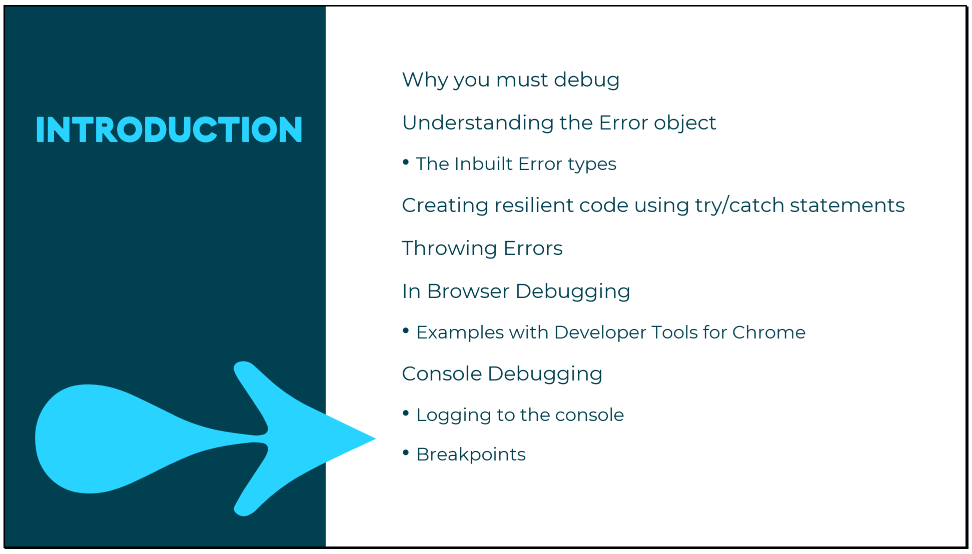
Slide 1

QA POWERING POTENTIAL

ERROR HANDLING & DEBUGGING

JavaScript Fundamentals

Slide 2



INTRODUCTION

Why you must debug

Understanding the Error object

- The Inbuilt Error types

Creating resilient code using try/catch statements

Throwing Errors

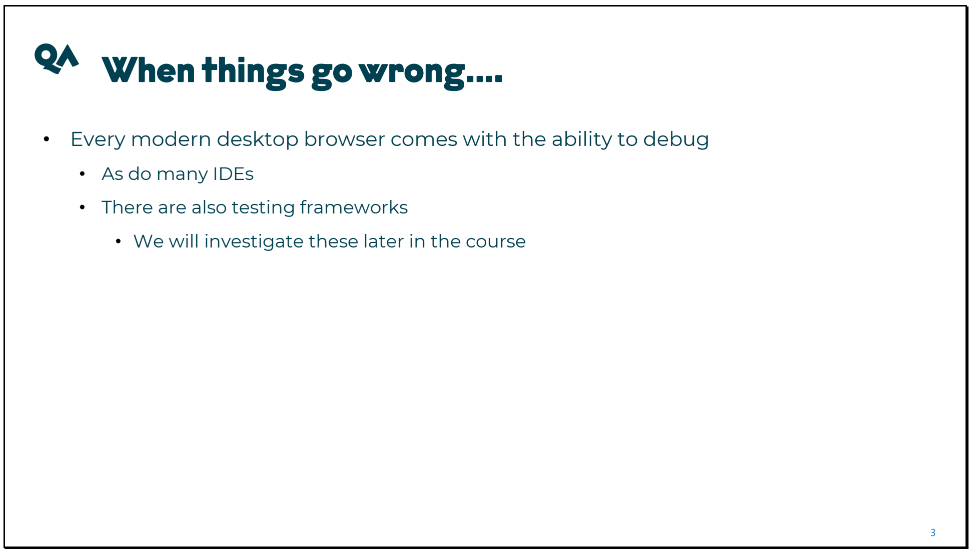
In Browser Debugging

- Examples with Developer Tools for Chrome

Console Debugging

- Logging to the console
- Breakpoints

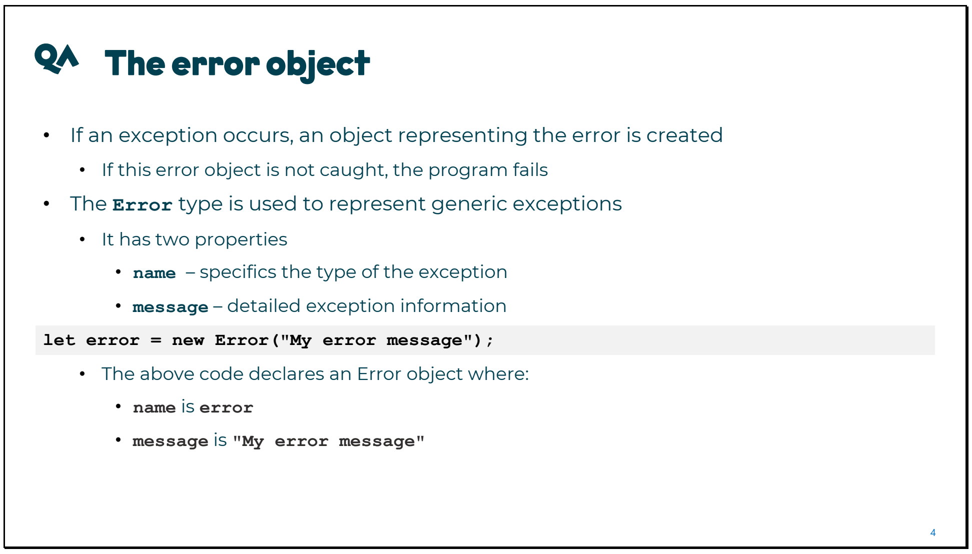
Slide 3



When things go wrong....

- Every modern desktop browser comes with the ability to debug
  - As do many IDEs
  - There are also testing frameworks
    - We will investigate these later in the course

Slide 4



The error object

- If an exception occurs, an object representing the error is created
  - If this error object is not caught, the program fails
- The **`Error`** type is used to represent generic exceptions
  - It has two properties
    - **`name`** – specifics the type of the exception
    - **`message`** – detailed exception information

```
let error = new Error("My error message");
```

  - The above code declares an Error object where:
    - `name` is `error`
    - `message` is `"My error message"`

4

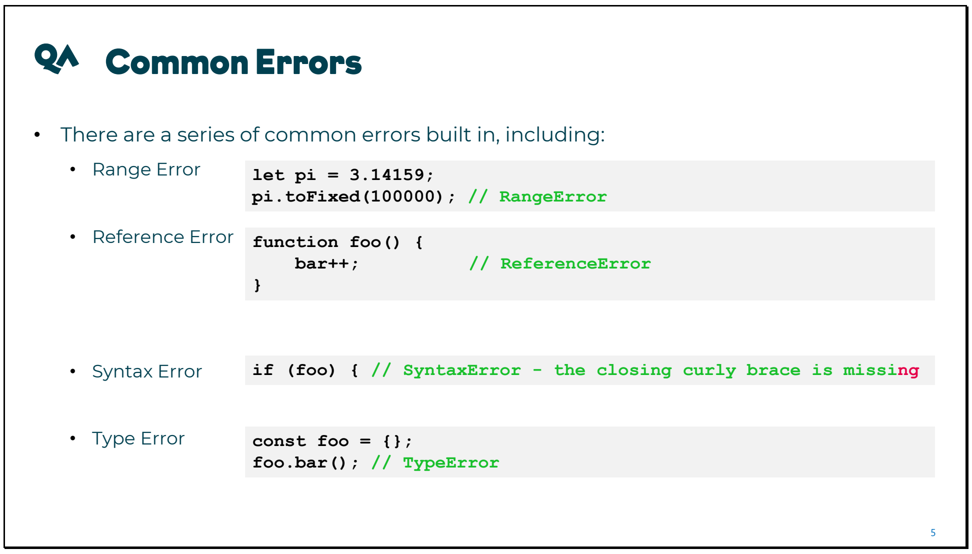
When a JavaScript statement generates an error, it is said to throw an exception. Instead of proceeding to the next statement, the JavaScript interpreter checks for exception handling code. If there is no exception handler, then the program returns from whatever function threw the exception. This is repeated for each function on the call stack until an exception handler is found or until the top level function is reached, causing the program to terminate.

The "Error" type is used to represent generic exceptions. This type of exception is most often used for implementing user defined exceptions. The topic of creating user-defined exceptions will be revisited later in this article. "Error" objects are instantiated by calling their constructor as shown in the following example:

```
let error = new Error("error message");
```

"Error" objects contain two properties: "name" and "message". The "name" property specifies the type of exception (in this case "Error"). The "message" property provides a more detailed description of the exception. The "message" gets its value from the string passed to the exception's constructor. The remaining exception types represent more specific types of errors, but they are all used in the same fashion as the generic "Error" type.

Slide 5



Common Errors

- There are a series of common errors built in, including:
  - Range Error

```
let pi = 3.14159;
pi.toFixed(100000); // RangeError
```

  - Reference Error

```
function foo() {
     bar++;          // ReferenceError
}
```

  - Syntax Error

```
if (foo) { // SyntaxError - the closing curly brace is missing
```

  - Type Error

```
const foo = {};
foo.bar(); // TypeError
```

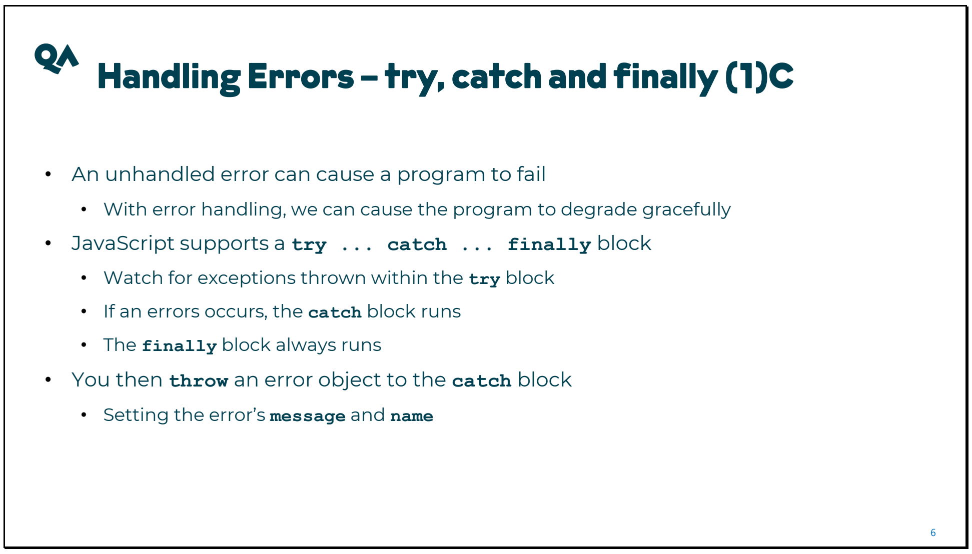
5

RangeError
Are generated by numbers that fall outside of a specified range. In the code above, the argument is expected to be between 0 and 20 (although some browsers support a wider range). If the value of “digits” is outside of this range, then a “RangeError” is thrown.

ReferenceError
Is thrown when a non-existent variable is accessed. These exceptions commonly occur when an existing variable name is misspelled. In the example above, a “ReferenceError” occurs when “bar” is accessed. Note that this example assumes that “bar” does not exist in any active scope when the increment operation is attempted.

SyntaxError
Occurs when the rules of the JavaScript language are broken. Developers who are familiar with languages, such as C and Java are used to encountering syntax errors during the compilation process. However, because JavaScript is an interpreted language, syntax errors are not identified until the code is executed. Syntax errors are unique as they are the only type of exception that cannot be recovered from. The following example generates a syntax error because the “if” statement is missing a closing curly brace.

Slide 6



Handling Errors – try, catch and finally (1)C

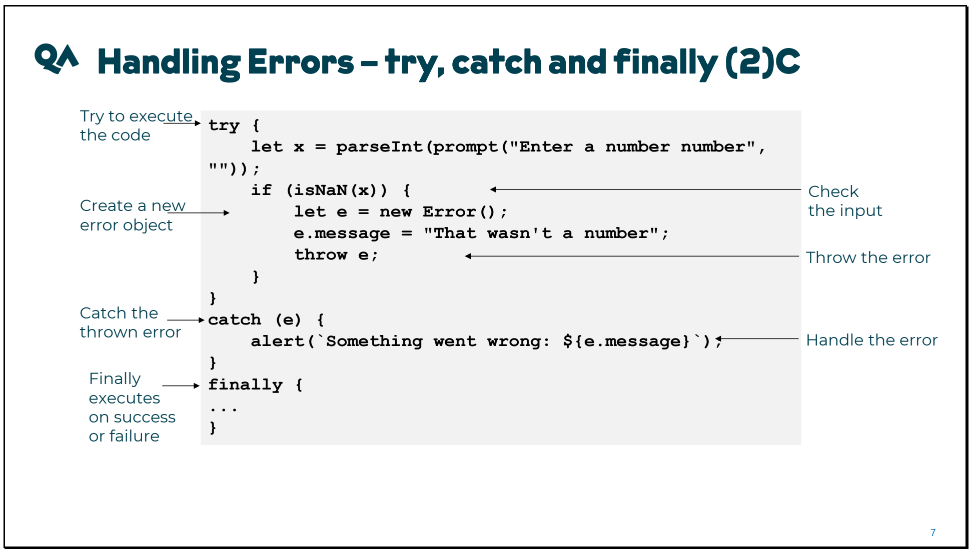
- An unhandled error can cause a program to fail
  - With error handling, we can cause the program to degrade gracefully
- JavaScript supports a **`try ... catch ... finally`** block
  - Watch for exceptions thrown within the **`try`** block
  - If an errors occurs, the **`catch`** block runs
  - The **`finally`** block always runs
- You then **`throw`** an error object to the **`catch`** block
  - Setting the error's **`message`** and **`name`**

6

The try…catch…finally block is common to most modern languages. Processing proceeds as follows: within the try block, if no exceptions are thrown, the finally block is run.
If an exception is thrown, the catch block is run followed by the finally block.
Please note that the finally block runs, even if the try or catch blocks return out of the current function.

Both the catch and the finally blocks are optional: you do not need to catch the exceptions, you do not need to have code that runs after the try … catch. If the exception is not caught, it is thrown to calling function. If the exception is not caught at any level, then browser will log the error.

Slide 7



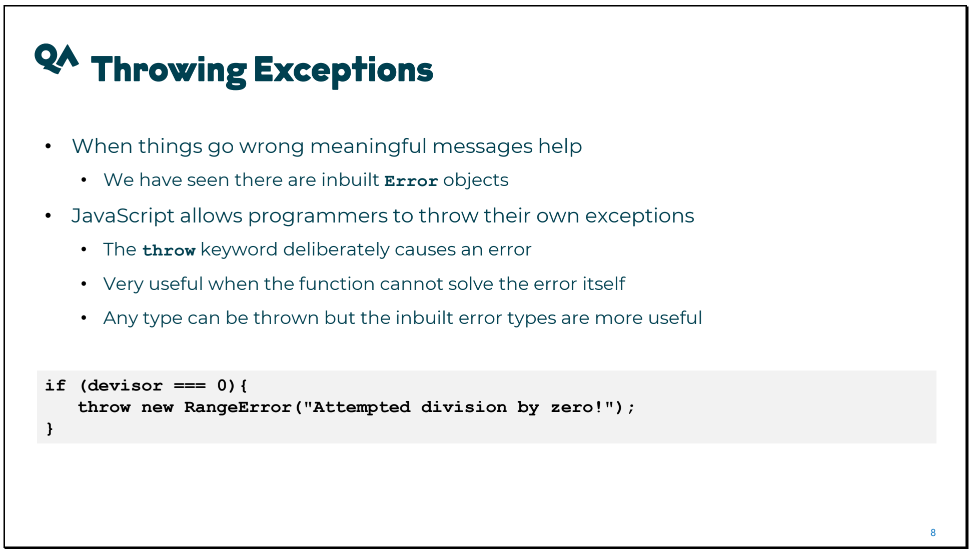
Handling Errors – try, catch and finally (2)C

```
try {
    let x = parseInt(prompt("Enter a number number",
""));
    if (isNaN(x)) {
        let e = new Error();
        e.message = "That wasn't a number";
        throw e;
    }
}
catch (e) {
    alert(`Something went wrong: ${e.message}`);
}
finally {
...
}
```

Try to execute the code
Create a new error object
Catch the thrown error
Finally executes on success or failure
Check the input
Throw the error
Handle the error
7

The above code takes us through an error handling scenario using try catch and finally. It also introduces us to the JavaScript error object. The Error object represents a runtime error allowing a developer to wrap up useful information about what went wrong. As we can see above, it was possible to provide a message and even name the error.

Slide 8



Throwing Exceptions

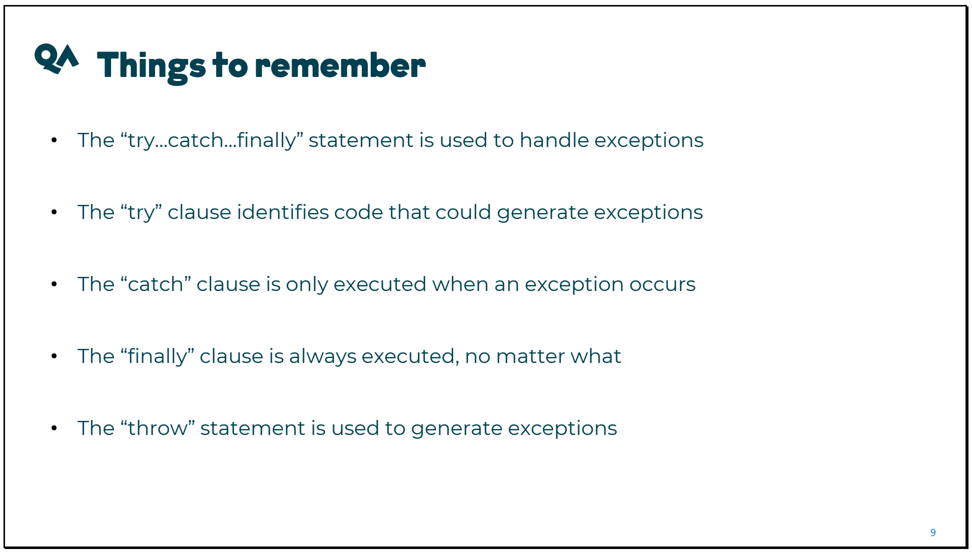
- When things go wrong meaningful messages help
  - We have seen there are inbuilt **`Error`** objects
- JavaScript allows programmers to throw their own exceptions
  - The **`throw`** keyword deliberately causes an error
  - Very useful when the function cannot solve the error itself
  - Any type can be thrown but the inbuilt error types are more useful

```
if (devisor === 0){
   throw new RangeError("Attempted division by zero!");
}
```

8

JavaScript allows programmers to throw their own exceptions via the appropriately named “throw” statement. This concept can be somewhat confusing to inexperienced developers. After all, developers strive to write code that is free of errors, yet the “throw” statement intentionally introduces them. However, intentionally throwing exceptions can actually lead to code that is easier to debug and maintain. For example, by creating meaningful error messages, it becomes easier to identify and resolve problems.

Slide 9



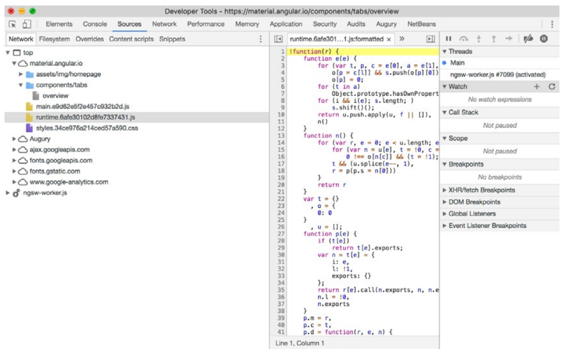
Things to remember

- The “try…catch…finally” statement is used to handle exceptions
- The “try” clause identifies code that could generate exceptions
- The “catch” clause is only executed when an exception occurs
- The “finally” clause is always executed, no matter what
- The “throw” statement is used to generate exceptions

Slide 10

Debugging Demonstration

- Contains a JavaScript Console
  - DOM explorer
  - Console
  - CSS style browser
  - Can debug
    - Command line interface
    - Not trivial



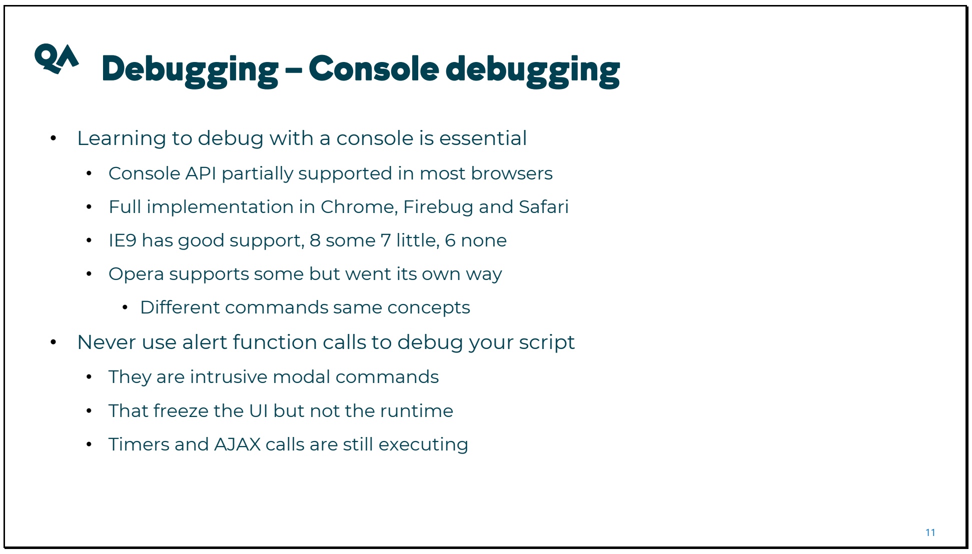
```
!function(r) {
    function e(e) {
        for (var t, p, c = e[0], a = e[1],
            o[p = c[l]] && s.push(o[p][0])
            o[p] = 0;
        for (t in a)
            Object.prototype.hasOwnPropert
        for (i && i(e); s.length; )
            s.shift()();
        return u.push.apply(u, f || []),
        n()
    }
    function n() {
        for (var r, e = 0; e < u.length; e
            for (var n = u[e], t = !0, c =
                0 !== o[n[c]] && (t = !1);
            t && (u.splice(e--, 1),
            r = p(p.s = n[0]))
        }
        return r
    }
    var t = {}
      , o = {
        0: 0
    }
      , u = [];
    function p(e) {
        if (t[e])
            return t[e].exports;
        var n = t[e] = {
            i: e,
            l: !1,
            exports: {}
        };
        return r[e].call(n.exports, n, n.e
        n.l = !0,
        n.exports
    }
    p.m = r,
    p.c = t,
    p.d = function(r, e, n) {
```

Line 1, Column 1

10

Your instructor may choose to give a demonstration of using a browser’s debug tools.

Slide 11



Debugging – Console debugging

- Learning to debug with a console is essential
  - Console API partially supported in most browsers
  - Full implementation in Chrome, Firebug and Safari
  - IE9 has good support, 8 some 7 little, 6 none
  - Opera supports some but went its own way
    - Different commands same concepts
- Never use alert function calls to debug your script
  - They are intrusive modal commands
  - That freeze the UI but not the runtime
  - Timers and AJAX calls are still executing

11

As a client side developer, you need to learn to debug and test. As we have discovered so far, JavaScript is an amazingly powerful programming language, but things can easily go wrong and you will need to identify how and why.

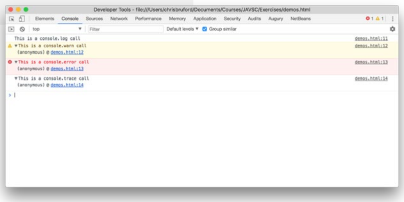
Some developers use alert calls to test a function is being called. One simple piece of advice: DON'T! Debugging with alerts holds the UI in a state of limbo, but does not hold any asynchronous operations, such as timers or AJAX feeds.

Slide 12

Debugging – Console logging

- The console API has a number of useful functions

Function	Description
console.log()	Writes a message to the console.
console.warn()	As above with visual "warning" icon
console.error()	As above with visual "error" icon
console.trace()	As above with a call trace

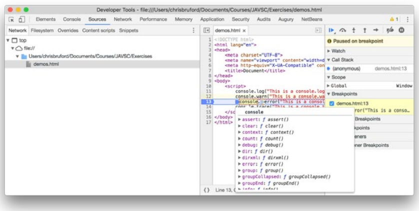


```
  This is a console.log call                         demos.html:11
⚠ ▼This is a console.warn call                       demos.html:12
    (anonymous) @ demos.html:12
⊗ ▼This is a console.error call                      demos.html:13
    (anonymous) @ demos.html:13
  ▼This is a console.trace call                      demos.html:14
    (anonymous) @ demos.html:14
> |
```

Slide 13

Debugging – breakpoints

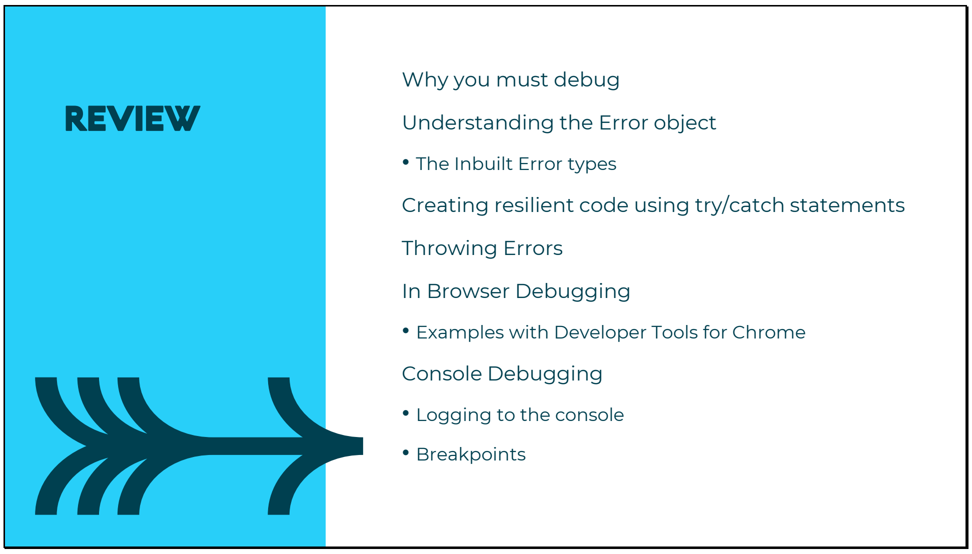
- Breakpoints pauses the code allowing examination and debugging
  - From the developer tools select the sources panel and select from a JavaScript source file
  - Set breakpoint by clicking in the gutter of the line you want to pause execution at
  - Hover over the source code to inspect variables and functions
  - Delete the breakpoint by clicking the blue tag breakpoint indicator


Developer Tools - file:///Users/chrisbruford/Documents/Courses/JAVSC/Exercises/demos.html
Elements Console Sources Network Performance Memory Application Security Audits Augury NetBeans
Network Filesystem Overrides Content scripts Snippets
top
file://
Users/chrisbruford/Documents/Courses/JAVSC/Exercises
demos.html
Paused on breakpoint
Watch
Call Stack
(anonymous) demos.html:13
Scope
Global Window
Breakpoints
demos.html:13
Line 13,

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=d
    <meta http-equiv="X-UA-Compatible" con
    <title>Document</title>
</head>
<body>
    <script>
        console.log("This is a console.log
        console.warn("This is a console.wa
        console.error("This is a consol
        console.trace("This is a console.t
    </sc
</body>
</html>
```

```
console
▶ assert: ƒ assert()
▶ clear: ƒ clear()
▶ context: ƒ context()
▶ count: ƒ count()
▶ debug: ƒ debug()
▶ dir: ƒ dir()
▶ dirxml: ƒ dirxml()
▶ error: ƒ error()
▶ group: ƒ group()
▶ groupCollapsed: ƒ groupCollapsed()
▶ groupEnd: ƒ groupEnd()
▶ info: ƒ info()
```

Slide 14



REVIEW

Why you must debug

Understanding the Error object

- The Inbuilt Error types

Creating resilient code using try/catch statements

Throwing Errors

In Browser Debugging

- Examples with Developer Tools for Chrome

Console Debugging

- Logging to the console
- Breakpoints