Debugging + Errors

Download Demo Code <../js-debugging-demo.zip>

Goals

- Define three of the most common errors in JavaScript and how to debug them
- Examine the Chrome Dev Tools for debugging
- Catch errors using try/catch/finally blocks
- Throw errors with specific messages

Debugging

- You are going to make mistakes!
- Let’s examine ways to better debug
- Let’s first examine some common errors

SyntaxError

- You’ve seen this one before!
- You have to fix these right away!

```
"awesome

function first( {}

let = "nice!"
```

ReferenceError

- Thrown when you try to access a variable that is **not defined**
- This does not mean **undefined**

```
function sayHi(){
  let greeting = "hi!";
}

sayHi();

greeting;   // ReferenceError
```

TypeError

- Trying to do something with a type that you can not
- Accessing properties on ***undefined*** or ***null***
- Invoking *("calling")* something that is not a function

```
"awesome".splice() // TypeError

let obj = {}

obj.firstName.moreInfo // TypeError
```

Two Kinds of Bugs

- An error is thrown — **easier**
- You didn't get what you wanted — **harder!**

A process for debugging

- Make assumptions
- Test assumptions
- Prove assumptions
- Repeat

console.log

- Be mindful about what you print out
- Great for a sanity check
- Even better when you add parameters

```
console.log("We made it!");

console.log("The value of x is --->", x);
```

JavaScript Debugger

- Watch execution of code and examine at any point
- Built into Chrome (other browsers have similiar abilities)
- Can debug in-browser code *or* Node

Starting Debugger

View code or adding "breakpoints":

View → ***Developer*** → ***Developer Tools*** → ***Sources*** tab

Click left of line of code to add a blue breakpoint

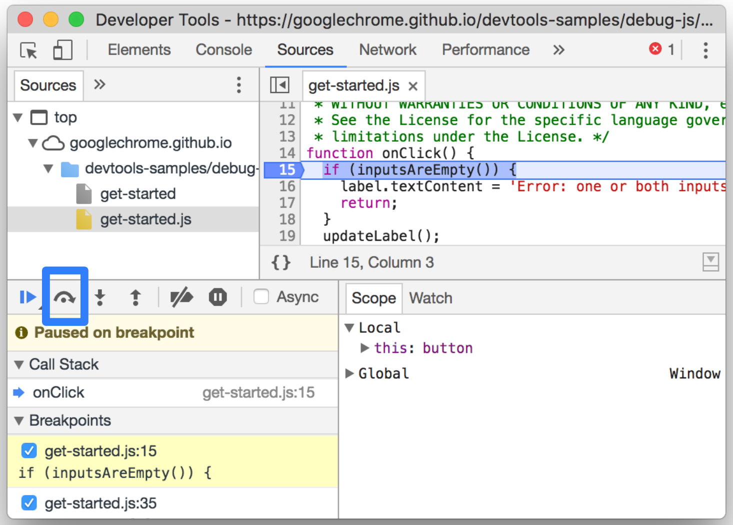
Can put breakpoint into code itself:

```
function myFunction() {
  let x = 1;

  debugger;   // <-- will always stop here

  // rest of code follows ...
}
```

Step Over



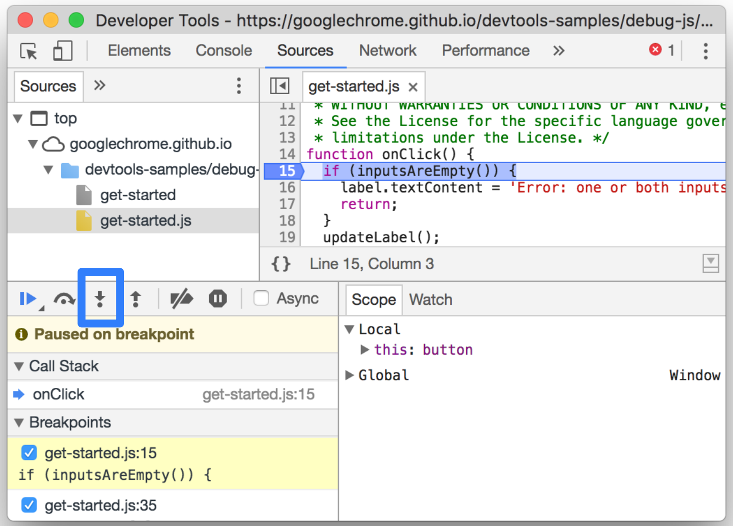
```
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, e
 * See the License for the specific language gover
 * limitations under the License. */
function onClick() {
  if (inputsAreEmpty()) {
    label.textContent = 'Error: one or both inputs
    return;
  }
  updateLabel();
```

```
if (inputsAreEmpty()) {
this: button
```

<_images/step-over-outline.svg>

Run current line, but don’t debug into any function calls

Step Into



```
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, e
 * See the License for the specific language gover
 * limitations under the License. */
function onClick() {
  if (inputsAreEmpty()) {
    label.textContent = 'Error: one or both inputs
    return;
  }
  updateLabel();
```

Line 15, Column 3

Paused on breakpoint

Call Stack

onClick get-started.js:15

Breakpoints

get-started.js:15
`if (inputsAreEmpty()) {`

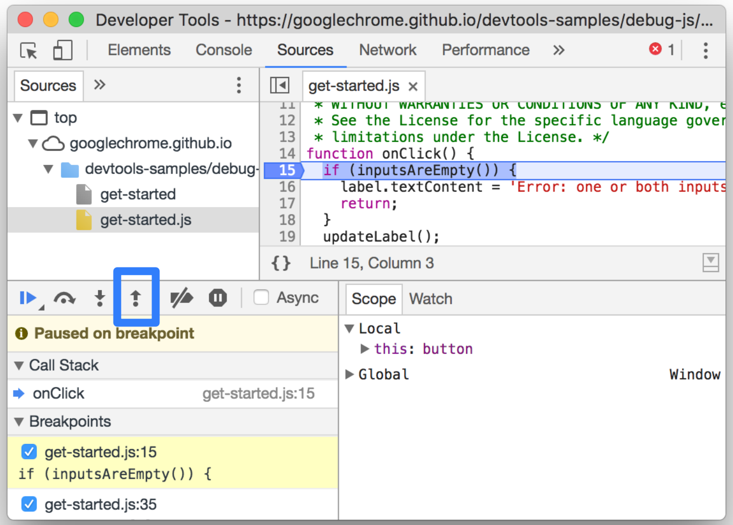
get-started.js:35

```
▼Local
  ▶this: button
▶Global                          Window
```

<_images/step-into-outline.svg>

Run current line, stepping into any function calls

Step Out



```
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, e
 * See the License for the specific language gover
 * limitations under the License. */
function onClick() {
  if (inputsAreEmpty()) {
    label.textContent = 'Error: one or both inputs
    return;
  }
  updateLabel();
```

Line 15, Column 3

Paused on breakpoint

Call Stack

onClick get-started.js:15

Breakpoints

get-started.js:15
`if (inputsAreEmpty()) {`

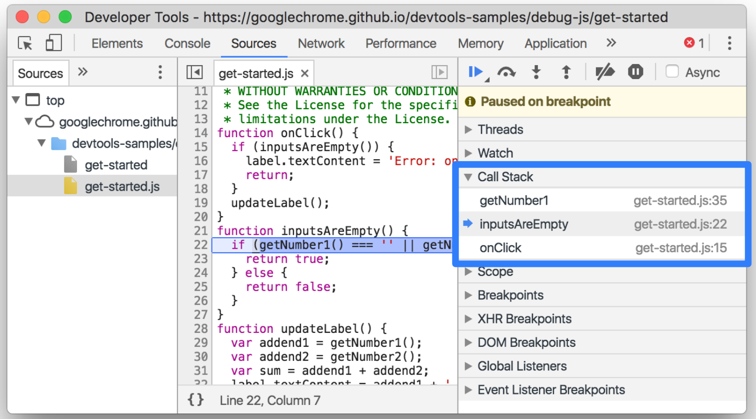
get-started.js:35

```
▼Local
  ▶this: button
▶Global                          Window
```

<_images/step-out-outline.svg>

Return from this function to caller

Call Stack



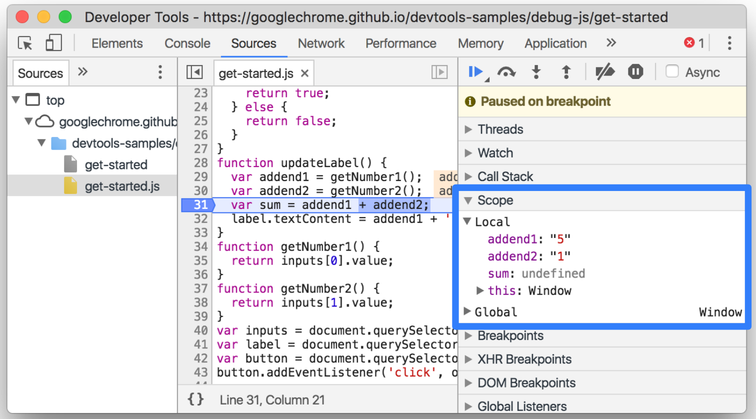
```
 * WITHOUT WARRANTIES OR CONDITION
 * See the License for the specifi
 * limitations under the License.
function onClick() {
  if (inputsAreEmpty()) {
    label.textContent = 'Error: or
    return;
  }
  updateLabel();
}
function inputsAreEmpty() {
  if (getNumber1() === '' || getN
    return true;
  } else {
    return false;
  }
}
function updateLabel() {
  var addend1 = getNumber1();
  var addend2 = getNumber2();
  var sum = addend1 + addend2;
```

```
getNumber1       get-started.js:35
inputsAreEmpty   get-started.js:22
onClick          get-started.js:15
```

<_images/call-stack.svg>

Shows *stack* of function calls that got you here

Scope



```
    return true;
  } else {
    return false;
  }
}
function updateLabel() {
  var addend1 = getNumber1();
  var addend2 = getNumber2();
  var sum = addend1 + addend2;
  label.textContent = addend1 + '
}
function getNumber1() {
  return inputs[0].value;
}
function getNumber2() {
  return inputs[1].value;
}
var inputs = document.querySelecto
var label = document.querySelector
var button = document.querySelecto
button.addEventListener('click', o
```

```
Local
  addend1: "5"
  addend2: "1"
  sum: undefined
  this: Window
Global                    Window
```

<_images/scope.svg>

Shows current value of variables

Can click to change value

Tips To Avoid Bugs

Plan First

> *Software and cathedrals are much the same — first we build them, then we pray.*
>
> *—Sam Redwine, 1988*

Keep It Simple

> *Everyone knows that debugging is twice as hard as writing a program in the first place.*
>
> *If you're as clever as you can be when you write it, how will you ever debug it?*
>
> *—Brian Kernighan, The Elements of Programming Style*

(That's a particularly excellent intermediate book, by the way)

Common JavaScript Bugs

- ***==*** is very loose about comparisons (***===*** isn't)
  - `7 == "7"`
- Objects & arrays are not equal to similar objects & arrays
  - `[1, 2, 3] !== [1, 2, 3]`
- Calling function with missing arguments makes those arguments ***undefined***
- Calling function with extra arguments is ignored (the extra arguments are ignored).
- Getting a missing property from object/index from array is ***undefined***

Good News

> *If debugging is the process of removing bugs, then programming must be the process of putting them in.*
>
> *—Edsger W. Dijkstra*

- Bugs are an opportunity to improve debugging skills & to learn something
- You will have lots of chances to practice this valuable skill!

Error Handling

- Sometimes errors can not be avoided!
- This especially happens when working with external APIs / other people's data
  - Connection failures
  - The API is down
- Instead of errors crashing our program, let's handle them gracefully!

Let's "try" it out!

```
functionThatDoesNotExist();
console.log("did we make it?");
```

```
try {
  functionThatDoesNotExist();
} catch (err){
  console.log("something went wrong!", err);
}

console.log("did we make it?");
```

Notice the important difference here!

Introducing try / catch

```
try {
  // place the code you would like to try to run
} catch(err){
  // if an error occurs, run whatever code we place in here!
}

// keep going!
```

try and ***catch*** create block scope, so if you define a variable inside using ***let*** or ***const*** it will only exist in that block.

The parameter to ***catch*** is optional, but we highly recommend using it so you can see what the error is and any information about it!

Another example

```
function displayInitials(user){
  let firstNameLetter = user.firstName[0].toUpperCase();
  let lastNameLetter = user.lastName[0].toUpperCase();
  return `Hello ${firstNameLetter}.${lastNameLetter}`;
}
```

What happens when we don’t pass in an object?

What happens when our object does not have the correct keys or values?

Using try / catch

```
function displayInitials(user){
  let firstNameLetter;
  let lastNameLetter;
  try {
    firstNameLetter = user.firstName[0].toUpperCase();
    lastNameLetter = user.lastName[0].toUpperCase();
  } catch(e){
    return "Invalid input!";
  }
  return `Hello ${firstNameLetter}.${lastNameLetter}`;
}
```

Strive to only place code that will throw an error inside your try/catch block!

When to use try/catch

Think about using try/catch when:

- The output of your program is unpredictable
- There is any chance that an unexpected error may occur
- You don't want a tremendous amount of conditional logic

Making your own errors!

You can actually create your own errors in JavaScript using the ***throw*** keyword

```
console.log("an error is coming....");
throw "What just happened?!?";
console.log("did we make it?");
```

Exploring with try/catch

```
console.log("an error is coming....");

try {
  throw "Oh no!";
} catch (err) {
  console.log("what happened?", err);
}
```

All we see here is the string that we passed to throw with no information about the kind of error or where it happened.

Throwing better errors

Instead of just using the ***throw*** keyword, you can make a new Error object using the ***new*** keyword.

```
console.log("an error is coming....");

try {
  throw new Error("Oh no!");
} catch (err) {
  console.log("what kind of error?", err.name);
  console.log("what is the message?", err.message);
  console.log("where did it happen?", err.stack);
}
```

Notice that we get more information here including where the error occured!

We'll discuss what ***new*** does in much more detail later in the course, but for now you should use it when making your own errors.

When should you use this?

- any time you want to hide errors from a user
- any time you want to produce custom errors for your users' benefit.
- it works as a nice fallback in areas of your code that could fail under unusual circumstances that were not foreseen during development
- it can hide overly-technical error messages from users who wouldn't understand them anyhow
- use it for portions of your code where you suspect errors will occur that are beyond your control

When should you not use this?

If you know an error is going to occur, because in this case you would want to debug the problem, not hide it.

You still have to debug on your own!

finally

try/catch blocks come with another optional keyword ***finally*** which will execute code regardless if an error is thrown or not

```
try {
  undefined(); // this will throw a TypeError
} catch (err){
  console.log("something went wrong!", err);
} finally {
  console.log("we're all done!");
}
```

You will rarely find yourself using this, but it's good to know if you want to be explicit about what to run no matter what in a try/catch block

Recap

- A SyntaxError happen when JavaScript is unable to parse certain syntax
- A ReferenceError occurs when JavaScript tries to access a variable that is not aware of - remember your scopes!
- A TypeError occurs when JavaScript runs an invalid expression on the wrong type
- You can throw your own errors using the throw keyword
- Using try/catch/finally blocks are an excellent way to handle errors when they may be unpredictable