CS 50 Web Design

APCSP Module 2: Internet

Unit 3: JavaScript

LECTURE 8: PROGRAM STRUCTURES

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Program structures in JavaScript include the following: conditional statements, switch, for-loop, while-loop, do-while loop, and functions.
- Break levels for loops and functions.
- Random number generation
- A random number game design: Hangman
- Brief introduction for functional programming

Objectives

- By the end of the lesson, you will be familiar and know how the website works using JavaScripts.
  - Discuss the introduction to JavaScript and using conditional statements.
  - Understand the coding syntax using the break keywords.
  - Explain thoroughly the coding styles of different kinds of loops.

Conditionals and Logic

SECTION 1

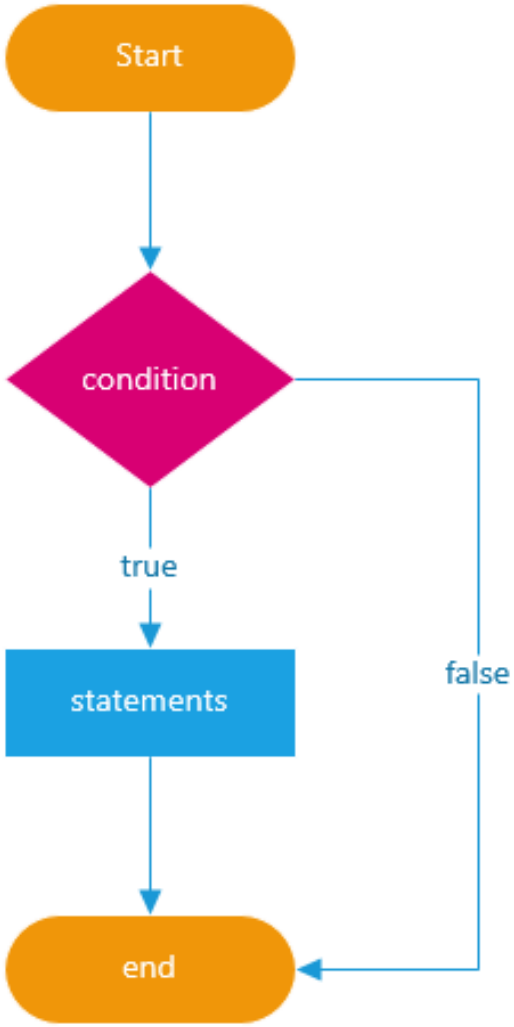
Conditionals and Logic

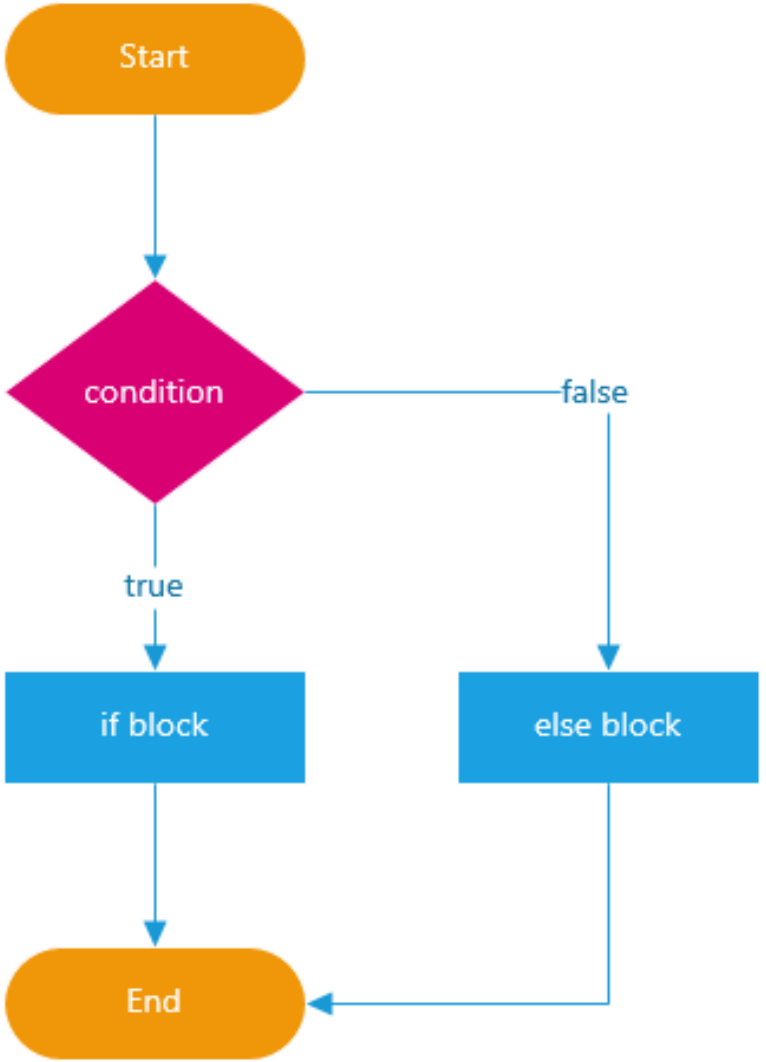
Conditionals and Logic are used to perform different actions based on different conditions.

- Very often when we write code, we wanted to perform different actions for different decisions.
- We can use Conditionals and Logic in our code to do this.

Conditionals and Logic

- In JavaScript, we have the following conditional statements:
  - Use **if** to specify a block of code to be executed, if a specified condition is true
  - Use **else** to specify a block of code to be executed, if the same condition is false.
  - Use **else if** to specify a new condition to test, if the first condition is false.
  - Use **switch** to specify many alternative blocks of code to be executed.


Start
condition
true
statements
false
end


Start
condition
false
true
if block
else block
End

The **if** Statement

- Use **if** to specify a block of code to be executed, if a specified condition is true.
- **Syntax:**

```
if (condition) {
block of code to be executed
if
the condition is true
}
```

- Note that **if** is in lower case. Uppercase letters will generate a JavaScript error.

The **if** Statement

- Example:

Make a “Good day” greeting if the hour is less than 18:00:

```
if (hour < 18) {
  greeting = "Good day";
}
```

- The result of greeting will be:

Good day

The **else** Statement

- Use **else** to specify a block of code to be executed, if the same condition is false.

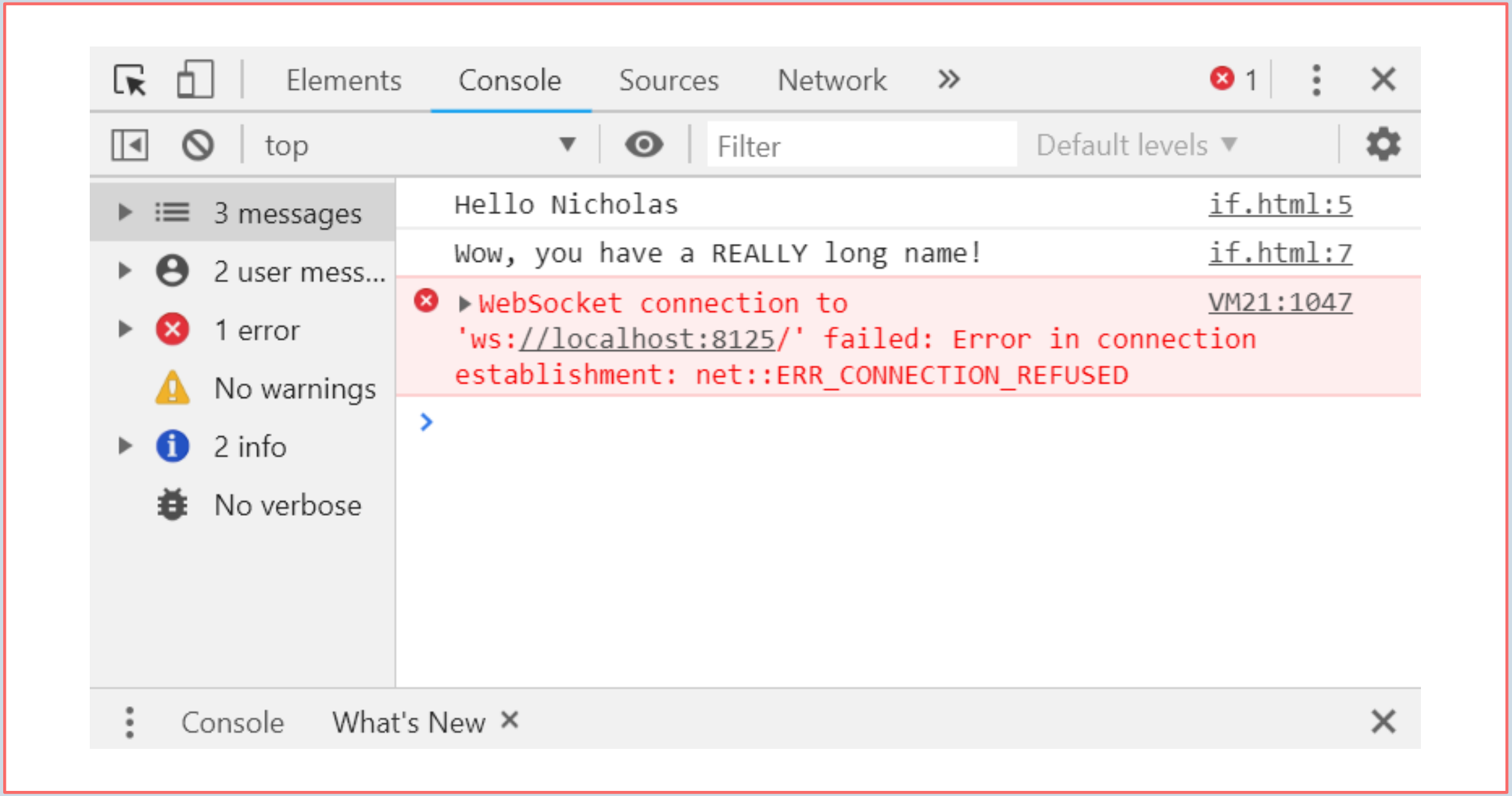
```
if (condition) {
 block of code to be executed
 if the condition is true

} else {
 block of code to be executed
 if
 the condition is false

}
```

```
<html>
    <body>
    <script>
        var name = "Nicholas";
        console.log("Hello " + name);
        if (name.length > 7) {
            console.log("Wow, you have a REALLY long name!");
        }
    </script>
    </body>
</html>
```

Demo Program: if.html


Elements
Console
Sources
Network
1
top
Filter
Default levels
3 messages
2 user mess...
1 error
No warnings
2 info
No verbose
Hello Nicholas
if.html:5
Wow, you have a REALLY long name!
if.html:7
WebSocket connection to 'ws://localhost:8125/' failed: Error in connection establishment: net::ERR_CONNECTION_REFUSED
VM21:1047
Console
What's New

The **else** Statement

- **Example:**
- If the hour is less than 18, create a “Good day" greeting, otherwise "Good evening":

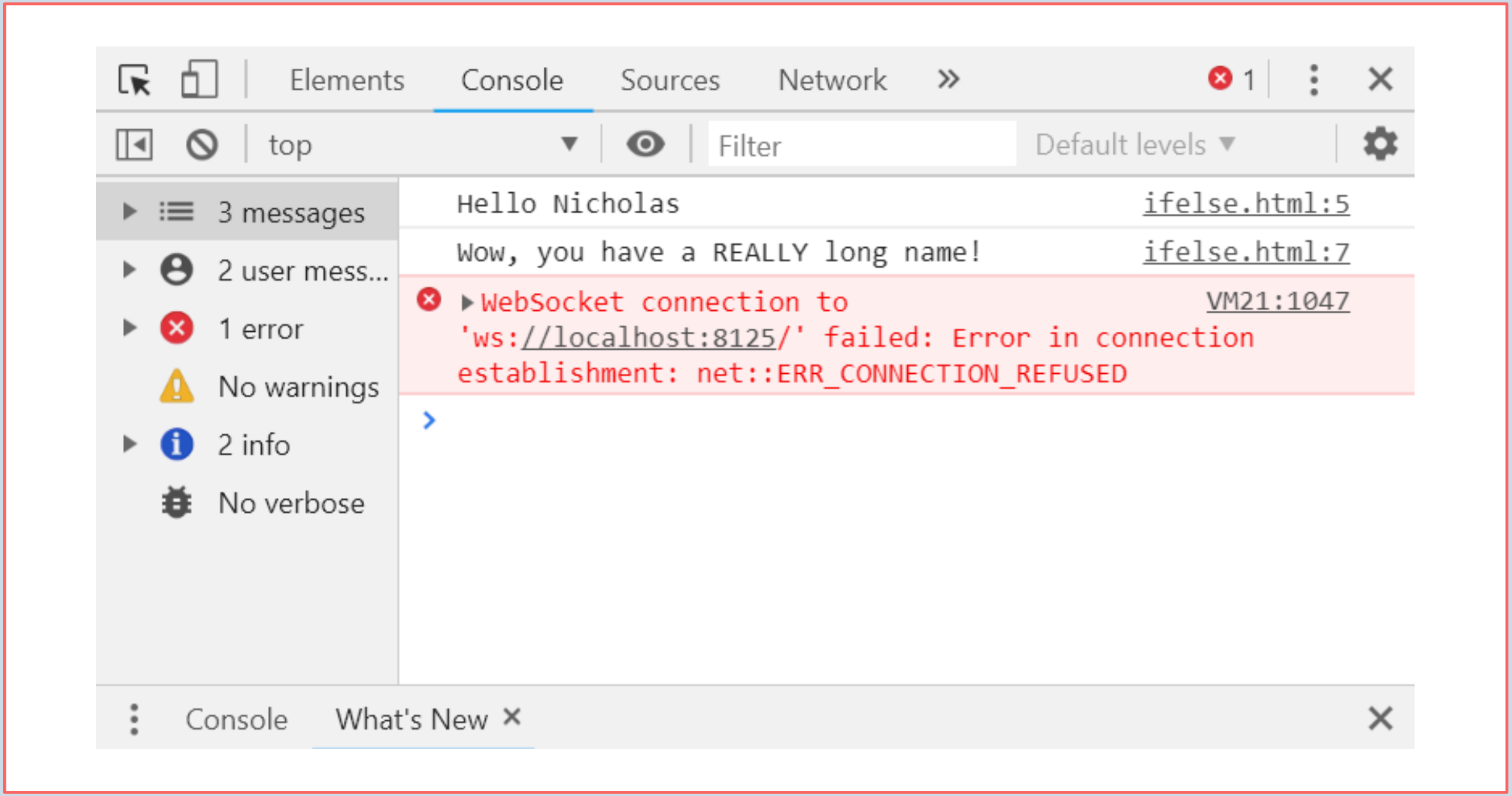
```
if (hour < 18) {
greeting = "Good day";
} else {
greeting = "Good evening";
}
```

- The result of greeting will be:

Good day

Demo Program: ifelse.html

```
1  <html>
2      <body>
3      <script>
4          var name = "Nicholas";
5          console.log("Hello " + name);
6          if (name.length > 7) {
7              console.log("Wow, you have a REALLY long name!");
8          }
9          else {
10             console.log("Your name isn't very long.");
11         }
12     </script>
13     </body>
14 </html>
```


Elements
Console
Sources
Network
1
top
Filter
Default levels
3 messages
2 user mess...
1 error
No warnings
2 info
No verbose
Hello Nicholas
ifelse.html:5
Wow, you have a REALLY long name!
ifelse.html:7
WebSocket connection to 'ws://localhost:8125/' failed: Error in connection establishment: net::ERR_CONNECTION_REFUSED
VM21:1047
Console
What's New

The **else if** Statement

- Use **else if** to specify a new condition to test, if the first condition is false.

```
if (condition1) {
  block of code to be executed if
  condition1 is true
} else if (condition2) {
  block of code to be executed if
  the condition1 is false and
  condition2 is true
} else {
  block of code to be executed if
  the condition1 is false and
  condition2 is false
}
```

The **else if** Statement

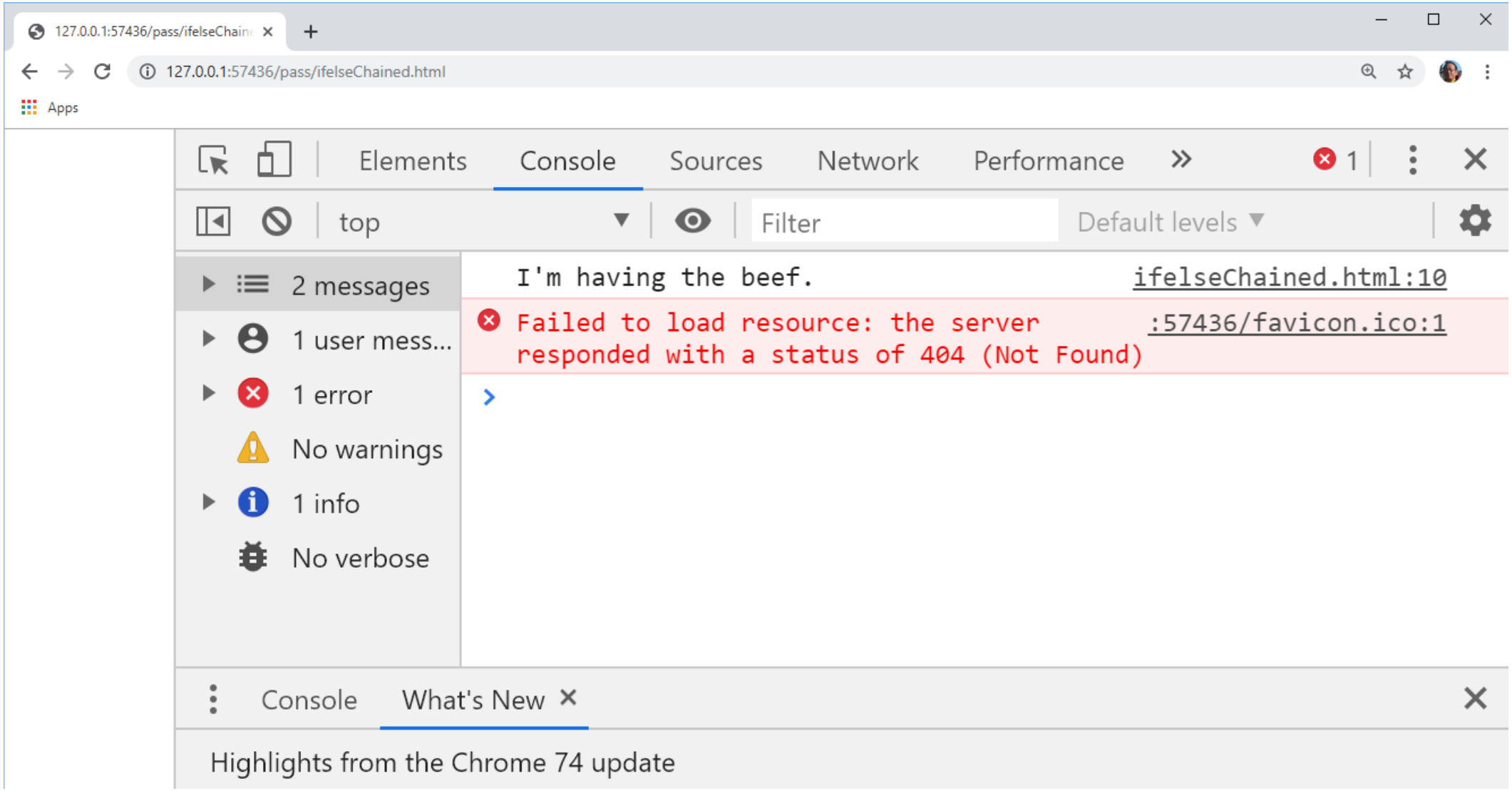
- **Example:**
- If time is less than 10:00, create a “Good morning” greeting, if not, but time is less than 20:00, create a “Good day” greeting, otherwise a "Good evening":

```
if (time < 10) {
greeting = "Good morning";
} else if (time < 20) {
greeting = "Good day";
} else {
greeting = "Good evening";
}
```

Demo Program: ifelseChained.html

- Nested if-else-if-else statements.
- Maybe replaced by switch-statement in some situations.

```
<html>
    <body>
    <script>
        var lemonChicken = false;
        var beefWithBlackBean = true;
        var sweetAndSourPork = true;
        if (lemonChicken) {
            console.log("Great! I'm having lemon chicken!");
        } else if (beefWithBlackBean) {
            console.log("I'm having the beef.");
        } else if (sweetAndSourPork) {
            console.log("OK, I'll have the pork.");
        } else {
            console.log("Well, I guess I'll have rice then.");
        }
    </script>
    </body>
</html>
```


127.0.0.1:57436/pass/ifelseChain
127.0.0.1:57436/pass/ifelseChained.html
Apps
Elements
Console
Sources
Network
Performance
1
top
Filter
Default levels
2 messages
1 user mess...
1 error
No warnings
1 info
No verbose
I'm having the beef.
ifelseChained.html:10
Failed to load resource: the server responded with a status of 404 (Not Found)
:57436/favicon.ico:1
Console
What's New
Highlights from the Chrome 74 update

Switch

SECTION 2

The **switch** Statement

- Use **switch** to specify many alternative blocks of code to be executed.
- **Syntax:**

```
switch(expression) {
   case n:  /* code block */
   break;
   case n:  /* code block */
   break;
   default: /* default code
   block */
}
```

The **switch** Statement

- How switch works?
- The switch expression is evaluated once.
- The value of the expression is compared with the value of each case.
- If there is a match, the associated block of code is executed.

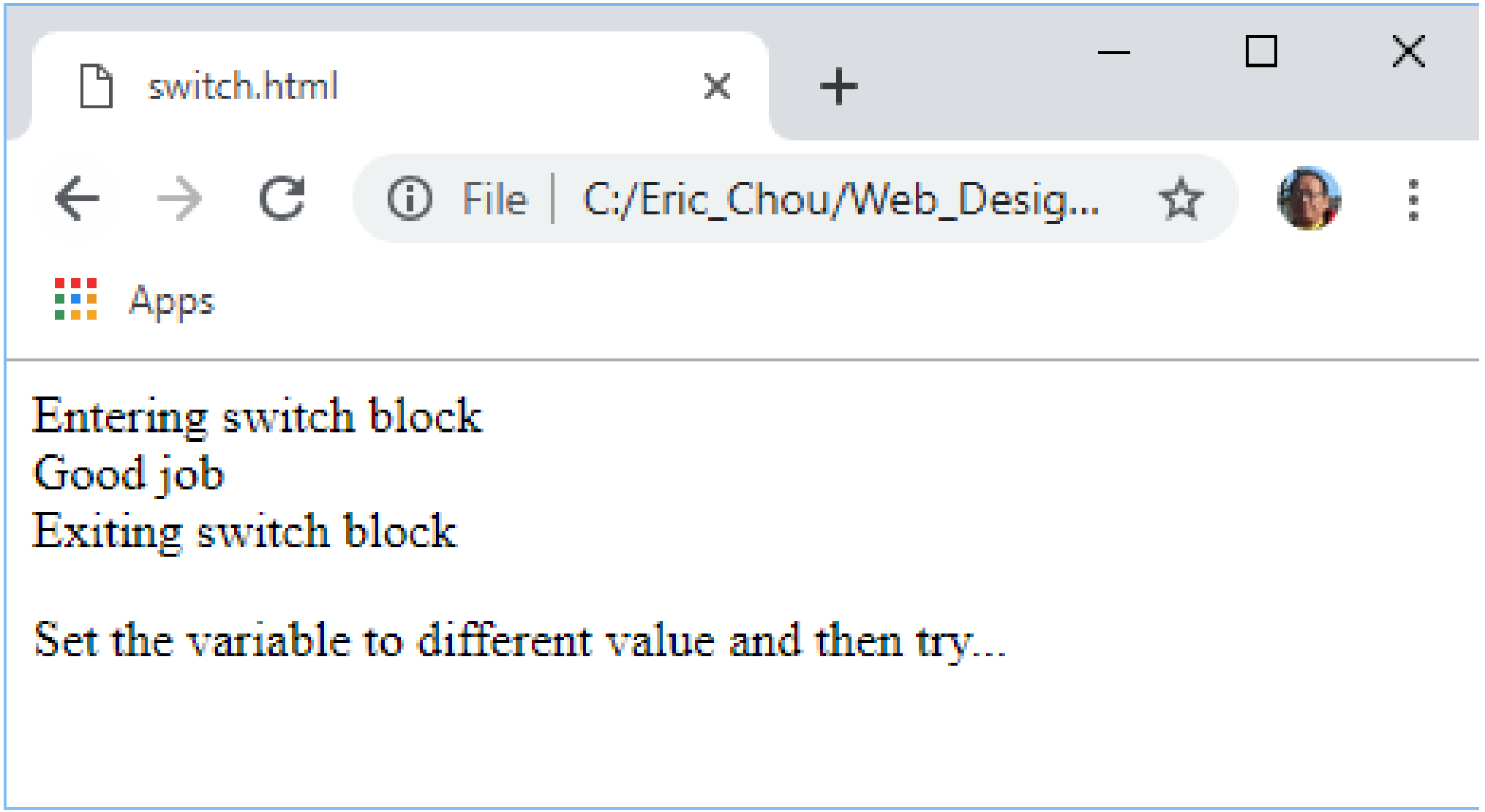
The **switch** Statement

- Example:
  - The getDay() method returns the weekday as a number between 0 and 6. (Sunday=0, Monday=1, Tuesday=2 ..)
  - Use the weekday number to calculate weekday name.

Switch
Demo Program: switch.html

- 6-way switch.
- String as switch variable
- Output to document.

```
<html>
   <body>
      <script type = "text/javascript">
            var grade = 'A';
            document.write("Entering switch block<br />");
            switch (grade) {
               case 'A': document.write("Good job<br />");    break;
               case 'B': document.write("Pretty good<br />"); break;
               case 'C': document.write("Passed<br />");      break;
               case 'D': document.write("Not so good<br />"); break;
               case 'F': document.write("Failed<br />");      break;
               default:  document.write("Unknown grade<br />")
            }
            document.write("Exiting switch block");
      </script>
      <p>Set the variable to different value and then try...</p>
   </body>
</html>
```


switch.html
File | C:/Eric_Chou/Web_Desig...
Apps
Entering switch block
Good job
Exiting switch block
Set the variable to different value and then try...

Conditional Operators

SECTION 3

JavaScript if else shortcut: conditional operator

- JavaScript provides a conditional operator that can be used as a shortcut of the if statement. The following illustrates the syntax of the conditional operator.

```
condition ? expression_1 : expression_2
```

- Like the if statement, the condition is an expression that evaluates to true or false. If the condition evaluates to true, the operator returns the value of the expression_1; otherwise, it returns the value of the expression_2.

JavaScript if else shortcut: conditional operator

- For example, to display a different label for the login button based on the value of the **`isLoggedIn`** variable, you could use the conditional operator as follows:

```
isLoggedIn ? "Logout" : "Login";
```

- You can also assign a variable depending on the result of the ternary operator.

```
// only register if the age is greater than 18
var allowRegister = age > 18 ? true : false;
```

JavaScript if else shortcut: conditional operator

- If you want to do more than a single operation per case, you need to separate operation using a comma (,) as the following example:

```
age > 18 ? (
    alert("OK, you can register."),
    redirectTo("register.html");
) : (
    stop = true,
    alert("Sorry, you are too young!")
);
```

- In this tutorial, you have learned how to use the JavaScript if else statement to execute a statement when a condition evaluates to true and executes another statement when the condition evaluates to false.

Loops

SECTION 4

Different Kinds of Loops

- JavaScript supports different kinds of loops:
  - **for** – loops through a block of code a number of times.
  - **for/in** – loops through the properties of an object.
  - **while** – loops through a block of code while a specified condition is true.
  - **do/while** - also loops through a block of code while specified condition is true.

JavaScript **for** loop

- Loops can execute a block of code a number of times.
- Loops are handy, if you want to run the same code over and over again, each time with a different value. Often this is the case when working with arrays:

```
text += cars[0] + "<br>";
text += cars[1] + "<br>";
text += cars[2] + "<br>";
text += cars[3] + "<br>";
text += cars[4] + "<br>";
text += cars[5] + "<br>";
```

JavaScript **for** loop

- The for loop is often the tool you will use when you want to create a loop.
- The for loop has the following syntax:

```
for(statement1; statement2; statement3){
    code block to be executed
}
```

JavaScript **for** loop

- Often this is the case when working with arrays:
- We can write:

```
text += cars[0] + "<br>";
text += cars[1] + "<br>";
text += cars[2] + "<br>";
text += cars[3] + "<br>";
text += cars[4] + "<br>";
text += cars[5] + "<br>";
```

```
for (i = 0; i < cars.length; i++){
    text += cars[i] + "<br>";
}
```

JavaScript **for** loop

- Statement1 is executed before the loop (the code block) starts.
- Statement2 defines the condition running the loop (the code block)
- Statement3 is executed each time after the loop (the code block) has been executed.

- **Example:**

```
for (i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

JavaScript **for/in** loop (for-each-attribute)

- The JavaScript for/in statement loops through the properties of an object:

```
var person = {fname:"John",
              lname:"Doe", age:25};

var text = "";

var x;

for (x in person) {
  text += person[x];
}
```

JavaScript **while** loop

- The while loop loops through a block of code as long as a specified condition is true.
- **Syntax:**

```
while (condition) {
code block to be executed
}
```

JavaScript **while** loop

- **Example:**
- The code in the loop will run over and over again, as long as variable (i) is less than 10.

```
while (i < 10) {
  text += "The number is " +
  i;
  i++;
}
```

JavaScript **do/while** loop

- The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.
- **Syntax:**

```
do {
code block to be executed
}
while (condition);
```

JavaScript **do/while** loop

- **Example:**
- The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

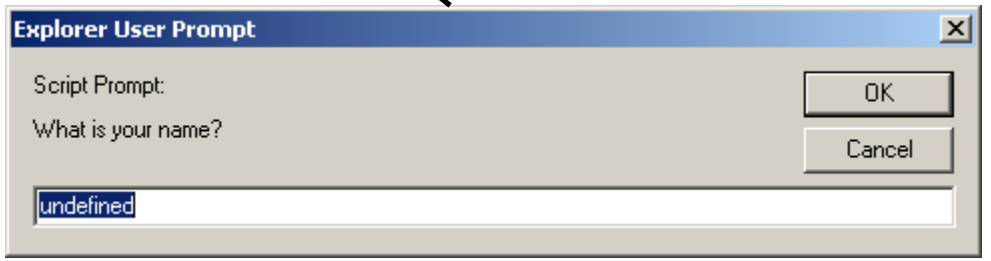
```
do {
 text += "The number is "
         + i;
 i++;
}
while (i < 10);
```

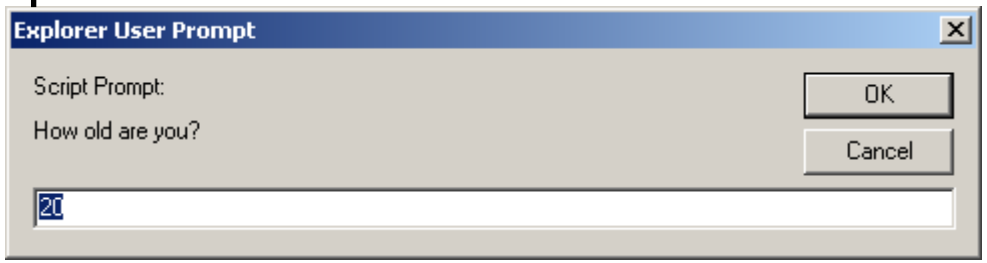
Input/Output with Browsers

SECTION 5

alert(), confirm(), and prompt()

```
<script type="text/javascript">
alert("This is an Alert method");
confirm("Are you OK?");
prompt("What is your name?");
prompt("How old are you?","20");
</script>
```





`alert()` and `confirm()`

```
alert("Text to be displayed");
```

- Display a message in a dialog box.
- The dialog box will block the browser.

```
var answer = confirm("Are you sure?");
```

- Display a message in a dialog box with two buttons: "OK" or "Cancel".
- `confirm()` returns **`true`** if the user click "OK". Otherwise it returns **`false`**.

prompt()

```
prompt("What is your student id number?");
prompt("What is your name?”, "No name");
```

- Display a message and allow the user to enter a value
- The second argument is the "default value" to be displayed in the input textfield.
- Without the default value, "undefined" is shown in the input textfield.
- If the user click the "OK" button, **`prompt()`** returns the value in the input textfield as a string.
- If the user click the "Cancel" button, **`prompt()`** returns null.

Break Levels

SECTION 6

break

The **break** Keyword

- When the JavaScript code interpreter reaches a **break** keyword, it breaks out of the switch block.
  - This will stop the execution of more code and case testing inside the block.
  - Note: When a match is found, and the job is done, it is time for a break. There is no need for more testing.

Label statement

- Before discussing the break statement, let's talk about the label statement first.
- In JavaScript, you can label a statement for later use. The following illustrates the syntax of the label statement.

```
label: statement;
```

- The label can by any valid identifier.
- The following example labels the loop using the outer label.

```
outer: for (var i = 0; i < 5; i++) {
    console.log(i);
}
```

- You can reference to the label by using the break or continue statement. Typically, you use the label with nested loop such as for, do-while, and while loop.

JavaScript break statement

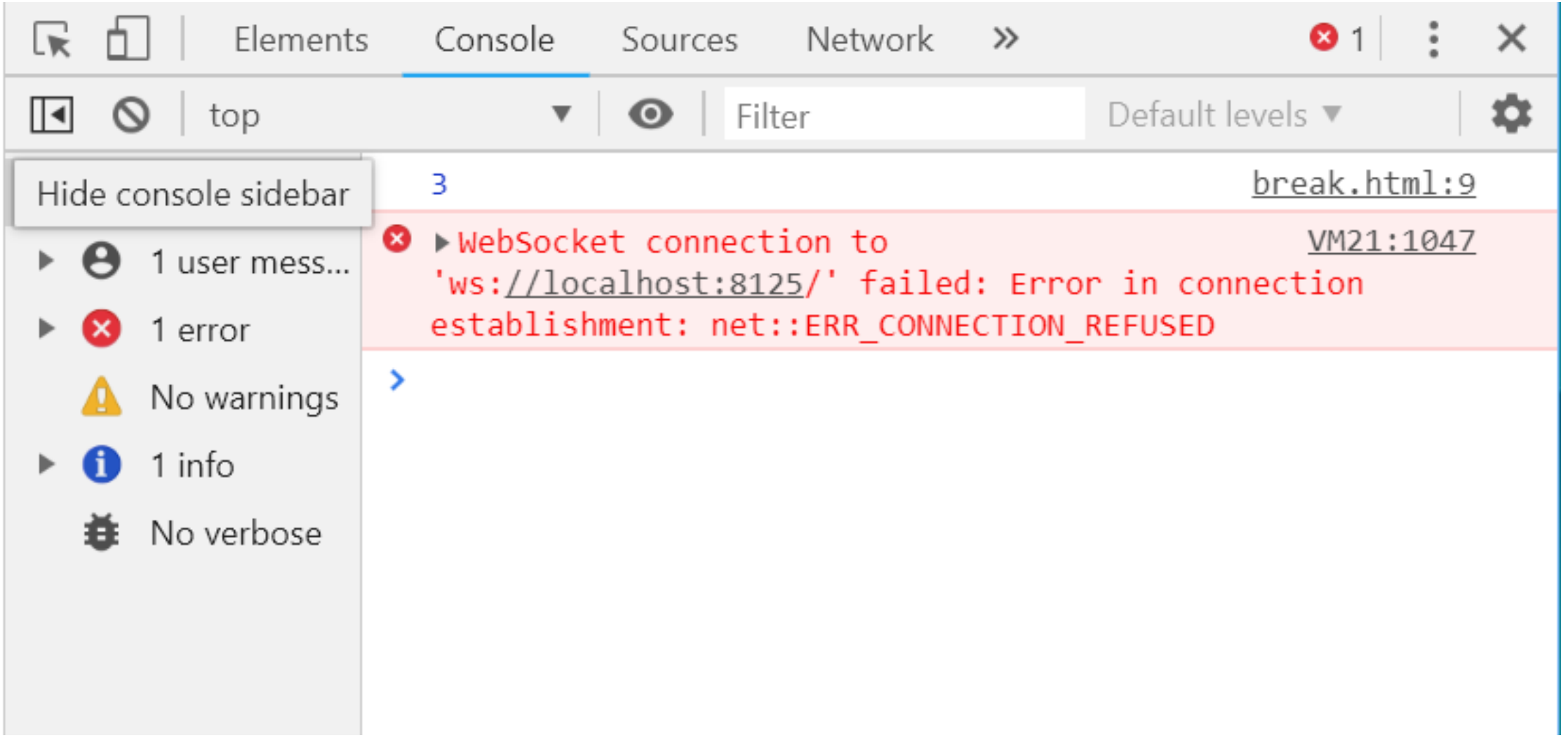
Demo Program: break.html

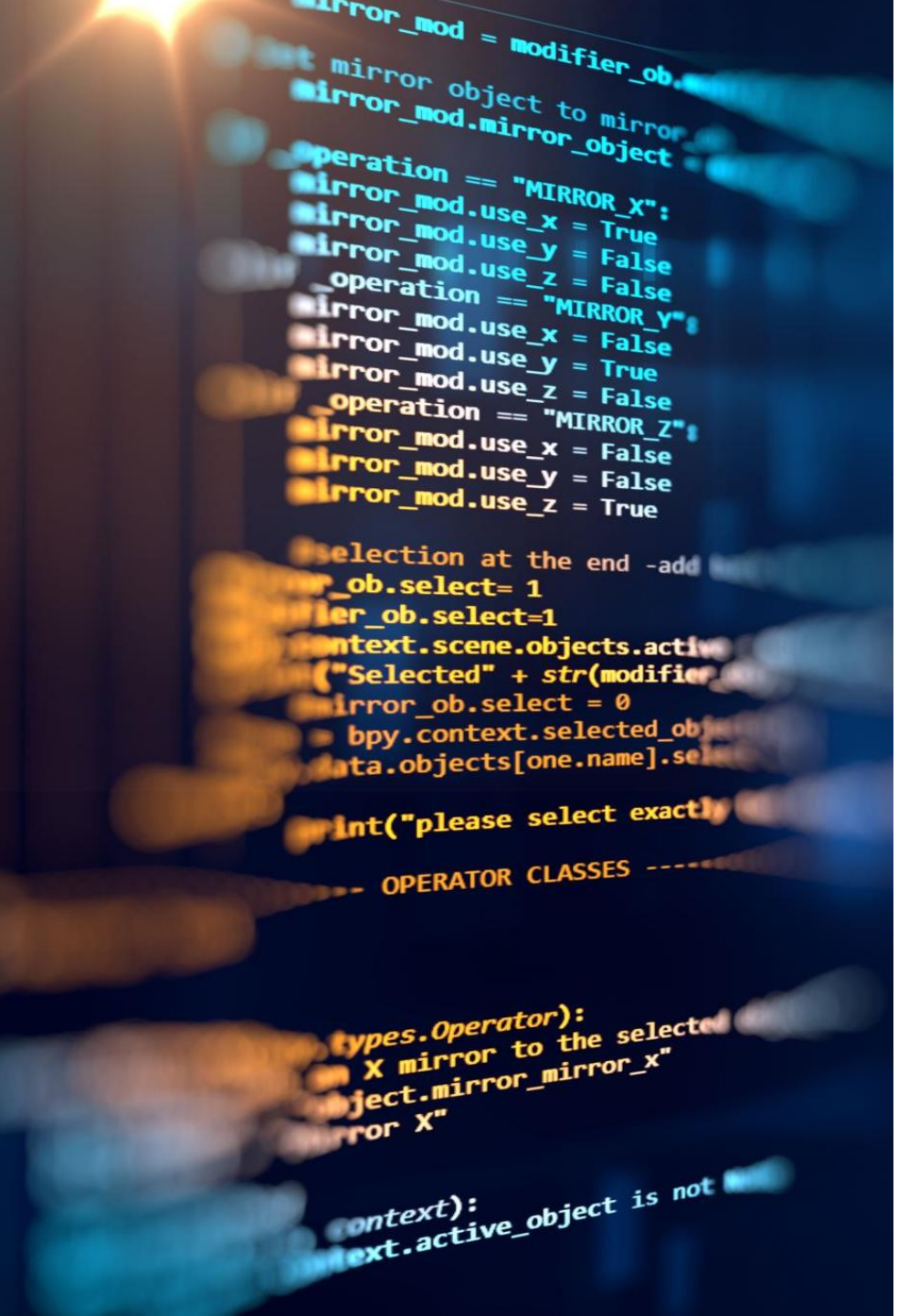
- The break statement gives you a fine-grained control over the execution fo the code in a loop. The break statement terminates the loop immediately and passes control over the next statement after the loop.
- Here's an example.

```
1 for (var i = 1; i < 10; i++) {
2     if (i % 3 == 0) {
3         break;
4     }
5 }
6 console.log(i); // 3
```

JavaScript break statement

- In this example, the `for` loop increments the variable `i` from 1 to 10. In the body of the loop, the if statement checks if `i` is evenly divisible by 3. If so, the break statement is executed and the loop is terminated.
- The control is passed to the next statement outside the loop that outputs the variable i to the web console.
- Besides controlling the loop, you also use the break statement to terminate a case branch in the switch block.


Elements
Console
Sources
Network
1
top
Filter
Default levels
Hide console sidebar
1 user mess...
1 error
No warnings
1 info
No verbose
3
break.html:9
WebSocket connection to 'ws://localhost:8125/' failed: Error in connection establishment: net::ERR_CONNECTION_REFUSED
VM21:1047



Using **break** statement to exit nested loop

- As mentioned earlier, you use the break statement to terminate a label statement and transfer control to the next statement following the terminated statement. The syntax is as follows:

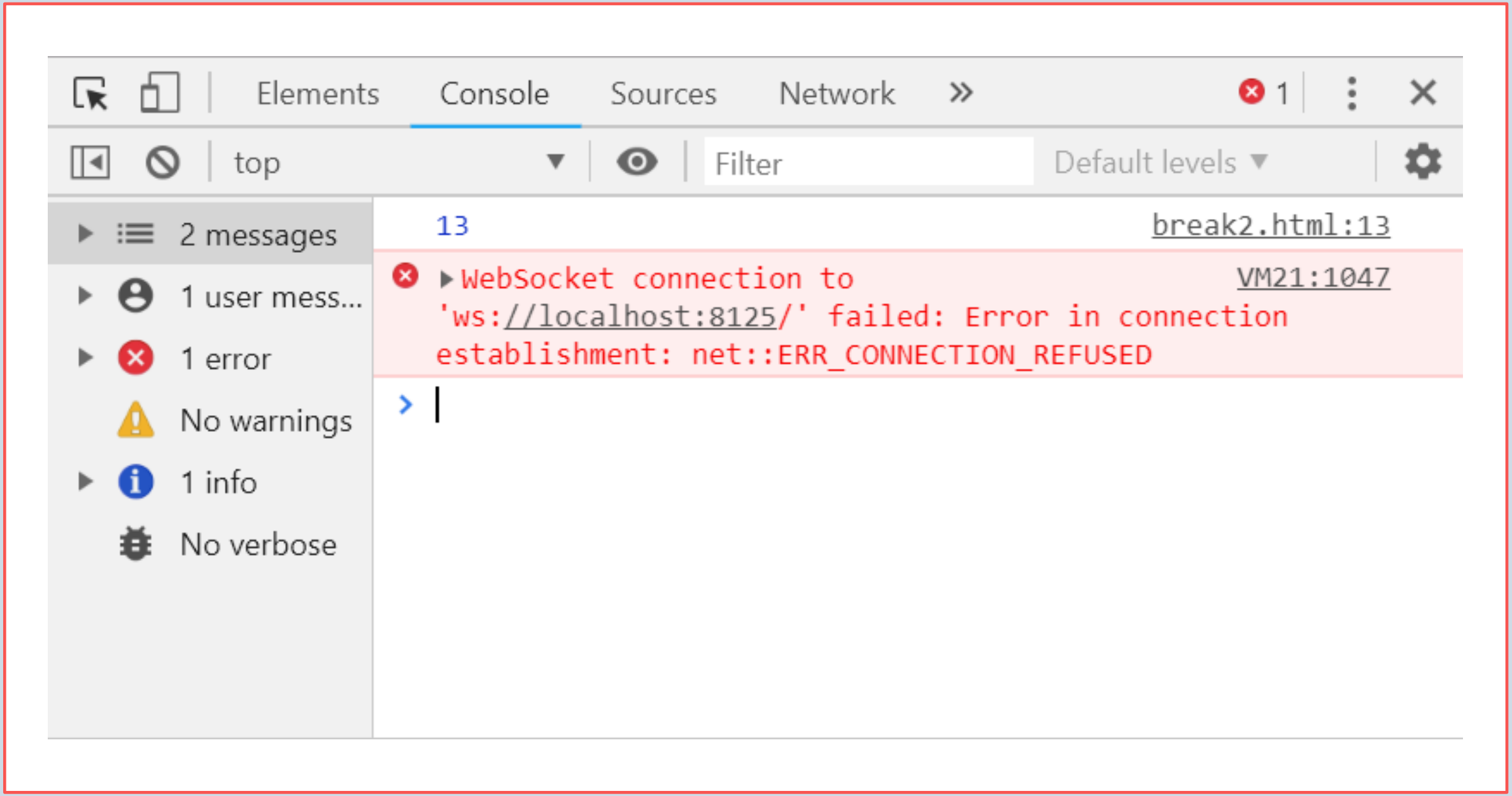
```
break label;
```

Using **break** statement to exit nested loop

Demo Program: break2.html

- The `break` statement is typically used to exit the nested loop. See the following example.

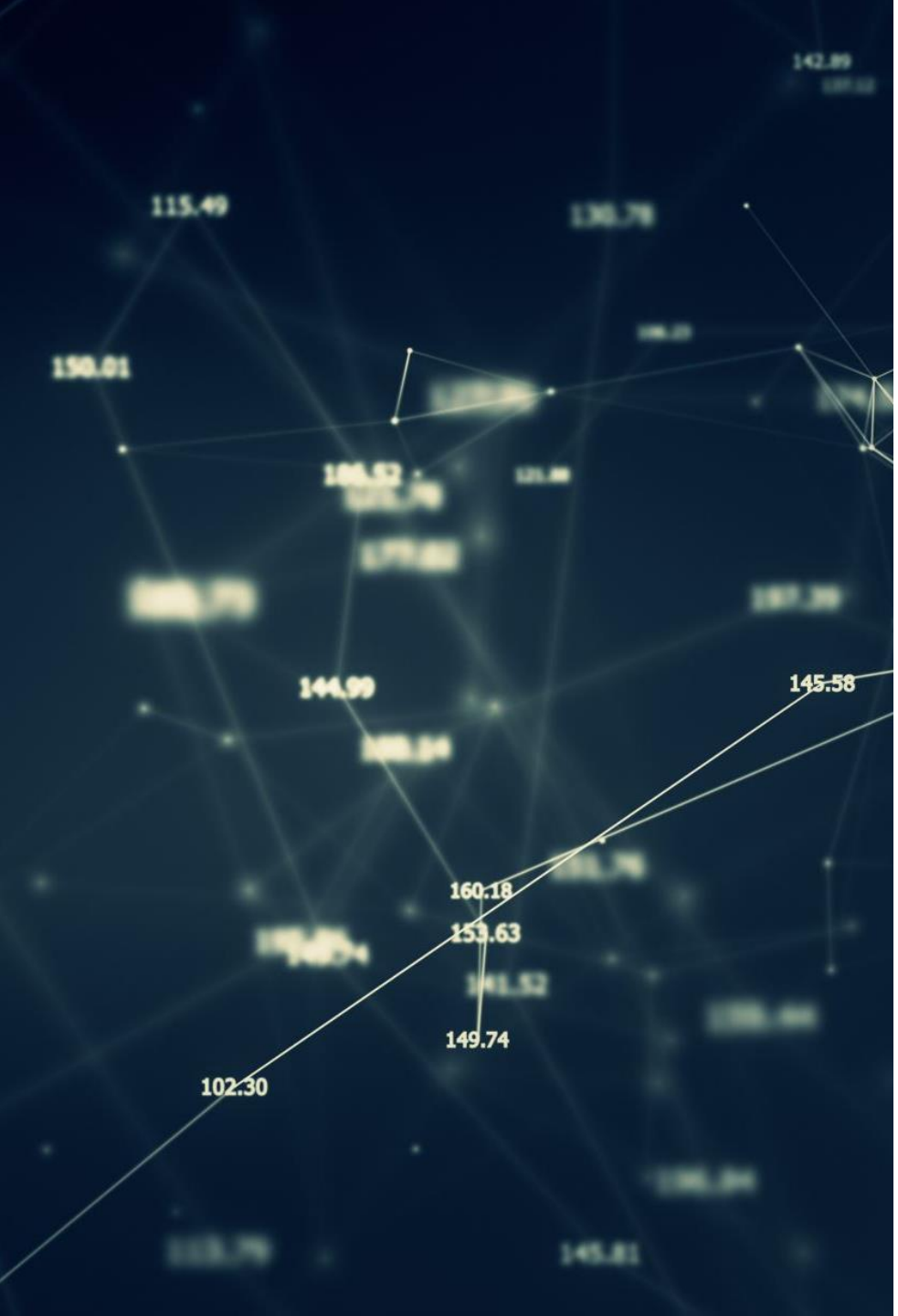
```
1  var iterations = 0;
2  top: for (var i = 0; i < 5; i++) {
3      for (var j = 0; j < 5; j++) {
4          iterations++;
5          if (i === 2 && j === 2) {
6              break top;
7          }
8      }
9  }
10 console.log(iterations); // 13
```


Elements
Console
Sources
Network
1
top
Filter
Default levels
2 messages
1 user mess...
1 error
No warnings
1 info
No verbose
13
break2.html:13
WebSocket connection to 'ws://localhost:8125/' failed: Error in connection establishment: net::ERR_CONNECTION_REFUSED
VM21:1047

continue

JavaScript **continue**: Skips the Current Iteration of a Loop

- **Summary:** in this tutorial, you will learn how to use the JavaScript continue statement to skip the current iteration of a loop.
- The **continue** statement skips the current iteration of a loop and goes to the next one. Because of this, the continue statement must appear in the body of a loop or you will get an error.
- Similar to the **break** statement, the continue statement has two forms: labeled and unlabeled. For more information on the label statement, see the break statement tutorial.



Using unlabeled JavaScript **continue** statement

- The unlabeled continue statement skips the current iteration of a **for**, **do-while**, or **while loop**. The continue statement skips the rest of the code to the end of the innermost body of a loop and evaluates the expression that controls the loop.

Using unlabeled JavaScript **continue** statement

- In a for loop, the continue skips all the statements underneath it and pass the execution of the code to the update expression, in this case, it is **i++;**

```
for (var i = 0; i < count; i++) {
    if (condition)
        continue; // Jumps to expression: i++
    // more statement here
}
```

Using unlabeled JavaScript **continue** statement

- In a while or do-while loop, it jumps back to the expression that controls the loop.

```
while (expression)  // continue jumps here
{
    if (condition) {
        continue; // Jumps to expression
    }
    // more statements here
    // ...
}
```

```
do{
    if (condition) {
        continue; // Jumps to expression
    }
    // more statements here
    // ...
}while(expression); // continue jumps here
```

Using unlabeled JavaScript **continue** statement

Demo Program: continue.html (3)

```
1  var s = 'This is a JavaScript continue statement demo.';
2  var counter = 0;
3  for (var i = 0; i < s.length; i++) {
4      if (s.charAt(i) != 's') {
5          continue;
6      }
7      //
8      counter++;
9  }
10 console.log('The number of s found in the string is ' + counter);
```

Using JavaScript **continue** with a label

The `continue` statement can include an optional label as follows:

```
continue label;
```

The `label` can be any valid identifier.

See the following example.

```
// continue with a label
outer: for (var i = 1; i <= 3; i++) {
    for (var j = 1; j <= 3; j++) {
        if ((i == 2) && (j == 2)) {
            console.log('continue to outer');
            continue outer;
        }
        console.log("[i:" + i + ",j:" + j + "]");
    }
}
```

pass in JavaScript

```
<html>
    <body>
    <script>
    var i=0;
        document.write("<h1>Demo of Pass: </h1>");
        // single ; can be used for pass
        if (i==0) {
            ;
        }
        document.write("Passed...");
    </script>
    </body>
</html>
```

pass in JavaScript

Demo Program: pass.html

USE SINGLE ; SYMBOL AS PASS STATEMENT;

return

Hangman Game

SECTION 7

Hangman Game

- In this chapter we'll build a **Hangman** game! We'll learn how to use dialogs to make the game interactive and take input from someone playing the game.
- Hangman is a word-guessing game. One player picks a secret word, and the other player tries to guess it.
- For example, if the word were TEACHER, the first player would write:

_ _ _ _ _ _ _

Hangman Game

- The guessing player tries to guess the letters in the word. Each time they guess a letter correctly, the first player fills in the blanks for each occurrence of that letter.
- For example, if the guessing player guessed the letter E, the first player would fill in the Es in the word TEACHER like so:

Hangman Game

- When the guessing player guesses a letter that isn't in the word, they lose a point and the first player draws part of a stick-man for each wrong guess. If the first player completes the stickman before the guessing player guesses the word, the guessing player loses.
- In our version of Hangman, the **JavaScript** program will choose the word and the human player will guess letters. We won't be drawing the stickman, because we haven't yet learned how to draw in JavaScript (we'll learn how to do that in Chapter 13).

Interacting with a Player

SECTION 8

Interacting with a Player

- To create this game, we have to have some way for the guessing player (human) to enter their choices.
- One way is to open a pop-up window (which JavaScript calls a prompt) that the player can type into.

Creating a Prompt

- First, let's create a new HTML document. Using **File ▸ Save As**, save your *page.html* file from Chapter 5 as *prompt.html*. To create a prompt, enter this code between the <script> tags of *prompt.html* and refresh the browser:

```
var name = prompt("What's your name?");

console.log("Hello " + name);
```

- Here we create a new variable, called name, and assign to it the value returned from calling prompt("What's your name?"). When prompt is called, a small window (or *dialog*) is opened, which should look like Figure 7-1.

Creating a Prompt

Demo Program: prompt.html

Figure 7-1. A prompt dialog

Creating a Prompt

- Calling `prompt("What's your name?")` pops up a window with the text "What's your name?" along with a text box for input. At the bottom of the dialog are two buttons, Cancel and OK. In Chrome, the dialog has the heading *JavaScript*, to inform you that JavaScript opened the prompt.
- When you enter text in the box and click OK, that text becomes the value that is returned by `prompt`. For example, if I were to enter my name into the text box and click OK, JavaScript would print this in the console:

```
Hello Nick
```

- Because I entered *Nick* in the text box and clicked OK, the string `"Nick"` is saved in the variable `name` and `console.log` prints `"Hello " + "Nick"`, which gives us `"Hello Nick"`.

Using confirm to Ask a Yes or No Question

- The `confirm` function is a way to take user input without a text box by asking for a yes or no (Boolean) answer. For example, here we use `confirm` to ask the user if they like cats (see Figure 7-2).
- If so, the variable `likesCats` is set to true, and we respond with "You're a cool cat!"
- If they don't like cats, `likesCats` is set to false, so we respond with "Yeah, that's fine. You're still cool!"

Using confirm to Ask a Yes or No Question

Demo Program: confirm.html

```
var likesCats = confirm("Do you like cats?");
if (likesCats) {
  console.log("You're a cool cat!");
} else {
  console.log("Yeah, that's fine. You're still cool!");
}
```

Figure 7-2. A confirm dialog

The answer to the **confirm** prompt is returned as a Boolean value. If the user clicks OK in the **confirm** dialog shown in Figure 7-2, true is returned. If they click Cancel, false is returned.

Using Alerts to Give a Player Information

- If you want to just give the player some information, you can use an alert dialog to display a message with an OK button. For example, if you think that JavaScript is awesome, you might use this `alert` function:

```
alert("JavaScript is awesome!");
```

- Figure 7-3 shows what this simple alert dialog would look like.
- Alert dialogs just display a message and wait until the user clicks OK.

Figure 7-3. An alert dialog

Why Use alert Instead of console.log?

- Why use an alert dialog in a game instead of using console.log?
  1. First, because if all you want to do is tell the player something, using alert means the player doesn't have to interrupt game play to open the console to see a status message.
  2. Second, calling alert (as well as prompt and confirm) pauses the JavaScript interpreter until the user clicks OK (or Cancel, in the case of prompt and confirm).
- That means the player has time to read the alert. On the other hand, when you use console.log, the text is displayed immediately and the interpreter moves on to the next line in your program.

Designing Your Game

SECTION 9

Designing Your Game

- Before we start writing the Hangman game, let's think about its structure. There are a few things we need our program to do:
  1. Pick a random word.
  2. Take the player's guess.
  3. Quit the game if the player wants to.
  4. Check that the player's guess is a valid letter.
  5. Keep track of letters the player has guessed.
  6. Show the player their progress.
  7. Finish when the player has guessed the word.

Pseudo Code

- Apart from the first and last tasks (picking a word for the player to guess and finishing the game), these steps all need to happen multiple times, and we don't know how many times (it depends on how well the player guesses). When you need to do the same thing multiple times, you know you'll need a loop.
- But this simple list of tasks doesn't really give us any idea of what needs to happen when. To get a better idea of the structure of the code, we can use ***pseudocode***.

Pseudo Code

- Pseudocode is a handy tool that programmers often use to design programs. It means “fake code,” and it’s a way of describing how a program will work that looks like a cross between written English and code.
- Pseudocode has loops and conditionals, but other than that, everything is just plain English. Let’s look at a pseudocode version of our game to get an idea:

```
Pick a random word

While the word has not been guessed {
  Show the player their current progress
  Get a guess from the player

  If the player wants to quit the game {
    Quit the game
  }
  Else If the guess is not a single letter {
    Tell the player to pick a single letter
  }
  Else {
    If the guess is in the word {
      Update the player's progress with the guess
    }
  }
}

Congratulate the player on guessing the word
```

Pseudo Code

- As you can see, none of this is real code, and no computer could understand it. But it gives us an idea of how our program will be structured, before we get to actually writing the code and having to deal with the messy details, like how we're going to pick a random word.

Tracking the State of the Word

- In the previous pseudocode, one of the first lines says, “Show the player their current progress.” For the Hangman game, this means filling in the letters that the player has guessed correctly and showing which letters in the secret word are still blank. How are we going to do this? We can actually keep track of the player’s progress in a similar way to how traditional Hangman works: by keeping a collection of blank spaces and filling them in as the player guesses correct letters.
- In our game, we’ll do this using an array of blanks for each letter in the word. We’ll call this the answer array, and we’ll fill it with the player’s correct guesses as they’re made. We’ll represent each blank with the string "_".

Tracking the State of the Word

- The answer array will start out as a group of these empty entries equal in number to the letters in the secret word. For example, if the secret word is *fish*, the array would look like this:

  ["_", "_", "_", "_"]

- If the player correctly guessed the letter *i*, we'd change the second blank to an *i*:

  ["_", "i", "_", "_"]

- Once the player guesses all the correct letters, the completed array would look like this:

  ["f", "i", "s", "h"]

- We'll also use a variable to keep track of the number of remaining letters the player has to guess. For every occurrence of a correctly guessed letter, this variable will decrease by 1. Once it hits 0, we know the player has won.

Designing the Game Loop

- The main game takes place inside a **while** loop (in our pseudocode, this loop begins with the line
- "While the word has not been guessed"). In this loop we display the current state of the word being guessed (beginning with all blanks); ask the player for a guess (and make sure it's a valid, single-letter guess); and update the answer array with the chosen letter, if that letter appears in the word.

Designing the Game Loop

- Almost all computer games are built around a loop of some kind, often with the same basic structure as the loop in our Hangman game. A game loop generally does the following:
  1. Takes input from the player
  2. Updates the game state
  3. Displays the current state of the game to the player

Designing the Game Loop

- Even games that are constantly changing follow this same kind of loop — they just do it really fast. In the case of our Hangman game, the program takes a guess from the player, updates the answer array if the guess is correct, and displays the new state of the answer array.
- Once the player guesses all letters in the word, we show the completed word and a congratulatory message telling them that they won.

Coding the Game

SECTION 10

Coding the Game

- Now that we know the general structure of our game, we can start to go over how the code will look.
- The following sections will walk you through all the code in the game. After that, you'll see the whole game code in one listing so you can type it up and play it yourself.

Choosing a Random Word

- The first thing we have to do is to choose a random word. Here's how that will look:

```
❶ var words = [
    "javascript",
    "monkey",
    "amazing",
    "pancake"
  ];
❷ var word = words[Math.floor(Math.random() * words.length)];
```

Choosing a Random Word

- We begin our game at ❶ by creating an array of words (*javascript*, *monkey*, *amazing*, and *pancake*) to be used as the source of our secret word, and we save the array in the words variable. The words should be all lowercase.
- At ❷ we use Math.random and Math.floor to pick a random word from the array, as we did with the random insult generator in Chapter 3.

Creating the Answer Array

- Next we create an empty array called answerArray and fill it with underscores (_) to match the number of letters in the word.

```
var answerArray = [];
❶ for (var i = 0; i < word.length; i++) {
answerArray[i] = "_";
}
var remainingLetters = word.length;
```

Creating the Answer Array

- The for loop at ❶ creates a looping variable i that starts at 0 and goes up to (but does not include) word.length. Each time around the loop, we add a new element to answerArray, at answerArray[i].
- When the loop finishes, answerArray will be the same length as word. For example, if word is "monkey" (which has six letters), answerArray will be ["_", "_", "_", "_", "_", "_"] (six underscores).
- Finally, we create the variable remainingLetters and set it to the length of the secret word. We'll use this variable to keep track of how many letters are left to be guessed. Every time the player guesses acorrect letter, this value will be *decremented* (reduced) by 1 for each instance of that letter in the word.

Coding the Game Loop

- The skeleton of the game loop looks like this:

```
while (remainingLetters > 0) {
  // Game code goes here
  // Show the player their progress
  // Take input from the player
  // Update answerArray and remainingLetters for every correct guess
}
```

- We use a while loop, which will keep looping as long as remainingLetters > 0 remains true. The body of the loop will have to update remainingLetters for every correct guess the player makes. Once the player has guessed all the letters, remainingLetters will be 0 and the loop will end.
- The following sections explain the code that will make up the body of the game loop.

Showing the Player's Progress

Figure 7-4. Showing the player's progress using `alert`

- The first thing we need to do inside the game loop is to show the player their current progress:

```
alert(answerArray.join(" "));
```

- We do that by joining the elements of answerArray into a string, using the space character as the separator, and then using alert to show that string to the player. For example, let's say the word is monkey and the player has guessed m, o, and e so far. The answer array would look like this ["m", "o", "_", "_", "e", "_"], and answerArray.join(" ") would be "m o _ _ e _". The alert dialog would then look like Figure 7-4.

Handling the Player's Input

- Now we have to get a guess from the player and ensure that it's a single character.

```
❶ var guess = prompt("Guess a letter, or click Cancel to stop playing.");
❷ if (guess === null) {
        break;
❸ } else if (guess.length != 1) {
        alert("Please enter a single letter.");
  } else {
❹       // Update the game state with the guess
  }
```

Handling the Player's Input

- At ❶, `prompt` takes a guess from the player and saves it to the variable `guess`. One of four things will happen at this point.
- First, if the player clicks the Cancel button, then `guess` will be `null`.
- We check for this condition at ❷ with `if (guess === null).` If this condition is true, we use break to exit the loop.

Handling the Player's Input

- The second and third possibilities are that the player enters either nothing or too many letters. If they enter nothing but click OK, guess will be the empty string "". In this case, guess.length will be 0. If they enter anything more than one letter, guess.length will be greater than 1.
- At ❸, we use else if (guess.length !== 1) to check for these conditions, ensuring that guess is exactly one letter. If it's not, we display an alert saying, "Please enter a single letter."
- The fourth possibility is that the player enters a valid guess of one letter. Then we have to update the game state with their guess using the else statement at ❹, which we'll do in the next section.

Updating the Game State

- Once the player has entered a valid guess, we must update the game's answerArray according to the guess. To do that, we add the following code to the else statement:

```
❶ for (var j = 0; j < word.length; j++) {
❷    if (word[j] === guess) {
                answerArray[j] = guess;
❸               remainingLetters--;
      }
   }
```

Updating the Game State

- At ❶, we create a for loop with a new looping variable called j, which runs from 0 up to word.length. (We're using j as the variable in this loop because we already used i in the previous for loop.)
- We use this loop to step through each letter of word. For example, let's say word is pancake. The first time around this loop, when j is 0, word[j] will be "p". The next time, word[j] will be "a", then "n", "c", "a", "k", and finally "e".

Updating the Game State

- At ❷, we use `if (word[j]===guess)` to check whether the current letter we're looking at matches the player's guess. If it does, we use `answerArray[j]=guess` to update the answer array with the current guess. For each letter in the word that matches guess, we update the answer array at the corresponding point.
- This works because the looping variable j can be used as an index for `answerArray` just as it can be used as an index for word, as you can see in Figure 7-5.

Index (j)		0	1	2	3	4	5	6	
word	"	p	a	n	c	a	k	e	"
answerArray	["	_	", "	_	", "	_	", "	_	", "	_	", "	_	", "	_	"]

Figure 7-5. The same index can be used for both `word` and `answerArray`.

Updating the Game State

- For example, imagine we've just started playing the game and we reach the for loop at ❶. Let's say `word` is `"pancake"`, guess is `"a"`, and `answerArray` currently looks like this:

```
["_","_","_","_","_","_","_"]
```

- The first time around the `for` loop at ❶, `j` is 0, so `word[j]` is `"p"`. Our guess is `"a"`, so we skip the if statement at ❷ (because `"p" === "a"` is `false`). The second time around, j is 1, so `word[j]` is `"a"`.

- This *is* equal to `guess`, so we enter the if part of the statement. The line `answerArray[j] = guess;` sets the element at index 1 (the second element) of `answerArray` to guess, so `answerArray` now looks like this:

```
["_","a","_","_","_","_","_"]
```

Updating the Game State

- The next two times around the loop, word[j] is "n" and then "c", which don’t match guess. However, when j reaches 4, word[j] is "a" again. We update answerArray again, this time setting the element at index 4 (the fifth element) to guess. Now answerArray looks like this:

  ["_", "a", "_", "_", "a", "_", "_"]

- The remaining letters don’t match "a", so nothing happens the last two times around the loop. At the end of this loop, answerArray will be updated with all the occurrences of guess in word.

- For every correct guess, in addition to updating answerArray, we also need to decrement remainingLetters by 1. We do this at ❸ using remainingLetters--;. Every time guess matches a letter in word, remainingLetters decreases by 1. Once the player has guessed all the letters correctly, remainingLetters will be 0.

Ending the Game

SECTION 11

Ending the Game

- As we've already seen, the main game loop condition is **`remainingLetters > 0`**, so as long as there are still letters to guess, the loop will keep looping. Once **`remainingLetters`** reaches 0, we leave the loop. We end with the following code:

```
alert(answerArray.join(" "));
alert("Good job! The answer was " + word);
```

- The first line uses alert to show the answer array one last time. The second line uses alert again to congratulate the winning player.

The Game Code
Demo Program:
HangMan.html

- Now we've seen all the code for the game, and we just need to put it together. What follows is the full listing for our Hangman game. I've added comments throughout to make it easier for you to see what's happening at each point. It's quite a bit longer than any of the code we've written so far, but typing it out will help you to become more familiar with writing JavaScript.
- Create a new HTML file called *hangman.html* and type the following into it:

```
<!DOCTYPE html>
<html>
    <head><title>Hangman!</title></head>
<body>
<h1>Hangman!</h1>
<script>
    // Create an array of words
    var words = ["javascript", "monkey", "amazing", "pancake"];
    // Pick a random word
    var word = words[Math.floor(Math.random() * words.length)];
    // Set up the answer array
    var answerArray = [];
    for (var i = 0; i < word.length; i++) {
        answerArray[i] = "_";
    }
    var remainingLetters = word.length;
    // The game loop
    while (remainingLetters > 0) {
        // Show the player their progress
        alert(answerArray.join(" "));
        // Get a guess from the player
        var guess = prompt("Guess a letter, or click Cancel to stop playing.");
```

```
23          if (guess === null) {
24              // Exit the game loop
25              break;
26          } else if (guess.length !== 1) {
27              alert("Please enter a single letter.");
28          } else {
29              // Update the game state with the guess
30              for (var j = 0; j < word.length; j++) {
31                  if (word[j] === guess) {
32                      answerArray[j] = guess;
33                      remainingLetters--;
34                  }
35              }
36          }
37          // The end of the game loop
38      }
39      // Show the answer and congratulate the player
40      alert(answerArray.join(" "));
41      alert("Good job! The answer was " + word);
42  </script>
43  </body>
44  </html>
```

The Game Code

- If the game doesn't run, make sure that you typed in everything correctly. If you make a mistake, the JavaScript console can help you find it. For example, if you misspell a variable name, you'll see something like Figure 7-6 with a pointer to where you made your mistake.

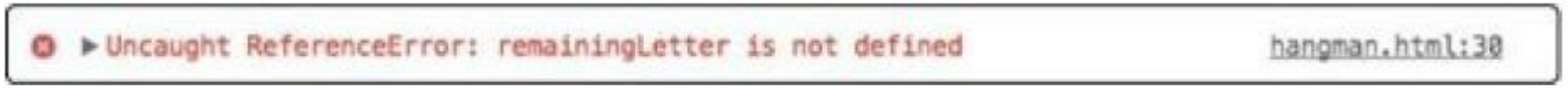
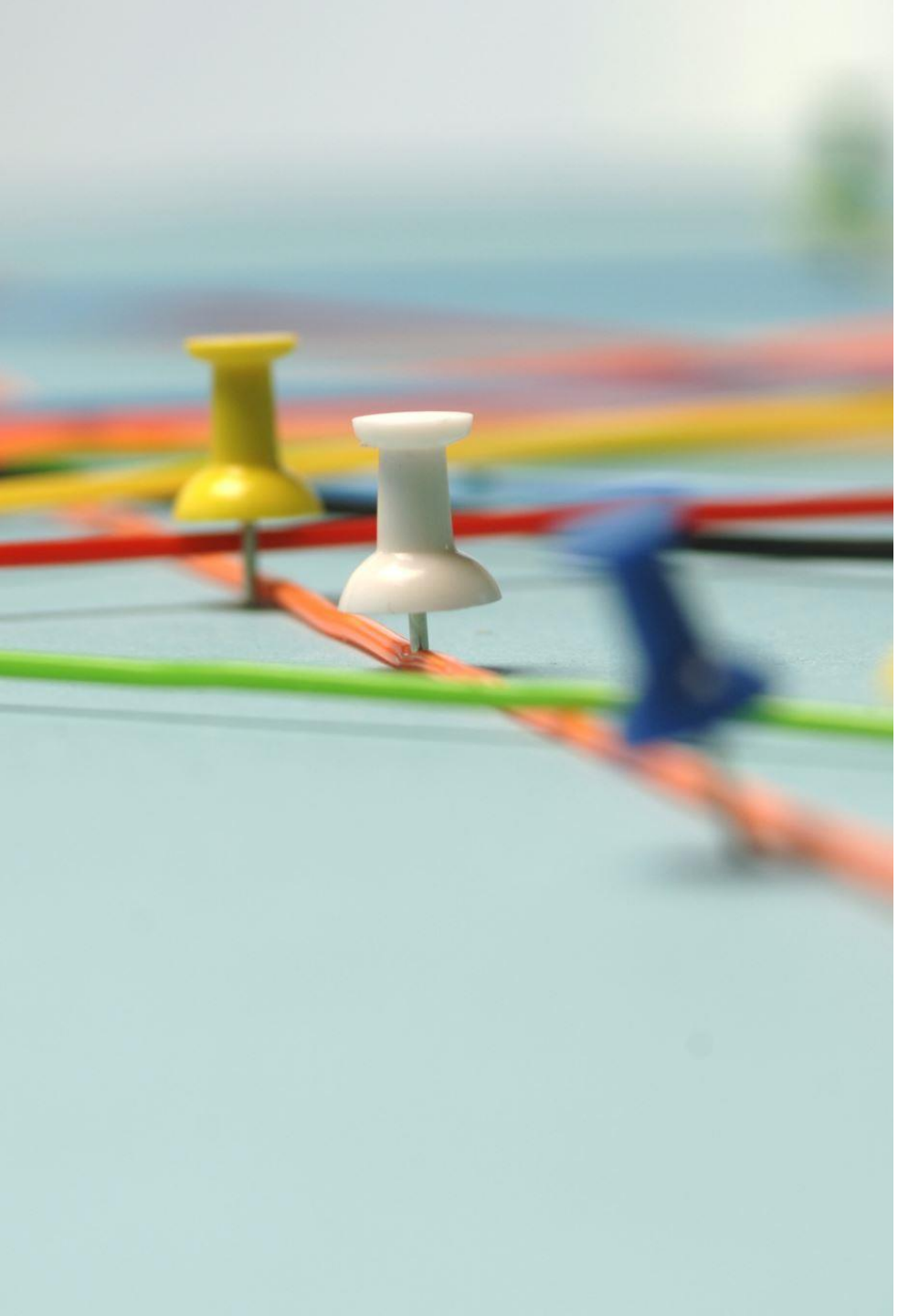


Figure 7-6. A JavaScript error in the Chrome console



The Game Code

- If you click **`hangman.html:30`**, you'll see the exact line where the error is. In this case, it's showing us that we misspelled **`remainingLetters`** as **`remainingLetter`** at the start of the **`while`** loop.
- Try playing the game a few times. Does it work the way you expected it to work? Can you imagine the code you wrote running in the background as you play it?

Functions

SECTION 12

The Basic Anatomy of a Function

- Figure below shows how a function is built. The code between the curly brackets is called the *function body*, just as the code between the curly brackets in a loop is called the *loop body*.

```
function () {
    console.log("Do something");
}
```

The function body goes between curly brackets.

Creating a Simple Function

- Let’s create a simple function that prints Hello world!. Enter the following code in the browser
- console. Use SHIFT-ENTER to start each new line without executing the code.

```
var ourFirstFunction =
function () {
console.log("Hello world!");
};
```

- This code creates a new function and saves it in the variable **ourFirstFunction**.

Calling a Function

- To run the code inside a function (the function body), we need to call the function. To call a function, you enter its name followed by a pair of opening and closing parentheses, as shown here.

```
> ourFirstFunction();
Hello world!
```

- Calling **ourFirstFunction** executes the body of the function, which is **console.log("Hello world!");**, and the text we asked to be printed is displayed on the next line: **`Hello world!`**.
- But if you call this function in your browser, you'll notice that there's a third line, with a little leftfacing arrow, as shown in Figure 8-2. This is the return value of the function.

Functional Call

```
> ourFirstFunction();
  Hello, world!
< undefined
```

Figure 8-2. Calling a function with an undefined return value

Functional Call with undefined Return Value

- A **return** value is the value that a function outputs, which can then be used elsewhere in your code.
- In this case, the return value is undefined because we didn't tell the function to return any particular value in the body of the function.

Functional Call with undefined Return Value

- All we did was ask it to print a message to the console, which is not the same as returning a value.
- A function always returns **undefined** unless there is something in the function body that tells it to return a different value. (We'll look at how to specify a return value in Returning Values from Functions.)

In-Class Demonstration Program

- Function returning undefined

FUNCTION2.HTML

Parameters

SECTION 13

Passing Arguments into Functions

- **ourFirstFunction** just prints the same line of text every time you call it, but you'll probably want your functions to be more flexible than that.
- Function *arguments* allow us to pass values into a function in order to change the function's behavior when it's called.
- Arguments always go between the function parentheses, both when you create the function and when you call it.

Passing Arguments into Functions

- The following **`sayHelloTo`** function uses an argument (name) to say hello to someone you specify.

```
var sayHelloTo =
function (name) {
  console.log("Hello " +
name + "!");
};
```

Passing Arguments into Functions

- We create the function in the first line and assign it to the variable **sayHelloTo**. When the function is called, it logs the string “Hello ” + name + “!”, replacing name with whatever value you pass to the function as an argument.
- Figure 8-3 shows the syntax for a function with one argument.

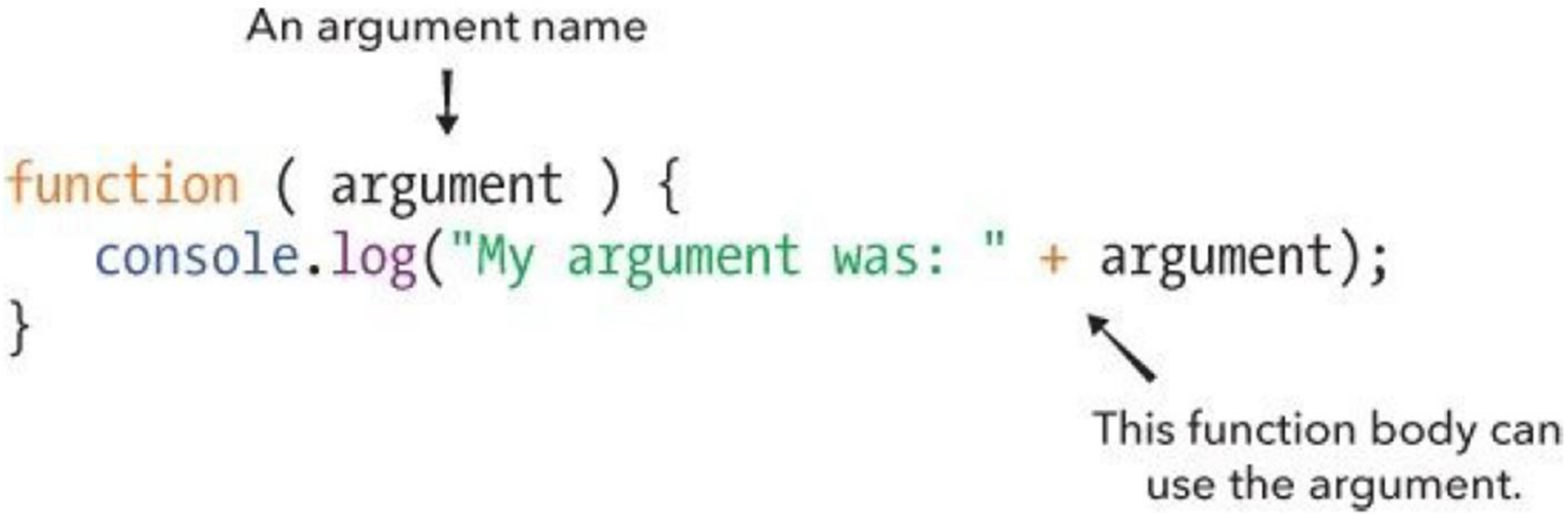


Figure 8-3. The syntax for creating a function with one argument

Passing Arguments into Functions

- To call a function that takes an argument, place the value you'd like to use for the argument between the parentheses following the function name. For example, to say hello to Nick, you would write:

```
sayHelloTo("Nick");
Hello Nick!
```

Passing Arguments into Functions

- Or, to say hello to Lyra, write:

```
sayHelloTo("Lyra");
Hello Lyra!
```

- Each time we call the function, the argument we pass in for name is included in the string printed by the function. So when we pass in “Nick”, the console prints “Hello Nick!”, and when we pass in “Lyra”, it prints "Hello Lyra!".

Demo Program: sayHello.html

```
1  <html>
2  <body>
3  <script>
4    var sayHelloTo = function (name) {
5     document.write("Hello " + name + "!<br>");
6     };
7    sayHelloTo("Eric");
8    sayHelloTo("Tom");
9    sayHelloTo("Johnny");
10 </script>
11 </body>
12 </html>
```

Printing Cat Faces!

```
var drawCats = function (howManyTimes){
for (var i = 0; i < howManyTimes; i++){
     console.log(i + " =^.^=");
  }
};
```

Printing Cat Faces!

Demo Program: printCats.html

```
drawCats(5);
```

```
0 =^.^=
1 =^.^=
2 =^.^=
3 =^.^=
4 =^.^=
```

```
<html>
<body>
<script>
var drawCats = function (howManyTimes){
for (var i = 0; i < howManyTimes; i++){
      document.write(i + " =^.^=<br>");
  }
};
 drawCats(5);
</script>
</body>
</html>
```

Multiple Arguments

SECTION 14

Passing Multiple Arguments to a Function

Each argument name is separated by a comma.

```
function (argument1, argument2) {
    console.log("My first argument was: " + argument1);
    console.log("My second argument was: " + argument2);
}
```

The function body can use both arguments.

Passing Multiple Arguments to a Function

- The following function, **printMultipleTimes**, is like drawCats except that it has a second argument called **whatToDraw**.

```
var printMultipleTimes = function (howManyTimes, whatToDraw){
for (var i = 0; i < howManyTimes; i++){
    console.log(i+" "+whatToDraw);
  }
};
```

Multiple Parameters

Demo Program: printManyTimes.html

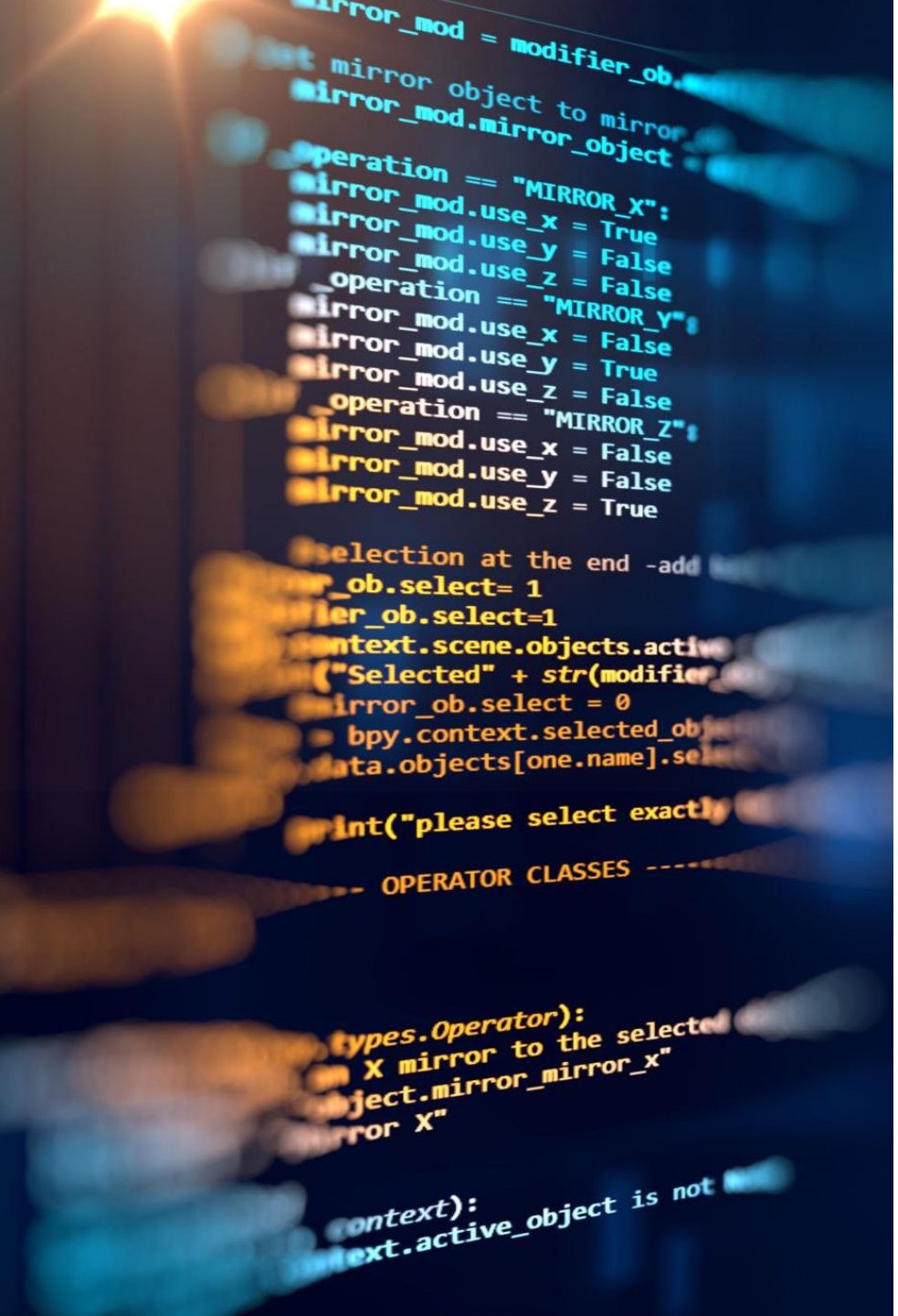
```
> printMultipleTimes(5, "=^.^=");
0 =^.^=
1 =^.^=
2 =^.^=
3 =^.^=
4 =^.^=
```

```
printMultipleTimes(4, "^_^");
0 ^_^
1 ^_^
2 ^_^
3 ^_^
```

```
<html>
<body>
<script>
var printMultipleTimes = function (howManyTimes, whatToDraw){
for (var i = 0; i < howManyTimes; i++){
      document.write(i+" "+whatToDraw+"<br>");
  }
};
 printMultipleTimes(5, "=^.^=");
 printMultipleTimes(4, "^_^");
</script>
</body>
</html>
```

Return Value

SECTION 15



Returning Values from Functions

- The output of a function is called the return value. When you call a function that returns a value, you can use that value in the rest of your code (you could save a return value in a variable, pass it to another function, or simply combine it with other code).
- For example, the following line of code adds 5 to the return value of the call to **`Math.floor(1.2345)`**:

```
> 5 + Math.floor(1.2345);
6
```

Define a Function with Return Value

- Let's create a function that returns a value. The function double takes the argument number and returns the result of number * 2. In other words, the value returned by this function is twice the number supplied as its argument.

```
var double = function (number) {
    return number * 2;
};
> double(3);
6
```

In-Class Demonstration Program

- Dice (6 faces)
- Dice (multiple faces, base)

DICE.HTML

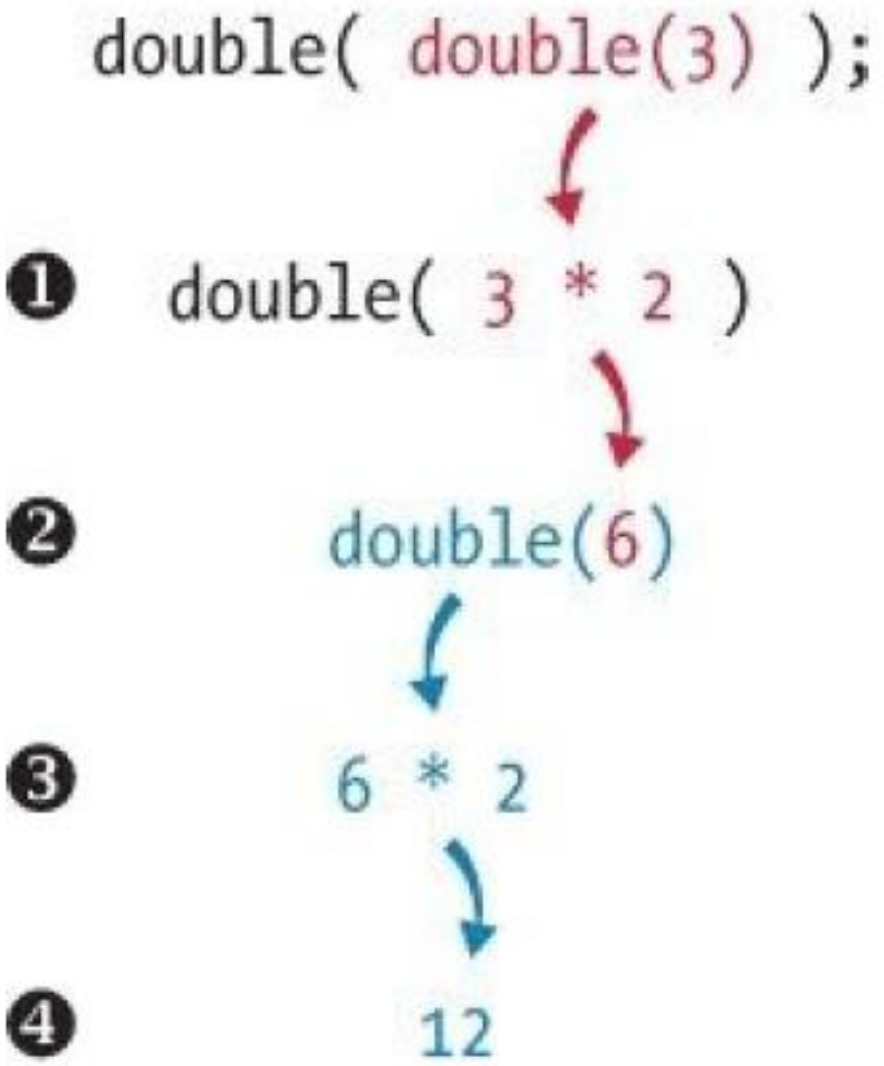
Function Call as value

SECTION 16

Using Function Calls as Values

- You can also pass a function call into another function as an argument, and the function call will be substituted with its return value.
- In this next example we call double, passing the result of calling **`double`** with 3 as an argument. We replace double(3) with 6 so that **`double(double(3))`** simplifies to **`double(6)`**, which then simplifies to 12.

```
> double(double(3));
12
```


double(double(3));
❶ double(3 * 2)
❷ double(6)
❸ 6 * 2
❹ 12

Procedural Abstraction [Optional]

SECTION 17

Procedural Abstraction

- Procedural abstraction is the idea that each method should have a coherent conceptual description that separates its implementation from its users.
- You can encapsulate behavior in methods that are internal to an object or methods that are widely usable.

Using Functions to Simplify Code

In Chapter 3, we used the methods **`Math.random`** and **`Math.floor`** to pick random words from arrays and generate random insults. In this section, we'll re-create our insult generator and simplify it by creating functions.

A Function to Pick a Random Word

PART 1

A Function to Pick a Random Word

Demo Program: randomWords.html

```
var randomWords = ["Planet", "Worm",
                   "Flower", "Computer"];
var pickRandomWord = function (words) {
    return words[Math.floor(Math.random()
                 * words.length)];
};

> pickRandomWord(randomWords);
"Flower"
```

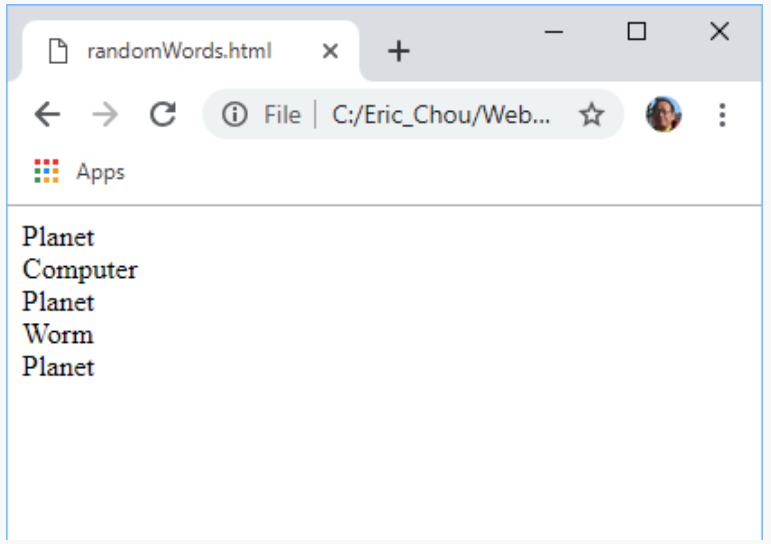
A Function to Pick a Random Word

- We can use this same function on any array. For example, here's how we would pick a random name from an array of names:

```
> pickRandomWord(["Charlie", "Raj", "Nicole", "Kate", "Sandy"]);

"Raj"
```

```
1  <html>
2  <body>
3  <script>
4    var randomWords = ["Planet", "Worm", "Flower", "Computer"];
5  var pickRandomWord = function (words) {
6      return words[Math.floor(Math.random() * words.length)];
7  };
8  for (var i=0; i<5; i++){
9      var w = pickRandomWord(randomWords);
10     document.write(w+"<br>");
11 }
12 </script>
13 </body>
14 </html>
```



A Random Insult Generator

PART 2

```
var randomBodyParts = ["Face", "Nose", "Hair"];
var randomAdjectives = ["Smelly", "Boring", "Stupid"];
var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];

// Pick a random body part from the randomBodyParts array:
var randomBodyPart = randomBodyParts[Math.floor(Math.random() * 3)];
// Pick a random adjective from the randomAdjectives array:
var randomAdjective = randomAdjectives[Math.floor(Math.random() * 3)];
// Pick a random word from the randomWords array:
var randomWord = randomWords[Math.floor(Math.random() * 5)];
// Join all the random strings into a sentence:
var randomString = "Your " + randomBodyPart + " is like a " + ↵
randomAdjective + " " + randomWord + "!!!";
randomString;
"Your Nose is like a Stupid Marmot!!!"
```

A Random Insult Generator
Demo Program: insult1.html

- Now let's try re-creating our random insult generator, using our function that picks random words. First, here's a reminder of what the code from Chapter 3 looked like:

A Random Insult Generator

- Notice that we end up repeating ***words*[Math.floor(Math.random() * *length*)]** quite a few times in this code. Using our **pickRandomWord** function, we could rewrite the program like this:

```
var randomBodyParts = ["Face", "Nose", "Hair"];
var randomAdjectives = ["Smelly", "Boring", "Stupid"];
var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];

// Join all the random strings into a sentence:
var randomString = "Your " + pickRandomWord(randomBodyParts) + ↵
" is like a " + pickRandomWord(randomAdjectives) + ↵
" " + pickRandomWord(randomWords) + "!!!";

randomString;
"Your Nose is like a Smelly Marmot!!!"
```

A Random Insult Generator

- There are two changes here. First, we use the **`pickRandomWord`** function when we need a random word from an array, instead of using **`words[Math.floor(Math.random()*length)]`** each time. Also, instead of saving each random word in a variable before adding it to the final string, we're adding the return values from the function calls directly together to form the string. A call to a function can be treated as the value that the function returns. So really, all we're doing here is adding together strings.
- As you can see, this version of the program is a lot easier to read, and it was easier to write too, since we reused some code by using a function.

```
<html>
<body>
<script>
var pickRandomWord = function (words) {
    return words[Math.floor(Math.random() * words.length)];
};
var randomBodyParts = ["Face", "Nose", "Hair"];
var randomAdjectives = ["Smelly", "Boring", "Stupid"];
var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];
// Pick a random body part from the randomBodyParts array:
var randomBodyPart = randomBodyParts[Math.floor(Math.random() * 3)];
// Pick a random adjective from the randomAdjectives array:
var randomAdjective = randomAdjectives[Math.floor(Math.random() * 3)];
// Pick a random word from the randomWords array:
var randomWord = randomWords[Math.floor(Math.random() * 5)];
// Join all the random strings into a sentence:
var randomString = "Your " + randomBodyPart + " is like a " +
randomAdjective + " " + randomWord + "!!!";
document.write(randomString+"<br>");
</script>
</body>
</html>
```

Making the Random Insult Generator into a Function

PART 3

Making the Random Insult Generator into a Function

- We can take our random insult generator one step further by creating a larger function that produces random insults. Let's take a look:

```
generateRandomInsult = function () {
  var randomBodyParts = ["Face", "Nose", "Hair"];
  var randomAdjectives = ["Smelly", "Boring", "Stupid"];
  var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];
  // Join all the random strings into a sentence:
  var randomString = "Your " + pickRandomWord(randomBodyParts) + ↵
  " is like a " + pickRandomWord(randomAdjectives) + ↵
  " " + pickRandomWord(randomWords) + "!!!";

❶   return randomString;
};
```

```
generateRandomInsult();
"Your Face is like a Smelly Stick!!!"
generateRandomInsult();
"Your Hair is like a Boring Stick!!!"
generateRandomInsult();
"Your Face is like a Stupid Fly!!!"
```

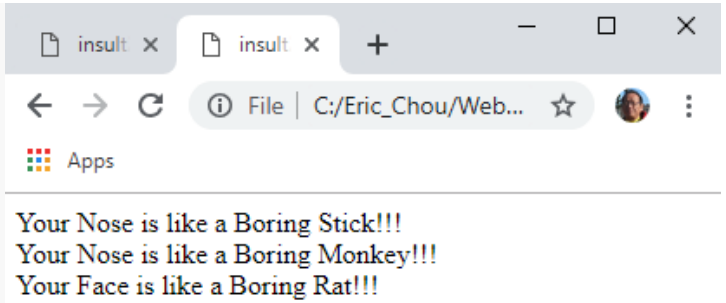
Making the Random Insult Generator into a Function

Making the Random Insult Generator into a Function

- Our new **`generateRandomInsult`** function is just the code from before placed inside a function with no arguments. The only addition is at ❶, where we have the function return **`randomString`** at the end. You can see a few sample runs of the preceding function, and it returns a new insult string each time.
- Having the code in one function means we can keep calling that function to get a random insult, instead of having to copy and paste the same code every time we want a new insult.

Demo Program: insult2.html

```
1  <html>
2  <body>
3  <script>
4  var pickRandomWord = function (words) {
5      return words[Math.floor(Math.random() * words.length)];
6  };
7
8  var generateRandomInsult = function () {
9      var randomBodyParts = ["Face", "Nose", "Hair"];
10     var randomAdjectives = ["Smelly", "Boring", "Stupid"];
11     var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];
12     // Join all the random strings into a sentence:
13     var randomString = "Your " + pickRandomWord(randomBodyParts) +
14                        " is like a " + pickRandomWord(randomAdjectives) +
15                        " " + pickRandomWord(randomWords) + "!!!";
16     return randomString;
17     };
18 for (var i=0; i<3; i++){
19     var randomString = generateRandomInsult();
20     document.write(randomString+"<br>");
21   }
22 </script>
23 </body>
24 </html>
```



Return as a Break Level

SECTION 18

Leaving a Function Early with return

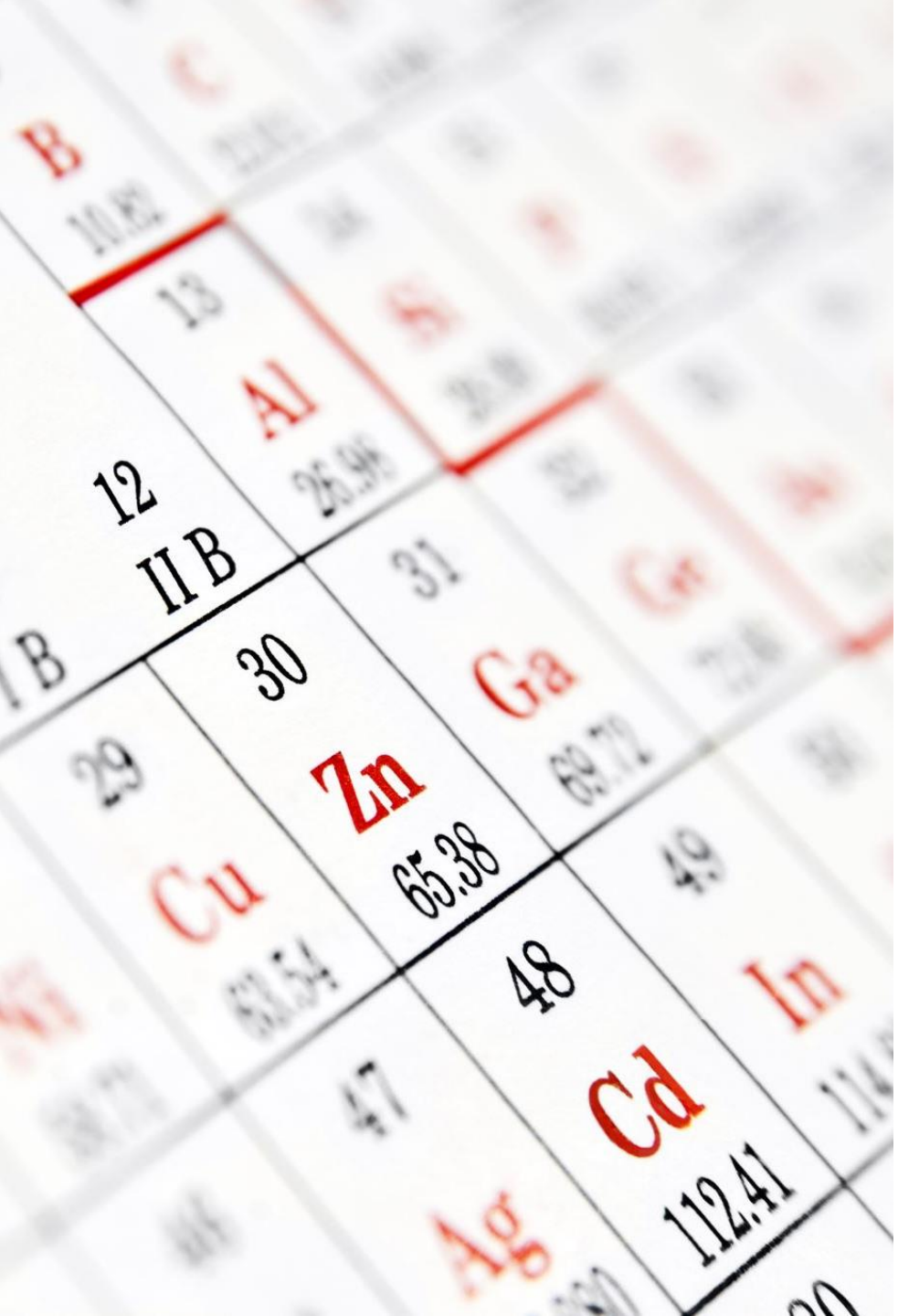
- One common way to use return is to leave a function early if any of the arguments to the function are *invalid*; that is, if they're not the kind of arguments the function needs in order to run properly.
- For example, the following function returns a string telling you the fifth character of your name. If the name passed to the function has fewer than five characters, the function uses return to leave the function immediately.
- This means the return statement at the end, which tells you the fifth letter of your name, is never executed.

Return as a break level

```
  var fifthLetter = function (name) {
❶   if (name.length < 5) {
❷     return;
    }

    return "The fifth letter of your name is " + name[4] + ".";
  };
```

- At ❶ we check to see whether the length of the input name is less than five. If it is, we use return at ❷ to exit the function early.



Return as a break level

- Let's try calling this function.

```
> fifthLetter("Nicholas");
"The fifth letter of your name is o."
```

- The name *Nicholas* is longer than five characters, so **fifthLetter** completes and returns the fifth letter in the name *Nicholas*, which is the letter *o*. Let's try calling it again on a shorter name:

```
> fifthLetter("Nick");
undefined
```

- When we call **fifthLetter** with the name *Nick*, the function knows that the name isn't long enough, so it exits early with the first return statement at ❷. Because there is no value specified after the return at ❷, the function returns **undefined**.

Replacement of else by return [Optional]

SECTION 19

```
var medalForScore = function (score) {
  if (score < 3) {
❶     return "Bronze";
  }

❷   if (score < 7) {
      return "Silver";
  }

❸   return "Gold";
};
```

Replacement of Else by Return

- We can use multiple return keywords inside different if statements in a function body to have a function return a different value depending on the input. For example, say you're writing a game that awards players medals based on their score. A score of 3 or below is a bronze medal, scores between 3 and 7 are silver, and anything above 7 is gold. You could use a function like **medalForScore** to evaluate a score and return the right kind of medal, as shown here:

Function as HTML Wrapper [Optional]

SECTION 20

HTML Wrapper **Demo Program: wrapper.html**

- In order to write an object to the document, functions can be used as HTML wrappers.
- Using function calls can make repetition more efficient.
- The template can be pre-defined and then, brought into JavaScript section.
- Template literals can be used (but not demonstrated in here.)
- The pre-defined template is in the field.html file.

```
<html>
<body>
<script>
 function print(student){
    return '<fieldset><ul>'+
           '<li>Name: '+student.name+'</li>'+
           '<li>Age: '+student.age+'</li>'+
           '<li>Address: '+student.address+', '+student.city+', '+student.state+', '+
           student.zipcode+'</li>'+
           '</ul></fieldset>';
  }
var s1 = {
  name: "Tommy Jones",
  age: 15,
  address: "1 A Street",
  city: "Los Angeles",
  state: "CA",
  zipcode: "90007"
 };
var s2 = {
  name: "Nancy Adams",
  age: 16,
  address: "100 B Street",
  city: "Las Vegas",
  state: "NV",
  zipcode: "80109"
 };
 document.write(print(s1));
 document.write(print(s2));
</script>
</body>
</html>
```

- Name: Tommy Jones
- Age: 15
- Address: 1 A Street, Los Angeles, CA, 90007

- Name: Nancy Adams
- Age: 16
- Address: 100 B Street, Las Vegas, NV, 80109