O'REILLY®

~~Fourth~~ Fifth Edition

Head First C#

This is part of an early release preview of the 5th edition of Head First C# by Andrew Stellman and Jenny Greene. We'll release the final version of this PDF when the book is published in late 2023.

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

Andrew Stellman
& Jennifer Greene

This is the .NET MAUI project from Chapter 5. At the beginning of the chapter you'll create an app to calculate damage for a tabletop role-playing game—but the app has a bug! This PDF includes that first part of the project, then skips to the second part at the end of the chapter were you use what you learned about encapsulation to fix the bug.

A Brain-Friendly Guide

Let's help Owen roll for damage

Owen was so happy with his ability score calculator that he wanted to create more C# programs he can use for his games, and you're going to help him. In the game he's currently running, any time there's a sword attack he rolls dice and uses a formula that calculates the damage. Owen wrote down how the **sword damage formula** works in his game master notebook.

Here's the description of the sword damage formula in Owen's game master notebook.

- To find the number of hit points (HP) of damage for a sword attack, roll 3d6 (three 6-sided dice) and add "base damage" of 3HP.
- Some swords are flaming, which causes an extra 2HP of damage.
- Some swords are magic. For magic swords, the 3d6 roll is multiplied by 1.75 and rounded down, and the base damage and flaming damage are added to the result.

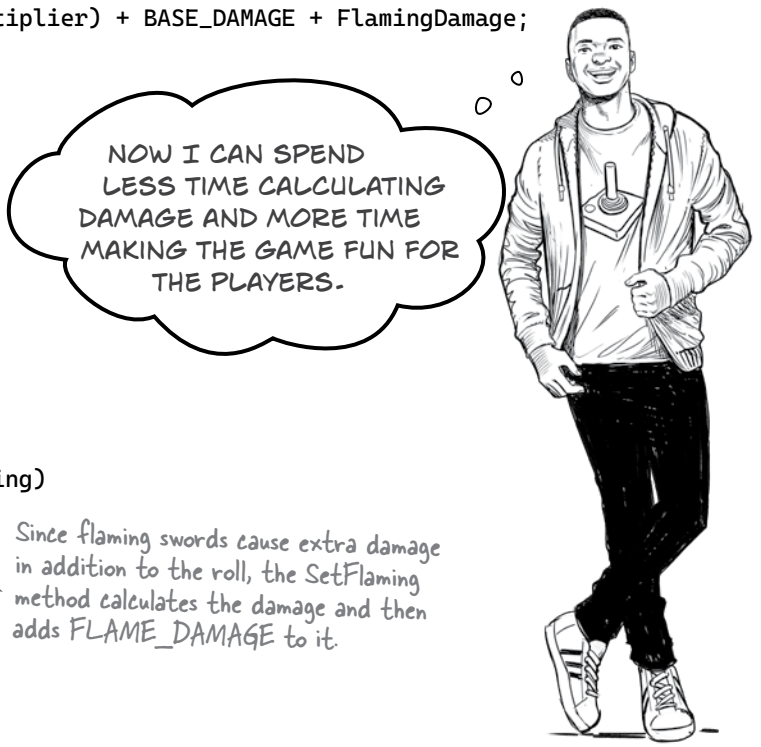
Here's **a class called SwordDamage** that does the calculation. Read through the code carefully—you're about to create an app that uses it.

```
class 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;
        }
    }
}
```

Here's a useful C# tool. Since the base damage or flame damage won't be changed by the program, you can use the `const` keyword to declare them as constants, which are like variables except that their value can never be changed. If you write code that tries to change a constant, you'll get a compiler error.

Here's where the damage formula gets calculated. Take a minute and read the code to see how it implements the formula.

Since flaming swords cause extra damage in addition to the roll, the SetFlaming method calculates the damage and then adds FLAME_DAMAGE to it.



Create a console app to calculate damage

Let's build a console app for Owen that uses the SwordDamage class. It will print a prompt to the console asking the user to specify whether the sword is magic and/or flaming, then it will do the calculation. Here's an example of the output of the app:

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 0
Rolled 11 for 14 HP
```

Rolling 11 for a non-magic, non-flaming sword will cause 11 + 3 = 14 HP of damage.

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 0
Rolled 15 for 18 HP

0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 1
Rolled 11 for 22 HP
```

Rolling 11 for a magic sword will cause (round down 11 × 1.75 = 19) + 3 = 22

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 1
Rolled 8 for 17 HP

0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 2
Rolled 10 for 15 HP
```

Rolling 17 for a magic flaming sword causes (round down 17 × 1.75 = 29) + 3 + 2 = 34.

```
0 for no magic/flaming, 1 for magic, 2 for flaming, 3 for both, anything else to quit: 3
Rolled 17 for 34 HP
```

Exercise

Draw a class diagram for the SwordDamage class. **Then create a new console app** and add the SwordDamage class. While you're carefully entering the code, take a really close look at how the SetMagic and SetFlaming methods work, and how they work a little differently from each other. Once you're confident you understand it, you can build out the top-level statements. Here's what they'll do:

Create a new instance of the SwordDamage class.

Write the prompt to the console and read the key. Call Console.ReadKey(false) so the key that the user typed is printed to the console. If the key isn't 0, 1, 2, or 3, **`return`** to end the program.

Roll 3d6 by calling Random.Shared.Next(1, 7) three times and adding the results together, and set the Roll field.

If the user pressed 1 or 3 call SetMagic(true): otherwise call SetMagic(false). You don't need an **`if`** statement to do this: `key == '1'` returns true, so you can use `||` to check the key directly inside the argument.

If the user pressed 2 or 3, call SetFlaming(true); otherwise call SetFlaming(false). Again, you can do this in a single statement using == and `||`.

Write the results to the console. Look carefully at the output and use `\n` to insert line breaks where needed.

Exercise Solution

SwordDamage
Roll MagicMultiplier FlamingDamage Damage
CalculateDamage SetMagic SetFlaming

This console app rolls for damage by creating a new instance of the SwordDamage class that we gave you and generating output that matches the example.

```
SwordDamage swordDamage = new SwordDamage();
while (true)
{
    Console.Write("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;
    int roll = Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7)
        + Random.Shared.Next(1, 7);
    swordDamage.Roll = roll;
    swordDamage.SetMagic(key == '1' || key == '3');
    swordDamage.SetFlaming(key == '2' || key == '3');
    Console.WriteLine("\nRolled " + roll + " for " + swordDamage.Damage + " HP\n");
}
```

Yes! We can build a MAUI app that uses the same class.

Let's find a way to **reuse** the SwordDamage class in a MAUI app. The first challenge for us is how to provide an *intuitive* user interface. A sword can be magic, flaming, both, or none, so we need to figure out how we want to handle that in the UI—and there are a lot of options.

One way to design it would be to use a picker with four options, like this:

But that's a little... weird? There's got to be a better way to design the app, right?

Design a MAUI version of the damage calculator app

Let's build a .NET MAUI damage calculator app for Owen. We'll give you the code-behind C# code for it. Your job will be to create the XAML that works with the C# code.

In this project, you'll be working with two new things that you haven't used yet:

- Your app will use two **checkboxes**. A checkbox is a control that should be very familiar to you—it's a box that displays a check when you click it, and is empty when you click it again. In MAUI, the Checkbox control has a Boolean value that's true if the box is checked and false if the box is not unchecked.
- The C# code in your app will use **string interpolation** to build a string to display to the user. You've been using the + operator to build strings by concatenating values together. String interpolation does the same thing, but in a way that's easier to read.

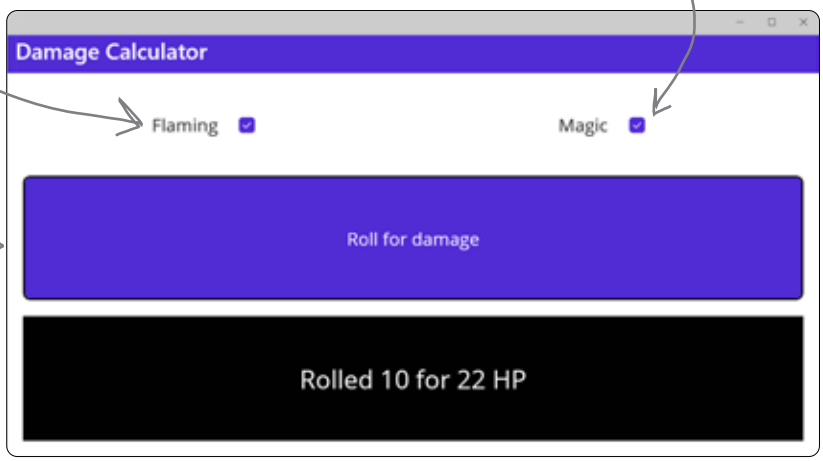
How your damage calculator app will work

Here's the main page for the damage calculator. It has two Checkbox controls to turn flaming and magic on and off, a Button to roll for damage, and a Label to display the results.

- When you click the button, it generates three random numbers to do a 3d6 roll (just like the console app did), then uses the SwordDamage class to display the damage.
- Clicking on a checkbox causes the label to update automatically. When you check or uncheck either of the checkboxes, it updates the SwordDamage fields, recalculates the damage, and updates the label.

When you check the Flaming box, it calls the SwordDamage.SetFlaming, passing it true if the box is checked and false if it's unchecked, and then calls a method to update the label to display the damage.

The Magic checkbox works just like the Flaming one, except it calls SetMagic instead of SetFlaming.



Clicking the button does a new random 3d6 roll, then updates the Roll field and displays the damage.

Exercise

It's not easy to find—don't feel bad if you don't see it yet!

There's a bug in the code for this app! Can you spot it?

Create a new MAUI app called **SwordDamageCalculator**. Here's the code-behind for the MainPage.xaml.cs file:

```
public partial class MainPage : ContentPage
{
    SwordDamage swordDamage = new SwordDamage();

    public MainPage()
       {
             InitializeComponent();
        swordDamage.SetFlaming(Flaming.IsChecked);
        swordDamage.SetMagic(Magic.IsChecked);
        RollDice();
    }

    private void RollDice()
    {
        swordDamage.Roll = Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7)
            + Random.Shared.Next(1, 7);
        DisplayDamage();
    }

    private void DisplayDamage()
    {
        Damage.Text = $"Rolled {swordDamage.Roll} for {swordDamage.Damage} HP";
    }

    private void Flaming_CheckedChanged(object sender, CheckedChangedEventArgs e)
    {
        swordDamage.SetFlaming(e.Value);
        DisplayDamage();
    }

    private void Magic_CheckedChanged(object sender, CheckedChangedEventArgs e)
    {
        swordDamage.SetMagic(e.Value);
        DisplayDamage();
    }

    private void Button_Clicked(object sender, EventArgs e)
    {
        RollDice();
    }
}
```

The Checkbox control has a IsChecked property with a Boolean value that's true if the box is checked and false if it's not checked. When the page first loads, it uses that property to set the fields on the SwordDamage object match the checkboxes. Then it calls the RollDice method, which rolls 3d6 (just like your console app did) and calls a method to display the calculated damage.

The DisplayDamage method uses string interpolation to create the string and display it in a label. To use string interpolation, put a $ in front of a string, and any values to insert into the string in {curly brackets}.

The Checkbox control has a CheckChanged event that gets called any time it's checked or unchecked. You can get the checkbox value using `e.Value`.

Clicking the button calls the same RollDice method that the page called when it first loaded.

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: `{{ }}`

We didn't forget about accessibility! We wanted to concentrate on giving you practice laying out a grid on your own. You'll add semantic properties to the app later in the chapter. But this is a great time to add them on your own if you want to.

Exercise

Get the C# code ready for the SwordDamageCalculator app.

- Add the SwordDamage class to your app.
- Add the code-behind C# code we gave you to MainPage.xaml.cs file.
- Modify AppShell.xaml to change the title to `"Damage Calculator"`.

Write the XAML for the SwordDamageCalculator app's main page.

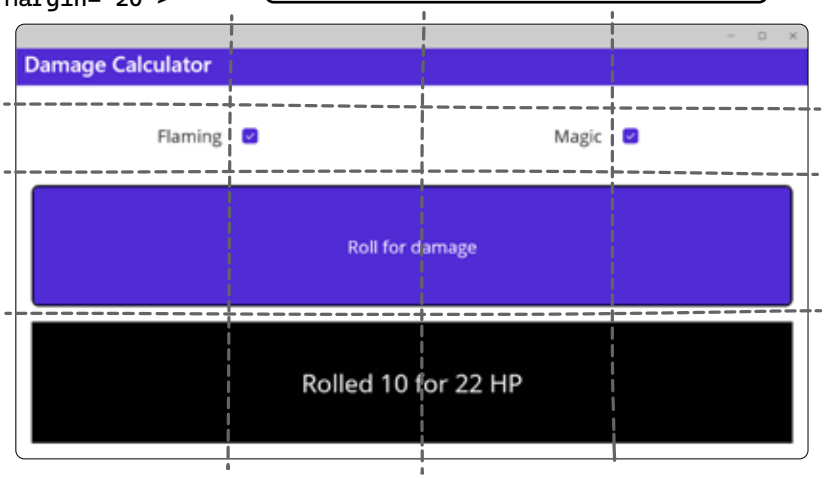
The main page uses a grid:

We gave the Grid a margin of 20 so it has a little space around all of the controls.

```
<ContentPage ... >
    <ScrollView>
        <Grid Margin="20">
```

This grid has four columns of equal width.

This grid has three rows. The bottom two rows are twice as high as the top row.




```
        </Grid>
    </ScrollView>
</ContentPage>
```

Add two Label controls and two Checkbox controls to the top row (Grid.Row="0"), a Button control to the middle row (Grid.Row="1"), and a Label to the bottom row (Grid.Row="2") and set their properties so they match the screenshot:

- The Flaming and Magic labels have these properties: `FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"` – this causes them to be centered vertically but moved to the right of the cell.
- Give the Checkbox controls x:Name values of "Flaming" and "Magic" and have CheckChanged events call the event handler methods that we gave you. Set these properties to move them to the left of the cell: `VerticalOptions="Center" HorizontalOptions="Start" Margin="20,0,0,0"`
- We gave the Button control a 2-pixel black border using its BorderColor and BorderWidth properties and set its font size to Medium. It spans all 4 columns in the row. Make it call the Clicked event handler that we gave you.
- The Label at the bottom has these properties: `x:Name="Damage" Margin="0,20,0,0" BackgroundColor="Black" TextColor="White" FontSize="Large" VerticalTextAlignment="Center" HorizontalTextAlignment="Center"`

Exercise Solution

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SwordDamageCalculator.MainPage">

    <ScrollView>

        <Grid Margin="20">
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition Height="2*"/>
                <RowDefinition Height="2*"/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition/>
                <ColumnDefinition/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>

            <Label Text="Flaming"
                   FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"/>

            <CheckBox x:Name="Flaming" Grid.Column="1"
                      VerticalOptions="Center" HorizontalOptions="Start"
                      CheckedChanged="Flaming_CheckedChanged" Margin="20,0,0,0" />

            <Label Text="Magic" Grid.Column="2"
                   FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"/>

            <CheckBox x:Name="Magic" Grid.Column="3"
                      VerticalOptions="Center" HorizontalOptions="Start"
                      CheckedChanged="Magic_CheckedChanged" Margin="20,0,0,0" />

            <Button Grid.Row="1" Grid.ColumnSpan="4" Margin="0,20,0,0"
                    BorderColor="Black" BorderWidth="2"
                    Text="Roll for damage" FontSize="Medium" Clicked="Button_Clicked"/>

            <Label x:Name="Damage" Grid.Row="2" Grid.ColumnSpan="4" Margin="0,20,0,0"
                BackgroundColor="Black" TextColor="White" FontSize="Large"
                VerticalTextAlignment="Center" HorizontalTextAlignment="Center" />
        </Grid>

    </ScrollView>

</ContentPage>
```

The bottom two rows are twice as tall as the top row.

The four columns are all the same width.

We asked you to set the HorizontalOptions properties for these Label and Checkbox controls so the Labels are at the right sides of their cells and the Checkboxes are at the left sides so they're close to each other.

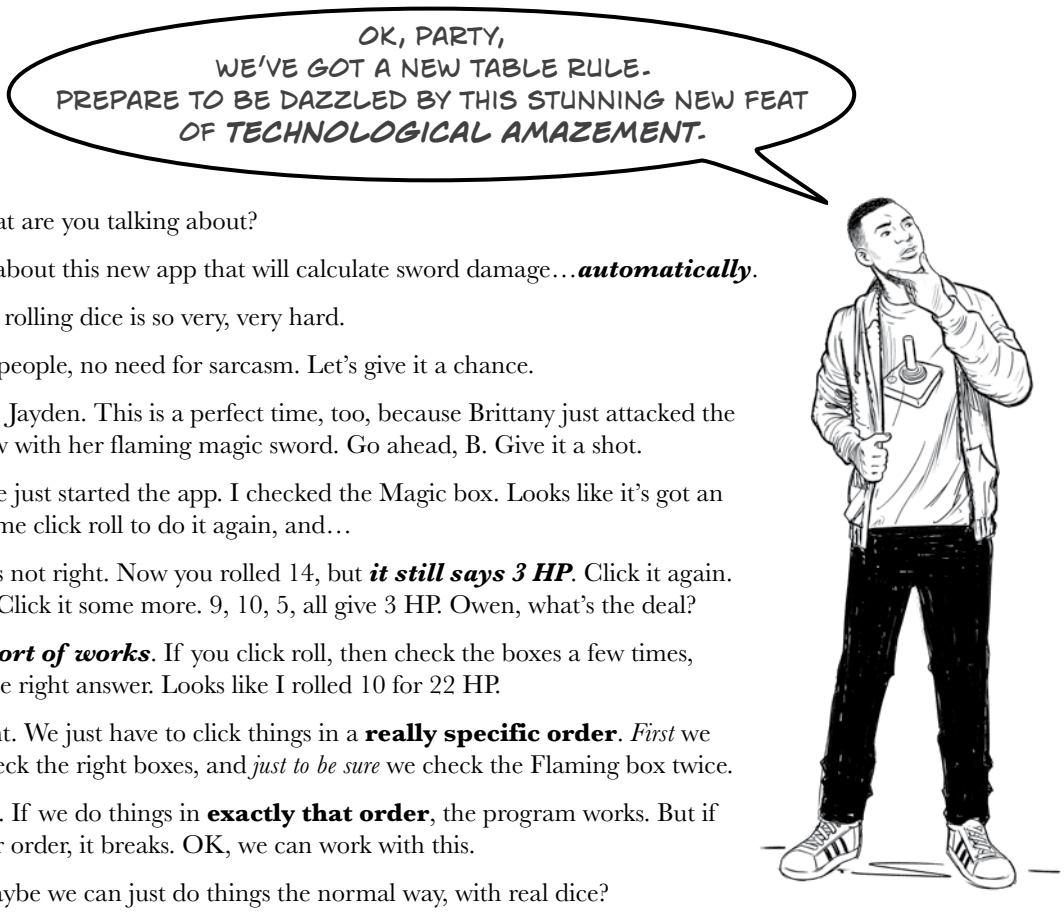
The Checkbox controls call the CheckChanged event handlers in the C# code-behind that we gave you.

We gave you many of the properties to help you make your app match our screenshot. But this is a great chance to get creative! Try different colors, font sizes, horizontal and vertical options, and even grid layouts. Experimenting is a great way to get these ideas into your brain quickly.

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.

We told you there was a bug in the code! How do you think you'll fix it?



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?

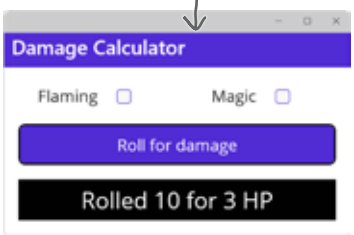
Brittany: Hey, ***it sort of works***. If you click roll, then check the boxes a few times, eventually it gives the right answer. Looks like I rolled 10 for 22 HP.

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.

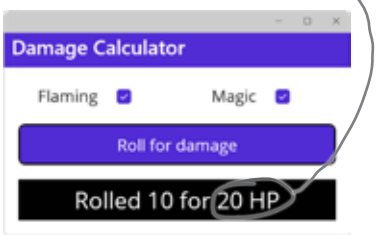
Owen: You're right. If we do things in **exactly that order**, the program works. But if we do it in any other order, it breaks. OK, we can work with this.

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

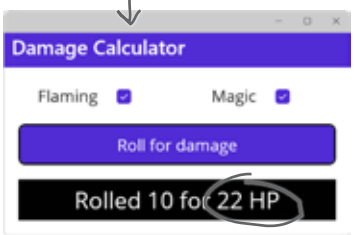
Brittany and Jayden are right. The program works, but only if you do things in a specific order. When it starts up something's wrong—it always calculates 3HP.



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



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



Let's try to fix that bug

When you run the program, what's the first thing that it does? Let's take a closer look at this method at the very top of the MainPage class in the code-behind for the page in MainPage.xaml.cs:

```
public partial class MainPage : ContentPage
{
    SwordDamage swordDamage = new SwordDamage();

    public MainPage()
    {
        InitializeComponent();
        swordDamage.SetMagic(Flaming.IsChecked);
        swordDamage.SetFlaming(Magic.IsChecked);
        RollDice();
    }
```

This method is a constructor. It doesn't have a return type and its name matches the class name. Your MAUI page is actually an instance of a class called MainPage. It gets called when the MainPage class is first instantiated, so we can use it to initialize the instance.

You've been using this method since Chapter 1. Now you know its name: it's 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. When your app starts up and creates an instance of MainPage, first it initializes the fields—including creating a new SwordDamage object—and then it calls the constructor.

The program calls RollDice just before showing you the window, and we see the problem every time we click roll, so ***maybe*** we can fix this by guessing what's wrong, and ttrying out a possible solution to the problem?

Make these changes to the RollDice method so it sets the SwordDamage fields just like the constructor does:

Redo this!

```
public void RollDice()
{
    swordDamage.Roll = Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7)
        + Random.Shared.Next(1, 7);
    swordDamage.SetFlaming(Flaming.IsChecked);
    swordDamage.SetMagic(Magic.IsChecked);
    DisplayDamage();
}
```

Maybe we just need to call the SwordDamage object's methods before we display the damage. Will that fix the bug?

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. OK, it looks like our fix worked! There's just one more thing to test. **Check the Flaming box** and click the button and…

Oops! It's still not working

It's still not working. When you click the button, it does the 1.75 magic multiplier, but it doesn't add the extra 3 HP for flaming. You still need to check and uncheck the Flaming checkbox to get the right number. So the app is 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.

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.

Open the Output window in Visual Studio by choosing ***Output*** (Ctrl+W O) from the View menu in Wndows, or ***Application Output*** from the View >> Other Windows menu on a Mac.

Luckily, .NET gives us a really useful method for this: **Debug.WriteLine**. Any time you want to print output lines just for debugging purposes, call that method. 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;
```

Next, **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 set Damage to {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 = {Damage} (roll: {Roll})");
}

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

Run your app again. Now you'll see debug messages in the application output window.

Open the Output window to see the debug messages. If you're on a Mac, check the bottom of the Visual Studio window for an "Application Output" tab.

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. Diagnostic tools like Debug.WriteLine are there to help you think through the problems.

You can sleuth out this bug without setting any breakpoints. Developers do this all the time. Now you can to do it too!

Sleuth it Out

A Scandal at the Gaming Table

Let's use the **Output window** to debug the app. You can always press the Clear All (or) button to clear it.

Run your program and watch the output. As it loads, you'll see a bunch of lines with messages about the CLR loading various DLLs (that's normal, just ignore them for now). Then you'll see this output:

```
CalculateDamage set Damage to 3 (roll: 0)
SetMagic finished: Damage = 3 (roll: 0)
```

These two lines will be printed twice. Can you figure out why?

That gives us a clue about the first mystery—why does the app always show 3 HP when it starts up, no matter what the roll is? It's because it's using 0 for the roll. ***It must not be setting the SwordDamage.Roll property.***

Let's keep looking for more clues. Check the Flaming box. When we did it, our was 14, so this is what it printed:

```
CalculateDamage set Damage to 17 (roll: 14)
SetFlaming finished: Damage = 19 (roll: 14)
```

Make sure you modified RollDice to call SetFlaming and SetMagic, or you won't see this output.

19 is the correct answer for that roll: 14 plus base damage of 3 plus 2 for a flaming sword. So far so good.

Thanks to the debug messages, you can see what methods are being called. The SetFlaming method called CalculateDamage, which calculated 17 and then executed the first Debug.WriteLine statement. It then added FLAME_DAMAGE (2) to get the value up to 19, and finally executed the second Debug.WriteLine statement.

Now press the button to roll again. The program should write four more lines to the Output window:

```
CalculateDamage set Damage to 15 (roll: 12)
SetFlaming finished: Damage = 17 (roll: 12)
CalculateDamage set Damage to 15 (roll: 12)
SetMagic finished: Damage = 15 (roll: 12)
```

Wait a minute! SetFlaming added 2 to Damage to set it to 17. But then SetMagic called CalculateDamage, which set it back to 15.

We rolled a 12, so it should calculate 17 HP. But it didn't—it calculated 15. What does the debug output tell us?

First the RollDice method called SetFlaming, which calculated 17 just like before.

But ***then*** it 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***:

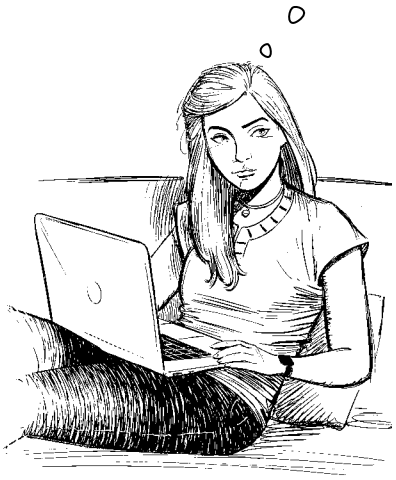
1. Set the Roll field to the 3d6 roll.
2. Call the SetMagic method.
3. Call the SetFlaming method.
4. Do not call the CalculateDamage method. SetFlaming does that for you.

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 MAUI version didn't. The console app used the SwordDamage class in the specific way that it works. The MAUI app ***called them in the wrong order***, so it got incorrect results.

Now we know what caused the bug! But we won't fix this project yet. We'll return to fix the project after learning more about encapsulation.

SO THE METHODS JUST NEED TO BE **CALLED IN A PARTICULAR ORDER**. WHAT'S THE BIG DEAL? I JUST NEED TO FLIP AROUND THE ORDER THAT I CALL THEM, AND MY CODE WILL START WORKING.



People won't always use your classes in exactly the way you expect.

And most of the time those "people" who are using your classes are you! You might be writing a class today that you'll be using tomorrow, or next month. Luckily, C# gives you a powerful technique to make sure your program always works correctly—even when people do things you never thought of. It's called **encapsulation** and it's really helpful for working with objects. The goal of encapsulation is to restrict access to the "guts" of your classes so that all of the class members are ***safe to use and difficult to misuse***. This lets you design classes that are much more difficult to use incorrectly—and that's a ***great way to prevent bugs*** like the one you sleuthed out in your sword damage calculator.

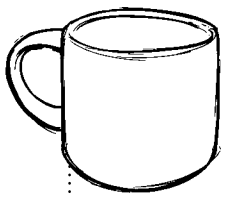
there are no Dumb Questions

Q: What's the difference between Console.WriteLine and Debug.WriteLine?

A: The Console class is used by console apps to get input from and send output to the user. It uses the three **standard streams** provided by your operating system: standard input (stdin), standard output (stdout), and standard error (stderr). Standard input is the text that goes into the program, and standard output is what it prints. (If you've ever piped input or output in a shell or command prompt using <, >, |, <<, >>, or || you've used stdin and stdout.) The Debug class is in the System.Diagnostics namespace, which gives you a hint about its use: it's for helping diagnose problems by tracking down and fixing them. Debug.WriteLine sends its output to **trace listeners**, or special classes that monitor diagnostic output from your program and write them to the console, log files, or a diagnostic tool that collects data from your program for analysis.

Q: Can I use constructors in my own code?

A: Absolutely. A constructor is a method that the CLR calls when it first creates a new instance of an object. It's just an ordinary method—there's nothing weird or special about it. You can add a constructor to any class by declaring a method **without a return type** (so no void, int, or other type at the beginning) that has the **same name as the class**. Any time the CLR sees a method like that in a class, it recognizes it as a constructor and calls it any time it creates a new object and puts it on the heap.



Relax

We're not done with constructors.

We'll do a lot more work with constructors later in the chapter. For now, just think of a constructor as a special method that you can use to initialize an object.

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 MAUI app. (This exercise will go more easily if you create a new console app for the first two parts and a new MAUI 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. Add a method to Program.cs called RollDice that returns the results of a 3d6 roll.
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 MAUI app to use the well-encapsulated SwordDamage class

1. Copy the code from Part 1 into a new MAUI app. Copy the XAML from the project earlier in the chapter.
2. Create a RollDice method that sets swordDamage.Roll to the results of a 3d6 roll, then calls DisplayDamage.
3. Make the Button_Clicked event handler call the RollDice method (and do nothing else).
4. Your SwordDamage class now has a constructor that takes one argument. Change the declaration so it looks like this (it doesn't matter what number you use, because the app will do a new random roll in the constructor):

```
SwordDamage swordDamage = new SwordDamage(3);
```

5. In the MainPage constructor, set the SwordDamage object's Flaming and Magic properties so they match the checkboxes. Then call the RollDice method to roll and display the damage.
6. Change the CheckedChanged event handlers for both checkboxes to use the Magic and Flaming properties instead of the old SetMagic and SetFlaming methods, then call DisplayDamage.
7. **Make your app more accessible.** Add SemanticProperties.Description properties to the checkbox labels and SemanticProperties.Hint properties to the checkboxes and buttons. Use the screen reader to test it.

Test everything. Use the debugger or Debug.WriteLine statements to make sure that it all 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;
    private bool magic;
    private bool flaming;
    private int roll;

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

    /// <summary>
    /// Sets or gets the 3d6 roll.
    /// </summary>
    public int Roll
    {
        get { return roll; }
        set
        {
            roll = value;
            CalculateDamage();
        }
    }

    /// <summary>
    /// True if the sword is magic, false otherwise.
    /// </summary>
    public bool Magic
    {
        get { return magic; }
        set
        {
            magic = value;
            CalculateDamage();
        }
    }

    /// <summary>
    /// True if the sword is flaming, false otherwise.
    /// </summary>
    public bool Flaming
    {
        get { return flaming; }
        set
        {
            flaming = value;
            CalculateDamage();
        }
    }
```

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

Here are the backing fields for the Roll, Magic, and Flaming.

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

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

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.

```
    /// <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 top-level statements in Program.cs for the console app:

```
SwordDamage swordDamage = new SwordDamage(RollDice());
while (true)
{
    Console.Write("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($"\nRolled {swordDamage.Roll} for {swordDamage.Damage} HP\n");
  }

int RollDice()
{
    return Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7);
}
```

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

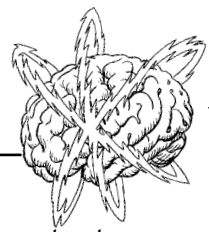
Exercise Solution

Here's the code-behind in the MainPage.xaml.cs file.

It doesn't matter what we pass to the SwordDamage constructor, because we do a new roll in the page constructor. Is this the best way to handle this?

```
using System.Diagnostics;
public partial class MainPage : ContentPage
{
    SwordDamage swordDamage = new SwordDamage(3);

    public MainPage()
    {
        InitializeComponent();
        swordDamage.Flaming = Flaming.IsChecked;
        swordDamage.Magic = Magic.IsChecked;
        RollDice();
    }

    private void RollDice()
    {
        swordDamage.Roll = Random.Shared.Next(1, 7) + Random.Shared.Next(1, 7)
            + Random.Shared.Next(1, 7);
        DisplayDamage();
    }

    private void DisplayDamage()
    {
        Damage.Text = $"Rolled {swordDamage.Roll} for {swordDamage.Damage} HP";
    }

    private void Flaming_CheckedChanged(object sender, CheckedChangedEventArgs e)
    {
        swordDamage.Flaming = e.Value;
        DisplayDamage();
    }

    private void Magic_CheckedChanged(object sender, CheckedChangedEventArgs e)
    {
        swordDamage.Magic = e.Value;
        DisplayDamage();
    }

    private void Button_Clicked(object sender, EventArgs e)
    {
        RollDice();
    }
}
```

The constructor sets the SwordDamage object's Flaming and Magic properties to match the checkboxes, then does an initial roll.



Brain Power

We had you pass 3 to the SwordDamage constructor because the MainPage constructor will call RollDice and set the Roll property to a new value. *This is a little messy!* Can you think of a better, more readable way to handle this?

Make the screen reader announce each roll

Do this!

Turn on the app, start the screen reader, and close your eyes. With the semantic properties, it should be pretty easy to use. Announcements can makes a big difference for people who depend on a screen reader. Wouldn't it be great if it always announced the result every time you clicked the button to roll for damage?

Luckily, it's easy to do that! The **SemanticScreenReader.Default.Announce** method tells the screen reader to announce text. Add it to your DisplayDamage method to announce the roll, then use your operating system's screen reader to test it out.

```
private void DisplayDamage()
{
    Damage.Text = $"Rolled {swordDamage.Roll} for {swordDamage.Damage} HP";
    SemanticScreenReader.Default.Announce(Damage.Text);
}
```

Accessibility is important! You don't need to wait for us to tell you to add semantic properties or test your app with a screen reader. Thinking about accessibility early is a great way to improve your overall coding skills!

Exercise Solution

Here are the semantic propertes we asked you to add to make the app more accessible:

```
<Label Text="Flaming"
       FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"/>

<CheckBox x:Name="Flaming" Grid.Column="1"
          VerticalOptions="Center" HorizontalOptions="Start"
          SemanticProperties.Description="Determines if the sword is flaming"
          CheckedChanged="Flaming_CheckedChanged" Margin="20,0,0,0" />

<Label Text="Magic" Grid.Column="2"
       FontSize="Medium" VerticalOptions="Center" HorizontalOptions="End"/>

<CheckBox x:Name="Magic" Grid.Column="3"
          VerticalOptions="Center" HorizontalOptions="Start"
          SemanticProperties.Description="Determines if the sword is flaming"
          CheckedChanged="Magic_CheckedChanged" Margin="20,0,0,0" />

<Button Grid.Row="1" Grid.ColumnSpan="4" Margin="0,20,0,0"
       BorderColor="Black" BorderWidth="2"
       SemanticProperties.Hint="Throws a 3d6 roll for damage"
       Text="Roll for damage" FontSize="Medium" Clicked="Button_Clicked"/>

<Label x:Name="Damage" Grid.Row="2" Grid.ColumnSpan="4" Margin="0,20,0,0"
       BackgroundColor="Black" TextColor="White" FontSize="Large"
       SemanticProperties.Hint="The results of the damage roll "
       VerticalTextAlignment="Center" HorizontalTextAlignment="Center" />
```