You could initialize the identity matrix with the following code:

C#

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Any accessible indexer that contains an accessible setter can be used as one of the expressions in an object initializer, regardless of the number or types of arguments. The index arguments form the left side of the assignment, and the value is the right side of the expression. For example, these are all valid if `IndexersExample` has the appropriate indexers:

C#

```
var thing = new IndexersExample {
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

For the preceding code to compile, the `IndexersExample` type must have the following members:

C#

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] {  set { ... }; }
```

## Object Initializers with anonymous types

Although object initializers can be used in any context, they are especially useful in LINQ query expressions. Query expressions make frequent use of anonymous types, which can only be initialized by using an object initializer, as shown in the following declaration.

C#

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Anonymous types enable the `select` clause in a LINQ query expression to transform objects of the original sequence into objects whose value and shape may differ from the original. This is useful if you want to store only a part of the information from each object in a sequence. In the following example, assume that a product object (`p`) contains many fields and methods, and that you are only interested in creating a sequence of objects that contain the product name and the unit price.

C#

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

When this query is executed, the `productInfos` variable will contain a sequence of objects that can be accessed in a `foreach` statement as shown in this example:

C#

```
foreach(var p in productInfos){...}
```

Each object in the new anonymous type has two public properties that receive the same names as the properties or fields in the original object. You can also rename a field when you are creating an anonymous type; the following example renames the `UnitPrice` field to `Price`.

C#

```csharp
select new {p.ProductName, Price = p.UnitPrice};
```

## Collection initializers

Collection initializers let you specify one or more element initializers when you initialize a collection type that implements IEnumerable and has `Add` with the appropriate signature as an instance method or an extension method. The element initializers can be a simple value, an expression, or an object initializer. By using a collection initializer, you do not have to specify multiple calls; the compiler adds the calls automatically.

The following example shows two simple collection initializers:

C#

```csharp
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

The following collection initializer uses object initializers to initialize objects of the `Cat` class defined in a previous example. Note that the individual object initializers are enclosed in braces and separated by commas.

C#

```csharp
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

You can specify null as an element in a collection initializer if the collection's `Add` method allows it.

C#

```csharp
List<Cat> moreCats = new List<Cat>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

You can specify indexed elements if the collection supports read / write indexing.

C#

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

The preceding sample generates code that calls the Item[TKey] to set the values. You could also initialize dictionaries and other associative containers using the following syntax. Notice that instead of indexer syntax, with parentheses and an assignment, it uses an object with multiple values:

C#

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },


    {42, "forty-two" }
};
```

This initializer example calls Add(TKey, TValue) to add the three items into the dictionary. These two different ways to initialize associative collections have slightly different behavior because of the method calls the compiler generates. Both variants work with the `Dictionary` class. Other types may only support one or the other based on their public API.

## Object Initializers with collection read-only property initialization

Some classes may have collection properties where the property is read-only, like the `Cats` property of `CatOwner` in the following case:

C#

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
}
```

You will not be able to use collection initializer syntax discussed so far since the property cannot be assigned a new list:

C#

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

However, new entries can be added to `Cats` nonetheless using the initialization syntax by omitting the list creation (`new List<Cat>`), as shown next:

C#

```
CatOwner owner = new CatOwner
{
    Cats =
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

The set of entries to be added simply appear surrounded by braces. The above is identical to writing:

C#

```
CatOwner owner = new CatOwner();
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

## Examples

The following example combines the concepts of object and collection initializers.

C#

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat> moreCats = new List<Cat>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);
        foreach (Cat c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null value.");
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

The following example shows an object that implements IEnumerable and contains an `Add` method with multiple parameters, It uses a collection initializer with multiple elements per item in the list that correspond to the signature of the `Add` method.

C#

```
    public class FullExample
    {
        class FormattedAddresses : IEnumerable<string>
        {
            private List<string> internalList = new List<string>();
            public IEnumerator<string> GetEnumerator() =>
internalList.GetEnumerator();

            System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

            public void Add(string firstname, string lastname,
                string street, string city,
                string state, string zipcode) => internalList.Add(
                $@"{firstname} {lastname}
{street}
{city}, {state} {zipcode}"
                );
        }

        public static void Main()
        {
            FormattedAddresses addresses = new FormattedAddresses()
            {
                {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
                {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
            };

            Console.WriteLine("Address Entries:");

            foreach (string addressEntry in addresses)
            {
                Console.WriteLine("\r\n" + addressEntry);
            }
        }

        /*
         * Prints:

            Address Entries:

            John Doe
            123 Street
            Topeka, KS 00000

            Jane Smith
            456 Street
            Topeka, KS 00000
         */
    }
```

`Add` methods can use the `params` keyword to take a variable number of arguments, as shown in the following example. This example also demonstrates the custom

implementation of an indexer to initialize a collection using indexes.

C#

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> :
IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new
Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator()
=> internalDictionary.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();

        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key,
(IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue> stored-
Values))
                internalDictionary.Add(key, storedValues = new List<TValue>
());

            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary2

            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie",
"Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" }
}
            };

        Console.WriteLine("Using first multi-valued dictionary created with a
collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");
            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created
with a collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created
with a collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)

            {
                Console.WriteLine(member);
            }
        }
    }

    /*
     * Prints:

        Using first multi-valued dictionary created with a collection initial-
izer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
        Using second multi-valued dictionary created with a collection ini-
tializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection initial-
izer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse


     */
}
```

## See also

- Use object initializers (style rule IDE0017)
- Use collection initializers (style rule IDE0028)
- C# Programming Guide
- LINQ in C#
- Anonymous Types