# Intermediate C# Labs

## Table of Contents

* [First Lab](#first-lab)
* [Literal Strings](#literal-strings)
* [Tuples](#tuples)
* [Records](#records)
* [Deconstruction](#deconstruction)
* [Exceptions](#exceptions)
* [Events](#events)
* [Lambda](#lambda)
* [Inheritance](#inheritance)
* [Interfaces](#interfaces)
* [LInQ API](#linq-api)
* [Linq Query](#linq-query)
* [Attributes & Reflection](#attributes--reflection)
* [Collections](#collections)

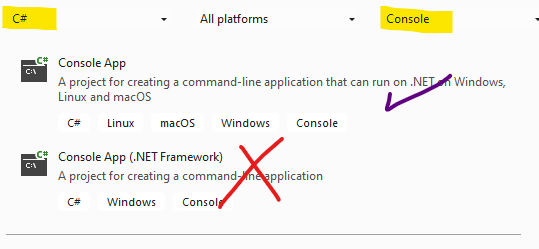
# First Lab

## Overview

In this lab, we will create a console application and a simple class and test code

|  |  |
| --- | --- |
| Exercise Folder | FirstLab |
| Builds On | None |
| Time to complete | 20 minutes |

## In this lab, we will create a console application and a simple class and test code

1. Open Visual Studio Community 2022 and choose *Create new Project*
2. In the filters, choose *C#* and *Console* 
3. Choose a *Console* application (note - there are 2 project types. .Net Core has multiple platforms while .Net framework is Windows only. Choose .Net Core projects)
4. Name your project *FirstLab*
5. Change the name of the solution to *Labs*
6. Change the location of the project to whatever folder works best
7. Right-click on the new project in the *Solution Explorer* and choose *Add/Class*
8. Name the class **Employee**

## Class Design

* Change the class to public
* Remove the unneeded *usings*
* Create fields
  + firstName
  + lastName
  + salary
  + TAX\_RATE (const)
* Create Properties
  + FirstName
  + LastName
  + Salary
  + YtdGrossPay (auto property, private set)
  + HireDate (auto property, **init** instead of **set**)
* Pay Method - calculate taxes, return net. Increase YtdGrossPay
* Constructor - accept name, salary, and hiredate arguments

## Main

In *Program.cs* in top-level code, create a sample Employee. Set/Get properties Pay the employee

# Literal Strings

## Overview

In this lab, we demonstrate string literals.

|  |  |
| --- | --- |
|  |  |
| Exercise Folder | LiteralStrings |
| Builds On | [FirstLab](../FirstLab) |
| Time to complete | 10 minutes |

1. Override ***ToString*** in the *Employee* class
2. Return a formatted string containing all important employee information.
3. Use an interpolated literal string# Tuples

# Tuples

## Overview

This independent lab demonstrates various tuple features.

|  |  |
| --- | --- |
| Exercise Folder | Tuple |
| Builds On | None |
| Time to complete | 30 minutes |

## Instructions

1. Create a new console project (in your current solution), or add this Tuple.csproj to your solution.
   * Name the project *Tuple*
2. Create a public static class named *TemperatureConverter*
3. Add these functions
   * FromCelcius(double val)
   * FromFahrenheit(double val)
   * FromKelvin
4. Each function should return a tuple containing the values of the other temperature scales. For instance, *FromCelcius* should return the F° and K° values
5. In *Program.cs* demonstrate each function with several examples
6. Show both deconstructed values and the inbuilt tuple *ToString* method

### Formulas

To get K from C, add 273.15 to the C value  
To get F from C, F = (C x 9/5) + 32  
To get C from F, C = (F-32) X 5/9

# Records

## Overview

This lab demonstrates the **record** type specifier.

|  |  |
| --- | --- |
| Exercise Folder | Records |
| Builds On | Tuple |
| Time to complete | 10 minutes |

Building on the *Tuples* lab, define a single record type to use for the result of each temperature conversion function.

# Deconstruction

## Overview

In this lab, add deconstructors to the *Employee* class.

|  |  |
| --- | --- |
| Exercise Folder | Deconstruction |
| Builds On | LiteralStrings |
| Time to complete | 30 minutes |

## Steps

1. Add the following deconstructors to the *Employee* class.

public void Deconstruct (out string FullName, out double Salary,   
 out double YtdGross )  
public void Deconstruct(out string FirstName, out string LastName,   
 out double Salary,   
 out double YtdGross, out DateTime HireDate)

1. Update *Program.cs* to verify the deconstructors# xUnit Intro

## Overview

In this lab, we get started with xUnit.  
Using TDD process, we will define some new capabilities in the *Employee* class.

|  |  |
| --- | --- |
| Exercise Folder | xUnitIntro |
| Builds On | Deconstruction |
| Time to complete | 40 minutes |

## Steps

1. Create a new xUnit project in your solution Xunit
2. Copy *Employee.cs* from the last project into this project
3. Rename the initial file (UnitTest1.cs) to *EmployeeTest.cs* and change the class name to *EmployeeTest*

We want to define a new property in the Employee class

1. Tenure - a read-only int property that calculates the number of years the employee has been employed
2. Update the Pay() method as follows:
   * If the employee's tenure is >= 5 years, their pay is increased by 2% from their base salary
   * Create the test first! For this scenario, a **Theory** is probably the best choice.

[Theory]  
[InlineData(6, 100.0, 94.2)]  
[InlineData(0, 100.0, 92.35)]  
[InlineData(3, 200.0, 184.7)]  
public void TestPay(int tenure, double sal, double pay)  
{

# Exceptions

## Overview

In this lab, we will specify boundaries on the Employee properties and behaviors by specifying expected exceptions. This lab builds on the xUnit lab.

|  |  |
| --- | --- |
| Exercise Folder | Exceptions |
| Builds On | xUnitIntro |
| Time to complete | 30 minutes |

## Steps

1. Add tests that expect exceptions to verify the following rules
   * An employee's first and last name must be at least 1 character in length and a max of 30. Only alpha, spaces, dash, apostrophes allowed
   * An employee's salary must be between 50 and 500 inclusive
   * An employee's hire date must be in the past
2. Run the test to verify failure
3. Modify *Employee* to satisfy the tests# Generics Exercise

## Overview

In this code exercise, you will convert a regular class into a generic.

|  |  |
| --- | --- |
| Exercise Folder | Generics |
| Builds On | None |
| Time to complete | 20 minutes |

## Steps

1. Add Generics.csproj to your current solution
2. All tests should succeed
3. Review the code for the class ObjectArrayList .
4. Change the name of the class to GenericArrayList
5. Convert the class to a generic by lifting the object type from the class, replacing with a generic parameter
6. Update the test class to utilize the generic
7. Re-run tests# Delegates

|  |  |
| --- | --- |
| Exercise Folder | Delegates |
| Builds On | Exceptions |
| Time to complete | 30 minutes |

## Overview

Our customer has identified an additional requirement (imagine that!).

Some employees are in regions (state, province or city) that imposes an additional income tax.

In order to handle this scenario, the designer decided to add a delegate to the Employee class that can be called to obtain the additional local tax.

The signature of the delegate is double func(double amt). The gross amount being paid is passed in and the local tax is returned.

Here's what you need to do for this lab:

* Use this project as a starting point, or build on *Exceptions*
* Add the tests below to EmployeeTest
* Add a delegate definition to Employee.cs that returns a double and accepts a double as an argument
* Add a property to Employee.cs named LocalTaxMethod that is of the delegate type
* In Pay(amt), modify so that if there is a LocalTaxMethod, it is called and its result is added to the tax

[Fact]  
public void NullLocalTaxTest()  
{  
 var e = new Employee("Hank", "Hill", 100, DateTime.Now.AddYears(-5).AddDays(-10));  
 e.LocalTaxMethod = null;  
 e.Pay();  
}  
[Fact]  
public void LocalTaxCalledTest()  
{  
 var e = new Employee("Hank", "Hill", 100, DateTime.Now.AddYears(-5).AddDays(-10));  
 bool called = false;  
 e.LocalTaxMethod = (amt) => { called = true; return 0; };  
 e.Pay();  
 Assert.True(called);  
}  
[Fact]  
public void LocalTaxAddedTest()  
{  
 var e = new Employee("Hank", "Hill", 100, DateTime.Now.AddYears(-5).AddDays(-10));  
 var nt = e.Pay();  
 e.LocalTaxMethod = (amt) => 10;  
 Assert.Equal(nt - 10, e.Pay(), 2);  
}

# Events

## Overview

In this lab, we will add an event to the Employee class. This lab builds on the *Delegate* lab.

|  |  |
| --- | --- |
| Exercise Folder | Events |
| Builds On | Delegates |
| Time to complete | 30 minutes |

## Steps

* Add a delegate to *Employee* void NotifyPay(Employee emp, double net)
* Define an event to notify clients when the employee is paid
* Design unit tests
* Implement the feature in *Pay()*

# Lambda

## Overview

Convert private delegate methods into lambdas.

|  |  |
| --- | --- |
| Exercise Folder | Lambda |
| Builds On | Events |
| Time to complete | 10 minutes |

## Steps

* Replace the private methods/variables in the *EmployeeTest* with lambda expressions.# Moq

## Overview

We will create a class to represent *Company* and use mocks to decouple employees from the company for testing.

|  |  |
| --- | --- |
| Exercise Folder | MoqLab |
| Builds On | Lambda |
| Time to complete | 30 minutes |

## Steps

1. Continue with your last project
   * copy source files from the generic project if needed
2. Using Nuget, search and install Moq
3. Create a new class named *CompanyTest*
4. Create a new class named *Company*
5. Using the *Quick Refactorings Tool (screwdriver)* extract the interface
6. *IEmployee* from the class **Employee**
   * Note - exclude the deconstructors
7. Design tests to specify the behavior of the new *Company* class
   * *CompanyTest Constructor*:
     + Create a new company with a name and id; stash in an instance variable.
     + Create 2 or 3 mock IEmployee objects - stash in instance variables
   * *TestCompanyHire* create a test to hire and verify that the employees where hired
   * *TestCompanyPay* create a test to hire and pay employees - verify that they are paid
   * Verify that the tests fail
8. Implement *Company* so that tests pass

The initial *Company* class is shown below:

public class Company  
{  
 public Company(string name, string taxId)  
 {  
 throw new NotImplementedException();  
 }  
  
 public string Name { get; set; }  
 public string TaxId { get; set; }  
 public GenericArrayList<IEmployee> Employees { get; set; }  
 public void Hire(IEmployee emp)  
 {  
 throw new NotImplementedException();  
 }  
 public double Pay()  
 {  
 throw new NotImplementedException();  
 }  
}

# Inheritance

In this lab, we will expand the types of people that can be hired, including contractors and interns.

## Builds On - Moqlab

## Overview

* Define new classes
  + *Contractor* - contractors make a fixed rate X 50 hours
  + *Intern* - interns are always paid a stipend of $50
  + *HumanResource* - the base class for contractors, interns, and employees
* Define tests for the new classes
* Update Company to contain *Human Resource* objects

## Steps

1. Create new classes, *HumanResource, Company, Intern*
2. Change *Employee* to inherit from *HumanResource*
3. Pull up from *Employee* to *HumanResource* the common fields and properties
   * first and last name
   * Hiredate
   * YtdGrossPay
4. Define *Pay* as a abstract method in *HumanResource*
5. Define a constructor in *HumanResource* taking first and last name as parameters
6. Fix up Employee class
7. Change *Contractor* and *Intern* to inherit from *HumanResource*
8. Re-run unit tests - company and employee tests should still succeed
9. Create new classes *ContractorTest & InternTest*
   * implement tests to specify *Pay* results
   * verify new tests fail
   * implement *Intern & Contractor* so that tests pass
10. Update *CompanyTest* to mock *HumanResource* rather than *IEmployee*
11. Update *Company* to contain *HumanResource* objects

# Interfaces

In this lab, we eliminate the fat-interface smell with *Company*'s dependency on *HumanResource*. We will also implement the  [*Composite Pattern*](https://en.wikipedia.org/wiki/Composite_pattern)

## Builds On - Inheritance

## Overview

Determine *Company*'s dependency and design an interface that contains the exact methods/properties required by company. Replace *Company*'s dependency on *HumanResource* with the new interface type.

Also, we will allow companies to be nested within companies.

## Steps

1. Create a new interface named *IPayable*
   * single method: *void Pay()*
2. Add *IPayable* to the *HumanResource* base list
   * is there any purpose in *IEmployee* any more?
3. Modify *Company* replacing all occurances of *HumanResource* with *IPayable*
4. Update *CompanyTest* - change mocks

The composite pattern is an elegant pattern that will greatly increase the functionality of our application. How can we adjust the design so that companies can contain companies?# Extension Methods In this lab, we create an extension to the string class.

## Builds On - Interfaces

## Overview

The **Java** string class has a cool method named **Matches** This method takes a regular expression as a parameter and returns a true if the string matches the expression and false if it doesn't.

Let's take that a step further and add a method to the string class to *Validate* a string with a regular expression.

Add a class to the *Interfaces* project named *Extensions*. Change the class to public static Define a static bool extension method named *Validate*. The first **this** argument is a string (extending string class) and the second is a regular expression. If the string doesn't start with a ^ and/or end with a $, add them. Return true if this string matches the regex or false if it doesn't

Update the validation methods in *HumanResource* to use the extension method.

# LInQ API

In this lab, we will experiment with Linq queries on a database of movies.

## Overview

The project contains a file *MoviesJson.txt* that contains JSON encoded movie data.  
Each line in the file contains a movie title, genre, year, rating, and cast. Review the file *MovieDb.cs*. This class deserializes the data into a list of *Movie* objects.

Note - the properties of *MoviesJson.txt* are set to copy the file to the output folder so that it is in the same directory as the application .exe.

## Lab

In the top-level statements (*Program.cs*) experiment with queries

* How many movies did your favorite movie star appear in?
* What is the oldest movie with your star?
* What is the most recent?
* Did your favorite star appear in any movies with another favorite?
* Create a new list of just G rated movies with just the title, year, and first cast member
* etc...

# Linq Query

Modify your code from the previous lab to use Linq Query

## Builds on Linq API

# Attributes & Reflection

In this lab, we will define an attribute and create a tool that extracts information from assemblies

## Creates a new Console Application

### Documenter

* Create a new class named AuthorAttribute - inherits from Attribute
  + Define a constructor parameter accepting a string argument
* Create a console application named *Documentor*
* Create a class named *AuthorDocumentor* that loads the an assembly and searches for classes attributed with the [Author] attribute.
* Display the name of the class and the author

Here is a hint of the documentor class:

public class AuthorDocumenter  
{  
 public AuthorDocumenter(string assemblyPath)  
 {  
 AssemblyPath = assemblyPath;  
 }  
  
 public string AssemblyPath { get; set; }  
  
 public void Scan()  
 {  
 var assembly = System.Runtime  
 .Loader  
 .AssemblyLoadContext.Default  
 .LoadFromAssemblyPath(AssemblyPath);  
 var types = assembly.GetTypes()  
 .Where(t => t.IsDefined(typeof(AuthorAttribute)));  
 foreach (var t in types)  
 {  
 ...

## Previous lab

* In a previous lab, add the *Documenter* project as a dependency
* Mark some classes with the [Author] attribute

Run the documenter, passing the assembly name of the project containing attributes.

# Collections

In this lab, we will replace our *GenericArrayList* with a standard library class.

## Part 1

* Delete the source file *GenericArrayList*. Good riddance!
* Replace the class in *Company* with *List*
* Change the *Pay* method to utilize **Linq**

## Part 2

New requirement - we want to prevent duplicate hires

* Create a new test to specify that if the same employee is hired more than once, only one will be hired
  + do this for the same object (reference) and for different, equal employees
  + note - we don't need to test .Net collections, but we do want to specify that the payables are compatible with hash sets and sorted sets