

O'REILLY®

Fourth  
Edition

# Head First

# C#

A Learner's Guide to  
Real-World Programming  
with C# and .NET Core

---

Andrew Stellman  
& Jennifer Greene



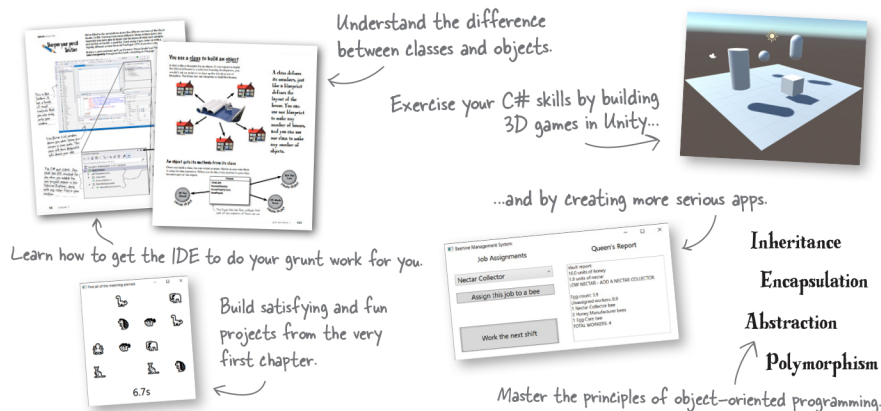
A Brain-Friendly Guide

# Head First

# C#

## What will you learn from this book?

Dive into C# and create apps, user interfaces, games, and more using this fun and highly visual introduction to C#, .NET Core, and Visual Studio. With this completely updated guide, which covers C# 8.0 and Visual Studio 2019, beginning programmers like you will build a fully functional game in the opening chapter. Then you'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, doing hands-on exercises, and building real-world applications. By the time you're done, you'll be a solid C# programmer—and you'll have a great time along the way!



## What's so special about this book?

Based on the latest research in cognitive science and learning theory, *Head First C#* uses a visually rich format to engage your mind rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multisensory learning experience is designed for the way your brain really works.

"Thank you so much!  
Your books have  
helped me to launch  
my career."

—Ryan White  
Game Developer

"Andrew and Jennifer  
have written a  
concise, authoritative,  
and most of all, fun  
introduction to C#  
development."

—Jon Galloway  
Senior Program Manager on the  
.NET Community Team  
at Microsoft

"If you want to learn  
C# in depth and have  
fun doing it, this is THE  
book for you."

—Andy Parker  
Fledgling C# programmer

.NET

US \$64.99

CAN \$85.99

ISBN: 978-1-491-97670-8



O'REILLY®

# Head First C#

Fourth Edition

WOULDN'T IT BE DREAMY IF  
THERE WAS A C# BOOK THAT WAS  
MORE FUN THAN MEMORIZING  
A DICTIONARY? IT'S PROBABLY  
NOTHING BUT A FANTASY...



Andrew Stellman  
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

**O'REILLY®**

# Head First C#

## Fourth Edition

by Andrew Stellman and Jennifer Greene

Copyright © 2021 Jennifer Greene, Andrew Stellman. All rights reserved.

Printed in the United States of America.

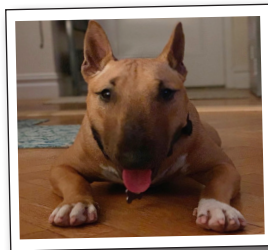
Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

<b>Series Creators:</b>	Kathy Sierra, Bert Bates
<b>Cover Designer:</b>	Ellie Volckhausen
<b>Brain Image on Spine:</b>	Eric Freeman
<b>Editors:</b>	Nicole Taché, Amanda Quinn
<b>Proofreader:</b>	Rachel Head
<b>Indexer:</b>	Potomac Indexing, LLC
<b>Illustrator:</b>	Jose Marzan
<b>Page Viewers:</b>	Greta the miniature bull terrier and Samosa the Pomeranian

## Printing History:

November 2007: First Edition.  
May 2010: Second Edition.  
August 2013: Third Edition.  
December 2020: Fourth Edition



The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C#*, and related trade dress are trademarks of O'Reilly Media, Inc.

Microsoft, Windows, Visual Studio, MSDN, the .NET logo, Visual Basic, and Visual C# are registered trademarks of Microsoft Corporation.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI]

[2020-11-13]

## 5 encapsulation

# Visual Studio for Mac Chapter 5 Projects



### Welcome to the Visual Studio for Mac projects for Chapter 5.

In this chapter, you're learning about the power of **encapsulation**, a way of programming that helps you make code that's flexible, easy to use, and difficult to misuse. You'll **make your objects' data private**, and add **properties** to protect how that data is accessed. There are two Mac projects for Chapter 5: in the first one, you'll build a Blazor app to help Owen and his role-playing game party roll for damage. But that app will go **tragically wrong**, so in the second part, you'll use **encapsulation** to fix it.



## Let's help Owen roll for damage

Owen was really happy with his sword damage console app. Now let's build a Blazor web app version of the same app to make it easier for him to use at his role-playing game night. **Create a new Blazor WebApp project called BlazorSwordDamage.** But instead of reusing the existing SwordDamage class, right-click on the project, choose **Add >> New Class...** from the menu, and **add a class called SwordDamage.**

Here's the complete code for the SwordDamage class. It contains all of the members from your console app. But this time we kept the `public SwordDamage()` method that Visual Studio for Mac automatically adds when you create a new class.

```
using System;
namespace BlazorSwordDamage
{
    public class SwordDamage
    {
        public SwordDamage()
        {
        }

        public const int BASE_DAMAGE = 3;
        public const int FLAME_DAMAGE = 2;

        public int Roll;
        public decimal MagicMultiplier = 1M;
        public int FlamingDamage = 0;
        public int Damage;

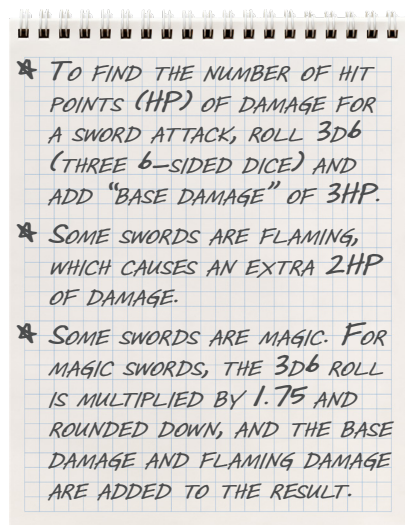
        public void CalculateDamage()
        {
            Damage = (int)(Roll * MagicMultiplier) + BASE_DAMAGE + FlamingDamage;
        }

        public void SetMagic(bool isMagic)
        {
            if (isMagic)
            {
                MagicMultiplier = 1.75M;
            }
            else
            {
                MagicMultiplier = 1M;
            }
            CalculateDamage();
        }

        public void SetFlaming(bool isFlaming)
        {
            CalculateDamage();
            if (isFlaming)
            {
                Damage += FLAME_DAMAGE;
            }
        }
    }
}
```

Visual Studio for Mac adds this method when you create a new class. You've ignored it (or even deleted it) in your previous projects. This time, make sure you leave it in.

Here's the description of the sword damage formula in Owen's game master notebook—we copied it here so you don't have to flip back to the chapter to see it.



## Here's how the HTML markup will work



Your Blazor app will have two checkboxes to set the options for flaming and magic swords, a button to roll for damage, and text to display the results of the roll.

We'll lay it out with our familiar Bootstrap tags that we used in the previous projects:

- ★ The page will have three rows. The bottom two rows have a top 5-space spacer (`mt-5`). Each row's content is centered (`justify-content-center`).
- ★ The top row will have two `col-3` columns.
- ★ The middle row will have a single `col-4` column.
- ★ The bottom row will have a single `col-6` column.

Here's a new bit of Bootstrap to help lay things out.

- ★ The left column in the top row will use the **text-left class** to align its contents to the left side.
- ★ The right column in the top row will use the **text-right class** to align its contents to the right side.
- ★ The columns in the bottom rows will have the **text-center class**, which tells them to center their contents.

The last piece of the puzzle is the markup to create a checkbox. Here's the HTML for the flaming sword checkbox:

```
<div class="col-3 text-left">
  <input class="form-check-input" type="checkbox" id="flaming" />
  <label class="form-check-label" for="flaming">
    Flaming
  </label>
</div>
```

This markup uses the same `<input>` tag that you used for sliders and other controls. Setting `type="checkbox"` tells it to create a checkbox. We also added a `<label>` to add the "Flaming" text next to the checkbox. The `for` attribute on the label matches the `id` attribute on the input, which is how the page knows which input the label is associated with.

Here's what the page looks like:

row	col-3 text-left <input type="checkbox"/> Flaming	col-3 text-right <input checked="" type="checkbox"/> Magic
row mt-5	col-4 text-center <div>Roll for damage</div>	
row mt-5	col-6 text-center <div>Rolled 11 for 22 HP</div>	

We'll give you the HTML markup for this page. But before we do, open the *Index.razor* file in your project and try creating it yourself. How far can you get on your own?

## Add the HTML markup for Owen's app

Here's the HTML markup—**add it to your app's *Index.razor* file** (delete everything in the file before you add it). Run your web app after you add it—the page won't work yet, but you'll be able to see the layout, check the boxes, and click the button.

@page "/"

Do this!

```
<div class="container">
  <div class="row justify-content-center">
    <div class="col-3 text-left">
      <input class="form-check-input" type="checkbox" id="flaming" />
      <label class="form-check-label" for="flaming">
        Flaming
      </label>
    </div>
    <div class="col-3 text-right">
      <input class="form-check-input" type="checkbox" id="magic" />
      <label class="form-check-label" for="magic">
        Magic
      </label>
    </div>
  </div>
  <div class="row justify-content-center mt-5">
    <div class="col-4 text-center">
      <button type="button" class="btn btn-primary">
        Roll for damage
      </button>
    </div>
  </div>
  <div class="row justify-content-center mt-5">
    <div class="col-6 text-center">
      <h3>Rolled 11 for 22 HP</h3>
    </div>
  </div>
</div>
```

☐ Flaming
 ☒ Magic

Roll for damage

Rolled 11 for 22 HP

You can check the boxes even though the app doesn't actually do anything yet.



## Add Blazor data binding to the HTML markup

You'll use Blazor data binding—and it will work just like your previous projects:

- ★ The checkboxes will have `@onchange` event handlers that work just like the range (slider) input in the Experiment with Controls project in Chapter 2.
- ★ The “Roll for damage” button will have an `@onclick` event handler that works just like the one for the button in the Card Picker project in Chapter 3.
- ★ The text at the bottom of the page that shows the result of the roll uses `@` to bind to a field in the code, just like you did with the guacamole price in the Sloppy Joe's menu project in Chapter 4.

Here's what you need to change in the HTML markup:

@page "/"

```
<div class="container">
  <div class="row justify-content-center">
    <div class="col-3 text-left">
      <input class="form-check-input" type="checkbox" id="flaming"
        @onchange="UpdateFlaming" />
      <label class="form-check-label" for="flaming">
        Flaming
      </label>
    </div>
    <div class="col-3 text-right">
      <input class="form-check-input" type="checkbox" id="magic"
        @onchange="UpdateMagic" />
      <label class="form-check-label" for="magic">
        Magic
      </label>
    </div>
  </div>
  <div class="row justify-content-center mt-5">
    <div class="col-4 text-center">
      <button type="button" class="btn btn-primary"
        @onclick="RollDice">
        Roll for damage
      </button>
    </div>
  </div>
  <div class="row justify-content-center mt-5">
    <div class="col-6 text-center">
      <h3>@damageText</h3>
    </div>
  </div>
</div>
```

Change this!

We added a line break between the id attribute and the new `@onchange` event handler to make it easier to read.



Watch it!

**Your project won't build if the binding doesn't work.**

*If you try to build the project after you make these changes, you'll see errors listed in the Errors window telling you that `UpdateFlaming`, `UpdateMagic`, `RollDice`, and `damageText` don't exist in the current context. We're about to give you the rest of the code to get your app to run.*

## The C# code for the Blazor damage calculator

Add this @code section to your *Index.razor* page. It creates instances of *SwordDamage* and *Random*, and makes the checkboxes and button calculate damage:

Do this!

```
@code {  
  
    Random random = new Random();  
    SwordDamage swordDamage = new SwordDamage();  
  
    string damageText = "";  
  
    private void UpdateFlaming(ChangeEventArgs e)  
    {  
        swordDamage.SetFlaming((bool)e.Value);  
        DisplayDamage();  
    }  
  
    private void UpdateMagic(ChangeEventArgs e)  
    {  
        swordDamage.SetMagic((bool)e.Value);  
        DisplayDamage();  
    }  
  
    protected override void OnInitialized()  
    {  
        swordDamage.SetMagic(false);  
        swordDamage.SetFlaming(false);  
        RollDice();  
    }  
  
    public void RollDice()  
    {  
        swordDamage.Roll = random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);  
        DisplayDamage();  
    }  
  
    void DisplayDamage()  
    {  
        damageText = "Rolled " + swordDamage.Roll + " for " + swordDamage.Damage + " HP";  
    }  
}
```

When you add a *ChangeEventArgs* argument *e* to your event handler, you can its Value member to get the value of the control that called it. You used *Value* in the *Experiment with Controls* project in Chapter 2 to get the value of several controls. A checkbox's event handler will always set *Value* to a boolean, so you'll need to cast it before using it to call the *SwordDamage* object's *SetMagic* or *SetFlaming* method.



### Ready Bake Code

You've already seen that there are *many different* ways to write the code for a specific program. For most projects in this book, it's great if you can find a different—but equally effective—way to solve the problem. But for Owen's damage calculator, we'd like you to enter the code exactly as it appears here because (spoiler alert) we've included a few bugs on purpose.

**Read through this code very carefully.**  
**Can you spot any bugs before you run it?**

## Tabletop talk (or maybe...dice discussion?)

It's game night! Owen's entire gaming party is over, and he's about to unveil his brand-new sword damage calculator. Let's see how that goes.

OKAY, PARTY,  
WE'VE GOT A NEW TABLE RULE.  
PREPARE TO BE DAZZLED BY THIS STUNNING NEW FEAT  
OF TECHNOLOGICAL AMAZE~~MENT~~.

**Jayden:** Owen, what are you talking about?

**Owen:** I'm talking about this new app that will calculate sword damage...*automatically*.

**Matthew:** Because rolling dice is so very, very hard.

**Jayden:** Come on, people, no need for sarcasm. Let's give it a chance.

**Owen:** Thank you, Jayden. This is a perfect time, too, because Brittany just attacked the rampaging were-cow with her flaming magic sword. Go ahead, B. Give it a shot.

**Brittany:** Okay. We just started the app. I checked the Magic box. Looks like it's got an old roll in there, let me click roll to do it again, and...

**Jayden:** Wait, that's not right. Now you rolled 14, but it still says 3 HP. Click it again. Rolled 11 for 3 HP. Click it some more. 9, 10, 5, all give 3 HP. Owen, what's the deal?

**Brittany:** Hey, it sort of works. If you click roll, then check the boxes a few times, eventually it gives the right answer. Looks like I rolled 10 for 22 HP.

**Jayden:** You're right. We just have to click things in a **really specific order**. *First* we click roll, *then* we check the right boxes, and *just to be sure* we check the Flaming box twice.

**Owen:** You're right. If we do things in **exactly that order**, the program works. But if we do it in any other order, it breaks. Okay, we can work with this.

**Matthew:** Or...maybe we can just do things the normal way, with real dice?



Brittany and Jayden are right. The program works, but only if you do things in a specific order. Here's what it looks like when it starts.

☐ Flaming ☐ Magic

Roll for damage

Rolled 10 for 3 HP

Let's try to calculate damage for a flaming magic sword by checking Flaming first, then Magic second. Uh-oh—that number is wrong.

☒ Flaming ☒ Magic

Roll for damage

Rolled 10 for 20 HP

But once we click the Flaming box twice, it displays the right number.

☒ Flaming ☒ Magic

Roll for damage

Rolled 10 for 22 HP



## Let's try to fix that bug

What do you think is wrong with the code? Let's experiment with it:

- ★ Start your app, then **click the button a bunch of times**. It generates a new random die roll each time and updates the text at the bottom of the page, but the calculated damage doesn't change.
- ★ **Click a checkbox**. Now the calculated damage updates.
- ★ **Click the button a bunch more times**. It still doesn't update.

So the app only updates its damage when a checkbox is clicked, but not when the button is clicked. Well, that should be easy to fix. **Make this change to the RollDice method:**

```
public void RollDice()
{
    swordDamage.Roll = random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
    swordDamage.CalculateDamage();
    DisplayDamage();
}
```

Now we'll calculate  
the damage every time  
the button is clicked.

← Fix this!

Now **test your code**. Run your program and click the button a few times. So far so good—the numbers look correct. Now **check the Magic box** and click the button a few more times. Okay, it looks like our fix worked!

But wait a minute. Something still seems weird. Try this:

1. Check the Flaming box
2. Click the button to generate a new roll – keep your eye on the damage number
3. Uncheck the Flaming box – the damage number doesn't change
4. Check the Flaming box again – now the damage number changes

That's not right at all. So the program's still broken.



WE TOOK A GUESS AND **QUICKLY** WROTE  
SOME CODE, BUT IT DIDN'T FIX THE PROBLEM  
BECAUSE **WE DIDN'T REALLY THINK** ABOUT WHAT  
ACTUALLY CAUSED THE BUG.

### Always think about what caused a bug before you try to fix it.

When something goes wrong in your code, it's **really tempting to jump right in** and immediately start writing more code to try to fix it. It may feel like you're taking action quickly, but it's way too easy to just add more buggy code. It's always safer to take the time to figure out what really caused the bug, rather than just try to stick in a quick fix.

## Use `Debug.WriteLine` to print diagnostic information

In the last few chapters you used the debugger to track down bugs, but that's not the only way developers find problems in their code. In fact, when professional developers are trying to track down bugs in their code, one of the most common things they'll do first is to **add statements that print lines of output**, and that's exactly what we'll do to track down this bug.

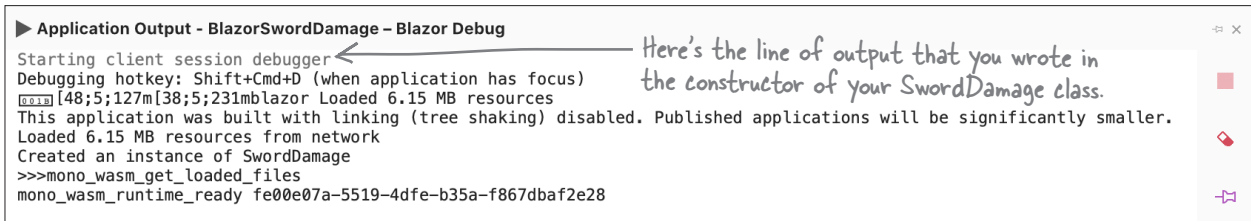
Let's make your Blazor app write a line to the output. Add this to the top of your `SwordDamage` class:

```
using System;
namespace BlazorSwordDamage
{
    public class SwordDamage
    {
        public SwordDamage()
        {
            Console.WriteLine("Created an instance of SwordDamage");
        }
    }
}
```

This method is a **constructor**. It gets called when the `SwordDamage` class is first instantiated. It doesn't have a return type and its name matches the class name. We can use it to do something when the `SwordDamage` object is created—in this case, write a line to the output.

This method is a **constructor**. When a class has a constructor, it's the very first thing that gets run when a new instance of that class is created. So when the app instantiates the class, it will write that line.

Now **run your app**, and once it's running **open the Application Output window** in Visual Studio for Mac by clicking `► Application Output - BlazorSwordDamage - Blazor Debug` at the bottom of the window. You can also open it by choosing *Other Windows >> Application Output - BlazorSwordDamage - Blazor Debug* from the View menu. Make sure you choose the application output window that ends with “- Blazor Debug” because that's the window that will show you the output from your app. It should contain the line of output that you wrote:



Any text that you print by calling `Console.WriteLine` from a Blazor web app is displayed in this window. You should only use `Console.WriteLine` for *displaying output your users should see*. Instead, any time you want to print output lines just for debugging purposes you should use **`Debug.WriteLine`**. The `Debug` class is in the `System.Diagnostics` namespace, so start by adding a `using` line to the top of your `SwordDamage` class file:

```
using System.Diagnostics;
```

Now **modify your constructor** so it uses `Debug.WriteLine` instead of `Console.WriteLine`:

```
public SwordDamage()
{
    Debug.WriteLine("Created an instance of SwordDamage");
}
```

Run your app again. You'll see the same message in the Application Output window.



## Make your app print useful diagnostic information

Writing messages to the output is actually a really useful tool for debugging—that’s why the class name is “Debug.” It works really well when you combine it with a useful C# tool called **string interpolation** that lets you easily construct strings using your app’s variables and fields.

### String interpolation

You’ve been using the `+` operator to concatenate your strings. It’s a pretty powerful tool—you can use any value (as long as it’s not `null`) and it will safely convert it to a string (usually by calling its `ToString` method). The problem is that concatenation can make your code really hard to read.

Luckily, C# gives us a great tool to concatenate strings more easily. It’s called **string interpolation**, and to use it all you need to do is put a dollar sign in front of your string. Then to include a variable, a field, or a complex expression—or even call a method!—you put it inside curly brackets. If you want to include curly brackets in your string, just include two of them, like this: `{{ }}`

Go ahead and **add a `Debug.WriteLine` statement** to the end of the `CalculateDamage` method:

```
public void CalculateDamage()
{
    Damage = (int)(Roll * MagicMultiplier) + BASE_DAMAGE + FlamingDamage;
    Debug.WriteLine($"CalculateDamage finished: {Damage} (roll: {Roll})");
}
```

Now **add another `Debug.WriteLine` statement** to the **end** of the `SetMagic` method, and one more to the **end** of the `SetFlaming` method. They should be identical to the one in `CalculateDamage`, except that they print “SetMagic” or “SetFlaming” instead of “CalculateDamage” to the output:

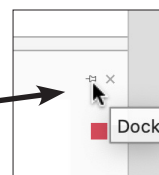
```
public void SetMagic(bool isMagic)
{
    // the rest of the SetMagic method stays the same
    Debug.WriteLine($"SetMagic finished: {Damage} (roll: {Roll})");
}

public void SetFlaming(bool isFlaming)
{
    // the rest of the SetFlaming method stays the same
    Debug.WriteLine($"SetFlaming finished: {Damage} (roll: {Roll})");
}
```

Once you dock a window, you won't see a button for it at the very bottom of the IDE anymore, but you can still get to it from the [View >> Other Windows](#) menu.

Now your program will print useful diagnostic information to the Output window. You’re ready to gather clues! ***It’s time to sleuth out this bug.***

Use the small pushpin button in the Application Output – Debug window to dock it inside one of the panels in Visual Studio for Mac—that way it will stay open when you switch between Visual Studio and your browser.



You can sleuth out this bug without setting any breakpoints. That's something developers do all the time... so you should learn how to do it, too!



## Sleuth it out

Let's use the **Output window** to debug the app. Run your program and watch the Output window. Once the app loads, press the Clear Console (🗑️) button to clear the Application Output window. Then **press the button three times**. We rolled a 12, then an 11, then an 8. This is what the app printed to our Application Output window:

Application Output - BlazorSwordDamage - Blazor Debug	Breakpoints	Locals	Watch	Threads
CalculateDamage finished: 15 (roll: 12)				
CalculateDamage finished: 14 (roll: 11)				
CalculateDamage finished: 11 (roll: 8)				

11 is the correct answer—8 plus a base damage of 3. **Press the button a bunch of times until you roll an 11**. The last line in the Application Output window should now be:

CalculateDamage finished: 14 (roll: 11)

Now let's reproduce the bug. **Check the Flaming box**—the damage should now go up to 16, which is what we'd expect, since flaming swords add 2HP of damage. The app should write this output to the Application Output window:

SetFlaming finished: 16 (roll: 11)  
CalculateDamage finished: 16 (roll: 13)

Now **press the button a bunch of times until you roll a 12**. The damage should now be 15, and the output should be:

CalculateDamage finished: 15 (roll: 12)

Here comes the bug. **Uncheck the Flaming box**. The damage will stay 15—which is *definitely not right*—and the output will end with these two lines:

CalculateDamage finished: 15 (roll: 12)  
SetFlaming finished: 15 (roll: 12)

So what happened? Thanks to our debug output, we can sleuth it out. We know that CalculateDamage finished *before* SetFlaming. We were calculating flaming damage for a roll of 12, so it should calculate 17 HP.

First it called SetFlaming, which set Damage to 17—that's correct:  $12 + 3$  (base) + 2 (flaming).

But before SetFlaming finished, it called CalculateDamage—and we see that its line comes before the SetFlaming line, so the program called the CalculateDamage method, which **overwrote the Damage field** and set it back to 15.

The problem is that **SetFlaming was called before CalculateDamage**, so even though it added the flame damage correctly, calling CalculateDamage afterward undid that. So the real reason that the program doesn't work is that the fields and methods in the SwordDamage class **need to be used in a very specific order**:

1. Set the Roll field to the 3d6 roll.
2. Call the SetMagic method.
3. Call the SetFlaming method.
4. Do not call the CalculateDamage method, because SetFlaming does that for you.

Aha! Now we actually know why the program is broken.

**Debug.WriteLine** is one of the most basic—and useful!—debugging tools in your developer toolbox. Sometimes the quickest way to sleuth out a bug in your code is to strategically add Debug.WriteLine statements to give you important clues that help you crack the case.

**And that's why the console app worked, but the Blazor app didn't.** The console app used the SwordDamage class in the specific way that it works. The Blazor app called the methods in the wrong order, so it got incorrect results.

**You're done with the Mac guide for now. You can head back to Chapter 5—resume after the “Sleuth it Out” element. We'll come back and fix this project at the end of the chapter.**



OKAY, WE KNOW THE CODE FOR OUR SWORD DAMAGE APP HAS SOME PROBLEMS. WHAT CAN WE DO ABOUT IT?

### We can use encapsulation to fix Owen's app.

At the beginning of Chapter 5, we asked you to create a Blazor web app version of the SwordDamage calculator that you made for Owen... but we ran into some trouble because the SwordDamage that we gave you had a flaw: its methods had to be called in a *very specific order* or it doesn't work—and that caused a bug in your web app. Now we'll use **encapsulation** to make the SwordDamage class work correctly no matter what order its members are used, and we'll update both the console app and Blazor web app to use the new class.

*Here's how you'll fix your app.*

#### 1 First you'll modify the SwordDamage class to make it well encapsulated.

Right now your SwordDamage class needs you to use its members in a specific order: first call the SetMagic method, then call the SetFlaming method, then set the Roll field, then call its CalculateDamage method. You'll replace that complicated set of fields and methods with something **much more simple**: you'll make the CalculateDamage method private, and you'll use properties to set flaming, magic, and the die roll—and those properties will always call CalculateDamage, so your apps can use them in any order that you want.

#### 2 Then you'll modify the console app to use the new SwordDamage class.

Your new console app will be very similar to the one from the beginning of the chapter—it just needs a few small modifications to use the new SwordDamage class.

#### 3 Finally you'll modify the Blazor web app to use the new SwordDamage class.

You'll need to make a few changes to your web app. You'll replace the checkbox event handlers from earlier:

```
<input class="form-check-input" type="checkbox" id="magic"
      @onchange="UpdateMagic" />
```

with bindings like this that automatically set properties on your app's instance of SwordDamage:

```
<input class="form-check-input" type="checkbox" id="magic"
      @bind="swordDamage.Magic" />
```

and use data binding to make your page update automatically when SwordDamage updates its properties:

```
<h3>Rolled @swordDamage.Roll for @swordDamage.Damage HP</h3>
```

When you @bind a checkbox to a boolean property or field, it automatically updates when the checkbox is checked.



## Exercise

Use what you've learned about encapsulation to fix Owen's sword damage calculator. First, modify the `SwordDamage` class to replace the fields with properties and add a constructor. Once that's done, update the console app to use it. Finally, fix the WPF app. (This exercise will go more easily if you create a new console app for the first two parts and a new WPF app for the third.)

### Part 1: Modify `SwordDamage` so it's a well-encapsulated class

1. Delete the `Roll` field and replace it with a property named `Roll` and a backing field named `roll`. The getter returns the value of the backing field. The setter updates the backing field, then calls the `CalculateDamage` method.
2. Delete the `SetFlaming` method and replace it with a property named `Flaming` and a backing field named `flaming`. It works like the `Roll` property—the getter returns the backing field, the setter updates it and calls `CalculateDamage`.
3. Delete the `SetMagic` method and replace it with a property named `Magic` and a backing field named `magic` that works exactly like the `Flaming` and `Roll` properties.
4. Create an auto-implemented property named `Damage` with a public get accessor and private set accessor.
5. Delete the `MagicMultiplier` and `FlamingDamage` fields. Modify the `CalculateDamage` method so it checks the property values for the `Roll`, `Magic`, and `Flaming` properties and does the entire calculation inside the method.
6. Add a constructor that takes the initial roll as its parameter. Now that the `CalculateDamage` method is only called from the property set accessors and constructor, there's no need for another class to call it. Make it private.
7. Add XML code documentation to all of the public class members.

### Part 2: Modify the console app to use the well-encapsulated `SwordDamage` class

1. Create a static method called `RollDice` that returns the results of a 3d6 roll. You'll need to store the `Random` instance in a static field instead of a variable so both the `Main` method and `RollDice` can use it.
2. Use the new `RollDice` method for the `SwordDamage` constructor argument and to set the `Roll` property.
3. Change the code that calls `SetMagic` and `SetFlaming` to set the `Magic` and `Flaming` properties instead.

### Part 3: Modify the Blazor app to use the well-encapsulated `SwordDamage` class

1. Copy the code from Part 1 into a new Blazor web app. Copy the HTML markup from earlier in the chapter.
2. Modify the markup:
  - Replace everything between `<h3>` and `</h3>` with to bind directly to the `SwordDamage` object  
`<h3>Rolled @swordDamage.Roll for @swordDamage.Damage HP</h3>`
  - Replace `@onchange="UpdateFlaming"` with `@bind="swordDamage.Flaming"`
  - Replace `@onchange="UpdateMagic"` with `@bind="swordDamage.Magic"`
3. Modify the code in the `@code { }` section at the bottom of your `Index.razor` file:
  - Your new `SwordDamage` class no longer has a `CalculateDamage` method, so remove lines that call it.
  - You removed the `SetFlaming` and `SetMagic` methods from your `SwordDamage` class, so remove all calls to those methods. You're not using the `UpdateFlaming` or `UpdateMagic` event handlers, so remove those too.
  - Now that you modified the "Rolled ... for ... HP" line at the bottom of the page to bind directly, you can delete the `DisplayDamage` method. Delete the `damageText` field, and all calls to the `DisplayDamage` method, too.
  - The new `SwordDamage` class has a constructor with one parameter—just pass it 10. It doesn't matter what value you use here, because you'll roll the dice when the page is initialized.

**Test everything. Use the debugger or Debug.WriteLine statements to make sure that it REALLY works.**



## Exercise Solution

Now Owen finally has a class for calculating damage that's much easier to use without running into bugs. Each property recalculates the damage, so it doesn't matter what order you call them in. Here's the code for the well-encapsulated **SwordDamage** class:

```
class SwordDamage
{
```

```
    private const int BASE_DAMAGE = 3;
    private const int FLAME_DAMAGE = 2;
```

← Since these constants aren't going to be used by any other class, it makes sense to keep them private.

```
    /// <summary>
    /// Contains the calculated damage.
    /// </summary>
    public int Damage { get; private set; }
```

← The Damage property's private set accessor makes it read-only, so it can't be overwritten by another class.

```
    private int roll;
```

```
    /// <summary>
    /// Sets or gets the 3d6 roll.
    /// </summary>
    public int Roll
```

```
    {
        get { return roll; }
        set
        {
            roll = value;
            CalculateDamage();
        }
    }
```

Here's the Roll property with its private backing field. The set accessor calls the CalculateDamage method, which keeps the Damage property updated automatically.

```
    private bool magic;
```

```
    /// <summary>
    /// True if the sword is magic, false otherwise.
    /// </summary>
    public bool Magic
```

```
    {
        get { return magic; }
        set
        {
            magic = value;
            CalculateDamage();
        }
    }
```

← The Magic and Flaming properties work just like the Roll property. They all call CalculateDamage, so setting any of them automatically updates the Damage property.

```
    private bool flaming;
```

```
    /// <summary>
    /// True if the sword is flaming, false otherwise.
    /// </summary>
    public bool Flaming
```

```
    {
        get { return flaming; }
        set
        {
            flaming = value;
            CalculateDamage();
        }
    }
```







## Exercise Solution

```

/// <summary>
/// Calculates the damage based on the current properties.
/// </summary>
private void CalculateDamage()
{
    decimal magicMultiplier = 1M;
    if (Magic) magicMultiplier = 1.75M;

    Damage = BASE_DAMAGE;
    Damage = (int)(Roll * magicMultiplier) + BASE_DAMAGE;
    if (Flaming) Damage += FLAME_DAMAGE;
}

/// <summary>
/// The constructor calculates damage based on default Magic
/// and Flaming values and a starting 3d6 roll.
/// </summary>
/// <param name="startingRoll">Starting 3d6 roll</param>
public SwordDamage(int startingRoll)
{
    roll = startingRoll;
    CalculateDamage();
}

```

← All of the calculation is encapsulated inside the `CalculateDamage` method. It only depends on the get accessors for the `Roll`, `Magic`, and `Flaming` properties.

← The constructor sets the backing field for the `Roll` property, then calls `CalculateDamage` to make sure the `Damage` property is correct.

Here's the code for the Main method of the console app:

```

class Program
{
    static Random random = new Random();

    static void Main(string[] args)
    {
        SwordDamage swordDamage = new SwordDamage(RollDice());
        while (true)
        {
            Console.WriteLine("0 for no magic/flaming, 1 for magic, 2 for flaming, " +
                               "3 for both, anything else to quit: ");
            char key = Console.ReadKey().KeyChar;
            if (key != '0' && key != '1' && key != '2' && key != '3') return;
            swordDamage.Roll = RollDice();
            swordDamage.Magic = (key == '1' || key == '3');
            swordDamage.Flaming = (key == '2' || key == '3');
            Console.WriteLine($"{random.Next(1, 7)} Rolled {swordDamage.Roll} for {swordDamage.Damage} HP\n");
        }
    }

    private static int RollDice()
    {
        return random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
    }
}

```

← It made sense to move the 3d6 roll into its own method since it's called from two different places in `Main`. If you used "Generate method" to create it, the IDE made it private automatically.



## Exercise Solution

Here's the complete *Index.razor* file for your Blazor web app, including HTML markup and C# code. Did you notice how much less C# code you need in it? That's one way well-encapsulated classes help you write better code—you don't need to write as much additional code to use them.

@page "/"

```
<div class="container">
  <div class="row justify-content-center">
    <div class="col-3 text-left">
      <input class="form-check-input" type="checkbox" id="flaming"
        @bind="swordDamage.Flaming" />
      <label class="form-check-label" for="flaming">
        Flaming
      </label>
    </div>
    <div class="col-3 text-right">
      <input class="form-check-input" type="checkbox" id="magic"
        @bind="swordDamage.Magic" />
      <label class="form-check-label" for="magic">
        Magic
      </label>
    </div>
  </div>
  <div class="row justify-content-center mt-5">
    <div class="col-4 text-center">
      <button type="button" class="btn btn-primary"
        @onclick="RollDice">
        Roll for damage
      </button>
    </div>
  </div>
  <div class="row justify-content-center mt-5">
    <div class="col-6 text-center">
      <h3>Rolled @swordDamage.Roll for @swordDamage.Damage HP</h3>
    </div>
  </div>
</div>
```

Your checkboxes now use @bind to set *SwordDamage* properties. You don't need extra event handlers any more, so your code is simpler.

When the user clicks a checkbox, the binding updates the property, which does the calculation and sets the Roll and Damage properties so they're updated automatically.

@code {

```
Random random = new Random();
SwordDamage swordDamage = new SwordDamage(10);

protected override void OnInitialized()
{
    RollDice();
}

public void RollDice()
{
    swordDamage.Roll = random.Next(1, 7) + random.Next(1, 7) + random.Next(1, 7);
}
}
```

Now all the page needs to do is calculate a new die roll when the page loads or the user clicks the button.

We kept the *Random* instance and the code to generate the random 3d6 roll in the *Index.razor* page. Do you think that was a good decision? Or would it make more sense to add a static *Random* field to the *SwordDamage* class and move the *RollDice* method into that class? There's not necessarily a right or wrong answer. That's is a good example of code aesthetics—beauty is in the eye of the beholder.