Bonus Level A

Visual Studio Overview

Speedrun

- The Code Window is where you edit code. It makes it easy to navigate your code, IntelliSense helps you know what types and members are available and how they work, and Quick Actions provide you with a way to fix compiler errors and warnings and refactor code.
- The Solution Explorer shows you a high-level view of your solution, project, and file structure.
- The Properties window lets you edit the properties of anything that is selected.
- The Error List shows you the errors and warnings in your current code.
- Visual Studio is highly configurable. Most configuration is done through the Options Dialog.

Mastering the IDE that you are using is worth the time. You may spend hours in it every day. Investing time to get good with it is common sense. Visual Studio is a huge program. A single level can't cover everything, but we'll cover the basics.

Windows

The Visual Studio user interface is essentially just a collection of windows, where each window provides a different view of your program's state. Initially, the various windows' complexity can be intimidating, but you can focus on one window at a time to learn how it works.

Several of Visual Studio's windows are open initially, but there are many more hidden from view until you need them. We will not cover every window here, but you can find long lists of

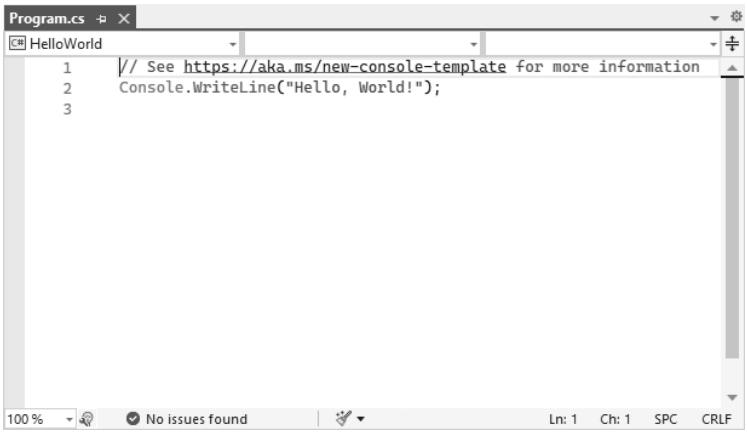
menu items that will open up additional windows to work with by going through the main menu.

You can resize, rearrange, and pop out windows to get the arrangement that works best for you. After you have made a complete mess and want a clean slate, you can go to **Window** > **Reset Window Layout** to get back to the default view.

In the rest of this section, we'll look at the most versatile windows of Visual Studio. A few others are covered in other parts of this book.

The Code Window

The code window is the main part of Visual Studio's editor and where you spend most of your time.

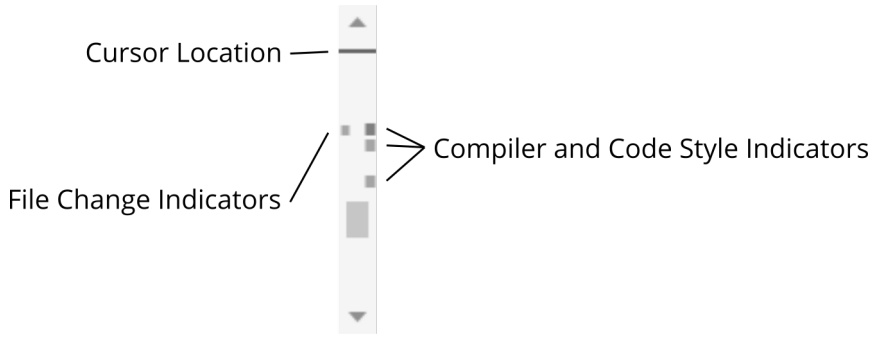


This window appears straightforward, but a lot of power is hidden away in context menus and popups that only appear when needed. We will look at some of this in more detail in a minute.

The code window is a multi-tab document view, allowing you to open up many source code files simultaneously and switch between them. It has many features common to multi-tab views. For example, you can right-click on the tabs at the top to access commands like **Close All Tabs** or **Close All But This**. You can pin specific tabs and **Close All But Pinned**. You can do **New Horizontal Document Group** and **New Vertical Document Group** to view two files side by side (or one above the other). Your first programs may contain only a single file, so these features may not get much use initially. But before long, you will spread your programs over many files, at which point, these tools become very useful.

On the left are line numbers, which are helpful when talking with people about your code ("Line 17 looks weird to me"). Line numbers are also helpful when resolving compiler errors and debugging because line numbers are often displayed. Next to that are little boxes **+** and **−** in them. This feature is called *code folding*, and you can use it to expand and collapse sections of code.

A vertical scrollbar on the right side of an editor tab embeds some helpful information beyond what scrollbars typically have.



The cursor's location shows up as a blue line, crossing the whole scrollbar area.

On the left edge of the scrollbar, you will see yellow and green marks that indicate the editing status of the file. Yellow means the code has been edited but not saved. Green means it has been edited since the file was last opened, but it has been saved.

On the right edge of the scrollbar are red, green, and gray marks. These indicate errors, warnings, and suggestions about your code. Errors are red, warnings are green, and suggestions are gray. I try to leave files in a clean state, so I use these markers to ensure that I don't leave any issues behind when I walk away from a file.

While the scrollbar gives you a file-wide view of things, the code within the editor also has markers and annotations for errors, warnings, and suggestions, usually in the form of a squiggly underline of the relevant code.

Code Navigation

There are a lot of tools in the editor for getting around in your code.

Ctrl + Click on a code element, and you will jump to its definition. This is great when you say, "Wait, what is this method/class/thing again?" **Ctrl + Click** to jump over and see!

Right-click on a code element and choose **Find All References** to see where something is used. The results will appear in the Search Window (usually at the bottom).

As you begin jumping around in the code, you will find two other shortcut keys helpful: **Ctrl + -** and **Ctrl + Shift + -**. (That is the minus key by the 0 key.) **Ctrl + -** will take you back to the last place you were editing, while **Ctrl + Shift + -** takes you forward.

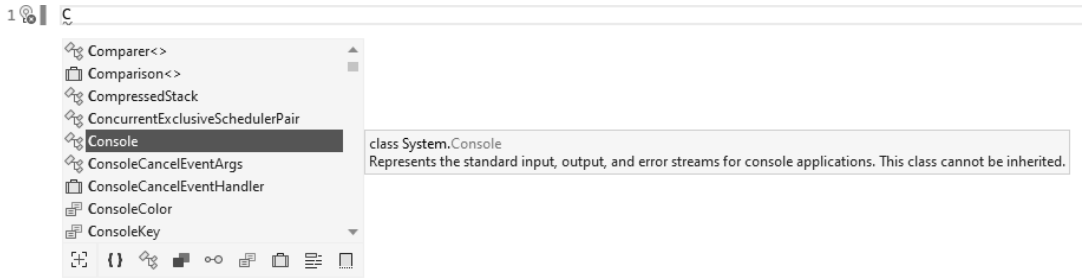
Ctrl + T opens up the **Go to All** search, which lets you find types, methods, etc., by name.

Ctrl + F will let you search for text in the current file, while **Ctrl + Shift + F** will look across in the solution.

IntelliSense

IntelliSense is a feature in the editor that surfaces useful information while typing and performs auto-complete. As you start typing, IntelliSense pops up below what you're typing with suggestions and information. IntelliSense can save you from all sorts of typo-related mistakes, refresh your memory at precisely the right time, and save you from typing long, descriptive names.

To illustrate, let's say I want to do the traditional `Console.WriteLine("Hello, World!");`. I begin by typing the initial "`C`", which triggers IntelliSense.



It tries to highlight frequently used things. In this case, with the hint of "C," **`Console`** was the first suggestion. Exactly what I wanted. You can also see that it brought up the **`Console`** class's documentation so that I don't have to go to the Internet to read about it.

Since the thing I wanted is highlighted, I can press **Enter** to auto-complete the name. Or I can use the arrow keys to pick something else.

After typing a period, IntelliSense pops up again with the next suggestion. Its first choice is **`WriteLine`**, which was what I wanted, so I press **Enter** a second time.

Next, I type the left parenthesis, which brings up IntelliSense a third time, showing me all the different versions (overloads) of **`WriteLine`**, and how to use them. It initially brought up the one with no parameters, which is not what I wanted. But I can type the quotation mark character, which gives IntelliSense the information to know I want a **`WriteLine`** with a **`string`** parameter, and it finds the documentation for it.

It can't suggest what text I will type, so I need to type the **`"Hello, World!"`** myself.

The whole line has 34 characters, but I could have gotten away with typing out 17—half that many. Most of those came from typing out **`"Hello, World!"`** which is a **`string`** literal. Some lines are better, and some are worse. Just because IntelliSense has suggested something doesn't mean you can't just keep typing. But IntelliSense will still save you a lot of keystrokes and help you avoid typos.

If IntelliSense is ever in your way, hitting **Esc** shuts it down. If you ever need it but don't have it, **Ctrl + Space** brings it back up. You can also turn it off entirely.

Quick Actions

Visual Studio can often provide Quick Actions to help fix issues and perform common tasks as you work. These come in the form of either a lightbulb icon or a screwdriver icon. Sometimes, the lightbulb icon has a red marker on it.

Let's illustrate Quick Actions with an example. A typical Hello World program looks like this:

```
Console.WriteLine("Hello, World!");
```

But suppose we had a typo and had this instead:

```
Console.WritLine("Hello, World!");
```

This code won't compile because **`Console`** doesn't have a **`WritLine`** method. That **`WritLine`** word gets underlined in red to mark the issue. If you hover over it, you'll see a popup that shows the compiler error, and you'll also see the Quick Action lightbulb icon with a red marker on it:

If you place the cursor on the line with the issue, you'll also see the lightbulb icon show up in the left margin by the line numbers.

You can either click on the lightbulb icon or press **Ctrl + .** to open the Quick Actions popup, which gives you a list of actions that may address the issue. Each action shows you a preview of what the code would look like when finished.

In this case, the top actions will fix the compiler error ("Change 'WritLine' to 'WriteLine'").

A yellow lightbulb icon indicates a change that Visual Studio recommends you make. If the yellow lightbulb has a red marker on it, the change will fix a compiler error.

A screwdriver icon indicates something Visual Studio can do without implying that it is an improvement. These are refactoring suggestions—ways to change your code without changing behavior. You can consider doing these Quick Actions, but only if it will change your code for the better.

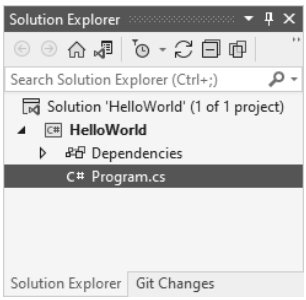
For some refactorings, the change happens instantaneously. For others, a dialog will appear to get more information before performing the change. In other cases, it will change the code most of the way but allow you to continue typing to finish the change. For example, extracting a method creates a new method with the name **`NewMethod`**. It will highlight the name and ask you to type in a better name. As you type, the name will change in all locations where it is used simultaneously. You can hit either **Enter** or **Esc** when you're done editing, and the change will be applied.

Two warnings about Quick Actions:

1. Just because it is there doesn't mean you should automatically do it. This is especially true of the actions with a screwdriver, which are possibilities, not fixes. The code is yours, not Visual Studio's. If you don't like a change, don't do it. If you don't understand a change, don't do it.
2. Be cautious when there are multiple to choose from. It is easy to accidentally pick one you didn't intend because it happens so fast. Keep an eye on how a Quick Action changes your code.

Quick Actions are a fantastic tool to help you get your code working and well organized.

The Solution Explorer



The Solution Explorer shows you a high-level view of your whole solution in a tree structure.

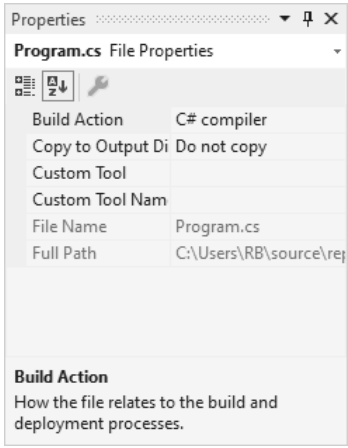
This window lets you jump around in your program quickly and gives you a high-level view of your program's structure. This tree structure has all the features you'd expect of a tree structure, like being able to expand and collapse nodes and drag-and-drop.

Every item has a context menu that you can open by right-clicking on the item. Each item type has different options in its context

menu. You will use the commands in this context menu heavily, so it is worth getting familiar with what is there for each of the different types of elements.

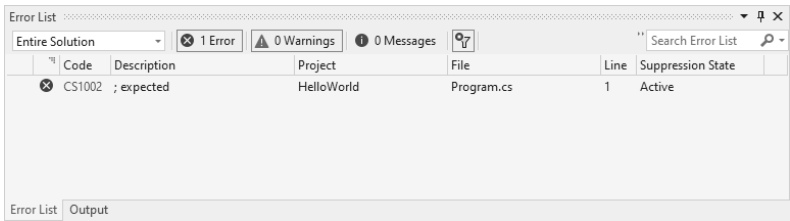
The Properties Window

The Properties window starts in the lower right part of Visual Studio. It displays miscellaneous properties for whatever you have selected, though this is most useful when you've got an item selected in the Solution Explorer.



The Error List

The Error List shows you the problems that occurred when you last compiled your program, making it easy to find and fix them.



You can double-click on any item in the list, and the Code Window will open to the place where the problem occurred in your code. You can show or hide all errors, warnings, or messages by clicking the buttons at the top.

Other Windows

While these are some of the most valuable windows in Visual Studio, there are many others. You can find these throughout the main menu. The **View** menu lists many windows, and each other top-level menu item has a **Windows** item underneath it containing items that will open additional windows. It is worth taking some time to tinker with these so you have a feel for what's available when you need it.

THE OPTIONS DIALOG

Visual Studio is highly configurable. It's difficult to count them, but there are thousands or tens of thousands of things you can configure in Visual Studio. If you don't like the behavior of something, there's a good bet it is configurable somehow.

While we won't talk through all of Visual Studio's options, it is worth pointing out that you get to the Options dialog through the **Tools > Options** menu item. Options are grouped in pages, and each page fits in a tree structure hierarchy, which you can navigate on the left. It is also searchable. For example, if you want to turn on or off the display of line numbers, searching for "line numbers" filters the tree to just the relevant pages.

Knowledge Check **Visual Studio** **25 XP**

Check your knowledge with the following questions:

1. What is the name of the feature that provides auto-complete and suggestions as you type?
2. What is the name of the feature that gives you quick fixes and refactorings?
3. Which window shows you all the files in your program?
4. Which window shows you the list of problems currently in your code?

Answers: (1) IntelliSense. **(2)** Quick Actions. **(3)** Solution Explorer. **(4)** Error List

BONUS LEVEL B

COMPILER ERRORS

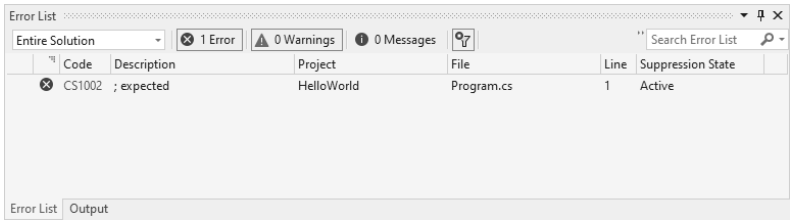
Speedrun

- The Error List shows you the errors and warnings in your code.
- Treat compiler warnings like errors—fix them before moving on.
- Resolve compiler errors by looking for Quick Actions, not waiting too long before attempting to compile (so they don't collect), and being careful with copying and pasting code from the Internet, among other things.

While trying to build working programs, you will create far more broken programs. The compiler's job is converting your C# code into something that the computer can run. If you make something that cannot be converted, the compiler will let you know. It is easy to think of the compiler as the enemy, throwing problems in your path, but instead, think of it as an ally trying to give you vital information to help you build software that works.

CODE PROBLEMS: ERRORS, WARNINGS, AND MESSAGES

The compiler reports actual and potential problems as errors, warnings, and messages. The Error List window, which starts at the bottom left of Visual Studio, displays these items whenever you compile and have problems.



A compiler error means the compiler cannot turn what you wrote into something the computer can understand. The C# compiler is strict, ensuring you don't make mistakes prevalent in other programming languages. The language itself is designed to help you build good software.

A compiler warning indicates that the compiler can turn what you wrote into working code, but it seems suspicious. The compiler is nearly always correct. You should eliminate all compiler warnings before moving on to the next feature. They almost always represent latent bugs, waiting to jump out at the worst possible time.

A compiler message is the lowest severity and represents something that the compiler noticed and thought notable but not alarming. You can decide whether to fix these, but I like seeing a clean Error List, so I usually fix them.

How to Resolve Compiler Errors

Fixing compiler errors is a skill that takes time to hone, but it gets easier with time. As you get more comfortable with C# programming, most errors will amount to you saying, "Oh right, I forgot about that." A couple of quick keystrokes later, you are off and running. Sometimes, it can be tricky to figure out how to solve a compiler issue or even know what the error means, especially as a beginner. Let's look at some tips to help make that go more smoothly.

Compile Often

It is wrong to think that good programmers crank away for hours writing code and have it work the first time they run it. No programmer can do that reliably. Programmers learn that a lot can go wrong quickly. Instead, we take baby steps and compile and run frequently. Baby steps allow us to confirm that things are going well or correct course if needed. It avoids the dreaded compiler error landslide in the first place.

Use a Quick Action

Many compiler errors have a Quick Action (Bonus Level A) to resolve the issue for you. Using a Quick Action is a quick win for issues with a clear-cut cause and a straightforward fix. But don't get lazy; understand what the Quick Action does and verify that it did what you expected. You are the programmer here, not Visual Studio.

Make Sure You Understand the Key Parts of the Error Message

The following code leads to the compiler error "CS0116: A namespace cannot directly contain members such as fields or methods."

```
namespace ConsoleApp1
{
    int x;
}
```

When you encounter a mysterious error, start by ensuring you know what all of the essential words in it mean. Do you know what a namespace is? What a member, field, or method is? Once you understand the parts or the message, the whole typically also makes sense.

Backup or Undo

Sometimes a seemingly simple edit results in a bunch of errors. In these cases, use the Undo button (**Edit > Undo** or **Ctrl + Z**) to go back to find which specific change caused the problem. Undo helps you pinpoint where exactly things went wrong, which is essential information.

Be Cautious with Internet Code

The Internet is full of examples of code that you can and should reuse. It is a veritable treasure trove of information and samples, indispensable in modern programming. But Internet code is often meant to be illustrative; it sometimes has typos or is incomplete. Sometimes, the author calls out missing pieces. Other times, they assume or forget. And sometimes, the code is for older versions of C#, and there are better alternatives now.

Plopping random chunks of code into your program is dangerous because of these reasons. You never know what they have accounted for. They certainly could not consider your specific needs when they wrote it.

I recommend this: do not put any code into your program unless you understand all of it. Ideally, you'd study the material until you know it well enough to write it out yourself, but copy and paste is too convenient a tool to adhere to that strictly. Programming is hard enough when you understand each line. Don't make it harder by embedding Mysterious Runes of the Internet in your code. Seek to understand.

Be Careful Transcribing Code

If you are manually typing in code from another source (like a book), be careful. It is easy to get a single misplaced character that causes a compiler error (or worse, that does not cause a compiler error but still causes a problem!). If you get a compiler error after transcribing something, carefully double-check everything. Be meticulous; it is easy to miss a semicolon.

Fix the Errors that Make Sense

If you have many errors, start with the ones with an obvious solution. Some errors lead to other errors. If you fix the ones you know how to fix, others may also disappear.

Look Around

The C# compiler is good at pointing out problem spots, but it doesn't always get it right. Sometimes the real problem is many lines away. Start where the compiler directs, but don't limit your problem solving to a single line.

Take a Break

Sometimes, you just need to let your subconscious mind churn on a problem for a while. If you encounter a hard-to-solve compiler error and can't solve it after a time, take a break. You may find yourself thinking of more things to try in the shower, while out getting some exercise, or while taking a coffee break. Just don't walk away for too long. Nobody should give up on programming because they encountered a stubborn compiler error.

Read the Documentation

Visual Studio makes it easy to look up information for specific compiler errors. In the Error List, you will see a code for each error. Clicking on the error code will open up a web search for the error, which usually leads you to the official documentation about the error. These pages typically contain good suggestions on what to try next.

Ask for Help

If everything else fails, ask for help. Maybe another programmer around you can help you sort through the issue. Or perhaps a web search will find somebody who had the same problem in the past and wrote down their solution.

stackoverflow.com is a programming Q&A site full of questions and answers for almost any problem you may encounter. It is a lifesaver for programmers. But fair warning: Stack Overflow has some specific rules that are aggressively enforced. If you choose to ask questions, look to see if it has been asked first, and read the site's rules before asking.

There is a Discord server for this book with a community of others who can help you get unstuck (**csharpplayersguide.com/discord**), especially if it relates specifically to something in this book.

COMMON COMPILER ERRORS

Let's go through some examples with some of the more common compiler errors.

"The name 'x' doesn't exist in the current context"

This error happens when you try to use a variable that has never been created or is not in scope in the location you are trying to use.

It could be that you mistyped something. This is easy to fix: change the spelling, and you're done. In some cases, you spelled it correctly on the line with the error but misspelled it when you declared it. You may need to go back to where the variable is declared to check.

Other times this is a scope issue. A common occurrence is declaring a variable in block scope but then attempting to use it outside of the block:

```
for (int index = 0; index < 10; index++) { /* ... */ }

index = 10; // Can't use this here. It doesn't exist after the loop.
```

Address this by declaring the variable outside of the block instead:

```
int index;
for (index = 0; index < 10; index++) { /* ... */ }
// Can use index after the loop now.
```

") expected", "} expected", "] expected", and "; expected"

These errors tell you that your grouping symbols have gone wrong. Usually, it means you forgot to place one of these, but other times it is because you got them out of order.

Fixing this is sometimes easier said than done:

```
for (int x = 0; x < 10; x++)
{
    for (int y = 0; y < 10; y++)
    {
              // Missing curly brace here.

    MoreCode();
}             // Error shows up here.
```

Based on the vertical alignment and whitespace above, the missing curly brace belongs to the **`for`** loop. But since whitespace does not matter in C#, the compiler will dutifully grab the next curly brace to close the loop, and the error shows up later than you may have assumed.

Extra or missing braces and brackets can lead to many other compiler errors because the compiler thinks everything is in a different logical spot than you intended. If this shows up in a list with many other errors, try fixing this first. It may automatically fix the rest.

Cannot convert type 'x' to 'y'

The following three data conversion errors are common:

- Cannot implicitly convert type 'x' to 'y'.
- Cannot convert type 'x' to 'y'.
- Cannot implicitly convert type 'x' to 'y'. An explicit conversion exists (are you missing a cast?)

These typically appear when you mistake a variable or expression's type for another. Make sure that everything is using the types that you expected. You may need to cast.

"not all code paths return a value"

If a method has a non-void return type, every path out of the method must return a value. This error is the compiler telling you that there is a way out of the method that does not return something. For example:

```
int DoSomething(int a)
{
    if (a < 10) return 0;
}
```

If **`a`** is less than **`10`**, then **`0`** is returned. But if **`a`** is **`10`** or more, no return value is defined. Fix this by adding an appropriate return statement where the compiler marked.

"The type or namespace name 'x' could not be found"

The compiler must be able to find a type's definition for you to use it. This error indicates that it failed to find it. This could be a typo, but the cause is often a missing **`using`** directive at the top of the file. Figure out the type's fully qualified name and add a **`using`** directive (Level 33) for that namespace. There is also typically a Quick Action available.

Alternatively, it could be that the type is in a library or package that you have not referenced. If so, add a reference to it (Level 48).

Knowledge Check **Compiler Errors** **25 XP**

Check your knowledge with the following questions:

1. **True/False.** Your program can still run when it has compiler warnings.

2. Name three ways to help work through or reduce tricky or problematic compiler errors.

Answers: (1) True. **(2)** Any subsection headings under *How to Resolve Compiler Errors* are good answers.

Bonus Level C

Debugging Your Code

Speedrun

- Debugging lets you take a detailed look at how your program executes, making it easier to see what is going wrong.
- Use breakpoints to suspend the execution of your program at critical points.
- When suspended, you can step forward a little (or a lot) at a time.
- In some cases, you can even edit code as it is running and resume without restarting.

Getting code to compile is only the first step of making working code. *Debugging* is the act of removing bugs from your program. Being good at debugging is a skill that requires practice but is valuable to learn. You may not be good at it the first time, but you will get better.

When you debug your code, you do not have to go alone. In this level, we will focus on learning how to use a powerful tool that aids you in debugging your code: the *debugger*. A debugger allows you to do interactive debugging. It will enable you to pause execution, inspect your program's variables, and step through your code one line at a time to see how it is changing. The debugger can help you see the problem more clearly, making it easier to fix bugs. Once you learn how to use the debugger, you will use it daily.

Some Broken Sample Code

Debugging large programs can be challenging. Let's start with this simple illustrative example:

```
Console.Write("Enter a number: ");
double number = Convert.ToDouble(Console.ReadLine());
double clampedNumber = Clamp(0, 10, number);
Console.WriteLine($"Clamped: {clampedNumber}");

double Clamp(double value, double min, double max) // Also see Math.Clamp.
{
    if (value < min) return min;
    if (value < max) return max;
```

```
    return value;
}
```

This **`Clamp`** method takes a value and a range. It returns the number closest to the original value while still in the given range. Suppose our range is 0 to 10. A value of 20 should result in 10. A value of -1 should result in 0. A value of 5 should result in 5.

That was the intent, but it is not what is happening. Passing in 20 gives us 10 as expected. But -1 and 5 also return 10. Did we just create a complicated way to return only 10?

Let's debug this and find out what's going on.

PRINT DEBUGGING

Without a debugger, you might think to use **`Console.WriteLine`** to display relevant things as the program is running. This approach is called *print debugging*. Even with a debugger, it has its uses. The code below adds print debugging to **`Clamp`** to see what is happening:

```
double Clamp(double value, double min, double max)
{
    Console.WriteLine($"value={value} min={min} max={max}");
    if (value < min) { Console.WriteLine("returning min"); return min; }
    if (value < max) { Console.WriteLine("returning max"); return max; }
    Console.WriteLine("returning value");
    return value;
}
```

This change may be enough to reveal the problem. Perhaps you can see it in the output below:

```
Enter a number: -1
value=0 min=10 max=-1
returning min
Clamped: 10
```

Print debugging has its uses, but it also has two drawbacks:

1. You must change the code. You might break other things while adding (or later removing) these statements. And if you forget to remove them, your program will have more output than you intended. Adding all of these statements also makes the code harder to read.
2. Your program can quickly display lots of these debug statements, producing so much data that it is hard to see the problem in the clutter.

USING A DEBUGGER

The alternative to print debugging is to use the debugger. This tool is designed to make it easy to walk through your code a little at a time and inspect the state of everything as you go. It is usually the fastest way to figure out what is going on without changing code.

When you run your program from Visual Studio, it automatically attaches a debugger to it. (You can also run without attaching a debugger by either (a) picking the **Debug > Start Without Debugging** menu item, (b) pressing **Ctrl + F5**, or (c) picking the light green arrow next to the regular start button.)

BREAKPOINTS

The first step in using the debugger is getting it to suspend your program as it is running. Pausing will happen in any of the following situations:

1. **Hitting pause while running.** The pause button replaces the green play arrow after launching. The debugger pauses your program when you press this, but the computer runs instructions so fast that this is not a very precise tool.
2. **When an exception is encountered.** When an unhandled exception (Level 35) is encountered, the debugger will suspend your program rather than terminate it, allowing you to inspect your program's state while it is in its death throes.
3. **At a breakpoint.** A breakpoint allows you to mark a line or expression as a stopping point. When your program reaches the marked code, it will suspend the program.

Breakpoints are the most versatile way to inspect your program as it runs, and they are a crucial tool to master. In Visual Studio, you can add breakpoints in several ways:

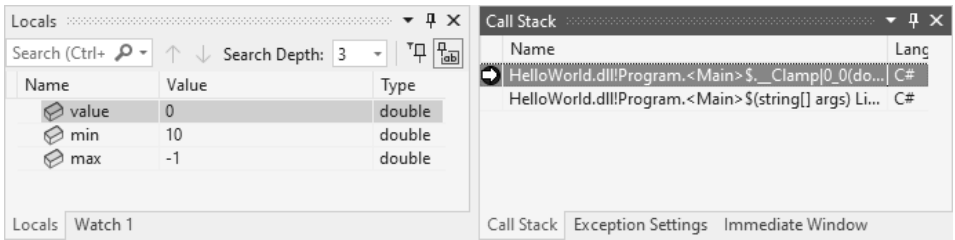
- Right-click on a line of code and choose **Breakpoint > Insert Breakpoint**.
- With the cursor on the interesting code, press **F9**.
- Click on the area left of the line numbers where breakpoints show up.

When a breakpoint is added, you will see a red dot in the bar left of the line numbers. The code with the breakpoint will also be marked with a maroon color.

```
6   double Clamp(double value, double min, double max) // Also see Math.Clamp.
7   {
8       if (value < min) return min;
9       if (value < max) return max;
10      return value;
11  }
```

Removing a breakpoint is as easy as clicking on the red breakpoint dot or pressing **F9**.

Your running program will pause when it reaches a line with a breakpoint on it. Once suspended, Visual Studio shows you many tool windows to inspect your running program.



Your view may vary somewhat. If you are missing one of these windows, you can open them from the menu under **Debug > Windows**.

Most of these windows are self-explanatory, but let's look at some of the more useful ones.

1. **Call Stack.** This window shows the current state of the call stack. The image above shows it on the right half. The current method is at the top of the stack. The method that called it appears under it. The picture shows only two methods on the call stack, but most situations will have far more. You can double-click on any of the call stack frames to focus on that method in the stack, changing the details of many of the other windows.
2. **Locals.** This window shows the current value of all local variables and parameters. It can also usually let you type in new values for any variable.

3. **Watch.** This window allows you to type in expressions that you want to track the value of. It is most useful when you want to keep an eye on the value of an expression that isn't already in a variable. For example, you could enter **min > max** as a watch. You would expect this to be false, but doing this reveals our first problem: it is **`true`** at this breakpoint!
4. **Immediate Window.** This window lets you type in code that is evaluated immediately, showing the result. You can use variables and methods from your code here as well.

On top of that, the code editor gains special abilities while debugging. If you hover over a variable or an expression, it will show you its current value.

With a breakpoint on the first line of **`Clamp`**, we can see that if we type in **`-1`**, **`value`** is **`0`**, **`min`** is **`10`**, and **`max`** is **`-1`**. These variables are scrambled! The debugger does not fix bugs for you; it just provides you with the information needed to find the bug. Knowing we're passing our arguments in the wrong order, the fix is easy. We simply put the arguments in the proper order:

```
Console.Write("Enter a number: ");
double number = Convert.ToDouble(Console.ReadLine());
double clampedNumber = Clamp(number, 0, 10);
Console.WriteLine($"Clamped: {clampedNumber}");
```

With this change, entering **`-1`** gives us the correct value of **`0`**. When we try **`5`**, it does not give us the expected **`5`**, nor does **`20`** produce the expected **`10`**. We have a second bug.

STEPPING THROUGH CODE

Once the debugger has suspended your program, you can step through your code one line at a time to see how it changes. The buttons for this in Visual Studio are on the main toolbar:

From left to right, these buttons are **Step Into**, **Step Over**, and **Step Out**.

Each allows you to move forward a little bit in your program in different ways. **Step Into** and **Step Over** are the same most of the time, advancing a single line in your code. The difference between them comes on a line with a method call. **Step Into** goes into the called method, while **Step Over** just goes to the following line in the current method. **Step Out** advances until the current method completes.

If you right-click on a line, you can also choose **Run to Cursor**, which will execute statements until the line is reached, letting you fast-forward to a specific spot.

There is also this scary option in that same context menu: **Set Next Statement**. This command sets the chosen line as the next one to run. It does not run the intervening code. It is powerful, letting you jump to other places in the method at your whim. But it is dangerous because it can run your code in unnatural ways, which can break things.

When you are ready to resume running, press the green Continue button at the top of Visual Studio. This will resume running like normal until the next breakpoint is reached.

Stepping through code can help us determine why our code is returning the wrong value. We can put a breakpoint at the start of **`Clamp`** and then step through it to see which code path is running. As expected, it does the first **`if`** check and continues to the following line. But on the line **`if (value < max)`**, it goes to the return statement, contrary to what we would expect. When we see this, we can determine that the **`if`** statement's condition is wrong—it should be **`> max`**, not **`< max`**. After this second fix, the program is now working correctly.

Edit and Continue and Hot Reload

Visual Studio has two closely related features that make debugging even better: *Edit and Continue* and *Hot Reload*. These two features both allow you to make changes to your code while it is running and have the changes applied immediately without needing to recompile and restart your program.

If you are stopped at a breakpoint, you can edit the running method. By saving the file, the new, updated version will be recompiled and used going forward. This is the Edit and Continue feature.

Even if you're not stopped at a breakpoint, you can make changes to your code and ask Visual Studio to take your changes and apply them to the running program. This is the Hot Reload feature, and you can activate it with the red flame icon:

Hot Reload doesn't require you to be stopped at a breakpoint, but it does not swap out any currently running method. (You won't see the changes until the method gets called again.)

By default, you must push the Hot Reload button to trigger it, but if you click on the dropdown arrow next to it, you will see an option to enable Hot Reload on File Save, which prevents you from needing to push the button every time.

Between these two features, most situations allow you to edit code while it is running. The only case that isn't covered is editing the currently running method without pausing it first. Perhaps this limitation will be removed someday.

These two features are not able to apply every imaginable edit. Certain aggressive edits, referred to as "rude" edits, won't work. The list of rude edits shrinks with each update, and it is hard to describe every scenario anyway. Rather than memorizing some obscure rules, just attempt the desired edit. If it can be applied, great! If not, just recompile and restart.

BREAKPOINT CONDITIONS AND ACTIONS

Breakpoints can be more nuanced than just "always stop when you reach here." Breakpoint conditions let you use an expression that must be true to engage the breakpoint. Conditions are helpful for frequently hit breakpoints, but you only want to stop under specific scenarios. Breakpoint actions let you display text instead of (or in addition to) suspending the program. The text can include expressions to evaluate inside curly braces, like interpolated strings.

Both breakpoint conditions and actions can be configured by right-clicking on the red circle in the left gutter and choosing the corresponding item.

Knowledge Check **Debugging** **25 XP**

Check your knowledge with the following questions:

1. **True/False**. You can attach a debugger to a program built in the Release configuration.
2. **True/False**. The debugger will suspend your program if an unhandled exception occurs.
3. **True/False**. In some cases, you can edit your source code while execution is paused and resume with the changes.

Answers: (1) True. **(2)** True. **(3)** True.