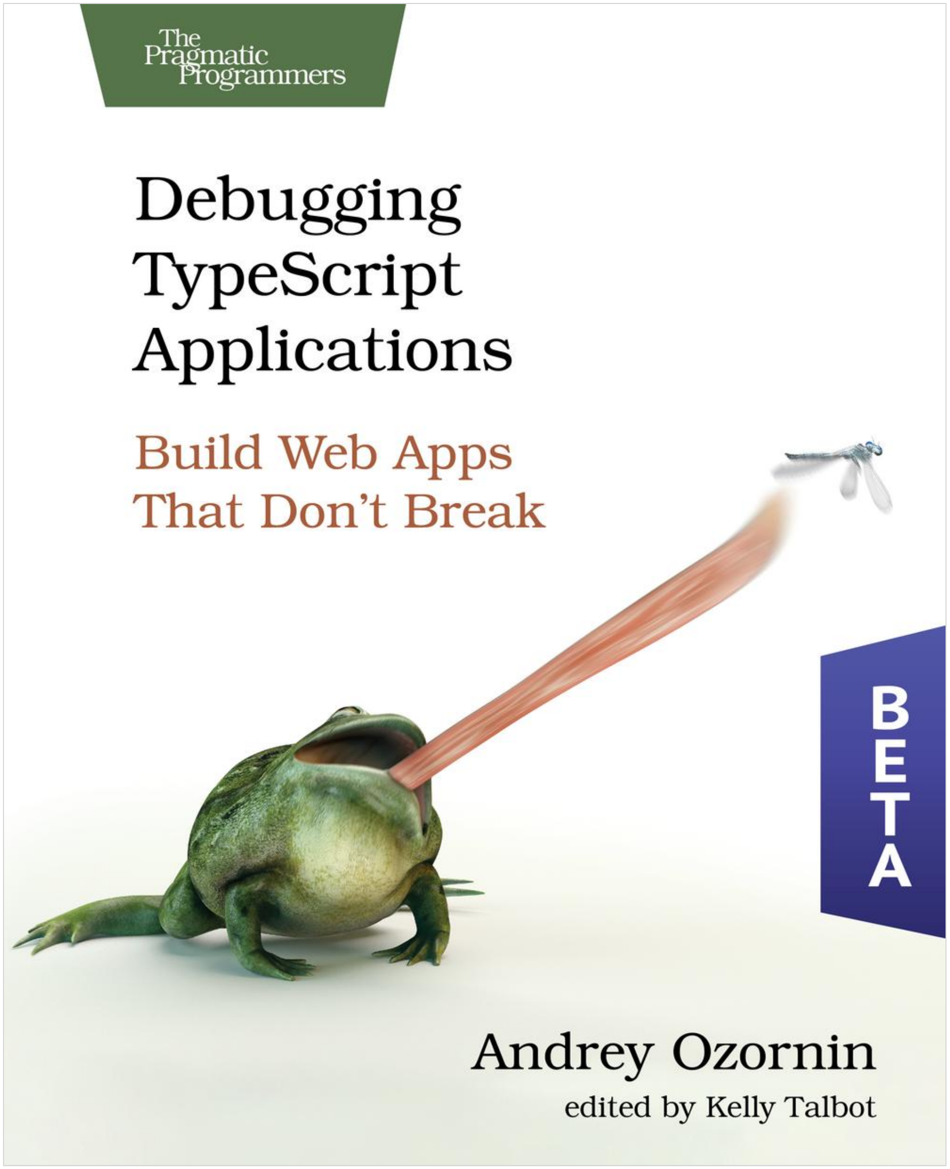
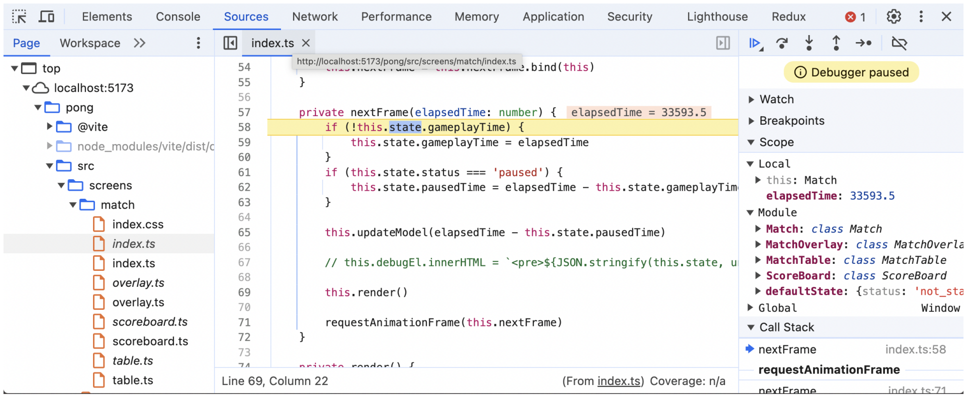

The Pragmatic Programmers
Debugging TypeScript Applications
Build Web Apps That Don't Break
BETA
Andrey Ozornin
edited by Kelly Talbot

Getting to Know the Sources Panel

In our case, after pausing execution, chances are that you ended up in the method called nextFrame. It is not the only possible place, just the most likely one. Try to press that button again to resume script execution, and then pause another time and yet one more time to find out what other processes are running, and follow along when the execution stops in nextFrame.



```
    this.nextFrame = this.nextFrame.bind(this)
  }

  private nextFrame(elapsedTime: number) {  elapsedTime = 33593.5
    if (!this.state.gameplayTime) {
      this.state.gameplayTime = elapsedTime
    }
    if (this.state.status === 'paused') {
      this.state.pausedTime = elapsedTime - this.state.gameplayTim
    }

    this.updateModel(elapsedTime - this.state.pausedTime)

    // this.debugEl.innerHTML = `<pre>${JSON.stringify(this.state, u

    this.render()

    requestAnimationFrame(this.nextFrame)
  }

  private render() {
```

Line 69, Column 22 (From index.ts) Coverage: n/a

```
Local
  this: Match
  elapsedTime: 33593.5
Module
  Match: class Match
  MatchOverlay: class MatchOverla
  MatchTable: class MatchTable
  ScoreBoard: class ScoreBoard
  defaultState: {status: 'not_sta
Global                      Window
```

```
nextFrame               index.ts:58
requestAnimationFrame
nextFrame               index.ts:71
```

Like anything else ever written, code makes the most sense within a larger context. Every module is a part of a bigger system, and understanding a module is difficult without understanding the system at least on some level.

The File Tree

In the left panel, you find the file tree. The folder hierarchy overview shows where the module fits in the structure. From the path /pong/src/screens/match/index.ts, you can conclude that you're seeing the file of a game screen called match of the pong game. So, you're in the right place.

Note that every .ts file has a duplicate. It's a source map of the very same code transpiled to ES6, thus a little closer to what is actually being executed by the JavaScript engine. It comes handy sometimes, when, for instance, short TypeScript constructions like enum tend to generate longer JavaScript code, and TypeScript source maps obscure the lower-level steps that are actually happening.

The Scope

In the right panel, under the Scope, you can find all the values available in the current moment of the script execution process. They are categorized in three buckets:

- Local scope: variables within the current function or code block
- Module scope: values declared on the top level or imported into the current module
- Global scope: properties of the global `Window` object

Usually, the local and module-wide scopes are the most relevant ones you use to get familiar with the surroundings.

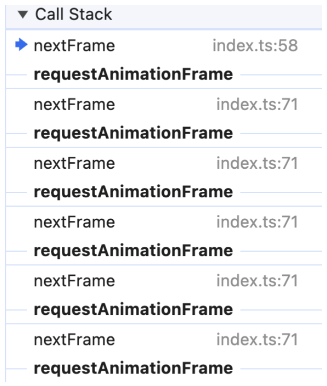
In our case, the local scope has only one variable, `elapsedTime`, so you can conclude the method does calculations depending on the moment in time, which is typical for animations. Also, note that the type of execution context, `this`, is annotated as `Match`. Feel free to expand it and explore its properties.

The module context offers more food for thought. It has, among others, `defaultModelState` and `MatchOverlay`. This might initially seem insignificant, but here are only some of the conclusions you might draw observing just that:

- The `defaultModelState.gameplayTime` has a non-zero value that doesn't look like a default one, and this value updates every time you pause and unpause execution. It means that the state object is mutable. It is not bad per se and is widespread in game development, but it can cause trouble. In some frameworks, such as React, the mutable state is considered an antipattern.
- The `defaultModelState.status` equals `'not_started'` and the ball is moving, so the gameplay must be controlled by something else. Possibly, mistakenly.
- The name `MatchOverlay` next to `MatchTable` looks like a class representing an element covering the match table, so you might want to explore that as well.

The Call Stack

The Call Stack can be found in the same right panel as the scope. It usually provides useful system knowledge, as we'll see later in this chapter. If you expand it and scroll down a little bit, you will see that the function `nextFrame` is repeatedly calling itself using `requestAnimationFrame`. It is a common pattern of `requestAnimationFrame` usage, so it is nothing special. However, taking into account that the method name is `nextFrame` and its argument is `elapsedTime`, you can be confident that you're observing a method asynchronously computing and drawing an animation frame based on a moment in time.

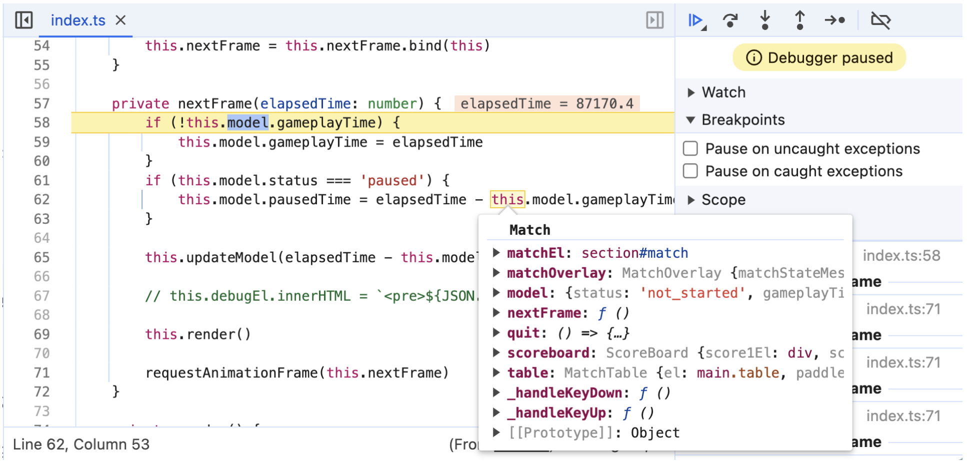


```
▼ Call Stack
➡ nextFrame                index.ts:58
  requestAnimationFrame
  nextFrame                index.ts:71
  requestAnimationFrame
  nextFrame                index.ts:71
  requestAnimationFrame
  nextFrame                index.ts:71
  requestAnimationFrame
  nextFrame                index.ts:71
  requestAnimationFrame
  nextFrame                index.ts:71
  requestAnimationFrame
```

You might be wondering why I spend so much time talking about context and only now get to the code itself, the core of a programmer's work. You may argue that most of the time, you don't need to learn about other modules to find and fix a particular issue, and I agree on that point. However, there are good reasons to still do it. The first is learning. Through debugging and close inspection, you can and will see places you wouldn't find otherwise. I've been in countless situations when a look at context before diving into code not only benefited me long-term or had educational value but also helped me find the issue faster because I noted something odd in the surrounding values. And to notice that something is odd, you need to know what normal looks like. So, observe closely; observe repeatedly. And with that said, let's jump into the code.

Navigating the Code

When the code execution is on pause, the debugger evaluates all expressions in the current function scope down to the current pause point. You can see their values next to each line where they are referenced (see line 57 in the screenshot). Hover over a class or variable name, and you will see its value in a tooltip. It can be local, module, or a globally defined symbol, but note that you will see only values that are declared in a scope visible from the current point in execution.



```
54         this.nextFrame = this.nextFrame.bind(this)
55     }
56
57     private nextFrame(elapsedTime: number) {  elapsedTime = 87170.4
58         if (!this.model.gameplayTime) {
59             this.model.gameplayTime = elapsedTime
60         }
61         if (this.model.status === 'paused') {
62             this.model.pausedTime = elapsedTime - this.model.gameplayTim
63         }
64
65         this.updateModel(elapsedTime - this.mode
66
67         // this.debugEl.innerHTML = `<pre>${JSON.
68
69         this.render()
70
71         requestAnimationFrame(this.nextFrame)
72     }
73
```

```
Match
▸ matchEl: section#match
▸ matchOverlay: MatchOverlay {matchStateMes
▸ model: {status: 'not_started', gameplayTi
▸ nextFrame: ƒ ()
▸ quit: () => {…}
▸ scoreboard: ScoreBoard {score1El: div, sc
▸ table: MatchTable {el: main.table, paddle
▸ _handleKeyDown: ƒ ()
▸ _handleKeyUp: ƒ ()
▸ [[Prototype]]: Object
```

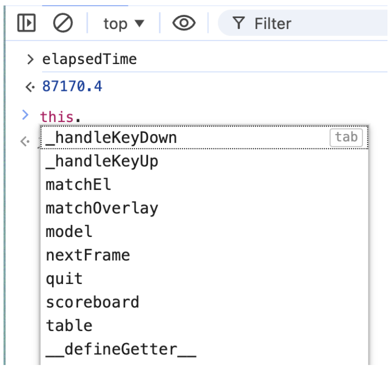
Debugger can only see values that are currently in memory, which means that you won't see things the garbage collector has already removed from RAM, even if they should be available: for instance, via closure.

While the execution is still paused on the first line of the method (line 58), right-click on line 65 and "Continue to here" in the context menu. Note that line 65 is now highlighted and the expressions are evaluated.

Inspect the Code Using the Console

Press Escape on the keyboard. The already familiar console opens below. You can use it to inspect the values during the execution.

Type elapsedTime and press Enter. Type this. (including the period after). Browse through autocompletion to see what properties are available in the current execution context.



```
> elapsedTime
< 87170.4
> this.
<  _handleKeyDown
   _handleKeyUp
   matchEl
   matchOverlay
   model
   nextFrame
   quit
   scoreboard
   table
   __defineGetter__
```

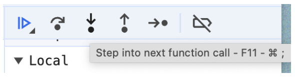
Choose matchOverlay from the dropdown and press Tab to auto-complete. Type . (a period) to see its field list. Note that you see even the fields that are declared as private in TypeScript, for example, matchOverlay.matchStateMessageEl and matchOverlay.quit. Choose quit from the dropdown list of properties and press Enter. The method's listing will appear.

Being able to inspect private properties is a powerful tool in debugging. Also, it means that those properties are not private in the full meaning of the word, and code consumers can abuse it and use private methods in runtime by overriding TypeScript warnings, which is another common source of bugs that are hard to find. You should never do it, and either remove such usages or make the method signatures public when you notice other developers doing it.

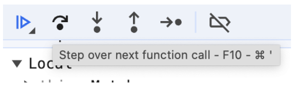
Note that you cannot see properties declared using # prefix in debugger this way. Those are truly private and can be accessed only from the class where they are defined. Also, it's super common that with source maps, variables that you see in code listing are not available in the console. For instance, import { member } from "module" sometimes gets transformed into _module.member, and naively typing member in the console will result in a ReferenceError. A workaround for this is to toggle source maps off by pressing Cmd+Shift+P in Chrome developer tools and check the actual JavaScript code being executed.

Stepping in a stepwise execution

Next, to "Resume execution" in the right panel, find "Step into next function call" (↓). Press it while on the line with the this.updateModel call.



You'll find yourself on the first line of the same class's updateModel method. Press the second button in the row, "Step over next function call" (↷).



You'll move the execution to the next line and see the timeDelta value calculated.

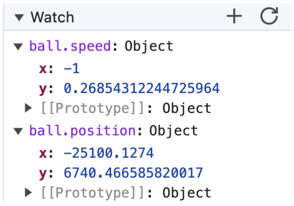
Run JS Snippets During Execution

Now, for the beautiful part: you can change values on the run! Type timeDelta = 100 in the debugging console and step forward one more time, then step into this.table.updateModel(timeDelta) by pressing ↓. You'll find yourself in a different file, table.ts, in a function that updates model of the tennis table. Notice that its input argument, timeDelta, is equal to 100, as you specified it.

You can run any JavaScript the same way right in the middle of execution, even DOM modifications, and even perform network requests. Feel free to run something like document.body.style.backgroundColor = 'red' and immediately see the background color updated.

Watch Expressions

Find and expand "Watch" in the right sidebar. Press the "Add watch expression" (+) button and add some expressions you want to observe, ball.position and ball.speed. It's useful to know that you can use not only values but any executable expressions, which will be re-evaluated at every step. For instance, if you wanted to know how far the ball is from the center, you could add the watch expression Math.sqrt(ball.position.x ** 2 + ball.position.y ** 2).



```
▼ ball.speed: Object
    x: -1
    y: 0.26854312244725964
  ▶ [[Prototype]]: Object
▼ ball.position: Object
    x: -25100.1274
    y: 6740.466585820017
  ▶ [[Prototype]]: Object
```

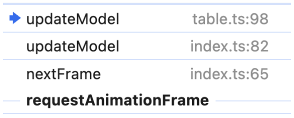
Step through the function using "Step over" and explore how these values change and what code branches are executed. Note what changes are expected to happen to the table tennis contents in 100 milliseconds (because we deliberately set timeDelta to 100).

The MatchTable.updateModel method calculates and updates values in the local tennis table model, which contains the coordinates and speed of the ball and paddles.

We've found out that Match.updateModel updates a local model and triggers an update of its children by calling MatchTable.updateModel. We've learned what each model has and what real values appear there. Let's move on.

Navigating the Call Stack

Look at the call stack. Note that you're two levels deeper than nextFrame where you landed initially (remember, you pressed ↓ twice).



```
➡ updateModel        table.ts:98
  updateModel        index.ts:82
  nextFrame          index.ts:65
  requestAnimationFrame
```

Click on other items in the call stack. It won't affect execution, but you can inspect the code at every level of the call tree. You won't be able to watch expressions or execute commands on levels other than the top one unless you step out of it.

In the row of the execution control buttons, find and press “Step out of the current function” (↑) twice to go back to nextFrame.

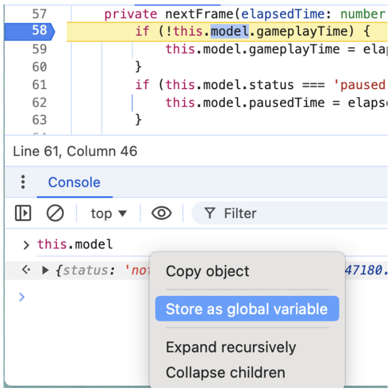
You’ll end up on the next executable line inside nextFrame, which is the call to this.render(). Feel free to step into it and see what it does. It calls the render method on the screen children, which updates the visual elements according to the models we updated earlier.

```
private render() {
        this.table.render()
        this.scoreboard.render()
}
```

Now you can execute the application step-by-step, going deeper if you need. We’ve learned observation and navigation; let’s figure out what precedes the start of animation and how we can change it.

Storing Values as Global Variables

A true hidden gem of Chrome Developer Tools is storing values as global variables. While staying on the breakpoint in nextFrame, type in console this.model and press Enter. Right-click on the logged object and choose “Store as global variable.”

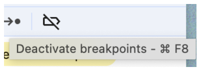


```
private nextFrame(elapsedTime: number
    if (!this.model.gameplayTime) {
        this.model.gameplayTime = ela
    }
    if (this.model.status === 'paused
        this.model.pausedTime = elaps
    }
```

Line 61, Column 46

```
> this.model
< ▸ {status: 'no                     47180.
```

It will be stored as temp1. Now, in the right sidebar, click “Deactivate breakpoints” and resume execution.



The game goes on. The match model that is normally stored in a private property of a class becomes available for your manipulation.

Type `temp1.status = 'paused'`. Now you can programmatically pause and unpause the game while it's running in background, as well as unstart it by setting it back to `'not_started'`, which wouldn't be possible from the UI under any circumstances.