

The background is a dark, atmospheric illustration. It features a stone archway or doorway. On the left side of the arch, there is a glowing yellow light source, possibly a lantern or a small fire, casting a warm glow. The interior of the arch is filled with a blue-tinted scene showing several figures in a dimly lit environment, possibly a cave or a large hall. The figures appear to be engaged in some activity, with one figure holding a long staff or pole. The overall mood is mysterious and ancient.

Level 1

Methods

What Are We Going to Make?

Using C# we're going to create an application for storing and retrieving band information.

What our application will do:

- Store information about a band and its musicians
- Announce the band
- Announce the musicians

>>>

What is the name of your band?

\$ The C Sharps

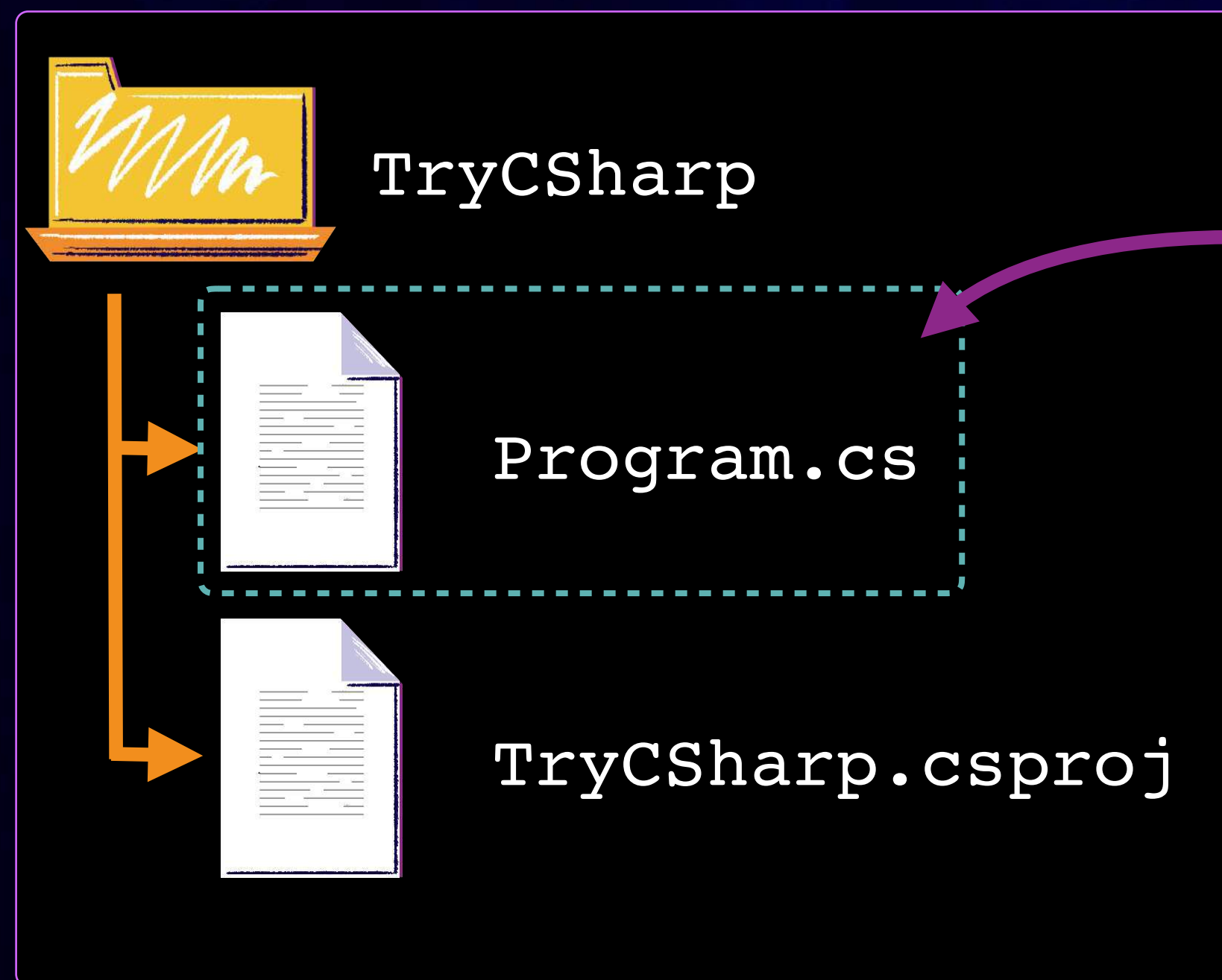
Welcome The C Sharps to the stage!

In this level:

- Collect the band's name
- Announce the band

Creating Our New Application

All C# console applications contain a `csproj` file and a `Program.cs` file.



The entry point to our application

The Program.cs File

The generated Program.cs file contains a Program class and Main method.

Program.cs

```
using System;  
class Program  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Hello World!");  
    }  
}
```

Class Program

Method Main

Methods allow us to break our code into reusable functional blocks. We should create a separate method to announce our band... but how do we do that?

Method Name and Parameters

Methods **require** a name and parameters.

Name - The name we'll use to call the method throughout our program.

Parameters - values passed into the method.

*Name uses the PascalCase
naming convention*

*Parameter names use the
camelCase naming convention*

```
AnnounceBand(string bandName)
{
}
```

*Parameters must be
prefixed by their data type*

*This covers what our method takes in as parameters,
but what about the type of data it returns?*

Method Return Type and Statement

Methods **require** a return type and return statement.

Return type - The data type we expect to get back from the method.

```
string AnnounceBand(string bandName)
{
    return "Welcome " + bandName;
}
```

Return statement - Passes a value to the caller of the method.

But how do we call our method?

Calling a Method

To call a method, we use its Name and replace the parameters with the values to be sent to the method.

```
string AnnounceBand(string bandName)
{
    return "Welcome " + bandName;
}
```

coming from previous slide with magic move

*AnnounceBand prepends
"Welcome " to the provided bandName*

*Call the method using its Name
followed by the parameter values*

```
string announcement = AnnounceBand("The C Sharps");
Console.WriteLine(announcement);
```

animate this code box first



Welcome The C Sharps

Method Calls Can Be Used as Arguments

When a method is used as an argument, the data returned by the method will be used as the argument.

```
string AnnounceBand(string bandName)
{
    return "Welcome " + bandName;
}
```

We can use AnnounceBand as an argument to Console.WriteLine

```
Console.WriteLine(AnnounceBand("The C Sharps"));
```



Welcome The C Sharps

We should just move the Console.WriteLine inside our AnnounceBand method.

Method Error

AnnounceBand is throwing an error because its return type is a string, but isn't returning anything.

*We are writing to the console, but NOT
returning a **string** from the method*

```
string AnnounceBand(string bandName)
{
    Console.WriteLine("Welcome " + bandName);
}
```



ERROR: Not all code paths return a value

How do we write a method that will not return anything?

Return Type void When Nothing Returned

Methods that don't return anything should use the **void** return type.

Indicates no value will be returned by the method...

```
void AnnounceBand(string bandName)
{
    Console.WriteLine("Welcome " + bandName);
}
```



Welcome The C Sharps

...and return statement is not required

Expand Welcome Message

Announcing our band should be more inviting.

Change our Announce to "Welcome _____ to the stage!"

```
void AnnounceBand(string bandName)
{
    Console.WriteLine("Welcome " + bandName + " to the stage!");
}
```



Welcome The C Sharps to the stage!

Set Up Main Method to Use AnnounceBand

The Main method is now calling the AnnounceBand method and passing it the band name.

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("What is the name of your band?");
        string name = Console.ReadLine();
        AnnounceBand(name);
    }

    void AnnounceBand(string bandName) {...}
}
```

Let us review how the program will actually run this code.

Method Execution Step-By-Step

Execution starts on the Main method then transfers to AnnounceBand.

Program.cs

```
static void Main(string[] args)
{
    1 Console.WriteLine("What is the name of your band?");
    string name = Console.ReadLine();
    AnnounceBand(name);
}

void AnnounceBand(string bandName)
{
    Console.WriteLine("Welcome " + bandName + " to the stage!");
}
```


Method Execution Step-By-Step

Execution starts on the Main method then transfers to AnnounceBand.

Program.cs

```
static void Main(string[] args)
{
    1 Console.WriteLine("What is the name of your band?");
    2 string name = Console.ReadLine();
    AnnounceBand(name);
}

void AnnounceBand(string bandName)
{
    Console.WriteLine("Welcome " + bandName + " to the stage!");
}
```


Method Execution Step-By-Step

Execution starts on the Main method then transfers to AnnounceBand.

Program.cs

```
static void Main(string[] args)
{
    1 Console.WriteLine("What is the name of your band?");
    2 string name = Console.ReadLine();
    > AnnounceBand(name);
}
    Main will transfer execution to AnnounceBand
void AnnounceBand(string bandName)
{
    3 Console.WriteLine("Welcome " + bandName + " to the stage!");
}
```



Methods Run Synchronously

Once `AnnounceBand` completes, it transfers execution back to the `Main` method.

Program.cs

```
static void Main(string[] args)
{
    1 Console.WriteLine("What is the name of your band?");
    2 string name = Console.ReadLine();
    > AnnounceBand(name);
    4 Main is finished executing
}

void AnnounceBand(string bandName)
{
    3 Console.WriteLine("Welcome " + bandName + " to the stage!");
}
```



AnnounceBand will transfer execution back to Main

Our Working Application

Our application now performs the functions we set out to accomplish in this level.

Current Features:

- Collects the band's name
- Announces the band

>>>

What is the name of your band?

\$ The C Sharps

Welcome The C Sharps to the stage!

A Quick Recap on Methods

We divide our executable code into logical pieces using methods.

- Methods contain our classes' executable code
- Method names are case sensitive
- To call a method, use its Name followed by values we intend to pass into the method in parenthesis



Level 2

Classes

Expanding Our Bands

Our Band needs to store their musicians to be able to announce them.

What our application will do:

- Store information about a band and its musicians
- Announce the band
- Announce the musicians

Band

Name: "The C Sharps"
Musicians: 4

In this level:

- Create our Band class
- Share methods between Band and Program

What Makes up a Band

Our band will have several properties including a Name and Musicians.

Band

```
Name: "The C Sharps"  
Musicians: 4
```

But how do we create this in code?

Classes Define Our Objects

The Class defines the structure of our Band, while an object is an actual instance of a Band.

The Band Class

Band.cs

```
class Band
{
    string Name;
    int Musicians;
}
```

Think of a class like a blueprint

Instance of the Band Class (object)

Band

Name: "The C Sharps"
Musicians: 4

*Think of an object as an example of
what the blueprint describes*

Declaring Our Band Class

To declare a class, we use the class keyword followed by the name we want for our class.

Band.cs

```
class Band  
{  
  
}
```

*File names typically match whatever
class is contained in that file*

Band (End Goal)

Name: "The C Sharps"
Musicians: 4

Instance Variables

Instance Variables define the information we store in each instance of our class.

Band.cs

```
class Band
{
    string Name;
    int Musicians;
}
```

Band (End Goal)

Name: "The C Sharps"
Musicians: 4

Okay, so where exactly does *Object* come into play with a Class

Instances of an Object

We can have several instances of our Band class, each with their own values.

Band.cs

```
class Band
{
    string Name;
    int Musicians;
}
```

Band 1

Name: The C Sharps
Musicians: 4

Band 2

Name: The F Sharps
Musicians: 2

Band 3

Name: The VB Nets
Musicians: 1

Refactor AnnounceBand Method

Our AnnounceBand Method should live in our Band class.

Band.cs

```
class Band
{
    string Name;
    int Musicians;
}
```

Currently AnnounceBand lives in our Program class even though it is designed to announce our Band, so we should move it into our Band class

Program.cs

```
void AnnounceBand(string bandName)
{
    Console.WriteLine("Welcome " + bandName + " to the stage!");
}
```


Refactor AnnounceBand Method

Our AnnounceBand Method should live in our Band class.

Band.cs

```
class Band
{
    string Name;
    int Musicians;

    void AnnounceBand(string bandName)
    {
        Console.WriteLine("Welcome " + bandName + " to the stage!");
    }
}
```

Now AnnounceBand lives in our Band class

Some Issues With AnnounceBand

Our AnnounceBand Method has a few smells we need to clean up.

Band.cs

```
class Band
{
    string Name;
    int Musicians;

    void AnnounceBand(string bandName)
    {
        Console.WriteLine("Welcome " + bandName + " to the stage!");
    }
}
```

The name AnnounceBand is redundant

We can now use our Name variable instead of using a parameter

Refactor AnnounceBand Method

Renaming the method and using the Name variable Announce better fits in the Band class.

Band.cs

```
class Band
{
    string Name;
    int Musicians;

    void Announce()
    {
        Console.WriteLine("Welcome " + Name + " to the stage!");
    }
}
```

Rename AnnounceBand to just Announce

Remove the parameter and use our Name variable instead

What happens when we attempt to compile this?

Reference Error

We're getting an error because Console isn't part of our Band class.

Band.cs



```
class Band
{
    string Name;
    int Musicians;

    void Announce()
    {
        Console.WriteLine("Welcome " + Name + " to the stage!");
    }
}
```

*Console doesn't exist within Band so
our compiler doesn't know what to do
here*

➔ **ERROR: The name 'Console' doesn't exist in the current context**

Namespaces

Namespaces are used to organize classes further allowing reuse of class names.

Global Namespace



Program Class



Band Class

*All of our classes live in a **Global** Namespace available throughout the application*

System Namespace

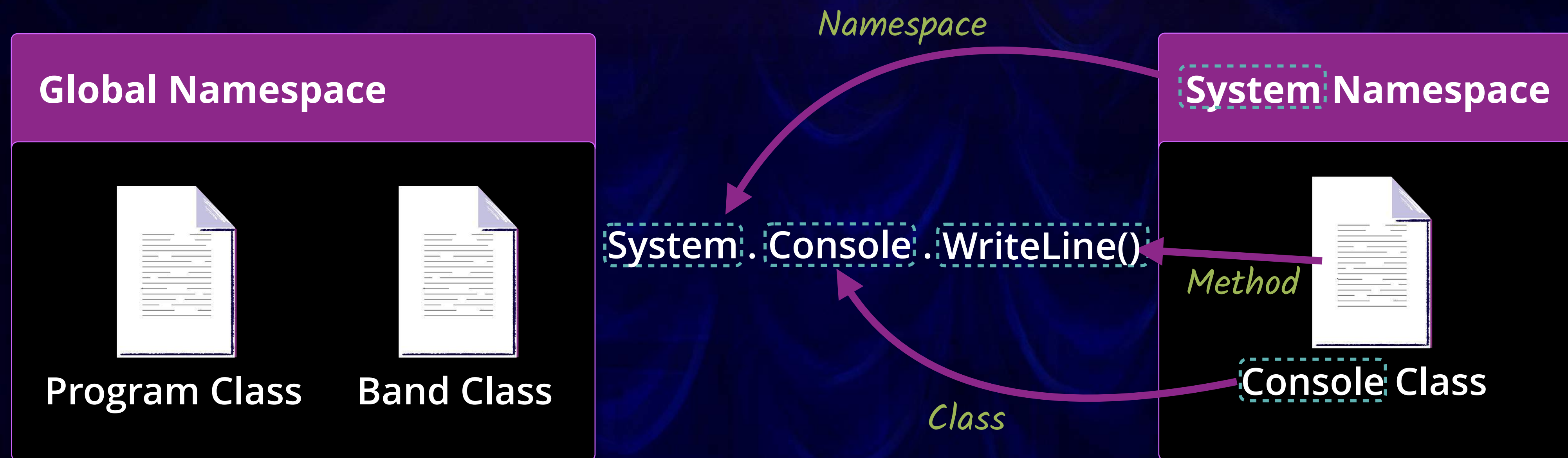


Console Class

*Console lives in the built in **System** namespace*

Accessing Outside Namespaces

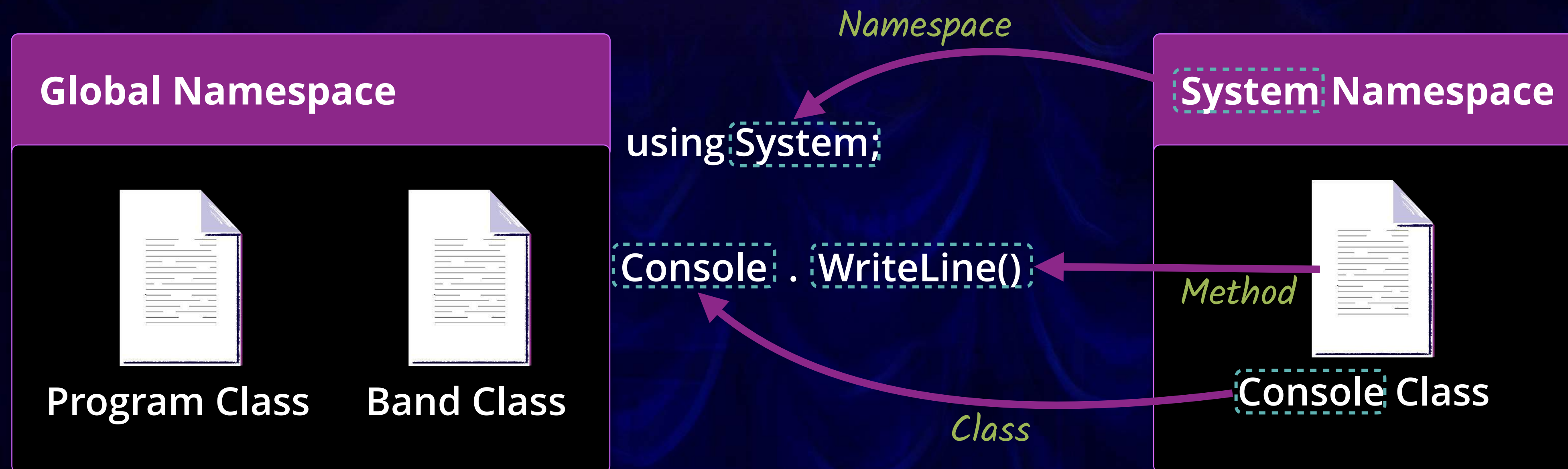
Namespaces can be accessed using the Namespace.Class.Method format.



Rewriting that every time can get annoying is there a shorter way of doing this?

Using Directives

You can reference a namespace throughout a file with a using directive.



We didn't get an error doing this in Program.cs because it already had using System; in it

This allows us to use our Console class throughout a file and only reference System once

Using Directive

Adding a using directive for System resolves our error.



Band.cs

```
using System;
```

```
class Band
```

```
{
```

```
    string Name;
```

```
    int Musicians;
```

```
    void Announce()
```

```
    {
```

```
        Console.WriteLine("Welcome " + Name + " to the stage!");
```

```
    }
```

```
}
```

*This gives Band access to our System
Namespace, letting our Compiler know where to
find Console*

Now how do we create a Band object using our Band class?

Create an Instance of Our Band Object

We can use `new Band()` to instantiate a new Band object.

Program.cs

```
...
static void Main(string[] args)
{
    Console.WriteLine("What is the name of your band?");
    Band band = new Band();
    band.Name = Console.ReadLine();
}
...
```

new Band() will create our new band object

Change Console.ReadLine to set our band's Name variable

Band Is Inaccessible

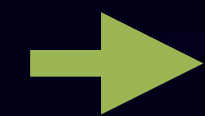
An error is being thrown, saying the Band class is not accessible.

Program.cs



```
...
static void Main(string[] args)
{
    Console.WriteLine("What is the name of your band?");
    Band band = new Band();
    band.Name = Console.ReadLine();
}
...
```

Band is throwing an error because we've not made it accessible to other classes



ERROR: 'Band' is inaccessible due to protection level

What is a protection level and why is it making Band inaccessible?

Understanding Protection Levels

Protection Levels are set by Access Modifiers and determine where classes, methods, etc can be accessed.

Public Access Modifier

```
public class Band
```

public makes the code accessible anywhere

Same Project

Same Class

Different Class

Different Project

Private Access Modifier

```
private class Band
```

private restricts the code to be accessible only in the same class



Access Modifiers default to private anytime one isn't provided

The Public Access Modifier

Adding public to classes, methods, variables, etc makes them visible to other classes.



Band.cs

```
using System;  
...  
public class Band  
{  
    public string Name;  
    public int Musicians;  
  
    public void Announce()  
    {  
        Console.WriteLine("Welcome " + Name + " to the stage!");  
    }  
}
```

This will make Band accessible from our Program class

Now we can finish calling our Announce method in our Program class's Main Method

Call The Announce Method

`band.Announce()` will run the Announce method and use the band's Name in the announcement.

Program.cs

```
...
static void Main(string[] args)
{
    Console.WriteLine("What is the name of your band?");
    Band band = new Band();
    band.Name = Console.ReadLine();
    band.Announce();
}
...
```

What is the name of your band?

>>>

\$ The C Sharps

Welcome The C Sharps to the stage!

A Quick Recap on Classes

Classes are used to further organize our code into collections.

- C# is an Object-Oriented programming language based around Classes
- Classes define what an object will look like
- An object is an actual instance of a class
- The default access-modifier is private, restricting code to only be used within the same class
- To use something from a different namespace you can use the full namespace.class.methodname or a using directive



Level 3

Groups of Objects

Create List of Musicians

Our bands will need to contain our musicians in order to store and announce them.

What our application will do:

- Store information about a band and it's musicians
- Announce the band
- Announce the musicians

In this level:

- Create a List of Musicians per band
- Create method to add a Musician to the List

Our Musician Object

We've created a simple Musician Class to hold data about our band's musicians.

Musician.cs

```
using System;

public class Musician
{
    public string Name;
    public string Instrument;

    public void Announce()
    {
        Console.WriteLine(Name + " on the " + Instrument + "!");
    }
}
```

Before we tackle groups of objects, we'll cover the short hand way to instantiate an object



Object_INITIALIZER

The Object_INITIALIZER allows us to instantiate and set variables in a single line.

Instantiating Musician without using Object_INITIALIZER

```
var musician = new Musician()  
Musician.Name = "Robert";  
Musician.Instrument = "Guitar";
```

Instantiating Musician with Object_INITIALIZER

```
Musician musician = new Musician{ Name="Robert", Instrument="Guitar" }
```

Between Curly Braces we set all our variables separating variables using commas

To add a group of Musicians to our Band class, we need to know how to group objects

Types of Groups of Objects

In C# we have several choices for grouping objects. Here are two examples:

Array

List

Arrays

Arrays are best used for a group of strongly-typed objects of a fixed number.

We create our Array of Musicians

Array

How many objects fit in our array

```
Musician[] musicians = new Musician[2];  
musicians[0] = new Musician{Name="Robert", Instrument="Guitar"}  
musicians[1] = new Musician{Name="Sarah", Instrument="Keyboard"}
```

If Thomas joins though we need to change the size of our band...

To change the Array size we have to recreate the entire array.

```
musicians = new Musician[3];  
musicians[0] = new Musician{Name="Robert", Instrument="Guitar"}  
musicians[1] = new Musician{Name="Sarah", Instrument="Keyboard"}  
musicians[2] = new Musician{Name="Thomas", Instrument="Drums"}
```



This is going to be a pain to manage with bands constantly changing members...

Lists

Lists are great for handling a group of strongly-typed objects of unknown or varying number.

List

The List's Data-Type

We create our List of Musicians

```
List<Musician> Musicians = new List<Musician>();  
Musicians.Add(new Musician{Name="Robert",Instrument="Guitar"});  
Musicians.Add(new Musician{Name="Sarah",Instrument="Keyboard"});
```

If Thomas joins we can just add him using Add

```
Musicians.Add(new Musician{Name="Thomas",Instrument="Drums"});
```

The List will resize automatically!



This will be much nicer for handling bands constantly changing members and won't break if we expand Musician later!

Musicians List

Add a using directive for System.Collections.Generic and change the Musician's variable.

Band.cs

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

public class Band
{
    public string Name;
    public List<Musician> Musicians;

    public void Announce() {...}
}
```

List is in the built-in

*System.Collections.Generic
Namespace*

*Change Musicians to a List of **Musician***

*Now that we have our list of **Musician** objects, we should create a method to add to it*

Create AddMusician Method

Declare our new Musician variable that we'll add to our Musicians List.

Band.cs

```
...  
public void AddMusician()  
{  
    Musician musician = new Musician();  
}  
...  
  
OR  
  
[var] musician = new Musician();
```

var will infer our variable's datatype based on whatever is after the = character



The compiled results with or without var will be the same so do what you prefer!

Create AddMusician Method

Set musician's Name and Instrument variables with Console.ReadLine.

Band.cs

```
...  
public void AddMusician()  
{  
    var musician = new Musician();  
    Console.WriteLine("What is the name of the musician to be added?");  
    musician.Name = Console.ReadLine();  
    Console.WriteLine("What instrument does " + musician.Name + " play?");  
    musician.Instrument = Console.ReadLine();  
}  
...
```


Create AddMusician Method

Add musician to our Musicians List using the Add method.

Band.cs

```
...  
public void AddMusician()  
{  
    var musician = new Musician();  
    Console.WriteLine("What is the name of the musician to be added?");  
    musician.Name = Console.ReadLine();  
    Console.WriteLine("What instrument does " + musician.Name + " play?");  
    musician.Instrument = Console.ReadLine();  
    Musicians.Add(musician);  
}  
...
```

Add will add the passed in musician object to the Musician's List

Error on musicians.Add

When we actually run our code we'll receive a `NullException` error on our `Add` method!



Band.cs

```
...
public void AddMusician()
{
    var musician = new Musician();
    Console.WriteLine("What is the name of the musician to be added?");
    musician.Name = Console.ReadLine();
    Console.WriteLine("What instrument does " + musician.Name + " play?");
    musician.Instrument = Console.ReadLine();
    Musicians.Add(musician);
}
...
```

If we look in the debugger we'll see Musicians is null!



ERROR: Object reference is not set to an instance of an object

Why is Musicians null? Shouldn't it be a List of Musician?

Null

Null is the absence of value. Not to be mistaken for zero as that is a value of nothing.

To the computer this says:

"You will be given a List of Musician; call it Musicians"

"Okay, Add musician to Musicians"



Band.cs

```
public class Band
{
    public List<Musician> Musicians;
    ...
    Musicians.Add(musician);
}
```

The computer responds:

"I don't understand! You only told me you were going to give me a list, you never actually gave it to me!"

Set a Default Value

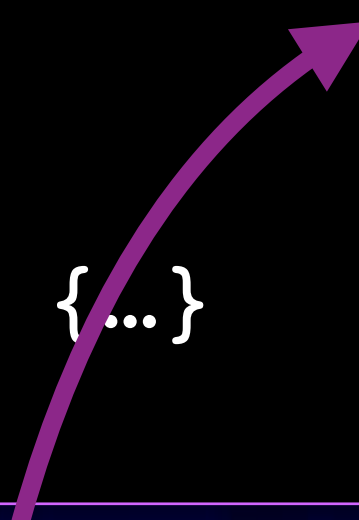
Set the Musicians variable to have a default value with `new List<Musician>()`.

Band.cs

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

public class Band
{
    public string Name;
    public List<Musician> Musicians = new List<Musician>();

    public void Announce() {...}
    public void AddMusician() {...}
}
```



This provides Musicians with initial empty List of Musician to work with

Musicians Internal Code Working

We added the underlying code for Musicians, we'll cover loops in the next level to wire it up!

Our Application now contains:

- A List of Musicians per band
- A method to add a Musician to the List

Example Running AddMusician Method

What is the name of the musician to be added?

>>>

\$ Robert

"What instrument does Robert play?"

>>>

\$ Guitar

A Quick Recap on Groups of Objects

Groups of Objects come in a variety of types which have their own pros and cons.

- Arrays work best with strongly-typed objects of fixed number
- Lists work best with strongly-typed objects of unknown or varying number
- You must reference System.Collections.Generic to use Lists



Level 4

Loops

Wiring Up Musicians Logic

Our bands contain musicians, but we'll need to expand our code with loops to utilize them.

What our application will do:

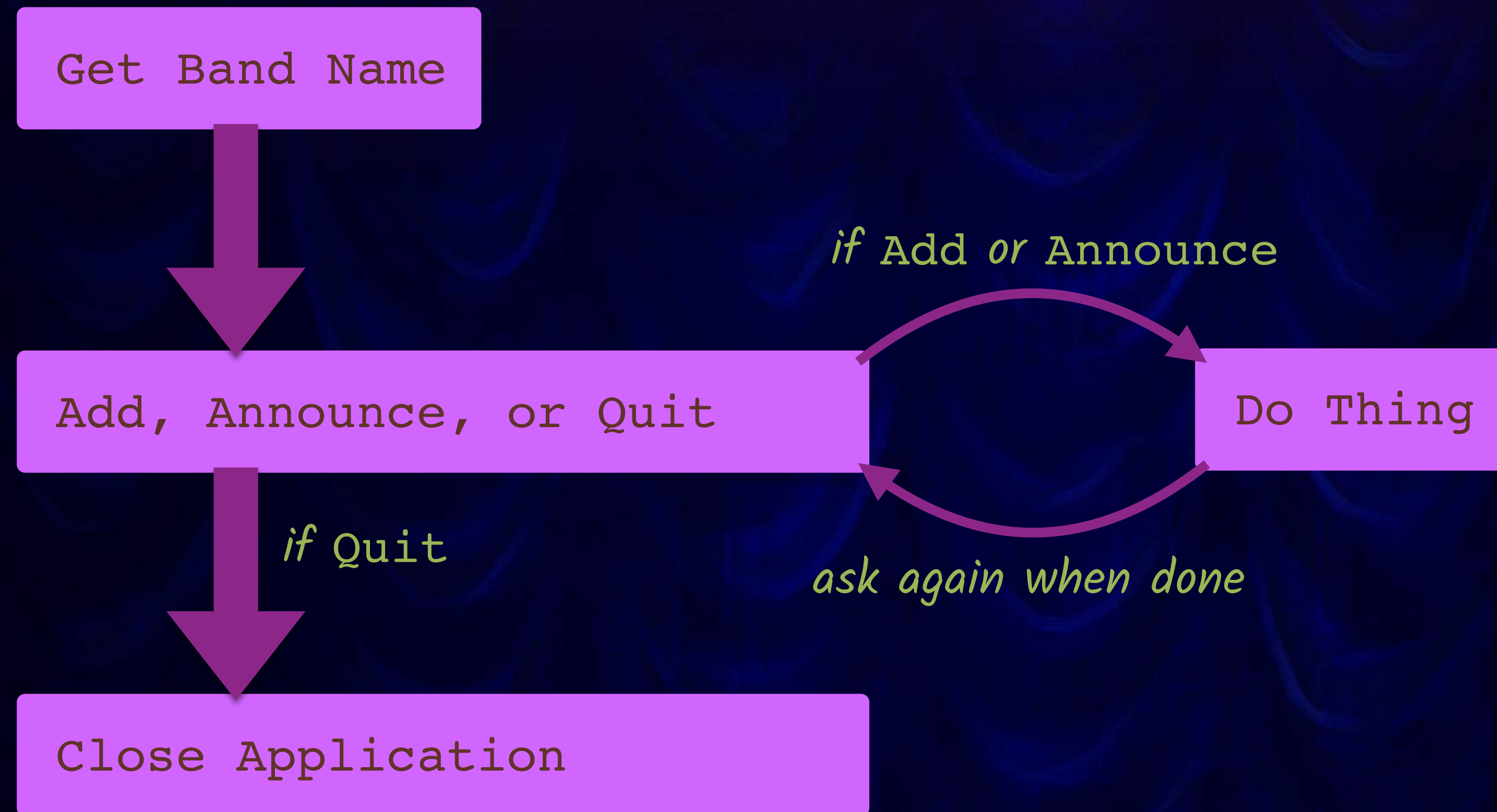
- Store information about a band and it's musicians
- Announce the band
- Announce the musicians

In this level:

- Create a Loop to add Musicians per band
- Create a Loop to announce musicians

Our Program Flow

Our application will get the band name, then ask to add a musician, announce the band, or quit.



How do we implement repeating to ask until Quit is used?

Add Command List

Before we create a loop, we'll inform the user of the accepted commands.

Program.cs

```
...
static void Main(string[] args)
{
    Console.WriteLine("What is the name of your band?");
    Band band = new Band();
    band.Name = Console.ReadLine();

    Console.WriteLine("Type 'Add' to add a musician.");
    Console.WriteLine("Type 'Announce' to announce the band.");
    Console.WriteLine("Type 'Quit' to quit the application.");
}
...
```

Now we can start implementing our loop!

while Loop

A while loop will continue to run until the `break` keyword is used or it's condition is false.



Program.cs

```
...
static void Main(string[] args)
{
    ...
    var repeat = true;
    while(repeat)
    {
    }
}
...
```

A new variable repeat will be used in our while loop's condition

This will run forever until loop is false or the break keyword is used



Be careful using loops, the above example has no way to exit the loop creating what's known as an infinite loop

Looping Conditions

Add our conditions that will handle adding a musician, announcing the band, or quitting the application.

Program.cs

```
while(repeat)
{
    Console.WriteLine("Add, Announce, or Quit?");
    var action = Console.ReadLine();
    if(action == "Add") {...}
    else if(action == "Announce") {...}
    else if(action == "Quit") {...}
    else
    {
        Console.WriteLine(action + " is not a valid command");
    }
}
```

*We'll also handle when the input
doesn't match any of our commands*

Add and Announce Commands

Add the appropriate calls to AddMusician **and** Announce methods from Band.

Program.cs

```
...  
while(repeat)  
{  
    ...  
    if(action == "Add")  
    {  
        band.AddMusician();  
    }  
    else if(action == "Announce")  
    {  
        band.Announce();  
    }  
    ...  
}
```

← AddMusician will be run when "Add" is entered

← Announce will be run when "Announce" is entered

break keyword

The break keyword escapes the loop at the point it's called.

Program.cs

```
...  
while(repeat)  
{  
    Console.WriteLine("Add, Announce, or Quit?");  
    ...  
    else if(action == "Quit")  
    {  
        break;  
    }  
    ...  
}  
...
```

The loop will exit at this point skipping any remaining code in the loop

What would this look like if we changed the while condition instead of using break?

Alternative: Escape Using while Condition

When the while condition is false, the loop will escape upon reaching the end of the while block.

Program.cs

```
...  
var repeat = true;  
while(repeat)  
{  
    Console.WriteLine("Add, Announce, or Quit?");  
    ...  
    else if(action == "Quit")  
    {  
        repeat = false;  
    }  
    ...  
}  
...
```

*When loop is set to false, the while loop
will escape once it finishes it's current loop*

With that done we need to update our Announce method to include our musicians

Foreach Loop

A foreach loop iterates through a group of objects one by one and runs code for each item.

Band.cs

```
...  
void Announce()  
{  
    Console.WriteLine("Welcome " + Name + " to the stage!");  
  
    foreach(var musician in Musicians)  
    {  
        musician.Announce();  
    }  
}  
...
```

foreach will loop through each musician and in Musicians and run their Announce method



foreach will run until it's run on every item in a group of objects, an unhandled exception is thrown, or the break keyword is used

Our Working Application

Our application now allows us to do everything we set out to do when we started.

Application features include:

- Stores information about a band and it's musicians
- Announces the band
- Announces the musicians

Our Running Application

Our loops will allow users to repeat actions until they use the Quit command.

```
Add, Announce, or Quit?
```

```
>>>
```

```
$ Add
```

```
What is the name of the musician to be added?
```

```
>>>
```

```
$ Robert
```

```
What instrument does Robert play?
```

```
>>>
```

```
$ Guitar
```

```
Add, Announce, or Quit?
```

```
>>>
```

```
$ Quit
```


A Quick Recap on Loops

Loops allow us to repeat code logically without rewriting it again and again.

- All loops escape immediately when the break keyword is used
- Always make sure there is a way to escape the loop!
(Infinite loops can be really bad)
- while loops will escape before running the first line in their block when their condition is false
- foreach loops will escape when they've run their code for every item in the collection

The background is a dark, atmospheric illustration. It features a stone archway or doorway. On the left side of the arch, there is a glowing yellow light source, possibly a lantern or a small fire, which casts a warm glow. The interior of the archway is dimly lit, showing silhouettes of several figures. One figure appears to be holding a long staff or pole. The overall color palette is dominated by dark blues, purples, and greys, with the yellow light providing a strong contrast. The entire image is framed by a decorative, ornate border.

Level 5

Method Overloads

Expanding Our Add Command

Instead of going through all the dialog, let's allow the user to add a Musician in one command.

Allow our Add to be used in the following ways:

- Add - Provides a step-by-step way to add a musician
- Add Name Instrument - Adds a musician with the provided Name and Instrument

Creating A New AddMusician Method

Our new AddMusician method accepts two strings and adds our Musician.

Band.cs

```
...
public void AddMusician() {...}

public void AddMusician(string name, string instrument)
{
    var musician = new Musician();
    musician.Name = name;
    musician.Instrument = instrument;
    Musicians.Add(musician);
}
...
```

*Hang on a sec... Won't having two **AddMusician** methods cause problems?*

Method Names Are Reusable

You can reuse Method Names so long as their Method Signatures are different.

Band.cs

```
...  
public void AddMusician() {...}  
  
public void AddMusician(string name, string instrument)  
{  
    var musician = new Musician();  
    musician.Name = name;  
    musician.Instrument = instrument;  
    Musicians.Add(musician);  
    Console.WriteLine(musician.Name + " was added.");  
}  
...
```

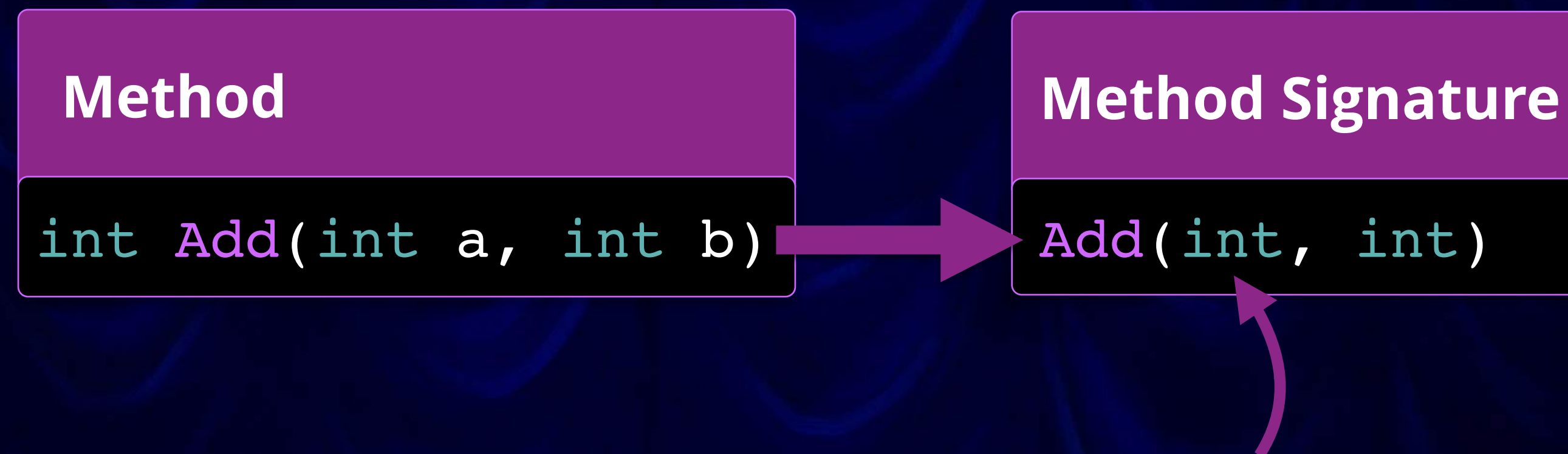
Valid because signatures are different

*Reusing a Method Name with a different
Method Signature is called a Method Overload*

What exactly is a Method Signature?

Method Signatures

Method Signatures consist of the **Method Name** and **Parameter Types**.

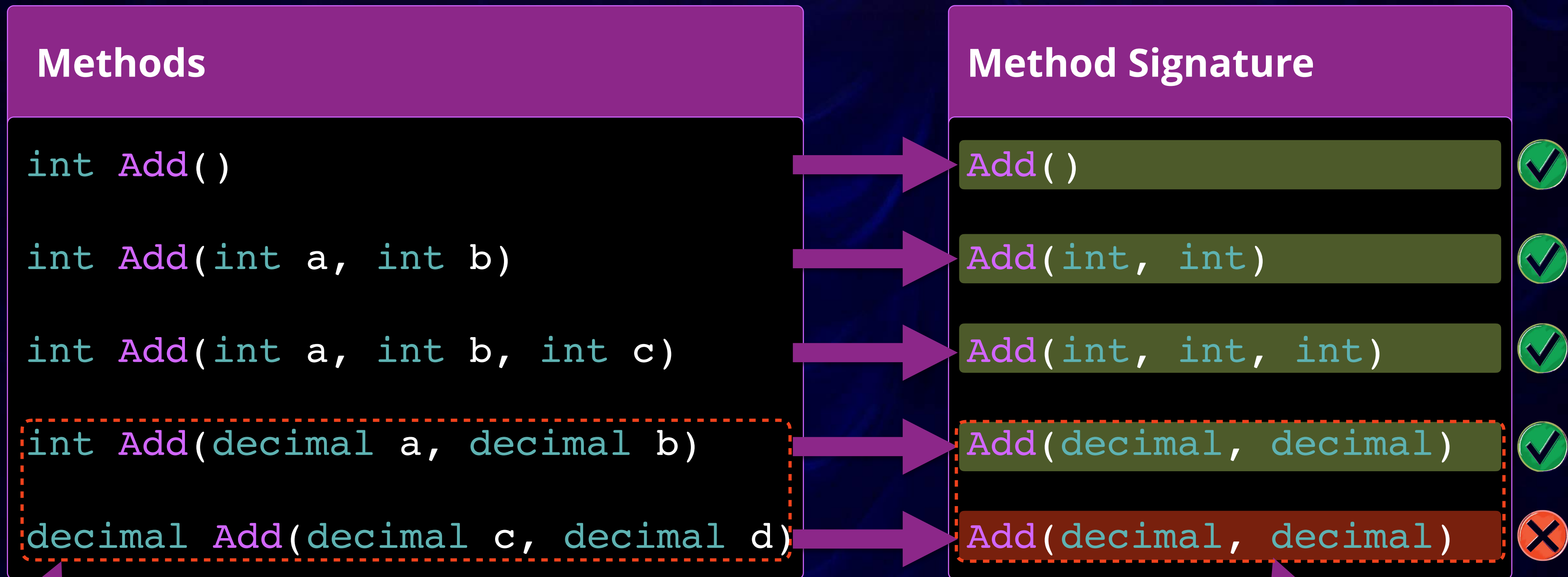


We call methods using their signature, which is why signatures must be unique

Let's look at some examples of non-conflicting and conflicting signatures

Method Signature Conflicts

Method Signatures are what we use to call our methods and determine duplication.



These conflict because Return Type is NOT part of a Method Signature

Okay back to our code!

Code Smells

Looking at our AddMusician methods, there's duplication here, and duplication stinks...

AddMusician()

```
var musician = new Musician();  
...  
musician.Name = ...  
...  
musician.Instrument = ...  
Musicians.Add(musician);
```

AddMusician(string, string)

```
var musician = new Musician();  
musician.Name = ...  
musician.Instrument = ...  
Musicians.Add(musician);  
...
```

These four lines effectively do the same thing!

In the event we change a parameter name, the name of our Musicians list, etc we will need to correct this in multiple places...

How can we clean this up to reduce duplication?


Refactor AddMusician()

We can refactor AddMusician() to utilize the AddMusician(string, string) method.

Band.cs

```
...  
public void AddMusician()  
{  
    var musician = new Musician();  
    Console.WriteLine("What is the name of the musician to be added?");  
    musician.Name = Console.ReadLine();  
    Console.WriteLine("What instrument does " + musician.Name + " play?");  
    musician.Instrument = Console.ReadLine();  
    Musicians.Add(musician);  
}  
  
public void AddMusician(string name, string instrument){...}  
...
```

*Our other method instantiates a **Musician** object, so let's get rid of this one*



Correct Variable Use

With **musician** gone we need declare variables for the **Name** and **Instrument**.



Band.cs

```
...  
public void AddMusician()  
{  
    Console.WriteLine("What is the name of the musician to be added?");  
    musician.Name = Console.ReadLine();  
    Console.WriteLine("What instrument does " + musician.Name + " play?");  
    musician.Instrument = Console.ReadLine();  
    Musicians.Add(musician);  
}
```

*These are now invalid assignments since **musician** no longer exists*

```
public void AddMusician(string name, string instrument){...}  
...
```


Call Our New AddMusician Method

Remove our call to `Musicians.Add` and instead call our new `AddMusician` method.



Band.cs

```
...  
public void AddMusician()  
{  
    Console.WriteLine("What is the name of the musician to be added?");  
    var name = Console.ReadLine();  
    Console.WriteLine("What instrument does " + name + " play?");  
    var instrument = Console.ReadLine();  
    Musicians.Add(musician);  
}  
  
public void AddMusician(string name, string instrument){...}  
...
```

*We can switch this line to call our other **AddMusician** method*

Refactor Complete

We've now eliminated the duplication between our two AddMusician methods.



Band.cs

```
...
public void AddMusician()
{
    Console.WriteLine("What is the name of the musician to be added?");
    var name = Console.ReadLine();
    Console.WriteLine("What instrument does " + name + " play?");
    var instrument = Console.ReadLine();
    AddMusician(name, instrument); AddMusician() collects the Name and Instrument
}                                before utilizing AddMusician(string, string) to
                                add the Musician
public void AddMusician(string name, string instrument){...}
...
```

Now we'll create our new **Add** command to directly access **AddMusician(string, string)**


Wiring Up Our New AddMusician Method

We now need to add the logic to have "Add Name Instrument" call our new AddMusician method.

Program.cs

```
...  
while(repeat)  
{  
    ...  
    if(action == "Add"){...}  
    else if(action.StartsWith("Add"))  
    {  
        ...  
    }  
    ...  
}
```

*This else if condition uses **StartsWith** to check if the string action starts with the string "Add"*



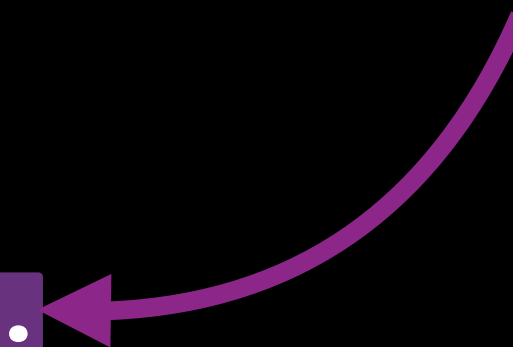
Wiring Up AddMusician(string, string)

We now need to add the logic to have "Add Name Instrument" call our new AddMusician method.

Program.cs

```
...
while(repeat)
{
    ...
    if(action == "Add")
        band.AddMusician();
    else if(action.StartsWith("Add"))
    {
        var arguments = action.Split(' ');
    }
    ...
}
```

*Split will create an array of separate strings
split where the provided character is used*



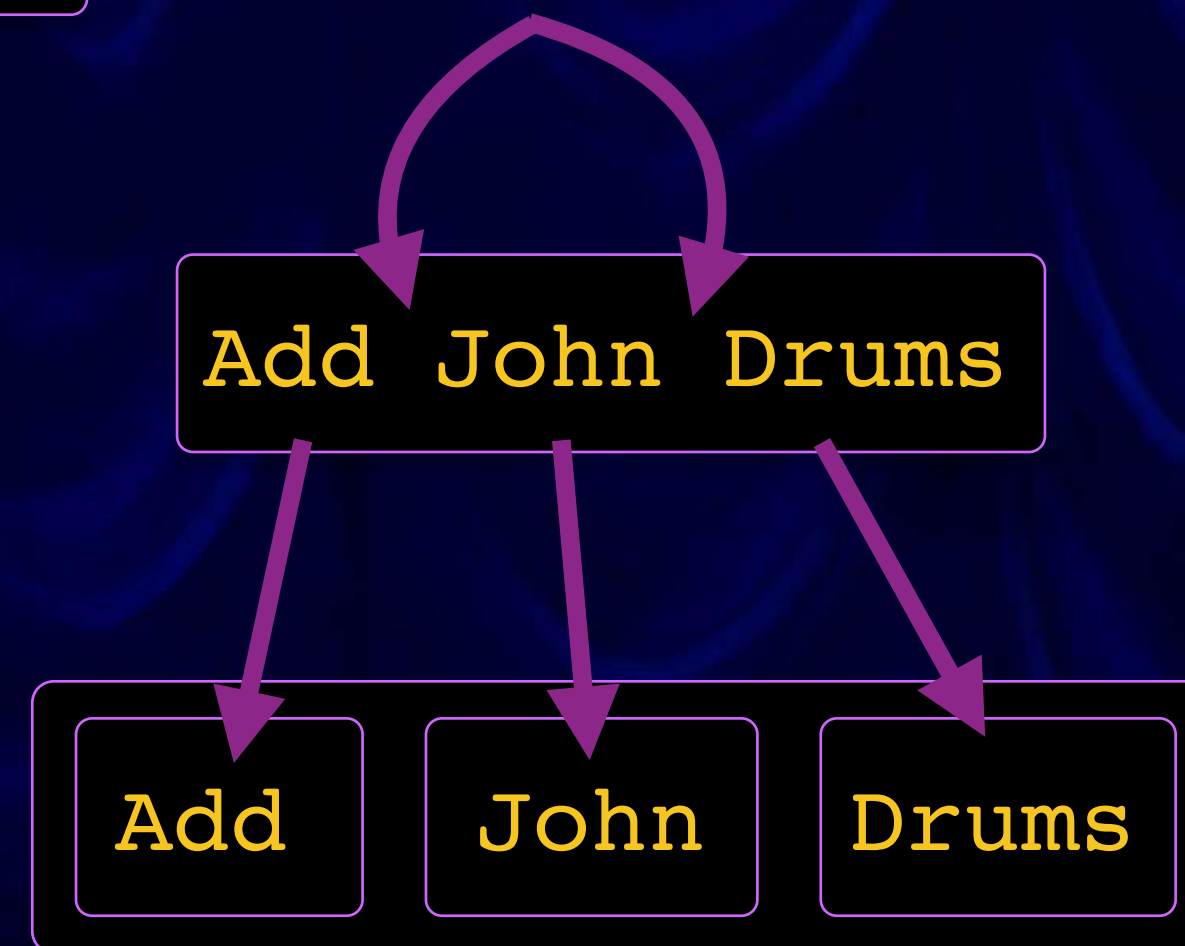
Understanding split

The `split` method creates an array of strings by splitting a string on a provided character.

Example Code

```
action = "Add John Drums";  
action.split(' ');
```

`split` breaks the string along the provided ' ' character



The resulting set of strings are then returned as an array of strings

Wiring Up AddMusician(string, string)

We now need to add the logic to have "Add Name Instrument" call our new AddMusician method.

Program.cs

```
if(action == "Add"){...}  
else if(action.StartsWith("Add"))  
{  
    var arguments = action.Split(' ');  
    if(arguments.Length == 3)  
    {  
        band.AddMusician(arguments[1], arguments[2])  
    }  
    else  
    {  
        band.AddMusician();  
    }  
}
```

If we get an array with a Length of 3 call our new AddMusician method

If the array didn't have a Length of 3 something wasn't entered right fallback to our AddMusician() method

Our Direct Add Command Now Works

Our users can now choose to go step by step, or skip the dialog and directly add musicians.

Add now works in the following ways:

- Add - Provides a step-by-step way to add a musician
- Add Name Instrument - Adds a musician with the provided Name and Instrument

Example Step by Step AddMusician

The step by step route provides a verbose instructions for users to add musicians.

Add, Announce, or Quit?

>>>

\$ Add

What is the name of the musician to be added?

>>>

\$ Robert

What instrument does Robert play?

>>>

\$ Guitar

Robert was added.

Example Directly AddMusician

The direct route allows users to enter musicians far more efficiently

Add, Announce, or Quit?

>>>

\$ Add Robert Guitar

Robert was added.

A Quick Recap on Method Overloads & Signatures

Method Overloads help with code clarity and avoid duplication

- **Method Signatures** consist of the method's **Name** and **Parameter Types**
- **Return Type** is **NOT** part of a method's signature
- Methods within a class may share the same **Name**, but cannot have the same **Signature**