
KEEPING IT
CLASSY
WITH
C#

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?*

KEEPING IT CLASSY WITH C#

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**?*

KEEPING IT CLASSY WITH C#

Method Signatures

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

Method

```
int Add(int a, int b)
```

Method Signature

```
Add(int, int)
```

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.

Methods	Method Signature	
`int Add()`	`Add()`	✓
`int Add(int a, int b)`	`Add(int, int)`	✓
`int Add(int a, int b, int c)`	`Add(int, int, int)`	✓
`int Add(decimal a, decimal b)`	`Add(decimal, decimal)`	✓
`decimal Add(decimal c, decimal d)`	`Add(decimal, decimal)`	✗

*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?

KEEPING IT CLASSY WITH C#

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);
}

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

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

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);
}

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

`AddMusician()` *collects the* ***Name*** *and* ***Instrument*** *before utilizing* `AddMusician(string, string)` *to add the* ***Musician***

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*

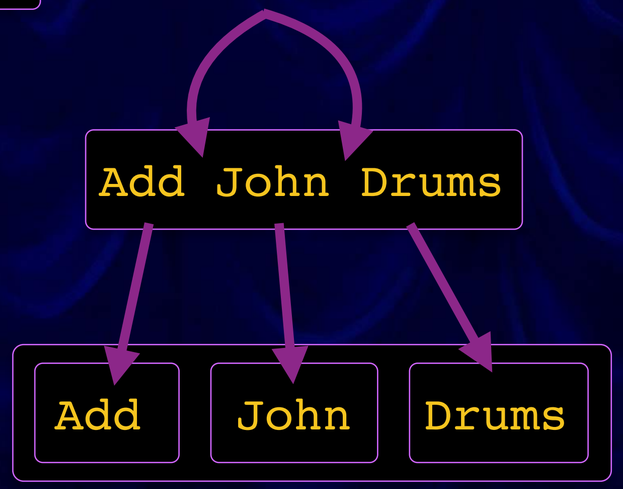
Understanding split

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.
```

KEEPING IT CLASSY WITH C#

Example Directly AddMusician

The direct route allows users to enter musicians far more efficiently

```
Add, Announce, or Quit?
>>>  $ Add Robert Guitar
Robert was added.
```

KEEPING IT CLASSY WITH C#

A Quick Recap on Method Overloads & Signatures

Method Overloads help with code clarity and avoid duplication

- **Method Signature**s 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**