Advanced JavaScript Debugging Techniques for Agile Teams

Zach Gardner
HTML5/JavaScript Consultant
Keyhole Software
zgardner@keyholesoftware.com
@zachagardner

Titanium Sponsors

Platinum Sponsors

Gold Sponsors

Motivation

Successful Agile projects recognize the following:

1. Time = fuel
2. Developers write code and fix code

Validate Your Assumptions

The most fundamental principle behind debugging.

If you can list out your assumptions, you can be effectively debug.

Format

A pattern will be presented with the following:

1. Case study
2. Formal name
3. Elevator pitch
4. Detailed description
5. Use cases
6. Pros and cons

Breakpoint on Specific Condition

```
function somethingImportant(fn, scope) {
  My.library.fireEvent('EVENT_NAME', fn, scope);
}

My.library.fireEvent = function (e, fn, scope) {
  var listeners = this.listeners[e];
  if (!listeners) {
    return;
  }
  this.fireListeners(listeners, fn, scope);
```

Conditional Breakpoints

Only trigger a breakpoint when an internal or external expression evaluates to true.

Most languages and IDEs have the concept of a conditional breakpoint.

Breakpoints can be set in the execution environment, or in the code through use of the `debugger` statement.

Conditional Breakpoints

```
function myFunction (x) {

   var something = new Class();

if (x == 10) { debugger; }

   something.doSomething(x);

}
```

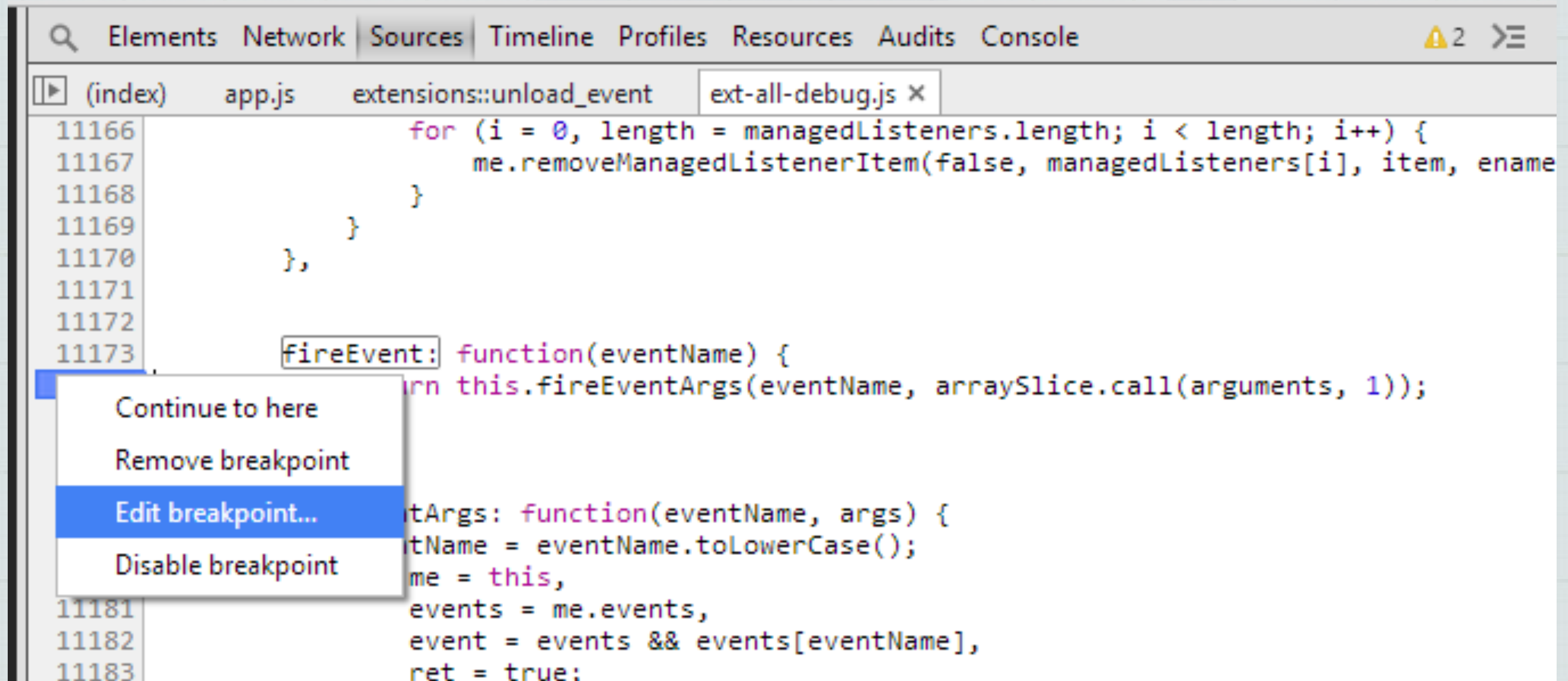
Conditional Breakpoints

Allowing the code to define breakpoints is preferable because:

1. Breakpoints sync up with code changes.
2. Can manage breakpoints between browsers.
3. Ensure the flow is correct while determining the fix.
4. Helps remind developers to fix all bugs before committing.

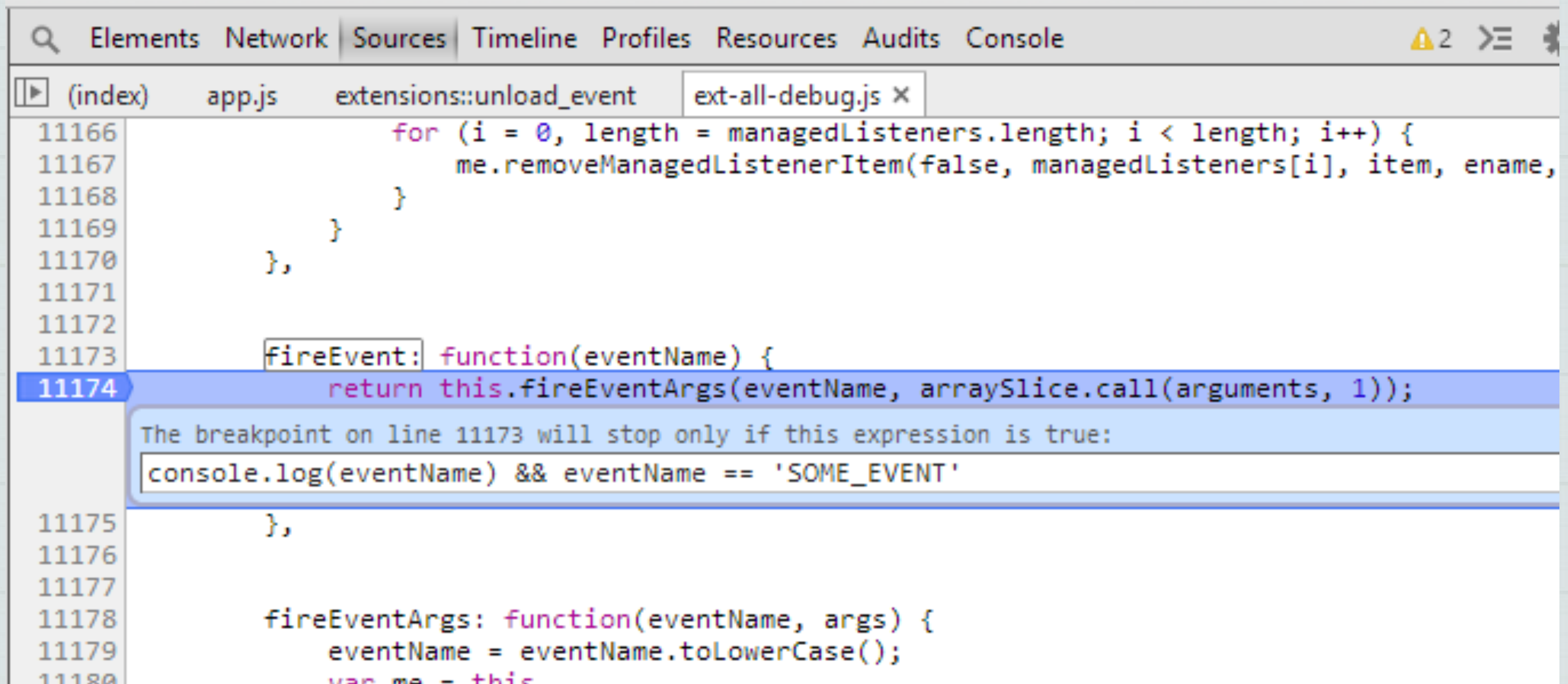
Conditional Breakpoints

Chrome and IE allow conditional breakpoints in Dev Tools:



Conditional Breakpoints

When the expression evaluates to true, a breakpoint occurs:



Conditional Breakpoints

Some bugs only occur on the second or third time code is executed.

I've used conditional breakpoints to get in those specific cases:

```
function shouldWork() {
window.ZG = (window.ZG ? window.ZG + 1 : 1);
if (window.ZG == 2) { debugger; }

   doSomethingNormal();
}
```

Conditional Breakpoints

Best used when a well defined condition triggers a bug.

Some developers may prevent the triggering condition rather than the root cause.

Not all bugs have well defined triggers.

JavaScript Needs Private/Protected

```
function myClass() {
    // Private
    this.private = 123;
    this.setPrivate = function (x) { this.private = x; }
    return this;
}

var goodClass = new myClass();
goodClass.setPrivate('This is good');

var badClass = new myClass();
badClass.private = 'This is bad';
```

Observing Object Changes

Chrome can observe changes in the properties of an object.

Public/private/protected may be well documented, but other sections of code may ignore the documentation.

This can cause errors to occur if getters and setters need to be called to ensure consistency of a class.

By hooking into the places where the object changes, Chrome allows the developer to see where the properties of an object are being modified.

Observing Object Changes

```
function myClass() {
var varToCheck = this.varToCheck;
this.__defineSetter__('varToCheck', function (v) {
debugger; varToCheck = v; return varToCheck; });
this.__defineGetter__('varToCheck', function () {
return varToCheck });
    // Do normal stuff
}

var badClass = new myClass();
badClass.varToCheck = 5; // Executes the debugger
```

Observing Object Changes

Can quickly track down the problem in a large code base if getting or setting outside of the getter or setter is the cause.

The private variable may be several objects deep, or the containing object may be redefined.

Tracking every getter can lead to noise.

Only works in Chrome.

Ever Debugged a Function Like This

```
drawAxis: function(init) {
    var chart = this.chart,
        surface = chart.surface,
        bbox = chart.chartBBox,
        store = chart.getChartStore(),
        l = store.getCount(),
        centerX = bbox.x + (bbox.width / 2),
        centerY = bbox.y + (bbox.height / 2),
        rho = Math.min(bbox.width, bbox.height) /2,
        sprites = [], sprite,
        steps = this.steps,
        i, j, pi2 = Math.PI * 2,
        cos = Math.cos, sin = Math.sin;

    if (this.sprites && !chart.resizing) {
        this.drawLabel();
        return;
    }

    if (!this.sprites) {
        //draw circles
        for (i = 1; i <= steps; i++) {
            sprite = surface.add({
                type: 'circle',
                x: centerX,
                y: centerY,
                radius: Math.max(rho * i / steps, 0),
                stroke: '#ccc'
            });
            sprite.setAttributes({
                hidden: false
            }, true);
            sprites.push(sprite);
        }
        for (i = 0; i < l; i++) {
            sprite = surface.add({
                type: 'path',
                path: ['M', centerX, centerY, 'L', centerX + rho * cos(i / l
* pi2), centerY + rho * sin(i / l * pi2), 'Z'],
                stroke: '#ccc'
            });
            sprite.setAttributes({
                hidden: false
            }, true);
            sprites.push(sprite);
        }
    } else {
        sprites = this.sprites;
        //draw circles
        for (i = 0; i < steps; i++) {
            sprites[i].setAttributes({
                x: centerX,
                y: centerY,
                radius: Math.max(rho * (i + 1) / steps, 0),
                stroke: '#ccc'
            }, true);
        }
        //draw lines
        for (j = 0; j < l; j++) {
            sprites[i + j].setAttributes({
                path: ['M', centerX, centerY, 'L', centerX + rho * cos(j / l *
pi2), centerY + rho * sin(j / l * pi2), 'Z'],
                stroke: '#ccc'
            }, true);
        }
    }
    this.sprites = sprites;

    this.drawLabel();
```

Binary Search

Divide a function in two halves, and see which half produces the bug.

By segmenting a problem into two halves, the scope can quickly be narrowed to focus in on the cause of the bug.

Once the offending segment is found, divide it into two halves.

The halves can be unequal, but it works best if they're close.

Binary Search

Works well when functions are very modular.

If written well, can isolate offending code quickly.

Best used when you have no idea where the cause of the bug is.

Can lead to false positives.

Doesn’t guarantee results.

Can be time consuming if it is the only technique used.

Fn 1 calls Fn 2 calls Fn 3 calls…

```
function primaryFunction () {
   // stuff
   secondaryFunction();
   // more stuff
}

function secondaryFunction () {
   // stuff
   tertiaryFunction();
}
```

Bottom-up

Observe the effect of calling a function until the bug is observed.

In Bottom-up debugging, start with putting a `debugger` statement in code that happens before the bug occurs.

Put a watch statement to check for the bug.

Step over a function, and see if the watch statement indicates the function just called triggered the bug.

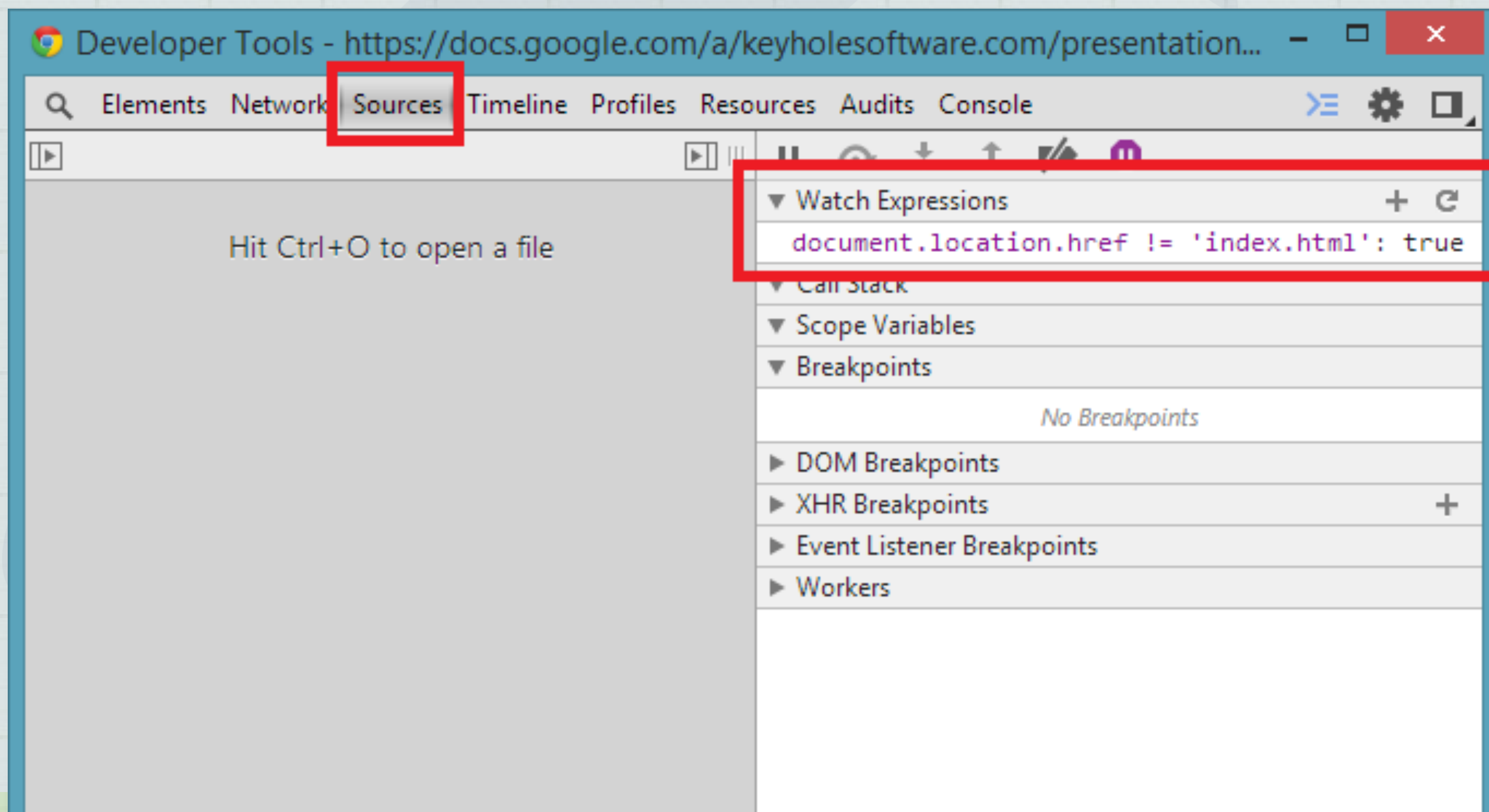
If it didn’t, go to the next.

If it did, restart the process, then step into the function.

Bottom-up

When a button was clicked, some code was causing redirecting to the login page.

I listened for click, and added a watch expression:



Bottom-up

Can quickly isolate bug if working code is known, and if the bug can be easily identified.

Doesn't necessarily lead to finding true source of bug.

May lead to laziness when debugging.

Takes a long time to identify, restart, step into, identify, etc.

Function Incorrectly Called

```
function throwAnExecption() {
   // Throw an exception
}

function callsThrowAnException () {
   throwAnException();
}

function somethingComplicated() {
   callsThrowAnException();
```

Top-down

Find the cause of a bug when the symptom is well defined.

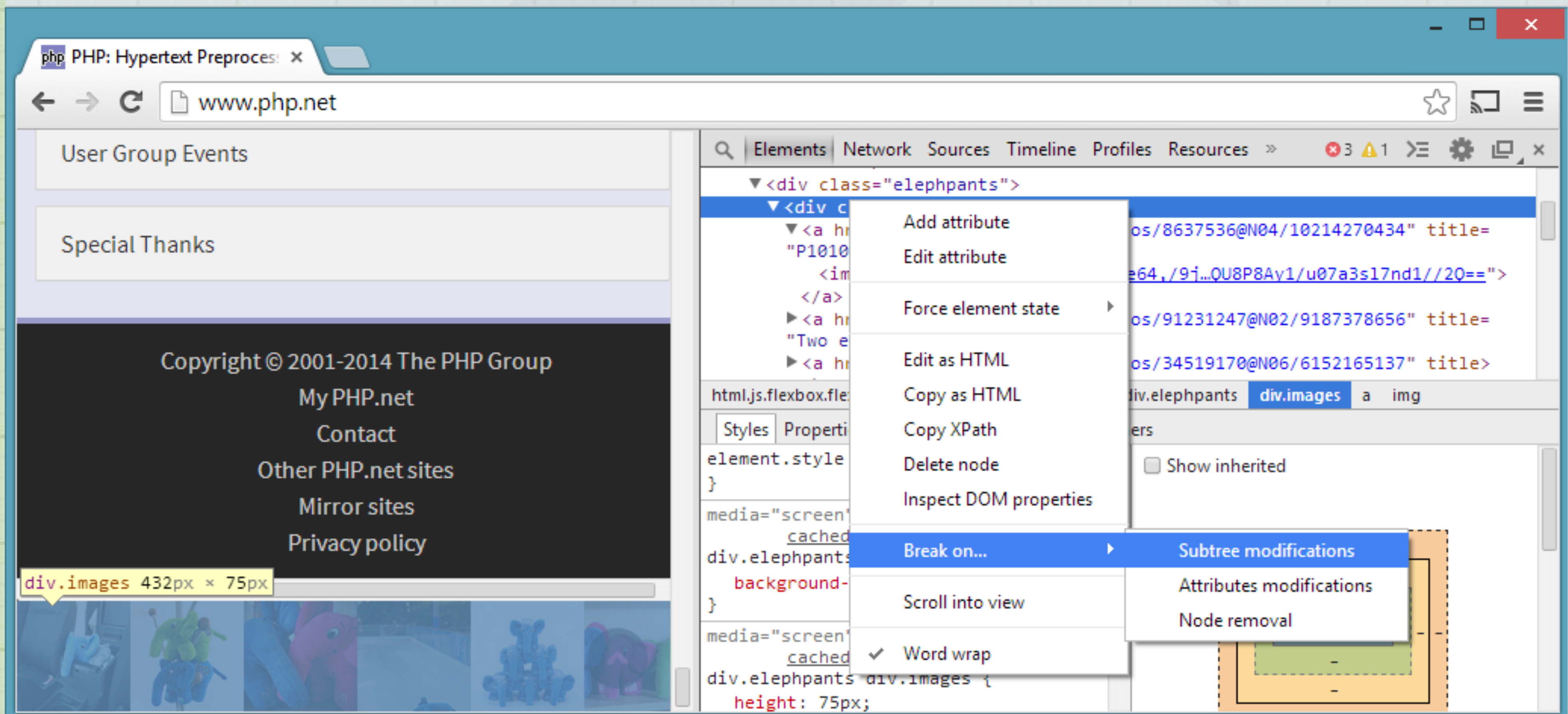
If a bug can be easily observed, the root cause needs to be determined.

It may be calling a function outside of the normal flow, or determining the source of incorrectly shown HTML.

Top-down

Some widgets were rendering with a width and height of 0.

It was difficult to determine where width/height is calculated, so I observed when the DOM attributes were changed:



Top-down

Observing DOM changes is a quick way to see if the bug occurs.

Can also be done for exceptions or a `throw` in a `try/catch`.

There may be so many valid DOM changes that the breakpoint produces excessive noise.

Some bugs are very difficult to directly observe.

Not Obvious Issue

```
function onClick() {
   alert('I was clicked!');
}

function onRender(button) {
   button.on('click', onClick);
}
```

Event Listener Breakpoints

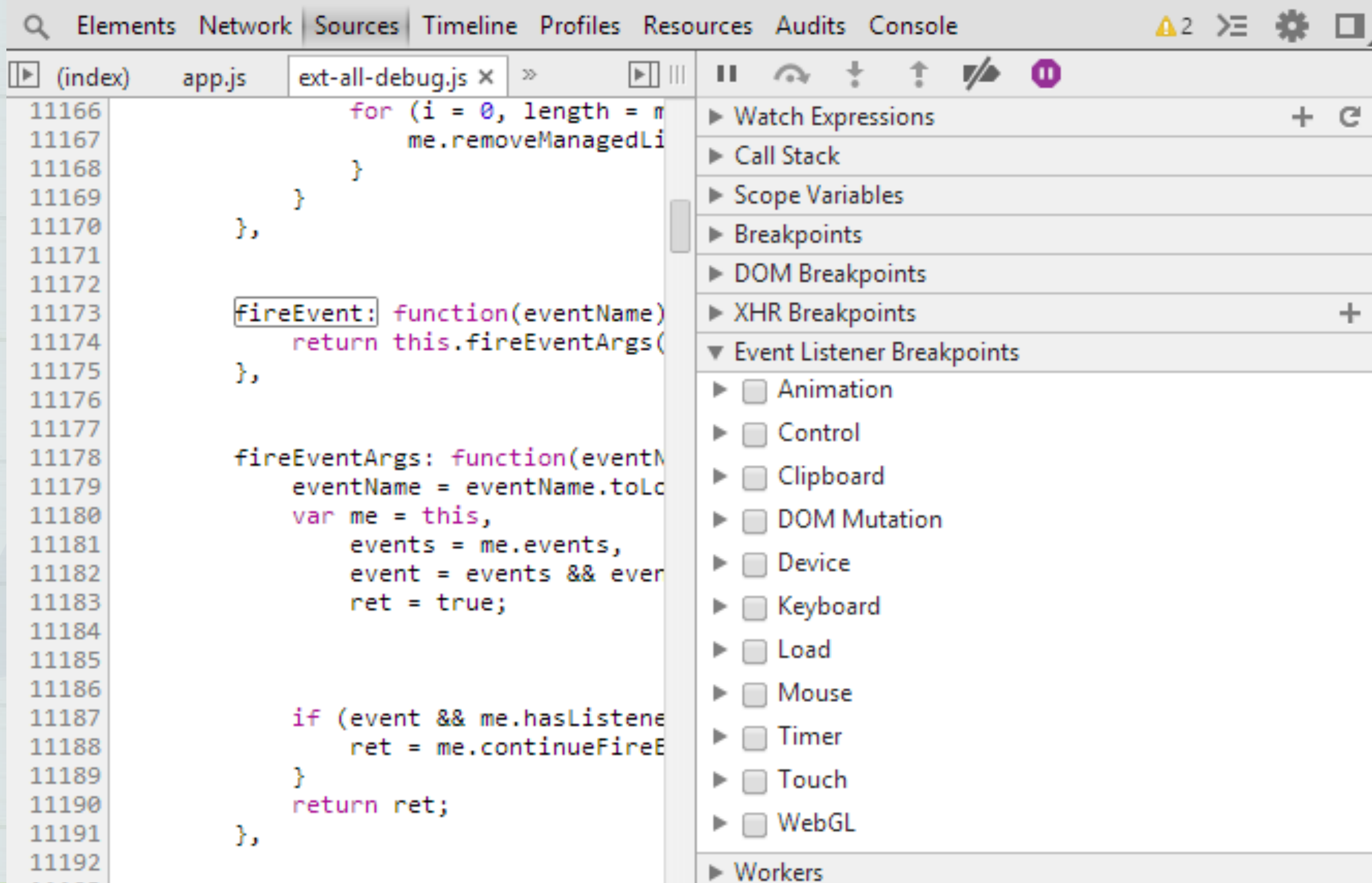
Observe how a existing code responds to a DOM event.

Chrome's Dev Tools allow a developer to select which events they care about.

When the event is executed, the developer can step through the calls to find why it is working differently than they think it should.

Event Listener Breakpoints

By stepping through the code on click, I can find where my assumptions fail:



Event Listener Breakpoints

When a well defined condition is known, it's easy to setup an event listener breakpoint to find out exactly what's going on.

This method makes it easy to narrow the focus on a large code base.

This can lead to Heisenbugs, and make it impossible to debug.

It can take a long time to find where an assumption is invalidated.

QA Best Practices

1. Always turn on debuggers before testing
2. When filing a bug, include as much information as possible:
   a. What steps did you take to see the error?
   b. What browser were you in when the error happened?
   c. Can you provide a screenshot of the error?
   d. Did the error show up in the developer tools?
3. When you see a bug, repeat the process to make sure the exact steps can be determined.

Conclusion

JavaScript is the wild west of development.

What can go wrong will go wrong.

Developers need to understand effective debugging patterns:

1. Conditional Breakpoints
2. Observing Object Changes
3. Binary Search
4. Bottom-up
5. Top-down
6. Event Listener Breakpoints