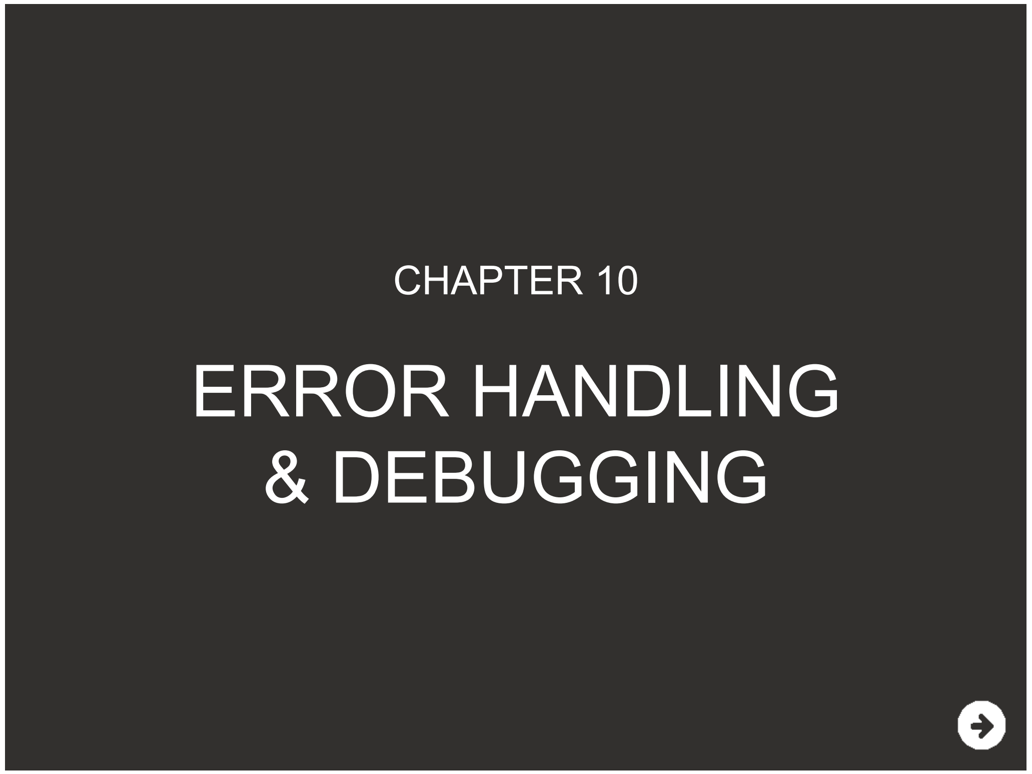
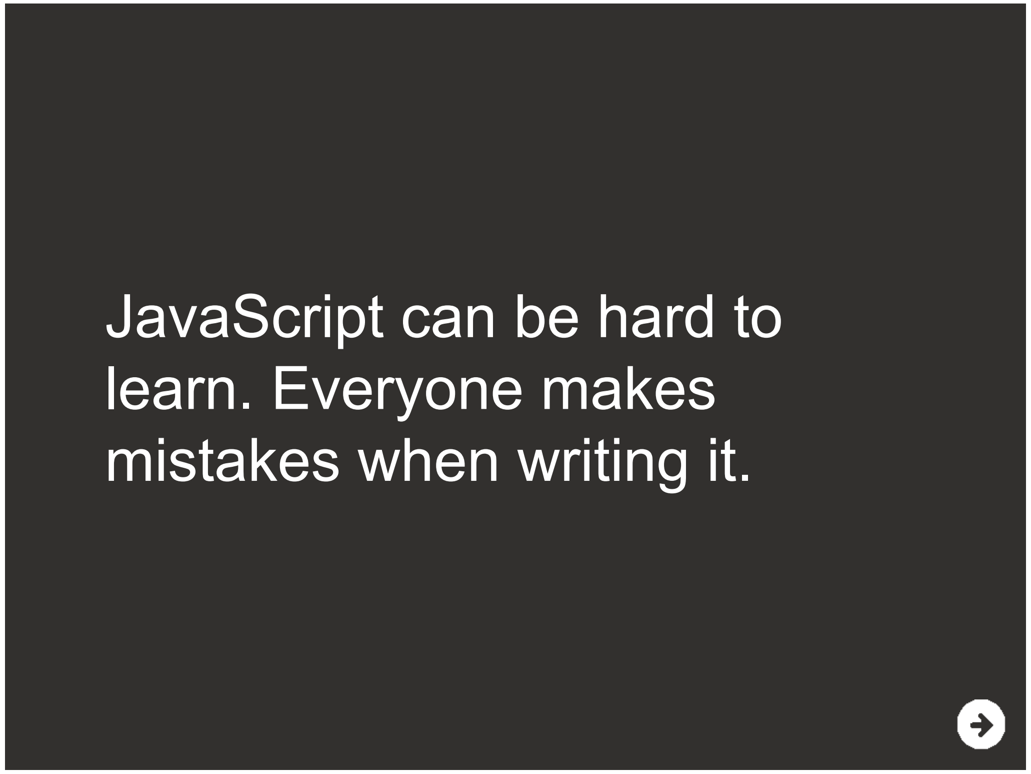
JAVASCRIPT & JQUERY

interactive front-end web development



CHAPTER 10

ERROR HANDLING & DEBUGGING



JavaScript can be hard to learn. Everyone makes mistakes when writing it.

Error messages can help you understand what has gone wrong and how to fix it.

You will learn about:

The console and developer tools
Common problems
Handling errors

HOW JAVASCRIPT WORKS

To find the source of an error it helps to understand how scripts are processed.

The **order of execution** is the order in which lines of code are executed or run.

LOOK AT THIS SCRIPT:

```
    function greetUser() {
  return 'Hello ' + getName();
}

function getName() {
  var name = 'Molly';
  return name;
}

var greeting = greetUser();
alert(greeting);
```

```
    function greetUser() {
  return 'Hello ' + getName();
}

function getName() {
  var name = 'Molly';
  return name;
}

① var greeting = greetUser();
alert(greeting);
```

```
② function greetUser() {
  return 'Hello ' + getName();
}

function getName() {
  var name = 'Molly';
  return name;
}

var greeting = greetUser();
alert(greeting);
```

```
    function greetUser() {
  return 'Hello ' + getName();
}

③ function getName() {
  var name = 'Molly';
  return name;
}

var greeting = greetUser();
alert(greeting);
```

```
    function greetUser() {
  return 'Hello ' + getName();
}

function getName() {
  var name = 'Molly';
  return name;
}

var greeting = greetUser();
```
4
```
alert(greeting);
```

There are **execution contexts**:

One global context
And a new execution context for each new function

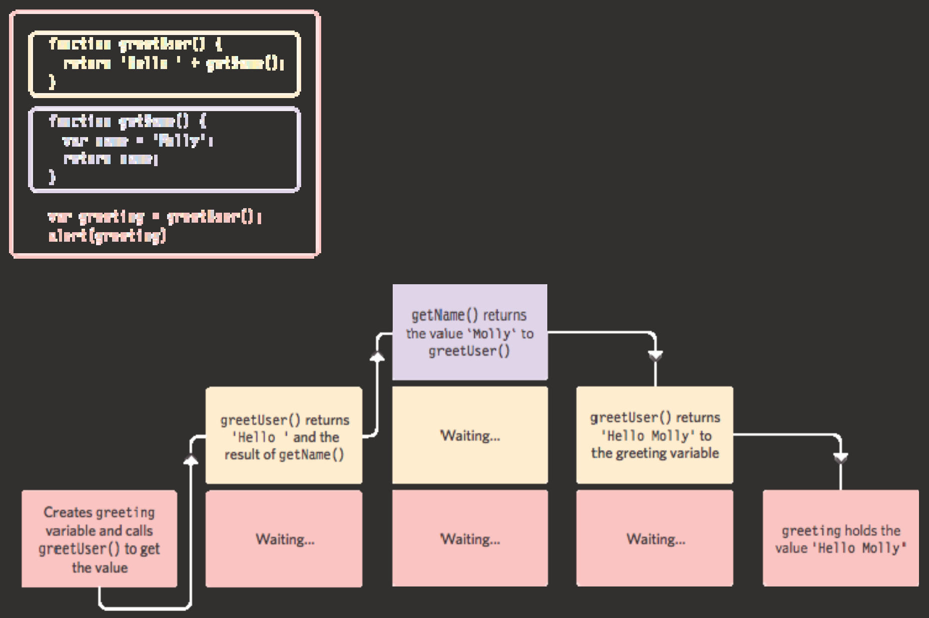
GLOBAL CONTEXT
(global scope)

FUNCTION CONTEXT
(function-level scope)

```
    function greetUser() {
  return 'Hello ' + getName();
}

function getName() {
  var name = 'Molly';
  return name;
}

var greeting = greetUser();
alert(greeting);
```

The JavaScript interpreter processes code one line at a time.

If a statement needs data from another function, it stacks (or piles) functions on top of the current task.



```
function greetUser() {
  return 'Hello ' + getName();
}
function getName() {
  var name = 'Molly';
  return name;
}
var greeting = greetUser();
alert(greeting)
```

Creates greeting variable and calls greetUser() to get the value
greetUser() returns 'Hello ' and the result of getName()
getName() returns the value 'Molly' to greetUser()
Waiting...
greetUser() returns 'Hello Molly' to the greeting variable
Waiting...
Waiting...
Waiting...
greeting holds the value 'Hello Molly'

When a script enters a new execution context, there are two phases of activity:

1: **Prepare**
2: **Execute**

ERRORS

If a JavaScript statement generates an error, then it throws an **exception**.

It stops… and looks for exception handling code.

If error handling code cannot be found in the current function, it goes up a level.

If error handling code cannot be found at all, the script stops running.

An `Error` object is created.

`Error` objects help you find where your errors are.

Browsers have tools to help you read them.

`Error` objects have these properties:

`name`	type of execution
`message`	description of error
`fileName`	name of JavaScript file
`lineNumber`	line number of error

Seven types of `Error` object:

```
Error
SyntaxError
ReferenceError
TypeError
RangeError
URIError
EvalError
```

A DEBUGGING WORKFLOW

Debugging is about deduction and eliminating potential causes of errors.

To find out where the problem is, you can check…

1

- The error message
- The line number
- The type of error

2

How far the script has run

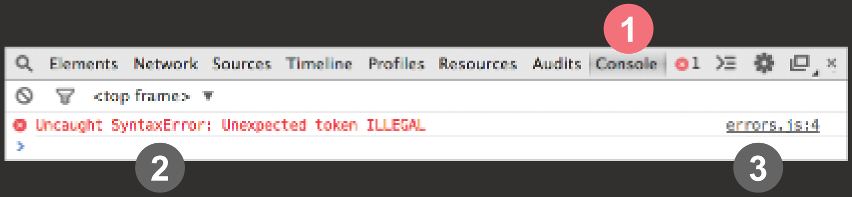
3

Values in code by setting breakpoints and comparing the values you expect to what the variables hold

THE CONSOLE & DEVELOPER TOOLS

All modern browsers have developer tools to help you debug scripts.

Start by opening the JavaScript console.


1
Elements Network Sources Timeline Profiles Resources Audits Console
1
<top frame>
2
3

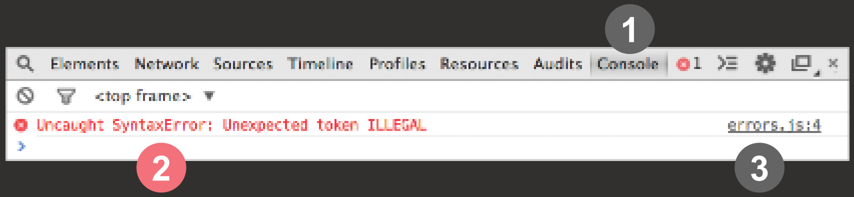
```
Uncaught SyntaxError: Unexpected token ILLEGAL                errors.js:4
```

1: **Console** is selected

2: **Type of error** (`SyntaxError`)

3: **File name** and **line number**:

(`errors.js:4`)


1
Elements Network Sources Timeline Profiles Resources Audits Console
1
<top frame>
2
3

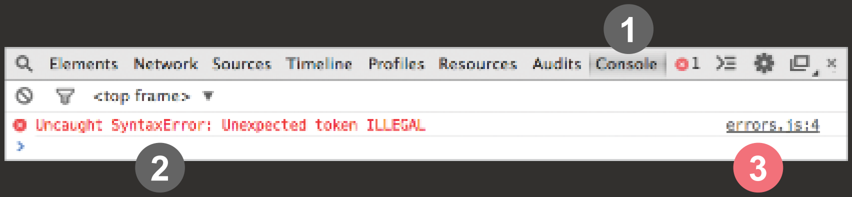
```
Uncaught SyntaxError: Unexpected token ILLEGAL                errors.js:4
```

1: **Console** is selected

2: **Type of error** (`SyntaxError`)

3: **File name** and **line number**:

(`errors.js:4`)


1
Elements Network Sources Timeline Profiles Resources Audits Console
1
<top frame>
2
3

```
Uncaught SyntaxError: Unexpected token ILLEGAL                errors.js:4
```

1: **Console** is selected

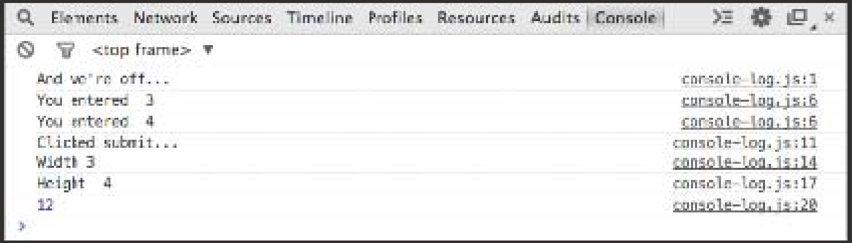
2: **Type of error** (`SyntaxError`)

3: **File name** and **line number**:

(`errors.js:4`)

You can just type code into the console and it will show you a result.

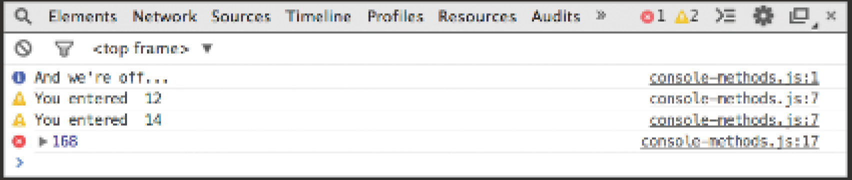
The `console.log()` method will write code to the console as it is processed.



```
And we're off...                          console-log.js:1
You entered  3                            console-log.js:6
You entered  4                            console-log.js:6
Clicked submit...                         console-log.js:11
Width 3                                   console-log.js:14
Height  4                                 console-log.js:17
12                                        console-log.js:20
>
```

These methods show messages like `log()` but have a slightly different style:

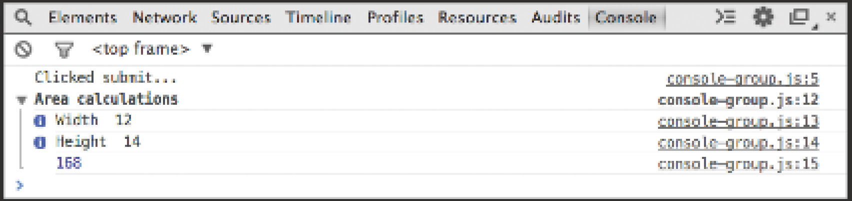
```
console.info()
console.warn()
console.error()
```


Elements Network Sources Timeline Profiles Resources Audits » 1 2
<top frame>

```
And we're off...                       console-methods.js:1
You entered  12                        console-methods.js:7
You entered  14                        console-methods.js:7
▶ 168                                  console-methods.js:17
>
```

You can group error messages with:

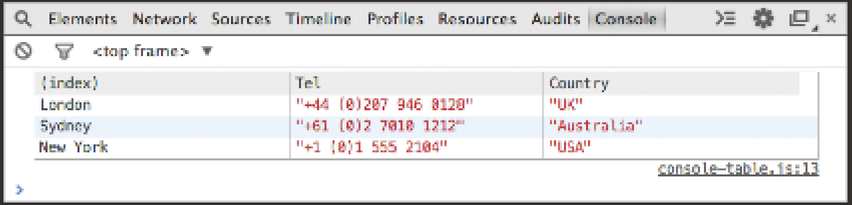
```
console.group('Areas');
  console.info('Width ', width);
  console.info('Height ', height);
  console.log(area);
console.groupEnd();
```


Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>

```
  Clicked submit...                    console-group.js:5
▼ Area calculations                    console-group.js:12
    Width  12                          console-group.js:13
    Height  14                         console-group.js:14
    168                                console-group.js:15
>
```

You can write arrays and object data into a table with:

console.table(*objectname*);

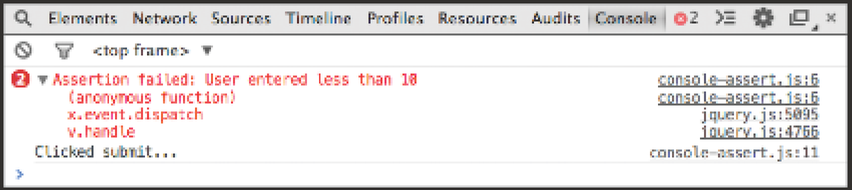


(index)	Tel	Country
London	"+44 (0)207 946 0120"	"UK"
Sydney	"+61 (0)2 7010 1212"	"Australia"
New York	"+1 (0)1 555 2104"	"USA"

```
                                                   console-table.js:13
```

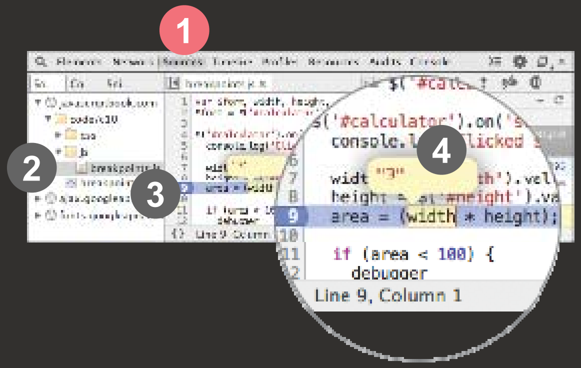
You can write on a condition with:

```
console.assert(this.value > 10,
                 'User entered less than 10');
```



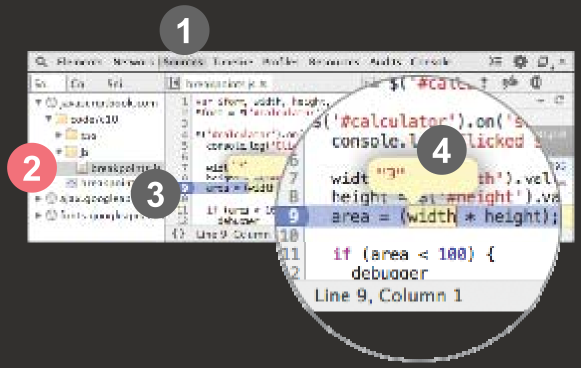
```
2 ▼ Assertion failed: User entered less than 10        console-assert.js:6
       (anonymous function)                            console-assert.js:6
       x.event.dispatch                                   jquery.js:5095
       v.handle                                           jquery.js:4766
    Clicked submit...                                  console-assert.js:11
```

Breakpoints let you pause the script on any line, allowing you to then check the values stored in variables.



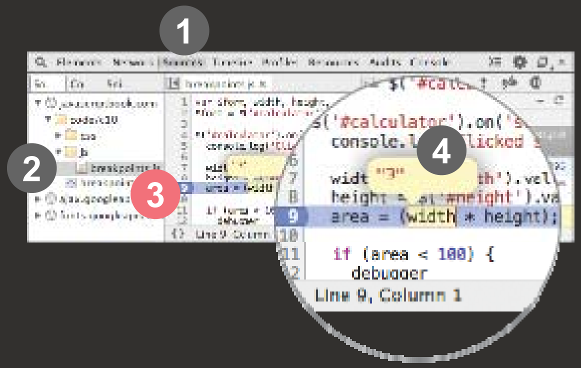
```
$('#calculator').on('s
console.l     licked
widt "3"      th').val
height = $('#height').va
area = (width * height);

if (area < 100) {
  debugger
Line 9, Column 1
```

1: **Sources** is selected



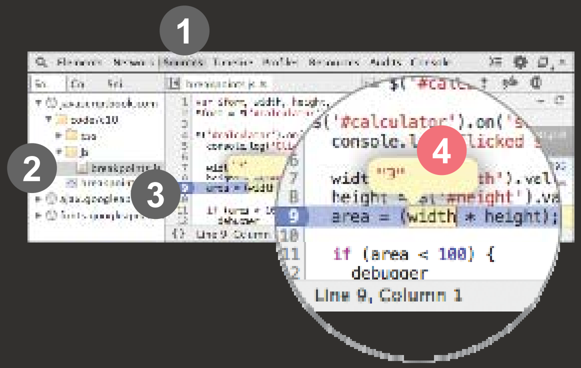
```
$('#calculator').on('s
console.l     licked
widt "3"      th').val
height = $('#height').va
area = (width * height);

if (area < 100) {
  debugger
Line 9, Column 1
```

2: **Script** is chosen



```
$('#calculator').on('s
console.l     licked
widt "3"      th').val
height = $('#height').va
area = (width * height);

if (area < 100) {
  debugger
Line 9, Column 1
```

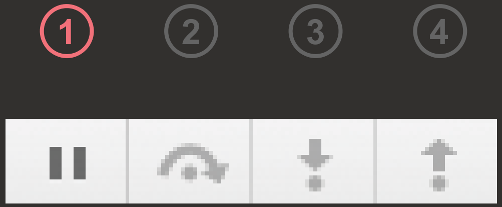
3: **Line number** is clicked on



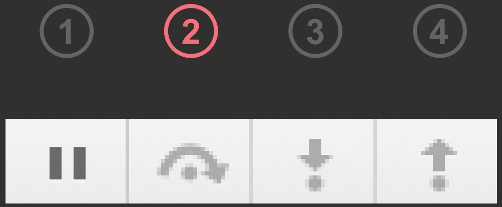
```
$('#calculator').on('s
console.l     licked
widt "3"      th').val
height = $('#height').va
area = (width * height);

if (area < 100) {
  debugger
Line 9, Column 1
```

4: **Variable** is hovered over

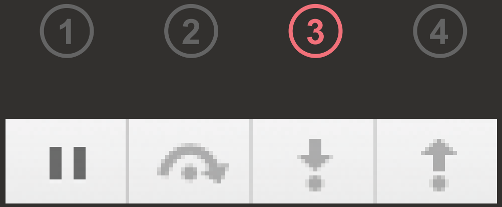
If you have several breakpoints, you can step through them one by one.



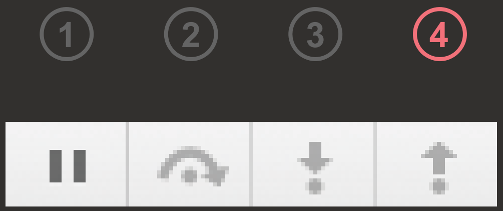
1: **Pause** button turns into a **play** button when a breakpoint is encountered



2: Go to next line of code and **step through** the lines one-by-one



3: **Step into** a function call



4: **Step out of** a function that you stepped into

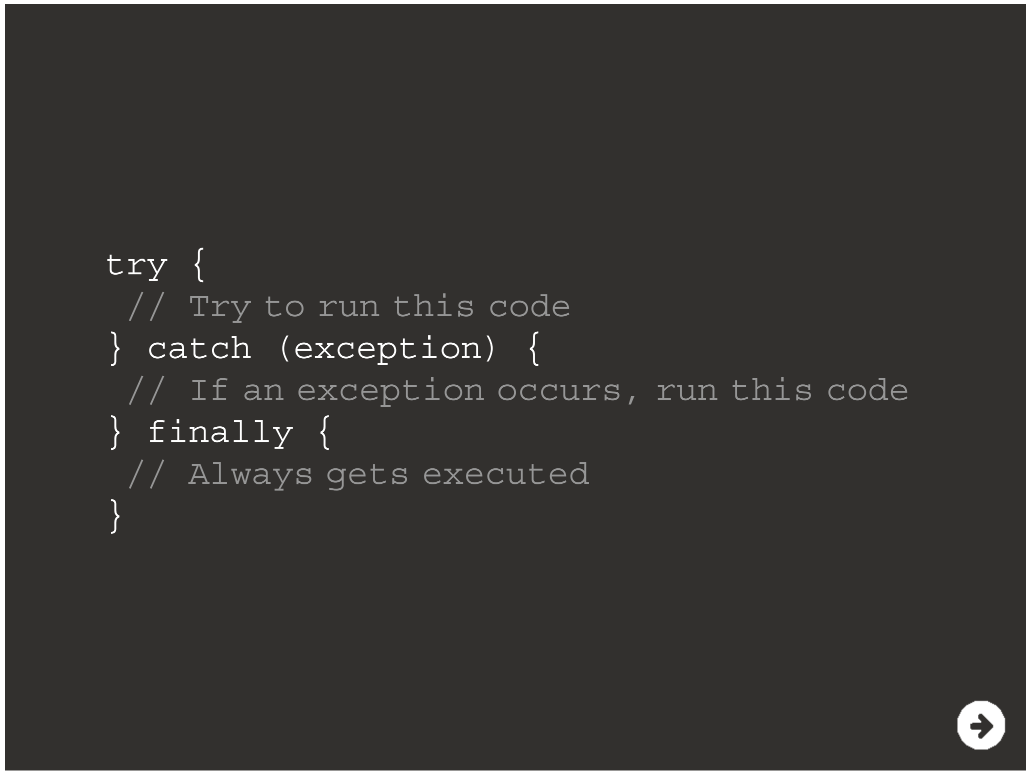
You can create a breakpoint with the `debugger` keyword:

```
if (area < 100) {
  debugger;
}
```

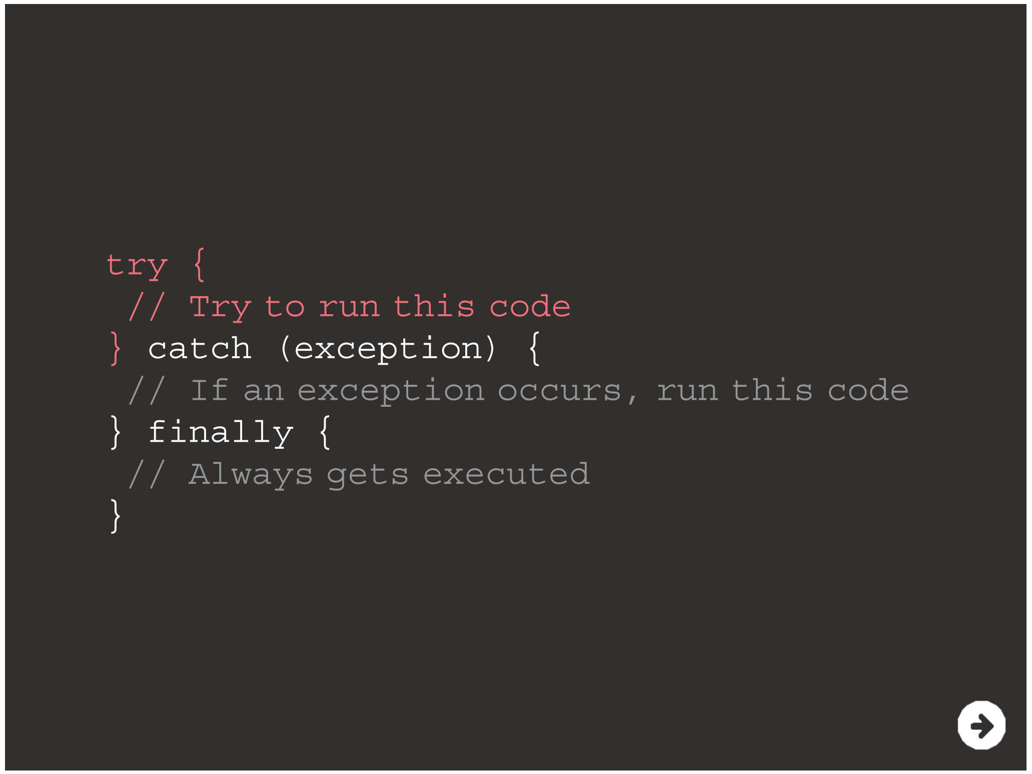
HANDLING EXCEPTIONS

If you know your code could fail, you can use `try`, `catch`, and `finally`.

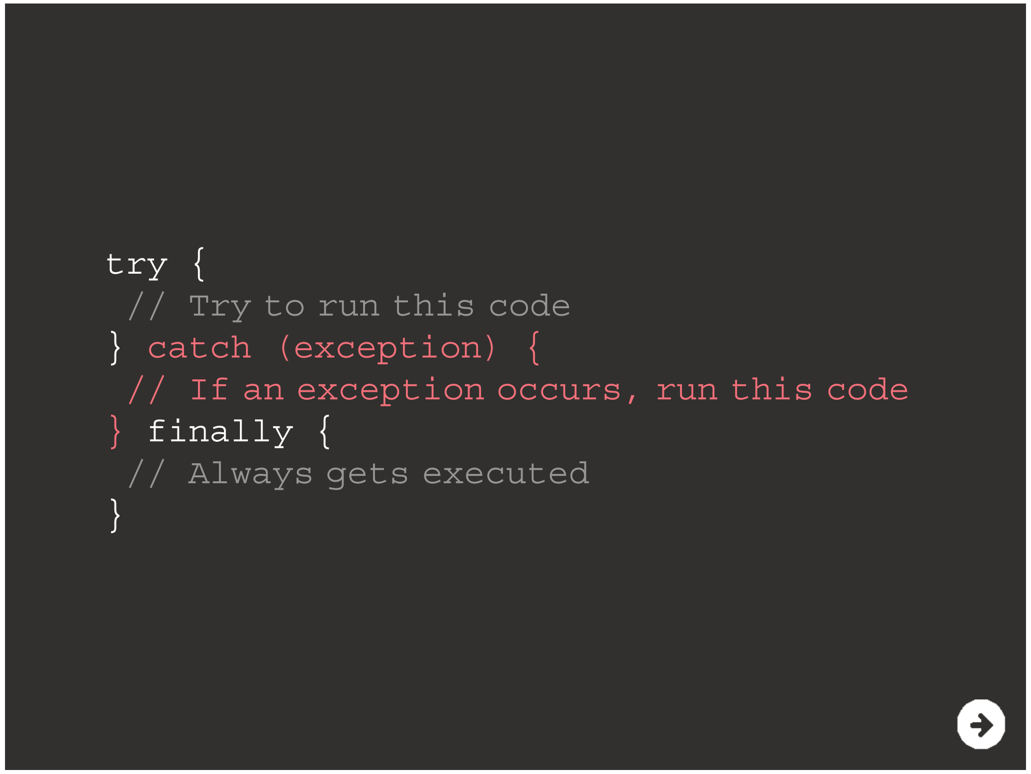
Each gets its own code block.



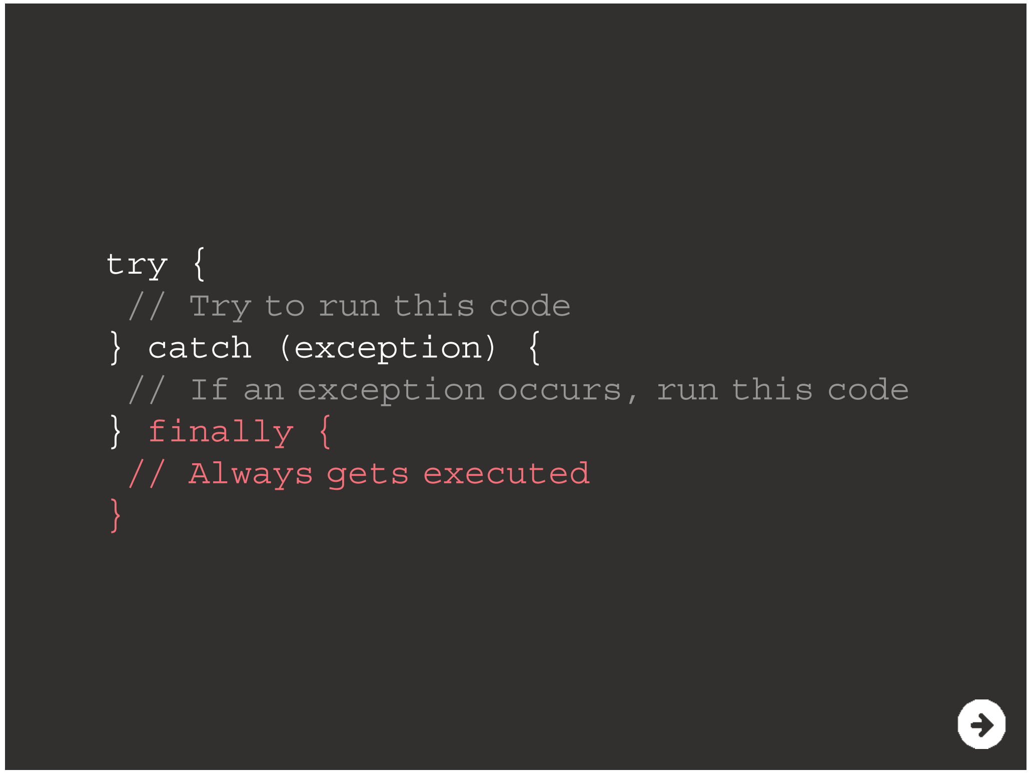
```
try {
 // Try to run this code
} catch (exception) {
 // If an exception occurs, run this code
} finally {
 // Always gets executed
}
```



```
try {
 // Try to run this code
} catch (exception) {
 // If an exception occurs, run this code
} finally {
 // Always gets executed
}
```



```
try {
 // Try to run this code
} catch (exception) {
 // If an exception occurs, run this code
} finally {
 // Always gets executed
}
```



```
try {
 // Try to run this code
} catch (exception) {
 // If an exception occurs, run this code
} finally {
 // Always gets executed
}
```