Accelebrate®

ACCELERATED LEARNING, CELEBRATED RESULTS®

C# Language

https://www.accelebrate.com
(877) 849-1850 ❖ info@accelebrate.com

Customized Technical Training

Accelebrate
ACCELERATED LEARNING, CELEBRATED RESULTS®

On-Site, Customized Private Training

Don't settle for a one-size-fits-all class! Let Accelebrate tailor a private class to your group's goals and experience. Classes can be delivered at your site or online (or a combination of both) worldwide. Visit our web site at **https://www.accelebrate.com** and contact us at **sales@accelebrate.com** for details.

Public Online Training

Need to train just 1-3 people? Attend one of our regularly scheduled, live, instructor-led public online classes. Class sizes are small (typically 3-6 attendees) and you receive just as much hands-on time and individual attention from your instructor as our private classes. For course dates, times, outlines, pricing, and registration, visit **https://www.accelebrate.com/public-training-schedule.**

Newsletter

Want to find out about our latest class offerings? Subscribe to our newsletter **https://www.accelebrate.com/newsletter**.

Blog

Get insights and tutorials from our instructors and staff! Visit our blog, **https://www.accelebrate.com/blog** and join the discussion threads and get feedback from our instructors!

Learning Resources

Get access to learning guides, tutorials, and past issues of our newsletter at the Accelebrate library, **https://www.accelebrate.com/library**.

Call us for a training quote!

877 849 1850

Accelebrate, Inc. was founded in 2002 with the goal of delivering private training that rapidly achieves participants' goals. Each year, our experienced instructors deliver hundreds of classes online and at client sites all over the US, Canada, and abroad. We pride ourselves on our instructors' real-world experience and ability to adapt the training to your team and their objectives. We offer a wide range of topics, including:

- Angular, React, and Vue
- JavaScript
- Data Science using R, Python, & Julia
- Tableau & Power BI
- .NET & VBA programming
- SharePoint & Office 365
- DevOps
- iOS & Android Development
- PostgreSQL, Oracle, and SQL Server
- Java, Spring, and Groovy
- Agile
- Web/Application Server Admin
- HTML5 & Mobile Web Development
- Ansible & Chef
- AWS & Azure
- Adobe & Articulate software
- Docker, Kubernetes, Ansible, & Git
- IT leadership & Project Management
- AND MORE *(see back)*

> *"I have participated in several Accelebrate training courses and I can say that the instructors have always proven very knowledgeable and the materials are always very thorough and usable well after the class is over."*
>
> — Rick, AT&T

Like us on Facebook + Follow us on Twitter + Watch us on YouTube

Visit our website for a complete list of courses!

Adobe & Articulate

- Adobe Captivate
- Adobe Creative Cloud
- Adobe Presenter
- Articulate Storyline / Studio
- Camtasia
- RoboHelp

AWS, Azure, & Cloud

- AWS
- Azure
- Cloud Computing
- Google Cloud
- OpenStack

Big Data

- Actian Matrix Architecture
- Alteryx
- Apache Spark
- Greenplum Architecture
- Kognitio Architecture
- Teradata
- Snowflake SQL

Data Science and RPA

- Blue Prisim
- Django
- Julia
- Machine Learning
- Python
- R Programming
- Tableau
- UiPath

Database & Reporting

- BusinessObjects
- Crystal Reports
- MongoDB
- MySQL
- Oracle
- Oracle APEX
- Power BI
- PivotTable and PowerPivot
- PostgreSQL
- SQL Server
- Vertica Architecture & SQL

DevOps, CI/CD & Agile

- Agile
- Ansible
- Chef
- Docker
- Git
- Gradle Build System
- Jenkins
- Jira & Confluence
- Kubernetes
- Linux
- Microservices
- Red Hat
- Software Design

Java

- Apache Maven
- Apache Tomcat
- Groovy and Grails
- Hibernate
- Java & Web App Security
- JBoss
- Oracle WebLogic
- Scala
- Selenium & Cucumber
- Spring Boot
- Spring Framework

JS, HTML5, & Mobile

- Angular
- Apache Cordova
- CSS
- D3.js
- HTML5
- iOS/Swift Development
- JavaScript
- MEAN Stack
- Mobile Web Development
- Node.js & Express
- React & Redux
- Vue

Microsoft & .NET

- .NET Core
- ASP.NET
- Azure DevOps
- C#
- Design Patterns
- Entity Framework Core
- F#
- IIS
- Microsoft Dynamics CRM
- Microsoft Exchange Server
- Microsoft Office 365
- Microsoft Project
- Microsoft SQL Server
- Microsoft System Center
- Microsoft Windows Server
- PowerPivot
- PowerShell
- Team Foundation Server
- VBA
- Visual C++/CLI
- Web API

Other

- Blockchain
- C++
- Go Programming
- IT Leadership
- ITIL
- Project Management
- Regular Expressions
- Ruby on Rails
- Rust
- Salesforce
- XML

Security

- .NET Web App Security
- C and C++ Secure Coding
- C# & Web App Security
- Linux Security Admin
- Python Security
- Secure Coding for Web Dev
- Spring Security

SharePoint

- Power Automate & Flow
- SharePoint Administrator
- SharePoint Developer
- SharePoint End User
- SharePoint Online
- SharePoint Site Owner

SQL Server

- Azure SQL Data Warehouse
- Business Intelligence
- Performance Tuning
- SQL Server Administration
- SQL Server Development
- SSAS, SSIS, SSRS
- Transact-SQL

Teleconferencing Tools

- Adobe Connect
- GoToMeeting
- Microsoft Teams
- WebEx
- Zoom

Web/Application Server

- Apache Tomcat
- Apache httpd
- IIS
- JBoss
- Nginx
- Oracle WebLogic

Visit ***www.accelebrate.com/newsletter*** *to sign up and receive our* ***newsletters*** *with information about* ***new courses****,* ***free webinars****,* ***tutorials****, and* ***blog articles****.*

Call us for a training quote! 877 849 1850 (US/Canada) or +1 678 648 3133

.NET and C#

Agenda

- .NET Runtime and Class Library
- Visual Studio
- Data Types and Variables
- Enums, Structures and Classes
- Memory Management
- Branching and Flow Control
- Object-Oriented Techniques
- Interfaces
- Generics
- Delegates

.NET and C#

Agenda

- Attributes
- Arrays and Collections
- LINQ

.NET

Runtime and class library

- Common Language Runtime (CLR)
  - Managed execution environment
  - Compiles intermediate language (IL) code to native code
  - Provides for memory management, type safety, and security
- Base Class Library (BCL)
  - Collection of classes for common tasks

.NET

Unified type system

- .NET defines a Common Type System (CTS)
  - Same primitive types defined for all .NET languages
  - All types inherit from a single root `object` type
  - Support for user defined value and reference types

Visual Studio

IDE

- Visual Studio can be used to create many types of projects
  - Console applications
  - Windows Forms
  - Windows Presentation Foundation (WPF)
  - ASP.NET Web Forms
  - ASP.NET MVC
  - Windows Communications Foundation (WCF)
  - Class libraries
  - And more...

Visual Studio

IDE

- Provides Intellisense and a full-featured debugger
- Each project in Visual Studio produces a single .NET assembly
  - Executable assemblies are packaged as a .exe
  - Class library assemblies are packaged as a .dll
  - Contains metadata, IL code, and resources

C# Language

Hello, World

```
class Hello
{
  static void Main()
  {
    System.Console.WriteLine("Hello, World");
  }
}
```

Hello, World

Compilation

- C# source files typically have the extension .cs
- Code can be compiled with the Microsoft C# compiler using the command line

```
csc.exe hello.cs
```

- This would produce an executable assembly named hello.exe

Hello, World

Namespaces

- The “Hello, World” program can include a `using` directive that references the `System` namespace

```
using System;
```

- Namespaces provide a hierarchical means of organizing C# programs and libraries

Hello, World

Main method

- The Hello class declared by the “Hello, World” program has a method named `Main`
- By convention, a static method named Main serves as the entry point of a program and can have one of the following signatures:

```
static void Main() {...}

static void Main(string[] args) {...}

static int Main() {...}

static int Main(string[] args) {...}
```

The C# Language

Program structure

- The key organizational concepts in C# are programs, namespaces, types, members, and assemblies
  - C# programs consist of one or more source files
  - Programs declare types, which contain members and can be organized into namespaces
    - Classes and interfaces are examples of types
    - Fields, methods, properties, and events are examples of members
  - When C# programs are compiled, they are physically packaged into assemblies (typically .exe or .dll)

The C# Language

Program structure

- Assemblies contain executable code in the form of Intermediate Language (IL) instructions, and symbolic information in the form of metadata
  - Before it is executed, IL code is automatically converted to processor-specific code by the just-in-time (JIT) compiler of the .NET Common Language Runtime
- Because an assembly is a self-describing unit of functionality containing both code and metadata, there is no need for `#include` directives and header files in C#
  - If a program uses types in another assembly, the assembly can be referenced using the compiler's `/r` option

```
csc /r:acme.dll test.csc
```

Program Structure

Compilation

- C# permits the source text of a program to be stored in several source files
  - When a multi-file C# program is compiled, all of the source files are processed together as if they were in one large file
  - Forward declarations are never needed in C# because, with very few exceptions, declaration order is insignificant
  - C# does not limit a source file to declaring only one public type, nor does it require the name of the source file to match a type declared in the source file

Types and Variables

Values and references

- There are two kinds of types in C#: value types and reference types
  - Variables of value types directly contain their data
    - Each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other
  - Variables of reference types store references to their data
    - It is possible for two variables to reference the same object and, therefore, possible for operations on one variable to affect the object references by the other variable

Types and Variables

Value Types

- Simple types
  - Signed integral: `sbyte`, `short`, `int`, `long` (8, 16, 32, and 64 bits)
  - Unsigned integral: `byte`, `ushort`, `unit`, `ulong` (8, 16, 32, and 64 bits)
  - Unicode characters: `char` (UTF-16)
  - IEEE floating point: `float`, `double` (32 and 64 bits)
  - High-precision decimal: `decimal` (128 bits)
  - Boolean: `bool`

Types and Variables

Value Types

- Enum types

```
enum DayOfWeek { Monday, Tuesday, ... };
```

- Struct types

```
struct Point
{
  float X;
  float Y;
}
```

Types and Variables

Reference Types

- Class types

```
class Person { ... };
```

- Interface types

```
interface IGetsPaid { ... };
```

- Array types

```
int[] nums;
```

- Delegate types

```
delegate int Foo(int a);
```

Types and Variables

Nullable types

- For each non-nullable value type T, there is a corresponding nullable type T? which can hold an additional value of null
  - The syntax for a nullable type is short for Nullable<T>
  - An instance of a nullable type T? has two public properties
    - HasValue of type bool
    - Value of type T

Enums, Structures, and Classes

Enums

- An enum type is a distinct type with named constants
  - Every enum type has an underlying type which must be one of the eight integral types
  - Defaults to int with a starting value of zero

Enums, Structures, and Classes

Structs

- A struct type represents a structure with data members and function members
  - Structs are value types and do not require heap allocation
  - Do not support user-specified inheritance

Enums, Structures, and Classes

Classes

- A class defines a data structure that contains data members and function members
  - Classes are reference types
  - Support single inheritance and polymorphism

Memory Management

Garbage collection

- The .NET CLR provides garbage collection for all heap allocated objects
- Runs on a background thread
  - May be deferred to avoid affecting application performance
  - Will run sooner if available memory is low
- Destructor syntax can be used for code that should run when memory is collected

```
class Person
{
  ~Person() { ... }
}
```

Memory Management

Dispose pattern

- For objects that acquire external resources, the dispose pattern should be employed to provide more control

```
class Repository : IDisposable
{
  void Dispose()
  {
    // free resources
  }
}
```

Memory Management

Using statement

- The using statement can be used to call Dispose( ) in a guaranteed fashion for a specified object

```
using (Repository r = new Repository)
{
  r.DoStuff();
}
```

```
Repository r = new Repository();
try
{
  r.DoStuff();
}
finally
{
  r.Dispose();
}
```

Memory Management

Boxing and unboxing

- The .NET CTS allows any type to be treated as if of type object
  - Values of value types are treated as objects by performing boxing and unboxing operations
  - Boxing incurs the overhead of a heap allocation and produces an object that needs to be garbage collected

```
static void Main()
{
  int i = 123;
  object o = i;    // boxing
  int j = (int)o;  // unboxing
}
```

Memory Management

Method parameters

- A value parameter corresponds to a local variable that gets its value from the argument that was passed
  - Modifications do not affect the argument that was passed
- A reference parameter represents the same storage location as the argument variable
  - Declared with the ref modifier
- An output parameter is similiar to a reference parameter but the initial value of the caller-provided argument is unimportant
  - Declared with the out modifier
- A parameter array permits a variable number of arguments

```
public static void WriteLine(string fmt, params object[] args) { ... }
```

Branching and Flow Control

Statements

- Selection statements
    - if and switch
- Iteration statements
    - while, do, for, and foreach
- Jump statements
    - break, continue, throw, and return

Branching and Flow Control

Statements

- Selection statements
    - if and switch
- Iteration statements
    - while, do, for, and foreach
- Jump statements
    - break, continue, throw, and return

Object-Oriented Techniques

Classes and objects

- Instances of a class are created using the new operator
  - Allocates memory for a new instance
  - Invokes a constructor to initialize the instance
  - Returns a reference to the instance

```
Person p = new Person("Bill", "Gates");
```

Object-Oriented Techniques

Members

- Members of a class are either static members (belong to the class) or instance members (belong to the object)

Member	Description
Constants	Constant values associated with the class
Fields	Variables of the class
Methods	Actions that can be performed by the class
Properties	Actions for reading and writing field values
Indexers	Actions invoked via a index-style syntax
Events	Notifications that can be generated by the class
Operators	Conversions and expression operators supported by the class
Constructors	Actions to initialize instance of the class or the class itself
Destructors	Actions to perform before instances of the class are destoryed
Types	Nested types declared by the class

Object-Oriented Techniques

Accessibility

- Each member of a class has an associated accessibility which controls the the code able to access the member
  - public : Access is not limited
  - protected : Access limited to this class or classes derived from this class
  - internal : Access limited to this assembly
  - protected internal : Access limited to this assembly or class derived from this class
  - private : Access limited to this class

Object-Oriented Techniques

Class modifiers

- abstract
  - Can inherit from but can not instantiate
- sealed
  - Can instantiate but cannot inherit from
- partial
  - Allows for a single class to span multiple physical files

Object-Oriented Techniques

Virtual methods

- When an instance methos includes the virtual modifier, the method is said to be a virtual method
  - When a virtual method is invoked, the runtime type of the instance determines the implementation to invoke
  - When a non-virtual method is invoked, the compile-time type of the instance determines the implementation to invoke
- A virtual method can be overridden in the derived class
  - Requires use of the override modifier

Object-Oriented Techniques

Abstract methods

- An abstract method is a virtual method with no implementation
  - Declared with the abstract modifier
  - Requires class to be abstract
  - Must be overridden in every non-abstract derived class

Object-Oriented Techniques

Properties

- Properties are a natural extension of fields
- A property is declared like a field, except that the declaration ends with a get accessor and / or a set accessor
- A property that has both a get and set accessor is a read-write property
- A property that has only a get accessor is a read-only property
- A property that has only a set accessor is a write-only property

Object-Oriented Techniques

Interfaces

- An interface defines a contract that can be implemented by classes and structs
  - Can contain methods, properties, events, and indexers
  - An interface can not provide implementations
  - Interfaces may employ multiple inheritance

```
interface IControl { void Parent(); }
interface ITextBox : IControl { void SetText(string text); }
interface IListBox : IControl { void SetItems(string[] items); }
interface IComboBox : ITextBox, IListBox { }
```

Generics

Type parameters

- A method or class definition may specify a set of type parameters with angle brackets

```
public void Foo<T>(T obj1, T obj2);
```

```
public class Pair<TFirst, TSecond>
{
  public TFirst First;
  public TSecond Second;
}
```

```
Pair<int, string> pair = new Pair<int, string>();
```

Delegates

Type-safe function pointers

- A delegate type represents references to methods with a particular parameter list and return type
  - Make it possible to treat methods as entities that can be assigned to variables and passed as parameters

```
delegate void DoSomething(string msg);

public static void Main()
{
  DoSomthing ds = Foo;
  ds.Invoke("Hello");
  ds("Hello");
  ds.BeginInvoke("Hello", ...);
}

void Foo(string thing) { ... }
```

Delegates

Generics

- .NET includes some generic delegate types

```
delegate bool Predicate<T>(T obj);
```

```
delegate void Action<T>(T obj);
```

```
delegate int Comparison<T>(T x, T y);
```

```
delegate TResult Function<T, TResult>(T arg);
```

Delegates

Parameters

- Some framework methods take a parameter whose type is a generic delegate

```
void DoSomething()
{
  List<int> nums = new List<int> { 1, 2, 3 };
  List<int> results = nums.FindAll(IsEven);
  foreach (int n in results) Console.WriteLine(n);
}

bool IsEven(int i)
{
  return (i % 2) == 0;
}
```

Delegates

Anonymous delegates

- An anonymous delegate is a way specify the function for a delegate inline

```
void DoSomething()
{
  List<int> nums = new List<int> { 1, 2, 3 };
  List<int> results = nums.FindAll(
      delegate(int i) { return (i % 2) == 0; });

  foreach (int n in results) Console.WriteLine(n);
}
```

Delegates

Lambda expressions

- A lambda expression is a shorter way to define an anonymous delegate

```
void DoSomething()
{
  List<int> nums = new List<int> { 1, 2, 3 };
  List<int> results = nums.FindAll(d => (i % 2) == 0);

  foreach (int n in results) Console.WriteLine(n);
}
```

Delegates

Lambda expressions

- The left side of the lambda operator (=>) are the function's parameters
  - Types can be excluded as implicit typing can be used
- The right side of the lambda operator (=>) is the implementation of the function
  - The return keyword and brackets are options if only one statement

```
(int i) => { return (i % 2) == 0; }
```

```
i => { return (i % 2) == 0; }
```

```
i => (i % 2) == 0
```

Delegates

Lambda expressions

- Lambda expression can have any number of arguments (including none)

```
i, j => i + j
```

```
() => "Hello world!"
```

Delegates

Lambda expressions

- Lambda expression can also have any number of statements

```
i, j => {
  i = i * 2;
  j = j + 10;
  if (i > j) {
    return i;
  } else {
    throw new ApplicationException();
  }
}
```

Delegates

Lambda expressions

- A lambda expression can be used anywhere an instance of a delegate is required
- Using a lambda expression is always optional
  - A named or anonymous delegate can be used instead

Attributes

Declarative modifiers

- User-defined types of declarative information can be attached to program entities and retrieved at runtime
  - Specified using attributes
- All attribute classes derive from the System.Attribute base class
- When an attribute is requested via reflection, the constructor for the attribute is invoked and the resulting attribute instance is returned

New Language Features

Implicitly typed local variables

```
public void Foo()
{
  int i = 5;
  string str = "Hello";
  Person p = new Person("Bill", "Gates");
}
```

```
public void Foo()
{
  var i = 5;
  var str = "Hello";
  var p = new Person("Bill", "Gates");
}
```

New Language Features

Implicitly typed local variables

- Variables still strongly-typed
- Compiler determines type at compile-time (not runtime)
- Can only be used for local variables

⊘
```
public var Foo() { }
```

⊘
```
public void Foo(var x) { }
```

⊘
```
public class Person
{
  private var Name = "Joe";
}
```

New Language Features

Implicitly typed local variables

- Must be initialized at the time of declaration
- Can be set to null but cannot be initialized to null

```
var s = "Test";
s = null;
```

⊘
```
var s = null;
s = "Test";
```

New Language Features

Implicitly typed local variables

- Cannot change type after declaration

```
var i = 5;

i = 7;

⊘ i = "Ten";
```

New Language Features

Implicitly typed local variables

- Can be used with collections if all values are valid for one type

```
var a = new[] { 1, 2, 3 };  // int[]

var b = new[] { 1.5, 2.2, 3.75 };  // double[]

var c = new[] { "Hi", null, "Hello" };  // string[]

var d = new[] { new Person(), new Person() };  // Person[]

var d = new List<string> { "Bill", "Steve" };

⊘ var o = new[] { 1, "two" };
```

New Language Features

Implicitly typed local variables

- Can be used in foreach construct

```
var people = GetPeople();

foreach (var p in people) {
  Console.WriteLine(p.LastName);
}
```

New Language Features

Implicitly typed local variables

- Should not be used as a simple time saver
- There are times, when using LINQ, where implicit typing is required

New Language Features

Extension methods

- Extension methods allow for the appearance of extending an existing type without modifying the type itself
- Defined as static members of a static class
- Use the keyword this to indicate the type being extended

```
static class MyExtensions
{
  public static string GetAssembly(this object obj)
  {
    return Assembly.GetAssembly(obj.GetType()).FullName;
  }

  public static int Square(this int i)
  {
    return i * i;
  }
}
```

New Language Features

Extension methods

- LINQ uses extension methods to extend existing collection types
- Are only available when extension method's namespace is included with a using directive

New Language Features

Object initialization syntax

- Object initialization syntax allows for the setting of properties as part of the construction statement

```
Person p = new Person();
p.FirstName = "Bill";
p.LastName = "Gates";
```

```
Person p = new Person { FirstName = "Bill", LastName = "Gates" };
```

New Language Features

Object initialization syntax

- Can be used with a non-parameterless constructor

```
Person p = new Person("Bill") { LastName = "Gates" };
```

New Language Features

Object initialization syntax

- Setting of properties occurs after the constructor has completed

```
Person p = new Person("Bill") { FirstName = "Steve" };
```

```
Person p = new Person("Bill");
p.FirstName = "Steve";
```

New Language Features

Object initialization syntax

- Can be nested if desired

```
Rectangle r = new Rectangle();
Point p1 = new Point();
p1.X = 10;
p1.Y = 10;
r.TopLeft = p1;
Point p2 = new Point();
p2.X = 200;
p2.Y = 200;
r.BottomRight = p2;
```

```
Rectangle r = new Rectangle {
    TopLeft = new Point { X = 10, Y = 10 },
    BottomRight = new Point { X = 200, Y = 200 }};
```

New Language Features

Object initialization syntax

- Can be used with collections

```
List<Person> = new List<Person>
{
  new Person { FirstName = "Bill", LastName = "Gates" },
  new Person { FirstName = "Steve", LastName = "Jobs" }
}
```

New Language Features

Anonymous types

- Representing application data as instances of objects is a good idea
  - This makes many tasks such as data binding easier
- Query results would ideally also be returned as a collection of "ad hoc" objects based on what's being selected
  - Anonymous types provide for this

New Language Features

Anonymous types

- You define an anonymous type by combining implicit typing with object initialization syntax

```
var car = new { Make = "Ford", Model = "Focus", Speed = 55 };
```

LINQ

Introduction

- Language Integrated Query (LINQ) introduces queries as a first-class concept in the .NET languages
  - Compile-time support
  - Consistent query syntax across data sources

```
var query =
    from   c in Customers
    where  c.Country == "Italy"
    select c.CompanyName;

foreach (string name in query) {
  Console.WriteLine(name);
}
```

LINQ

How LINQ works

- LINQ defines a set of extension methods for collections which take delegates as parameters
- The compiler translates a LINQ expression into the appropriate method calls and delegates

```
var query = Customers
    .Where(c => c.Country == "Italy")
    .Select(c => c.CompanyName);

foreach (string name in query) {
  Console.WriteLine(name);
}
```

LINQ

Deferred execution

- By default, LINQ will not retrieve the results of a LINQ expression until you attempt to enumerate the results

```
var query =
    from   c in Customers
    where  c.Country == "Italy"
    select c.CompanyName;

foreach (string name in query) {
  Console.WriteLine(name);
}
```

LINQ

Deferred execution

- Enumerating over the result of a LINQ expression a second time will cause a reevaluation of the expression
- Deferred execution can be effectively disabled by forcing LINQ to populate a "disconnected" collection

```
var query =
   (from   c in Customers
    where  c.Country == "Italy"
    select c.CompanyName).ToList();

foreach (string name in query) {
  Console.WriteLine(name);
}
```

LINQ

Data projection

- The results of a LINQ expression can be projected into another type

```
var query =
    from   emp in Employees
    where  emp.Department == "Sales"
    select new Person { FirstName = emp.FirstName,
                        LastName = emp.LastName };

foreach (Person p in query) {
  Console.WriteLine(p.FirstName);
}
```

LINQ

Data projection

- Data projection can be combined with anonymous types
  - Can be especially useful to retrieve "sub results" for use in data binding

```
var query =
    from   emp in Employees
    where  emp.Department == "Sales"
    select new { FirstName = emp.FirstName,
                 LastName = emp.LastName };

foreach (var p in query) {
  Console.WriteLine(p.FirstName);
}
```