Discards - C# Guide

docs.microsoft.com/en-us/dotnet/csharp/discards

- 07/21/2017
- 7 minutes to read

Starting with C# 7.0, C# supports discards, which are temporary, dummy variables that are intentionally unused in application code. Discards are equivalent to unassigned variables; they do not have a value. Because there is only a single discard variable, and that variable may not even be allocated storage, discards can reduce memory allocations. Because they make the intent of your code clear, they enhance its readability and maintainability.

You indicate that a variable is a discard by assigning it the underscore (`_`) as its name. For example, the following method call returns a 3-tuple in which the first and second values are discards and *area* is a previously declared variable to be set to the corresponding third component returned by *GetCityInformation*:

C#

```
(_, _, area) = city.GetCityInformation(cityName);
```

In C# 7.0, discards are supported in assignments in the following contexts:

- Tuple and object deconstruction.
- Pattern matching with is and switch.
- Calls to methods with `out` parameters.
- A standalone `_` when no `_` is in scope.

When `_` is a valid discard, attempting to retrieve its value or use it in an assignment operation generates compiler error CS0301, "The name '_' does not exist in the current context". This is because `_` is not assigned a value, and may not even be assigned a storage location. If it were an actual variable, you could not discard more than one value, as the previous example did.

Tuple and object deconstruction

Discards are particularly useful in working with tuples when your application code uses some tuple elements but ignores others. For example, the following `QueryCityDataForYears` method returns a 6-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#

```
using System;
using System.Collections.Generic;

public class Example
{
   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);
   }
}
```

For more information on deconstructing tuples with discards, see Deconstructing tuples and other types.

The `Deconstruct` method of a class, structure, or interface also allows you to retrieve and deconstruct a specific set of data from an object. You can use discards when you are interested in working with only a subset of the deconstructed values. 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#

```
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;
   }


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



      var (fName, _, city, _) = p;
      Console.WriteLine($"Hello {fName} of {city}!");
   }
}
```

For more information on deconstructing user-defined types with discards, see Deconstructing tuples and other types.

Pattern matching with `switch` and `is`

The *discard pattern* can be used in pattern matching with the is and switch keywords. Every expression always matches the discard pattern.

The following example defines a `ProvidesFormatInfo` method that uses is statements to determine whether an object provides an IFormatProvider implementation and tests whether the object is `null`. It also uses the discard pattern to handle non-null objects of any other type.

C#

```
using System;
using System.Globalization;

public class Example
{
  public static void Main()
  {
    object[] objects = { CultureInfo.CurrentCulture,
                         CultureInfo.CurrentCulture.DateTimeFormat,
                         CultureInfo.CurrentCulture.NumberFormat,
                         new ArgumentException(), null };
    foreach (var obj in objects)
      ProvidesFormatInfo(obj);
  }

  private static void ProvidesFormatInfo(object obj)
  {
    switch (obj)
    {
      case IFormatProvider fmt:
        Console.WriteLine($"{fmt} object");
        break;
      case null:
        Console.Write("A null object reference: ");
        Console.WriteLine("Its use could result in a NullReferenceException");
        break;
      case object _:
        Console.WriteLine("Some object type without format information");
        break;
    }
  }
}
```

Calls to methods with out parameters

When calling the `Deconstruct` method to deconstruct a user-defined type (an instance of a class, structure, or interface), you can discard the values of individual `out` arguments. But you can also discard the value of `out` arguments when calling any method with an out parameter.

The following example calls the DateTime.TryParse(String, out DateTime) method to determine whether the string representation of a date is valid in the current culture. Because the example is concerned only with validating the date string and not with parsing it to extract the date, the `out` argument to the method is a discard.

C#

```
using System;

public class Example
{
  public static void Main()
  {
    string[] dateStrings = {"05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                            "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                            "5/01/2018 14:57:32.80 -07:00",
                            "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                            "Fri, 15 May 2018 20:10:57 GMT" };
    foreach (string dateString in dateStrings)
    {
      if (DateTime.TryParse(dateString, out _))
        Console.WriteLine($"'{dateString}': valid");
      else
        Console.WriteLine($"'{dateString}': invalid");
    }
  }
}
```

A standalone discard

You can use a standalone discard to indicate any variable that you choose to ignore. The following example uses a standalone discard to ignore the Task object returned by an asynchronous operation. This has the effect of suppressing the exception that the operation throws as it is about to complete.

C#

```
using System;
using System.Threading.Tasks;

public class Example
{
  public static void Main()
  {
    ExecuteAsyncMethods().Wait();
  }

  private static async Task ExecuteAsyncMethods()
  {
    Console.WriteLine("About to launch a task...");
    _ = Task.Run(() => { var iterations = 0;
                         for (int ctr = 0; ctr < int.MaxValue; ctr++)
                           iterations++;
                         Console.WriteLine("Completed looping operation...");
                         throw new InvalidOperationException();
                       });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
  }
}
```

Note that `_` is also a valid identifier. When used outside of a supported context, `_` is treated not as a discard but as a valid variable. If an identifier named `_` is already in scope, the use of `_` as a standalone discard can result in:

- Accidental modification of the value of the in-scope `_` variable by assigning it the value of the intended discard. For example:

  ```
  private static void ShowValue(int _)
  {
    byte[] arr = { 0, 0, 1, 2 };
    _ = BitConverter.ToInt32(arr, 0);
    Console.WriteLine(_);
  }
  ```

- A compiler error for violating type safety. For example:

  ```
  private static bool RoundTrips(int _)
  {
    string value = _.ToString();
    int newValue = 0;
    _ = Int32.TryParse(value, out newValue);
    return _ == newValue;
  }
  ```

- Compiler error CS0136, "A local or parameter named '_' cannot be declared in this scope because that name is used in an enclosing local scope to define a local or parameter." For example:

  ```
  public void DoSomething(int _)
  {
   var _ = GetValue();
  }
  ```

See also