Chapter 9: nameof Operator

The `nameof` operator allows you to get the name of a **variable**, **type** or **member** in string form without hard-coding it as a literal.

The operation is evaluated at compile-time, which means that you can rename a referenced identifier, using an IDE's rename feature, and the name string will update with it.

Section 9.1: Basic usage: Printing a variable name

The `nameof` operator allows you to get the name of a variable, type or member in string form without hard-coding it as a literal. The operation is evaluated at compile-time, which means that you can rename, using an IDE's rename feature, a referenced identifier and the name string will update with it.

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

Would output

> myString

because the name of the variable is "myString". Refactoring the variable name would change the string.

If called on a reference type, the `nameof` operator returns the name of the current reference, *not* the name or type name of the underlying object. For example:

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // Returns "greeting"
Console.WriteLine(nameof(mailMessageBody)); // Returns "mailMessageBody", NOT "greeting"!
```

Section 9.2: Raising PropertyChanged event

Snippet

```
public class Person : INotifyPropertyChanged
{
    private string _address;

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public string Address
    {
        get { return _address; }
        set
        {
            if (_address == value)
            {
```

```
                return;
            }

            _address = value;
            OnPropertyChanged(nameof(Address));
        }
    }
}

...

var person = new Person();
person.PropertyChanged += (s,e) => Console.WriteLine(e.PropertyName);

person.Address = "123 Fake Street";
```

Console Output

> Address

Section 9.3: Argument Checking and Guard Clauses

Prefer

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));
        ...
    }
}
```

Over

```
public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException("orderLine");
        ...
    }
}
```

Using the `nameof` feature makes it easier to refactor method parameters.

Section 9.4: Strongly typed MVC action links

Instead of the usual loosely typed:

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

You can now make action links strongly typed:

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

Now if you want to refactor your code and rename the `UserController.LogIn` method to `UserController.SignIn`, you don't need to worry about searching for all string occurrences. The compiler will do the job.

Section 9.5: Handling PropertyChanged events

Snippet

```
public class BugReport : INotifyPropertyChanged
{
    public string Title { ... }
    public BugStatus Status { ... }
}

...

private void BugReport_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var bugReport = (BugReport)sender;

    switch (e.PropertyName)
    {
        case nameof(bugReport.Title):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Title);
            break;

        case nameof(bugReport.Status):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Status);
            break;
    }
}

...

var report = new BugReport();
report.PropertyChanged += BugReport_PropertyChanged;

report.Title = "Everything is on fire and broken";
report.Status = BugStatus.ShowStopper;
```

Console Output

> Title changed to Everything is on fire and broken
>
> Status changed to ShowStopper

Section 9.6: Applied to a generic type parameter

Snippet

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}
```

```
...

var myClass = new SomeClass<int>();
myClass.PrintTypeName();

Console.WriteLine(nameof(SomeClass<int>));
```

Console Output

> TItem
>
> SomeClass

Section 9.7: Printing a parameter name

Snippet

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

...

int myValue = 10;
DoSomething(myValue);
```

Console Output

> paramValue

Section 9.8: Applied to qualified identifiers

Snippet

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));
Console.WriteLine(nameof(MyClass));
Console.WriteLine(nameof(MyClass.MyNestedClass));
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

Console Output

> MyNamespace
>
> MyClass
>
> MyNestedClass
>
> MyStaticProperty