Tutorial: Explore primary constructors

Article • 06/23/2023

C# 12 introduces *primary constructors*, a concise syntax to declare constructors whose parameters are available anywhere in the body of the type.

In this tutorial, you will learn:

- ✓ When to declare a primary constructor on your type
- ✓ How to call primary constructors from other constructors
- ✓ How to use primary constructor parameters in members of the type
- ✓ Where primary constructor parameters are stored

Prerequisites

You need to set up your machine to run .NET 8 or later, including the C# 12 or later compiler. The C# 12 compiler is available starting with Visual Studio 2022 version 17.7 or the .NET 8 SDK.

Primary constructors

You can add parameters to a `struct` or `class` declaration to create a *primary constructor*. Primary constructor parameters are in scope throughout the class definition. It's important to view primary constructor parameters as *parameters* even though they are in scope throughout the class definition. Several rules clarify that they're parameters:

1. Primary constructor parameters may not be stored if they aren't needed.
2. Primary constructor parameters aren't members of the class. For example, a primary constructor parameter named `param` can't be accessed as `this.param`.
3. Primary constructor parameters can be assigned to.
4. Primary constructor parameters don't become properties, except in record types.

These rules are the same as parameters to any method, including other constructor declarations.

The most common uses for a primary constructor parameter are:

1. As an argument to a `base()` constructor invocation.
2. To initialize a member field or property.
3. Referencing the constructor parameter in an instance member.

Every other constructor for a class **must** call the primary constructor, directly or indirectly, through a `this()` constructor invocation. That rule ensures that primary constructor parameters are assigned anywhere in the body of the type.

Initialize property

The following code initializes two readonly properties that are computed from primary constructor parameters:

C#

```
public readonly struct Distance(double dx, double dy)
{
    public readonly double Magnitude { get; } = Math.Sqrt(dx * dx +
dy * dy);
    public readonly double Direction { get; } = Math.Atan2(dy, dx);
}
```

The preceding code demonstrates a primary constructor used to initialize calculated readonly properties. The field initializers for `Magnitude` and `Direction` use the primary constructor parameters. The primary constructor parameters aren't used anywhere else in the struct. The preceding struct is as though you'd written the following code:

C#

```
public readonly struct Distance
{
    public readonly double Magnitude { get; }

    public readonly double Direction { get; }

    public Distance(double dx, double dy)
    {
        Magnitude = Math.Sqrt(dx * dx + dy * dy);
        Direction = Math.Atan2(dy, dx);
    }
}
```

The new feature makes it easier to use field initializers when you need arguments to initialize a field or property.

Create mutable state

The preceding examples use primary constructor parameters to initialize readonly properties. You can also use primary constructors when the properties aren't readonly.

Consider the following code:

```C#
public struct Distance(double dx, double dy)
{
    public readonly double Magnitude => Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction => Math.Atan2(dy, dx);

    public void Translate(double deltaX, double deltaY)
    {
        dx += deltaX;
        dy += deltaY;
    }

    public Distance() : this(0,0) { }
}
```

In the preceding example, the `Translate` method changes the `dx` and `dy` components. That requires the `Magnitude` and `Direction` properties be computed when accessed. The `=>` operator designates an expression-bodied `get` accessor, whereas the `=` operator designates an initializer. This version adds a parameterless constructor to the struct. The parameterless constructor must invoke the primary constructor, so that all the primary constructor parameters are initialized.

In the previous example, the primary constructor properties are accessed in a method. Therefore the compiler creates hidden fields to represent each parameter. The following code shows approximately what the compiler generates. The actual field names are valid CIL identifiers, but not valid C# identifiers.

```C#
public struct Distance
{
    private double __unspeakable_dx;
    private double __unspeakable_dy;

    public readonly double Magnitude => Math.Sqrt(__unspeakable_dx *
__unspeakable_dx + __unspeakable_dy * __unspeakable_dy);
    public readonly double Direction => Math.Atan2(__unspeakable_dy,
__unspeakable_dx);

    public void Translate(double deltaX, double deltaY)
    {
        __unspeakable_dx += deltaX;
        __unspeakable_dy += deltaY;
    }

    public Distance(double dx, double dy)
```

```
    {
        __unspeakable_dx = dx;
        __unspeakable_dy = dy;
    }
    public Distance() : this(0, 0) { }
}
```

It's important to understand that the first example didn't require the compiler to create a field to store the value of the primary constructor parameters. The second example used the primary constructor parameter inside a method, and therefore required the compiler to create storage for them. The compiler creates storage for any primary constructors only when that parameter is accessed in the body of a member of your type. Otherwise, the primary constructor parameters aren't stored in the object.

Dependency injection

Another common use for primary constructors is to specify parameters for dependency injection. The following code creates a simple controller that requires a service interface for its use:

C#

```
public interface IService
{
    Distance GetDistance();
}

public class ExampleController(IService service) : ControllerBase
{
    [HttpGet]
    public ActionResult<Distance> Get()
    {
        return service.GetDistance();
    }
}
```

The primary constructor clearly indicates the parameters needed in the class. You use the primary constructor parameters as you would any other variable in the class.

Initialize base class

You can invoke a base class' primary constructor from the derived class' primary constructor. It's the easiest way for you to write a derived class that must invoke a primary constructor in the base class. For example, consider a hierarchy of classes that

represent different account types as a bank. The base class would look something like the following code:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = accountID;
    public string Owner { get; } = owner;

    public override string ToString() => $"Account ID: {AccountID},
Owner: {Owner}";
}
```

All bank accounts, regardless of the type, have properties for the account number and an owner. In the completed application, other common functionality would be added to the base class.

Many types require more specific validation on constructor parameters. For example, the `BankAccount` has specific requirements for the `owner` and `accountID` parameters: The `owner` must not be `null` or whitespace, and the `accountID` must be a string containing 10 digits. You can add this validation when you assign the corresponding properties:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = ValidAccountNumber(accountID)
        ? accountID
        : throw new ArgumentException("Invalid account number", name-
of(accountID));

    public string Owner { get; } = string.IsNullOrWhiteSpace(owner)
        ? throw new ArgumentException("Owner name cannot be empty",
nameof(owner))
        : owner;

    public override string ToString() => $"Account ID: {AccountID},
Owner: {Owner}";

    public static bool ValidAccountNumber(string accountID) =>
    accountID?.Length == 10 && accountID.All(c => char.IsDigit(c));
}
```

The previous example shows how you can validate the constructor parameters before assigning them to the properties. You can use builtin methods, like String.IsNullOrWhiteSpace(String), or your own validation method, like

`ValidAccountNumber`. In the previous example, any exceptions are thrown from the constructor, when it invokes the initializers. If a constructor parameter isn't used to assign a field, any exceptions are thrown when the constructor parameter is first accessed.

One derived class would present a checking account:

C#

```
public class CheckingAccount(string accountID, string owner, decimal
overdraftLimit = 0) : BankAccount(accountID, owner)
{
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Deposit amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -overdraftLimit)
        {
            throw new InvalidOperationException("Insufficient funds
for withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"Account ID: {AccountID},
Owner: {Owner}, Balance: {CurrentBalance}";
}
```

The derived `CheckingAccount` class has a primary constructor that takes all the parameters needed in the base class, and another parameter with a default value. The primary constructor calls the base constructor using the `: BankAccount(accountID, owner)` syntax. This expression specifies both the type for the base class, and the arguments for the primary constructor.

Your derived class isn't required to use a primary constructor. You can create a constructor in the derived class that invokes the base class' primary constructor, as shown in the following example:

C#

```
public class LineOfCreditAccount : BankAccount
{
    private readonly decimal _creditLimit;
    public LineOfCreditAccount(string accountID, string owner, deci-
mal creditLimit) : base(accountID, owner)
    {
        _creditLimit = creditLimit;
    }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Deposit amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -_creditLimit)
        {
            throw new InvalidOperationException("Insufficient funds
for withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"{base.ToString()},
Balance: {CurrentBalance}";
}
```

There's one potential concern with class hierarchies and primary constructors: it's possible to create multiple copies of a primary constructor parameter as it's used in both derived and base classes. The following code example creates two copies each of the `owner` and `accountID` field:

C#

```
public class SavingsAccount(string accountID, string owner, decimal
interestRate) : BankAccount(accountID, owner)
{
    public SavingsAccount() : this("default", "default", 0.01m) { }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Deposit amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < 0)
        {
            throw new InvalidOperationException("Insufficient funds
for withdrawal");
        }
        CurrentBalance -= amount;
    }

    public void ApplyInterest()
    {
        CurrentBalance *= 1 + interestRate;
    }

    public override string ToString() => $"Account ID: {accountID},
Owner: {owner}, Balance: {CurrentBalance}";
}
```

The highlighted line shows that the `ToString` method uses the *primary constructor parameters* (`owner` and `accountID`) rather than the *base class properties* (`Owner` and `AccountID`). The result is that the derived class, `SavingsAccount` creates storage for those copies. The copy in the derived class is different than the property in the base class. If the base class property could be modified, the instance of the derived class won't see that modification. The compiler issues a warning for primary constructor parameters that are used in a derived class and passed to a base class constructor. In this instance, the fix is to use the properties of the base class.

Summary

You can use the primary constructors as best suits your design. For classes and structs, primary constructor parameters are parameters to a constructor that must be invoked. You can use them to initialize properties. You can initialize fields. Those properties or fields can be immutable, or mutable. You can use them in methods. They're parameters, and you use them in what manner suits your design best. You can learn more about primary constructors in the C# programming guide article on instance constructors and the proposed primary constructor specification.