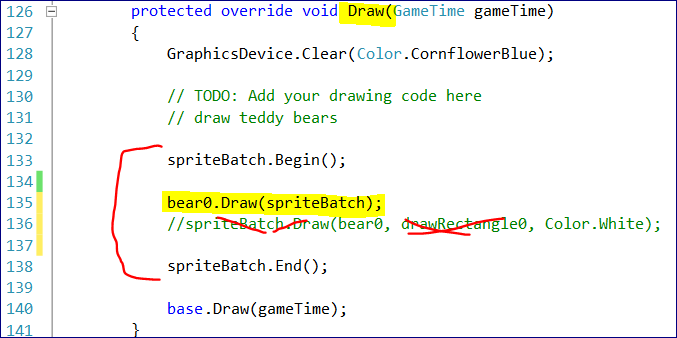
Now we can define “bear0” as a TeddyBear variable type

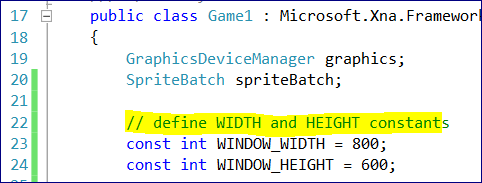


Teddy bears should load their content when they’re instantiated.



Calling the DRAW method





# **WEEK 3**

**6.1 Intro to Strings (14:25)**

// prompt for and read in gamertag

//String is a class, string (with a small "s") is a datatype

**// strings are immutable -- we can change the string object**

Console.Write("Enter gamertag: ");

string gamertag = Console.ReadLine();

// prompt for and read in level

Console.Write("Enter level: ");

**// "Parse" converts a string representation to a number**

int level = int.**Parse**(Console.ReadLine());

// extract the first character of the gamertag

char firstGamertagCharacter = gamertag[0];

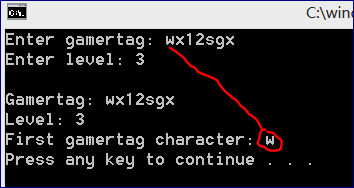
// print out values

Console.WriteLine();

Console.WriteLine("Gamertag: " + gamertag);

Console.WriteLine("Level: " + level);

Console.WriteLine("First gamertag character: " + firstGamertagCharacter);



**6.2 String Operations (11:45)**

**// read in csv string**

Console.Write("Enter name and percent (name, percent): ");

string csvString = Console.ReadLine();

**// find comma location**

**// "IndexOf" returns the index of the first occurence, if the element was not found it returns "-1"**

int commaLocation = csvString.IndexOf(",");

**// extract name and percent**

string name = csvString.Substring(0, commaLocation);

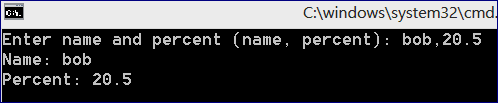
float percent = float.Parse(csvString.Substring(commaLocation + 1));

// print name and percent

Console.WriteLine("Name: " + name);

Console.WriteLine("Percent: " + percent);

Console.WriteLine();



**7.1 Selection If Statements (16:54)**

**// ask for and get user answer**

Console.Write("Pick up the shiny thing? (y, n) ");

char answer = Console.ReadLine()[0];

// print appropriate message

if (answer == 'y' || answer == 'Y')

{

Console.WriteLine("You have the shiny object");

}

else if (answer == 'n' || answer == 'N')

{

Console.WriteLine("You DO NOT have the shiny object");

}

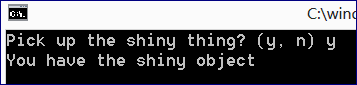
else

{

Console.WriteLine("You are a n00b");

}

Console.WriteLine();



**7.2 Selection Switch Statements (8:17)**

// ask for and get user answer

Console.Write("Pick up the shiny thing? (y, n) ");

char answer = Console.ReadLine()[0];

// print appropriate message

**switch** (answer)

{

case 'y':

case 'Y':

Console.WriteLine("You have a shiny thing");

break;

case 'n':

case 'N':

Console.WriteLine("You DO NOT have a shiny thing");

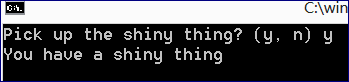
break;

default:

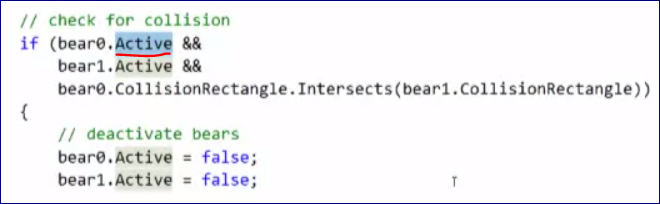
Console.WriteLine("You are a n00b");

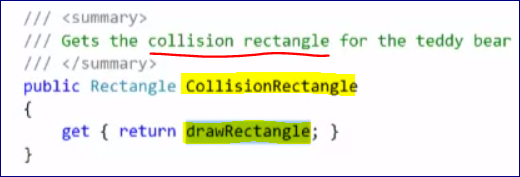
break;

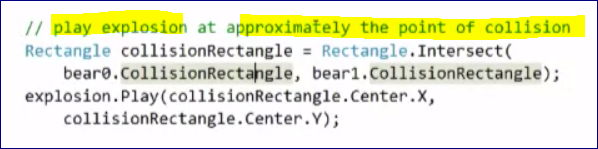
}



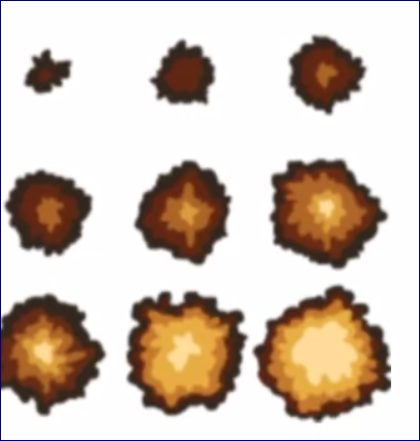
**7.3 Selection in Games (18:22)**

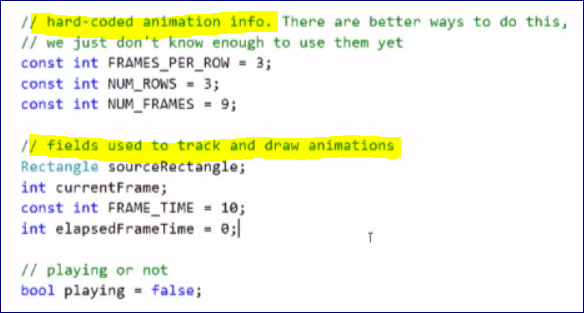


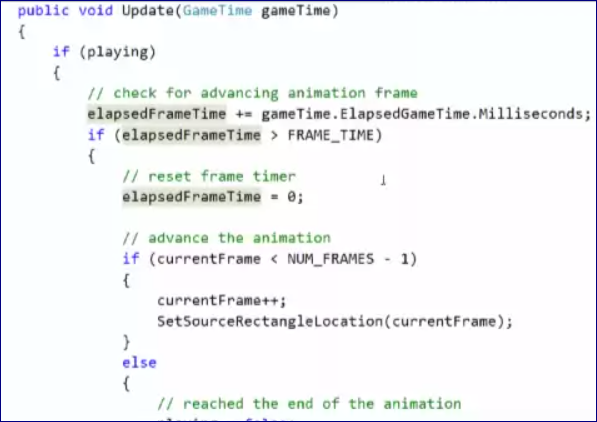




Sprite strip





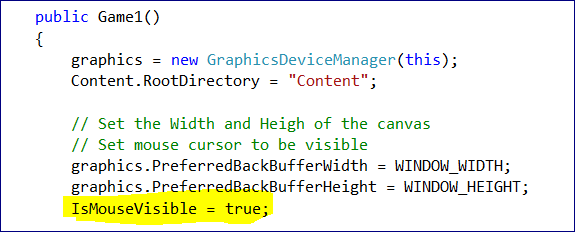


# **WEEK 4**

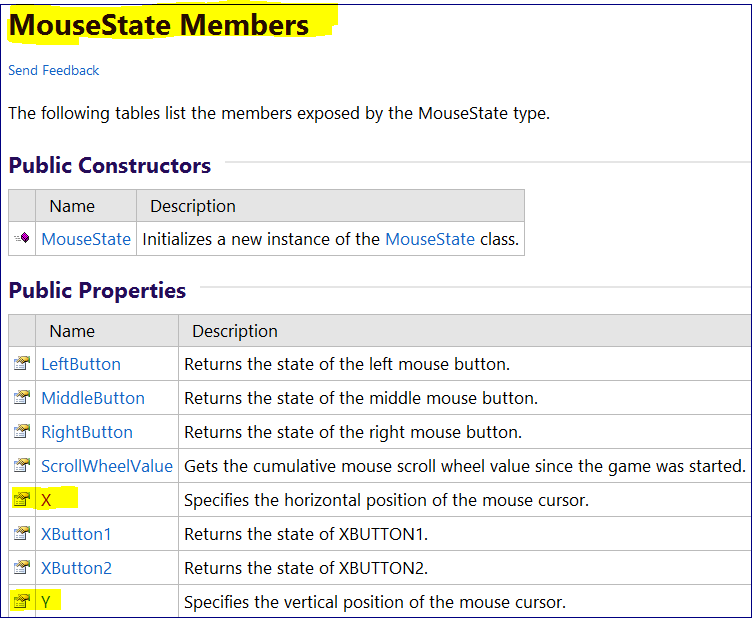
**8.1 XNA Mouse Location Processing (11:38)**

The Mouse GetState method returns the current state of the mouse.

MouseState mouse = Mouse.GetState();



Search for that in HELP (documentation in Visual Studio)



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.Graphics;

using Microsoft.Xna.Framework.Input;

using Microsoft.Xna.Framework.Media;

namespace ProgrammingAssignment2

{

/// <summary>

/// This is the main type for your game

/// </summary>

public class Game1 : Microsoft.Xna.Framework.Game

{

const int WINDOW\_WIDTH = 800;

const int WINDOW\_HEIGHT = 600;

GraphicsDeviceManager graphics;

SpriteBatch spriteBatch;

// STUDENTS: add your sprite variables here

// Create 5 "Texture2D" variables

Texture2D character0;

Texture2D character1;

Texture2D character2;

Texture2D character3;

Texture2D character4;

// used to handle generating random values

Random rand = new Random();

const int CHANGE\_DELAY\_TIME = 1000;

int elapsedTime = 0;

// used to keep track of current sprite and location

Texture2D currentCharacter;

Rectangle drawRectangle = new Rectangle();

public Game1()

{

graphics = new GraphicsDeviceManager(this);

Content.RootDirectory = "Content";

// Set the Width and Heigh of the canvas

// Set mouse cursor to be visible

graphics.PreferredBackBufferWidth = WINDOW\_WIDTH;

graphics.PreferredBackBufferHeight = WINDOW\_HEIGHT;

IsMouseVisible = true;

}

/// <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);

// STUDENTS: load the images here

// Loading the images from the Content folder to the Texture2D variables

character0 = Content.Load<Texture2D>("Tomato\_100x100");

character1 = Content.Load<Texture2D>("Pear\_100x100");

character2 = Content.Load<Texture2D>("Banana\_100x100");

character3 = Content.Load<Texture2D>("Kiwi\_100x100");

character4 = Content.Load<Texture2D>("Cherry\_100x100");

// set currentCharacter to one of the sprite variables

currentCharacter = character0;

// build drawRectangle for the "character0" at the center of the window

drawRectangle.X = WINDOW\_WIDTH / 2 - currentCharacter.Width / 2;

drawRectangle.Y = WINDOW\_HEIGHT / 2 - currentCharacter.Height / 2;

drawRectangle.Height = currentCharacter.Height;

drawRectangle.Width = currentCharacter.Width;

}

/// <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

if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)

this.Exit();

// Make character follow the mouse

MouseState mouse = Mouse.GetState();

drawRectangle.X = mouse.X - currentCharacter.Width / 2;

drawRectangle.Y = mouse.Y - currentCharacter.Height / 2;

// clamp character in window

if (drawRectangle.Left < 0)

{

drawRectangle.X = 0;

}

if (drawRectangle.Right > WINDOW\_WIDTH)

{

drawRectangle.X = WINDOW\_WIDTH - drawRectangle.Width;

}

if (drawRectangle.Top < 0)

{

drawRectangle.Y = 0;

}

if (drawRectangle.Bottom > WINDOW\_HEIGHT)

{

drawRectangle.Y = WINDOW\_HEIGHT - drawRectangle.Height;

}

elapsedTime += gameTime.ElapsedGameTime.Milliseconds;

if (elapsedTime > CHANGE\_DELAY\_TIME)

{

elapsedTime = 0;

// STUDENTS: Generate a random number between 0 and 4

//int spriteNumber = rand.Next(5);

// STUDENTS

// Assign the "currentSprite" variable to a sprite based on the generated random number between 0 and 4

//if (spriteNumber == 0)

//{

// currentCharacter = character0;

//}

//else if (spriteNumber == 1)

//{

// currentCharacter = character1;

//}

//else if (spriteNumber == 2)

//{

// currentCharacter = character2;

//}

//else if (spriteNumber == 3)

//{

// currentCharacter = character3;

//}

//else if (spriteNumber == 4)

//{

// currentCharacter = character4;

//}

// Update X

//drawRectangle.X +=50;

}

base.Update(gameTime);

}

/// <summary>

/// 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);

// STUDENTS: draw "currentSprite" on the canvas

spriteBatch.Begin();

spriteBatch.Draw(currentCharacter, drawRectangle, Color.White);

spriteBatch.End();

base.Draw(gameTime);

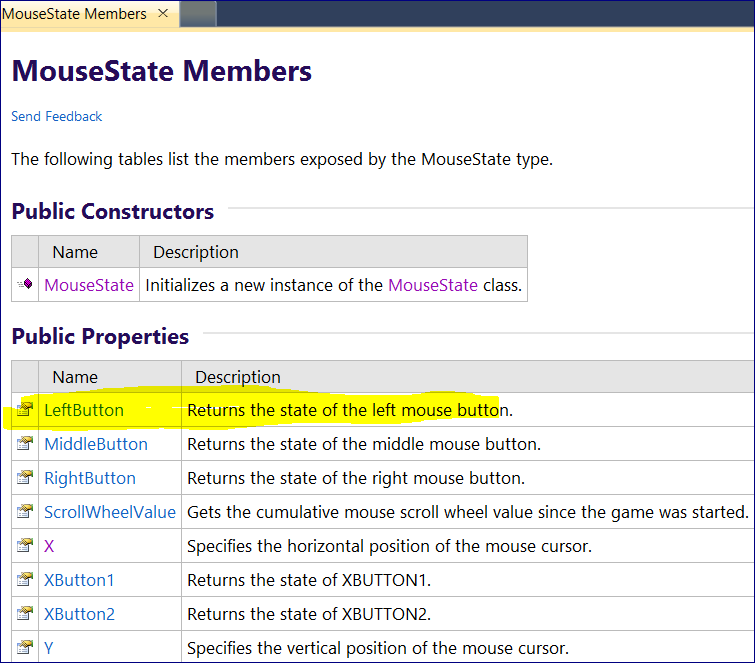
}

}

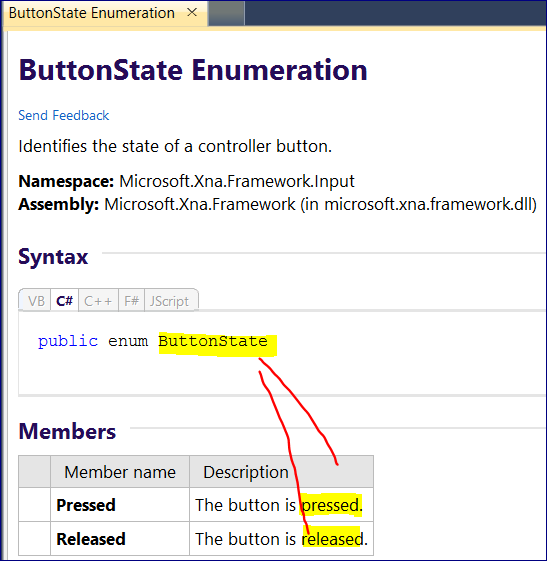
}

**8.2 XNA Mouse Button Processing (14:11)**

A mouse has 5 buttons.



**An enumeration defines a data type with a specific set of values.**



**8.3 XNA Controller Thumbstick Processing (16:18)**

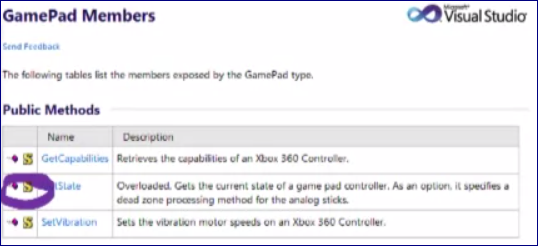
Sometimes we can call a method not on an object but on the class

E.g.

Console.WriteLine();

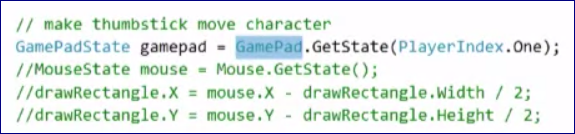
**With static methods.**

Static methods are called using a class name rather than an object name.



“S” in front a particular method stands for “Static”

So we call that method using the class name rather than the instance name.





**8.4 XNA Controller Button Processing (11:00)**

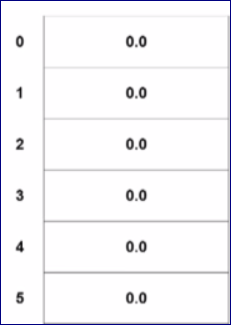
**8.5 XNA Controller Force Feedback (8:13)**

**9.1 Arrays (13:24)**

Arrays are objects.

A particular array can store JUST A SINGLE DATATYPE.

A single array can store many objects of the same type.



The value stored at each location in the array is called an **element**. So, we talk about an array of elements.

The number of each location in the array is called an **index**.

Create an array that can store maximum of 10 integer values.

int[] myArray = new int[10];

// by putting [] you specify an array, in [5] you specify the number of elements you want to store

**Create a Texture2D array:**

Texture2D[] **character** = new Texture2D[5];

We access individual elements of an array using the array name and an index

currentCharacter = **character**[0];

OR

currentCharacter = **character**[rand.Next(5)];

That was a ONE-Dimensional array.

Arrays can have the following number of dimensions: as many as you want

These are examples of MULTI-Dimensional arrays

int[] myArray1 = new int[10]; // one-dimensional array

int[,] myArray2 = new int[5,2]; // two-dimensional array

int[,,] myArray3 = new int[5, 2, 4]; // three-dimensional array

//Using an array to add elements

character[0] = Content.Load<Texture2D>("Tomato\_100x100");

character[1] = Content.Load<Texture2D>("Pear\_100x100");

character[2] = Content.Load<Texture2D>("Banana\_100x100");

character[3] = Content.Load<Texture2D>("Kiwi\_100x100");

character[4] = Content.Load<Texture2D>("Cherry\_100x100");

**9.2 Collection Classes (8:45)**

When we do not know the number of the elements that we need to store.

When the number of elements increases and decreases, so should do the storage.

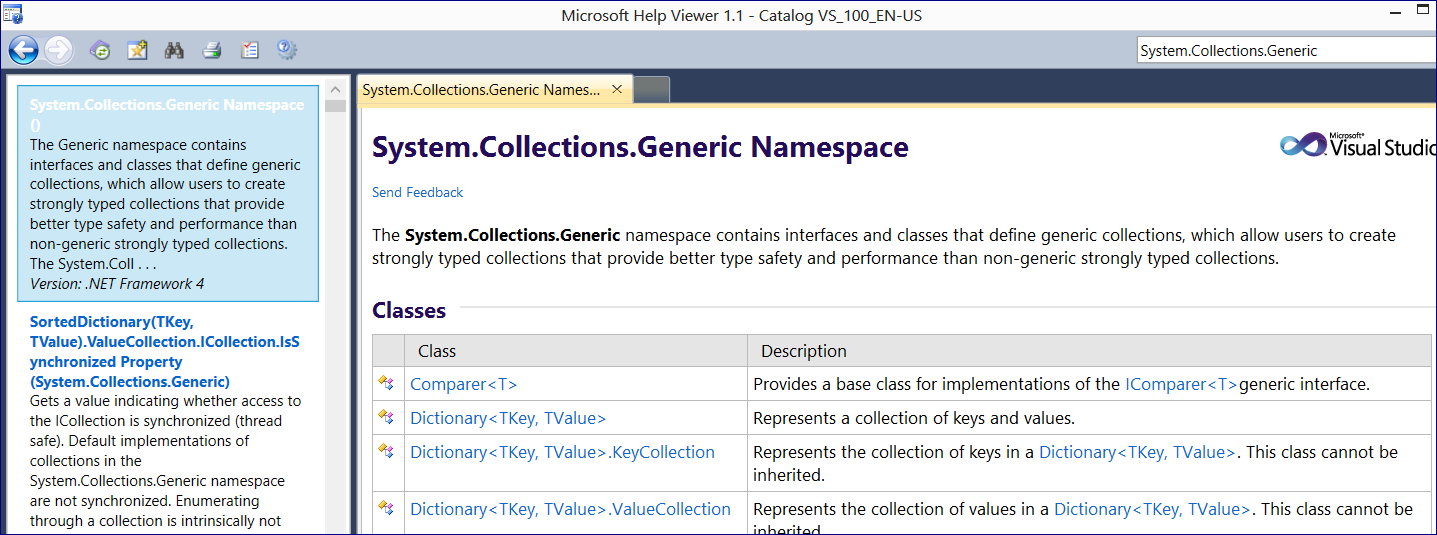
Collection classes hold collections of elements.

We can store anything but just A SINGLE DATA TYPE.

We cannot mix datatypes in a particular collection.

**Look in help for System.Collections.Generic**





Different collections are useful for particular tasks

We will use



// Using a Collection Classes

List<Texture2D> character = new List<Texture2D>();

//Using a Class Collection to add elements

character.Add(Content.Load<Texture2D>("Tomato\_100x100"));

character.Add(Content.Load<Texture2D>("Pear\_100x100"));

character.Add(Content.Load<Texture2D>("Banana\_100x100"));

character.Add(Content.Load<Texture2D>("Kiwi\_100x100"));

character.Add(Content.Load<Texture2D>("Cherry\_100x100"));

* **Unlike with arrays, when we create a collection object we don’t need to know the size.**
* **We access individual elements of a List using the list name and an index (the same as we do in array)**

# **WEEK 5**

**10.1 For Loops: The Basics (16:32)**

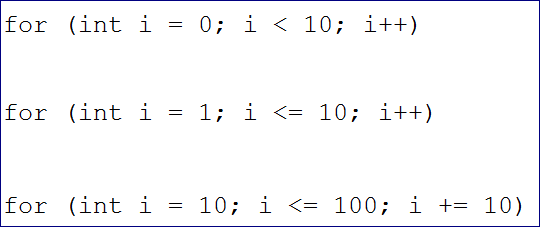
When we know how many times we’ll need to iterate when we get to a loop during execution of our program, we should use a for or foreach loop.

Top of Form

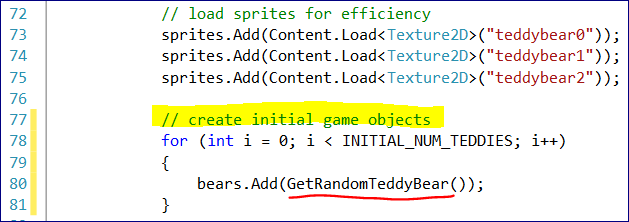
In the statement for (int i = 0; i < entities.Count; i++) the i is called the loop control variable.

the i < entities.Count part is called the Boolean part

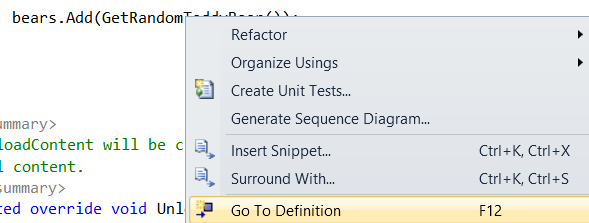
the i++ part is called the modifier.



--================



If we want to check a particular method, right click on it🡪 Go to Definition (F12)



private TeddyBear GetRandomTeddyBear()

{

Texture2D sprite = sprites[rand.Next(3)];

return new TeddyBear(sprite,

rand.Next(WINDOW\_WIDTH - sprite.Width),

rand.Next(WINDOW\_HEIGHT - sprite.Height),

WINDOW\_WIDTH, WINDOW\_HEIGHT);

}

--===============================

**protected override void LoadContent()**

{

// Create a new SpriteBatch, which can be used to draw textures.

spriteBatch = new SpriteBatch(GraphicsDevice);

// load sprites for efficiency

sprites.Add(Content.Load<Texture2D>("teddybear0"));

sprites.Add(Content.Load<Texture2D>("teddybear1"));

sprites.Add(Content.Load<Texture2D>("teddybear2"));

**// create initial game objects**

for (int i = 0; i < INITIAL\_NUM\_TEDDIES; i++)

{

bears.Add(GetRandomTeddyBear());

}

}

**protected override void Draw(GameTime gameTime)**

{

GraphicsDevice.Clear(Color.CornflowerBlue);

**// draw teddy bears**

spriteBatch.Begin();

**for (int i = 0; i < bears.Count; i++)**

**{**

**bears[i].Draw(spriteBatch);**

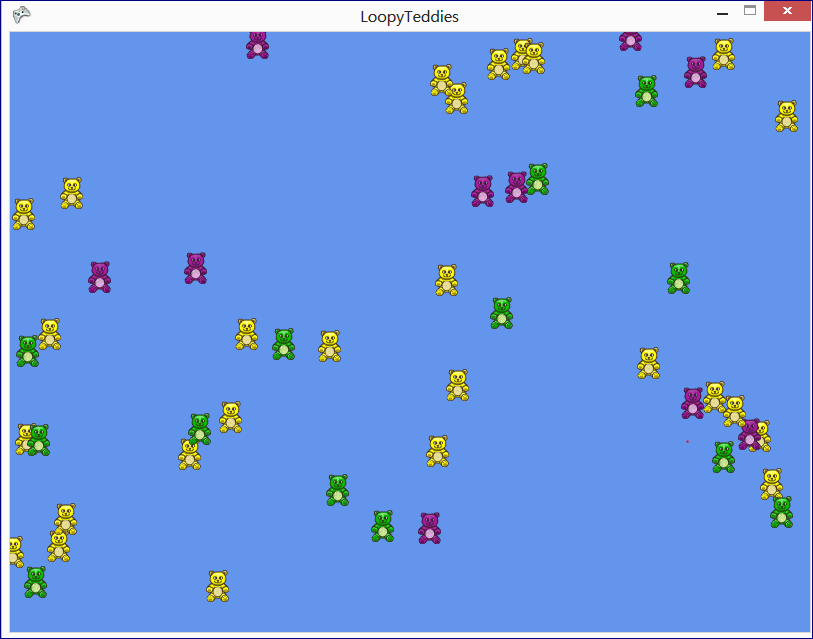
**}**

spriteBatch.End();

base.Draw(gameTime);

}

If we change const int INITIAL\_NUM\_TEDDIES = 50;



**protected override void Update(GameTime gameTime)**

{

// Allows the game to exit

if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)

this.Exit();

// spawn teddies as appropriate

// update teddy bears

for (int i = 0; i < bears.Count; i++)

{

bears[i].Update(gameTime);

}

base.Update(gameTime);

}

Spawn new Teddy Bears

// spawning support

const int TOTAL\_SPAWN\_DELAY\_MILLISECONDS = 1000;

int elapsedSpawnDelayMilliseconds = 0;

**protected override void Update(GameTime gameTime)**

{

// Allows the game to exit

if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)

this.Exit();

// spawn teddies as appropriate

elapsedSpawnDelayMilliseconds += gameTime.ElapsedGameTime.Milliseconds;

if (elapsedSpawnDelayMilliseconds >= TOTAL\_SPAWN\_DELAY\_MILLISECONDS)

{

elapsedSpawnDelayMilliseconds = 0;

bears.Add(GetRandomTeddyBear());

}

// update teddy bears

for (int i = 0; i < bears.Count; i++)

{

bears[i].Update(gameTime);

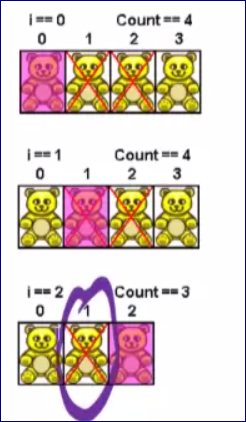
}

base.Update(gameTime);

}

**10.2 For Loops: Dead Teddies (12:27)**

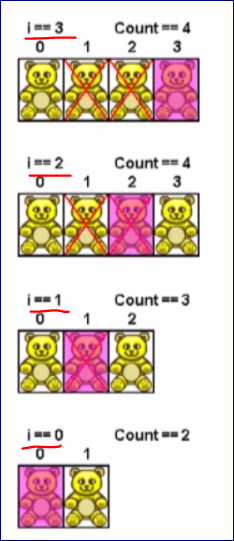
Using a for loop that moves from the beginning to the end of our list to remove dead teddy bears would be



Top of Form

Using a for loop that moves from the end to the beginning of our list to remove dead teddy bears would be correct.

**Removing items start from the back and move to the front!!!**



**protected override void Update(GameTime gameTime)**

{

// Allows the game to exit

if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)

this.Exit();

// spawn teddies as appropriate

elapsedSpawnDelayMilliseconds += gameTime.ElapsedGameTime.Milliseconds;

if (elapsedSpawnDelayMilliseconds >= TOTAL\_SPAWN\_DELAY\_MILLISECONDS)

{

elapsedSpawnDelayMilliseconds = 0;

bears.Add(GetRandomTeddyBear());

}

// update teddy bears

for (int i = 0; i < bears.Count; i++)

{

bears[i].Update(gameTime);

}

**// remove dead teddies**

for (int i = bears.Count - 1; i >= 0; i--)

{

if (!bears[i].Active)

{

bears.RemoveAt(i);

}

}

base.Update(gameTime);

}

Bottom of Form

**10.3 Foreach Loops (18:43)**

You can **NEVER** change the collection you’re iterating over inside a foreach loop.

We do not need an index of the element.

The foreach loop iterates over the collection starting at the first element.

The syntax:

// update and blow up teddy bears

foreach (TeddyBear teddyBear in bears)

**// Datatype var name collection name**

{

teddyBear.Update(gameTime);

}

Assuming entities is an **array** (not a collection)

foreach (GameEntity entity in entities)   
{   
entity.Update();   
}   
would

1. **update each game entity in the array**
2. **blow up if any element of the entities array is null**

**Explosion of Teddy Bears.**

// game objects

List<TeddyBear> bears = new List<TeddyBear>();

List<Explosion> explosions = new List<Explosion>();

**Load content**

explosionSprite = Content.Load<Texture2D>("explosion");

// get user input

MouseState mouse = Mouse.GetState();

**// update and blow up teddy bears**

foreach (TeddyBear teddyBear in bears)

{

teddyBear.Update(gameTime);

if (teddyBear.CollisionRectangle.Contains(mouse.X, mouse.Y))

{

teddyBear.Active = false;

explosions.Add(new Explosion(explosionSprite,

teddyBear.CollisionRectangle.Center.X,

teddyBear.CollisionRectangle.Center.Y));

}

}

**// update explosions**

foreach (Explosion explosion in explosions)

{

explosion.Update(gameTime);

}

**DRAW METHOD**

**// draw game entities**

spriteBatch.Begin();

foreach (TeddyBear teddyBear in bears)

{

teddyBear.Draw(spriteBatch);

}

foreach (Explosion explosion in explosions)

{

explosion.Draw(spriteBatch);

}

spriteBatch.End();

**UPDATE METHOD**

// update and blow up teddy bears

foreach (TeddyBear teddyBear in bears)

{

teddyBear.Update(gameTime);

if (teddyBear.CollisionRectangle.Contains(mouse.X, mouse.Y))

{

teddyBear.Active = false;

explosions.Add(new Explosion(explosionSprite,

teddyBear.CollisionRectangle.Center.X,

teddyBear.CollisionRectangle.Center.Y));

}

}

// update explosions

foreach (Explosion explosion in explosions)

{

explosion.Update(gameTime);

}

// remove dead teddies

for (int i = bears.Count - 1; i >= 0; i--)

{

if (!bears[i].Active)

{

bears.RemoveAt(i);

}

}

// remove dead explosions

for (int i = explosions.Count - 1; i >= 0; i--)

{

if (!explosions[i].Active)

{

explosions.RemoveAt(i);

}

}

**10.4 While Loops (13:47)**

Top of Form

The loop we’ll use to perform iteration in our C# code when we don’t know how many times to loop is the

**Input Validation** is a standard problem to solve with the WHILE LOOP.

We can avoid lots of loop problems by thinking about ITM, which means **Initialize Test Modify.**

static void Main(string[] args)

{

// print menu

Console.WriteLine("J - Jump");

Console.WriteLine("C - Crouch");

Console.WriteLine("Q - Quit");

Console.WriteLine();

// prompt for and get menu choice

Console.Write("Enter choice: ");

char choice = char.Parse(Console.ReadLine().ToUpper());

// validate input

while (choice != 'J' && choice != 'C' && choice != 'Q')

{

// print an error mesage

Console.WriteLine();

Console.WriteLine("Invalid input, must be 'J','C', or 'Q'");

Console.WriteLine();

// print menu

Console.WriteLine("J - Jump");

Console.WriteLine("C - Crouch");

Console.WriteLine("Q - Quit");

Console.WriteLine();

// prompt for and get menu choice

Console.Write("Enter choice: ");

choice = char.Parse(Console.ReadLine().ToUpper());

}

Console.WriteLine();

}



# **WEEK 6**

**11.1 Console Class: Fields and Properties (19:21)**

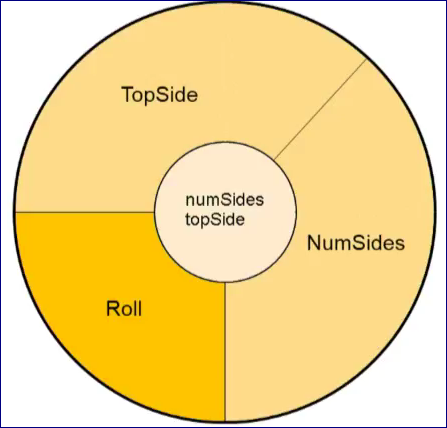
Writing our own class.

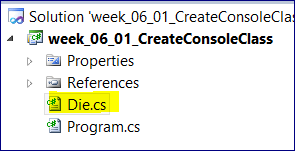
**State** -- We use fields to store the STATE of our objects. Properties are an effective way to provide access to fields.

**Behavior** – what an object can do, or what we can do with the object. We use METHODS to implement the behavior.

**Identity** – It lets us distinguish one object from another. We get IDENTITY when we call a CONSTRUCTOR to create a new object.

**Let’s create a Die class.**





namespace week\_06\_01\_CreateConsoleClass

{

/// <summary>

/// A die

/// </summary>

class Die

{

#region Fields

// fields to store the state internally

private int \_numSides;

private int \_topSide;

public Die(int numSides, int topSide)

{

this.\_numSides = numSides;

this.\_topSide = topSide;

}

#endregion

#region Properties

///<summary>

/// Gets the number of sides

///</summary>

// 1. Properties are CAPITALIZED

// 2. Use "return" keyword for the GET method

// 3. Use "value" keyword for the SET method

public int NumSides

{

get { return \_numSides;}

//set { \_numSides = value; } // we do not need to change the number of the die later on

}

///<summary>

/// Gets the Top Side

///</summary>

public int TopSide

{

get { return \_topSide; }

}

#endregion

}

}

**ENUMERATION**

Create a separate method. We do not need a class

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace week\_06\_01\_CreateConsoleClass

/// An enumeration for possible game states

{

public enum GameState

{

Play,

WaitForInput,

Respond

}

}

**Then in your main program you say:**

namespace week\_06\_01\_CreateConsoleClass

{

class Program

{

static void Main(string[] args)

{

GameState myState = GameState.Play;

}

}

}

**11.2 Console Class: Constructors (19:34)**

**We will create two different constructors:**

class Die

{

#region Fields

// fields to store the state internally

private int \_numSides;

private int \_topSide;

#endregion

#region Constructors

/// <summary>

/// Standard Constructor

/// </summary>

**public Die()**

// Constructors are always have the NAME OF THE CLASS

{

// inside of this constructor we will initialize fields that we have declared above

this.\_numSides = 6;

this.\_topSide = 1; // we can arbitrary take any number

}

/// <summary>

/// Constructor with num sodes specified

/// </summary>

/// <param name="numSides">number of sides for die</param>

**public Die (int numSides)**

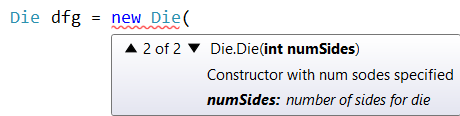
{

this.\_numSides = numSides;

this.\_topSide = 3;

}

Now we will have an overload:



**We can make the code better:**

#region Constructors

**/// Standard Constructor**

public Die():**this(6)**

{

}

**/// Constructor with num sodes specified**

public Die (int numSides)

{

this.\_numSides = numSides;

this.\_topSide = 3;

}

It means: find any constructor in the class that takes ONLY ONE argument and create an object

**11.3 Console Class: Method (12:34)**

**To create a method:**

* accessibility: public or private
* return type: e.g. string, int, void
* method name
* (optional parameter)
* {
* Code
* }

public void Roll()

{

this.\_topSide = rand.Next(1, this.\_numSides + 1);

Console.WriteLine("Top side after Rolling a Die: " + this.TopSide);

}

--===========

**Main program:**

static void Main(string[] args)

{

// test standard die

Die standardDie = new Die();

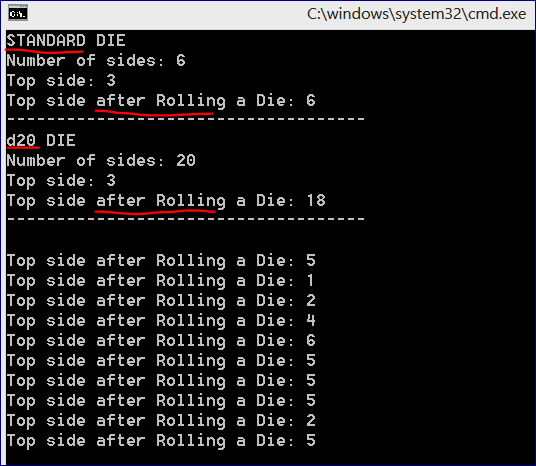
Console.WriteLine("STANDARD DIE");

Console.WriteLine("Number of sides: " + standardDie.NumSides);

Console.WriteLine("Top side: " + standardDie.TopSide);

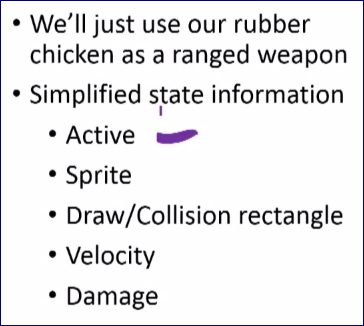
**standardDie.Roll();**

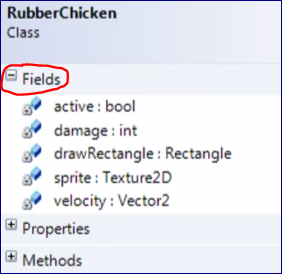
Console.WriteLine("------------------------------------");



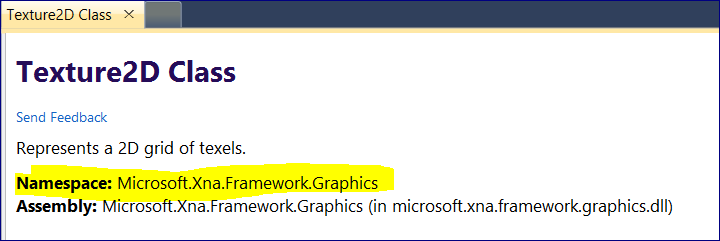
**11.4 XNA Class: Fields and Properties (20:57)**

Design and implement a RubberChicken class.





**We need to import a Texture2D class**



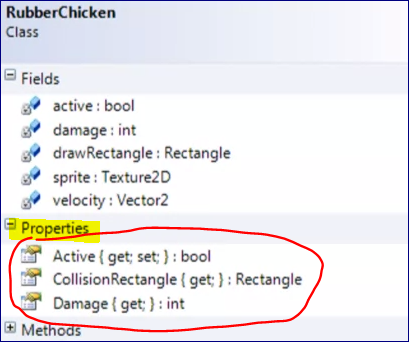
**Add namespaces:**

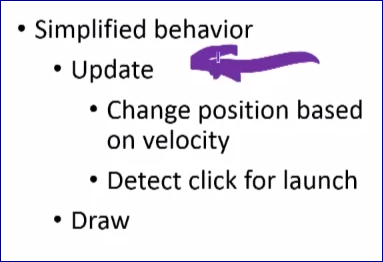
using Microsoft.Xna.Framework.Graphics; // Texture2D

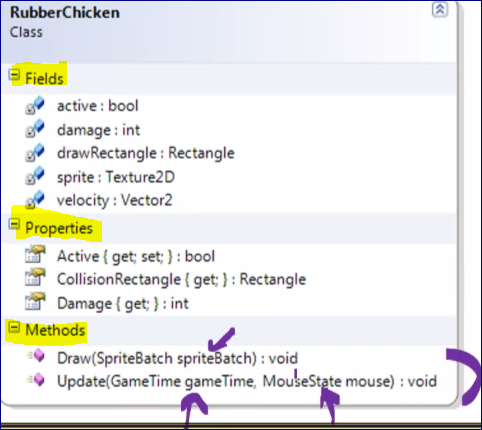
using Microsoft.Xna.Framework; // Rectangle

using Microsoft.Xna.Framework.Input; // MouseState

Create Properties:







**11.5 XNA Class: Constructor and Methods Part 1 (15:43)**

Objects get their identity when we call the CONSTRUCTOR.

/// <summary>

/// Constructor

/// </summary>

/// <param name="sprite">sprite</param>

/// <param name="location">location of center</param>

/// <param name="damage">damage from rubber chicken</param>

public RubberChicken(Texture2D sprite, Vector2 location, int damage)

{

this.sprite = sprite;

// build draw rectangle

drawRectangle = new Rectangle(

(int)location.X - sprite.Width/2

, (int) location.Y - sprite.Height / 2

,sprite.Width

,sprite.Height);

}

**Go to Game Class**

namespace RubberChicken

{

/// <summary>

/// This is the main type for your game

/// </summary>

public class Game1 : Microsoft.Xna.Framework.Game

{

GraphicsDeviceManager graphics;

SpriteBatch spriteBatch;

**// Declare a rubber chicken field in the Game class**

**RubberChicken rubberChicken;**

public Game1()

{

graphics = new GraphicsDeviceManager(this);

Content.RootDirectory = "Content";

}

**Load Method in the Game Class**

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

// create rubber chicken

rubberChicken = new RubberChicken(

Content.Load<Texture2D>("rubberchicken"),

new Vector2(graphics.PreferredBackBufferWidth / 2, graphics.PreferredBackBufferHeight / 2),

RUBBER\_CHICKEN\_DAMAGE);

}

**Rubber Chicken Class – DRAW METHOD**

#region Methods

/// <summary>

/// Draws the rubber chicken

/// </summary>

/// <param name="spriteBach">sprite batch</param>

public void Draw(SpriteBatch spriteBach)

{

spriteBach.Draw(sprite, drawRectangle, Color.White);

}

#endregion

**DRAW Method in the Game Class**

/// <summary>

/// 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

// draw the rubber chicken

spriteBatch.Begin();

rubberChicken.Draw(spriteBatch);

spriteBatch.End();

base.Draw(gameTime);

}

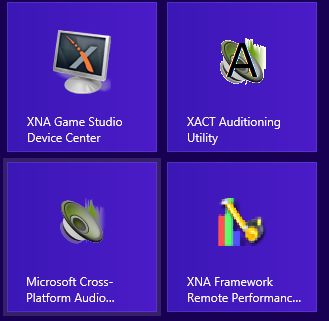
**11.6 XNA Class: Constructor and Methods Part 2 (15:06)**

**11.7 XNA Class: Use Part 1 (13:07)**

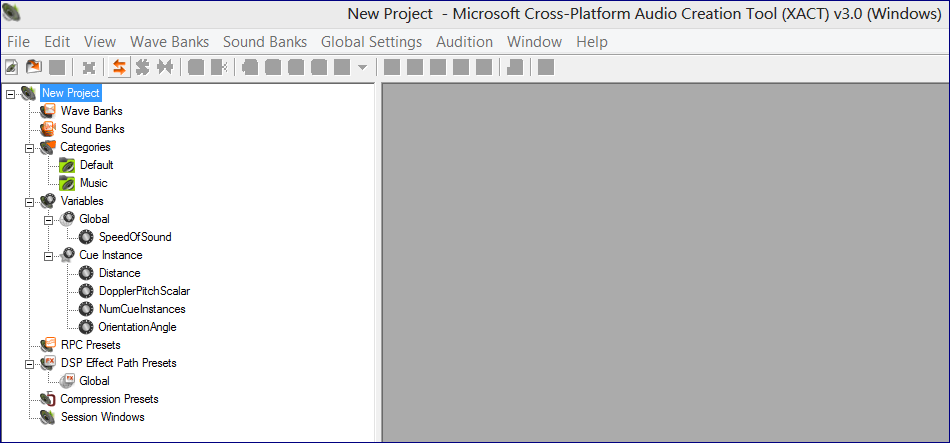
# **WEEK 7**

**12.1 XACT (10:06)**

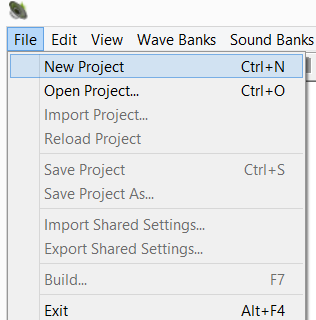
XACT is an acronym for Cross-Platform Audio Creation Tool

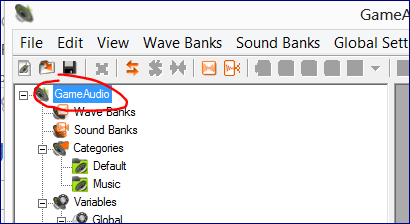


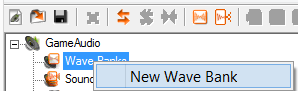
In your installed programs select “Microsoft Cross-Platform Audio Creation Tool”.



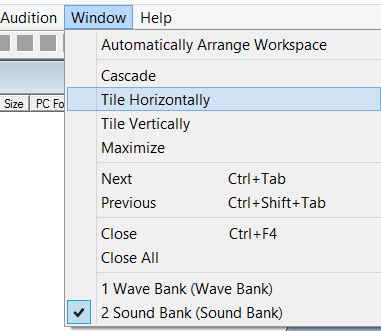
**File 🡪 New Project**

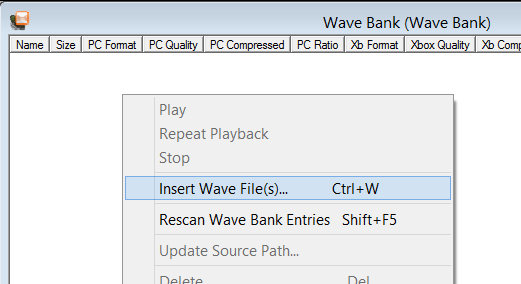


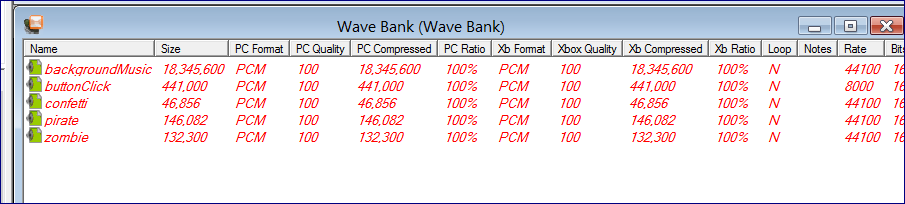




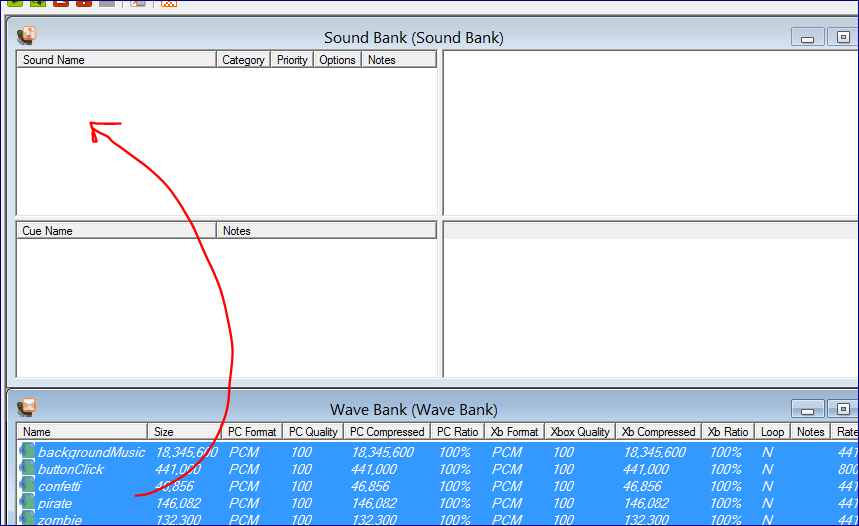


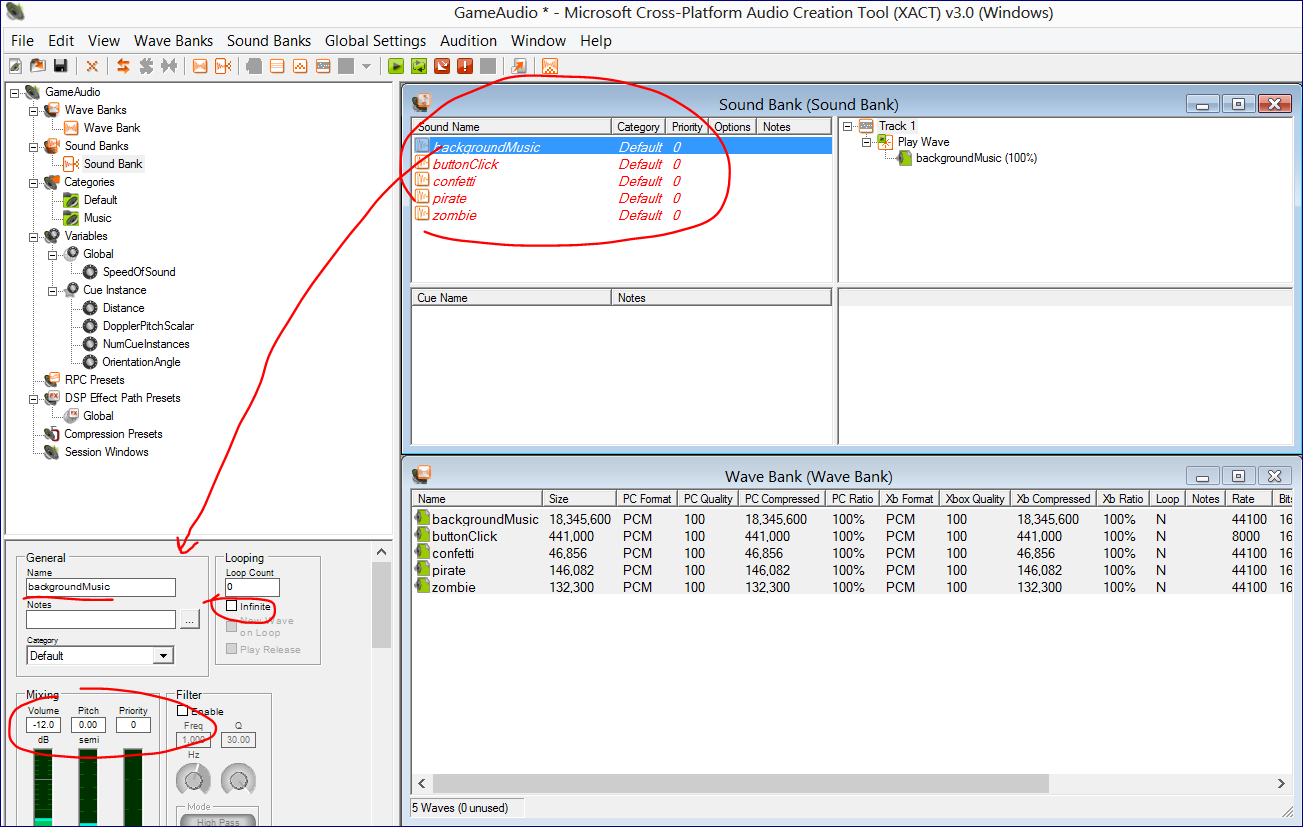






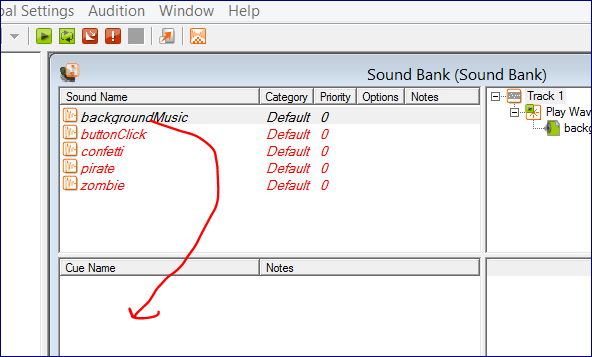
**The Wave Bank in an XACT project holds the actual wav files.**





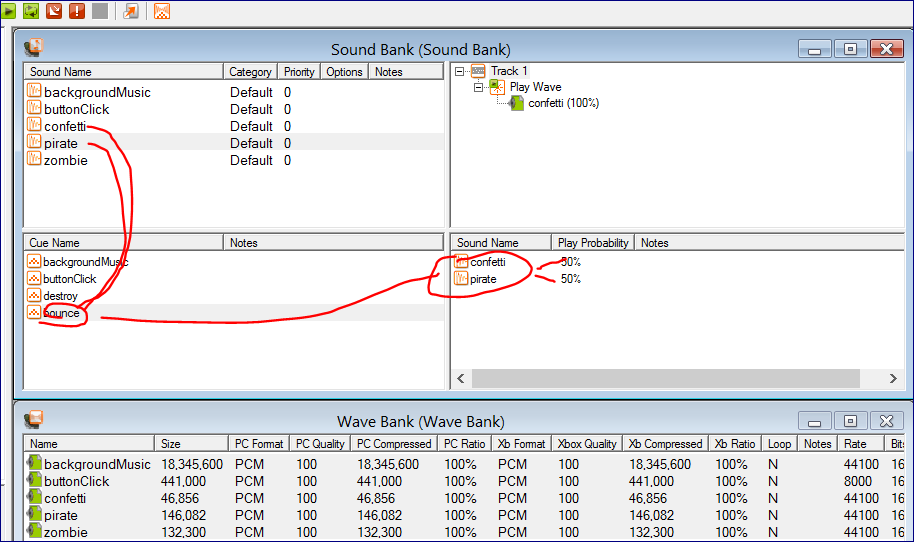
**We have a separate Sound Bank in an XACT project so we can manipulate sound properties without messing with the source wav files.**

**Cues will be played in the game**





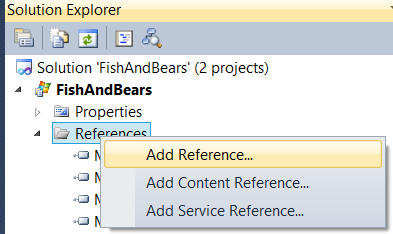
**We have separate Cues in an XACT project because**

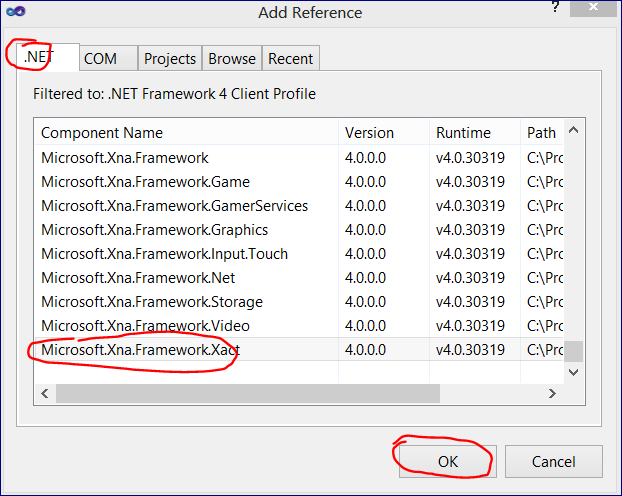


**12.2 XNA Audio With XACT (11:03)**

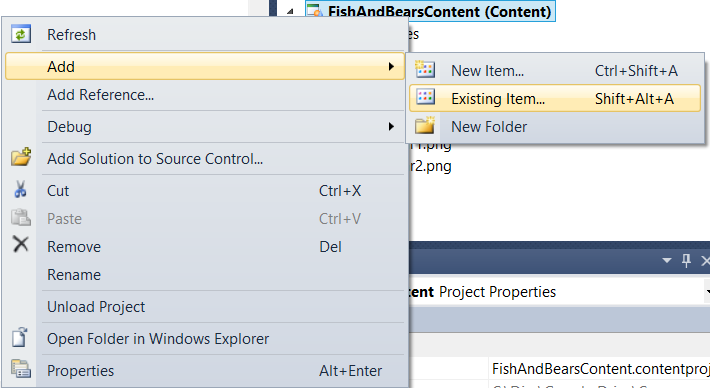
By default XNA template doesn’t have Sound Bank dll uploaded

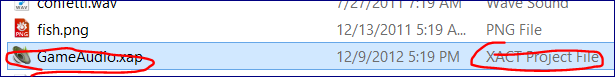
So, go to References – add Reference





Now we can add Content 🡪 Existing item





// load audio content

audioEngine = new AudioEngine(@"Content\GameAudio.xgs");

waveBank = new WaveBank(audioEngine, @"Content\Wave Bank.xwb");

soundBank = new SoundBank(audioEngine, @"Content\Sound Bank.xsb");

// start playing background music

soundBank.PlayCue("backgroundMusic");

private void CheckAndResolveFishBearCollisions()

{

// check collisions between fish and teddy bears

for (int i = bears.Count - 1; i >= 0; i--)

{

if (fish.Active && bears[i].Active &&

fish.CollisionRectangle.Intersects(bears[i].CollisionRectangle))

{

// check where the collision occurred

Rectangle overlap = Rectangle.Intersect(fish.CollisionRectangle,

bears[i].CollisionRectangle);

if (overlap.Left <= fish.Front && overlap.Right >= fish.Front)

{

// fish ate the bear

bears.RemoveAt(i);

// play eating sound

soundBank.PlayCue("destroy");

}

else

{

// non-head collision

bears[i].Bounce();

// play bouncing sound

soundBank.PlayCue("bounce");

}

}

}

Button Click

public void Update(MouseState mouse, SoundBank soundBank)

{

// check for mouse over button

if (drawRectangle.Contains(mouse.X, mouse.Y))

{

// highlight button

sourceRectangle.X = buttonWidth;

// check for click started on button

if (mouse.LeftButton == ButtonState.Pressed &&

buttonReleased)

{

clickStarted = true;

buttonReleased = false;

soundBank.PlayCue("buttonClick");

}

else if (mouse.LeftButton == ButtonState.Released)

{

buttonReleased = true;

// if click finished on button, change game state

if (clickStarted)

{

Game1.ChangeState(clickState);

clickStarted = false;

}

}

}

else

{

sourceRectangle.X = 0;

// no clicking on this button

clickStarted = false;

buttonReleased = false;

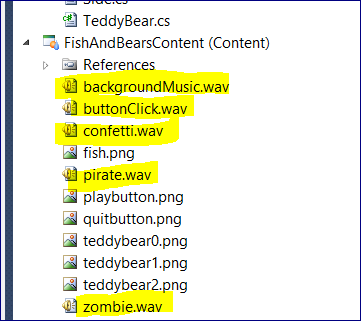
}

}

**12.3 XNA Audio Without XACT (10:53)**

For MAC users

Also just to use wav files.



public class Game1 : Microsoft.Xna.Framework.Game

{

// background music

SoundEffectInstance backgroundMusic;

// collision sound effects

SoundEffect destroy;

SoundEffect[] bounce = new SoundEffect[2];

Random rand = new Random();

// load and start playing background music

SoundEffect backgroundMusicEffect = Content.Load<SoundEffect>("backgroundMusic");

backgroundMusic = backgroundMusicEffect.CreateInstance();

backgroundMusic.IsLooped = true;

backgroundMusic.Play();

// load other sound effects

destroy = Content.Load<SoundEffect>("zombie");

bounce[0] = Content.Load<SoundEffect>("confetti");

bounce[1] = Content.Load<SoundEffect>("pirate");

**13.1 XNA Keyboard Input (10:41)**

Text Input and Output

**In the Update method create a** KeyboardState **object**

protected override void Update(GameTime gameTime)

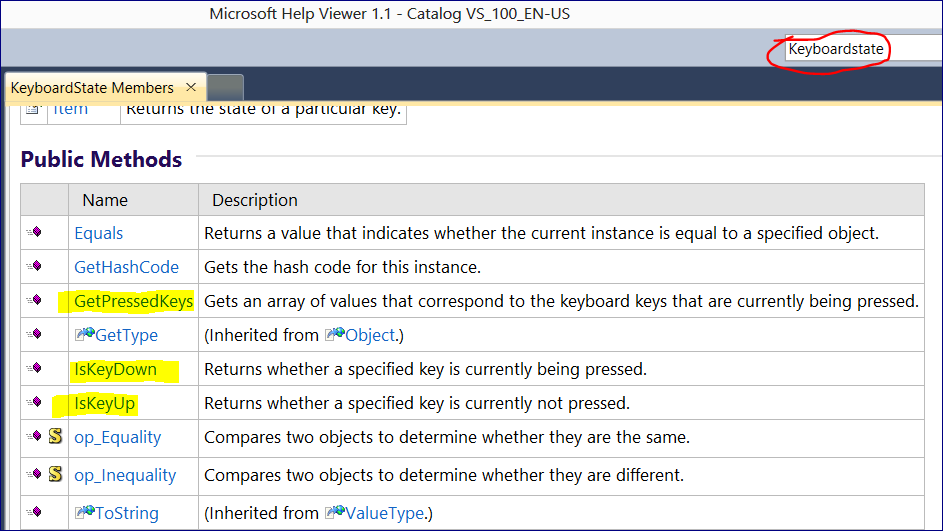
{

KeyboardState keyboard = Keyboard.GetState();

Keyboard.GetState() – returns a Keyboard object

To find out whether or not a particular key is currently pressed, we use **IsKeyDown**

**The Keyboard GetState method returns the current state of the keyboard.**



// Allows the game to exit

if (keyboard.IsKeyDown(Keys.Escape)) this.Exit();

// update the fish

fish.Update(keyboard);

**In the Fish – Update Method**

**FISH Class**

#region Public methods

/// <summary>

/// FISH Update METHOD necessary

/// </summary>

/// <param name="keyboard">current keyboard state</param>

public void Update(KeyboardState keyboard)

{

// move the fish based on the keyboard state

if (keyboard.IsKeyDown(Keys.Right))

{

X += FISH\_MOVE\_AMOUNT;

// set source rectangle for right image

sourceRectangle.X = 0;

front = Side.Right;

}

if (keyboard.IsKeyDown(Keys.Left))

{

X -= FISH\_MOVE\_AMOUNT;

// set source rectangle for left image

sourceRectangle.X = frameWidth;

front = Side.Left;

}

if (keyboard.IsKeyDown(Keys.Up))

{

Y -= FISH\_MOVE\_AMOUNT;

}

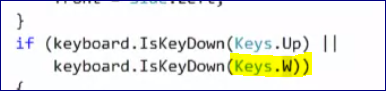
if (keyboard.IsKeyDown(Keys.Down))

{

Y += FISH\_MOVE\_AMOUNT;

}

}

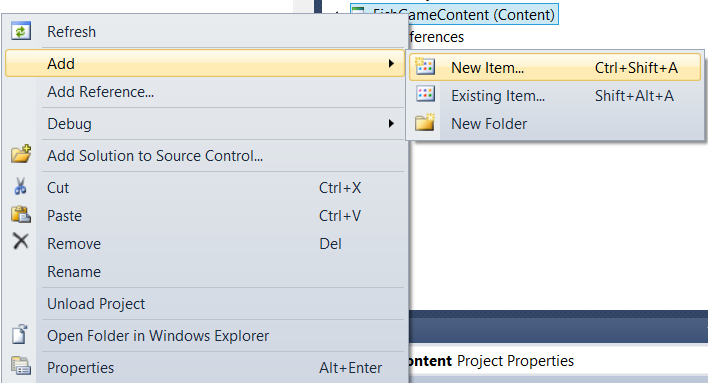


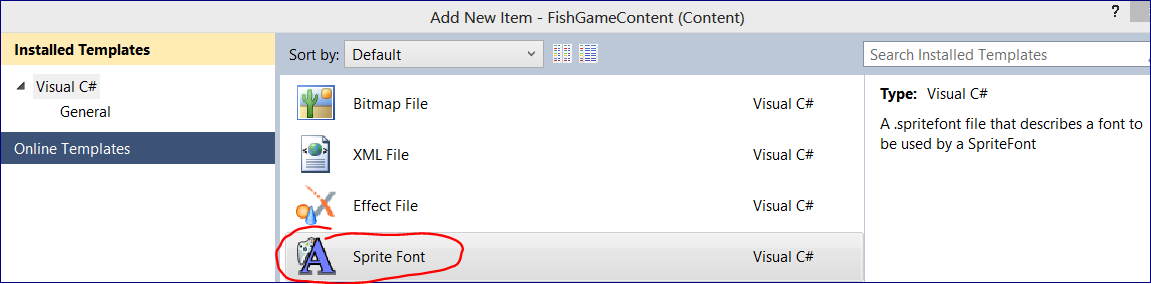
**13.2 XNA Text Output (12:32)**

To output text information in XNA we cannot use Console.WriteLine()

We need to use a special kind of content.

Go to Game Content Folder 🡪 Add 🡪 New Item 🡪 **Sprite Font**

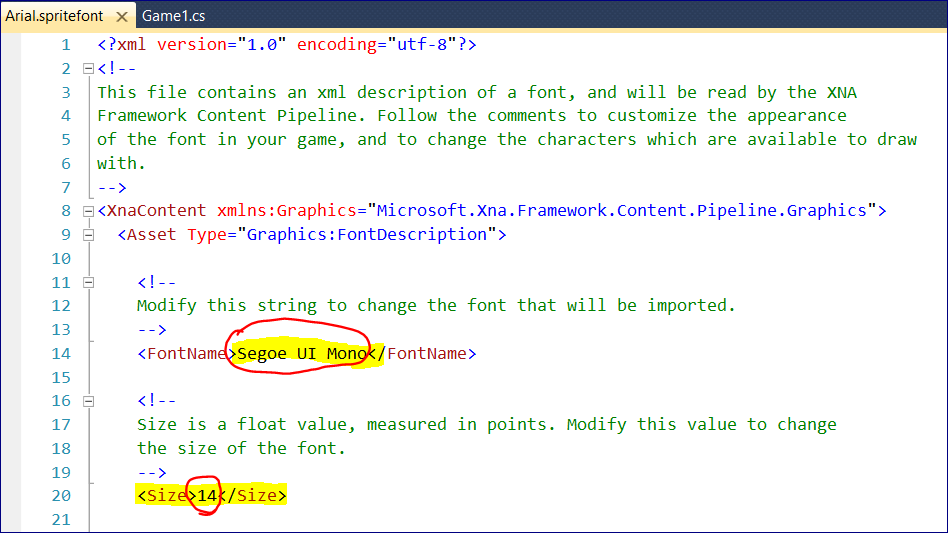




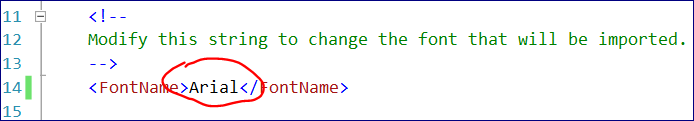


It creates an XML file. In this file we can change several parameters, e.g. the font name to Arial.

**Important: this font must be installed on the computer before the program build (F6).**



It will be then compiled to a .xnb file



// score handling

int score = 0;

const int BEAR\_SCORE = 10;

const string SCORE\_STRING\_PREFIX = "Score: ";

string scoreString;

public Game1()

{

graphics = new GraphicsDeviceManager(this);

Content.RootDirectory = "Content";

---=======

protected override void **LoadContent()**

{

// Create a new SpriteBatch, which can be used to draw textures.

spriteBatch = new SpriteBatch(GraphicsDevice);

// create the fish

fish = new Fish(Content, graphics.PreferredBackBufferWidth,

graphics.PreferredBackBufferHeight, "fish", graphics.PreferredBackBufferWidth / 2,

graphics.PreferredBackBufferHeight / 2);

// load sprite font and initialize score string

font = Content.Load<SpriteFont>("Arial");

scoreString = **GetScoreString**(score);

}

--=================

*Private method for simplicity reasons*

private string **GetScoreString**(int score)

{

return SCORE\_STRING\_PREFIX + score;

}

--=================

*Update method*

protected override void Update(GameTime gameTime)

{

// kill bears that are colliding with the fish

if (bear.Active && fish.CollisionRectangle.Intersects(bear.CollisionRectangle))

{

bear.Active = false;

score += BEAR\_SCORE;

scoreString = GetScoreString(score);

}

}

--=================

*Draw method*

spriteBatch.Begin();

spriteBatch.**DrawString**(font, scoreString, scorePosition, Color.White);

spriteBatch.End();

