Deconstructing tuples and other types

Article • 09/29/2022 • 10 minutes to read

A tuple provides a lightweight way to retrieve multiple values from a method call. But once you retrieve the tuple, you have to handle its individual elements. Working on an element-by-element basis is cumbersome, as the following example shows. The `QueryCityData` method returns a three-tuple, and each of its elements is assigned to a variable in a separate operation.

```C#
public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

         // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

Retrieving multiple field and property values from an object can be equally cumbersome: you must assign a field or property value to a variable on a member-by-member basis.

You can retrieve multiple elements from a tuple or retrieve multiple field, property, and computed values from an object in a single *deconstruct* operation. To deconstruct a tuple, you assign its elements to individual variables. When you deconstruct an object, you assign selected values to individual variables.

Tuples

C# features built-in support for deconstructing tuples, which lets you unpackage all the items in a tuple in a single operation. The general syntax for deconstructing a tuple is similar to the syntax for defining one: you enclose the variables to which each element is to be assigned in parentheses in the left side of an assignment statement. For example, the following statement assigns the elements of a four-tuple to four separate variables:

```C#
var (name, address, city, zip) = contact.GetAddressInfo();
```

There are three ways to deconstruct a tuple:

- You can explicitly declare the type of each field inside parentheses. The following example uses this approach to deconstruct the three-tuple returned by the `QueryCityData` method.

  ```C#
  public static void Main()
  {
      (string city, int population, double area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

- You can use the `var` keyword so that C# infers the type of each variable. You place the `var` keyword outside of the parentheses. The following example uses type inference when deconstructing the three-tuple returned by the `QueryCityData` method.

  ```C#
  public static void Main()
  {
      var (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

  You can also use the `var` keyword individually with any or all of the variable declarations inside the parentheses.

C#

```csharp
public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York 
City");

    // Do something with the data.
}
```

This is cumbersome and isn't recommended.

- Lastly, you may deconstruct the tuple into variables that have already been declared.

C#

```csharp
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;
    double area = 144.8;

    (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- Beginning in C# 10, you can mix variable declaration and assignment in a deconstruction.

C#

```csharp
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

You can't specify a specific type outside the parentheses even if every field in the tuple has the same type. Doing so generates compiler error CS8136, "Deconstruction 'var (...)' form disallows a specific type for 'var'.".

You must assign each element of the tuple to a variable. If you omit any elements, the compiler generates error CS8132, "Can't deconstruct a tuple of 'x' elements into 'y' variables."

Tuple elements with discards

Often when deconstructing a tuple, you're interested in the values of only some elements. You can take advantage of C#'s support for *discards*, which are write-only variables whose values you've chosen to ignore. A discard is chosen by an underscore character ("_") in an assignment. You can discard as many values as you like; all are represented by the single discard, `_`.

The following example illustrates the use of tuples with discards. The `QueryCityDataForYears` method returns a six-tuple with the name of a city, its area, a year, the city's population for that year, a second year, and the city's population for that second year. The example shows the change in population between those two years. Of the data available from the tuple, we're unconcerned with the city area, and we know the city name and the two dates at design-time. As a result, we're only interested in the two population values stored in the tuple, and can handle its remaining values as discards.

C#

```csharp
using System;

public class ExampleDiscard
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City",
1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 -
pop1:N0}");
    }

    private static (string, double, int, int, int, int)
QueryCityDataForYears(string name, int year1, int year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

User-defined types

C# doesn't offer built-in support for deconstructing non-tuple types other than the record and DictionaryEntry types. However, as the author of a class, a struct, or an interface, you can allow instances of the type to be deconstructed by implementing one or more `Deconstruct` methods. The method returns void, and each value to be deconstructed is indicated by an out parameter in the method signature. For example, the following `Deconstruct` method of a `Person` class returns the first, middle, and last name:

```C#
public void Deconstruct(out string fname, out string mname, out string lname)
```

You can then deconstruct an instance of the `Person` class named `p` with an assignment like the following code:

```C#
var (fName, mName, lName) = p;
```

The following example overloads the `Deconstruct` method to return various combinations of properties of a `Person` object. Individual overloads return:

- A first and last name.
- A first, middle, and last name.

- A first name, a last name, a city name, and a state name.

C#

```csharp
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string
lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class ExampleClassDeconstruction
```

```
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

Multiple `Deconstruct` methods having the same number of parameters are ambiguous. You must be careful to define `Deconstruct` methods with different numbers of parameters, or "arity". `Deconstruct` methods with the same number of parameters cannot be distinguished during overload resolution.

User-defined type with discards

Just as you do with tuples, you can use discards to ignore selected items returned by a `Deconstruct` method. Each discard is defined by a variable named "_", and a single deconstruction operation can include multiple discards.

The following example deconstructs a `Person` object into four strings (the first and last names, the city, and the state) but discards the last name and the state.

C#

```csharp
// Deconstruct the person object.
var (fName, _, city, _) = p;
Console.WriteLine($"Hello {fName} of {city}!");
// The example displays the following output:
//      Hello John of Boston!
```

Extension methods for user-defined types

If you didn't author a class, struct, or interface, you can still deconstruct objects of that type by implementing one or more `Deconstruct` extension methods to return the values in which you're interested.

The following example defines two `Deconstruct` extension methods for the System.Reflection.PropertyInfo class. The first returns a set of values that indicate the characteristics of the property, including its type, whether it's static or instance, whether it's read-only, and whether it's indexed. The second indicates the property's accessibility. Because the accessibility of get and set accessors can differ, Boolean values indicate whether the property has separate get and set accessors and, if it does, whether they have the same accessibility. If there's only one accessor or both the get and the set accessor have the same accessibility, the `access` variable indicates the accessibility of the property as a whole. Otherwise, the accessibility of the get and set accessors are indicated by the `getAccess` and `setAccess` variables.

C#

```csharp
using System;
using System.Collections.Generic;
using System.Reflection;

public static class ReflectionExtensions
{
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                   out bool isReadOnly, out bool isIndexed,
                                   out Type propertyType)
    {
        var getter = p.GetMethod;

        // Is the property read-only?
        isReadOnly = ! p.CanWrite;

        // Is the property instance or static?
        isStatic = getter.IsStatic;

        // Is the property indexed?
        isIndexed = p.GetIndexParameters().Length > 0;

        // Get the property type.
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool hasGetAndSet,
                                   out bool sameAccess, out string access,
                                   out string getAccess, out string setAccess)
    {
        hasGetAndSet = sameAccess = false;
        string getAccessTemp = null;
        string setAccessTemp = null;

        MethodInfo getter = null;
```

```
if (p.CanRead)
    getter = p.GetMethod;

MethodInfo setter = null;
if (p.CanWrite)
    setter = p.SetMethod;

if (setter != null && getter != null)
    hasGetAndSet = true;

if (getter != null)
{
    if (getter.IsPublic)
        getAccessTemp = "public";
    else if (getter.IsPrivate)
        getAccessTemp = "private";
    else if (getter.IsAssembly)
        getAccessTemp = "internal";
    else if (getter.IsFamily)
        getAccessTemp = "protected";
    else if (getter.IsFamilyOrAssembly)
        getAccessTemp = "protected internal";
}

if (setter != null)
{
    if (setter.IsPublic)
        setAccessTemp = "public";
    else if (setter.IsPrivate)
        setAccessTemp = "private";
    else if (setter.IsAssembly)
        setAccessTemp = "internal";
    else if (setter.IsFamily)
        setAccessTemp = "protected";
    else if (setter.IsFamilyOrAssembly)
        setAccessTemp = "protected internal";
}

// Are the accessibility of the getter and setter the same?
if (setAccessTemp == getAccessTemp)
{
    sameAccess = true;
    access = getAccessTemp;
    getAccess = setAccess = String.Empty;
}
else
{
    access = null;
    getAccess = getAccessTemp;
    setAccess = setAccessTemp;
}
```

```
    }
}

public class ExampleExtension
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\nThe {dateType.FullName}.{prop.Name} property:");
        Console.WriteLine($"   PropertyType: {propType.Name}");
        Console.WriteLine($"   Static:       {isStatic}");
        Console.WriteLine($"   Read-only:    {isRO}");
        Console.WriteLine($"   Indexed:      {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility, setAc-
cessibility) = prop;
        Console.Write($"\nAccessibility of the {listType.FullName}.{prop.Name}
property: ");

        if (!hasGetAndSet | sameAccess)
        {
            Console.WriteLine(accessibility);
        }
        else
        {
            Console.WriteLine($"\n   The get accessor: {getAccessibility}");
            Console.WriteLine($"   The set accessor: {setAccessibility}");
        }
    }
}
// The example displays the following output:
//       The System.DateTime.Now property:
//          PropertyType: DateTime
//          Static:       True
//          Read-only:    True
//          Indexed:      False
//
//       Accessibility of the System.Collections.Generic.List`1.Item property:
public
```

Extension method for system types

Some system types provide the `Deconstruct` method as a convenience. For example, the System.Collections.Generic.KeyValuePair<TKey,TValue> type provides this functionality. When you're iterating over a System.Collections.Generic.Dictionary<TKey,TValue> each element is a `KeyValuePair<TKey, TValue>` and can be deconstructed. Consider the following example:

```C#
Dictionary<string, int> snapshotCommitMap = 
new(StringComparer.OrdinalIgnoreCase)
{
    ["https://github.com/dotnet/docs"] = 16_465,
    ["https://github.com/dotnet/runtime"] = 114_223,
    ["https://github.com/dotnet/installer"] = 22_436,
    ["https://github.com/dotnet/roslyn"] = 79_484,
    ["https://github.com/dotnet/aspnetcore"] = 48_386
};

foreach (var (repo, commitCount) in snapshotCommitMap)
{
    Console.WriteLine(
        $"The {repo} repository had {commitCount:N0} commits as of November 
10th, 2021.");
}
```

You can add a `Deconstruct` method to system types that don't have one. Consider the following extension method:

```C#
public static class NullableExtensions
{
    public static void Deconstruct<T>(
        this T? nullable,
        out bool hasValue,
        out T value) where T : struct
    {
        hasValue = nullable.HasValue;
        value = nullable.GetValueOrDefault();
    }
}
```

This extension method allows all Nullable<T> types to be deconstructed into a tuple of `(bool hasValue, T value)`. The following example shows code that uses this extension method:

```C#
DateTime? questionableDateTime = default;
var (hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

questionableDateTime = DateTime.Now;
(hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

// Example outputs:
// { HasValue = False, Value = 1/1/0001 12:00:00 AM }
// { HasValue = True, Value = 11/10/2021 6:11:45 PM }
```

`record` types

When you declare a record type by using two or more positional parameters, the compiler creates a `Deconstruct` method with an `out` parameter for each positional parameter in the `record` declaration. For more information, see Positional syntax for property definition and Deconstructor behavior in derived records.

See also

- Deconstruct variable declaration (style rule IDE0042)
- Discards
- Tuple types