Level 1

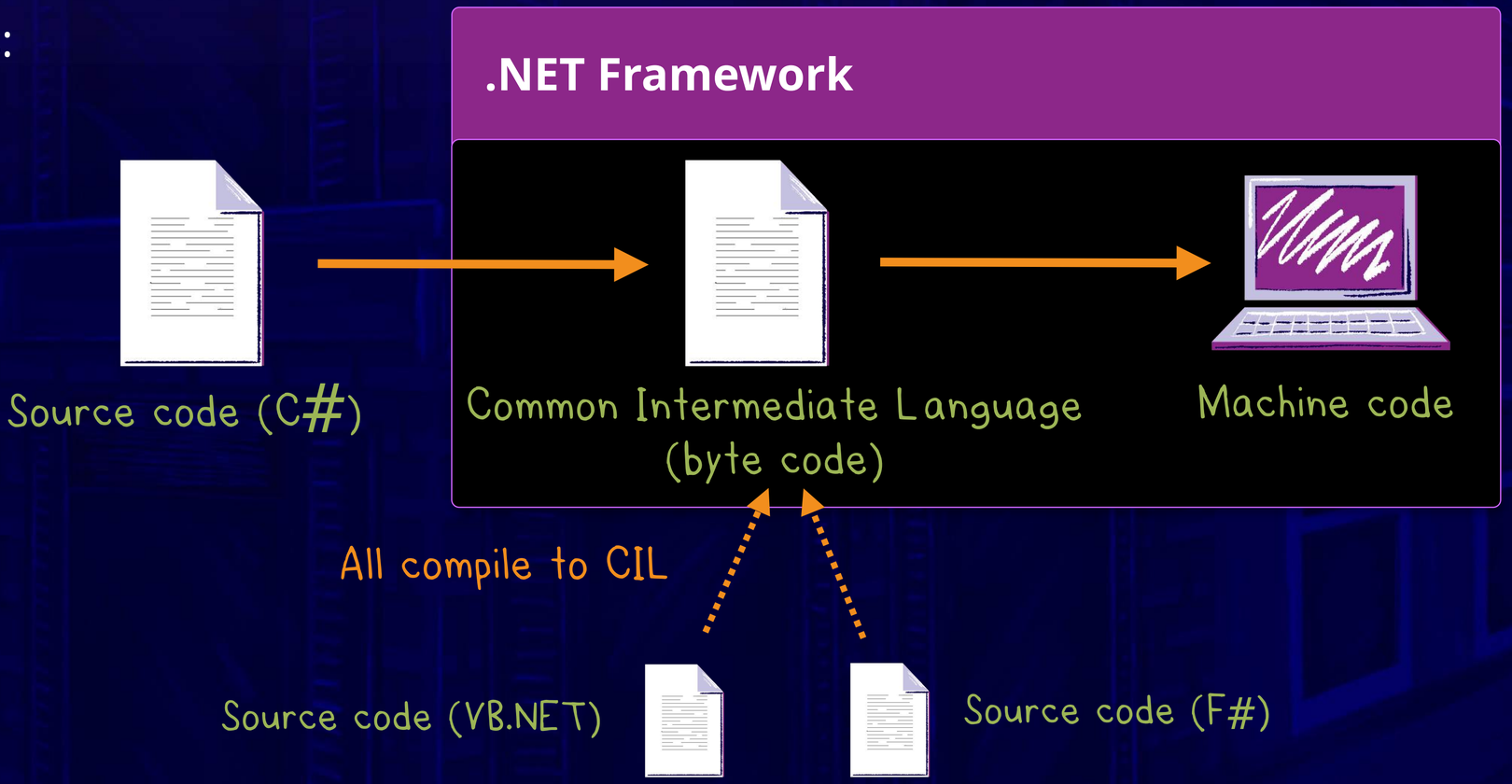
Introduction

What Is C#?

C# is a general purpose object-oriented programming language released in 2002 by Microsoft.

Some notable characteristics:

- Compiled
- Strongly typed
- .NET language



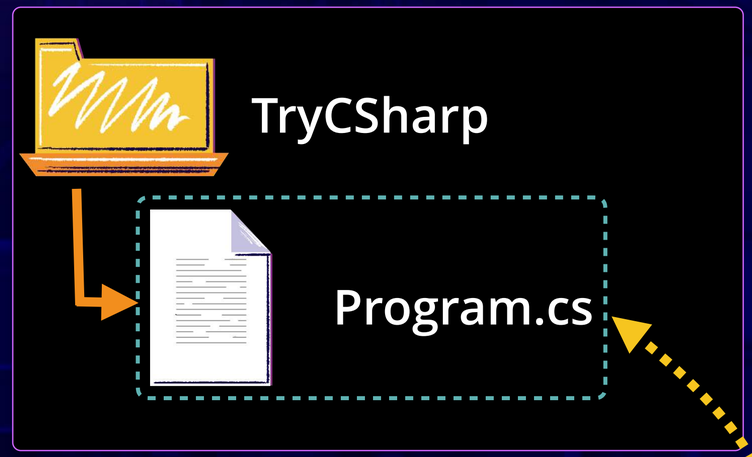
TRY C#

Creating a New C# Application

We can use the dotnet new console **command to create a C# application with a** Program.cs **file.**

Console

```
>>> $ dotnet new console
The template "Console Application"
created successfully.
```



All modern .NET applications start in the Program.cs file, so we'll start there

For more information on installation, visit go.codeschool.com/install-dot-net

TRY C#

The Program.cs File

This file is the entry point of our application. It's generated with the following code:

Program.cs

```
using System;

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

Classes allow us to separate our code into "objects"

Methods contain the executable code of our object

Start of an Application

When our application is run, execution starts from the Main() method.

Program.cs

```
using System;

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

This is the first line of code executed

Restoring Dependencies

Before we can run our application we need to use dotnet restore **to restore our dependencies**

Program.cs

```
using System;

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

dotnet restore needs to be run before you run the application the first time or anytime you change a dependency

Console

```
>>> $ dotnet restore
Restoring packages for TryCSharp…
```

Running the Application for the First Time

When we run the application, it prints “Hello World!” to the console.

Program.cs

```
using System;

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

Strings are a collection of characters wrapped in double quotes

Console

```
>>> $ dotnet run
Hello World!
```

Demo Application

Let’s make our existing application read input from the user and use that as part of our output.

We’ll need two things:

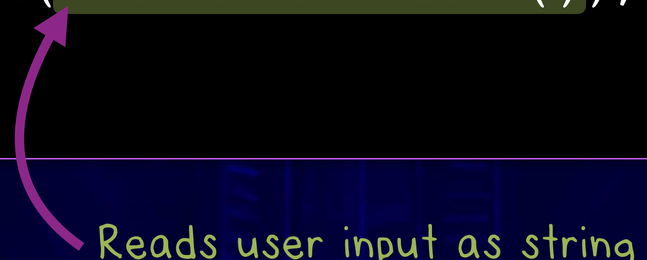
- Accept user input
- Concatenate strings

Reading User Input

The Console.ReadLine **method reads user input from the console line and returns it as a** string**.**

Program.cs

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Type a message");
        Console.WriteLine(Console.ReadLine());
    }
}
```

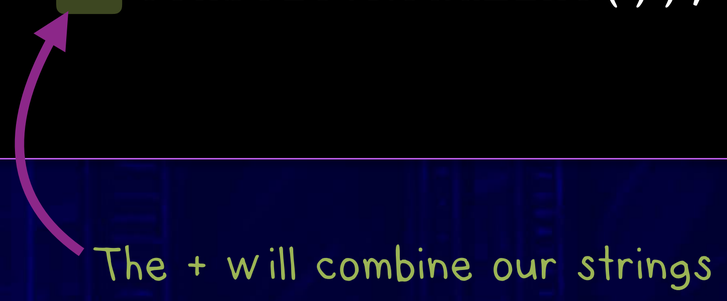


String Concatenation

We can use the + character to concatenate multiple strings.

Program.cs

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Type a message");
        Console.WriteLine("You said " + Console.ReadLine());
    }
}
```



Running the Demo Application

We can use the dotnet run **command to run our program.**

Program.cs

```
using System;

class Program
{
  ...
}
```

Behind the Scenes of String Concatenation

This is what happens behind the scenes when using user input from Console.ReadLine**.**

Step 1.

```
Console.WriteLine("You said " + Console.ReadLine());
```

User input is read

Step 2.

```
Console.WriteLine("You said " + "Hello World");
```

Strings are combined

Step 3.

```
Console.WriteLine("You said Hello World");
```

Quick Recap on Getting Started

We can use the dotnet **commands to create, compile, and run applications.**

```
>>>  $ dotnet new console
The template "Console Application"
created successfully.
```

Creates a new application

```
>>>  $ dotnet run
Type a message
>>>  $ Hello World
You said Hello World
```

Runs the application


Level 2
Variables

Collecting Band Information

Let's make our console application more useful and have it accept band information.

- Ask for **name of band**
- Ask for **number of members**
- Announce **name of band** and **number of members**

We ask for the name of the band and number of members in one place...

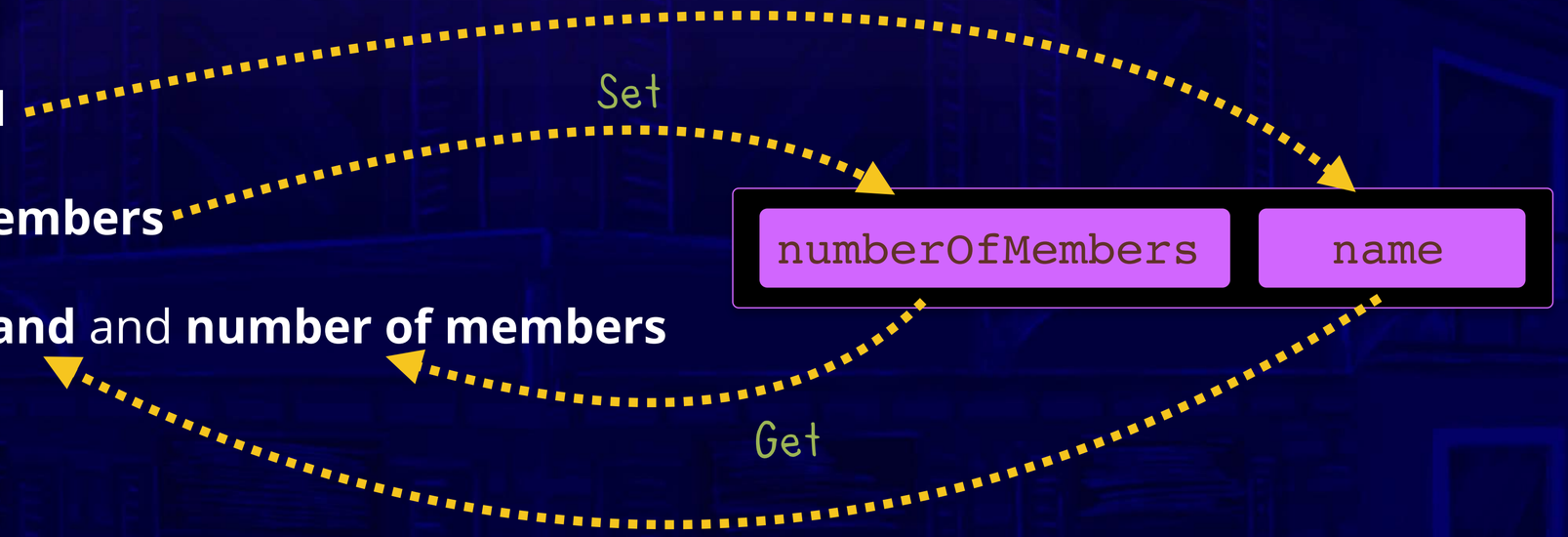
...but use that information someplace else.

TRY C#

Storing Information in Variables

Variables allow us to store information in memory for later use.

- Ask for **name of band**
- Ask for **number of members**
- Announce **name of band** and **number of members**



This saving and retrieving variables is known as "setting" and "getting"

TRY C#

Adding our Application's Messages

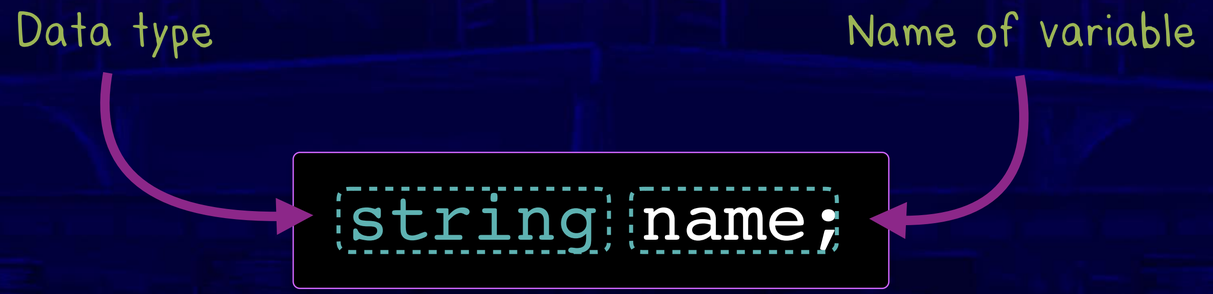
We'll first add all the things we expect to write to our users.

Program.cs

```
…
static void Main(string[] args)
{
  Console.WriteLine("What is the name your band?");

  Console.WriteLine("How many people are in your band?");

  Console.WriteLine("_________ has ___________ members.");
}
…
```

Declaring Variables

To declare a variable, you specify the data type and then what you'd like to name the variable.

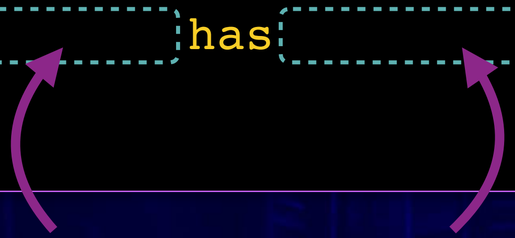


Declaring Our Variable & Setting It With ReadLine

We can set a variable using '=', which sets the variable on the left to the value being assigned on the right.

Program.cs

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

  Console.WriteLine("How many people are in your band?");

  Console.WriteLine("          has                 members.");
}
…
```



We Can Add Our Variable to Our Output

To use a variable, we just enter its name wherever we want to use it.

Program.cs

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

  Console.WriteLine("How many people are in your band?");

  Console.WriteLine(name + " has ___________ members.");
}
…
```

We need to use string concatenation to combine the variable and hard-coded strings

Data Types & Errors

The Console.ReadLine **method returns a** string**, but** numberOfMembers **is expecting an** int**!**

Program.cs

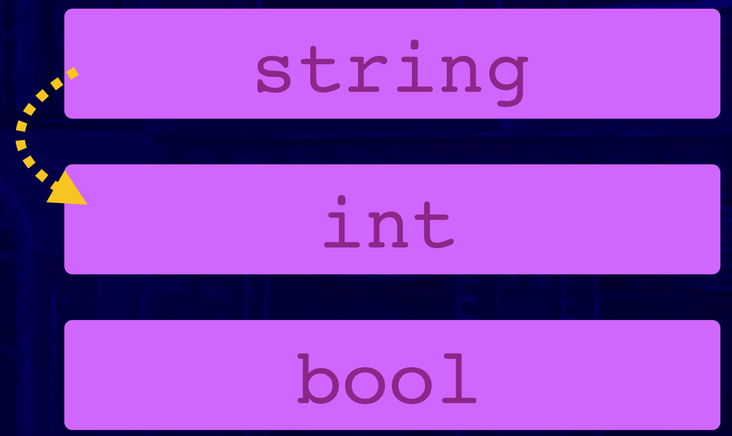
```
…
static void Main(string[] args)
{
  Console.WriteLine("What is the name your band?");
  string name = Console.ReadLine();

  Console.WriteLine("How many people are in your band?");
  int numberOfMembers = Console.ReadLine();

  Console.WriteLine(name + " has ___________ members.");
}
…
```

```
Cannot implicitly convert `string` to `int`
```

Data Types

There are 15 built-in data types in C#, but we're going to focus on string**,** int**, and** bool **for now.**

We can convert String to Int, but
we need to do so explicitly.



Sequence of characters *("Robert", "Cat", "The List", "!Do1*&")*

Whole numbers *(-1, 0, 1, 2)*

true **/** false **(values are lowercase)**

Converting Our string to an int

The int.Parse method allows us to convert our string into an int, fixing our error.

Program.cs

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

  Console.WriteLine("How many people are in your band?");
  int numberOfMembers = int.Parse(Console.ReadLine());

  Console.WriteLine(name + " has ___________ members.");
}
…
```

Coverts a string to an int

Adding Our numberOfMembers to Our Message

Since our string is between double quotes, we'll need to break it up to add our variable.

Program.cs

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

  Console.WriteLine("How many people are in your band?");
  int numberOfMembers = int.Parse(Console.ReadLine());

  Console.WriteLine(name + " has " + numberOfMembers + " members.");
}
…
```

numberOfMembers is an int, but we do not need to explicitly convert it to a string. Most data types can implicitly be converted to a string.

The Application — Now With Variables

We have made the application able to store and retrieve values provided by the user.

```
>>>  $ dotnet run
     What is the name of your band?
>>>  $ Awesome Inc
     How many people are in your band?
>>>  $ 3
     Awesome Inc has 3 members.
```

Ask for name of band

Ask for number of band members

Announce name of band and number of members

Quick Recap on Variables & Data Types

Variables are used to store values in memory for later use.

- Storing a value in a variable is *setting* it
- Retrieving a value from a variable is *getting* it
- Set a variable by using the = character followed by the desired value
- Get a variable by using the variable's name
- Not all data types can be converted implicitly (in many cases we'll have built-in Parse methods)


Level 3
Conditions

Handling Invalid User Input

When users enter something we can't parse, the application will break.

```
>>> $ dotnet run
What is the name of your band?
>>> $ Awesome Inc
How many people are in your band?
>>> $ Duck
Unhandled Exception:
System.FormatException: Input
string was not in correct format.
```

Our app is only expecting numbers here

Our Input Isn't Safe

Here, the Parse method is throwing an error because it can't covert the string to an int.

Program.cs

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

  Console.WriteLine("How many people are in your band?");
  int numberOfMembers = int.Parse(Console.ReadLine());

  Console.WriteLine(name + " has " + numberOfMembers + " members.");
}
…
```

```
Input string was not in correct format.
```

TryParse **Lets Us Safely Parse Variables**

The int.TryParse **method will return** true **or** false **depending on if it was able to parse the** string**.**

Program.cs

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

  Console.WriteLine("How many people are in your band?");
  int numberOfMembers = 0;
  int.TryParse(Console.ReadLine(),                    );

  Console.WriteLine(name + " has " + numberOfMembers + " members.");
}
…
```

Second parameter is what we are going to output the parsed value to

First parameter is what we want to parse

Output Parameter

Some methods use “output parameters” that allow it to set variables using the out keyword.

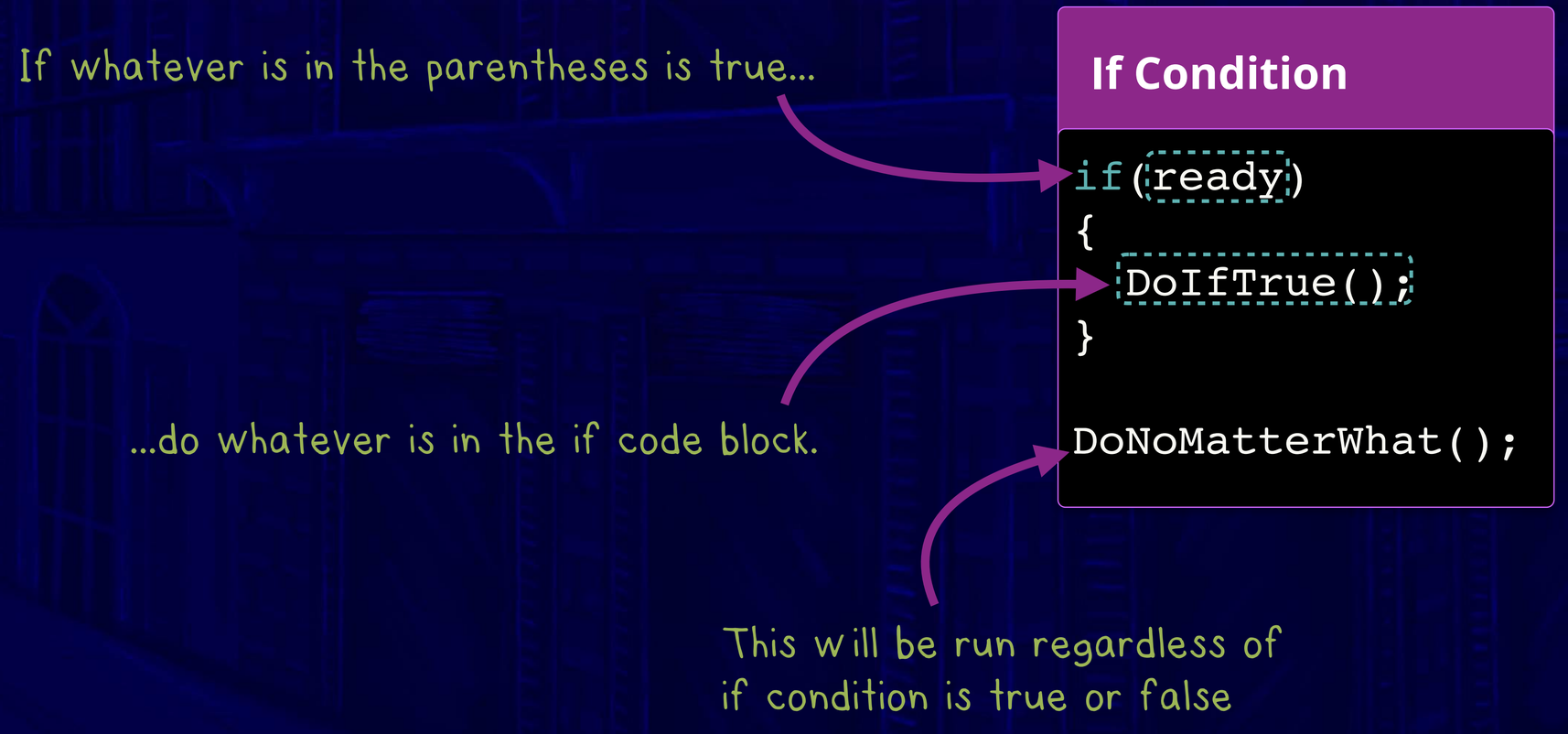
Program.cs

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

  Console.WriteLine("How many people are in your band?");
  int numberOfMembers = 0;
  int.TryParse(Console.ReadLine(), out numberOfMembers);

  Console.WriteLine(name + " has " + numberOfMembers + " members.");
}
…
```

numberOfMembers will be set if parsing is successful

Conditions

Conditions allow us to change our application's behavior based on specific circumstances.



Examples of Results

Depending on if ready **is** true **or** false **determines what code will get executed.**

If Condition

```
if(ready)
{
  DoIfTrue();
}

DoNoMatterWhat();
```

ready **is** *true*

```
DoIfTrue();
DoNoMatterWhat();
```

ready **is** *false*

```
DoNoMatterWhat();
```

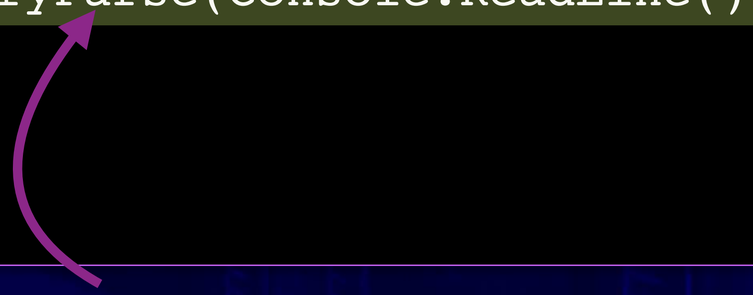
if ready is true, then we will execute both DoIfTrue and DoNoMatterWhat methods –
but if ready is false, we will skip the DoIfTrue method in the if block

int.TryParse **Returns** true **or** false

In the event the value could be parsed, TryParse **is** true — **otherwise, it's** false**.**

Program.cs

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

  Console.WriteLine("How many people are in your band?");
  int numberOfMembers = 0;
  if(int.TryParse(Console.ReadLine(), out numberOfMembers))
  {

  }
…
```



Returns true if it can parse the first parameter – otherwise, returns false

Not "!" Expression

Since we only want to change our behavior when TryParse **fails, we can use** Not (!)**.**

Program.cs

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

  Console.WriteLine("How many people are in your band?");
  int numberOfMembers = 0;
  if(!int.TryParse(Console.ReadLine(), out numberOfMembers))
  {

  }
…
```

This is effectively saying "if TryParse is NOT true" then...

The Environment.Exit Method

The Environment.Exit **method immediately exits the program.**

Program.cs

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

  Console.WriteLine("How many people are in your band?");
  int numberOfMembers = 0;
  if(!int.TryParse(Console.ReadLine(), out numberOfMembers))
  {
    Console.WriteLine("input was not valid");
    Environment.Exit(0);
  }
…
```

Providing a 0 here means the application will close saying it ran successfully

Invalid Input Handled

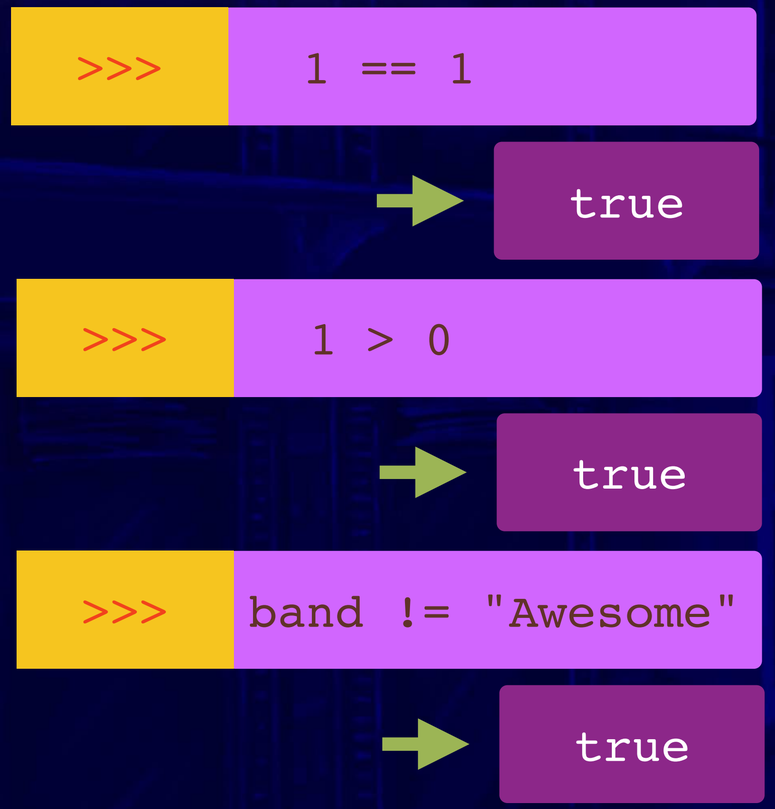
Now when something invalid is entered, they'll get a simple user-friendly message.

```
>>> $ dotnet run
What is the name of your band?
>>> $ Awesome Inc
How many people are in your band?
>>> $ Duck
input was not valid
```

Expressions

We often need to compare two things for a condition in an application.

- Do we have 1 band member?
- Do we have more than 0 band members?
- Is our band not named “Awesome”?



else **Conditions**

The else **condition executes a block of code if a condition is** false**.**

If whatever is in the parentheses is false...

...do whatever is in the else code block.

If Else Conditions

```
if(ready)
{
  DoIfTrue();
}
else
{
  DoIfFalse();
}
```

Series of Conditions

You can use else if **to create a series of conditions.**

If ready is true, do this...

If ready is false and the band is named "Awesome", do this...

If ready is false and the band is not named "Awesome", do this...

Series of Conditions

```
if(ready)
{
  DoIfTrue();
}
else if(name == "Awesome")
{
  DoFalseAndAwesome();
}
else
{
  DoIfFalseAndNotAwesome();
}
```

Declaring Our Band Type Using Conditions

We can use if, else if, **and** else **to write our type of band to the console.**

Program.cs

```
…
if(numberOfMembers < 1)
{
  Console.WriteLine("You must have at least 1 member");
  Environment.Exit(0);
}
else if(numberOfMembers == 1)
  Console.WriteLine(name + " is a solo");
else if(numberOfMembers == 2)
  Console.WriteLine(name + " is a duo");
else
  Console.WriteLine(name + " has " + numberOfMembers + " members");
…
```

Curly braces not required here

Curly braces are optional when you only have one line of code in a code block

The Application Is Complete

Our application is now more robust and ready to handle different information.
We've added two features:

- Printing the type of band (solo, duo, etc.) based on number of members
- Handling invalid user input

```
>>> $ dotnet run
What is the name of your band?
>>> $ Awesome Inc
How many people are in your band?
>>> $ 2
Awesome Inc is a duo
```

Quick Recap on Conditions & Expressions

We can change the flow of our code using conditions and expressions.

- The *if* statement only executes its block when the condition is *true.*
- The *else* statement only executes its block when the *if* condition is *false.*

Expressions	
==	is equal to
!=	is **NOT** equal to
!	before a condition passes the condition when it's **NOT** *true*