Week 4 WDD330

lunes, 8 de mayo de 2023 12:39

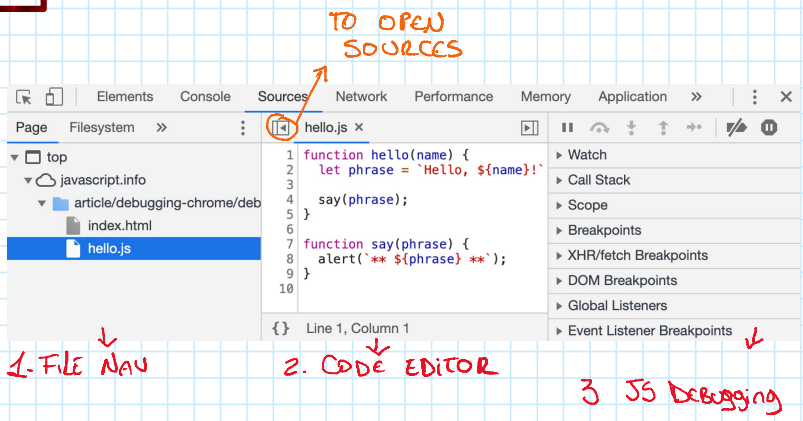
RUTH CHADONA!

DEBUGGING IN THE BROWSER

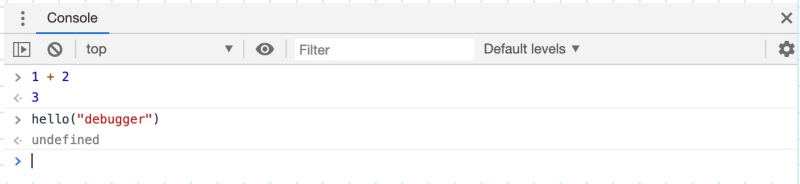
DEBUGGING IS THE PROCESS OF FINDING AND FIXING ERRORS WITHIN A SCRIPT

DEBUGGER FOR CHROME:

F12 => TO TURN ON DEVELOPER TOOLS

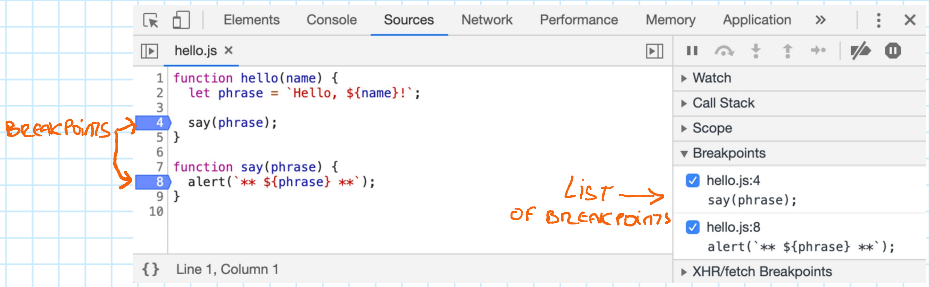


ESC TO OPEN A CONSOLE BELOW



BREAKPOINTS

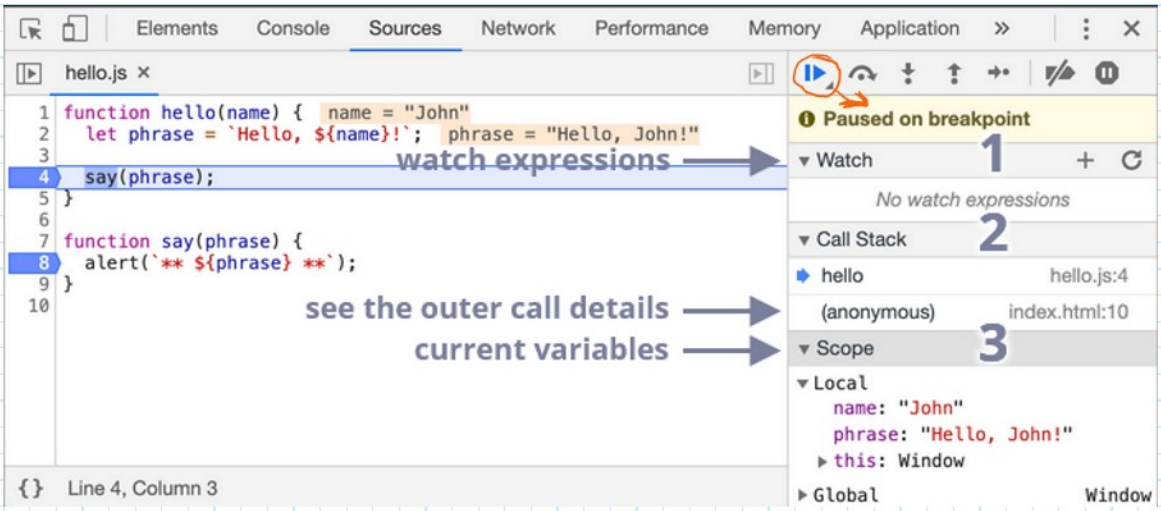
A POINT OF CODE WHERE DE DEBUGGER WILL AUTOMATICALLY PAUSE THE JS EXECUTION



THE COMMAND "DEBUGGER"

```
4   debugger;  // <-- the debugger stops here
5
```

WE CAN PAUSE THE CODE BY USING THE DEBUGGER COMMAND



1 WATCH = SHOWS CURRENT VALUES FOR ANY EXPRESSIONS

2 CALL STACK = SHOWS THE NESTED CALLS CHAIN

3 SCOPE = CURRENT VARIABLES (LOCAL AND GLOBAL)

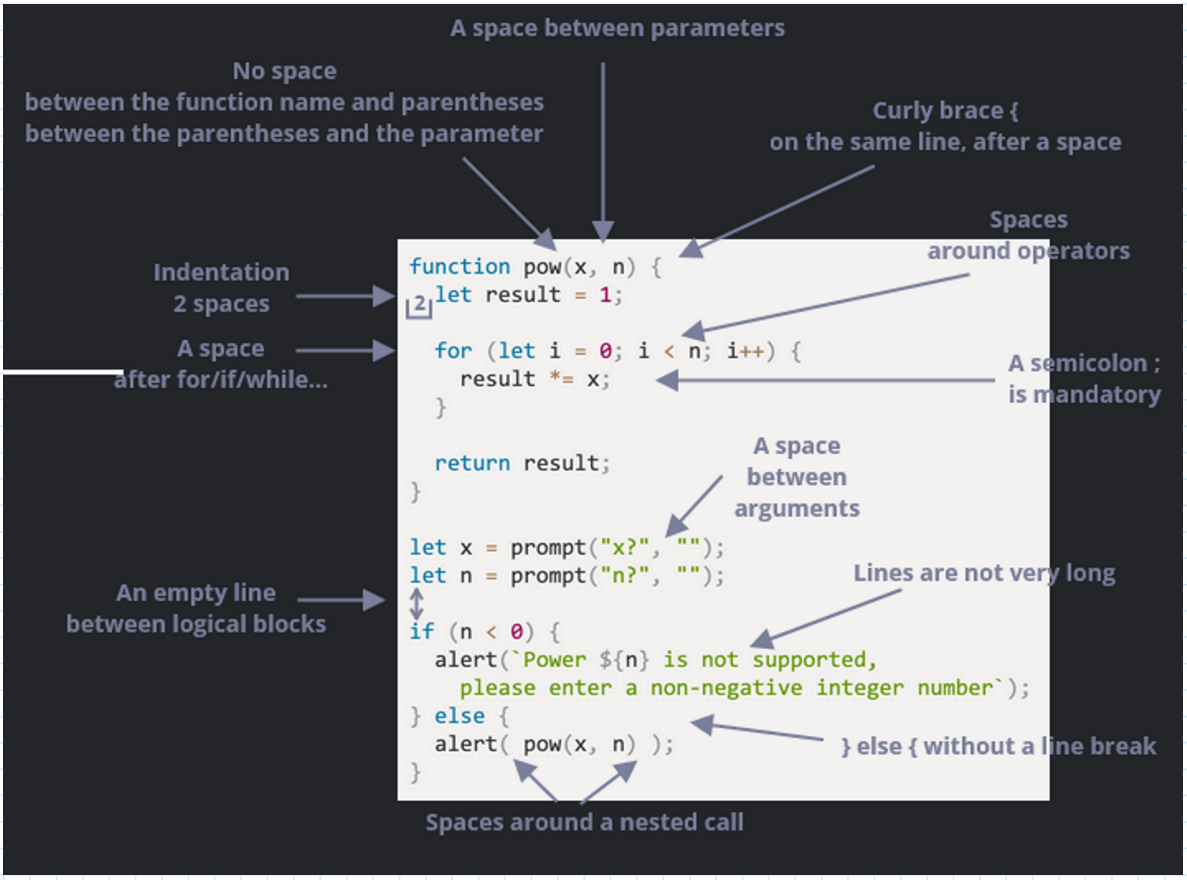
- – "Resume": continue the execution, hotkey `F8`.
- – "Step": run the next command, hotkey `F9`.
- – "Step over": run the next command, but *don't go into a function*, hotkey `F10`.
- – "Step into", hotkey `F11`.
- – "Step out": continue the execution till the end of the current function, hotkey `Shift+F11`.
- – enable/disable all breakpoints.
- – enable/disable automatic pause in case of an error.

"CONTINUE TO HERE" → RIGHT CLICK ON A LINE OF CODE, YOU WILL SEE THE OPTION. IT'S FOR MOVING MULTIPLE STEPS FOWARD TO THE LINE.

Logging: TO OUTPUT SOMETHING TO CONSOLE FROM OUR CODE, WE USE THE `CONSOLE.LOG()` FUNCTION.

CODING STYLE

OUR CODE MUST BE AS CLEAN AND EASY TO READ AS POSSIBLE



CURLY BRACES = - FOR A VERY BRIEF CODE IS ALLOWED ONE LINE

```
if (cond) return null.
```

- A BLOCK CODE IS USUALLY MORE READABLE

```
1  if (condition) {
2    // do this
3    // ...and that
4    // ...and that
5  }
```

LINE LENGTH = - IT'S BEST PRACTICE TO SPLIT LONG TEXT, CODES ...

- BACKTICK QUOTES ` ` ALLOW TO SPLIT STRING INTO MULTIPLE LINES

- THE MAXIMUM LINE LENGTH IS USSUALLY 80 OR 120 CHARACTERS

And, for `if` statements:

```
1  if (
2    id === 123 &&
3    moonPhase === 'Waning Gibbous' &&
4    zodiacSign === 'Libra'
5  ) {
6    letTheSorceryBegin();
7  }
```

INDENTS = - HORIZONTAL INDENTS: 2 OR 4 SPACES
- TAB KEY

```
show(parameters,
     aligned, // 5 spaces padding at the left
     one,
     after,
     another
  ) {
  // ...
}
```

- VERTICAL INDENTS: EMPTY LINES FOR SPLITTING CODE INTO LOGICAL BLOCKS
- NO MORE THAN 9 OF CODES WITHOUT VERTICAL SPACE

```
function pow(x, n) {
  let result = 1;
  //             <--
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  //             <--
  return result;
}
```

SEMICOLONS = - IT SHOULD BE PRESENT AFTER EACH STATEMENT, EVEN IF IT COULD POSSIBLY BE SKIPPED.
- DON'T FORGET SEMICOLONS ☹

NESTING LEVELS = - AVOID TO MANY LEVELS DEEP OR EXTRA NESTING

FUNCTION PLACEMENT

1. DECLARE THE FUNCTIONS ABOVE THE CODE THAT USES THEM.
2. CODE FIRST, THEN FUNCTIONS. (THIS IS PREFERRED)
3. MIXED: A FUNCTION IS DECLARED WHERE IT'S FIRST USED.

STYLE GUIDES

CONTAINS GENERAL RULES ABOUT "HOW TO WRITE" CODE.

Some popular choices:

- Google JavaScript Style Guide
- Airbnb JavaScript Style Guide
- Idiomatic.JS
- StandardJS
- (plus many more)

AUTOMATED LINTERS

- LINTERS ARE TOOLS THAT CAN AUTOMATICALLY CHECK THE STYLE OF YOUR CODE AND MAKE IMPROVING SUGGESTIONS
- USING A LINT IS RECOMMENDED

Here are some well-known linting tools:

- JSLint – one of the first linters.
- JSHint – more settings than JSLint.
- ESLint – probably the newest one.

"WHAT MAKES THE CODE MORE READABLE AND EASIER TO UNDERSTAND?"
"WHAT CAN HELP US AVOID ERRORS?"

ERROR HANDLING, "TRY... CATCH"

THIS SYNTAX CONSTRUCT ALLOWS US TO "CATCH" ERRORS SO THE SCRIPT DON'T DIE OR STOP AND HANDLER THE ERROR

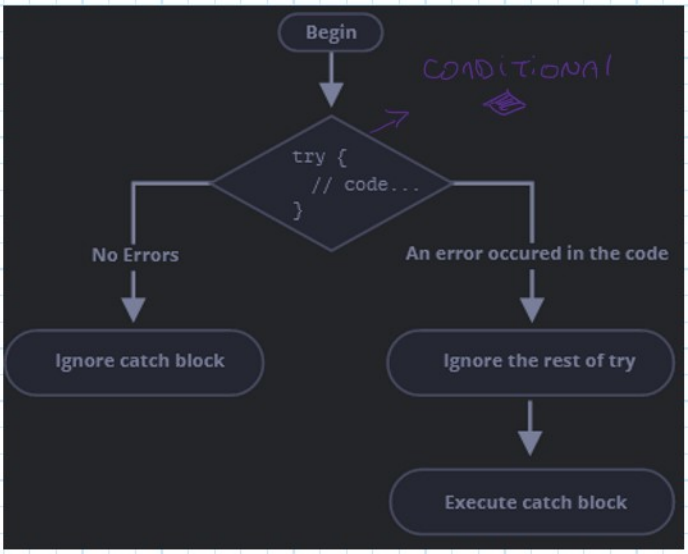
```
try {

  // code...

} catch (err) {

  // error handling

}
```



TRY... CATCH ONLY WORKS FOR RUNTIME ERRORS (VALID JS CODE)

- IF THE CODE IS SYNTACTICALLY WRONG, IT WON'T WORK.

TRY ... CATCH WORKS SYNCHRONOUSLY

- TO CATCH AN EXCEPTION INSIDE A SCHEDULE FUNCTION, THE TRY-CATCH MUST BE INSIDE THE FUNCTION : EXAMPLE WITH SETTIMEOUT()

```
setTimeout(function() {
  try {
    noSuchVariable; // try...catch handles the error!
  } catch {
    alert( "error is caught here!" );
  }
}, 1000);
```

ERROR OBJECT

JS GENERATES AN OBJECT WHEN AN ERROR OCCURS

- THE OBJECT IS PASS AS AN ARGUMENT

```
try {
  lalala; // error, variable is not defined!
} catch (err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala is not defined
  alert(err.stack); // ReferenceError: lalala is not defined at (...call stack)
                    (STRING)

  // Can also show an error as a whole
  // The error is converted to string as "name: message"
  alert(err); // ReferenceError: lalala is not defined
}
```

Optional “catch” binding

A recent addition

This is a recent addition to the language. Old browsers may need polyfills.

If we don't need error details, `catch` may omit it:

```
try {
  // ...
} catch { // <-- without (err)
  // ...
}
```

JSON.PARSE, TRY... CATCH EXAMPLE

```
let json = "{ bad json }";

try {

  let user = JSON.parse(json); // <-- when an error occurs...
  alert( user.name ); // doesn't work

} catch (err) {
  // ...the execution jumps here
  alert( "Our apologies, the data has errors, we'll try to request it one more time." );
  alert( err.name );
  alert( err.message );
}
```

- THIS WILL DISPLAY THE ERROR TO THE VIEW OF THE CLIENT
- JSON.PARSE GENERATES AN ERROR WHEN JSON IS MALFORMED AND THE SCRIP DIES.
- WITH A CATCH BLOCK YOU CAN: - SHOW MESSAGE
  - SEND A NEW NETWORK REQUEST
  - SUGGEST AN ALTERNATIVE TO THE VISITOR
  - SEND INFORMATION TO A LOGGING FACILITY
  - ETC

THROW OPERATOR

```
throw <error object>
```

- GENERATES AN ERROR
- IT'S PREFERABLY TO USE OBJECTS PREFERABLY WITH NAME AND MESSAGE PROPERTIES.

JS BUILT-IN CONSTRUCTORS FOR STANDARD ERRORS:

LET ERROR = NEW...

- ERROR()
- SYNTAXERROR()
- REFERENCEERROR()
- TYPEERROR()

```
let json = '{ "age": 30 }'; // incomplete data

try {

  let user = JSON.parse(json); // <-- no errors

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // (*)
  }

  alert( user.name );

} catch (err) {
  alert( "JSON Error: " + err.message ); // JSON Error: Incomplete data: no name
}
```

(*) THROW OPERATOR GENERATES A SYNTAX ERROR WITH THE GIVEN MESSAGE

→ CATCH BECAME A SINGLE PLACE FOR ALL ERROR HANDLING

RETHROWING

CATCH SHOULD ONLY PROCESS ERRORS THAT IT KNOWS AND "RETHROW" ALL OTHERS. (SKIP THEM)

1_ CATCH GETS ALL ERRORS
2_ IN THE CATCH (ERR) {...} BLOCK WE ANALYZE THE ERROR OBJECT ERR.
3_ IF WE DON'T KNOW HOW TO HANDLE IT, WE DO THROW ERR.

- WE CAN USE INSTANCEOF TO CHECK THE ERROR

```
let json = '{ "age": 30 }'; // incomplete data
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name");
  }

  blabla(); // unexpected error

  alert( user.name );

} catch (err) {

  if (err instanceof SyntaxError) {
    alert( "JSON Error: " + err.message );
  } else {
    throw err; // rethrow (*)
  }

}
```

- WE CAN USE MORE LEVELS OF TRY... CATCH TO HANDLER ERRORS

```
function readData() {
  let json = '{ "age": 30 }';

  try {
    // ...
    blabla(); // error!
  } catch (err) {
    // ...
    if (!(err instanceof SyntaxError)) {
      throw err; // rethrow (don't know how to deal with it)
    }
  }
}

try {
  readData();
} catch (err) {
  alert( "External catch got: " + err ); // caught it!
}
```

TRY... CATCH... FINALLY

```
try {
  alert( 'try' );
  if (confirm('Make an error?')) BAD_CODE();
} catch (err) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

The code has two ways of execution:

1. If you answer "Yes" to "Make an error?", then `try -> catch -> finally`.
2. If you say "No", then `try -> finally`.

THE FINALLY CLAUSE IS USED WHEN WE START DOING SOMETHING AND WANT TO FINALIZE IT IN ANY CASE OF OUTCOME

TRY... FINALLY (WITHOUT CATCH)

WE APPLY IT WHEN WE DON'T WANT TO HANDLE ERRORS BUT WANT TO MAKE SURE THAT WE STARTED ARE FINALIZED

GLOBAL CATCH

- FOR FATAL ERRORS OUTSIDE OF TRY... CATCH
- THE BROWSER CAN ASSIGN A FUNCTION TO THE SPECIAL WINDOW.ONERROR PROPERTY THAT WILL RUN IN CASE OF AN UNCAUGHT ERROR.

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

ERROR — LINE AND COLUMN — WHERE IN THE SCRIPT IS THE ERROR — ERR OBJ

- THIS PROPERTY DO NOT RECOVER THE SCRIPT EXECUTION BUT IT SENDS THE ERROR MESSAGE TO DEVELOPERS.