1. Introduction and Setup

1.1. What is JavaScript?

JavaScript is an interpreted language that was initially created to 'add life' to static web pages.

You can think of a web page as consisting of three separate layers:

- The HTML layer lets you specify the content of a web page.
- The CSS layer lets you specify the presentation of a web page, the way a web page looks.
- The JavaScript layer lets you specify the behavior of a web page, the way it interacts with the user.

Today, it is really hard to find a web page that does not use JavaScript. All web browsers, on computers, tablets and phones include JavaScript interpreters.

As we'll see later in this course, JavaScript has also become common in server-side programming.

1.2. Origins of JavaScript

In 1995, Netscape's Navigator was the dominant web browser. The company decided to add interactivity to HTML pages, with a lightweight programming language.

Brendan Eich developed a first version in 10 days. Its initial name was Mocha but it was quickly changed to LiveScript. In December 1995, Netscape and Sun had a licensing agreement that led to the programming language's final name, JavaScript. At that point, it was included in Netscape Navigator releases.

The name JavaScript came from the intended role for the language: Java was supposed to provide the large-scale building blocks for web applications, while JavaScript was to be the glue, connecting the blocks. Today JavaScript dominates the browser, Java is mostly non-existent there.

1.3. Standards, Names and Version Numbers

After JavaScript came out, Microsoft implemented the same language, under the different name JScript, in Internet Explorer 3.0. Netscape submitted the language for standardization to ECMA (European Computer Manufacturer's Association). Because Sun had a trademark on the word Java, the language to be standardized couldn't be called JavaScript. ECMAScript was chosen as the name of the standard language. Its implementations are officially called JavaScript and JScript. When talking unofficially about either the standard or its implementations, we use the term JavaScript.

The current version of ECMAScript is 5.1, released in June 2011. This is also known as ECMA-262, edition 5.1.

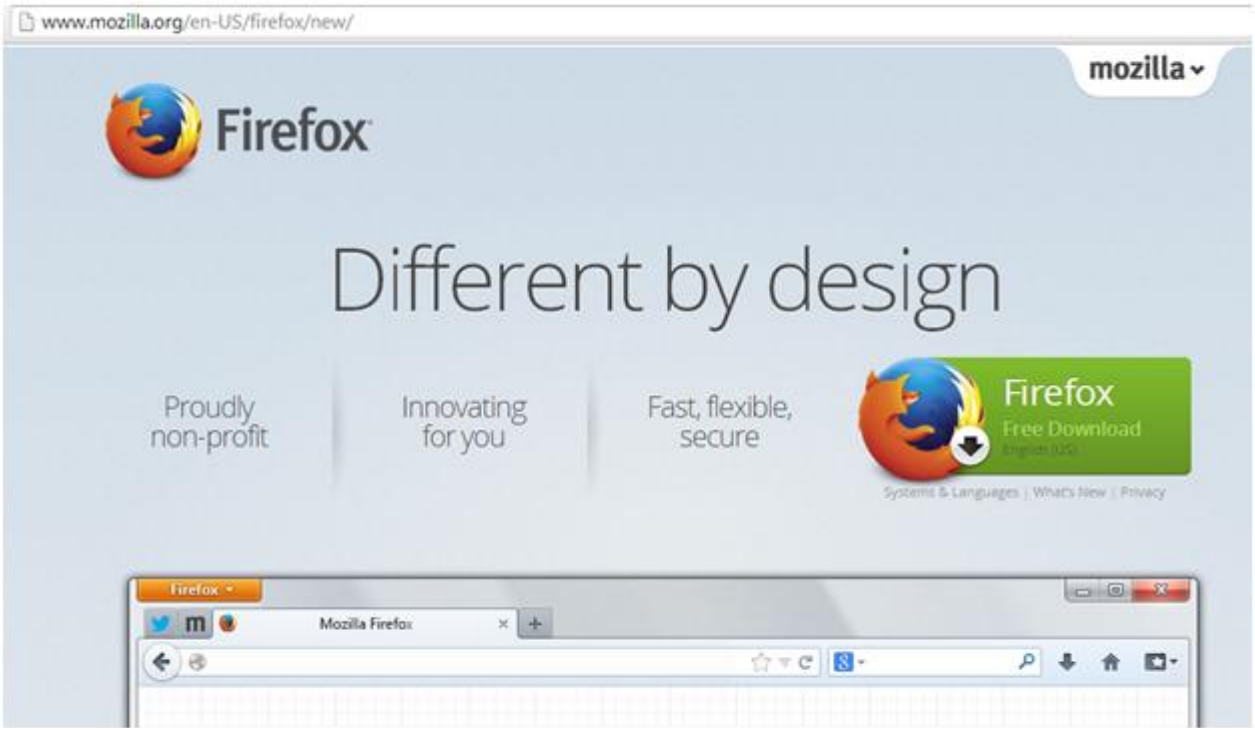
Sometimes you'll also see a JavaScript version number such as JavaScript 1.5 or JavaScript 1.8.5. These are Mozilla's version numbers: JavaScript 1.5 implements ECMAScript 3 and JavaScript 1.8.5 implements ECMAScript 5 with some added

features. Google JavaScript Engine is V8 and the current version of the V8 engine implements ECMAScript 5.

1.4. Setting Up Our Environment: Firefox and Firebug

To get started with JavaScript, we need an interpreter. All current web browsers include a JavaScript interpreter. In this course, we'll use Mozilla's Firefox.

To download the latest version of Firefox, go to http://www.mozilla.org/en-US/firefox/new/



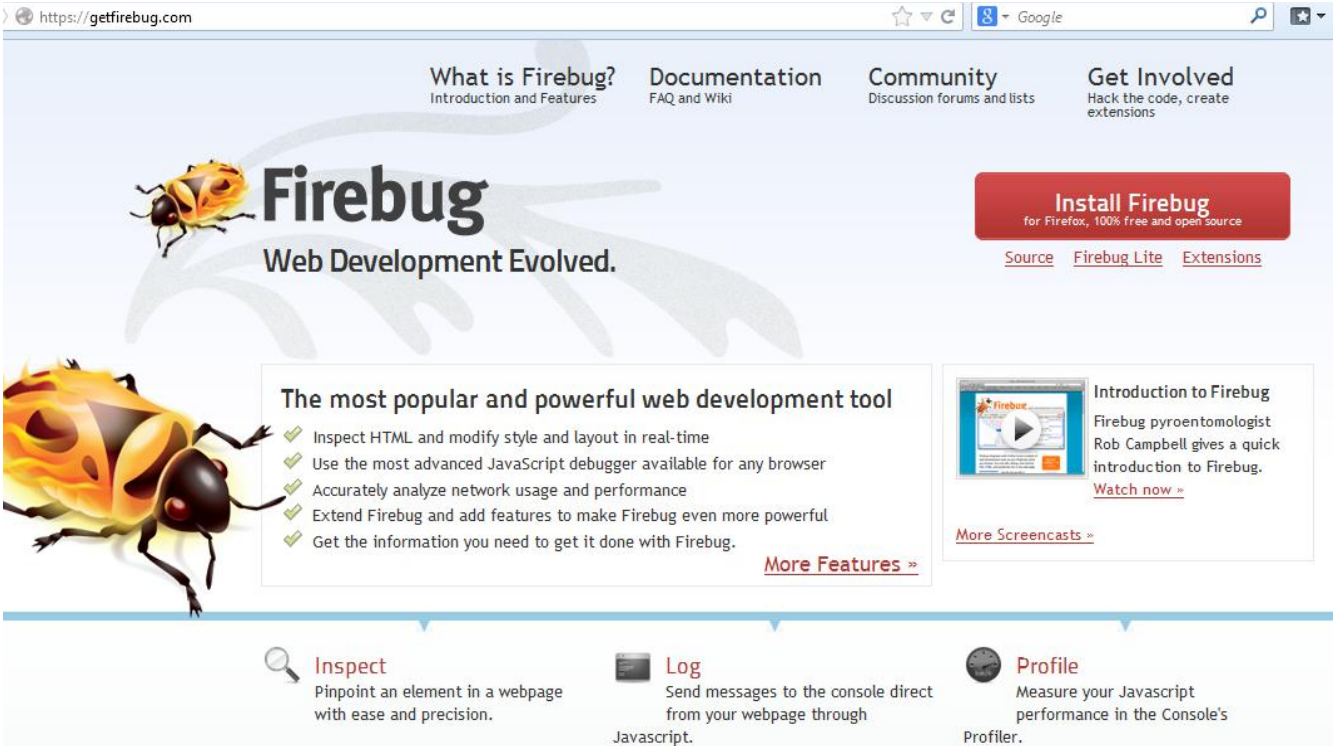
click on: Firefox Free Download

and follow the installation instructions.

Once you have installed Firefox, you'll need to install the Firebug extension. It is a powerful web development tool that facilitates the debugging, editing, and monitoring of any website's CSS, HTML, DOM, and JavaScript.

Go to :

https://getfirebug.com/



click on Install Firebug

and follow the installation instructions.

2. JavaScript Basics

2.1. Using the Firebug Console

We'll start by using the Firebug Console. The Console provides an interactive JavaScript command line to execute little bits of code. It is accessible from the Firefox browser.

From here on, instead of passively reading along, you should start Firebug and try to replicate the actions.

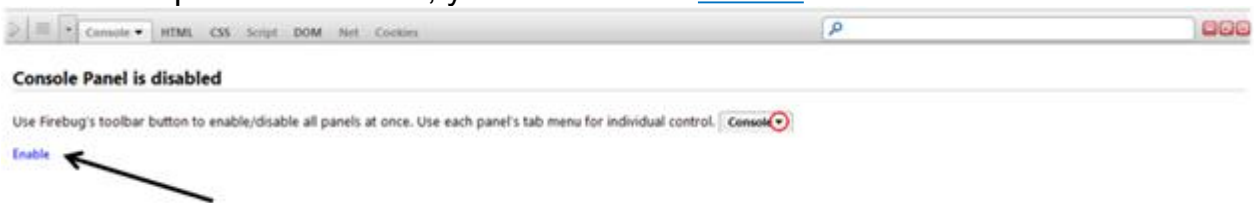
The following discussion is also illustrated in the screencast:

http://youtu.be/gGrW2e6TsZA

There are two main ways to open Firebug: you can click on the Firebug icon on the Firefox status bar or press the F12 key on your keyboard.

You can then select the console tab from the Firebug menu.

If the console panel is disabled, you can click on Enable to enable it



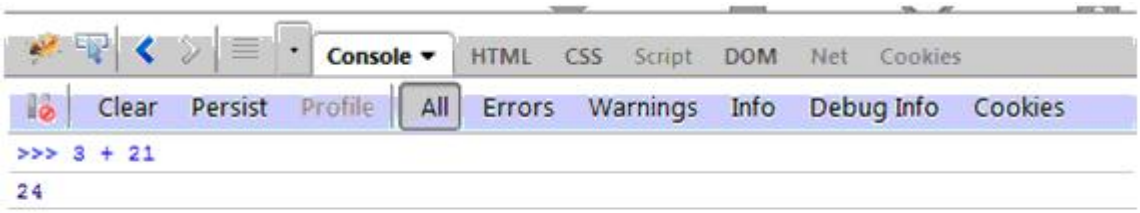
You can type JavaScript code directly at the '>>>' prompt at the bottom of the Firebug window. Whenever you enter a code fragment and press Enter, it will be executed. For instance, typing:

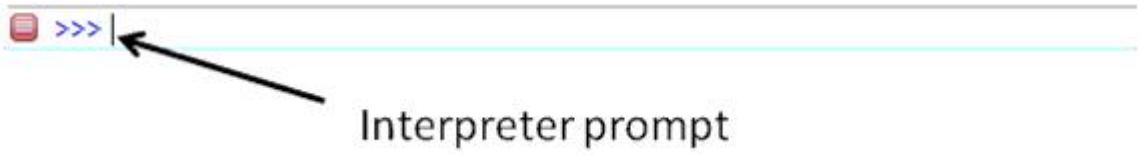
3 + 21

and pressing Enter will cause the following to be displayed on the top part of the window:

>>> 3 + 21

24





Let's try a few more operators:

>>> 4 * 5 // this is a comment. It will be ignored by the interpreter.

20

>>> 10 – 4 /* this is also a comment */

6

>>> 8 / 2 // this is the division operator

4

>>> 9 / 2

4.5

>>> 10 % 3 // this is the modulo operator also known as the remainder operator

When you're done, you can close Firebug by using the off button in the upper right corner of the Firebug console.

2.2. Comments

By now you may have noticed that JavaScript supports two forms of comments:

Line-ending comments starting with // such as in:

>>> 4 * 5 // this is a comment. It will be ignored by the interpreter.

These comments go on until the end of the line.

Block comments formed with /* */ such as in:

>>> 10 – 4 /* this is also a comment */

' /*' starts a comment that goes on until a '*/' is found.

Block comments may span multiple lines.

You will often see JavaScript files, classes, methods and properties commented as follows:

```
/**
 * This is a multiple line comment.
 * This style is used to conform to the JSDoc markup language.
 * JSDoc is used to generate documentation describing the application programming interface.
 */
```

The only delimiters required by JavaScript here are the opening /* and the closing */.

2.3. Operator Precedence

Now let's try the following to explore the order of operations:

>>> 1 + 2 * 3

7

>>>3 * 2 - 4

2

>>> 100 * 4 - 4 + 50 / 2

421

Multiplication has a higher precedence than addition. Division and multiplication always come before subtraction and addition. When multiple operators with the same precedence appear next to each other they are applied left-to-right.

When in doubt, just add parentheses.

>>> (1+2)*3

9

2.4. JavaScript Types

The primitive types of JavaScript include numbers (such as 4 and -6.8), strings (‘Hello’) and booleans (true and false).

The special values null and undefined are also primitive values.

Everything else is an object (member of type object).

The `typeof` operator is very useful. It returns a string value naming the type of the operand you give it. You can try the following in the Firebug console:

>>> typeof 4

"number"

>>> typeof -6.8

"number"

>>> typeof true

"boolean"

>>> typeof 'Hello'

"string"

>>> typeof undefined

"undefined"

>>> typeof null

"object"

Note that even though null is a primitive value, typeof null is “object”. This is considered a mistake in the original design.

2.5. Numbers

There is no distinction in JavaScript between integer and floating point numbers. All numbers are represented as floating point numbers.

Let’s try a few more things with numbers:

>>> 1.2e3 + 1000

2200

1.23e3 is the exponential notation (also known as the scientific notation) for 1200.

It is the same as 1.2 x 10^3.

>>> 4 / 0

Infinity

In JavaScript, dividing by 0 does NOT result in an error.

>>> -5.39 / 0

-Infinity

The value Infinity represents all values greater than 1.79769313486231570e + 308.

>>> typeof (4/0)

"number"

>>> typeof (5.39/0)

"number"

>>> typeof Infinity

"number"

>>> 0 / 0

NaN

Dividing 0 by 0 results in NaN, which is a special value. It stands for 'not a number'. However NaN itself is of type number. Contradictory? Maybe.

NaN is also not equal to anything including itself.

>>> typeof (0/0)

"number"

>>> typeof NaN

"number"

2.6. Booleans

Booleans can only have two values: true or false.

Comparisons operators return Boolean values:

>>> 5 > 3

true

>>> 2 < =1

false

>>> 2 == 1 + 1 // this is the equality comparison operator – is 2 equal to 1 + 1?

true

>>> 2 != 1 + 1 // this is the not equal comparison operator

False

We also have the strict equality operators. We'll see how they differ from == and != shortly.

>>> 2 === 1 + 1 // equal value and equal type

true

>>> 2 !==1 + 1 // different value or different type

False

Falsy and Truthy:

The following values are all considered to be false: 0, -0, null, “” (empty string), undefined, NaN and of course false.

Any other value is considered to be true (even the string "false").

Logical operators:

JavaScript supports three logical operators: && (and), ||(or) , and !(not).

The ! (not) operator returns true if the value given to it is false and false if the value given to it is true.

Special behavior of and and or operators in JavaScript:

We are generally used to the following behavior of and and or logical operators:

The and operator is supposed to return true only if both of the values given to it are true.

The or operator is supposed to return true if either of the values given to it is true.

There is a bit more to this in JavaScript: the and and or operators do not require that their operands be boolean values.

The && (and) operator in JavaScript produces the value of the first operand if the first operand is falsy; it does not even evaluate the second operand. Otherwise it produces the value of the second operand.

Similarly the || (or) operator in JavaScript produces the value of the first operand if the first operand is truthy; it does not even evaluate the second operand. Otherwise it produces the value of the second operand.

The above distinction does not change the behavior of the operators when dealing with pure Boolean values or expressions:

```
>>> 3 > 2 && 8 < 100
true
>>> 3 > 2 && 8 < 5
false
>>> 3 != 2 || 8 >  10
true
>>> !(3 > 2 && 8 < 100)
false
```

However, when dealing with non-Booleans, that distinction is important:

```
>>> "" && 7 > 2  // the empty string is falsy so the first operand is returned
""
>>> "hello" && 5  //  “hello” is truthy so the second operand is returned
5
```

We'll see later how to use && to avoid null references.

```
>>> 700 || 2 < 0  // 700 is truthy so the first operand is returned
700
>>> 0 || 500 // 0 is falsy so the second operand is returned
500
```

This behavior is sometimes used to supply default values.

```
name = name || ‘Friend’
// when name is the empty string, the default value 'Friend' will be used.
```

Operator Precedence:

Operators with higher precedence are performed before those with lower precedence.

Of the operators we just covered, ! (not) has the highest precedence, then the comparison operators (<, ==, etc.), then the logical and (&&) and finally the logical or(||).

Again when in doubt, use parentheses.

2.7. Strings

Strings are sequences of characters (16-bit Unicode characters, to be precise).

Strings may be enclosed in single quotes as in 'Hi' or in double quotes as in “Hello”.

Single quotes allow us to embed double quotes: '"I love coffee", she said'

Double quotes allow us to embed apostrophes (single quotes): "Isn't great?"

The \ (backslash) is the escape character. It can be used to insert new lines, apostrophes, quotes, and other special characters into a string. Whenever a backslash is found inside a string, it indicates that the character after it has a special meaning.

To see how this works, let's use the function console.log() to display output on the console.

From the Firebug console, try the following:

```
>>> console.log('Hello World')
```

Hello World

We can embed single quotes inside a single quoted string by using the escape character \'. The quote, now preceded by a backslash, will not end the string, but be part of it.

```
>>> console.log('Isn\'t great?')
```

Isn't great?

When an n character occurs after a backslash, it is interpreted as a new line.

```
 >>> console.log("first line\nsecond line")
```

first line

second line

Similarly \t is the control character for a tab.

```
>>> console.log('Foothill\tJavaScript for Programmers')
```

Foothill JavaScript for Programmers

Now what if we want to print an actual backslash? We'll need to escape it with another backslash. Only one will be printed.

```
>>> console.log("\\")
```

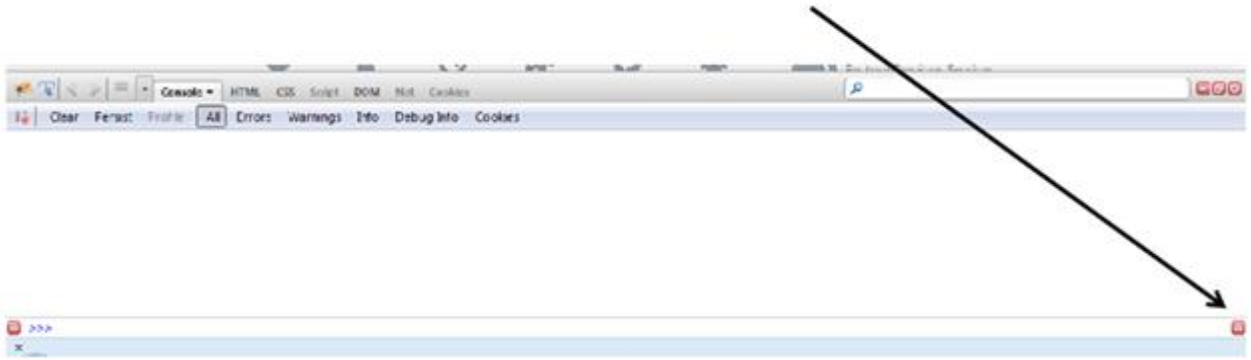
\

Multi-line Strings:

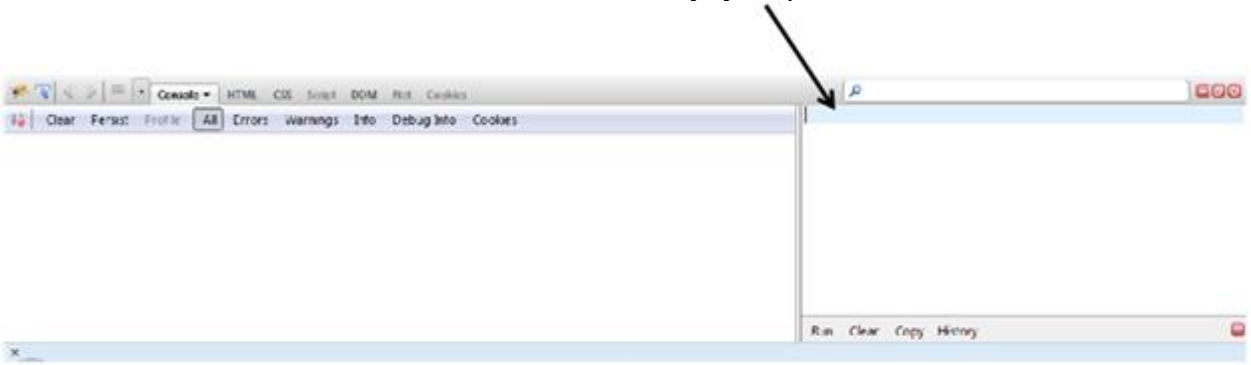
In ECMAScript 3, strings must be written on a single line.

In ECMAScript 5, you can break a string across multiple lines by ending each line but the last with a backslash (\). The backslashes and the new line characters that follow are NOT part of the string literal.

To see how that works, let's start by expanding the command line in Firebug so that we can enter multi-line segments of code. We do that by pressing on the button in the lower right corner with a triangle on it:

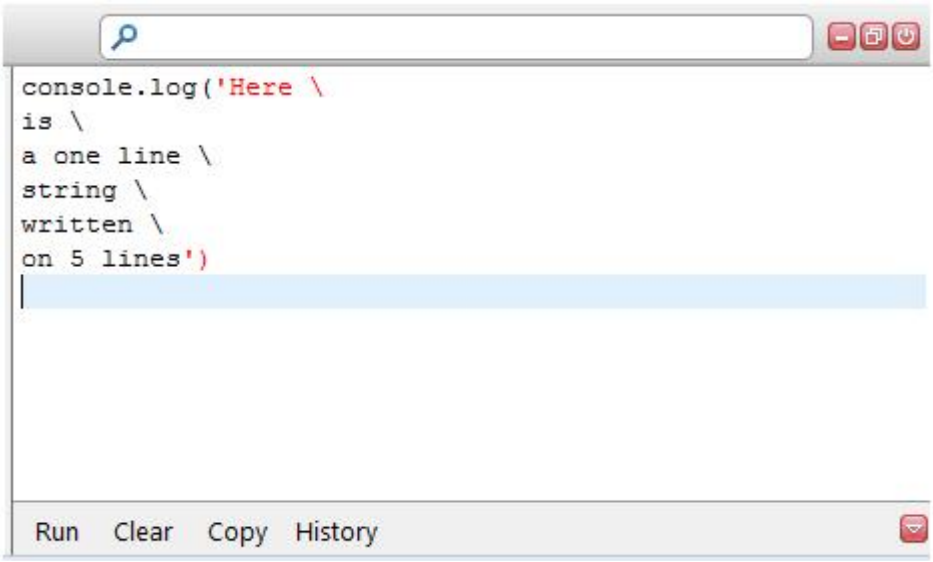


The window is then split vertically and you can enter multi-line commands on the right. To execute several lines of code entered that way, you press Run.



Let's type the following:

```
console.log('Here\
is \
a one line \
string \
written \
on 5 lines')
```



After you press Run, the following will appear on the console:

>>> console.log('Here \ is \ a one line \ string \ written \ on 5 lines')

Here is a one line string written on 5 lines

String length:

Strings have a length property.

>>> 'Foothill'.length

8

An empty string has a length of 0.

>>> "".length

0

Character Access:

There are two ways to access an individual character in a string.

Both ways use zero-based indexing: the first character is at position 0, the second at position 1 and so on.

charAt:

>>> 'Foothill'.charAt(0)

"F"

>>> 'Foothill'.charAt(4)

"h"

>>> 'Foothill'.charAt(7)

"l"

>>> 'Foothill'.charAt(8)

""

The square bracket notation: this method was introduced in ECMAScript 5:

>>> 'Foothill'[0]

"F"

>>> 'Foothill'[5]

"i"

>>> 'Foothill'[8]

undefined

>>> 'Foothill'[-1]

undefined

Note that strings are immutable. **Changing individual characters inside a string is not possible.**

Concatenating Strings:

You can make a new string by concatenating other strings together with the + operator.

>>> "Hello " + "World" +"!!!"

"Hello World!!!"

Note that this just puts the strings together. It does not add any space in between. If you need a space character, you need to include it.

Other useful string methods:

>>> 'Foothill'.**toUpperCase**()

"FOOTHILL"

>>> 'Foothill'.**toLowerCase**()

"foothill"

Remember that strings are immutable in JavaScript. Methods like toUpperCase() and toLowerCase() return new strings: they do not modify the string on which they are invoked.

>>> 'Foothill'.**indexOf**('hi') // returns the index of the first ‘hi’

4

>>> 'Foothill'.**indexOf**('o', 2); // returns the index of the first ‘o’ starting at position 2

2

>>> 'Foothill'.**indexOf**('hello') // returns -1 if the specified text is not found

-1

>>> 'Foothill'.**lastIndexOf**('o') // returns the index of the last ‘o’

2

>>> 'Foothill'.**search**('hi') // returns the index of the match if the search is successful

4

>>> 'Foothill'.**search**('Hi') // returns -1 if the search fails

-1

>>> 'Welcome to CS 21A!'.**replace**('21A', '22A') // replace 21A by 22A

"Welcome to CS 22A!"

The **substring()** method returns a subset of a string starting at one index and up to but not including the other.

```
>>>  'Foothill'.substring(4,6)
"hi"
```

If the second index is omitted, it defaults to the string length.

```
>>> 'Foothill'.substring(4)
"hill"
```

The substr() method is NOT the same as substring(). The substr() method returns the characters in a string beginning at the specified index through the **specified number of characters**.

Another difference is that substr() lets you specify a negative index, to start from the end of the string whereas substring does not.

```
>>> 'Foothill'.substr(4,2)  // returns 2 characters, starting at position 4
"hi"
>>> 'Foothill'.substr(4,4)  // returns 4 characters, starting at position 4
"hill"
```

The following will return 3 characters, starting at position -3 which is the 3rd character from the end.

```
>>> 'Foothill'.substr(-3,3)
"ill"
```

The **split()** method breaks a string into an array of strings based on the separator specified.

```
>>>  'JavaScript for Programmers'.split(' ') // split on space character
["JavaScript", "for", "Programmers"]
```

If the separator is omitted, the array returned contains one element consisting of the entire string. This is different than Python where the separator defaults to the space character.

```
>>> 'JavaScript for Programmers'.split()
["JavaScript for Programmers"]
```

If the separator is an empty string, you get an array of characters.

```
>>>  'JavaScript'.split('') //split on empty string
["J", "a", "v", "a", "S", "c", "r", "i", "p", "t"]
```

Comparing strings:

Two strings containing exactly the same characters in the same order are considered to be the same string.

```
>>> 'Foot'+ 'hill' == 'Foothill'
true
```

You can also use the comparison operators < > on strings. The comparison is of course based on the alphabetical sort. Upper case letters come before lower case ones.

```
>>> 'Alice' < 'Bob'
true
>>> 'Charlie' <= 'Bob'
false
>>> 'Charlie' >= 'Bob'
true
>>> 'Charlie' >= 'Charlie'
true
>>> 'alice' < 'Alice'
false
>>> 'Z' < 'a'
true
```

2.8. Strict Mode

Strict mode is a new feature in ECMAScript 5 that lets us impose stricter rules on a program, or a function. This strict context prevents certain previously allowed but unsafe actions from being taken. It provides stronger error checking and throws exceptions where errors used to pass silently. It also disables some 'features' of the language that are confusing or deficient.

To enable strict mode, we just add the following string at the top of the program or inside a function:

```
"use strict";
```

When covering various topics in this course, we will highlight the differences between strict mode and non-strict mode as they arise.

2.9. Variables

In JavaScript, variable names must begin with a letter, a $ character or an underscore.

Subsequent characters can be letters, digits, underscores, or $.

grade, grade1, $grade and _grade are all valid variable names. However 1grade and #grade are NOT.

The convention is to start all variable names in JavaScript with lower case letters.

Before you use a variable in a JavaScript program, you should declare it.

You declare JavaScript variables with the **var** keyword:

var grade; //semicolons are used to separate JavaScript statements

After the declaration, the variable has no value. If you try to access it, it has the value undefined.

grade;

undefined

To assign a value to the variable, use the = sign:

grade = 100;

You can also combine the declaration and the assignment in one step:

var grade = 100;

It's a good programming practice to declare all the variables you will need, in one place, at the beginning of your code.

You can declare many variables in one statement. Just start the statement with **var** and separate the variables by comma:

var homework = 100, midterm = 90, final = 95;

The declaration can also span multiple lines:

var homework = 100,

midterm = 90,

final = 95;

If you re-declare a JavaScript variable, it will not lose its value:

var grade = 95;

var grade;

grade;

95

JavaScript variables do not have a declared type: you can assign a value of any type to a variable, and you can later assign a value of a different type to the same variable

var grade = 100;

grade = ‘A’;

grade = true;

If you attempt to read the value of an undeclared variable, you get an error.

friend;

ReferenceError: friend is not defined

friend

However you may be able to assign a value to an undeclared variable **in non-strict mode**.

This is a source of many bugs and **you should always declare your variables whether running in strict mode or not.**

2.10. Semicolons, White Space and Line Breaks

Semicolons are used to separate JavaScript statements.

To try the following examples, make sure you expand the command line in Firebug first so that you can enter multi-line segments of code.

We usually add a semicolon at the end of each executable statement.

```
grade = 100;

name = 'Alice';
```

Using semicolons also makes it possible to write many statements on one line.

```
grade = 100;  name = 'Alice';
```

However it is best to put at most one statement on a line.

Optional Semicolons?

In the previous sections, we have seen examples with and without semicolons: that's because ending statements with semicolon is optional in JavaScript.

However JavaScript has some complicated rules as to when the semicolons can be safely omitted. It usually treats line breaks as semicolons only if it can't parse the code without the semicolons. But there are exceptions to this rule. One of these exceptions is that JavaScript will always interpret a line break after a 'return' as a semicolon.

So to be safe, it is best to always use semicolons at the end of your statements.

White Space:

JavaScript ignores extra spaces. You can add space characters or indentation to your code to make it more readable. The following lines are equivalent:

```
var grade=100;

var      grade  =  100;
```

Multiline Statements:

We have seen how you can use a backslash inside a string (in ECMAScript 5) to write a single string on multiple lines.

```
console.log('Here\

is \

a one line \

string \
```

written \

on 5 lines')

However, you cannot break up a JavaScript statement with a backslash.

If a statement does not fit on a single line, it is best to break it after a comma or a binary operator. Remember that JavaScript usually treats line breaks as semicolons only if it can't parse the code without the semicolons. It is also a good practice to indent the remainder of the statement for readability.

```
grade = 0.7 * 95 +
        0.15 * 100 +
        0.17 * 92;
```

However, **never break a return statement after the 'return'**. If a return statement returns a value, that value expression must begin on the same line as the return. Otherwise JavaScript will insert a semicolon right after the return and the statement will just return undefined.

```
return
true
```

is NOT the same as:

```
return true;
```

2.11. Type Conversions

Implicit Conversions:

JavaScript performs many type conversions implicitly. We've already seen how JavaScript converts non Booleans to true and false values as needed.

The same is true for other types: JavaScript will convert numbers to strings and strings to numbers without your explicit consent.

Addition and Concatenation:

If the operands are numbers or Booleans, then JavaScript will add them. Otherwise, JavaScript will convert ALL operands to strings and concatenate them:

```
>>> 10 + 5 + "$";
"15$"
```

10 and 5 are added first then 15 is converted to string and concatenated with "$".

```
>>> "$" + 10 + 5;
"$105"
```

"$" and 10 are concatenated after 10 is converted to a string. Then the result is concatenated with "5".

```
>>> true + 5
```

6

The Boolean true is converted to a number (1) then added to 5

```
>>> true + 'hello'
"truehello"
```

Here the Boolean true is converted to string first then concatenated with ‘hello’.

Multiplication, Subtraction, Division:

JavaScript will try to convert the operands to a number, or NaN if that is not possible:

```
>>>  '    5' * '8.2'
41
>>>5 * 'hello'
NaN
>>>'5' /'hi'
NaN
>>> false – 7  // false is converted to 0
-7
>>> true  * 4  // true is converted to 1
4
>>> ''  –  8  // the empty string is converted to 0
-8
```

Equality:

Because JavaScript converts values implicitly, the == equality operator does not always behave as you would expect.

```
>>> '' == 0  // empty string  converts to number 0 before comparison
true
>>> "0" == 0   // string “0” converts to number 0 before comparison.
true
>>> 0 == false  // Boolean False converts to number 0 before comparison.
true
>>> "0" == false // Both operands convert to numbers before comparison.
true
null == undefined
true
```

When this is an issue, you can use the strict equality operator instead (=== and the corresponding ! ==).

>>> 0 === false

false

>>> "0" === false

false

>>> "" === 0

false

>>> "" !== 0

true

>>> 5 - 5 === 0

true

>>> 'foot' + 'hill' === 'foothill'

True

Explicit Type Conversions:

To carry out explicit conversions, you can use Number(), String() and Boolean():

Number(value**)** - Converts the given value to a number. If the conversion is not possible, it returns NaN.

>>> Number(" .98")

0.98

>>> Number("Hello")

NaN

>>> Number("2.98e3")

2980

>>> Number("-2")

-2

String(value**)** - Converts the given value to a string.

>>> String (-2.7e3)

"-2700"

>>> String(false)

"false"

>>> String(15 + 20)

"35"

Boolean(value**)** - Converts the given value to a Boolean.

```
>>> Boolean(6.9)
true
>>> Boolean('False')
true
>>> Boolean('')
false
```

3. JavaScript Control Structures

3.1. Conditional Statements

if Statement:

The general syntax of an if statement is:

```
if (condition) {
    Statements to be executed if condition is truthy
}
```

The then block is executed if the expression is truthy. The parentheses around the condition are required. The indentation is just for readability. It does not affect the correctness of the code. The curly braces delimit the block of code to be executed.

It is recommended to always delimit the block of code with curly braces {} even if it is just one statement. It is just more readable and less error prone. Because of the implicit semicolon insertion, it is also recommended to **always start the curly braces on the same line as whatever they're opening.**

Example:

```
var balance, amount;
if  (balance >= amount) {
   balance = balance - amount;
}
```

if else Statement:

The general syntax of an if else statement is:

```
if  (condition) {
   Statements to be executed if condition is truthy
} else {
    Statements to be executed if condition is not truthy
}
```

Example:

```
var number, absolute ;
 if  (number >= 0) {
   absolute = number;
} else {
   absolute = - number;
}
```

Conditional Operator:

The conditional operator ? : may sometimes be used to achieve the same result as an if statement. The general syntax is as follows:

condition ? operand1 : operand2

operand1 is returned if the condition is truthy and operand2 is returned if the condition is falsy.

Example:

```
number >= 0 ? number : -number  // return the absolute value of number
```

Multiple if else if Statements:

Sometimes there is more than one condition that we need to check. We can use multiple if … else if… statements as follows:

```
if (condition 1) {
    // Execute code block #1
}
else if (condition 2) {
    // Execute code block #2
}
else if (condition 3) {
    // Execute code block #3
}
else {
    // If all else fails, execute block #4
 }
```

Example:

```
var grade, letterGrade;
if (grade >= 90) {
```

```
    letterGrade = 'A';
}
else if (grade >= 80) {
    letterGrade = 'B';
}
else if (grade >= 70) {
  letterGrade = 'C';
}
else if (grade >= 60) {
    letterGrade = 'D';
}
else {
    letterGrade = 'F';
}
```

3.2. The Switch Statement

The switch statement is used to perform a multiway branch based on the value of a given expression. The expression can produce a number or a string. The expression is compared to the values specified in each case clause. If there is a match, execution starts at the block of code associated with that case. If there is no match, the optional default statements are executed.

Use break to prevent the code from falling through into the next case. When using **switch inside a function, use a return** statement instead of a break statement.

Syntax:

```
switch (expression) {
case  value1:
   execute code block 1
   break;
case value2:
   execute code block 2
   break;
default:
   code to be executed if the expression is different from value1 and value2
}
```

Example:

```
var grade, gpa;
switch (grade) {
case "A":
   gpa = 4;
   break;
case "B":
   gpa = 3;
   break;
case "C":
   gpa = 2;
   break;
case "D":
   gpa = 1;
   break;
default:
   gpa = 0;
}
```

3.3. While Loops

The general syntax of a while statement is as follows:

```
while (expression) {
   block of code to execute
}
```

The expression is first evaluated. If it is falsy, then the block of code that follows is skipped completely. If the expression is truthy, then the block of code is executed repeatedly as long as the expression is truthy.

Example 1:

```
while (true) {
   console.log('Ha');
}
console.log('Bye');
```

Do not try that. This is an example of an infinite loop. The expression always evaluates to true. The loop does not terminate. It never gets to print (‘Bye’).

Example 2:

```
while (false) {
   console.log('This will never get printed');
}
console.log('Bye');
```

Bye

In this case the statement inside the loop never gets executed because the condition is always false.

Example 3:

```
var counter = 1;
while (counter < 10) {
   console.log(counter)
}
```

Here's another infinite loop.

What is missing? counter is always 1. We need to update counter inside the loop.

This is a very common mistake. **Remember to update the variable used in the while condition.**

Example 4:

```
var counter = 1;
while (counter <= 10) {
   console.log(counter++)  //  this is the increment operator.  More details next.
}
```

1

2

3

4

5

6

7

8

9

10

3.4. Increment and Decrement Operators

The increment operator is frequently used in loops.

In general, ++ is equivalent to adding 1 to the given variable. However there is a distinction between the post-increment and pre-increment operators.

Let's illustrate that difference with an example.

```
var counter = 1, result;

result = counter++;

console.log('counter:', counter);

console.log('result:', result);
```

```
counter: 2

result: 1
```

With post-increment (counter++), result is assigned the current value of counter, 1, then counter is incremented.

```
var counter = 1, result;

result = ++counter ;

console.log('counter:', counter);

console.log('result:', result);
```

```
counter: 2

result: 2
```

With pre-increment (++counter), counter is incremented first then result is assigned the incremented value, 2.

The decrement operator is similar. Here are two examples with post-decrement and pre-decrement:

```
var counter = 1, result;

result = counter—

console.log('counter:', counter);

console.log('result:', result);
```

```
counter: 0

result: 1
```

```
var counter = 1, result;

result = --counter;

console.log('counter:', counter);

console.log('result:', result);
```

```
counter: 0

result: 0
```

In a while loop, using the pre-increment or post-increment operators yield different results:

```
var counter = 1;
while (counter <= 3) {
   console.log(counter++);
}
```

```
1
2
3
```

```
var counter = 1;
while (counter <= 3) {
   console.log(++counter);
```

```
2
3
4
```

More Shorthand Operators:

JavaScript has some more shorthand operators that you'll also see in loops:

*=, /=, %=, += and -=.

```
var counter = 10;
counter -= 2;  // shorthand for counter = counter - 2
```

```
8
```

```
var counter = 10;
counter += 2;  // shorthand for counter = counter + 2
```

```
12
```

```
var counter = 10;
counter *= 2; // shorthand for counter = counter * 2
```

```
20
```

```
var counter = 10;
counter /= 2 ;  // shorthand for counter = counter / 2
```

```
5
```

```
var counter = 10;
counter %= 2;   // shorthand for counter = counter % 2
```

0

3.5. do ... while ... Statements

The general syntax of a do … while... statement is as follows:

```
do {
   block of code to execute
} while (expression);
```

The do … while loop is similar to the while loop, except that expression is tested at the bottom of the loop rather than at the top. This means that **the body of the loop is executed at least once.**

Note that **the do loop must always be terminated with a semicolon.**

Examples :

```
var counter = 4;
do {
  console.log(counter++)
} while (counter < 3);
```

4

The body of the loop is executed even though the condition is false from the beginning.

```
var counter = 1;
do {
  console.log(counter++)
} while (counter < 3);
```

1

2

3.6. for Statements

A for loop allows you to put the initialization, the test, and the update of the loop variable on a single line.

The general syntax is as follows:

```
for ( initialize ; test ; update) {
   Block of code to be executed
}
```

initialize is executed before the loop starts.

test defines the condition for running the loop.

update is executed each time after the loop (the code block) has been executed.

The advantage of a for loop over the equivalent while loop is that it prevents bugs due to forgetting to initialize or increment the loop variable.

Example:

```
for ( var counter  =  1; counter  < 5; counter++) {
   console.log(counter);
}
```

```
1
2
3
4
```

Note that **the parentheses after the for should always contain two semicolons,** even if you omit the initialization, test or update.

So if counter is initialized in previous statements, it would be OK to write:

```
for ( ; counter  < 5; counter++) {
   console.log(counter);
}
```

Note that **omitting the test would result in an infinite loop unless you include a break statement** inside the loop. **The break statement causes the innermost enclosing loop to exit immediately.**

4. Functions

4.1. Function Declarations, Parameters & Arguments

A function is a block of code that is defined once and is executed every time you call it.

To create a function, you use the 'function' keyword.

When the function keyword is followed by a name, the function will be stored under this name. JavaScript does not require functions to have a name. If a function is not given a name, it is said to be anonymous. If specified, the function name must begin with a letter, a $ character or an underscore. Subsequent characters can be letters, digits, underscores, or $. **The convention is to start all function names with lower case letters.** When a name includes multiple words, the convention is to begin all words after the first with an uppercase letter longFunctionName(). Functions that are supposed to be internal or hidden are given names that begin with an underscore.

After the name comes a list of parameter names between parentheses, and then finally the body of the function. The body of a function has to be enclosed in curly braces {}.

Example:

Below is a **function declaration statement** for a named function hello.

```
function hello(name) {
   name = name || 'Friend';
   console.log('Hello', name);
}
```

The hello function takes one parameter: name. If that name is the empty string, it assigns to it the default value 'Friend'. Remember that the or operator || returns the second operand if the first operand is falsy. The empty string is falsy.

The function hello then outputs a hello message to the console.

Once we define and name our function, we can call it with different values for the parameter name.

```
hello('Class');
hello('Alice');
hello('');
```

Parameters are the variables in the function and arguments are the values given to the variables at the point of call.

So outside the function, it is more common to talk about arguments.

In the function calls above, 'Class', 'Alice' and '' are the arguments to our function. They define the value that will be assigned to the parameter, name, inside the function definition.

After the 3 function calls above, the output will be:

Hello Class

Hello Alice

Hello Friend

4.2. Parameter Checking?

JavaScript does not generate any error when the number of arguments and the number of parameters do not match. If there are too many argument values, the extra argument values will be ignored. If there are too few argument values, the undefined value will be substituted for the missing values. There is no type checking on the argument values either: any type of value can be passed to any parameter.

Going back to our previous function hello, we can call it with 3 arguments as follows:

```
function hello(name) {
   name = name || 'Friend';
   console.log('Hello', name);
}
```

hello('Alice', 'Bob', 'Charlie');

Hello Alice

The arguments ‘Bob’ and ‘Charlie’ are simply ignored.

We can also call it with no argument as follows:

hello();

Hello Friend

JavaScript substitutes undefined for the parameter ‘name’. Since undefined is falsy, inside the function name gets assigned name || ‘Friend’ which is ‘Friend’.

Finally, we can call hello with numeric or Boolean arguments without generating any error.

hello(6);

hello(true);

Hello 6

Hello true

4.3. Return Value

In JavaScript, a function always has a return value.

If the function includes a return statement, that statement causes the function to stop executing and to return the value of its expression (if any) to the caller. If the return statement does not have an associated expression, it returns the undefined value.

If a function does not contain a return statement, it executes the code and returns undefined.

Example 1:

The function longest below returns the longest of two given words. The return value (longest word) can then be assigned to a variable, and accessed from outside the function.

```
function longest(word1, word2){

    if (word1.length > word2.length){

        return (word1);

    } else {

        return (word2);

    }
```

```
}

var name1 = longest('hi', 'hello');

var name2 = longest('JavaScript', 'Python');

var name3 = longest ('', 'Alice');

console.log(name1, name2, name3);
```

hello JavaScript Alice

Example 2:

In the previous section example, the function hello did not explicitly return anything. It just wrote a message to the console. If we try to display its return value, we'll see that it is undefined.

```
function hello(name){
   name = name || 'Friend';
   console.log('Hello', name);
}
console.log(hello('Alice'));
```

Hello Alice

undefined

4.4. Recursion

A function can also call itself. If it does, we call it a recursive function. Here's an example of a recursive function that computes the factorial of a number.

The factorial of a number is the product of all numbers up to and including that number.

factorial(3) = 1 x 2 x 3 = 6

factorial (4) = 1 x 2 x 3 x 4 = 24

Note that:

factorial(1) = 1

and

factorial(n) = factorial(n – 1) x n

Those 2 observations allow us to write the following recursive definition of factorial:

```
function factorial(number) {
        if (number <= 1) {
              return 1;
```

```
    } else{
        return factorial(number - 1) * number;
    }
}

factorial(5)
```

120

4.5. Function Definition Expressions

The function declaration statements that we have seen so far can only appear as top-level statements. They can appear in global code, or within other functions, but they cannot appear inside loops or conditionals.

However functions can also be defined in **function definition expressions.** These may appear anywhere that an expression can appear. In the example below a function is created with the function keyword and assigned to a variable, opposite. opposite is now a function and can be called with an argument.

```
var opposite = function(number) {
    return -number;
}
opposite(10);
```

-10

Note that you can call the typeof operator on the variable opposite and JavaScript will recognize it as a function.

```
typeof(opposite);
```

"function"

4.6. Nested Functions

Functions may be nested within other functions. Here's an example:

```
var variable = "global variable";
function outer() {
  var variable = "local variable";
  function inner() {
    return variable;  // inner has access to the variable defined in outer
  }
  return inner;   // A function may return another function
}
var innerFunction = outer();  // outer returns a function, so innerFunction is a  function.
innerFunction();
```

local variable

Nested functions can access the parameters and variables of the function they are nested within. However, the outer function does not have access to the variables and functions defined inside the inner function. This provides a sort of security for the variables of the inner function. We'll look more closely into the scope rules and closures in the next sections.

4.7. Variable Scopes

Hoisting:

All variables declared within a function are visible throughout the body of the function. It is as if all variable declarations in a function (but not the associated assignments) are "hoisted" to the top of the function.

Example:

myVar = "first"

var myVar;

is interpreted as:

var myVar;

myVar = "first"

Local JavaScript Variables

A variable declared (using var) within a function is a local variable and can only be accessed from within that function.

You can have local variables with the same name in different functions, because local variables are only recognized by the function in which they are declared and by its nested functions.

Local variables are deleted as soon as the function is completed.

Global JavaScript Variables

Variables declared outside a function, are global variables, and all code (inside or outside the function) can access them.

The lifetime of JavaScript variables starts when they are declared.

Global variables are deleted when you close the web page or exit the program.

Assigning Values to Undeclared JavaScript Variables

If you assign a value to a variable that has not been declared, **the outcome will depend on whether or not you are using strict mode.**

In strict mode, you'll get an exception.

In non-strict mode, the variable will automatically be declared as a global variable. To avoid this, **it is important to declare all variables that will be used inside a function at the beginning of that function.**

4.8. Closures

When functions are nested in JavaScript, the inner function has access to the scope of the outer function. As a result, the variables defined in the outer function will live longer than the outer function itself, if the inner function manages to survive beyond the life of the outer function. A closure is created when the inner function is somehow made available to any scope outside the outer function.

Here's an example:

```
function counter() {
   var number = 0;
   function increment() {
      return ++number;
   }
   return increment;
}
```

The function counter defines and initializes a number and then returns another function, increment. That inner function increment has access to the number initialized.

The variable, number, now survives outside of the function counter. Now we can keep two independent counts as follows:

```
var count1 = counter();
var count2 = counter();
console.log(count1());   //1
console.log(count1());   // 2
console.log(count2());   // 1
console.log(count1());   // 3
console.log(count2());   // 2
console.log(count1());   // 4
```

The result is:

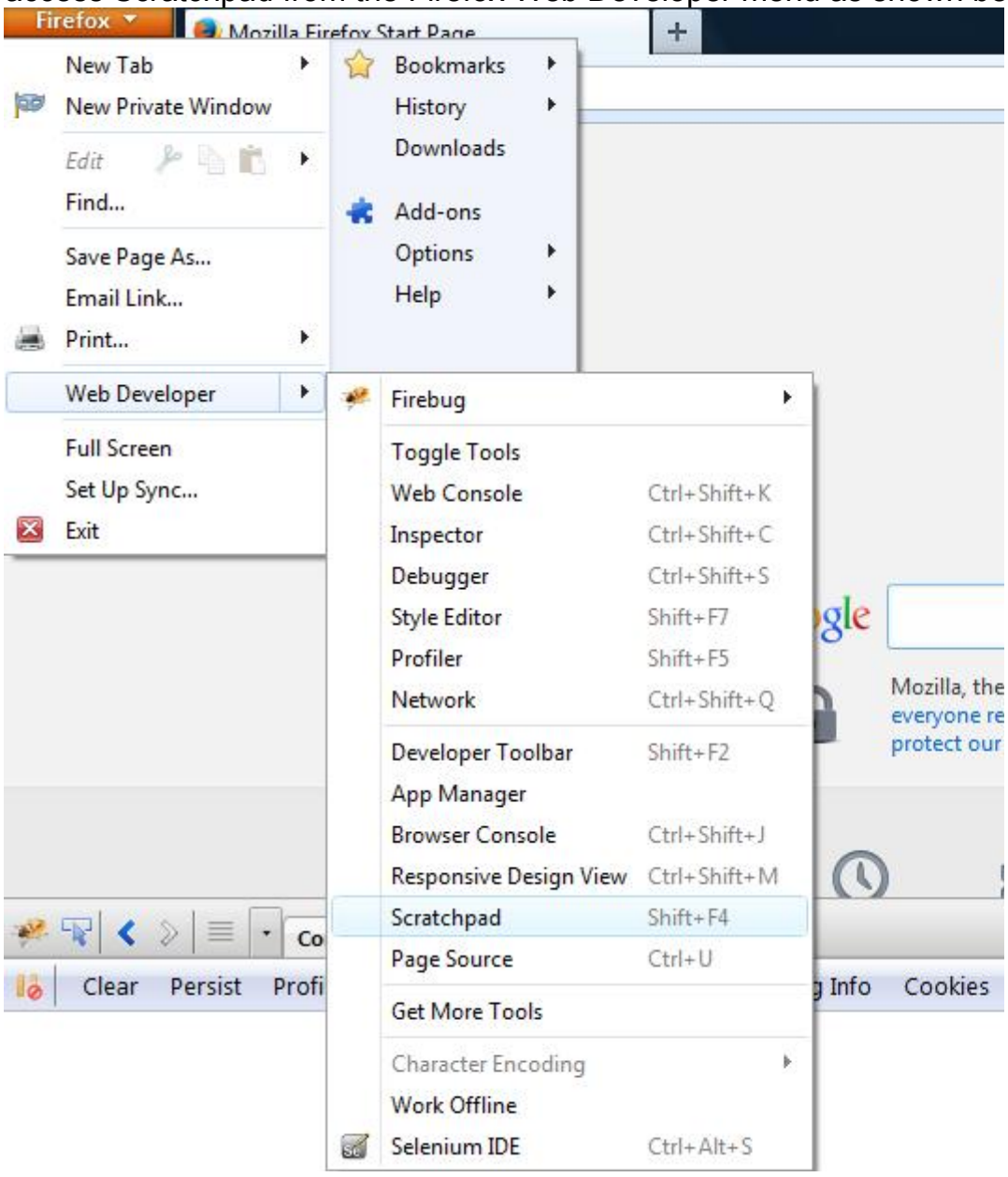
```
1
2
1
3
2
4
```

4.9. Scratchpad

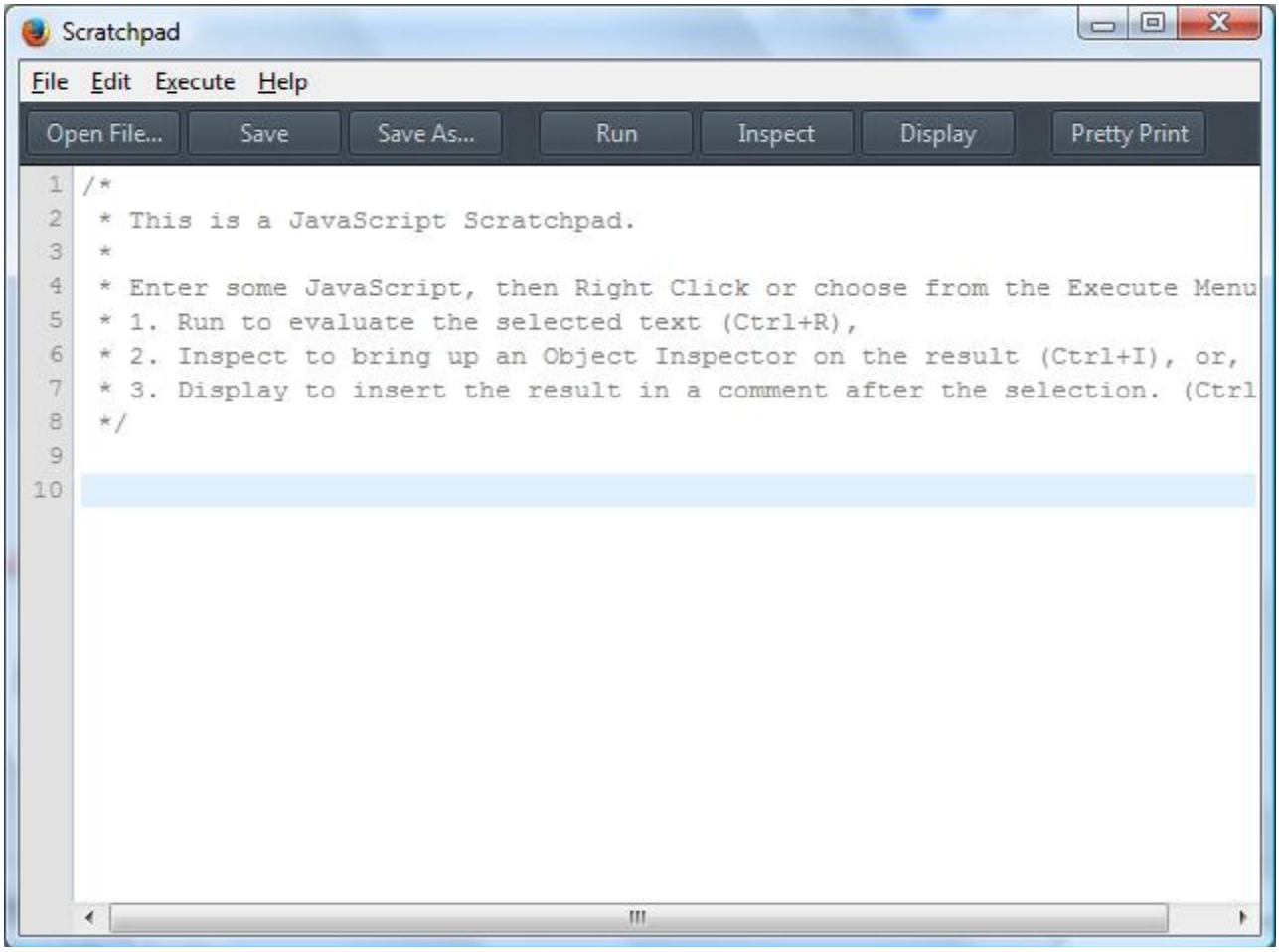
Scratchpad is another tool available from Firefox that may come in handy when you are developing JavaScript code. Sratchpad is basically a text editor that knows how to run JavaScript.

So far we have used Firebug to execute several lines of JavaScript code. Scratchpad will allow us to save that code in a file, come back to edit it and so on.

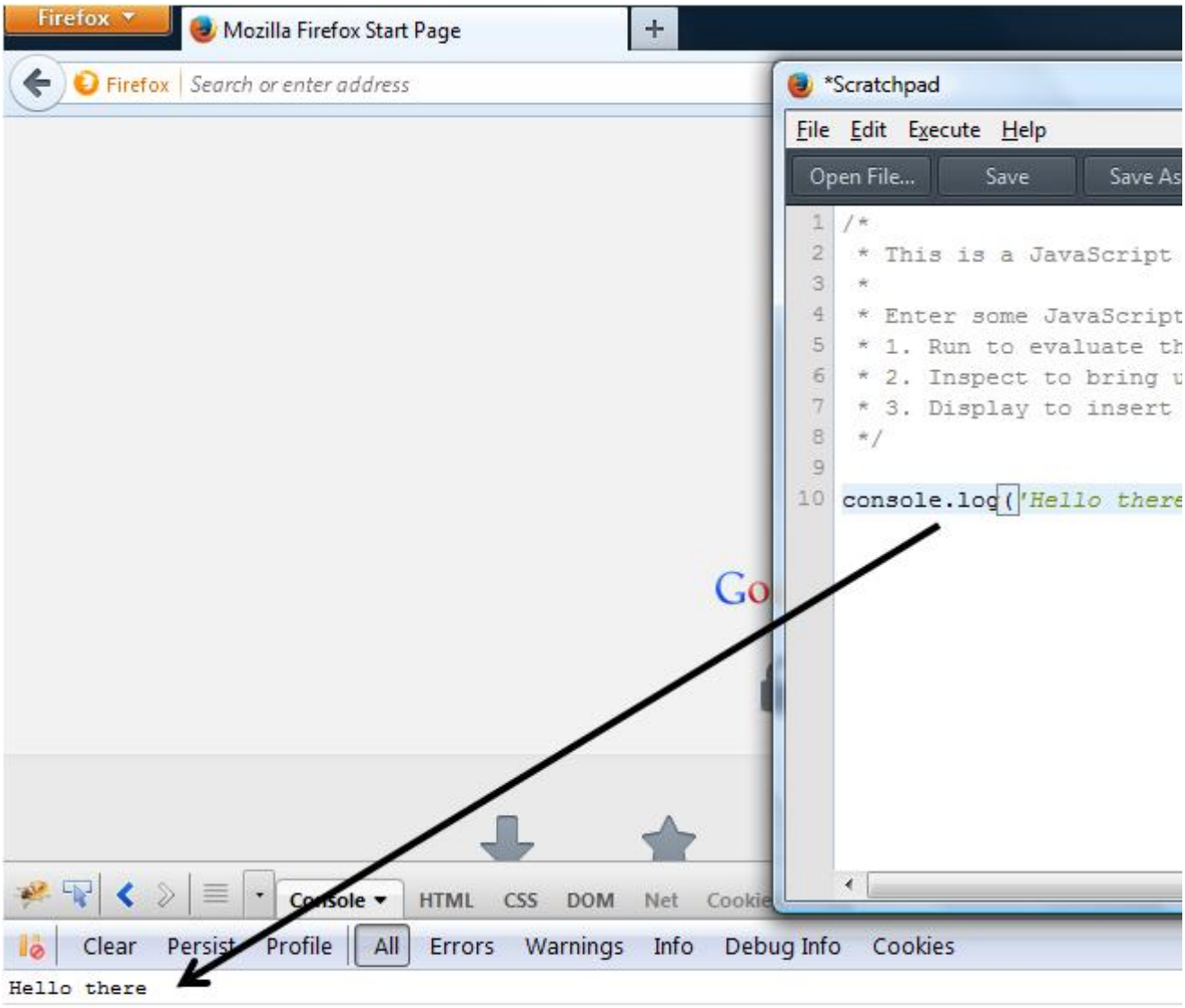
You can access Scratchpad from the Firefox Web Developer menu as shown below:



The Scratchpad window will open:



You may use Scratchpad to write, test and save the functions you write for the guessing game assignment (save them in game.js). Just be aware that when you use console.log to display your results, the output will appear in the Firebug console, not in the Scratchpad file.



For complete instructions on how to use Scratchpad, go to:

https://developer.mozilla.org/en-US/docs/Tools/Scratchpad

5. Objects

5.1. What is an object?

Almost everything in JavaScript is an Object: Functions, Arrays, even primitive datatypes such as strings can be treated as objects.

We can think of an object as an **unordered collection of properties**, each of which has a name and a value.

We have already seen the length property of a string:

var greeting = 'Hello';

greeting.length;

5

greeting["length"]; // another way to access the length property

5

There are 2 syntaxes for accessing the property of an object:

objectName.propertyName

or

objectName[“propertyName”]

Objects also have a certain behavior associated with them. This behavior is described by the methods that can be invoked on the objects.

We have already seen how to invoke the toLowerCase() method on a string:

var greeting = 'Hello';

var newGreeting = greeting.toLowerCase();

newGreeting;

"hello"

The general syntax for invoking a method on an object is:

objectName.methodName()

5.2. Creating JavaScript Objects

We can define and create our own objects. There are 3 ways to do that.

We can simply use an object literal. An object literal is a list of colon-separated name:value pairs, enclosed within curly braces.

For the Python programmers out there, it looks like a dictionary.

Example:

var hisCar={make:"Honda", model:"Civic"};

A property name is a valid JavaScript identifier or a string literal (the empty string is allowed).

A property value is any JavaScript expression.

The following examples are all valid objects:

var emptyObject = {}

This is just an empty object with no properties.

var herAccount = {'account holder': 'Alice', balance: 300};

The property name ‘account holder’ includes a space, so it is not a valid JavaScript identifier but it still can be a property name: we just enclose it in quotes.

Once we have created the object, we can access it and access its properties:

console.log(herAccount)

Object { account holder="Alice", balance=300}

console.log(herAccount.balance)

300

console.log(herAccount['account holder'])

Alice

The account holder property includes a space so we can't access it with the dot notation but we can always access it with the square bracket notation.

var person = {idNo: 8989000, account: herAccount}

Objects may also be nested in other objects. The value of the account property of person is another object.

console.log(person)

Object { idNo=8989000, account={...}}

console.log(person.account)

Object { account holder="Alice", balance=300}

5.3. Prototype

Every object is linked to a prototype object from which it can inherit properties.

When you create a new object from object literals ({property:value, …}), you don't have the option to specify the prototype.

All objects created from object literals have the same prototype, Object.prototype, an object that comes standard with JavaScript. It is denoted by Object {}.

In ECMAScript 5, you can get the prototype of any object by passing that object to Object.getPrototypeOf().

var herAccount = {'account holder': 'Alice', balance: 300};

console.log(Object.getPrototypeOf(herAccount))

Object {}

var person = {idNo: 8989000, account:herAccount}

console.log(Object.getPrototypeOf(person))

Object {}

Object.prototype is one of the rare objects that has no prototype: it does not inherit any properties.

console.log(Object.getPrototypeOf(Object.prototype))

null

We'll see that there are 2 other ways to create objects and these 2 ways let you specify the prototype.

5.4. Creating Objects with Object.create()

In ECMAScript 5, you can use the method Object.create() to create a new object. You specify the prototype for the new object as the first argument to the method. **This is the recommended way to create objects in JavaScript.**

Object.create() also takes an optional second argument that describes the properties of the new object. We'll talk about the second argument in a later section.

Here are some examples on how to use Object.create():

```
var car = {mileage: 0};
```

The variable car is created using an object literal. Its prototype is automatically set to Object.prototype (denoted by Object {} below). It is equivalent to calling Object.create as follows: car = Object.create(Object.prototype) and then adding the mileage property.

```
console.log(car);
```

```
Object { mileage=0}
```

```
console.log(Object.getPrototypeOf(car));
```

```
Object {}
```

```
var hondaCar = Object.create(car);
```

The variable hondaCar is created using the Object.create() method. Its prototype is **explicitly set** to the object referred to by car (denoted by Object{mileage=0} below).

```
console.log(Object.getPrototypeOf(hondaCar));
```

```
Object { mileage=0}
```

The object referred to by hondaCar has **inherited** the mileage property from its prototype car.

```
console.log(hondaCar);
```

```
Object { mileage=0}
```

We can add a new property to hondaCar. To add a new property, you just assign a value to it.

```
hondaCar.make = 'Honda';
```

```
console.log(hondaCar);
```

```
Object { make="Honda", mileage=0}
```

This does NOT change the prototype car. **The property is only added to the object itself but not to its prototype.**

```
console.log(car);
```

```
Object { mileage=0}
```

We can create a new object myCar with hondaCar as its prototype:

var myCar = Object.create(hondaCar);

console.log(Object.getPrototypeOf(myCar));

Object { make="Honda", mileage=0}

myCar inherits ALL the properties of hondaCar, including those that hondaCar itself inherited from car:

console.log(myCar);

Object { make="Honda", mileage=0}

We can add properties to myCar without affecting its prototype.

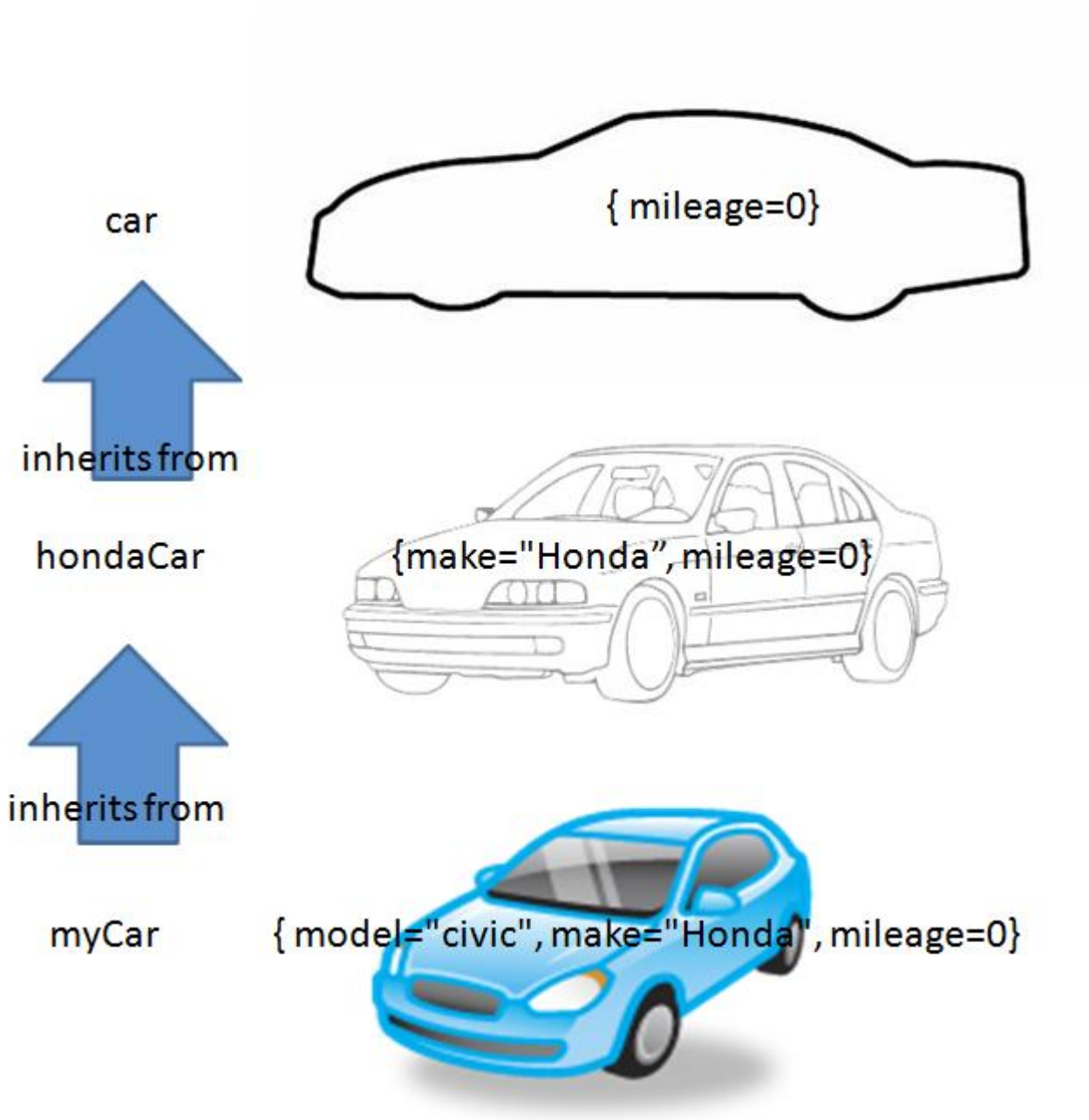
myCar.model = 'civic';

console.log(myCar);

Object { model="civic", make="Honda", mileage=0}

console.log(hondaCar);

Object { make="Honda", mileage=0}



5.5. Creating Objects with Constructors

We can also create and initialize an object with the new keyword. The new keyword must be followed by a function invocation. A function used in this way is called a **constructor.**

JavaScript includes some built-in constructors: Object, Date and Array are some of the built-in constructors.

The following example creates a new instance of an object using the Object constructor.

```
var herAccount=new Object();
```

Once the object is created, we can add properties to it:

```
herAccount.name = "Alice";
```

herAccount.balance = 300;

console.log(herAccount);

Object { name="Alice", balance=300}

Because herAccount was created with the Object constructor, its prototype is Object.prototype:

console.log(Object.getPrototypeOf(herAccount));

Object {}

We can also define our own constructor functions to initialize newly created objects.

The following example defines a constructor function to create a Car object and initialize it with the given property, make:

```
function Car(make) {
   this.make= make;
}
```

this refers to the new object being created. It is equivalent to the self parameter in Python. **It is automatically passed to the constructor. It does not have to be listed as a parameter.**

We'll see later that all methods have access to **this** – it is the object on which the method is invoked.

The convention is to **start the names of constructors with a capital letter**. This makes it easy to distinguish them from other functions.

We can also define properties and methods for the **prototype:** these properties and methods will be inherited by all **the objects created with this constructor. These properties and methods have to be defined under the constructor property 'prototype':**

Car.prototype.mileage = 0; // define a mileage property for the prototype

Now all objects created with the Car constructor will inherit the mileage property from Car.prototype.

We can also define methods for the prototype. **A method is a function assigned to the property of an object**. When a function is invoked on an object, the function has access to that object through **this**. Here we define an anonymous function and assign it to the drive property of Car.prototype. All objects created with the Car constructor will then inherit the drive method from Car.prototype.

```
Car.prototype.drive = function (distance ) {   // define a drive method for the proptotype
   this.mileage = this.mileage + distance;
}
```

Once we have defined the Car constructor, we can create new objects with the new

keyword as follows:

```
var myCar = new Car("Honda");
var yourCar = new Car("Porsche");
console.log(myCar);
```

```
Car { make="Honda", mileage=0, drive=function()}
```

```
console.log(myCar.make);
```

```
Honda
```

```
console.log(yourCar);
```

```
Car { make="Porsche", mileage=0, drive=function()}
```

```
console.log(yourCar.make);
```

```
Porsche
```

```
console.log(Object.getPrototypeOf(myCar));
```

```
Car { mileage=0, drive=function()}
```

```
console.log(Object.getPrototypeOf(yourCar));
```

```
Car { mileage=0, drive=function()}
```

Note that myCar and yourCar are 2 different objects. They have different values for their make property. They have the same prototype and they both inherited mileage and drive from it.

```
myCar.drive(10);
console.log(myCar.mileage);
```

```
10
```

```
myCar.drive(10);
console.log(myCar.mileage);
```

```
20
```

We can invoke the drive method on myCar and as a result its mileage property is updated.

5.6. More on Methods

We have seen that methods in JavaScript are just functions that are attached to objects through a property:

```
function depositAmount(amount){
   this.balance = this.balance + amount;
   return this;
}
account = {name: 'Alice', balance: 0, deposit: depositAmount};
```

```
console.log(account);
```

```
Object { name="Alice", balance=0, deposit=depositAmount()}
```

The deposit property of object account is a function that has been defined earlier and named depositAmount. It takes one parameter, amount.

We can invoke it on the account object as:

```
account.deposit(20)
```

When the function is invoked on an object as a method, it has access to the object through the keyword **this**. We say that the object becomes its **invocation context.** Note that this is passed implicitly to the method: it does not have to be listed as a parameter.

```
console.log(account);
```

```
Object { name="Alice", balance=20, deposit=depositAmount()}
```

In this example, the function was defined first, then attached to the object. We could have also defined the method directly as follows:

```
account =  {name: 'Alice',
            balance: 0,
            deposit: function(amount){
                        this.balance = this.balance + amount;
                        return this;
                    }
            };
```

Note that deposit is just a property of account, so we can also invoke it using the square bracket notation:

```
account["deposit"](50)
```

The above method returns the updated object . When you write methods that return updated objects, you can chain method invocations so that the object is operated on by several methods sequentially.

For instance, in our example, we could invoke the deposit method multiple times on account as follows:

```
account.deposit(20).deposit(100);
```

5.7. Properties and Inheritance

We have seen that we can access the properties of an object using the dot notation as well as the square bracket notation:

```
account.balance
```

```
account['balance']
```

With inheritance, things are a little more complicated as to which properties we are really accessing.

Each object has its own direct properties and it also inherits properties from the prototype object. You can use the **hasOwnProperty()** method on any object to distinguish between direct (own) and inherited properties.

Consider the following example:

```
var car = {mileage: 0};
var hondaCar = Object.create(car);
hondaCar.make = 'Honda';
var myCar = Object.create(hondaCar);
myCar.model = 'civic';
```

```
 myCar.hasOwnProperty('mileage');   //mileage is not a direct property of myCar
```

false

```
 hondaCar.hasOwnProperty('mileage'); //mileage is not a direct property of hondaCar
```

false

```
 car.hasOwnProperty('mileage'); //mileage is a direct property of car
```

true

```
>>> myCar.mileage;   // the inherited mileage property is accessible through myCar
```

0

So even though mileage is a property of car and not a direct property of myCar, I can still access it and get its (inherited) value through myCar.

In general, if the object does not have a direct property, JavaScript looks for the property in the prototype. If the prototype object does not have a direct property by that name either but has a prototype itself, then JavaScript looks for that property in the prototype of the prototype. This continues until the property is found or until an object with a null prototype is searched.

```
>>> myCar.color;
```

undefined

In this case, none of the objects in the prototype chain has a color property, so undefined is returned.

Now what happens if I assign a value to myCar.mileage?

```
>>> myCar.mileage = 10000;  // create a new property on myCar
```

10000

```
>>> car.mileage;
```

0

```
>>> hondaCar.mileage;
```

0

>>> myCar.hasOwnProperty('mileage');

true

Since myCar does not have a direct property mileage, the assignment creates a new property mileage on myCar. The mileage property inherited from car is now hidden by the newly created direct property with the same name. This mechanism allows us to selectively override inherited properties.

5.8. Prototype vs Class

JavaScript is an object-based language based on prototypes, rather than class.

All objects are instances and you construct an object hierarchy by assigning another object as its prototype.

Instead of inheriting properties by following the class chain, JavaScript objects inherit properties by following the prototype chain.

The major advantage lies in the following distinction.

The class relationship is a static relationship. A class definition specifies all properties of all instances of a class at once. We cannot add properties dynamically at run time.

The prototype relationship is a dynamic relationship. The prototype specifies an initial set of properties. We can add or remove properties dynamically to individual objects or to the prototype thus affecting an entire set of objects. When we add a new property to a prototype, that property is immediately visible in all of the objects that are based on that prototype – even those that were created prior to this addition.

6. Properties

6.1. Configurable Properties in ECMAScript 5

In EcmaScript 5, properties have attributes that specify whether they can be written, enumerated, or configured. These attributes may be set or reset with the **Object.defineProperty()** method.

When we create a property through assignment, as in myCar.color = ‘blue’, the property created is always writable (may be changed), enumerable (it shows up during property enumeration), and configurable (it may be deleted).

The Object.defineProperty() method gives us control over these extra details. We can use it to create a new property or to change the attributes of an existing property.

Consider the following example:

Object.defineProperty(myCar, 'year', { value : 2009, writable: false, enumerable: true, configurable: false});

Here we are defining a new ‘frozen’ property on the object myCar.

The value of year is set to 2009.

However that property is NOT **writable**, which means that I cannot change it with an

assignment statement.

It is **enumerable**, which means it shows up during enumeration of the properties on the corresponding object.

It is NOT **configurable**, which means its attributes cannot be changed with another call to defineProperty, and it cannot be deleted.

Let's see what all of that means in our example:

```
Object.defineProperty (myCar,
'year',
{ value : 2009,
writable: false,
enumerable: true,
configurable: false
}
    );
console.log(myCar);  // year is enumerable so it will show up with a value of 2009
Object { model="civic", year=2009, make="Honda", more...}
>>> myCar.hasOwnProperty('year');
true
```

Now let's try to change the value of year:

```
>>> myCar.year = 2010;
```

The outcome will depend on whether or not we are using strict mode.

In strict mode, we get an exception: "year" is read-only.

In non-strict mode, no error is generated but the assignment is ignored:

```
>>> myCar.year;
2009
```

Let's try to change the property to make it writable:

```
Object.defineProperty(myCar, 'year',  { value : 2009, writable: true, enumerable: true, configurable: false})
TypeError: can't redefine non-configurable property 'year'
```

That is not possible, because the property is not configurable.

When we introduced Object.create() we mentioned that it takes 2 arguments.

The first argument is the prototype for the new object and the optional second argument describes the properties of the new object.

So we could have also created the year and model properties together with the object as follows:

```
var myCar = Object.create(hondaCar, {  // hondaCar is the new object

                                        "year": { // year is a new property

                                                  value : 2009,

                                                  writable: false,

                                                  enumerable: true,

                                                  configurable: false

                                                },

                                        "model": { // model is a new property

                                                    value : "civic",

                                                    writable: true,

                                                    enumerable: true,

                                                    configurable: true

                                                  }

                                      }

                                );
```

Unless otherwise specified, in the context of this course, we'll assume the default case where properties are writable, enumerable and configurable.

6.2. Deleting Properties

To remove a configurable property from an object, we can use the delete operator. Delete will return true if the property is deleted successfully.

>>> delete myCar.model;

true

However it also returns true if the property does not exist:

>>> delete myCar.noSuchProperty;

true

The delete operator only deletes direct, own properties, not inherited ones. However it will also return true if you attempt to delete an inherited property without actually deleting it:

>>> delete myCar.make

true

>>> myCar.make

"Honda"

>>> myCar

Object { year=2009, make="Honda", mileage=0, more...}

To delete an inherited property, we'll need to delete it from the prototype object in which it is defined. It will then disappear from every object that inherits from that prototype.

>>>delete hondaCar.make

true

>>> myCar.make

undefined

In strict mode, attempting to delete a nonconfigurable property results in an exception. In non-strict mode, delete simply returns false in this case.

Note that in order to use strict mode, you need to run your code from Scratchpad. The console does not support strict mode.

Remember that we had defined the year property to be non configurable. Let's see what happen if we try to delete it:

In strict mode, we get an exception:

delete myCar.year

Exception: property "year" is non-configurable and can't be deleted

In non-strict mode, delete just returns false in and the property is NOT deleted.

delete myCar.year

false

myCar.year

2009

6.3. Testing Properties

Sometimes it is useful to determine if an object has a given property before attempting to access it.

The in operator may be used for that. It returns true if the object has a direct or inherited property with a given name.

>>> "mileage" in myCar; // mileage is an inherited property of myCar

true

>>> "model" in hondaCar // model is not a property of hondaCar

false

>>> "model" in myCar // model is a direct own property of myCar

true

We have also already seen how to use the hasOwnProperty() method to distinguish between direct (own) and inherited properties.

The propertyIsEnumerable() is even more specific. It returns true only if the named property is a direct property and it is enumerable.

6.4. Iterating Over Properties

The for...in loop allows us to iterate **over the enumerable properties** of an object.

The general syntax is as follows:

```
for (property in object) {
        do something with the property
}
```

The loop goes over the properties in **arbitrary order.**

Note that the loop will iterate over all enumerable properties of the object itself as well as those the object inherits from its prototype.

Let's go back to our car example where myCar has 2 inherited properties, mileage and make, and 3 own properties: model, year and vin. We made the vin (vehicle identification number) non-enumerable, so that it does not appear when we iterate over the properties. Inside the loop, we are building a string that will contain a description of the car. It will include all the properties (own and inherited) except the vin, since the vin property since it is non-enumerable.

```
var car = {mileage: 0};
var hondaCar = Object.create(car);
hondaCar.make = 'Honda';
var myCar = Object.create(hondaCar);
myCar.model = 'civic';
Object.defineProperty(myCar, 'year',  { value : 2009, writable: false, enumerable: true, configurable: false});
Object.defineProperty(myCar, 'vin', { value: 123456789, writable: false, enumerable: false, configurable: false});
var carDescription = "";
for (var prop in myCar) {
// add the string property name: property value followed by a new line
carDescription += prop + ": " + myCar[prop] + "\n";
}
console.log(carDescription);
```

```
model: civic
year: 2009
make: Honda
mileage: 0
```

7. Arrays

7.1. Arrays: Elements and Indexes

An array is an **ordered** collection of elements.

We can create arrays by using an array literal which is a just list of elements enclosed in square brackets.

var myArray = ["Alice", 89, true, 500.2];

Array elements need not all have the same type.

Each element has a position in the array, known as its **index**. JavaScript uses **zero-based indexing**. We can access individual elements of an array by their index as follows:

myArray[0];

"Alice"

myArray[1];

89

myArray[2];

true

myArray[3];

500.2

myArray[4];

undefined

Array elements may themselves be arrays:

var myGrades = [86, 90, [100, 85], 99];

myGrades[0];

86

myGrades[1];

90

myGrades[2];

[100, 85]

myGrades[3];

99

myGrades[4];

undefined

To access an element in the nested array, we use the square brackets twice: [ ][ ]

myGrades[2][0];

100

myGrades[2][1];

85

JavaScript arrays are dynamic: they grow or shrink as needed.

var myArray=[]; // we can start with an empty array

myArray;

[ ]

myArray[0]=2; // add a numeric element at index position 0.

myArray;

[2]

JavaScript arrays may be sparse: the elements don't need to have contiguous indexes and there may be gaps.

myArray[3] = 1; // add a numeric element at index position 3

myArray;

[2, undefined, undefined, 1]

You can also create sparse arrays by including extra commas inside the array literal:

var myArray = ['zero', , ,'three'];

myArray;

["zero", undefined, undefined, "three"]

7.2. Array Length

Arrays have a length property.

For nonsparse arrays, this property specifies the number of elements in the array.

var myNumbers = ['zero', 'one', 'two'];

myNumbers.length

3

var myGrades = [86, 90, [100, 85], 99]

myGrades.length

4

For sparse arrays, the length is one more than the largest index.

var sparseArray = [ ]; // start with an empty array

sparseArray[100] = 'Alice'; // add an element at index position 100

sparseArray.length;

101

```
var myArray = ['zero', , ,'three'];
myArray.length;
```
4

You can also create an array by using the array constructor and specifying the length:

```
var myArray = new Array(5);
myArray.length
```
5

The length is not an upper bound. If you add an element with an index that is greater than or equal to the current length, the length will increase to contain the new element. No out of bounds error is generated.

You can assign a new value to the length. Making the length larger does NOT allocate more space for the array.

```
var myArray = ['zero', , ,'three'];
myArray.length;
```
4

```
myArray.length = 6;  // assign a new larger value to the length
myArray;
```
["zero", undefined, undefined, "three", undefined, undefined]

Making the length smaller will truncate the array. The elements with an index greater than or equal to the new length will be deleted:

```
myArray.length = 1;  // assign a new smaller value to the length
myArray;
```
["zero"]

7.3. Rearranging an Array

We can rearrange the elements in an array.

We can **reverse** an array:

```
 var students= ['Alice', 'Charlie', 'Bob'];
students.reverse();
students;
```
["Bob", "Charlie", "Alice"]

The array is reversed in place. The elements are rearranged in the original array.

We can **sort** an array:

```
students.sort();
students;
```

["Alice", "Bob", "Charlie"]

The array is sorted in place. The elements are sorted in the original array. The default sort is alphabetical, even if the elements are not strings.

```
var myNumbers = [100, 6, 30];
myNumbers.sort();
```

[100, 30, 6]

In the alphabetical sort, the first character of the elements is compared and 1 comes before 3 which comes before 6.

We can specify a different sort by passing a **comparison function** to the sort method.

The elements are then sorted according to the return value of the comparison function. If a and b are the two elements being compared:

If compareFunction(a, b) is less than 0 -> a comes before b.

If compareFunction(a, b) is 0 -> leave a and b unchanged with respect to each other.

If compareFunction(a, b) is greater than 0 -> b comes before a.

So to compare numbers instead of strings, the compare function can simply return a - b:

```
function compareNumbers(a, b) {
   return a - b;
}
```

To sort the numbers in the array above, we can then write:

```
var myNumbers = [100, 6, 30];
myNumbers.sort(compareNumbers);
```

[6, 30, 100]

7.4. Splicing an Array

The splice method lets us perform complicated surgery on an array. It can delete some elements and replace them with other elements in one step.

The first argument to splice is an index in the array. The second argument is the number of elements to remove. The elements are removed from the given index on. The removed elements are also returned by the method. Any additional arguments get inserted into the array at that point:

```
var colors = ['red', 'white', 'blue'];
colors.splice(1, 1, 'green', 'yellow');
// start at index 1, remove 1 element and return it; insert ‘green’ and ‘yellow’.
```

["white"]

```
colors;
```

["red", "green", "yellow", "blue"]

colors.splice(2, 2, 'magenta');

// start at index 2, remove 2 elements and return them; insert ‘magenta’.

["yellow", "blue"]

colors;

["red", "green", "magenta"]

colors.splice(2, 0, 'cyan');

// start at index 2, remove 0 elements and return them; insert 'cyan'.

[]

colors;

["red", "green", "cyan", "magenta"]

7.5. The Array as a Stack

The push(), pop(), shift() and unshift() methods allow us to use an array as a first in last out stack.

The push() method adds the given elements to the end of the array and returns the new length of the array.

var colors = ['red', 'white', 'blue'];

colors.push('green', 'yellow');

5

colors;

["red", "white", "blue", "green", "yellow"]

The pop() method removes the last element from an array and returns that element.

>>> colors.pop()

"yellow"

colors; // the original array is mutated

["red", "white", "blue", "green"]

>>> colors.pop()

"green"

colors; // the original array is mutated

["red", "white", "blue"]

If you call pop() on an empty array, it returns an undefined value.

>>> [].pop()

undefined

The **unshift()** and **shift()** methods are similar to push() and pop() but they operate on the **beginning** of the array instead of at its end. As a result, **they are usually slower than push() and pop().**

The unshift() method inserts one or more elements to the beginning of an array and returns the new length of the array.

```
var colors = ['red', 'white', 'blue'];
colors.unshift('green', 'yellow');
```

5

```
colors;
```

["green", "yellow", "red", "white", "blue"]

The shift() method removes the first element from an array and returns that element.

```
colors.shift();
```

"green"

```
colors.shift();
```

"yellow"

If you call shift() on an empty array, it returns an undefined value.

```
[].shift();
```

undefined

7.6. Iterating and Mapping

The forEach() method iterates over an array and executes a given function once for each element.

The function you supply is passed three arguments: the value of the array element, the index of the array element, and the array itself.

If you only need the value and index, you can write a function with only two parameters — the additional arguments will be ignored:

```
function showElement( value, index) {
   console.log (index + '--->' + value);
}
var myArray = ['red', 'white', 'blue'];
myArray.forEach(showElement);
```

0--->red

1--->white

2--->blue

You can also use forEach() to modify the array you are invoking forEach() on.

```
function extraPoints( value, index, array) {
   array[index] = value + 5;
}
var grades = [85, 94, 82, 90];
grades.forEach(extraPoints);   // add 5 to each element of the array
grades;    // the array is modified.
```

[90, 99, 87, 95]

The map() method is similar to forEach() but it **returns a new array with the results of calling the given function on each element in this array**. In this case, the given function should have a return value.

```
function square (number) {
   return number * number;
}
var myArray = [1, 2, 3, 4, 5 ];
var mySquares = myArray.map(square);
mySquares;
```

[1, 4, 9, 16, 25]

```
myArray;  // the original array is unchanged,
```

[1, 2, 3, 4, 5]

forEach() and map() were introduced in ECMAScript 5.

7.7. Filtering and Testing

The filter() method creates a new array with all elements that pass the test implemented by the given function.

```
function isA(grade) {
   if (grade >= 90) {
      return true;
   } else {
      return false;
   }
}
var classGrades = [60, 90, 87, 100, 86]
```

var aGrades = classGrades.filter(isA)

aGrades; //aGrades will have the elements of classGrades that are >= 90.

[90, 100]

classGrades // The original array is unchanged.

[60, 90, 87, 100, 86]

If we need to pass an additional parameter to the filtering function, we can do that using 'this' as follows:

var **myA** = { threshold: 90 } // we create an object that will hold the threshold for an A.

var aGrades = classGrades.filter(isA, **myA**); // **we pass the object as a filter argument**.

And now we can use the object inside the filtering function as follows:

```
function isA(grade) {
   if (grade >= this.threshold) {   // 'this' refers to the myA object.
      return true;
   } else {
      return false;
   }
}
```

The every() method tests whether all elements in the array pass the test implemented by the given function.

classGrades.every(isA);

false

aGrades.every(isA);

true

The some() method tests whether some element in the array passes the test implemented by the given function.

classGrades.some(isA);

true

The filter(), every() and some() methods were introduced in ECMAScript 5.

7.8. What else can we do with an array?

The join() method joins all elements of an array into a string and returns that string. You can specify a separator. If you don't, the default separator is a comma. The original array is unchanged.

var colors = ['red', 'white', 'blue'];

colors.join(); // no separator is specified. It will default to a comma.

"red,white,blue"

colors; // the original array is unchanged.

["red", "white", "blue"]

colors.join(' and '); // we specify ‘ and ‘ with extra spaces as the separator.

"red and white and blue"

The slice() method returns a shallow (one level deep) copy of a portion of an array from a given index position up to but not including a second index position. We’ll examine the distinction between shallow and deep copy in the next section.

var myArray = ['zero', 'one', 'two', 'three'];

myArray.slice(1, 3);

["one", "two"]

We get a copy of a portion of the array from a 1 up to but not including 3.

The original array in unchanged.

myArray;

["zero", "one", "two", "three"]

myArray.slice(1);

["one", "two", "three"]

When the end index is not specified, the slice goes to the end of the array.

myArray.slice();

["zero", "one", "two", "three"]

When neither index is specified, we get a copy of the whole array.

When we specify a negative index, we start counting from the end of the array.

myArray.slice(-3);

["one", "two", "three"]

The concat() method returns a new array containing a shallow copy of the array on which it is invoked followed by each of the arguments to concat(). If an argument is an array, then each of its elements is concatenated individually. If one or more of these elements is an array (nested array), then it is concatenated as an array.

var grades = [100, 86];

var homework = [95, 98];

var newGrades = grades.concat(homework, 90);

Each of the elements of the array homework is concatenated individually.

[100, 86, 95, 98, 90]

grades; // The original array is unchanged

[100, 86]

Here the argument to concat() is an array, [homework, 90] containing an array, homework:

newGrades = grades.concat([homework, 90]);

[100, 86, [95, 98], 90]

The **indexOf()** method returns the first index at which a given element can be found in the array, or -1 if it is not present. The **lastIndexOf()** method returns the last index at which a given element can be found in the array, or -1 if it is not present.

These two methods were introduced in ECMAScript 5.

var myArray = [100, 85, 90, 100];

myArray.indexOf(100);

0

myArray.lastIndexOf(100);

3

myArray.indexOf(100, 1); // start searching from index 1 on.

3

myArray.lastIndexOf(100, 1); // start searching backward from index 1 on.

0

myArray.indexOf(65)

-1

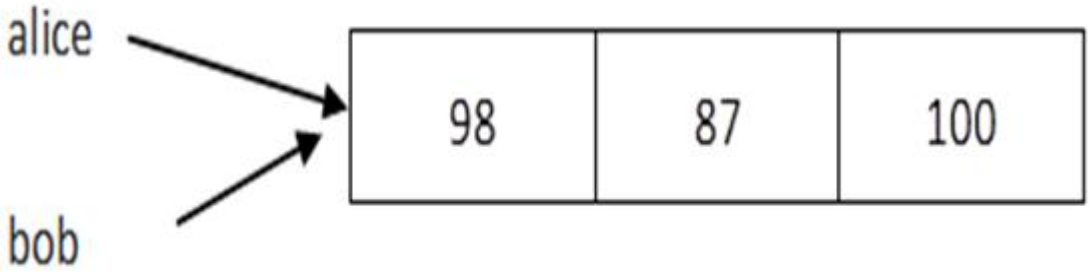
7.9. Shallow vs Deep

Assignment with an = on objects does not make a copy. Instead, the assignment makes the two variables point to the same object in memory.

Consider two array variables alice and bob. We create these variables as follows to hold Alice's and Bob's grades on various assignments.

var alice = [98,87,100];

var bob = alice;



alice;

[98, 87, 100]

bob;

[98, 87, 100]

What happens if we now change one element in the array alice?

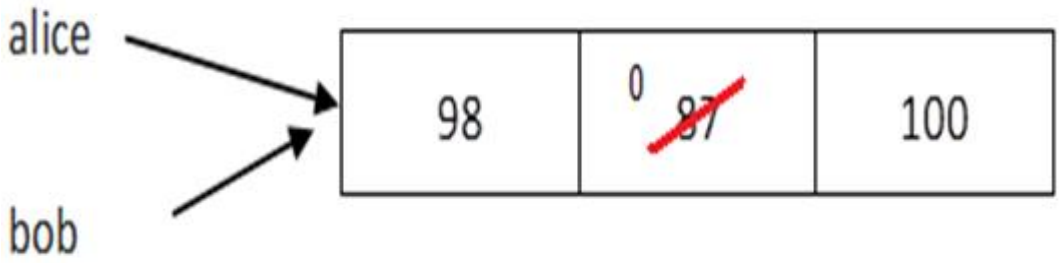
alice[1]=0;

alice;

[98, 0, 100]

bob;

[98, 0, 100]



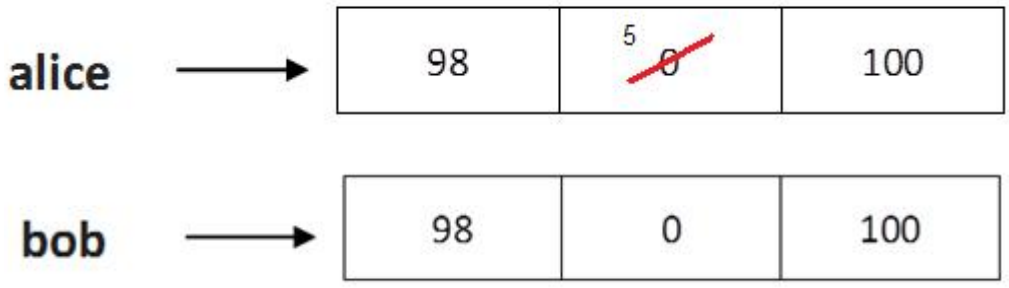
What if we wanted bob to have a different copy of the array, one that initially has the same values as alice but that is not affected by future changes to alice.

If the array we are copying is not nested, the following slice assignment will work.

bob=alice.slice(); // slice from 0 till the end of the array

bob;

[98, 0, 100]



alice[1]=5;

alice ;

[98, 5, 100]

bob

[98, 0, 100]

However with nested arrays the slice assignment will NOT work. It is a shallow copy.

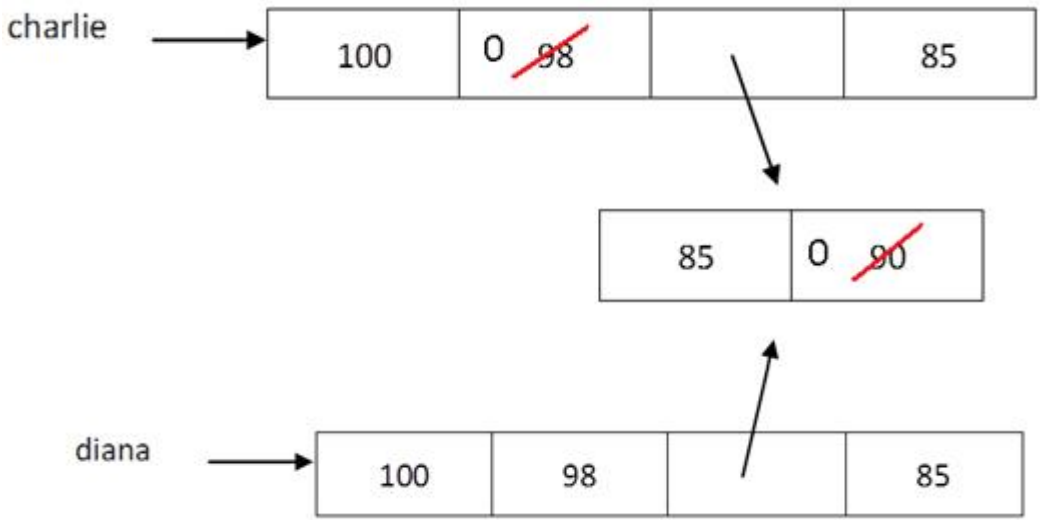
var charlie = [100,98,[85,90],85];

var diana=charlie.slice();

diana;

[100, 98, [85, 90], 85]

charlie[1]=0;



charlie;

[100, 0, [85, 90], 85]

Diana;

[100, 98, [85, 90], 85]

charlie[2][1]=0;

charlie ;

[100, 0, [85, 0], 85]

diana

[100, 98, [85, 0], 85]

We'll later see how to use JSON to get around this problem.

8. JavaScript in your browser

8.1. Javascript in HTML

The following sections assume that you are familiar with the basics of HTML. You don't have to be an expert, but you should be familiar with the structure of HTML documents and how the tags are used to display various kinds of content in your browser. If HTML is completely new to you or if you need a refresher, you can visit http://www.w3schools.com/html/default.asp and go through the first few topics in the tutorial.

The <script> tag:

You can embed JavaScript code in HTML documents by using the <script> tag.

Although you can write JavaScript code inline between a pair of <script> and </script> tags, **the recommended way is to include the code in an external file,** saved with the .js extension, and point to it using the **src attribute** of the <script> tag.

This is part of the **unobtrusive JavaScript approach to separate the programming logic from the HTML content.**

< script src ="../scripts/pause.js" > </ script >

A note about file names and path names:

The external file name includes the URL if the file is on another server, or the path name if the file is in the local directory.

In the following examples, I'll be saving html files in the directory CS22A/html/ and the JavaScript files in the directory CS22A/scripts/.

So from my html file location, to get to a script saved in the directory CS22A/scripts/, I'll have to go up to the parent directory CS22A denoted by ../ and then to scripts/pause.js.

Timeline:

The browser reads each line of HTML code from top to bottom, and processes and displays it. Consider the following html file.

hello1.html:

```
<!DOCTYPE html>

<html>
```

```
<body>

<h1>Hello World!</h1>

<script src="../scripts/pause.js"></script>

<p>Hello again!</p>

</body>

</html>
```

When a <script> element is encountered, the external file is usually read (downloaded) and the code is executed by the JavaScript interpreter, and then the parsing and rendering of the HTML continues until the end of the document is reached. This means that the parts of the document that come after the script (Hello again!) will not appear in the browser until the script has been downloaded and executed.

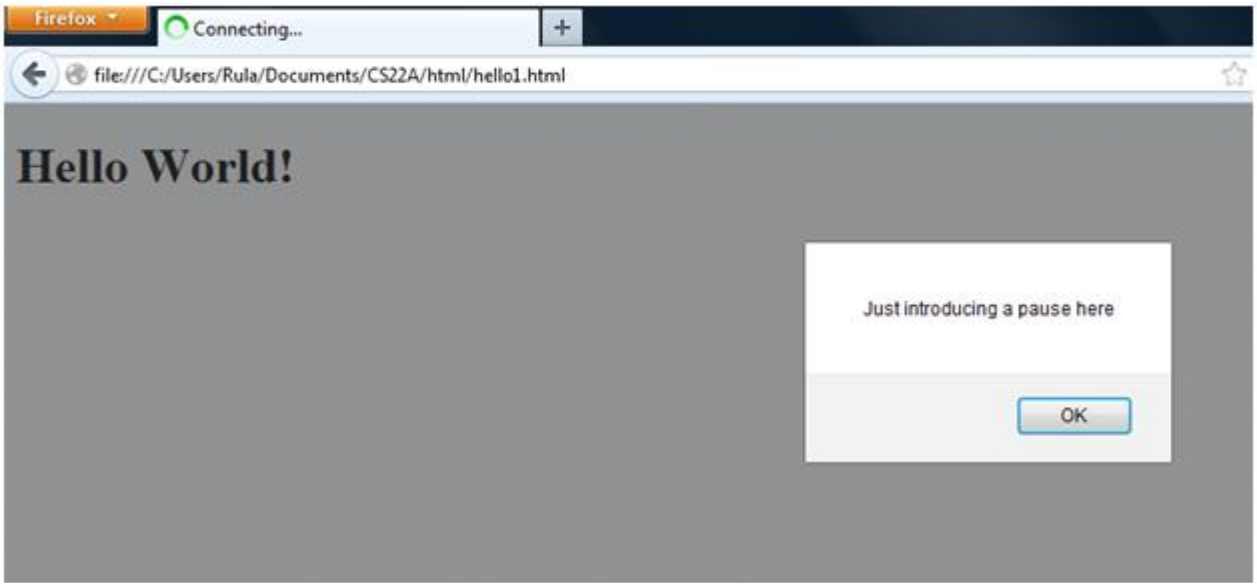
To see how that works, we can create the following script. We'll cover the alert() method in the next section but for now, we'll use it to make the execution of the script pause while waiting on the user.

pause.js

```
alert ("Just introducing a pause here");
```

Create and save the html file hello1.html shown above. Then create and save the JavaScript file pause.js as shown above. You may use Scratchpad to do that. Now open hello1.html with your browser. You can do that as shown below.

You'll get the screen shown below. Notice the Connecting… at the top of the window. Hello again does not appear on the page until you click on OK.



We can change the above default behavior by specifying additional attributes for the <script> tag.

The defer attribute causes the browser to defer execution of the script until after the document has been loaded and parsed.

hello2.html:

```
<!DOCTYPE html>

<html>

<body>

<h1>Hello World!</h1>

<script defer src="../scripts/pause.js"></script>

<p>Hello again!</p>

</body>

</html>
```

Note if you have several deferred scripts, they will run in the order in which they appear in the document.

You can open hello2.html in your browser window now and see the changes in the loading timeline.

The async attribute causes the browser to run the script as soon as possible but not to block document parsing while the script is being downloaded. The async attribute is new in HTML5.

hello3.html:

```
<!DOCTYPE html>

<html>

<body>

<h1>Hello World!</h1>

<script async src="../scripts/pause.js"></script>

<p>Hello again!<b>

</body>

</html>
```

Note that if you have several asynchronous scripts, they will run as they load, which means that they may execute out of order.

You can open hello3.html in your browser window now and see that for this example, there is no visible difference in the loading timeline between async scripts and deferred scripts.

Using the defer or async tags will usually result in a better user experience.

8.2. The Window Object

The window object represents a web browser window or frame, and you can refer to it with the identifier window. It is the global object in client-side JavaScript. This means that it is at the top of the scope chain and that **its properties and methods are global variables and global functions**. To refer to them in our code, we can just use the property name without prefixing it with window. So for example, window.alert() is the same as alert().

We'll cover some of these properties and methods next.

8.3. Dialog Boxes

The window object provides some methods that display simple dialog boxes.

The **alert()** method displays a message and waits for the user to dismiss the dialog. It should be used for messages that do not require any response on the part of the user, other than the acknowledgement of the message.

The **confirm()** method displays a message, waits for the user to click an OK or Cancel button and returns true (if the user clicks OK) or false (if the user clicks Cancel).

The **prompt()** method displays a message, waits for the user to enter a string, and returns that string.

Dialog boxes are modal windows - they block program execution and prevent the user from accessing the rest of the interface until the dialog box is closed. For this reason, **you should only use them sparingly.**

Example:

```
var name = prompt("Please enter your name");
var answer = confirm ("Would you like to continue?");
if (answer) {
     alert("Hello " + name);
} else {
     alert ("Bye");
}
```

8.4. Timers

Timers are important global functions of client-side JavaScript, and are therefore defined as methods of the window object even though they don't really have anything to do with the window.

The **setTimeout()** method calls a function to run after a specified delay (in milliseconds). It returns a value that can be passed to **clearTimeout()** to cancel the setTimeout().

The **setInterval()** method calls a function repeatedly, with a fixed time delay between each call. It returns a value that can be passed to **clearInterval()** to cancel any future invocations of the scheduled function.

The following code sets up a 60 seconds timer.

```
function timeIsUp(){
     alert("Time is up!");
}

setTimeout(timeIsUp, 60000);
```

8.5. Other Window Properties

The **location property** of the window object is itself an object that represents the current URL.

To see what the location property looks like for a given web page, go to www.Foothill.edu, activate Firebug, then at the prompt in the Firebug console, type: location (or window.location which is the same here.)

You'll get the following:

>>> location

http://www.foothill.edu/index.php { constructor={...},

href="http://www.foothill.edu/index.php", protocol="http:", more...}

And if you click to see more:

You can see that the location object has properties that describe the individual parts of the URL, such as protocol, pathname and hostname.

Similarly, the **history property** of the window object represents the browsing history.

The window object has **open()** and **close()** methods that allow you to open a new window and close the current window.

Finally, one of the most important properties of the window object is **document**: it is an object that represents the content displayed in the window. We'll cover the document object in details in the next module.

9. The Document Object Model

9.1. What is the Document Object Model?

The Document Object Model (DOM) is a World Wide Web Consortium standard. It is defined as a **platform and language independent interface** that allows programs to **access and update the content, structure and style of documents**.

A web page is a document. This document can be either displayed in the browser window, or as the HTML source. But it is the same document in both cases. The Document Object Model (DOM) provides yet another way to represent, store and manipulate that same document. **The DOM is a fully object-oriented representation of the web page**, and it can be accessed and modified with JavaScript.

The DOM API may be used for both HTML and XML documents. We'll focus on HTML here.

9.2. From HTML Document to DOM tree

The nested items in an HTML document are represented in the DOM as **a tree of objects**. This **tree** contains **nodes** representing **HTML tags** such as < body > and < p >, and nodes representing **the content inside the tags.**

The nodes that represent the tags are called element nodes and the nodes that represent the content inside the tags are called text nodes.

And there is also a node representing the whole document.

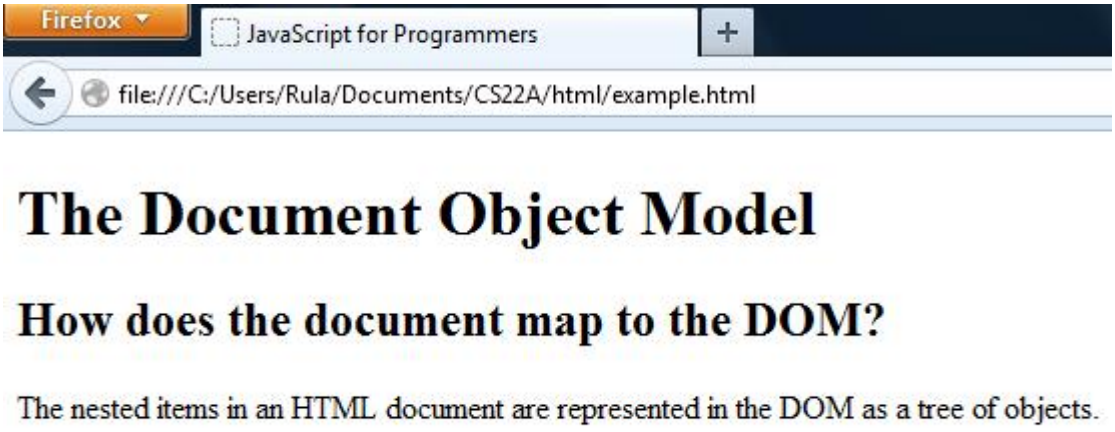
Let's look at a simple HTML document, example.html. Copy this document and follow along.

Note the special formatting used here with line breaks inserted inside the HTML tags. We'll use this special formatting to make it easier to deal with a white space issue in the DOM traversal. More on that later.

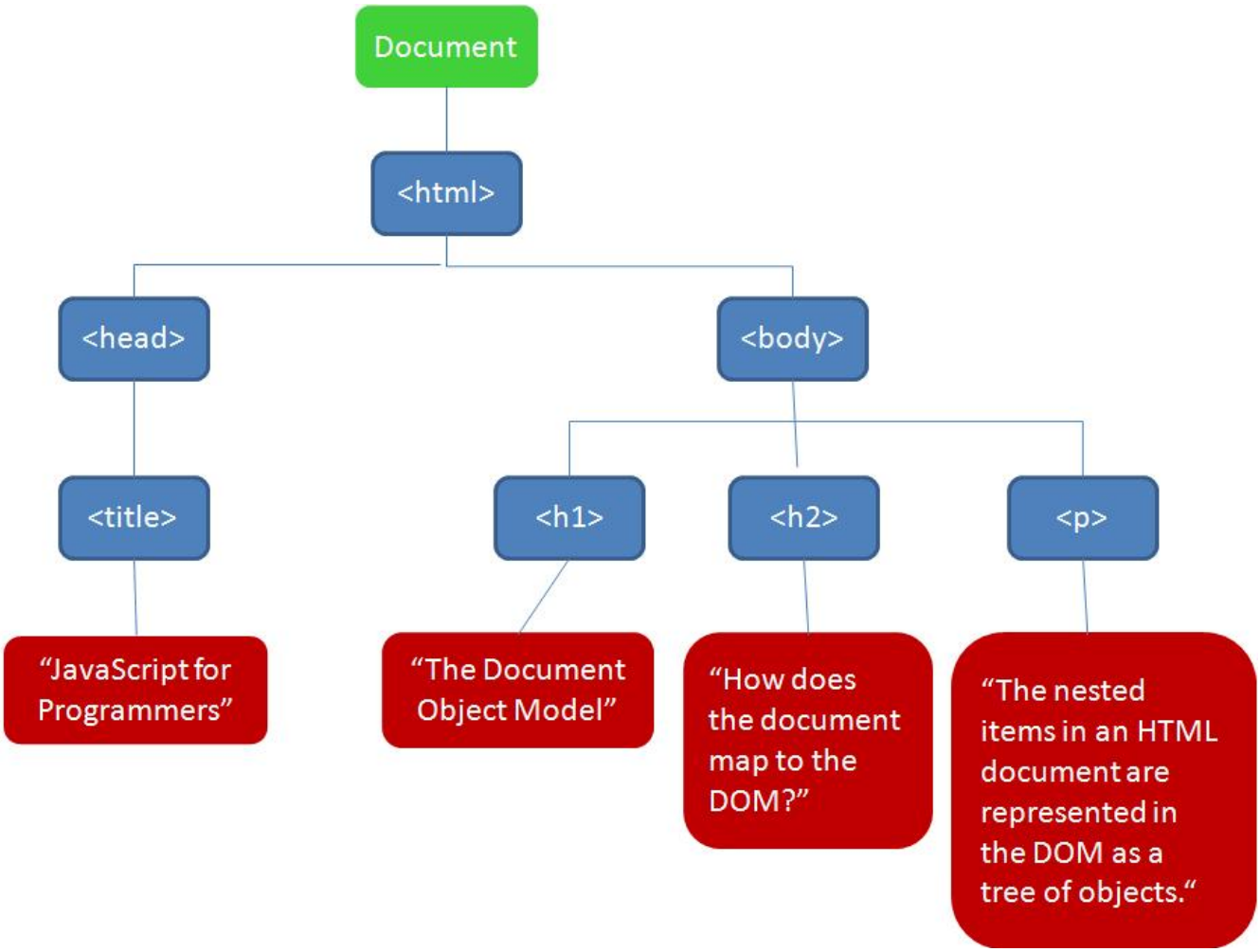
```
<!DOCTYPE html>
<html
        ><head
                ><title>JavaScript for Programmers</title
        ></head
        ><body
                ><h1>The Document Object Model</h1
                ><h2>How does the document map to the DOM?</h2
                ><p>The nested items in an HTML document are represented in the DOM as a tree of
objects.</p
        ></body
></html>
```

The above is the HTML source of our document (as we can see it in our editor).

We can also open the document in our browser, and this is what we get:



The corresponding DOM tree for our example.html document is as follows:



Our tree contains three different types of nodes.

At the root of the tree is the green **document node**: it represents the entire document.

The blue nodes are **element nodes** that represent HTML tags.

The red nodes are t**ext nodes** and they represent the content inside the tags.

9.3. Tree Terminology

The DOM tree is just like a family tree: there are **parents, children, siblings,** ancestors and descendants. The only difference is that every node except the root (the document) has **exactly one parent.**

Looking back at our previous example, we can observe the following:

The <html> element node has two children: the <head> node and the <body> node. The <head> node is the first child and the <body> node is the last child.

The parent of the <body> node is the <html> node.

The parent of the <head> node is also the <html> node.

The <head> node and the <body> node are siblings.

Similary, the <h1>, <h2> and <p> element nodes are all siblings.

The next sibling of <h2> is <p> and the previous sibling of <h2> is <h1>.

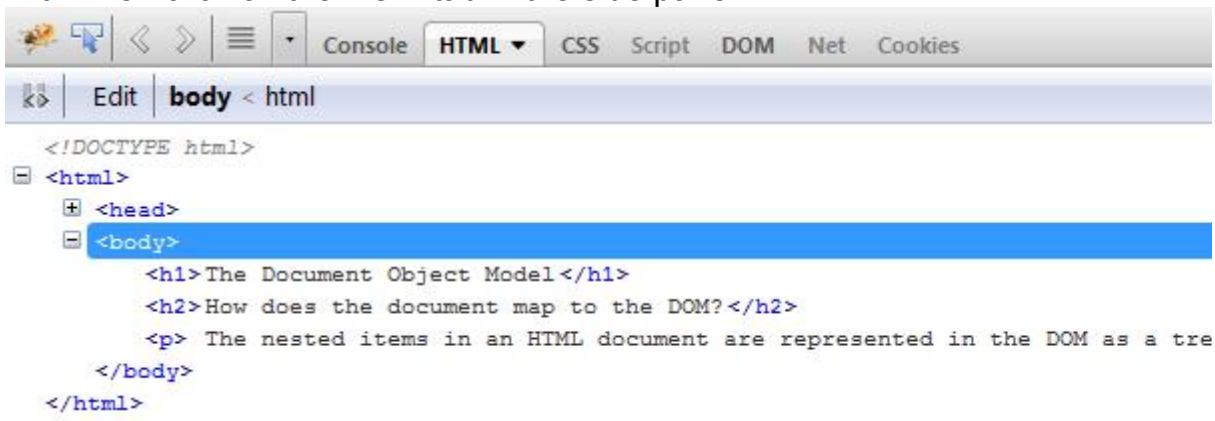
9.4. The DOM in Firebug

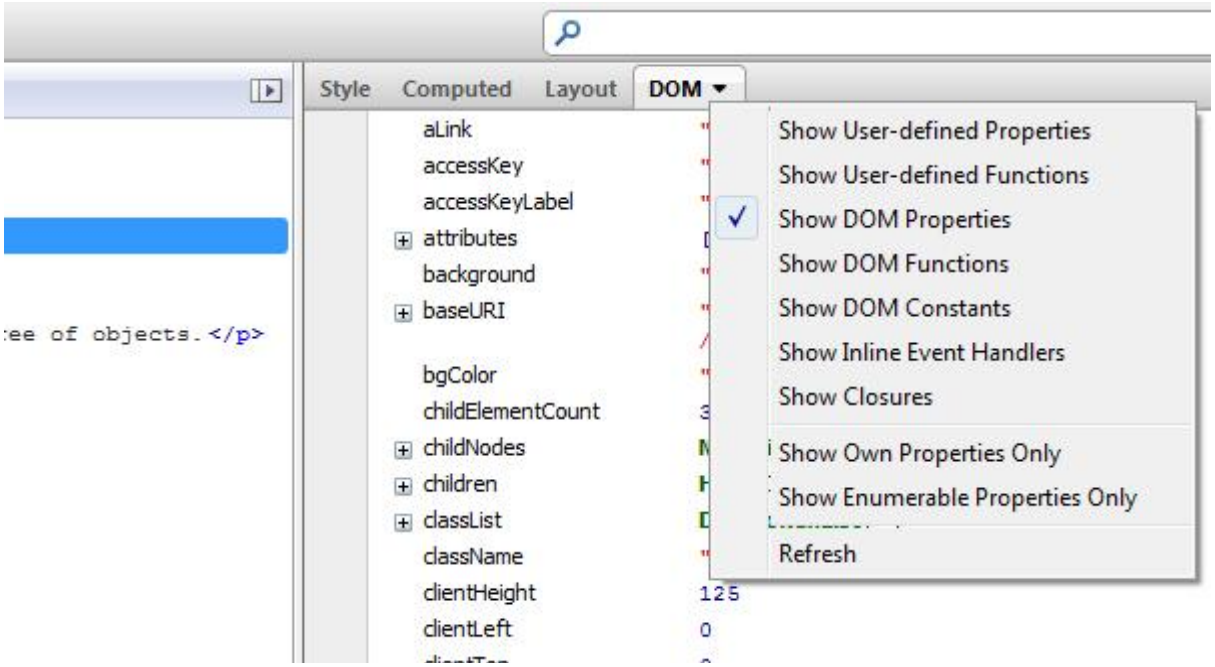
We can use Firebug to see the various DOM nodes and their relationships.

Open the file example.html in Firefox. You may do that by selecting Firefox -> New Tab -> New File. Click on the Firebug icon to open Firebug.

Then click on the HTML tab as shown below:

If the Firebug side panel is not shown, click on the triangle in the upper right corner to show it. Then click on the DOM tab in the side panel.

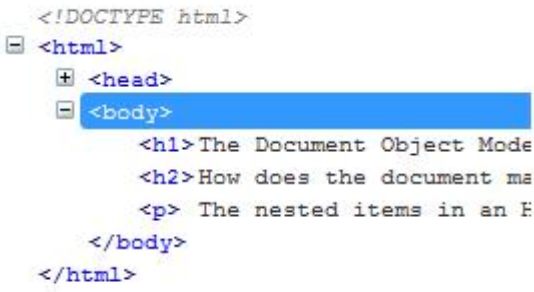




From here on, you can click on any HTML element in the Firebug left panel and the corresponding DOM object will be shown in the right side panel.

Note that the object properties are listed alphabetically.

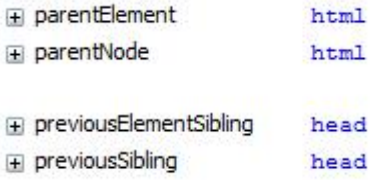
For example if you highlight the <body> tag in the left panel:



You can see its children in the right panel:

Note that we have childNodes and children. c**hildNodes contains all the nodes** (elements and text - blue and red) that are children of the <body> whereas **children contains only the element nodes** (blue only) that are children of <body>. In this case, since <body> does not have any text node children, childNodes and children have the same elements.

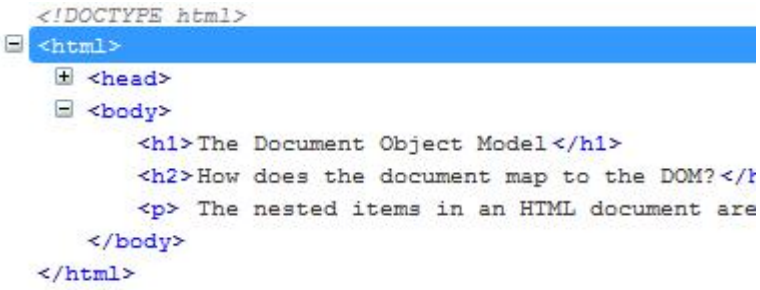
And if you scroll down in the right panel, you can also see its parent and its previous sibling:



Again there are different node and element properties. We have parentElement and parentNode and previousElementSibling and previousSibling.

In this case again, they are the same.

Consider however the <html> element.



Its **parent node is the document node** (green) and it has **no parent element** (null).

9.5. White space in the DOM

In our example, we used a special formatting in our HTML source file to deal with a white space issue in the DOM.

The different browsers deal with the white space (or new lines) inside the HTML source in an inconsistent manner. That can make **the node traversal of the DOM tree unpredictable**.

In Firefox, all white space in the document is represented as extra text nodes in the DOM. That includes the new line character between <html> and <head> below.

<html>

    <head>

So to avoid ending up with extra text nodes we put our line breaks inside the tags, where white space is ignored.

We'll see later how to use a framework like jQuery to deal with these inconsistencies between browsers.

Another option is to use the **element traversal** interface where the text nodes are ignored.

We'll see examples of both node and element traversals later in this module.

We will no longer use the special formatting of HTML source files. We'll just keep in mind that the DOM tree may have some extra text nodes containing white space.

9.6. Layout Engines and the DOM

The different browsers rely on layout engines to parse HTML into a DOM. In general, layout engines are responsible for parsing the source HTML, generating the DOM representation and displaying the content.

The different layout engines implement the DOM standards to **varying degrees of compliance.**

Webkit is the layout engine currently used by Safari (both desktop and mobile) and the Android browser. Blink is currently used by Chrome and Opera, Gecko is used by Firefox and Trident is used by Internet Explorer.

To figure out what DOM features a given browser supports, we need to go back to the underlying layout engine documentation.

In this course, we'll cover the features that are supported by most engines.

9.7. Selecting Elements: where to start?

When we want to access objects in an HTML page, we start with the document object, window.document or simply document.

The document object has two properties: documentElement and body.

document.documentElement will give you access to the top most <html> element.

document.body will will give you access to the <body> element.

Let's go back and open our example.html file in Firefox, then go the Firebug console,

and type the following. Note that you only need to type what comes after the >>> prompt. The text in blue is the output you get.

```
>>> document
```

```
Document example.html
```

```
>>> document.documentElement
```

```
<html>
```

```
>>> document.body
```

```
<body>
```

9.8. Selection by id

There are several methods defined on the document object to access individual elements.

The recommended method is the simplest one: **getElementById().**

When you need to access a specific element, give that element an HTML id attribute, and then you can look it up in your program using that id.

Note that any HTML element can have an id attribute. The value of this attribute must be **unique** within the document — no two elements in the same document can have the same id.

Let's modify our first example file to add some ids. Save the new document in the file idexample.html.

```
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript for Programmers</title>
    </head>
    <body>
        <h1 id = "main">The Document Object Model</h1>
        <h2 id = "subtitle">How does the document map to the DOM?</h2>
        <p id = "par1"> The nested items in an HTML document are represented in the DOM as a tree of objects.
        </p>
    </body>
</html>
```

Let's open idexample.html in Firefox, then go the console, and type the following. Here again you only need to type what comes after the >>> prompt. The text in blue is the output you get.

```
>>> document.getElementById("main")
```

<h1 id="main">

>>> document.getElementById("subtitle")

<h2 id="subtitle">

>>> document.getElementById("par1")

<p id="par1">

If the id does not exist in the document, getElementById returns null:

>>> document.getElementById("noid")

null

Note that the id is case sensitive. If we specify “Main” instead of “main”, we get null:

>>> document.getElementById("Main")

null

9.9. Selection by type

You can select all elements of a specified type (or tag name) using the **getElementsByTagName()** method. This method is **defined on the document object as well as on individual elements.** It returns **an array like object** containing all elements with a given tag, starting at the document level or at the specified element level:

To get all the elements with a <p> tag in the document:

>>> document.getElementsByTagName('p');

HTMLCollection[p#par1]

To get all the elements with an <h1> tag in the document:

>>> document.getElementsByTagName('h1');

HTMLCollection[h1#main]

We can use * as a wild card to get ALL tags in a document:

>>> document.getElementsByTagName('*');

HTMLCollection[html, head, title, body, h1#main, h2#subtitle, p#par1]

Note that **we get an array like object containing all the matching elements**. To access one particular element, we can index it with []. Indexing starts at 0.

>>> document.getElementsByTagName('*')[0];

<html>

Our search does not have to start at the document level. We can start at a given element and look for all the nested elements that satisfy the tag search:

>>> var myMain = document.getElementById("main");

>>> myMain

<h1 id="main">

The variable myMain now points to the element object <h1>.

We can look for any tags starting at myMain. Of course there is none.

>>> myMain.getElementsByTagName('*');

HTMLCollection[]

Let's examine an element with nested tags such as the <body> element. We'll first use the body property of the document to access the <body> element object and then we'll invoke the getElementsByTagName method on that body object:

var bodyObj = document.body;

>>> bodyObj.getElementsByTagName('*');

HTMLCollection[h1#main, h2#subtitle, p#par1]

Or, writing the above two statements in one line:

>>> document.body.getElementsByTagName('*')

HTMLCollection[h1#main, h2#subtitle, p#par1]

9.10. Selection by class name

Similarly, we can access elements by class name.

The class attribute specifies one or more class names for an element. It is usually used to point to a class in a style sheet to apply the same presentation styles to different elements.

In HTML5, it can be used on any HTML element.

You can select all elements of a specified class using the **getElementsByClassName()** method. **This method is defined on the document object as well as on individual elements**. It r**eturns an array like object** containing all matching elements, starting at the document level or at the specified element level.

getElementsByClassName() takes one argument, but that argument may specify **multiple class names separated by a space**. Only elements that **include all of the specified class names in their class attribute are matched**. The order of the identifiers does not matter.

Let's look at the following html source document classexample.html:

```
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript for Programmers</title>
    </head>
```

```
    <body>
        <h1 class = "important"> Layout Engines</h1>
        <p id = "mainidea" >
            <span id = "first" class = "info">The different browsers rely on layout engines. </span>
            <span id = "second" class = "important info">
                The different layout engines implement the DOM standards to varying degrees of
compliance.
            </span>
        </p>
    </body>
</html>
```

You can follow along by opening the document in Firefox and typing the commands below at the Firebug console.

To get all the elements with the 'important' class:

```
>>>document.getElementsByClassName('important');
```

```
HTMLCollection[h1.important, span#second.important]
```

We get the h1 heading element as well as the second span element.

Note that **we get an array like object containing all the matching elements.** To access one particular element, we can index it with []. Indexing starts at 0.

```
>>>document.getElementsByClassName('important')[0];
```

```
<h1 class="important">
```

To get all the elements with the 'info' class:

```
>>> document.getElementsByClassName('info');
```

```
HTMLCollection[span#first.info, span#second.important]
```

We get the first span element (with a class = ‘info’) as well as the second span element with class = ‘important info’.

To get all elements with both the 'info' and 'important' classes:

```
>>> document.getElementsByClassName('info important');
```

```
HTMLCollection[span#second.important]
```

We get the second span element with class = ‘important info’. Note that the order of the class names specified did not matter.

The method is also defined on element nodes (not just on document). To get all the elements that have a class of ‘important’ inside the ‘main idea’ paragraph, we can write the following:

```
>>> var mainPar = document.getElementById("mainidea")
>>> mainPar.getElementsByClassName("important");
HTMLCollection[span#second.important]
```

We can chain the two method invocations above into one statement:

```
>>> document.getElementById("mainidea").getElementsByClassName("important");
HTMLCollection[span#second.important]
```

Note that the h1 heading element does not appear here even though it has a class of "important". The search for the "important" class was restricted to the descendants of the paragraph element with id = "mainidea".

9.11. Node Based Traversal

Once we have selected an element from a Document, we sometimes need to find related items. There are two approaches to do that. One is based on a node traversal and the other is based on an element traversal.

The node traversal interface was the original one. It provides the following node object properties:

- parentNode
- childNodes
- firstChild
- lastChild
- nextSibling
- previousSibling

Let's illustrate these properties with our previous idexample.html.

Here it is again. Note that we did not use the special formatting to get rid of the line breaks between tags here.

idexample.html.

```
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript for Programmers</title>
    </head>
    <body>
        <h1 id = "main">The Document Object Model</h1>
        <h2 id = "subtitle">How does the document map to the DOM?</h2>
        <p id = "par1"> The nested items in an HTML document are represented in the DOM as a tree of
objects.
        </p>
    </body>
```

</html>

Again you can follow along by opening the document in Firefox and typing the commands below at the Firebug console.

>>> document.body.parentNode

<html>

>>> document.parentNode

null

The document node has no parent node so the value of parentNode is null.

>>> document.body.childNodes

NodeList[<TextNode textContent="\n ">, h1#main, <TextNode textContent="\n ">, h2#subtitle, <TextNode textContent="\n ">, p#par1, <TextNode textContent="\n \n">]

document.body.childNodes is an array like object that represents all the node children of <body>. Here it contains 3 element nodes <h1>, <h2> and <p> as well as 4 extra text nodes representing the line breaks in the source document.

We can access each child node by its index. Indexing starts at 0.

>>> document.body.childNodes[0]

<TextNode textContent="\n ">

>>> document.body.firstChild

<TextNode textContent="\n ">

document.body.firstChild is the same as document.body.childNodes[0].

>>> document.body.lastChild

<TextNode textContent="\n \n">

>>> document.body.nextSibling

null

The <body> node has no next sibling so the value is null.

>>> document.body.previousSibling

<TextNode textContent="\n ">

>>> document.body.childNodes[1]

<h1 id="main">

And to get the children nodes of <h1>:

>>> document.body.childNodes[1].childNodes

NodeList[<TextNode textContent="The Document Object Model">]

>>> document.body.childNodes[3]

<h2 id="subtitle">

>>> document.body.childNodes[3].nextSibling

<TextNode textContent="\n ">

>>> document.body.childNodes[3].childNodes

NodeList[<TextNode textContent="How does the document map to the DOM?">]

As you can see, navigating the DOM from node to node is tricky because of the extra newline text nodes.

9.12. Element Based Traversal

The more convenient alternative is to use the newer element traversal interface that implements an element only based navigation and ignores text nodes.

We can use the following element properties for the element traversal:

- firstElementChild
- lastElementChild
- previousElementSibling
- nextElementSibling
- childElementCount

In addition to the above standard properties, most current browsers implement the children property which returns only the children element nodes.

Again let's see how these properties work through an example.

>>> document.body.children

HTMLCollection[h1#main, h2#subtitle, p#par1]

document.body.children is an array like object that represents the element children of <body>. Here it contains 3 element nodes <h1>, <h2> and <p>.

We can access each child element by its index. Indexing starts at 0.

>>> document.body.children[0]

<h1 id="main">

>>> document.body.children[1]

<h2 id="subtitle">

>>> document.body.children[2]

<p id="par1">

>>> document.body.children[3]

undefined

>>> document.body.childElementCount

3

>>> document.body.firstElementChild

<h1 id="main">

You can see that document.body.firstElementChild is the same as document.body.children[0].

>>> document.body.lastElementChild

<p id="par1">

>>> document.body.nextElementSibling

null

The <body> element has no next sibling so the value is null.

>>> document.body.previousElementSibling

<head >

Note however that the text nodes do not appear in this navigation. The <h1> element has no (element) children.

>>> document.body.firstElementChild

<h1 id="main">

>>> document.body.firstElementChild.children

HTMLCollection[]

9.13. HTML Attributes and Element Properties

HTML elements consist of a tag name and a set of name-value pairs known as attributes. We have seen the id and the class attributes in our previous examples. We have also used the src attribute of the <script> tag.

HTML attributes are not case sensitive, but JavaScript property names are. To convert an attribute name to the JavaScript property, we need to convert it to lowercase,

The attribute name class is a reserved word in JavaScript. So the corresponding property name is className.

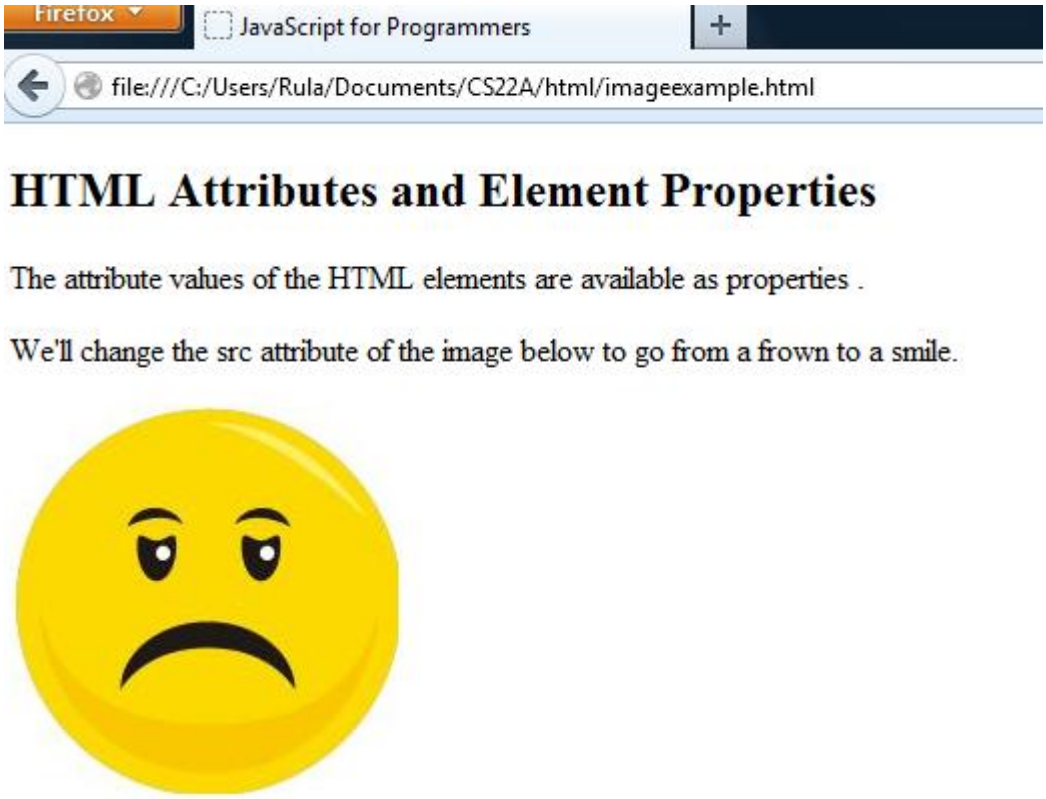
Let's create a new HTML source file that contains an image. The <image> tag has a src attribute that is similar to the src attribute we've encountered in <script> . It points to the file that contains the image to be displayed. To follow along, you can download the frown.gif and smile.gif files from Resources and save them in the same location as your HTML source file. Or you can create your own images.

imageexample.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript for Programmers</title>
```

```
    </head>
    <body>
        <h2>HTML Attributes and Element Properties</h2>
        <p> The attribute values of the HTML elements are available as properties .</p>
        <p> We'll change the src attribute of the image below to go from a frown to a smile. </p>
        <img id="myimage" src="frown.gif">
    </body>
</html>
```

Open the file imageexample.html in Firefox. You'll get the following:



Then enter the following at the Firebug console prompt:

```
>>> var frown = document.getElementById("myimage")
```

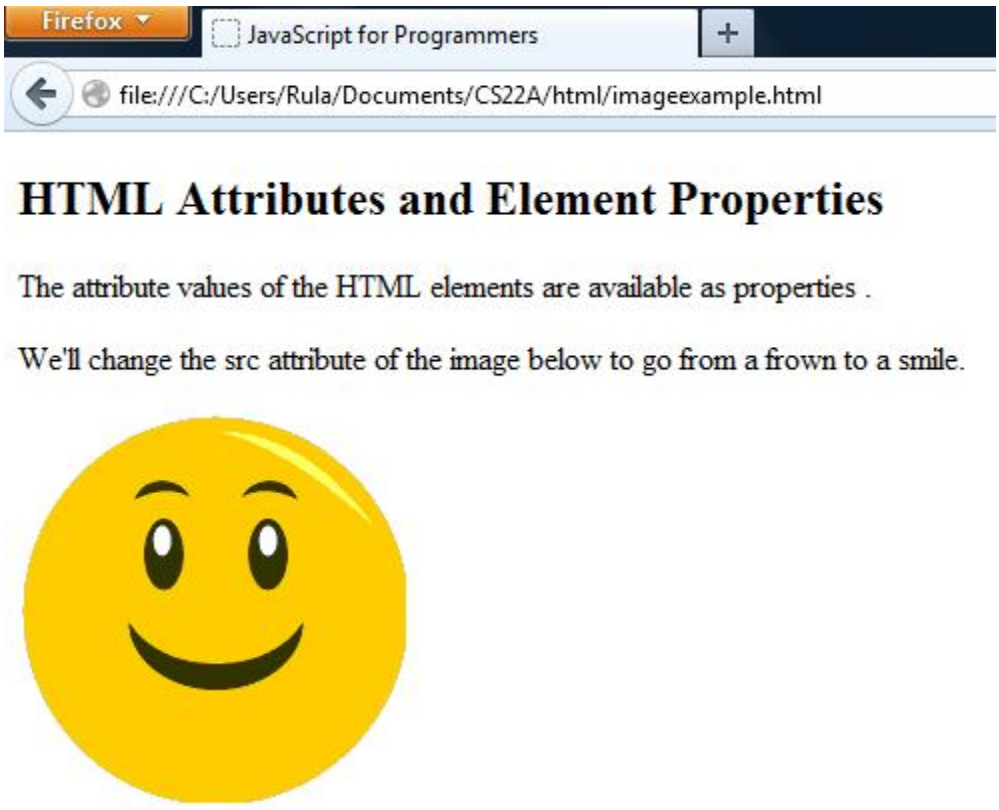
We first get the element corresponding to the <img> tag. We select it by id, which is the simplest way. We then query its src attribute.

```
>>> frown.src
"file:///C:/Users/Rula/Documents/CS22A/html/frown.gif"
```

Now we change its src attribute to point to the smiley image:

```
>>> frown.src = "smile.gif"
```

And the image is changed in the browser as shown below:



9.14. Element Content

We have seen how to get to the element object with methods such as getElementsById and getElementsByClassName. We have also seen how to change an element attribute.

But how do we access and possibly modify what's in these elements?

innerHTML:

One way to do that is with the **innerHTML** property of element objects. The property returns the content of an element as a string of markup. Basically it **returns everything in between the opening and closing tags of that element, including tags of nested elements, if any.**

Let's go back to our idexample.html source file in Firefox, and type the following at the console prompt:

>>> var myMain = document.getElementById("main");

>>> myMain

<h1 id="main">

The variable myMain now points to the element object <h1>.

The innerHTML property will return everything between the opening and closing <h1> tags:

>>> myMain.innerHTML;

"The Document Object Model"

>>> var myParagraph = document.getElementById("par1")

>>> myParagraph;

The variable myParagraph now points to the element object

>>> myParagraph.innerHTML;

" The nested items in an HTML document are represented in the DOM as a tree of objects."

Note that so far, the innerHTML property has returned text, with no HTML markup. That's just because the elements we were accessing did not have any nested tags.

Now let's look at the innerHTML property of the body element:

>>> document.body.innerHTML

"

<h1 id="main">The Document Object Model</h1>

<h2 id="subtitle">How does the document map to the DOM?</h2>

 The nested items in an HTML document are represented in the DOM as a tree of objects.

"

You can see that it includes everything between <body> and </body> including the white space and the nested tags and their text.

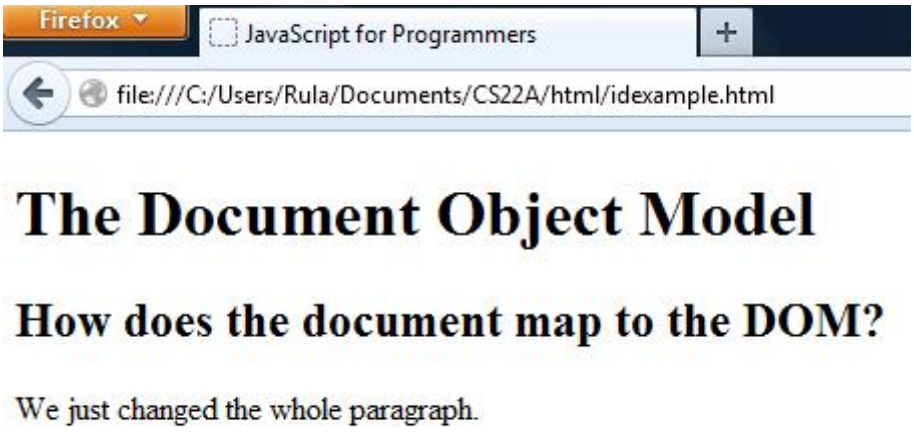
The innerHTML property is writable, so we can use it to modify the element.

>>> var myParagraph = document.getElementById("par1")

>>> myParagraph;

>>> myParagraph.innerHTML = " We just changed the whole paragraph."

And the text corresponding to paragraph par1 is modified:



We can even delete the entire content of the document by setting the innerHTML of the <body> element to the empty string.

>>> document.body.innerHTML = ''

And the whole document disappears.

There are several issues we need to be aware of when using innerHTML:

The first issue is that if you are inserting plain text, you need to be aware that it will be **parsed as HTM**L. So you'll need to use &lt instead of < and &gt instead of > and &amp instead of & and so on.

The second and most important issue is that using the innerHTML property comes with a security risk.

Consider the following code where you are assigning the innerHTML of some element to some variable that is outside your control. **someVariable could be any HTML code that invokes any script.**

document.getElementById('someId').innerHTML = someVariable;

For that reason, it is recommended not to use innerHTML when inserting plain text.

textContent:

The textContent property gives us access to the text content of the specified node, **and all its descendants**.

Going back to our idexample:

>>> document.body.textContent

"

The Document Object Model

How does the document map to the DOM?

The nested items in an HTML document are represented in the DOM as a tree of objects.

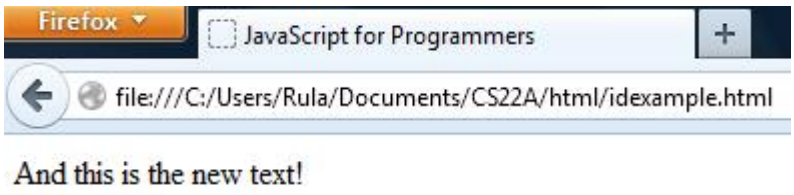
"

textContent concatenated the individual text content of <body> and all its descendants.

The textContent property is writable, so we can use it to modify plain text in our document.

>>> document.body.textContent = "And this is the new text!"

"And this is the new text!"



nodeValue and data:

We can also focus on the individual text nodes to access to the content of an element.

First we have to get to the text node of the given element. The first child of the par1 element is the text node that contains the actual text:

>>> document.getElementById("par1").firstChild

<TextNode textContent=" The nested items in an...M as a tree of objects.">

Let's query its nodeValue property:

>>> document.getElementById("par1").firstChild.nodeValue

" The nested items in an HTML document are represented in the DOM as a tree of objects."

Let's query its data property:

>>> document.getElementById("par1").firstChild.data

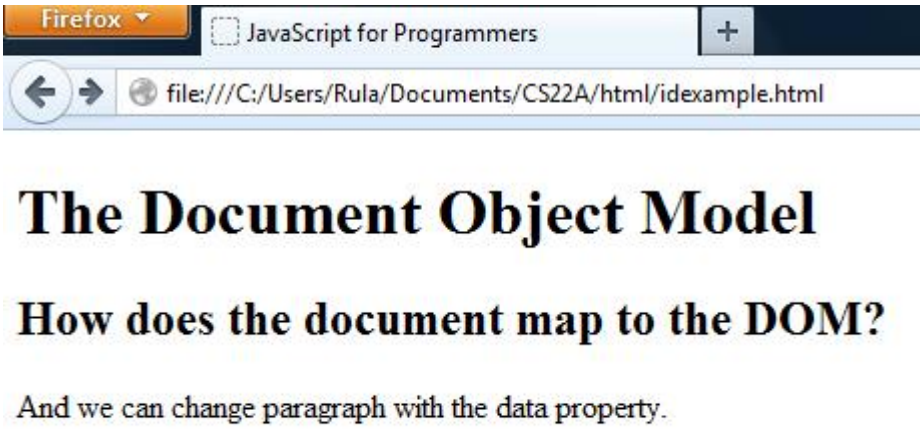
" The nested items in an HTML document are represented in the DOM as a tree of objects."

These properties are writable, so we can change the text content as follows:

>>> document.getElementById("par1").firstChild.data = "And we can change paragraph with the data property."

"And we can change paragraph with the data property."

And the text is modified in the document:



The nodeValue property also reflects the change.

>>> document.getElementById("par1").firstChild.nodeValue

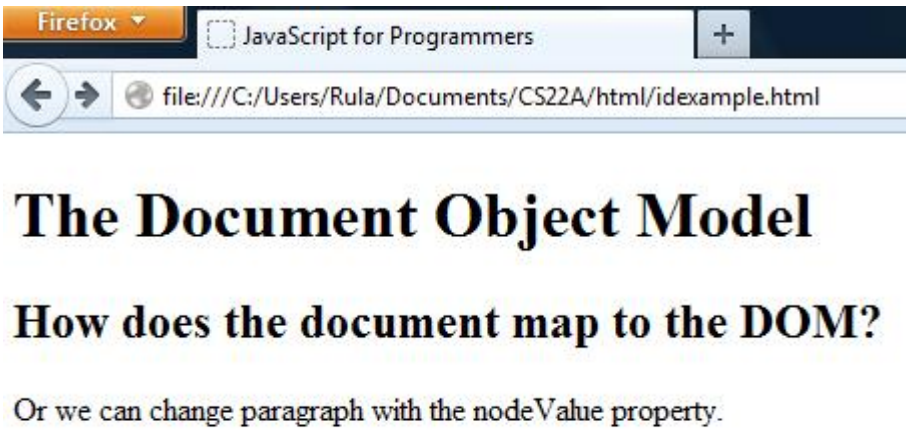
"And we can change paragraph with the data property."

Or we can change the nodeValue property as follows:

```
>>> document.getElementById("par1").firstChild.nodeValue = "Or we can change paragraph with the nodeValue property."
```

"Or we can change paragraph with the nodeValue property."

And the text is modified in the document:



The data property reflects the change.

```
>>> document.getElementById("par1").firstChild.data
```

"Or we can change paragraph with the nodeValue property."

The data property is automatically updated.

Note that these properties work on the text nodes only.

9.15. User Input

We have seen how to access and modify the text in our document with the innerHTML, textContent, nodeValue and data properties. Often we need to access text that the user enters and these properties do not help us.

We can use the input tag in HTML source documents to define a user entry field. This tag may or may not be inside a <form> tag.

Let's look at the following example:

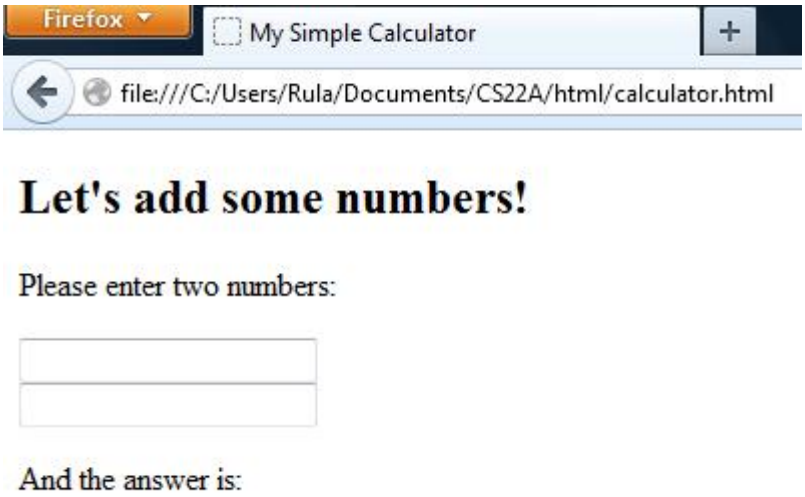
calculator.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Simple Calculator</title>
    </head>
    <body>
        <h2> Let's add some numbers! </h2>
        <p>Please enter two numbers:  </p>
```

```
        <input id = "first" type="number">
        <br>
        <input id = "second" type="number">
        <br>
        <p> And the answer is:</p>
        <p id="answer"></p>
    </body>
</html>
```

We have defined two <input> elements “first” and “second” where the user may enter numbers. We would like to compute the sum of these numbers and display it in the “answer” element.

Open calculator.html in Firefox and follow along. The following is displayed:



Let’s enter two numbers, say 6 and 18.

How do we access these numbers in our program? The properties innerHTML and textContent don’t give us what we need and we cannot use data or nodeValue because the <input> element has no text child node.

```
>>> document.getElementById("first").innerHTML
""
>>> document.getElementById("first").textContent
""
>>> document.getElementById("first").childNodes[0]
undefined
```

The **value** property may be used to access the input that the user enters.

```
>>> document.getElementById("first").value
```

"6"

>>> document.getElementById("second").**value**

"18"

Note first that the value property is giving us a **string** not a number. **To convert it to a number, we can use parseInt() or parseFloat()**. Here we use ParseFloat to convert the two strings to numbers and save them in two variables.

>>> var firstNumber = parseFloat(document.getElementById("first").value);

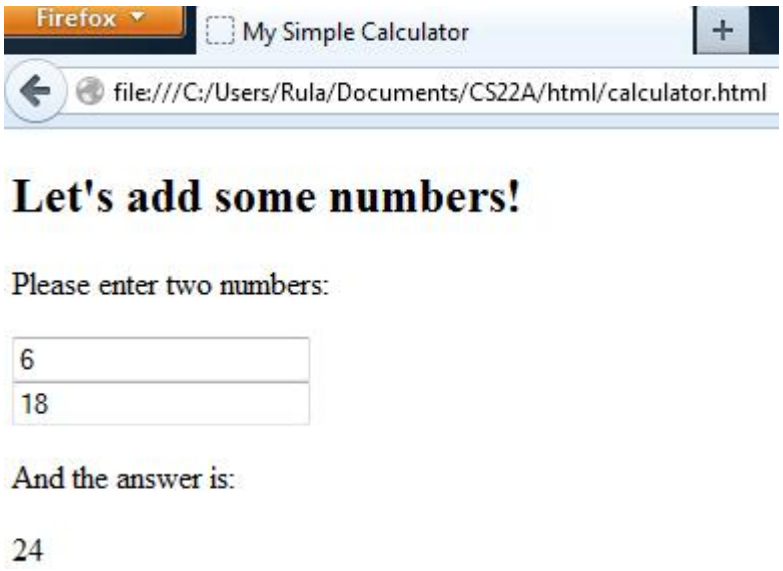
>>> var secondNumber = parseFloat(document.getElementById("second").value);

Then we compute the sum:

>>> var myAnswer = firstNumber + secondNumber;

And write it in the “answer” element:

>>> document.getElementById("answer").textContent = myAnswer;



Next week we’ll see how to add event handlers so that the answer is updated whenever the input changes.

10. Event Driven Programming

10.1. Events, Targets & Listeners

A client side JavaScript application is for the most part **event driven**. It sits and waits for events to happen.

Events can come from various sources. These sources include user activity such as key presses and clicks as well as network or browser state changes.

When a significant event occurs, the application responds to it (we say it **handles it**) then goes back to waiting for the next event.

So we can think of an event as anything significant that happens to the document, to the browser or to some element or object associated with it.

The following entities are associated with a given event:

The event type (or event name) describes the event. The type “click”, for example, means that the user clicked on an element. The type “keydown” means that a key on the keyboard was pushed down. The type “load” means that a document has finished loading from the network.

For a list of event types, visit https://developer.mozilla.org/en-US/docs/Web/Reference/Events

The event target is the object on which the event happened. The target could be the window, the document or a specific element. When a user clicks on a button, for example, the target is that button element. When the user enters text in an input field, the target is that input element.

The event handler or **event listener** is the function that responds to the event.

An application needs to register the event handler function with the browser, specifying an event type and an event target. When an event of the specified type occurs on the specified target, the browser invokes the handler.

An event object is an object that contains details about that event. Event objects are passed as an argument to the event handler function.

10.2. Registering Event Listeners

The recommended way to register events for current browsers is by using the addEventListener() method.

The addEventListener() method is invoked on an event target. It registers the specified listener on that event target . The event target may be any object that supports events such as an element in a document, the document itself or the window object.

addEventListener() takes three arguments.

The first is the **event type** for which the handler is being registered.

The second argument is the function that should be invoked when the specified type of event occurs. That is the **event listener** or handler.

The final argument is **a boolean value**. For now we’ll set it to false. We’ll revisit this argument a little later.

Note that the addEventListener() method allows us to register more than on event listener for a given event and a given target.

Let’s go back to our simple calculator example from the last module and illustrate the use of event listeners:

calculator.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Simple Calculator</title>
```

```
    </head>
    <body>
        <h2> Let's add some numbers! </h2>
        <p>Please enter two numbers:  </p>
        <input id = "first" type="number">
        <br>
        <input id = "second" type="number">
        <br>
        <p> And the answer is:</p>
        <p id="answer"></p>
    </body>
</html>
```

Instead of updating the web page manually from the console like we did in the previous module, we are ready to write a script that will do that as soon as the user enters or modifies one of the input values.

The first step is to add the script as an external file to the source HTML document.

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Simple Calculator</title>
    </head>
    <body>
        <h2> Let's add some numbers! </h2>
        <p>Please enter two numbers:  </p>
        <input id = "first" type="number">
        <br>
        <input id = "second" type="number">
        <br>
        <p> And the answer is:</p>
        <p id="answer"></p>
        <script async src="../scripts/add.js"></script>
    </body>
</html>
```

Now we are ready to create our script.

First let’s create the function that will add the two input elements and display the

answer. We'll call it update.

To write the update function, we basically reuse the code that we wrote at the console prompt in module 9.15.

```
function update() {

    // Get the two input numbers - default to 0 if a number is missing

    var firstNumber = parseFloat(document.getElementById("first").value) || 0;

    var secondNumber = parseFloat(document.getElementById("second").value) || 0;

    // then  compute the sum

    var myAnswer = firstNumber + secondNumber;

    // and write it in the "answer" element

    document.getElementById("answer").textContent = myAnswer;

};
```

Note that when an input value is the empty string '', parseFloat('') is NaN. In that case we want to use 0 for the value of the corresponding number.

Once we have defined the update function, we only need to add the following two lines to our main program.

```
document.getElementById("first").addEventListener("input", update, false);
document.getElementById("second").addEventListener("input", update, false);
```

And we have registered our function update with "input" events on the two input elements.

The **event type** here is "**input**".

We have two event targets:

One **target** is the element whose id is "first". That's the **first input**.

The other **target** is the element whose id is "second". That's the **second input**.

The **update** function is our **event listener** (or handler).

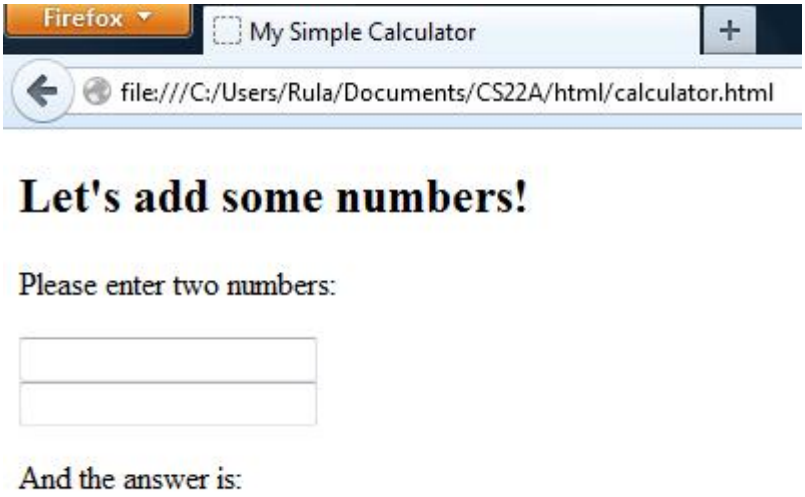
Let's save our complete program add.js in our scripts folder and test it with our web page.

add.js

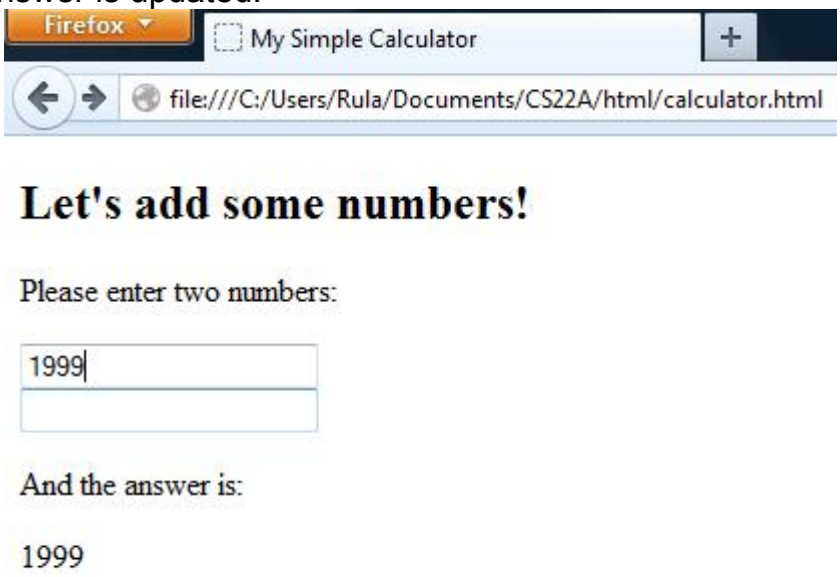
```
function update() {
    // Get the two input numbers - default to 0 if a number is missing
    var firstNumber = parseFloat(document.getElementById("first").value) || 0;
    var secondNumber = parseFloat(document.getElementById("second").value) || 0;
    // then  compute the sum
```

```
    var myAnswer = firstNumber + secondNumber;
    // and write it in the “answer” element
    document.getElementById("answer").textContent = myAnswer;
};
document.getElementById("first").addEventListener("change", update, false);
document.getElementById("second").addEventListener("change", update, false);
```

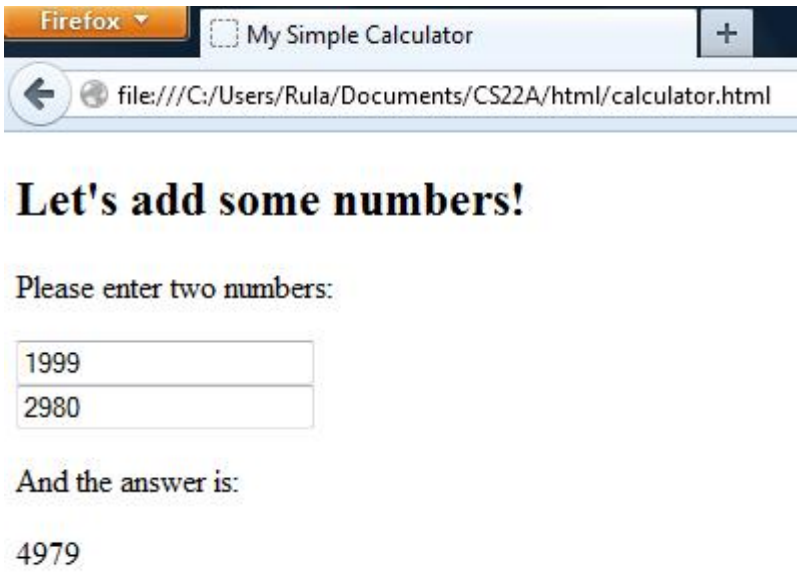
Let’s open the updated calculator.html file in Firefox. The following is displayed:



As soon as we enter a number, the input event is triggered, the function update() is called and the answer is updated.



As soon as we enter the second number the result is updated again:



The input event has been triggered again (on the second input this time) and the function update has been called again.

Every time we change one of the input fields, an input event will be triggered, the function will be called and the answer will be updated.

When we don't change any input, no change event is triggered and our application is not doing anything: it's just sitting there and waiting…

Let's add one more event listener to our calculator web page.

We'll register a help function that will be invoked **when the mouse is over the answer.** It will provide some obvious explanation as to how the answer was computed. We can do that by adding the following line to our add.js script:

```
document.getElementById("answer").addEventListener("mouseover", help, false);
```

The **event type** here is "**mouseover**".

The event **target** is the element whose id is "answer".

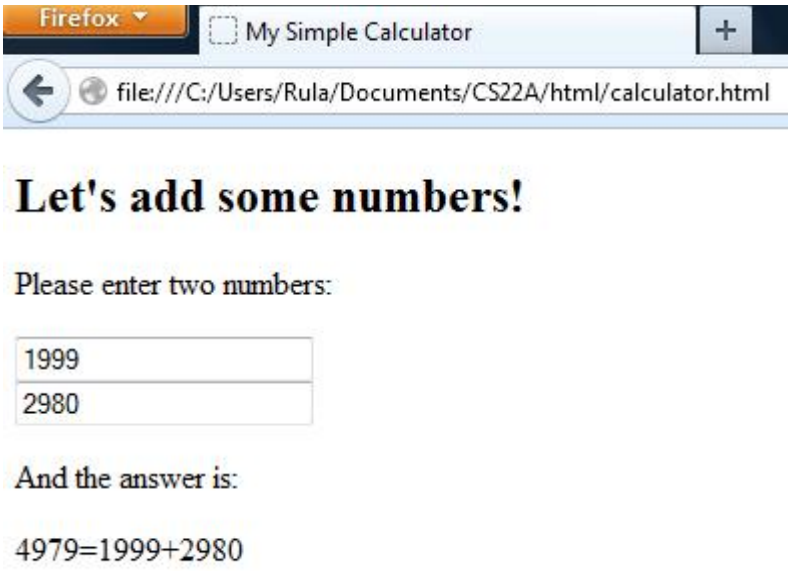
The **help** function is our **event listener** (or handler).

And now we need to define our help function.

```
function help() {

    // check that there is an answer currently displayed

      var currentAnswer = parseFloat (document.getElementById("answer").textContent);

      if (currentAnswer){

            document.getElementById("answer").textContent =

                  currentAnswer +

                  '=' +
```

```
                document.getElementById("first").value +
                '+' +
                document.getElementById("second").value;
    }
};
```

And we can see this new feature in action when we move our mouse over the answer in the web page.



10.3. The Event Object

The event object is passed as an argument to the event listener function.

Its properties provide details about the event. The type property, for example, specifies the type of the event that occurred and the target property specifies the target of the event.

The update function that we saw earlier did not use the event object. However there are instances where a listener is registered for several event types and on several targets and it is useful for the function to be able to access the specifics for a given invocation.

There are also some other details that may be relevant for a subset of events and/or targets. We'll illustrate that with a simple example that uses the HTML5 canvas element.

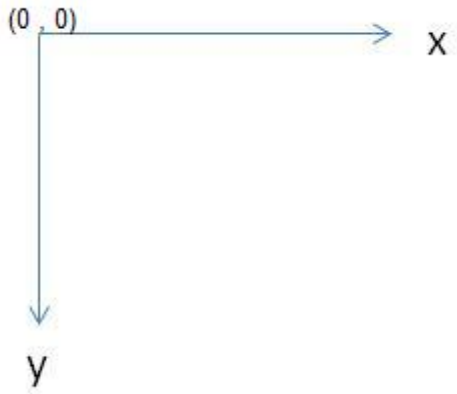
Here's some background on the <canvas> element that will make it easier to understand what's going on. For more details, check out the HTML5 reference.

The <canvas> element is used to draw graphics, on the fly, on a web page. It basically gives us a drawing surface inside the web page. Every canvas has a drawing context object. That is where the drawing methods and properties are defined. To access them, we first call getContext("2d") to get the context object. We

can then invoke the various drawing methods on the context object. In our example, we'll invoke the fillRect() method on the context object to draw a rectangle at a given position on the canvas.

The **origin of the coordinate system, for the canvas as well as the web page is at the upper left corner**, with the x coordinate increasing toward the right, and the y coordinate increasing toward the bottom:

The **base unit is the pixel**, with the top left pixel having coordinates (0,0).



We'll first create a source document, draw.html that defines a canvas element as follows:

draw.html

```
<!DOCTYPE html>
<html>
        <head>
    <title>Let's Draw</title>
  </head>
  <body>
    <h2> Just click inside the box </h2>
    <canvas id="myCanvas" width="300" height="300" style="border:1px solid #c3c3c3;">
    </canvas>
    <script async src="../scripts/draw.js"></script>
  </body>
</html>
```

Note that the height and width attributes specify the size of the canvas. We added a border so that we can see where we are supposed to draw.

We would like to capture the user clicks on the canvas as events in JavaScript and draw a small red square at the position of each click. In order to do that, we have to somehow have access to the position of the click event. The event object has that information.

The event object clientX property contains the horizontal coordinate within the application's client area at which the event occurred. To obtain the x coordinate with respect to the canvas we'll have to adjust for the canvas leftOffset.

Similarly **the event object clientY property contains the vertical coordinate** within the application's client area at which the event occurred. To obtain the y coordinate with respect to the canvas we'll have to adjust for the canvas topOffset.

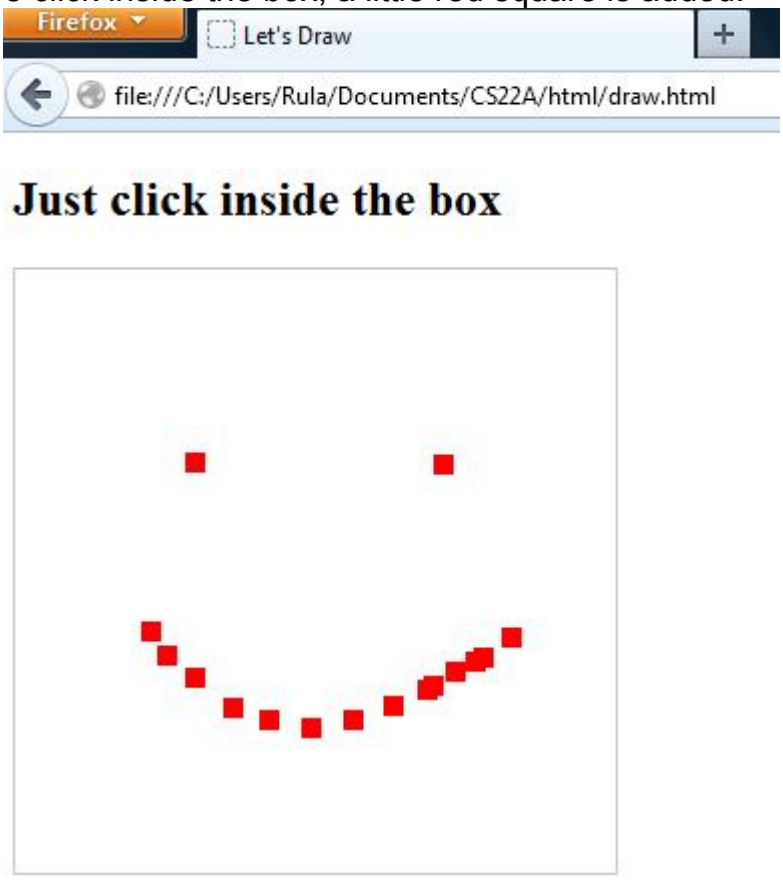
Here's our program:

draw.js

```
function drawSquare(event) {

    // Our function takes the event object as a parameter

    var myCanvas=document.getElementById("myCanvas");

    // compute the coordinates with respect to the canvas

    var x = event.clientX - myCanvas.offsetLeft;

    var y = event.clientY - myCanvas.offsetTop;

    // to access the canvas, we need its context

    var myContext = myCanvas.getContext("2d");

    // set the color to red

    myContext.fillStyle="#FF0000";

    // draw a 10 by 10 square starting at the click event position

    myContext.fillRect(x,y,10,10);
};

document.getElementById("myCanvas").addEventListener("click", drawSquare, false);
```

Initially, the following is displayed.

Then every time we click inside the box, a little red square is added:



10.4. Event Bubbling

So far we have encountered simple examples where the event listener is registered and invoked directly on the target that triggers it. When the first input was modified in our simple calculator example, the event registered on that first input element was invoked.

However in the general case, **most events bubble up the DOM tree**. So when an event is triggered on an element, it is also triggered on its ancestors in the DOM.

The event listener(s) registered on the target element (if any) are invoked first but then the listeners registered on the element's parent are invoked and then the listeners registered on the element's grandparent are invoked. And so on up the DOM tree…

Event bubbling allows us to **register a single listener on a common ancestor** element and handle events there.

So going back to our simple calculator example, in our add.js script, instead of registering the update listener on both target inputs, we could have registered it once on a common ancestor, say body.

So instead of:

```
document.getElementById("first").addEventListener("input", update, false);

document.getElementById("second").addEventListener("input", update, false);
```

We could have written:

```
document.body.addEventListener("input", update, false);
```

Note that in this case, the listener function can still figure out the specific descendant element that the event was triggered on by using the **target property of the event object** passed to the function. The element that the function is currently invoked on is accessible via by both the **currentTarget property of the event object** as well as by **'this'**. Let's go back to our update function and add some logging statements to illustrate that:

```
function update(event) {

    console.log('this:', this.nodeName);

    console.log('event.target:', event.target.id);

    console.log('event.currentTarget: ', event.currentTarget.nodeName);

    // Get the two input numbers - default to 0 if a number is missing

    var firstNumber = parseFloat(document.getElementById("first").value) || 0;

    var secondNumber = parseFloat(document.getElementById("second").value) || 0;

    // then  compute the sum
```

```
    var myAnswer = firstNumber + secondNumber;

    // and write it in the "answer" element

    document.getElementById("answer").textContent = myAnswer;

};
```

When the event is triggered by entering a number in the first input, we get:

this: BODY

event.target: first

event.currentTarget: BODY

And when the event is triggered by entering a number in the second input, we get:

this: BODY

event.target: second

event.currentTaget: BODY

The listener may also stop further bubbling by invoking the **stopPropagation()** method on the event object:

event.stopPropagation()

Finally, we said **most events bubble**. There are some events such as the focus, blur and scroll events that don't. In that case, the event listener has to be registered directly on the element where the event will be triggered.

10.5. Event Capturing

Event capturing is an event propagation method that goes in the opposite order as event bubbling.

Remember the third argument to the addEventListener() method? We have been setting it to false in our examples. It is actually a 'useCapture' parameter. When set to true, events are propagated down the DOM tree, from parent to child.

Let's say an event is triggered on a target element.

The event listeners registered on the highest ancestor of that target element are invoked first and then the listeners registered on the target element second highest ancestor and so on down the DOM tree…

Let's illustrate the difference between the two propagation methods with an example:

```
<!DOCTYPE HTML>
<html>
    <body>
        <div id="grandParent">
            <div id="parent">
```

```
            <p>Just enter anything below to see event propagation in action </p>
            <input id = "myInput" type="text">
        </div>
    </div>
    <script async src="../scripts/propagate.js"></script>
  </body>
</html>
```

We'll write some event listener functions that simply include some logging statements to identify the function that was invoked and the element it was invoked on.

We'll first register all these functions with the useCapture parameter set to false.

propagate.js

```
function grandParentHandler(event) {
    console.log('Grandparent handler');
    console.log('this:', this.id);
    console.log('event.target:', event.target.id);
};
function parentHandler(event) {
    console.log('Parent handler');
    console.log('this:', this.id);
    console.log('event.target:', event.target.id);
};
function elementHandler(event) {
    console.log('Event handler');
    console.log('this:', this.id);
    console.log('event.target:', event.target.id);
};
document.getElementById("grandParent").addEventListener("input",grandParentHandler,false);
document.getElementById("parent").addEventListener("input",parentHandler,false);
document.getElementById("myInput").addEventListener("input",elementHandler,false);
```

With the useCapture parameter set to false, events bubble up: the target event

handler in invoked first, then the parent's handler and then the grandparent's handler.

Event handler

this: myInput

event.target: myInput

Parent handler

this: parent

event.target: myInput

Grandparent handler

this: grandParent

event.target: myInput

Let's see what happens when the useCapture parameter is set to true.

First we change the last 3 lines in propagate.js to:

```
document.getElementById("grandParent").addEventListener("input",grandParentHandler,true);

document.getElementById("parent").addEventListener("input", parentHandler, true);

document.getElementById("myInput").addEventListener("input", elementHandler, true);
```

And the invocation order is reversed as we can see in our console output:

Grandparent handler

this: grandParent

event.target: myInput

Parent handler

this: parent

event.target: myInput

Event handler

this: myInput

event.target: myInput

Note that bubbling is much more common than capturing.

11. Scripting Style

11.1. Separation of Layers

So far our web pages have included content, provided by the HTML layer, and a certain behavior, implemented in JavaScript. It's about time we add some 'style' with the CSS layer.

CSS stands for Cascading Style Sheets. Styles define the presentation of a web

page, how HTML elements will be displayed.

There are several ways to associate styles with an HTML document.

In this course, we'll stick to our recommended approach of separation of layers. Just as we separated our JavaScript code from HTML documents, we'll separate our CSS from both JavaScript code and from HTML.

We'll place CSS in a separate external file that contains only CSS code. That file will be saved with a 'css' extension and will be referenced in our HTML using the <link> tag. An advantage of this approach is that we can then change the appearance of several web pages by editing a single css file.

Let's add an external style sheet to our classexample.html document:

```
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript for Programmers</title>
        <link rel = "stylesheet" type = "text/css" href = "mystyle.css" media = "all">
    </head>
    <body>
        <h1 class = "important">Layout Engines</h1>
        <p id = "mainidea" >
            <span id = "first" class = "info">The different browsers rely on layout engines. </span>
            <span id = "second" class = "important info">
                The different layout engines implement the DOM standards to varying degrees of
compliance.
            </span>
        </p>
    </body>
</html>
```

Note that the <link> tag must be placed inside the <head> element.

The <link> tag has several attributes:

The href attribute specifies the location of our external css file.

The rel attribute specifies the relationship between the HTML document and the linked document: in our case it's the stylesheet.

The type attribute describes the type of the linked document: it's "text/css".

Finally the media attribute specifies the media or device that the stylesheet is to be associated with. Valid media include "all", "handheld", "print", "screen" among others.

We have seen how to associate an external stylesheet with an HTML document but we have not seen what goes in that stylesheet.

CSS is a powerful language with a simple syntax. The in-depth study of CSS is beyond the scope of this course. We'll just demonstrate the basics with some examples.

A CSS stylesheet includes a set of **rules**.

Each rule consists of one or more **selectors**, and a **declaration block**.

```
selector {
    declaration block
}
```

The selector is the HTML element you want to style. The declaration block may contain several declarations. Each declaration consists of a property and a value. The property is the style attribute you want to change. Each property has a value.

You may select an element by its tag.

To define a style on <p> elements, you use p as your selector. You can include comments between /* and */. You may include more than one declaration in each rule. You separate them with a semicolon.

```
p {  /* the selector "p" matches all <p> elements */
color:blue;
font-size:20px;
}
```

To define a background color for the whole document, you write the following rule:

```
body {
    background-color:#c0e4fe;
}
```

You may also refer to the element by its id. You just prefix the id with #.

So to style the element with the id second, you use #second as your selector.

```
#second {  /* the selector matches the element with id "second" */
    color:red;
}
```

The red color for the more specific rule here will override the blue property for the more general p rule. The font-size will be used from the p rule.

Finally, **you may also refer to several elements by their class. You prefix the class name with a dot.**

So to style all elements the element with class = “important”, you use .important as

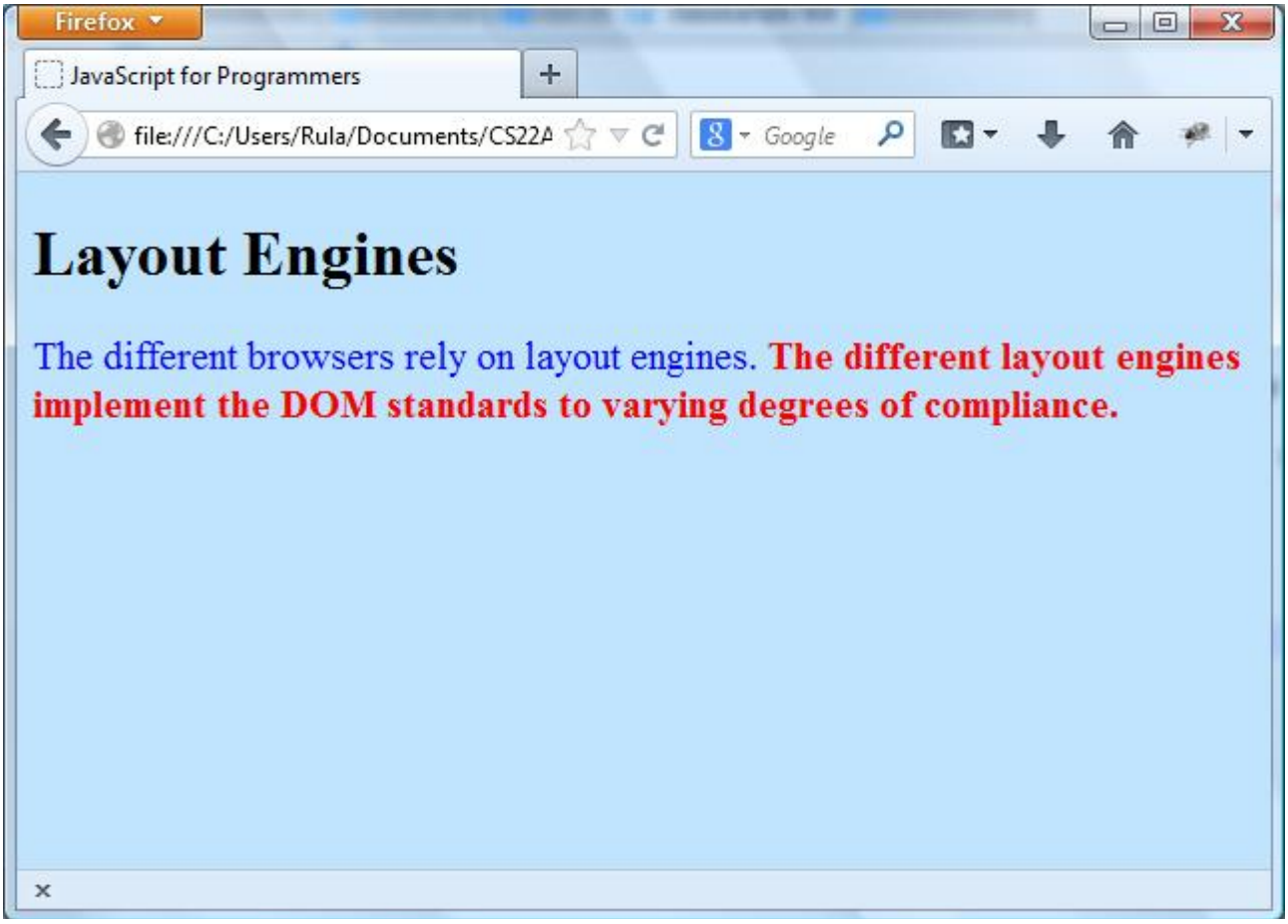
your selector.

```
.important { /* the selector matches all elements with class "important" */
font-weight: bold;
}
```

We are now ready to create our stylesheet, mystyle.css.

```
body {
    background-color:#c0e4fe;
}
p { /* the selector "p" matches all < p > elements */
    color:blue;
    font-size:20px;
}
#second {  /* the selector matches the element with id "second" */
    color:red;
}
.important {
    font-weight: bold;
}
```

You will see the changes if you open the modified classexample.html in Firefox. Just make sure you added the <link> tag to the html source first.



11.3. Changing Styles with JavaScript

Now that we've seen how styles are defined in CSS, we'll see how to change these styles from JavaScript.

The recommended way to change the style of a given element is to change the value of its class attribute. In the DOM, that attribute is denoted by className.

We'll see how to do that in the following simple example.

Let's associate our classexample.html source file with a JavaScript program, styleselect.js.

```
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript for Programmers</title>
        <link rel = "stylesheet" type = "text/css" href = "mystyle.css" media = "all">
    </head>
    <body>
        <h1 class = "important">Layout Engines</h1>
        <p id = "mainidea" >
            <span id = "first" class = "info">The different browsers rely on layout engines. </span>
```

```
                <span id = "second" class = "important info">
                    The different layout engines implement the DOM standards to varying degrees of compliance.
                </span>
            </p>
        <script async src="../scripts/styleselect.js"></script>
        </body>
</html>
```

Let's also modify our style sheet, mystyle.css, to add a new rule for class="selected" as shown below.

```
body {
    background-color:#c0e4fe;
}
p { /* the selector "p" matches all < p > elements */
    color:blue;
    font-size:20px;
}
#second {  /* the selector matches the element with id "second" */
    color:red;
}
.important {
    font-weight: bold;
}
.selected {
    color:green;
}
```

Now we create our JavaScript program, styleselect.jsprogram as follows:

```
function green(event) {
    event.target.className = event.target.className + " selected";
};

document.body.addEventListener("mouseover", green, false);
```

Here we have an event listener registered on the body of the document for a mouseover event. So whenever our mouse goes over a target element, the function green adds "selected" to its class attribute. We do that because we don't want to overwrite its existing class, we just want to add "selected" to it. Remember that className specify multiple class names separated by a space.

Once the class of the given attribute is changed to selected, its text color is changed to

green as per the CSS stylesheet.

Note again that the more specific CSS rules override the more general rules.

So the class selector .selected is more specific than the tag selector p. Hence a p element with a class = “selected” will be green not blue.

On the other hand the id selector #second is more specific than the class selector .selected so the element “second” will still be red even though it belongs to the selected class too.

11.4. Transition Example

We have seen how to use the timer functions available in JavaScript to create animated effects in JavaScript. CSS3 now defines a way to specify transitions and animations in stylesheets.

We can add a transition effect when changing from one style to another. W**e specify the transition for the class we are transitioning into.** We also specify **each CSS property** we want to add an effect to and the **duration of the transition**.

CSS transitions are supported in the current browsers; however Safari requires the prefix -webkit-.

To see how transitions can be controlled from JavaScript, we’ll first create a web page that implements a zooming transition effect, tree.html.

```
<!DOCTYPE html>
<html>
    <head>
        <title>JavaScript for Programmers</title>
        <link rel = "stylesheet" type = "text/ css" href = "transitions.css" media = "all">
    </head>
    <body>
      <h2>Transition Demo</h2>
        <p>We create the zooming transition in CSS and trigger it from JavaScript.</p>
        <br>
        <input id = "mybutton" type="button" value="Press Here">
        <div>
            <img id="tree" src="tree.png" alt="tree" class = "small">
        </div>
    <script async src="../scripts/tree.js"></script>
    </body>
</html>
```

And the corresponding stylesheet defines styles for the “small” and “large” classes with transitions on width and height.

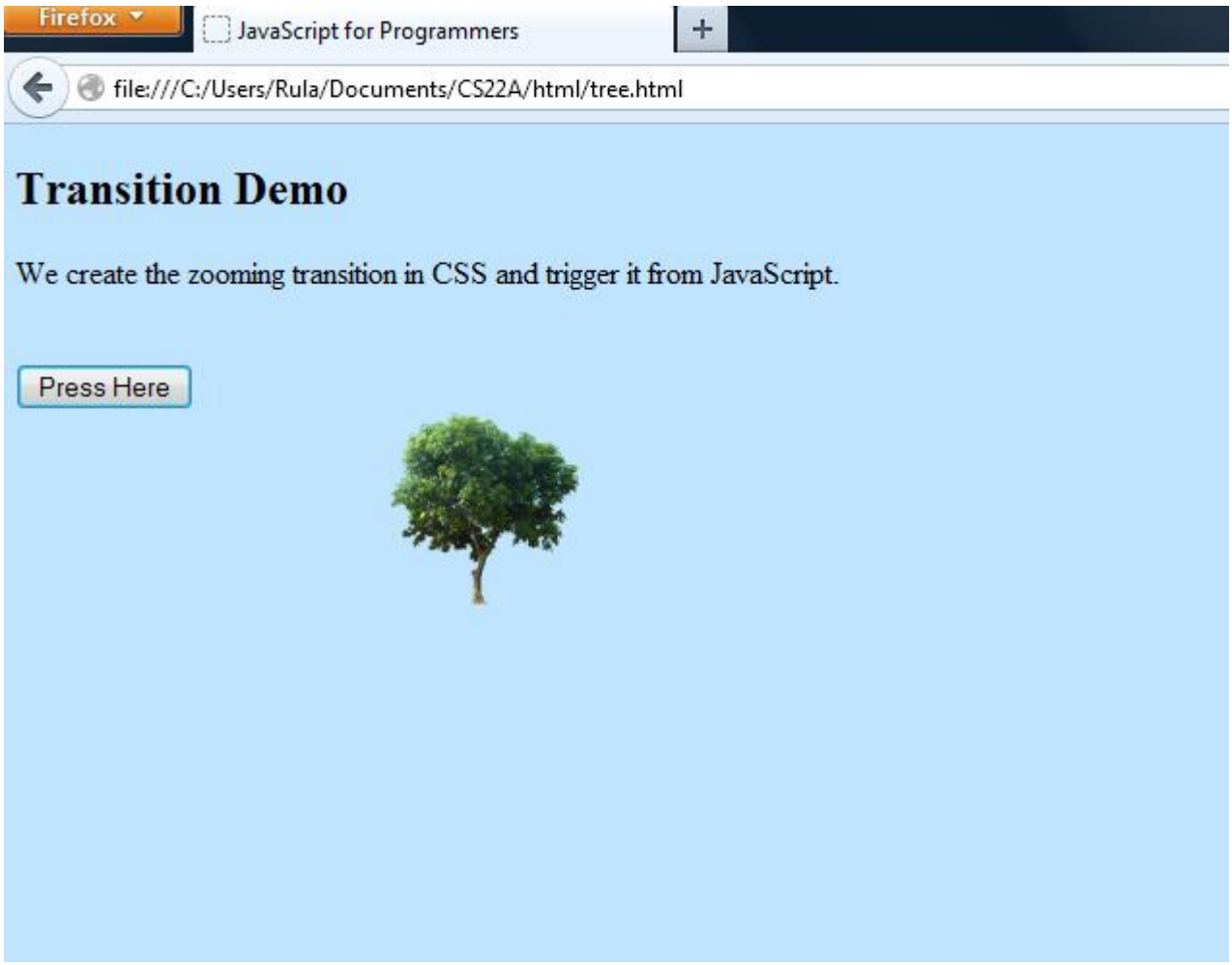
transitions.css

```
body {
    background-color:#c0e4fe;
}
.small{
    width:100px;
    height:100px;
    position: absolute;
    left:200px;
    transition:height 5s, width 5s;   /* transition on height&width, duration 5 seconds */
    -webkit-transition:height 5s, width 5s; /* Safari */
}
.large{
    width:300px;
    height:300px;
    position: absolute;
    left:200px;
    transition:height 5s, width 5s;   /* transition on height&width, duration 5 seconds */
    -webkit-transition:height 5s, width 5s; /* Safari */
}
```

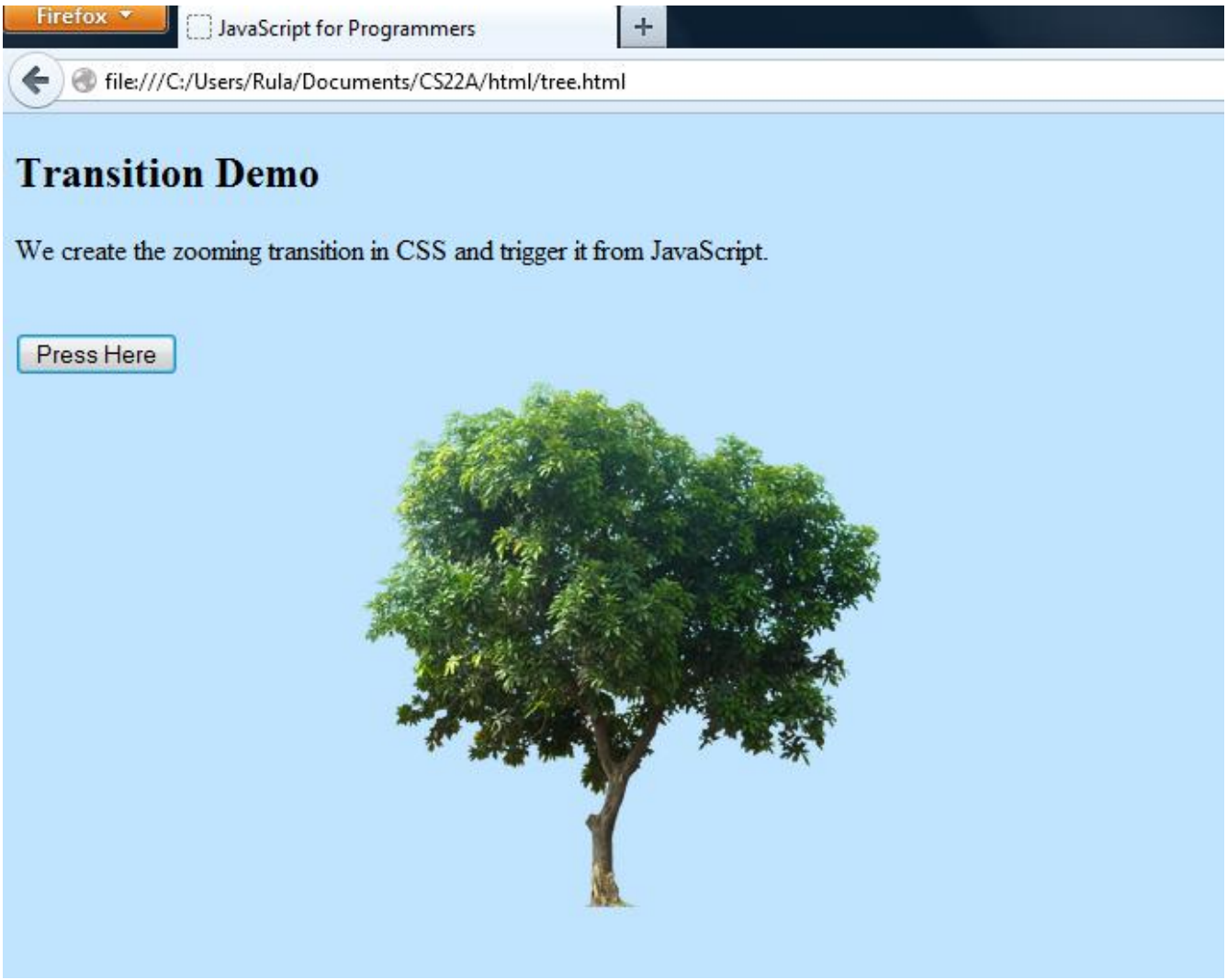
Finally we write our JavaScript program tree.js as follows:

```
function toggle(event) {

    if (document.getElementById("tree").className === "small"){

        document.getElementById("tree").className = "large";

    }else {

        document.getElementById("tree").className = "small";

    }

};

document.getElementById("mybutton").addEventListener("click", toggle, false);
```

The tree picture tree.png is available under Resources. You can see how the transition works by opening the tree.html file and pressing the button a couple of times.



And after you press the button:



11.5. Animation Example

In our next example, we'll create an animation in CSS and control it from JavaScript.

To create animations in CSS3, we specify two styles inside the **@keyframes rule** and the animation will gradually change from the current style (denoted by from) to the new style denoted by 'to'.

```
@keyframes moving{
        from {
        }
        to {
                transform: translateX(800px) rotate(360deg);
        }
}
```

Here the 'moving' animation will move the element 800 pixels along the horizontal X axis and rotate it 360 degrees.

To bind the animation to a selector (such as the “move” class), we must specify a name and a duration.

We can also specify other properties:

infinite will make the animation run forever, and **alternate** will cause the animation to play in reverse on alternate cycles.

```
.move{
        animation:moving 8s infinite alternate;
}
```

CSS animations are supported in the current browsers, however Safari and Chrome require the prefix -webkit-.

So in order to support Safari and Chrome, we update our CSS stylesheet to include the following:

```
.move{
        animation:moving 8s infinite alternate;
        -webkit-animation:moving 8s infinite alternate;/* Chrome and Safari */
}
@keyframes moving{
        from {
        }
        to {
                transform: translateX(800px) rotate(360deg);
        }
}
@-webkit-keyframes moving {
        from {
        }
        to {
                transform: translateX(800px) rotate(360deg);
        }
}
```

We then create a soccer.html document that will use our stylesheet to animate a soccer ball:

```
<!DOCTYPE html>
<html>
```

```
        <head>
                <title>JavaScript for Programmers</title>
                <link rel = "stylesheet" type = "text/css" href = "animate.css" media = "all">
        </head>
        <body>
                <h2>Animation Demo</h2>
                <p>We create the animation in CSS and trigger it from JavaScript.</p>
                <input id = "gobutton" type="button" value="GO">
                <input id = "stopbutton" type="button" value="STOP">
                <p>Click on the GO button to move the ball.</p>
                <img id="ball" src="ball.png" alt="ball">
                <script async src="../scripts/animate.js"></script>
        </body>
</html>
```

We refine our stylesheet to include a background color and style the buttons as follows:

animate.css

```
body {
        background-color:#98ff70;
}
p {
        font-size:20px;
}
[type = "button"] {
        width:80px;
}
.move{
        animation:moving 8s infinite alternate;
        -webkit-animation:moving 8s infinite alternate;/* Chrome and Safari */
}
@keyframes moving{
        from {
```

```
        }
        to {
                transform: translateX(800px) rotate(360deg);
        }
}
@-webkit-keyframes moving {
        from {
        }
        to {
                transform: translateX(800px) rotate(360deg);
        }
}
```

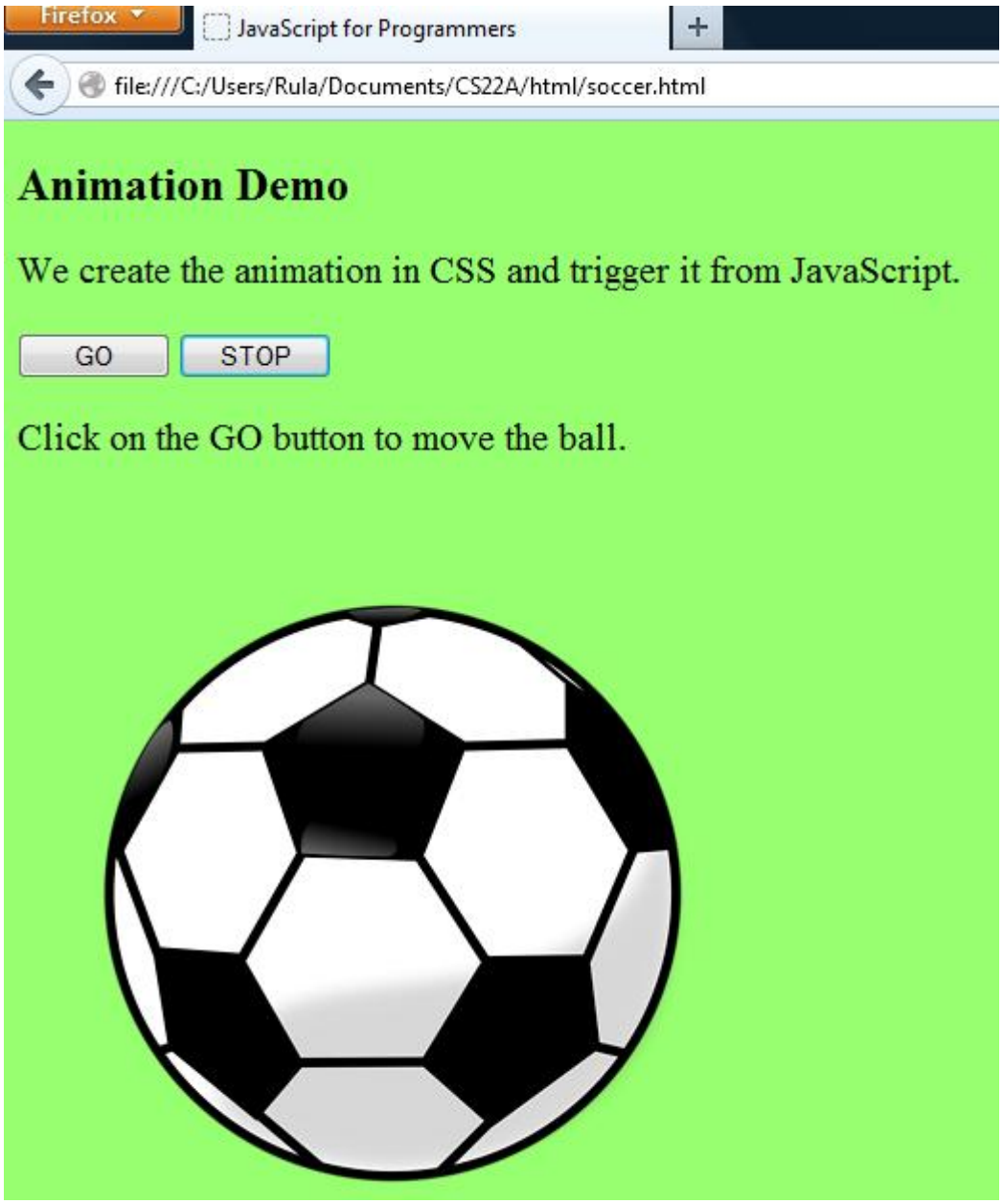
All we have left to do is create our JavaScript program to control the animation with user clicks. We create animate.js as follows.

```
function move(event) {

    document.getElementById("ball").className = "move";

};

function stop(event) {

    document.getElementById("ball").className = "stop";

};

document.getElementById("gobutton").addEventListener("click", move, false);

document.getElementById("stopbutton").addEventListener("click", stop, false);
```

The two listener functions are simply changing the class of the ball element.

Note that the file ball.png is available under Resources.

We can now open the file soccer.html start the animation by pressing on the Go button:



12. JSON

12.1. What is JSON?

JSON stands for JavaScript Object Notation. It is a data interchange format. It is used in the transmission of data between machines.

JSON is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of JavaScript but it is completely language independent.

JSON is built on two structures:

- A collection of name/value pairs similar to a JavaScript object.
- An ordered list of values similar to a JavaScript array.

In addition to arrays and objects, JSON supports strings, finite numbers, true, false, and null values.

The value undefined as well as the numbers NaN and Infinity are not representable in JSON.

The following examples all illustrate valid JSON format:

{"name":"Fido","hungry":false,"lonely":false,"unhealthy":false}

{"name":"Wanda","hungry":false,"unhealthy":false}

[100, 80, 90, 89, 99]

12.2. Object Serialization

Object serialization is the process of converting an object to a string from which it can later be restored. The function **JSON.stringify()** may be used in **ECMAScript 5** to serialize JavaScript objects and convert them to JSON format.

The **JSON.stringify() function converts an object, an array or a primitive value to JSON.**

Usually, this function is called with a single argument and returns the corresponding string.

Examples:

>>> JSON.stringify(6.3);

"6.3"

>>> JSON.stringify(false);

"false"

>>> JSON.stringify(myDog)

"{"name":"Fido","hungry":false,"lonely":false,"unhealthy":false}"

>>> var grades = [100, 80, 90, 89, 99]

>>> JSON.stringify(grades)

"[100,80,90,89,99]"

JSON.stringify() takes an optional **replacer** argument that may be one of two things:

- a function that will replace values before converting to string.
- an array that contains the names of properties to be included in the output. Note that the name 'replacer' is misleading in this case because the array is acting more like a filter rather than a replacer.

You can also specify an optional **space** argument that provides an indentation string or the number of spaces to use for indentation in order to get a more readable output.

If you specify a number, JSON.stringify() will insert newlines and use the specified number of spaces to indent each level of output.

If you specify a non-empty string instead, JSON.stringify() will insert newlines and use that string to indent each level.

Examples:

In the example below we only want the name and the hungry properties to be included in the output so we specify the array [‘name’, ‘hungry’] as our ‘replacer’ argument. We also specify 3 as our space argument, to have each level indented by 3 spaces.

```
>>> JSON.stringify(myDog, ['name', 'hungry'], 3)
"{
  "name": "Fido",
  "hungry": false
}"
```

If we don’t want to specify a replacer but still want to include a space argument, we can use null for the replacer. We can also specify any string to be used for indentation:

```
>>> JSON.stringify(myDog, null,  '--->');
"{
--->"name": "Fido",
--->"hungry": false,
--->"lonely": false,
--->"unhealthy": false
}"
>>>var restaurant=[{name: 'Red Apple', price: 3, rating: 2, distance: 2},
                  {name: "Zoe's Place", price: 1, rating: 5, distance: 1},
                  {name: 'Yumm', price: 2, rating: 3, distance: 5},
                  {name: 'California Foods', price: 5, rating: 4, distance: 8}];

>>> JSON.stringify(restaurant, null, 2);
"[
 {
  "name": "Red Apple",
  "price": 3,
  "rating": 2,
  "distance": 2
 },
 {
  "name": "Zoe's Place",
  "price": 1,
```

```
  "rating": 5,
  "distance": 1
 },
 {
  "name": "Yumm",
  "price": 2,
  "rating": 3,
  "distance": 5
 },
 {
  "name": "California Foods",
  "price": 5,
  "rating": 4,
  "distance": 8
 }
]"
```

12.3. Parsing JSON

The function **JSON.parse()** may be used **in ECMAScript 5** to parse JSON formatted strings.

The JSON.parse() function returns the object, array, or primitive value that the JSON string represents.

Examples:

```
>>> JSON.parse("-8.7");
-8.7
>>> JSON.parse("true");
true
>>> JSON.parse("[100, 87, 90]");
[100, 87, 90]
>>> var myString = '{"name":"Alice","phone":"555-1234"}'
>>> JSON.parse(myString);
Object { name="Alice", phone="555-1234"}
```

JSON.parse() takes an optional **reviver** argument that will transform the parsed value before it is returned.

For example the function standardize() below may be used to standardize the value of the country property after the parsing:

```
function standardize( propName, value) {
   // Standardize the country property
   if (propName === "country"){
      return "USA";
   };
 // Leave other properties unchanged
   return value;
};
>>> var myString = '{"name":"Alice", "country":"United States", "phone":"555-1234"}';
>>>JSON.parse(myString, standardize);
Object { name="Alice", country="USA", phone="555-1234"}
```

12.4. Deep Copy with JSON

To make a deep copy of an array (or any object) in JavaScript, we can simply convert it to JSON format and then parse it back.

Going back to our shallow copy example from module 7.9:

```
>>>var charlie = [100,98,[85,90],85];
>>>var diana=JSON.parse(JSON.stringify(charlie));
>>> diana
 [100, 98, [85, 90], 85]
>>> charlie[2][1]=0;
>>> charlie ;
[100, 0, [85, 0], 85]
>>> diana
 [100, 98, [85, 90], 85]
```

Or we can write a function to perform that:

```
// Make a deep copy of an array
function deepcopy(a) {
        return JSON.parse( JSON.stringify(a));
}
```

12.5. JSON vs XML

XML is another format used to structure, store, and transfer data.

XML is based on a tree structure, and uses tags to delimit elements (like HTML). However the tags are user-defined. So for example to represent the information included in the JSON format:

{"name":"Alice", "country":"United States", "phone":"555-1234"}

We could use the following in XML:

```
<person>
    <name>Alice</name>
    <country>United States</country>
    <phone>555-1234</phone>
</person>
```

Both JSON and XML use plain text and both formats are quite readable. In addition, both use hierarchical structures.

JSON results in shorter encodings for the same data, mainly because it does not require any closing tags. JSON also supports arrays. Above all, JSON is easier to parse and generate from within JavaScript as well as from many other programming languages.

For AJAX applications, JSON has become the preferred data interchange format.

13. Namespace Considerations

13.1. Global Variables and Functions as Properties

In a JavaScript program, when we declare a variable outside of any function, that variable becomes a property of the global object. Named functions that we declare in our program are also properties of the global object. In client side JavaScript that global object is window.

All we have to do to see the potential problem is open a web page in Firefox, go to the Firefox console and type 'window.' : Firebug brings up a list of all the enumerable properties defined on window. Some of these properties are global functions such as alert and setInterval and some are global variables. In addition, when we use the id attribute in our HTML document, a property corresponding to that id is added to the window object. So in our calculator example from module 10, when we added the ids "first" and "second" to our input elements, the window object got two properties window.first and window.second whose value is the corresponding input element.

All of this makes it harder to name global variables and functions in our program that do not conflict with existing property names.

We'll look at two approaches to mitigate the problem next.

13.2. Object as Namespace

One way to minimize the risk of name collision is to create a single global variable for our application: var myApp = {};

The object referenced by that variable then becomes the container for our application.

Our global variables will then be defined as properties of myApp and our functions will be defined as methods of myApp. The only variable that will be added to the global namespace is myApp. In client side JavaScript, the only property that will be added to the window object is myApp.

Let's go back to our add.js program that we used with our calculator.html document. Even though we did not define any global variables there, we had two functions update() and help(). We can define them as methods of myApp as follows:

```
var myApp = {};
myApp.update = function() {
    // Get the two input numbers
    var firstNumber = parseFloat(document.getElementById("first").value) || 0;
    var secondNumber = parseFloat(document.getElementById("second").value) || 0;
    // Then  compute the sum
    var myAnswer = firstNumber + secondNumber;
    // And write it in the "answer" element if it is a valid answer
    document.getElementById("answer").textContent = myAnswer;
};
myApp.help = function() {
    // check that there is an answer currently displayed
    var currentAnswer = parseFloat (document.getElementById("answer").textContent);
    if (currentAnswer){
        document.getElementById("answer").textContent =
            currentAnswer +
            '=' +
            document.getElementById("first").value +
            '+' +
            document.getElementById("second").value;
    }
};
document.getElementById("first").addEventListener("input", myApp.update, false);
document.getElementById("second").addEventListener("input", myApp.update, false);
document.getElementById("answer").addEventListener("mouseover", myApp.help, false);
```

Note that we have to specify myApp.update and myApp.help now in the addEventListener() invocation.

If we have global variables in our program, we can also define them as properties of myApp:

myApp.myGlobalVar = …

Note that another way to define the methods on myApp is within the curly braces as follows:

```
var myApp = {
    update: function() {
        // Get the two input numbers
        var firstNumber = parseFloat(document.getElementById("first").value) || 0;
        var secondNumber = parseFloat(document.getElementById("second").value) || 0;
        // Then  compute the sum
        var myAnswer = firstNumber + secondNumber;
        // And write it in the "answer" element if it is a valid answer
        document.getElementById("answer").textContent = myAnswer;
    },
    help: function() {
        // check that there is an answer currently displayed
        var currentAnswer = parseFloat (document.getElementById("answer").textContent);
        if (currentAnswer){
            document.getElementById("answer").textContent =
                currentAnswer +
                '=' +
                document.getElementById("first").value +
                '+' +
                document.getElementById("second").value;
        }
    }
};
document.getElementById("first").addEventListener("input", myApp.update, false);
document.getElementById("second").addEventListener("input", myApp.update, false);
document.getElementById("answer").addEventListener("mouseover", myApp.help, false);
```

13.3. Function as Namespace

Another approach to minimize the risk of name collision is to **use a function as a container for all our code and then invoke that function**. The function is then just a temporary namespace for our code.

Let's go back to our initial calculator example and implement that approach:

```
function myModule(){
    function update() {
        // Get the two input numbers
```

```
        var firstNumber = parseFloat(document.getElementById("first").value) || 0;
        var secondNumber = parseFloat(document.getElementById("second").value) || 0;
        // Then  compute the sum
        var myAnswer = firstNumber + secondNumber;
        // And write it in the "answer" element if it is a valid answer
        document.getElementById("answer").textContent = myAnswer;
    };
    function help() {
        // check that there is an answer currently displayed
        var currentAnswer = parseFloat (document.getElementById("answer").textContent);
        if (currentAnswer){
            document.getElementById("answer").textContent =
                currentAnswer +
                '=' +
                document.getElementById("first").value +
                '+' +
                document.getElementById("second").value;
        }
    };
    document.getElementById("first").addEventListener("input", update, false);
    document.getElementById("second").addEventListener("input", update, false);
    document.getElementById("answer").addEventListener("mouseover", help, false);
}
myModule();
```

We have to make sure we call that function that we just declared.

The only addition to the global namespace here is myModule.

If that's too much, **we can even define an anonymous function and invoke it** as follows. Note that for this to work, the whole thing has to be enclosed in parentheses.

```
(function(){
    function update() {
        // Get the two input numbers
        var firstNumber = parseFloat(document.getElementById("first").value) || 0;
        var secondNumber = parseFloat(document.getElementById("second").value) || 0;
        // Then  compute the sum
        var myAnswer = firstNumber + secondNumber;
```

```
        // And write it in the "answer" element if it is a valid answer
        document.getElementById("answer").textContent = myAnswer;
    };
    function help() {
        // check that there is an answer currently displayed
        var currentAnswer = parseFloat (document.getElementById("answer").textContent);
        if (currentAnswer){
            document.getElementById("answer").textContent =
                currentAnswer +
                '=' +
                document.getElementById("first").value +
                '+' +
                document.getElementById("second").value;
        }
    };
    document.getElementById("first").addEventListener("input", update, false);
    document.getElementById("second").addEventListener("input", update, false);
    document.getElementById("answer").addEventListener("mouseover", help, false);
}());
```

14. Libraries and Frameworks

14.1. Overview

So far in this course, we have focused on writing JavaScript code that works in most of the current browsers. That is not a luxury we have in the industry. The challenge in client side JavaScript programming is to deal with all the browser inconsistencies. For this reason, it is helpful to build JavaScript code on top of some of the available frameworks. These frameworks provide a higher level interface that is compatible and tested across all browsers.

Several JavaScript frameworks are available today. Here are just a few:

Prototype: http://prototypejs.org/

The Prototype library includes DOM and Ajax utilities.

YUI – http://yuilibrary.com/

YUI is the Yahoo! User Interface Framework. It is an open source JavaScript library for building interactive web applications. It includes language utilities, DOM utilities, as well as user interface widgets.

Dojo – http://dojotoolkit.org/

Dojo is also an open source library that includes UI widgets, DOM manipulation

utilities, a system for managing module interdependencies and more.

Closure - https://developers.google.com/closure/library/

The closure library is the JavaScript library that Google uses for Gmail.

jQuery - http://jquery.com/

Although jQuery does not do anything JavaScript cannot do, it has become extremely popular for several reasons:

- It uses a simple syntax (CSS selectors) for referring to elements in the document.
- It provides a simple way to operate on sets of elements as a group, rather than one at a time.
- It removes the need to deal with the browser incompatibilities in our code.
- There are several open source plugins available for jQuery, including a popular user interface toolkit jQuery UI.

We'll cover jQuery in more details in the next few sections.

14.2. Getting Started with jQuery

In this course, we'll download the jQuery library to our scripts directory and then refer to it in our web pages with the src attribute of the <script> tag. This is called serving our own copy.

In production code, we can also use a hosted version from a Content Delivery Network such as Google or Microsoft: the main advantage of doing that is that users will likely already have a copy in the browser's cache and no download will be necessary.

To download jQuery go to: http://jquery.com/

We'll use the uncompressed, development jQuery 1.11.0 since that version is best for debugging purposes. As you can see, that's just a JavaScript file. We'll save it in our scripts directory as jquery-1.11.0.js.

To use jQuery, we now include the following in our HTML document:

< script defer src =" ../scripts/jquery-1.11.0.js" > </ script >

14.3. jQuery Demo

At the core of jQuery is the ability to select some elements and do something with them. The basic syntax is:

$(selector).action()

The $ is an alias for the function called jQuery, so we could also use the following construct:

jQuery(selector).action()

jQuery supports most CSS3 selectors.

To follow along, download the following files available under Resources:

jqdemo.html

jqdemo.css

tree.png

car.png

You can then open jqdemo.html in Firefox, and use the Firebug console to try the commands below.

We can select elements by their tags:

>>> $('p'); // all the <p> elements in the document

Object[p, p, p]

We can select elements by their id:

>>> $('#profile'); //the selector matches the element with id "profile"

Object[div#profile]

We can select elements by class name:

>>> $('.important'); // the selector matches all elements with class important.

Object[span.important, li.important]

We can combine selectors:

$('li.important'); // the selector matches all li (list item) elements with class important

Object[li.important]

The selector below matches all <input> elements that are descendants of an element that has the id profile.

>>> $("#profile input")

Object[input#name property value = "" attribute value = "null", input#email property value = "" attribute value = "null"]

Notice the square brackets around the object returned by $.

The jQuery object returned by $ is an array like object. So to check whether a selection contains any elements, we can check its length property:

>>> $('.important').length;

2

>>> $('h1').length;

0

To access particular elements in that returned object, we can use the square brackets.

>>> $('.important')[0];

>>> $('.important')[1];

<li class="important">

Once we have a selection, we can call methods on the selection. When a method is used to get (or read) a value, it is called a getter. **In general, getters only get the value of the first element in the selection.**

When a method is used to set a value, it is called a setter. **Setters affect all elements in a selection. Setters also return the jQuery object allowing us to chain methods**.

Here are just some examples of the methods available on jQuery objects. For a full list, check out http://api.jquery.com/ .

Note jQuery uses the same method name for getter and setter.

Unlike other jQuery setters, the text method returns the text content of all the elements in the selection.

$('li').text(); / /text getter – get the text content of all **li elements**.

>>> $('li').text();

"It uses a simple syntax (CSS selectors) for referring to elements in the document. It provides a simple way to operate on sets of elements as a group, rather than one at a time.It removes the need to deal with the browser incompatibilities in our code.There are several open source plugins available for jQuery, including a popular user interface toolkit jQuery UI.

"

$('li').text('New Text'); //text setter - set the text content **of all li elements**.

Note that the web page has been modified and all the list items now consist of 'New Text'.

Make sure you reload the page to get back to the original text before proceeding with the examples below.

>>> $('li').html(); // get the innerHTML of the first list element

"It uses a simple syntax (CSS selectors) for referring to elements in the document. "

Enter a name and an email address in the input fields, then try the following:

>>> $('input').val() ; // val getter - get the value of **the first input element**

"Rula"

$('#profile input').val(''); // set the value of all the #profile input elements to the empty string

Note that the two input fields are cleared.

$('div').addClass('important'); // add the class important to all elements

Object[div#profile.important, div#lesson.important, div.important, div.important]

Note the change in style in the corresponding elements.

>>> $('div').removeClass('important'); // remove the class important from div elements

Object[div#profile, div#lesson, div, div]

With jQuery, we can hide and show HTML elements with the hide() and show() methods. Make sure to scroll down the page to see the tree.

>>> $('#tree').hide();   // Hide the tree

>>> $('#tree').show(); // Show the tree

With the toggle() method, hidden elements are shown and shown elements are hidden.

>>> $('#tree').hide(); // first hide the tree

Then toggle all img elements on the page:

>>> $('img').toggle();

The tree is shown, the car is hidden.

Toggle again:

>>> $('img').toggle();

Now the car is shown and the tree is hidden.

jQuery provides several methods that may be used to create animated effect even in older browser.

You can fade an element in and out of visibility.

Make sure you reload the page before you try the following:

>>> $('#tree').fadeOut("slow");

The tree slowly disappears. You can also specify the time in milliseconds.

>>> $('#tree').fadeIn(5000);

The tree reappears.

You can use animate() to change CSS properties.

>>> $('#car').animate({left:'500px'});

Event handling with jQuery

jQuery provides simple event registration methods for the most common events.

>>> $('#treebutton').click (function() { $('#tree').fadeToggle("slow");});

Now when you press on the tree button, the tree will fade in or out of visibility.

For all valid JavaScript events, **we can also use the use the method 'on' and provide the JavaScript event name:**

>>> $('#carbutton').**on**("click", function() { $('#car').fadeToggle("slow");});

And now whenever you press on the car button the car will also fade in and out of visibility.

The document ready event:

When it's important that the document object model be loaded and ready for our code to manipulate, we use the jQuery ready event as follows:

```
$(document).ready( function() {

    Include the code here

} ;
```

15. Client Server Architecture

15.1. Overview

So far we've been working with JavaScript in the browser, loading source documents from our file system.

This week, we'll cross over to the server side and see how we can use JavaScript there. We'll also implement our own web server.

Then we'll come back to the client side and cover communication with the server from that side.

Let's first take a look at the distinction between a client and a server and how these two communicate.

The browsers are all examples of clients: they request some information from a server and then allow the user to interact with that information.

A web server is a program that waits for clients to make a request and then delivers a response. The response may include an HTML document, or any other content such as JavaScript modules, images, style sheets or more.

15.2. HTTP

Clients and servers communicate using the HTTP protocol.

An HTTP request includes:

- The HTTP request method or "verb" such as POST or GET.
- The URL being requested.
- Optional headers that may include authentication information.
- An optional request body.

An HTTP response includes:

- A status code such as 200 for 'OK' (success) or 404 for 'Not Found'.
- Some headers that allow the server to pass additional information about the response such as the content type.
- The response body: this may be a source HTML document.

15.3. TCP/IP Addresses and Port Numbers

HTTP relies on TCP/IP to send and receive sequences of bytes.

Computers connected to a TCP/IP network have an IP address such as 74.125.239.102 or 128.242.106.42.

We usually connect to servers using the Domain Name System (DNS). The DNS maps IP addresses to domain names such as google.com or foothill.edu.

The special name localhost refers to the local computer and corresponds to the special IP address 127.0.0.1.

We could have more than one application running on the same computer. Each application must use a different port. The ports are numbered from 1 to 65535. In our examples, we'll use port 8080 for our web server. To connect to a server running at a specific port, we append : followed by the port number to the IP address or host name. For example, we'll be using http://localhost:8080 to connect to our web server.

Note that when we don't specify a port, the default port 80 is used.

16. Server-Side JavaScript

16.1. Overview

JavaScript is a general purpose language and there are several frameworks that offer access to a JavaScript interpreter outside the browser. The most prevalent of these frameworks are Rhino and Node.

Rhino is free and available from Mozilla. It is implemented in Java. It allows us to write JavaScript code that manipulates Java objects and uses Java methods.

Node is a more recent solution. It is written in C. It supports an asynchronous, event-driven model that makes it highly scalable. It also contains a built-in HTTP server library that allows us to run a web server without using any external software such as Apache.

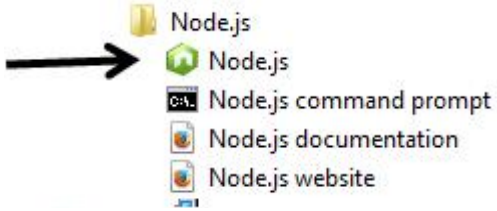
We'll take a closer look at Node in the following sections.

16.2. Getting Started with Node

Node is free and available from http://nodejs.org . Click on INSTALL to install it on your system.

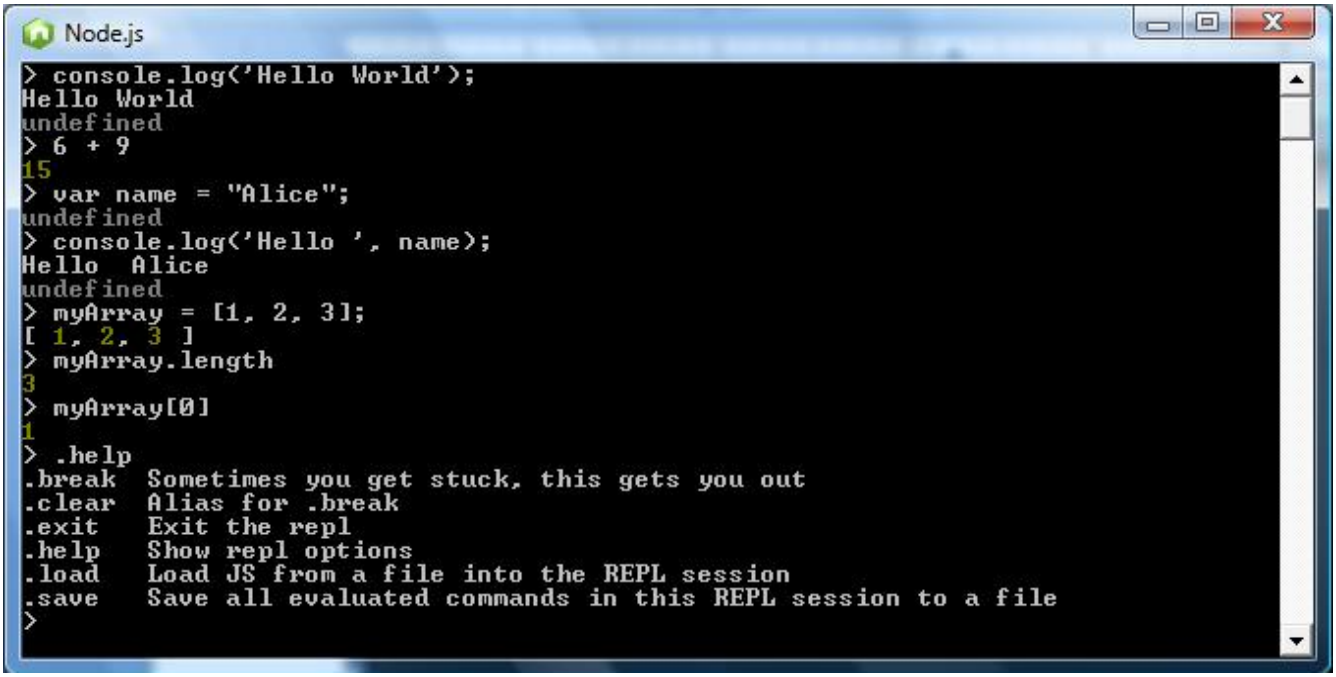
Once we have installed node, we have several options to run it.

Note that things will look slightly different on a MAC. Please post in the forums if you have any issues following along on a MAC.



The first option shown above, Node.js, gives us access to an interpreter shell where we can write JavaScript code. This shell is also known as the Read-Eval-Print-Loop (REPL).

We can type any valid JavaScript code and it is immediately executed.



The shell has also some special commands that start with a period (.). The most useful one is .help since it shows you the rest of the commands.

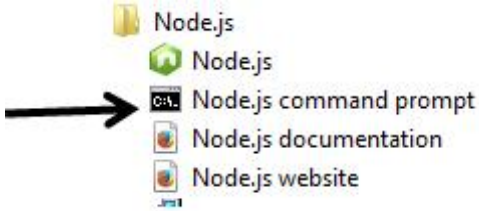
Another option is to run a JavaScript program from a file.

To do that we can use any editor to create and save the following one line file:

hellonode.js

console.log("Hello Node!");

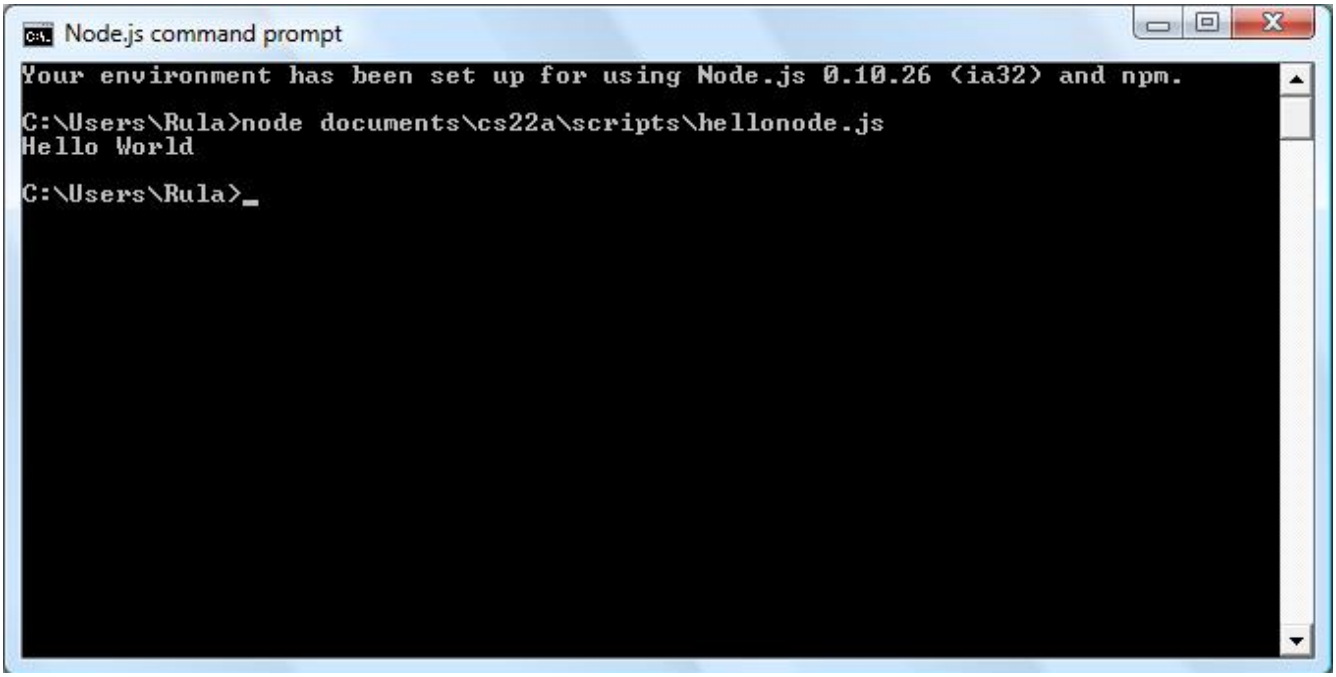
Then we can go to the Node Command Prompt...



... and invoke our program by typing node followed by the name of the file where we saved our code. In this case, it's hellonode.js:

node hellonode.js

Note that we may have to specify a path name if we have not saved the file in the current directory as shown below. Or we can change the current directory to the one where the file is saved.



16.3. Node Asynchronous Programming

Even though node is single-threaded, its event-driven, non-blocking approach makes it highly scalable. We'll illustrate this approach with a simple example that accesses the file system.

The file system module available in node is called fs. It gives us access to standard file operations.

To use this module, we need to 'load' it, or 'import' it. To do that in Node, we use the built-in function require(). require() reads and executes a JavaScript file and then returns its exported object. We can then access the different methods in that file through that object.

```
var fs = require('fs');
```

Now we have access to the fs module methods through our local variable fs.

All the methods in the fs module have asynchronous (non-blocking) and synchronous (blocking) forms.

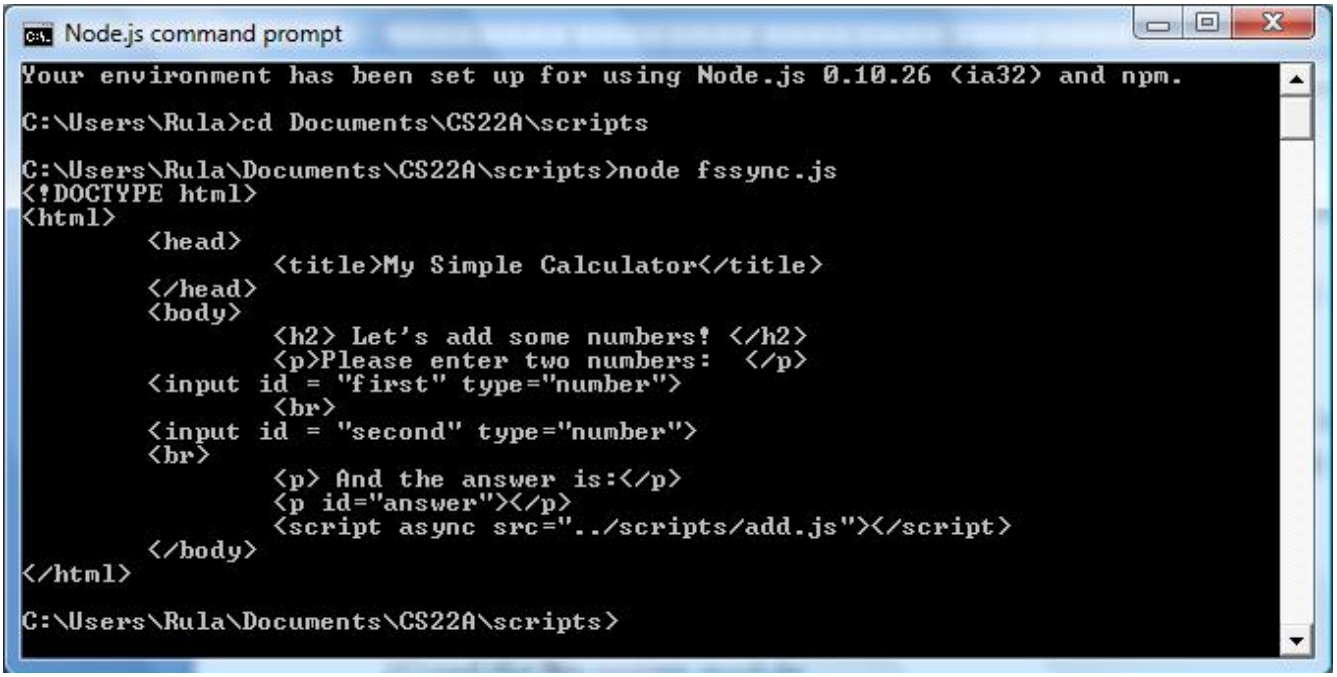
Let's first write a program that reads a local file, synchronously, and then displays its content. In this example we'll use the file calculator.html. We'll call the program fssync.js.

fssync.js

```
// Load the file system module
var fs = require("fs");
// Read the file synchronously
// Make sure the file path is correct for your system
```

```
var content = fs.readFileSync("../html/calculator.html", "utf8");
// Display the content
console.log(content);
```

After we create and save the file fssync.js, we can go the node command prompt and execute it by typing node fssync.js.



Now let's write a program that performs the same task asynchronously.

To do that we'll use the readFile() method. We'll also add some error handling.

```
// Load the file system module
var fs = require("fs");
// Read the file asynchronously and call the callback function when done
fs.readFile("../html/calculator.html", "utf8", function(error, content){
   if (error){
      console.log(error);
   } else {
      // If there is no error, display the output
      console.log(content);
   }
}
);
```

Note that the readFile() method takes one more argument than readFileSync(). It is the callback function. **Since the readFile() is asynchronous, there is no one waiting for its return value. Instead we provide a function that Node can invoke after readFile is complete.** That callback function then has access to the return value and can process it.

Another way to write the code above is:

```
//Callback Function to display the file content
function displayIt(error, content){
   if (error){
      console.log(error);
   } else {
      // If there is no error, display the output
      console.log(content);
   }
};
// Load the file system module
var fs = require("fs");
// Read the file asynchronously and call displayIt when done
fs.readFile("../html/calculator.html", "utf8", displayIt);
```

Note that readFile() supplies both the error and the content arguments to the callback function.

16.4. Our First Web Server

Now we are ready to write a program in Node that will run a very basic web server. This server will respond with the same web page for every request. We'll call the program firstserver.js.

To use the http library in Node, we first require it:

```
var http = require('http');
```

Now we can create a server by invoking http.createServer().

http.createServer takes an optional function as an argument. If present, that function is called whenever a request event occurs. That function is also passed a request and a response object arguments.

firstserver.js

```
//The function servePage will be called whenever the server
//receives a request
```

```
function servePage(request, response) {
   // 200 is the status code for success
   response.writeHead(200, {'Content-Type': 'text/html; charset = UTF-8'});
   // respond with a basic HTML web page
   response.write("<!DOCTYPE html>");
   response.write("<html>");
   response.write("<head>");
   response.write("<title>JavaScript for Programmers</title>");
   response.write("</head>");
   response.write("<body>");
   response.write("<h2>Server-side JavaScript</h2>");
   response.write("<h4>Node</h4>");
   response.write("<p>Node.js includes asynchronous libraries such as http and
fs.</p>");
   response.write("</body>");
   response.end("</html>");
};
// load the http module
var http = require('http');
// create a server object
var server = http.createServer(servePage);
// listen on port 8080
server.listen(8080, 'localhost');
// log an informational message
console.log('Server running at http://localhost:8080');
```

Let's take a closer look at our function servePage().

Note first that the request argument is ignored here since our server is serving the same page regardless of the request. **The response argument is used to return data back to the client.**

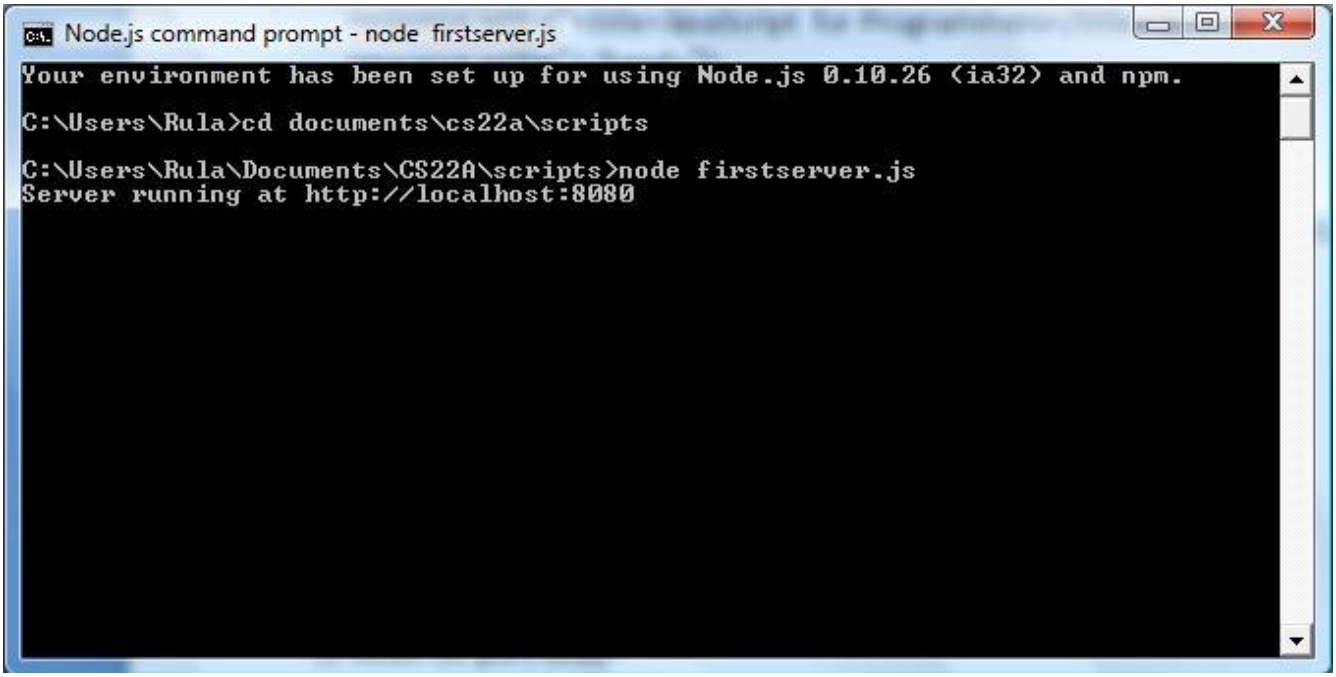
We first call the response.writeHead() method: we send the status code 200 for success and the Content-Type header.

Then we have several calls to response.write(). We simply write the HTML source document into the response body. These calls could have been all combined into one.

Finally we call response.end() with the last line in our file. And the response is

completed.

Once we create and save firstserver.js, we can run it in the Node command line by typing: node firstserver.js



The next step is to connect to the server using a web browser.

We can do that by typing http://localhost:8080 in the address bar.

The following web page should be displayed.

To stop your server, you can simply close the command line window or press ctrl C.

Now we can modify our server so that it serves the web page from a file instead.

It will still be a fixed web page, we'll call it nodedemo.html and it will contain the following:

nodedemo.html

<!DOCTYPE html>

<html>

```
<head>
<title>JavaScript for Programmers</title>
</head>
<body>
<h2>Server-side JavaScript</h2>
<h4>Node</h4>
<p>Node.js includes asynchronous libraries such as http and fs.</p>
</body>
</html>
```

We then modify our first server so that it opens and reads the content of nodedemo.html and then sends it in the response body. Let's call our modified server webserver.js.

webserver.js

```
//The following function will be called when the server
// is handling a request
function servePage(request, response) {
   // Read the file asynchronously
  // The filename is hardcoded here
   fs.readFile( '../html/nodedemo.html', function( err, content) {
      if (err) { // If there is an error, set the status code
         response.writeHead( 404,
                  {'Content-Type': 'text/plain; charset = UTF-8'});
         response.write( err.message); // Include the error message body
         response.end(); // Done
      } else { // Otherwise, the file was read successfully.
         response.writeHead( 200, // Set the status code
                  {'Content-Type': 'text/html; charset = UTF-8'});
         response.write(content); // Send file contents as response body
         response.end();
      }
   });
};
// Load the file system module
```

```
var fs = require("fs");
// load the http module
var http = require('http');
// create a server object
var server = http.createServer(servePage);
server.listen(8080, 'localhost');
console.log('Server running at http://localhost:8080');
```

Note that the way we have modified our server means that the HTML document will be read every time a request is received. This is overkill here since it is the same document, but it is good example to use as a model for this week's assignment where you'll have to send a different page with each request.

Once we save webserver.js, we can run it in the Node command line by typing:

node webserver.js.

Then we can connect to our server at http://localhost:8080 using a web browser.

The following web page should be displayed.

16.5. User Authentication

When a server implements user authentication, the user is prompted for a user name and password. The user is granted access to the website resources only if they provide a valid user name and password combination.

In the following example, we'll implement basic access authentication in Node. This is the simplest type of HTTP authentication.

We'll first need to install the http-auth module. Unlike the http and fs modules that come standard with Node, the http-auth module needs to be installed separately. We can do that by using the Node package manager (npm) from the Node command line as follows:

npm install http-auth

We'll also need to install htpasswd so that we can create and manage a password file.

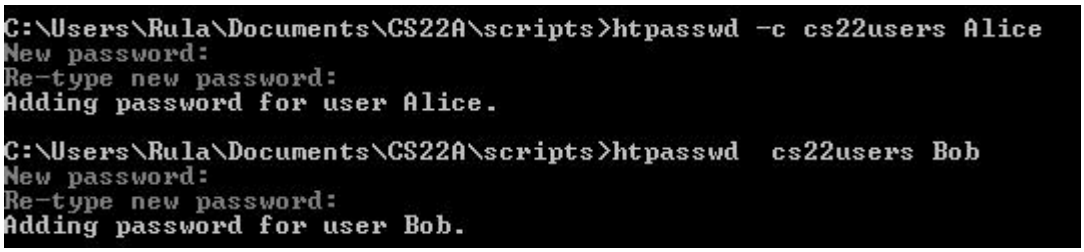
We can also do that from the command line:

npm install -g htpasswd

Once htpasswd is installed, we can invoke it from the command line to create a new password file and add users to it:

The –c option below allows us to create a new file cs22users and add Alice as a user in it. We are immediately prompted for a password for Alice.

htpasswd –c cs22users Alice



To add another user, we use htpasswd (without the –c option) on the same file.

htpasswd cs22users Bob

Again we are immediately prompted for a password for Bob.

If we open the cs22users file, we can see the user names with their encrypted passwords.

Alice:$apr1$oGaNKa2d$LU9WFj3sMihZd0owIjzqa1

Bob:$apr1$DyW5LzoE$Z4YQetiJDXc.B8D0JjiHI.

Now that we have these two users, we are ready to write a server that supports authentication.

Let's call our new server authserver.js.

authserver.js

```
// The function servePage will be called whenever the server
// receives an authenticated request.
// we can access the user name through the request object
// and use it to personalize our page.
function servePage(request, response) {
   response.writeHead(200, {'Content-Type': 'text/html; charset = UTF-8'});
   // respond with a personalized HTML web page
   response.write("<!DOCTYPE html>");
   response.write("<html>");
   response.write("<head>");
   response.write("<title>JavaScript for Programmers</title>");
```

```
   response.write("</head>");
   response.write("<body>");
   response.write("<h2>"+ request.user + " 's Lesson</h2>");
   response.write("<h4>Node</h4>");
   response.write("<p>Node.js includes a set of asynchronous libraries such as http and fs.</p>");
   response.write("</body>");
   response.end("</html>");
};
// load the http module
var http = require('http');
// load the http-auth module
var auth = require('http-auth');
// set the basic authentication options
var basic = auth.basic({
   realm: "CS22A area",
   // the following assumes that the password file is "cs22users"
   // and it is in the same directory as the current script
   file: "cs22users"
});
// create a server object with the basic authentication options
var server = http.createServer(basic, servePage);
// listen on port 8080
server.listen(8080, 'localhost');
// log an informational message
console.log('Server running at http://localhost:8080');
```

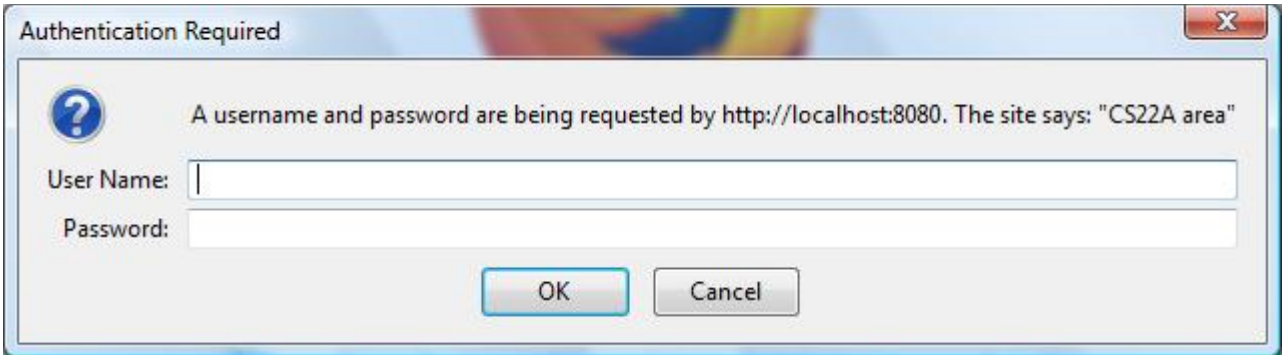
Note that the servePage() function has access to the user name through the request object: request.user.

Once we have created and saved authserver.js, we can run it by typing the following on the command line:

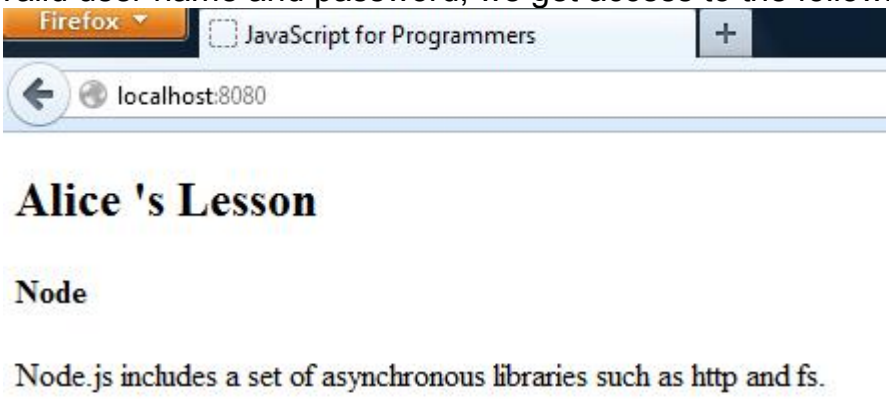
node authserver.js

Then we can go to the browser and type http://localhost:8080 in the address bar.

We get the following window prompting us for a user name and password:



If we type in a valid user name and password, we get access to the following web page:



17. Ajax

17.1. What is Ajax?

Ajax stands for Asnychronous JavasScript and XML.

It refers to a set of techniques for **exchanging data with the server without causing pages to reload.**

It allows web pages to be updated **asynchronously** by exchanging **small amounts of data with the server**. This of course results in a more responsive web application and a better user experience.

Google Maps is one of the best-known examples of Ajax applications.

AJAX is based on:

- JavaScript
- HTML and CSS
- The Document Object Model
- The XMLHttpRequest API to exchange data asynchronously with a server. Even though it is possible to implement Ajax by using < iframe > or < script > tags to communicate with the server, modern browsers all support the XMLHttpRequest API and the recommended way to implement Ajax is using the XMLHttpRequest object.
- A data exchange format: even though the x in Ajax stands for XML and XMLHttpRequest includes XML in its name, the use of XML is optional and becoming rare. The techniques support any text based format and JSON is currently used more

often.

17.2. Sending Requests to the Server

The XMLHttpRequest object is used to exchange data with the server.

We have seen last week (module 15) that clients and servers communicate using the HTTP protocol. They exchange HTTP requests and HTTP responses.

From the client side, to initiate the communication, **we first create an XMLHttpRequest object:**

var request = new XMLHttpRequest();

Then we specify the different parts of the HTTP request (method, url, headers, body) and send them. Let's see how to do that next.

The open() method:

We can use the open() method on our newly created request object to set the HTTP request method and url.

The general syntax for the open() method is as follows:

open (method, url, async, user, password)

The method parameter specifies **the HTTP request method or verb to use. The two most common methods are** GET and POST.

GET is used in most requests to request data from the server.

POST is typically used to submit data to the server.

We also use POST when we don't want a cached version of the server's response and when we are sending sensitive information.

The url parameter specifies **the url that is requested.** It points to a 'resource' on the server. This can be any kind of 'resource' such as a text file (txt or xml) or a program (asp or php).

The url specified here is usually a relative url (relative to the URL of the current document). If we specify an absolute url, it must include the same protocol, hostname, and port number as the current document. Otherwise it will usually result in an error, because it violates the same-origin policy.

Note that the url may also include a query string that allows the client to pass additional parameters to the server.

async is an optional parameter that specifies whether or not the request is sent asynchronously. If we do not specify it, it defaults to true.

The user and password parameters are optional and may be used for authentication purposes.

So for a GET request, to get the "data.txt" file from the server, we write:

var request = new XMLHttpRequest();

request.open(“GET”, “data.txt”);

To send the request to the server, we use the send() method. The send method may include the request body, if present.

GET requests don’t include a body, so for the GET request above, we simply write:

request.send();

So putting it all together, to send a GET request for the file “data.txt”, we have:

```
function sendRequest(){
   var request = new XMLHttpRequest();
   request.open("GET", "data.txt");
   request.send();
}
```

Let’s take a look at how a POST request would be implemented. Let’s assume that we would like to send some JSON encoded data to the server.

We first create the XMLHttpRequest object:

var request = new XMLHttpRequest();

Then we set the method and the resource. Note that the resource name, someurl, will depend on the specific setup of your server.

request.open(“POST”, “someurl”);

We also have to set the request header. We can do that with the setRequestHeader method. We are basically telling the server that the request body contains JSON encoded data.

request.setRequestHeader("Content-Type", "application/json");

And finally we send our request body to the server with send. The variable data contains the data that we need to send to the server:

request.send(JSON.stringify(data));

So putting it all together, to send a POST request for some JavaScript object data, we have:

```
function postRequest(){
   var request = new XMLHttpRequest();
   request.open("POST", "someurl");  // someurl depends on your server
   request.setRequestHeader("Content-Type", "application/json");
   request.send(JSON.stringify(data));
}
```

17.3. Handling Asynchronous Responses

When we send our request asynchronously, our program does not block until the server responds: that's a good thing, it can go ahead and perform other tasks and it can be more responsive to the user.

However the program needs to be somehow notified when the request is complete so that it can retrieve it.

To do that we can listen for 'readystatechange' events on the XMLHttpRequest object.

The 'readystatechange' event is based on the readyState property of the XMLHttpRequest object. It is triggered when the readyState property changes.

The readyState property describes the status of the request. It has one of the following values:

0: UNSENT - open() has not been called yet

1: OPENED

2: HEADERS_RECEIVED

3: LOADING – loading the response

4: DONE

To listen for readystatechange events, we set the onreadystatechange property of the XMLHttpRequest object to our event handler function.

We could also use addEventListener() but the onreadystatechange property is commonly used here.

To set the onreadystatechange property using an anonymous function definition:

```
request.onreadystatechange = function(){
 if (request.readyState === 4 && request.status === 200){
   // add code here to retrieve the response

  }
 };
```

Note that because **the readystatechange event is triggered several times before the response is complete, we first need to check that the request is complete when handling this event:**

if (request.readyState===4...

And because the request may or may not succeed, we must also check the HTTP status code. As we have seen before, 200 means OK.

So we have:

```
if (request.readyState===4 && request.status===200){
 ...
```

We 'll see next how to retrieve the response.

17.4. Retrieving the Responses

A complete HTTP response includes a status code, response headers, and a response body. We have access to all of these through the properties and methods of the XMLHttpRequest object.

We've already seen how to check the status of the response with request.status.

request.statustext is also available: it is a string with values such as "OK" or "Not Found".

We can use getResponseHeader() to access the response headers. To get the "Content-Type" information from the response header, we write:

var type = request.getResponseHeader("Content-Type");

We can use the responseText property to access the response body as plain text.

The responseXML property allows us to access the response body when it is an XML or XHTML document.

To display the response body on our web page in an element with id response, we can write:

document.getElementById("response").textContent = request.responseText;

17.5. Simple Demo

To illustrate how the various steps involved in communicating with the server come together, we'll use a basic example.

Note that in order to reuse the server you wrote in assignment # 8, we'll put the HTML document, the client-side script and the server-side script in the same folder.

Let's start with the HTML source file ajaxdemo.html.

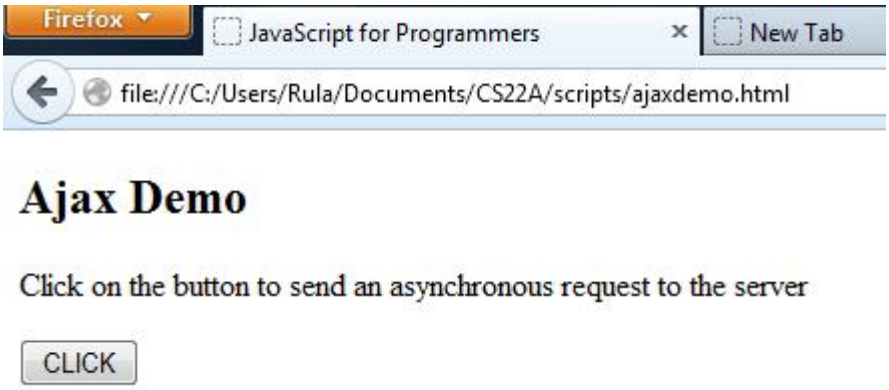
ajaxdemo.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript for Programmers</title>
</head>
<body>
<h2> Ajax Demo</h2>
<p>Click on the button to send an asynchronous request to the server </p>
<input id = "request" type="button" value="CLICK">
<p id="response"></p>
<script defer src="ajaxdemo.js"></script>
```

</body>

</html>

It's a very simple web page. You can open it and display it in your browser to see what it looks like but the Ajax features will not be active: the XMLHttpRequest API does not work with the FILE: scheme so we'll need to setup a server.



The goal is to have our client-side script ajaxdemo.js respond to a user click by sending an asynchronous request to the server and then displaying the additional data received in the <p> element with id = "response".

Let's write our client-side script ajaxdemo.js to handle that:

ajaxdemo.js

```
function sendRequest(){
   // create an XMLHttpRequest object
   var request = new XMLHttpRequest();
   // Specify a GET request for the file data.html
   request.open("GET", "data.html");
   request.send();
   // define the function to be called when the response is received.
   request.onreadystatechange = function () {
     // check that the response is complete and the request was successful
     if (request.readyState === 4 && request.status === 200){
       // Display the response
       document.getElementById("response").innerHTML = request.responseText;
     }
   };
};
```

```
// Add an event listener to call sendRequest when the button is clicked
document.getElementById("request").addEventListener("click", sendRequest);
```

The next step is to setup a server to serve ajaxdemo.html and the corresponding ajaxdemo.js. We also need to create the data.html file that will be requested by the client.

For our demonstration purposes, we'll create it with just one line:

data.html

```
<span>This is the additional data requested from the server.</span>
```

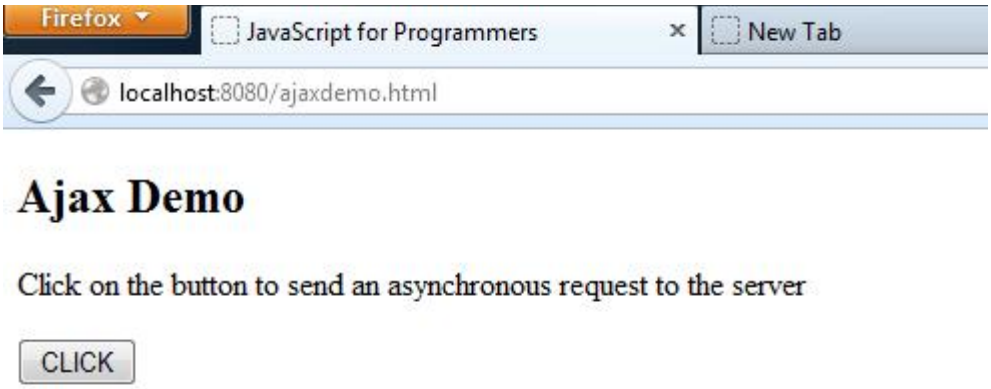
We'll use the server you wrote in assignment # 8, beswebserver.js.

From the Node command line, start the server by typing:

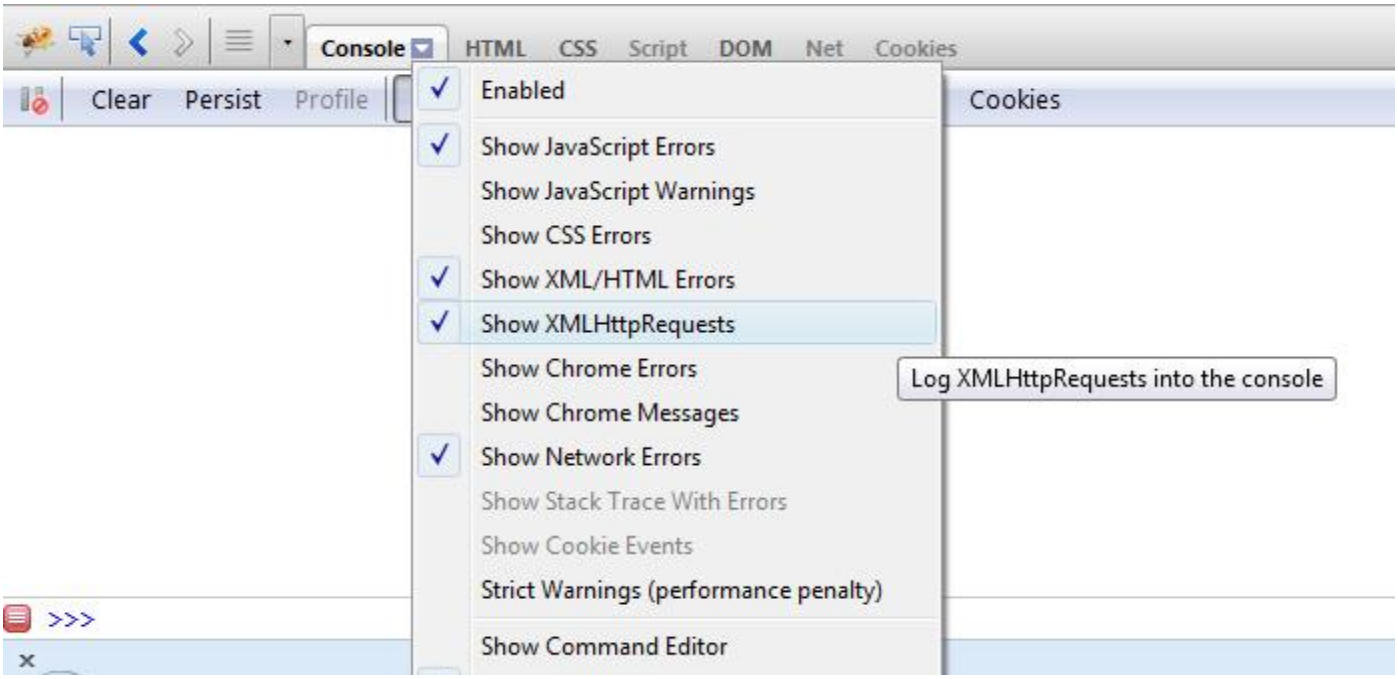
```
node bestwebserver.js
```

Then start Firefox and type http://localhost:8080/ajaxdemo.html in the address bar.

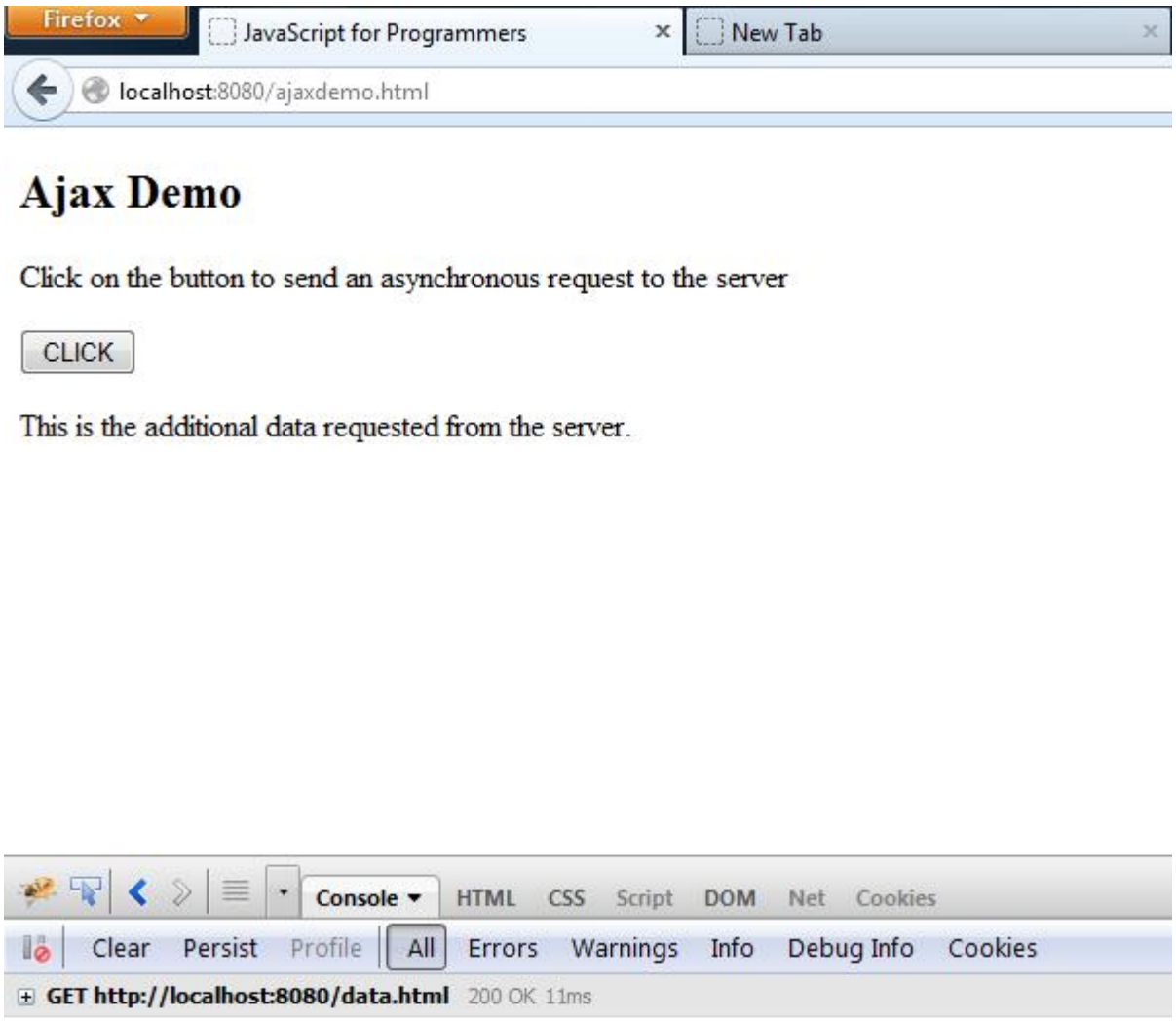
The following should be displayed:



Open the Firebug Console and make sure you select Show XMLHttpRequests from the console drop down menu as shown below:

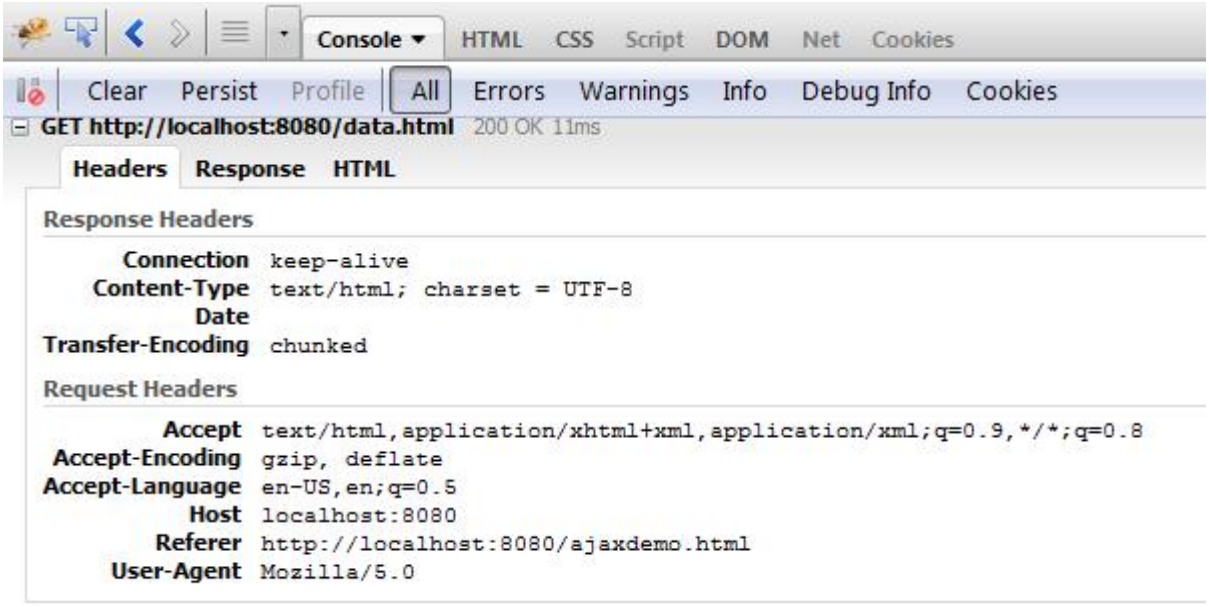


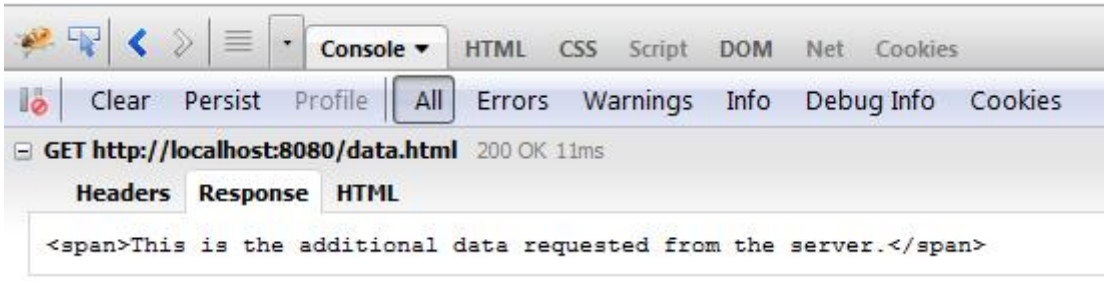
Then go to the Ajax Demo web page and click on the button. You get the following:



Note that the additional text displayed under the button came from the server's response.

Note also that you get details about the request and response in the Firebug console window. Click on the + sign next to the GET to see details about the request headers and the response.





17.6. Ajax with jQuery

jQuery includes some methods and functions that make Ajax simpler to implement.

To follow along, modify the html file used in the previous demo ajaxdemo.html to include jQuery:

ajaxdemo.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript for Programmers</title>
</head>
<body>
<h2> Ajax Demo</h2>
<p>Click on the button to send an asynchronous request to the server </p>
<input id = "request" type="button" value="CLICK">
```

```
<p id="response"></p>
<script defer src="jquery-1.11.0.js"></script>
<script defer src="ajaxdemo.js"></script>
</body>
</html>
```

The **load() method may be used to load a given url into each of the elements specified by a jQuery selector.**

In our previous demo, we could have rewritten our client side script ajaxdemo.js as follows:

ajaxdemo.js

```
function sendRequest(){
   // Load additional data from the server into the response element
   $("#response").load("data.html");
};
// Add an event listener to call sendRequest when the button is clicked
$("#request").on("click", sendRequest);
```

We can even select parts of the loaded document by specifying a jQuery selector following the url.

Let's modify our data.html file to include the following two lines with ids line1 and line2.

data.html

```
<span id = "line1" >This is the first line of the additional data requested from the server.</span>
<span id = "line2" >This is the second line of the additional data requested from the server.</span>
```

Then to load only line 2 in the response element of ajaxdemo.html, we can modify ajaxdemo.js as follows:

```
function sendRequest(){
   // Load additional data from the server into the response element
   $("#response").load("data.html #line2");
};
// Add an event listener to call sendRequest when the button is clicked
$("#request").on("click", sendRequest);
```

The next step is to run the server from the node command window:

node bestwebserver.js

Then enter http://localhost:8080/ajaxdemo.html in the address bar in Firefox.

The initial Ajax demo web page is displayed and after we click the button we get:

jQuery provides two additional functions jQuery.get() and jQuery.post() that allow us to send asynchronous GET and POST requests to the server without having to deal with the details of the XMLHttpRequest object.

Finally, the jQuery.ajax() function allows us to specify a more generic request with more control over the various request options.

17.7. Limitations

There are a few issues surrounding Ajax applications that we need to be aware of:

Security:

Care must be taken not to transfer sensitive data as plain text in Ajax calls.

Some consider that the increased communication between the client and the server may make the application more vulnerable to security threats.

Search Engine Indexing:

The content of AJAX applications will not usually appear in search engine results. Web crawlers do not crawl JavaScript code, so unless we use some other technique to make our content available to search engines, it will not appear in the search engine results.

Back Button and Bookmarking:

History management and bookmarking are more challenging to implement in Ajax applications.

18. Some More HTML5 APIs

18.1. Web Storage Overview

The HTML5 Web Storage API allows web applications to store **data locally on the user's device**. For instance a web app may store certain user preferences, or

progress through a lesson plan, or scores in a game.

This kind of storage is **specific to a single origin**.

The origin is defined as the combination of scheme, such as http, domain, such as foothill.edu, and port number such as 8080 for our own node server.

A script associated with one origin cannot read data stored by a script from a different origin.

For example the web pages
http://foothill.edu/counseling/ and http://foothill.edu/aid/ can share the data stored in web storage because they have the same origin: they have the same scheme(http), the same domain (foothill.edu) and the same port number (default for both).

However https://myetudes.org/portal and http://foothill.edu/counseling/ do not have the same origin. They have different schemes (http vs https) and different domains (myetudes.org vs foothill.edu). As a result they each have their own web storage and they can't read the data stored by the other.

It is also important to note that **the data stored through the web storage API is browser specific**. For example when we access a web app with Chrome, and then access it again with FireFox, any data stored through the web storage API during the first visit is not accessible during the second visit.

The Web Storage API is easy to use. It is more secure and faster than using cookies. It allows us to store large amounts of data without a negative impact on performance.

The web storage API includes the localStorage and sessionStorage objects.

Before we go into the details of each of these objects, it is important to note that the data here is stored as plain text, **unencrypted**. **As a result, this should never be used to store sensitive information** such as credit card numbers or passwords. The data is easily accessible to anyone who has access to that device (and that can be physical access or software access through spyware).

18.2. The localStorage Object

We have access to the localStorage object through the property of the window object: window.localStorage (or simply localStorage). It is technically a 'Storage object' but it works the same way as a regular JavaScript object.

The data is stored in name:value pairs, just like a regular JavaScript object.

Let's illustrate the use of localStorage object with a simple example.

Consider the following modified HTML file for our word guessing name where we invite the player to enter their name. We'll call the modified HTML file guesssave.html.

guessave.html

```
<!DOCTYPE html>

<html>

<head>
```

```
<meta charset="utf-8">
<title>Guessing Game</title>
<link rel = "stylesheet" type = "text/css" href = "guess.css" media = "all">
</head>
<body>
<p>Who is playing?</p>
<input id = "playername" type="text">
<p>Guess a Letter</p>
<p id = "display" class = "letters"></p>
<input id = "guess" type="text" maxlength='1' class = "letters" autofocus>
<div>
<input id = "guessbutton" type="button" value="GUESS">
</div>
<p>Wrong Letters</p>
<p id = "wrong" class = "letters wrong"> </p>
<progress id = "indicator" value="0" max="10">
</progress>
<div>
<input id = "restart" type="button" value="RESTART">
</div>
<script defer src="../scripts/guess.js"></script>
</body>
</html>
```

Now let's open the HTML source document in FireFox and enable the Firebug console.

Enter a name under the prompt: Who's playing?

Then enter the following at the Console prompt

>>>var player = document.getElementById("playername").value;

The variable player now contains the string “Alice” or any other name you entered.

To save the player’s name in the localStorage object, we can simply write:

>>>localStorage["lastplayer"] = player;

or:

>>>localStorage.lastplayer = player;

We are basically creating a “lastplayer” property of the localStorage object.

We can also use the setItem() property of the localStorage object to get the same result as above:

>>>localStorage.setItem(“lastplayer”, player);

We can now query the localStorage object by typing the following at the console prompt:

>>> localStorage

1 item in Storage lastplayer=”Alice”

We can use any of the following:

>>> localStorage.getItem(“lastplayer”); // use the getItem() method of the localStorage object

“Alice”

>>> localStorage[“lastplayer”];

“Alice”

>>> localStorage.lastplayer;

“Alice”

All values in localStorage are currently stored as strings. When we store a number, it is implicitly converted to a string.

Let’s say for example that we want to store the player’s score.

At the Firebug console prompt, enter:

>>> localStorage.setItem("score", 1);

Now let’s query the localStorage object:

>>> localStorage;

2 items in Storage lastplayer="Alice", score="1"

>>> localStorage.score;

"1"

Note that 1 is stored as the string “1”.

If for example we need to retrieve the score and update it, we must convert to a number first (otherwise we get a string concatenation instead of an addition).

localStorage["score"] = parseFloat(localStorage["score"]) + 5;

>>> localStorage["score"];

"6"

The following will not work. It performs string concatenation

>>> localStorage["score"] = localStorage["score"] + 5;

"15"

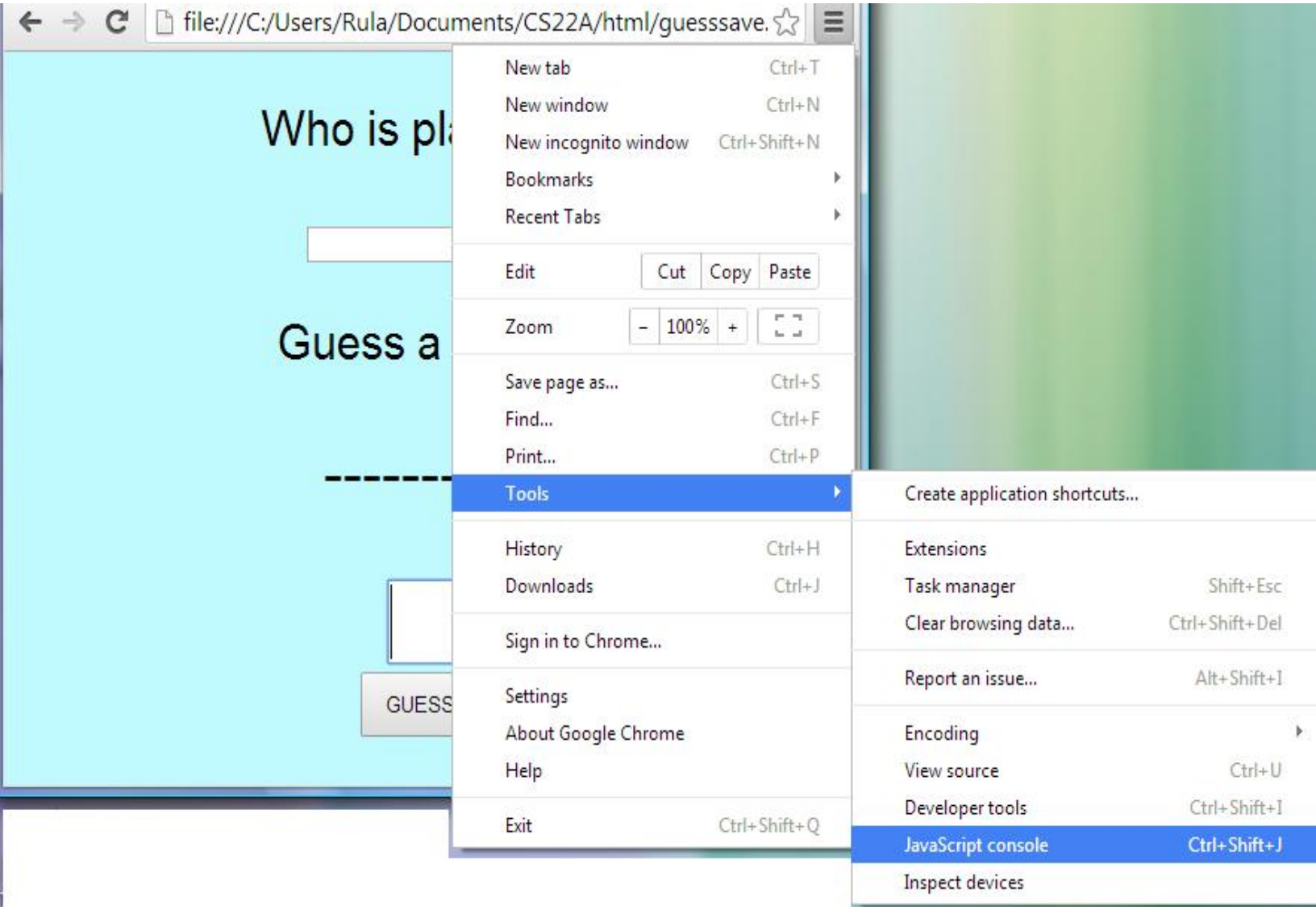
The data stored in the localStorage object is permanent. It has no expiration date and it does not disappear when the browser is closed.

We can close our browser completely, then reopen it and reload our page and then query the localStorage again and the information we saved is still there.

>>> localStorage

2 items in Storage lastplayer="Alice", score="6"

However if we open the same page in Chrome, we don't have access to the same localStorage. To see that, open the same quesssave.html page in Chrome. To access the console in Chrome you can select the JavaScript console from the Tools menu.



Querying the localStorage object in Chrome shows an empty object because **each storage object is specific to the browser.**

We said that the data stored in the localStorage object is permanent. This does not mean that we can never delete it if we choose to.

The removeItem() method will remove a given item from the localStorage object.

Let's go back to our web page in FireFox and type:

>>> localStorage.removeItem("lastplayer");

Then query our localStorage object:

>>> localStorage

1 item in Storage score="6"

We can use the clear() method to remove all items in the localStorage object.

>>> localStorage.clear();

>>> localStorage

0 items in Storage

18.3. The sessionStorage Object

The sessionStorage object is similar to the localStorage object. We can access it through the property of the window object: window.sessionStorage (or simply sessionStorage).

We can manipulate it, adding items, accessing items and removing items just like the localStorage object.

The difference is in lifetime and scope.

Let's consider scope first. The sessionStorage object is specific not just to the browser (Chrome vs FireFox) but to a specific tab within the browser.

To illustrate this, let's first run the same scenario as in the previous section but this time save our data in sessionStorage.

We first open our source HTML document guesssave.html in Firefox, enter a player name and then go to the Firebug console.

We enter the following at the Console prompt

>>>var player = document.getElementById("playername").value;

The variable player now contains the string "Alice" or any other name you entered.

To save the player's name in the **sessionStorage** object, we can simply write:

>>>sessionStorage["lastplayer"] = player;

Or:

>>>sessionStorage.setItem("lastplayer", player);

We can now query the sessionStorage object by typing the following at the console prompt:

>>> sessionStorage

1 item in Storage lastplayer="Alice"

Or:

>>> sessionStorage.getItem("lastplayer") // use the getItem() method of the session Storage object

"Alice"

All values in sessionStorage are also stored as strings. When we store a number, it is implicitly converted to a string.

>>> sessionStorage.setItem("score", 1);

>>> sessionStorage.score;

"1"

In order to retrieve the score and update it, we must convert it to a number first.

sessionStorage["score"] = parseFloat(sessionStorage["score"]) + 5;

>>> sessionStorage["score"];

"6"

Now let's open a new tab in Firefox and in that tab open the same web page guesssave.html.

At the console prompt in that second tab, type:

>>> sessionStorage

0 items in Storage

The sessionStorage that we defined in the first tab is only accessible from that tab.

Now let's close both tabs and then reopen guesssave.html in FireFox.

>>> sessionStorage

0 items in Storage

The sessionStorage object is cleared when the corresponding window is closed.

18.4. The Application Cache

In general, caching refers to the practice of storing recently accessed data so that it can be easily and quickly accessed later.

Browsers usually cache web pages on the user's device. For example, when we visit a web page, the browser may cache the source HTML document, the images, stylesheets and JavaScript files used. When at a later time we go back to revisit that same page (say with the back button), the browser can get a copy of all the files needed from the local device. Browsers have some rules to determine if the copy on the local device is 'fresh'.

This kind of caching results in a faster user response and a reduced load on the server and network.

The Application Cache, introduced in HTML5 deals with a slightly different aspect of caching. It allows us to have **web applications available offline,** when the device does not have internet access.

The application cache is different from the browser's cache. While the browser's cache has a size limit and keeps the most recently used data only, the application cache is permanent. Applications stored there remain there until they are explicitly deleted.

To be available offline, web applications have to install themselves in the application cache, use localStorage to store any associated data, and then synchronize with the server when they detect that the device is online again.

Let's go back to our simple canvas drawing example from module 10.3. Our goal is to make that simple app available when we are offline. To emulate offline, we'll initially serve our app using our node server server.js, then stop the server and access the app. We'll need to access it through our localhost:8080 url NOT through the file

scheme.

Here we modify the original draw.html slightly so that it can work with our node server. Let's name this document drawapp.html.

drawapp.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Let's Draw</title>
</head>
<body>
<h2> Just click inside the box </h2>
<canvas id="myCanvas" width="300" height="300" style="border:1px solid #c3c3c3;">
</canvas>
<script async src="draw.js"></script>
</body>
</html>
```

The corresponding JavaScript file draw.js contains the following:

draw.js

```
function drawSquare(event) {
    // Our function takes the event object as a parameter
    var myCanvas=document.getElementById("myCanvas");
    // obtain the coordinates with respect to the canvas
    var x = event.clientX - myCanvas.offsetLeft;
    var y = event.clientY - myCanvas.offsetTop;
    // to access the canvas, we need its context
    var myContext = myCanvas.getContext("2d");
    // set the color to red
    myContext.fillStyle="#FF0000";
    // draw a 10 by 10 square starting at the click event position
    myContext.fillRect(x,y,10,10);
};
document.getElementById("myCanvas").addEventListener("click", drawSquare, false);
```

To enable application cache, we must first **create a manifest file**. A manifest file is a text file that includes the resources that our application needs to run. It tells the browser what to cache and what to never cache.

The manifest file has three sections:

CACHE MANIFEST - Files listed under this header will be cached after they are downloaded for the first time. This is the section that we need here. **In order for an application to be available offline, its associated html source, JavaScript files, style sheets, any images or other resources needed must be listed in this section**.

NETWORK - Files listed under this header require a connection to the server, and will never be cached.

FALLBACK - Files listed under this header specifies fallback pages if a page is inaccessible.

Our focus here is the CACHE MANIFEST section. Let's create our manifest file.

In our case, our drawing application only needs the following:

drawapp.html

draw.js

So we create our manifest file drawapp.appcache as shown below. **The recommended file extension for a manifest file is "appcache"**.

drawapp.**appcache**

```
CACHE MANIFEST
drawapp.html
draw.js
```

The next step is to associate our manifest with the application. To do that we include the manifest attribute in the main source document's <html> tag:

drawapp.html

```
<!DOCTYPE html>
<html manifest="drawapp.appcache">
<head>
<meta charset="utf-8">
<title>Let's Draw</title>
</head>
<body>
<h2> Just click inside the box </h2>
<canvas id="myCanvas" width="300" height="300" style="border:1px solid #c3c3c3;">
</canvas>
<script async src="draw.js"></script>
</body>
</html>
```

There is one more detail about the manifest file that we need to address: it must

be served with content type “text/cache-manifest”.

We will use a modified version of our node server bestwebserver.js. This version checks the extension of the requested resource and serves it with the corresponding content type header. This way our manifest file will be served with the correct type: “text/cache-manifest”. We’ll call this version server.js.

server.js

```
// Simple node server - CS22A
// The following function will be called whenever
// the server receives a request.
function servePage(request, response) {
    // MIME types supported by this server
    var supportedTypes = {
        'html': 'text/html; charset = UTF-8',
        'txt': 'text/plain; charset = UTF-8',
        'js': 'application/javascript; charset = UTF-8',
        'appcache': 'text/cache-manifest; charset = UTF-8'
    }
    // Extract the filename and extension from the request.
    var filename = url.parse(request.url).pathname.substring(1);

    // If the user does not enter a file name,
    // we serve the page home.html - not required.
    if (!filename) {
        filename = 'home.html';
    }
    // Get the extension of the requested resource so we can determine the type
    var extension = filename.substring( filename.lastIndexOf(".") + 1)
    var type = supportedTypes[extension];  // type implied by extension

    // Read the file asynchronously
    fs.readFile( filename, function( err, content) {
        if (err) { // If there is an error, set the status code
            response.writeHead( 404,
                        {'Content-Type': 'text/plain; charset = UTF-8'});
            response.write( err.message); // Include the error message body
```

```
            response.write( ' - The page requested is not found.');
            response.end(); // Done
        } else { // Otherwise, the file was read successfully.
            response.writeHead( 200, // Set the status code
                        {'Content-Type': type});
            response.write(content); // Send file contents as response body
            response.end();
        }
    });
};
// load the url module
var url = require('url');
// Load the file system module
var fs = require("fs");
// load the http module
var http = require('http');
// create a server object
var server = http.createServer(servePage);
server.listen(8080, 'localhost');
console.log('Server running at http://localhost:8080');
```

We can now start our server from the node command prompt by typing:

node server.js

Then go to Firefox and write the following in the address bar:

http://localhost:8080/drawapp.html

Our simple drawing application page is displayed.

Now let's go back to the node command prompt window and stop the server by closing the window or pressing control C.

Then we go back to our Firefox window, close it completely then reopen it.

Now try to access **calculator.html** by typing the following in the address bar:

http://localhost:8080/calculator.html

We get an error message 'Unable to connect' because the server is no longer running.

However if we try to access our drawing application by typing the following:

http://localhost:8080/drawapp.html

We can see that the drawing application is loaded successfully from the application

cache even though our server is not running (or if we are **offline**).

18.5. History Management

We have access to the browsing history associated with a given window through the history property window.history or simply history. This property refers to an object that includes entries for the urls 'visited'.

We can simply type history at the Firebug prompt:

>>> history

6 history entries

Note that we don't have access to the stored entries, just their number. This is to make sure that a script running on one web page does not have access to a user's browsing history.

There are methods defined on the history object that allow us to navigate back and forward and these methods have the same effect as pressing the browser Back and Forward buttons.

We can try the following from the Firebug console:

>>>history.back() // this loads the previous web page

>>>history.forward() // this takes us forward one page

>>>history.go(-3) // this takes us 3 pages back

>>> history.go(2) // this takes us forward 2 pages

The issue we face is that today, **web applications dynamically change the web page content without loading a new web page**. We still want to allow the user to use the Back and Forward buttons to navigate between these dynamically created application states.

HTML5 introduced an API that allows us to implement seamless history management in a dynamic application. Let's see how to do that with a simple web page that is modified based on user input (without a page reload). For the sake of simplicity, we will not include any Ajax communication with the server in this example. The same approach can be used when we have Ajax communication with the server.

We'll call our source document historydemo.html.

historydemo.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript for Programmers</title>
</head>
<body>
<h2>History Management with Dynamic Pages</h2>
```

```
<p>Click on one of the buttons to change the page dynamically. </p>
<input id = "first" type="button" value="FIRST">
<input id = "second" type="button" value="SECOND">
<p id="response"></p>
<script defer src="../scripts/historydemo.js"></script>
</body>
</html>
```

Let's write a first version of our script historydemo.js without taking into account any history management:

historydemo.js

```
// This function takes a parameter, line, and writes the corresponding
// text to the element with id: response.
function displayIt (line) {
        // store the text corresponding to the two options in an object
        // with properties first and second.
        var lineContent = {first: "First Line", second: "Second line"};
        // display the corresponding line in the html element with id response
        document.getElementById("response").textContent = lineContent[line];
}
// this function handles click events on both buttons.
function update(event){
        // determine which button was clicked from the event.target
        displayIt(event.target.id);
};

// Add event listeners to call update() when a button is clicked
document.getElementById("first").addEventListener("click", update);
document.getElementById("second").addEventListener("click", update);
```

Once we have saved historydemo.html and historydemo.js, we can open historydemo.html with Firefox.

Let's press on the First and Second buttons a few times to change the web change.

What happens when we use the browser's back button <- to go back to the previous page? We go back to whatever initial page we had before we loaded historydemo.html. We can't navigate between the dynamic states of historydemo.html.

We can modify our JavaScript code to use the **pushState()** method and the **popstate** event as follows:

The pushState() method allows us to **add an entry to the history object.** The syntax is:

history.pushState(stateObject, title, URL)

stateObject is an object that contains all the information needed to restore the current state of the document. Any object that can be converted to a string with JSON.stringify() will work here. When the user navigates back or forth from one history entry to the other, a copy of the stateObject associated with the current history entry becomes available through **history.state**.

The title parameter is currently ignored by major browsers. It refers to a title to be associated with that state.

The URL parameter is optional. It allows us to associate **each state with a unique url**. This URL will appear in the browser's location bar. If an absolute url is specified, it must have the same origin as the current url (the url of the corresponding static web page).

Going back to our historydemo.html web page, there are two dynamic states that we need to associate with that page:

- The state right after the user has clicked the FIRST button.
- The state right after the user has clicked the SECOND button.

We modify our update() function in the historydemo.js file to push a state object corresponding to each of these two states:

```
function update(event){
        var stateObject;  // our state object will simply contain a string
        displayIt(event.target.id);
        // determine which button was clicked from the event.target
        // and push the corresponding state object.
        if (event.target.id === "first") {
                // identify the state corresponding to FIRST button
                stateObject = "first";
                history.pushState(stateObject,"First Line", "#first");
        } else {
                // identify the state corresponding to SECOND button
                stateObject = "second";
                history.pushState(stateObject,"Second Line", "#second");
        }
};
```

Note that we just picked a string here to identify our state ("first" or "second"). We could have picked 1 or 2 or a more complex object…

The next step is for our code **to generate the correct web page when the user navigates back or forth to a given state**.

The popState event is triggered on the window object when the user navigates back or forth. We can add an event listener for the popState event as follows:

window.addEventListener("popstate", generateState);

Before we define the generateState function, remember that when the user navigates back or forth from one history entry to the other, a copy of the stateObject associated with the current history entry becomes available through **history.state**.

```
// Generate the web page corresponding to the state in history.state
// If no state is available, restore the original web page
function generateState(){
        if (history.state) {
             displayIt(history.state);
        } else {
              document.getElementById("response").textContent = "";
        }
};
```

There is one last special case we need to cover. When a page (with dynamic states) reloads, it may have a non-null state object without a popstate event being triggered. To cover this special case, we add the following to our JavaScript code:

```
if (history.state) {
        generateState();
}
```

Putting it all together, we now have:

historydemo.js

```
// This function takes a parameter, line, and writes the corresponding
// text to the element with id: response.
function displayIt(line){
        // store the text corresponding to the two options in an object
        // with properties first and second.
        var lineContent = {first: "First Line", second: "Second line"};
        // display the corresponding line in the html element with id response
        document.getElementById("response").textContent = lineContent[line];
};

function update(event){
        var stateObject;  // our state object will simply contain a string
        displayIt(event.target.id);
```

```
        // determine which button was clicked from the event.target
        // and push the corresponding state object
        if (event.target.id === "first") {
                // identify the state corresponding to FIRST button
                stateObject = "first";
                history.pushState(stateObject,"First Line", "#first");
        } else {
                // identify the state corresponding to SECOND button
                stateObject = "second";
                history.pushState(stateObject,"Second Line", "#second");
        }
};

// Generate the web page corresponding to the state in history.state.
// If no state is available, restore the original web page.
function generateState(){
        if (history.state) {
              displayIt(history.state);
        } else {
                document.getElementById("response").textContent = "";
        }
};

// Add event listeners to call update() when a button is clicked
document.getElementById("first").addEventListener("click", update);
document.getElementById("second").addEventListener("click", update);
// Add even listener to handle popstate event on the window
window.addEventListener("popstate", generateState);
// Handle the special case of reload
if (history.state) {
        generateState();
}
```

To see how it works, we can now load historydemo.html in our browser, click on FIRST and SECONDS a few times to generate different dynamic pages then navigate with the browser back and forward buttons (<- and ->).

As you do that note that the correct dynamic page is loaded and that the corresponding url is displayed in the address bar:

file:///somepath/historydemo.html#first

file:///somepath/historydemo.html#second

where somepath depends on your configuration.

18.6. Geolocation

The HTML5 Geolocation API gives us access to the geographical position of a user with their permission. This allows us to generate directions, maps, and local information such as weather or local shopping and dining options.

The geographical position is available through the property: navigator.geolocation.

Here is a very simple example of how to use it. We start with the source document geodemo.html.

geodemo.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JavaScript for Programmers</title>
</head>
<body>
<h2>Where in the world are we?</h2>
<p>Click on the button to find out. </p>
<input id = "findme" type="button" value="FINDME">
<p>Latitude:  <span id="latitude"></span></p>
<p>Longitude: <span id="longitude"></span></p>
<script defer src="../scripts/geodemo.js"></script>
</body>
</html>
```

We would like to display the latitude and longitude of the user's position when the user clicks on the FINDME button.

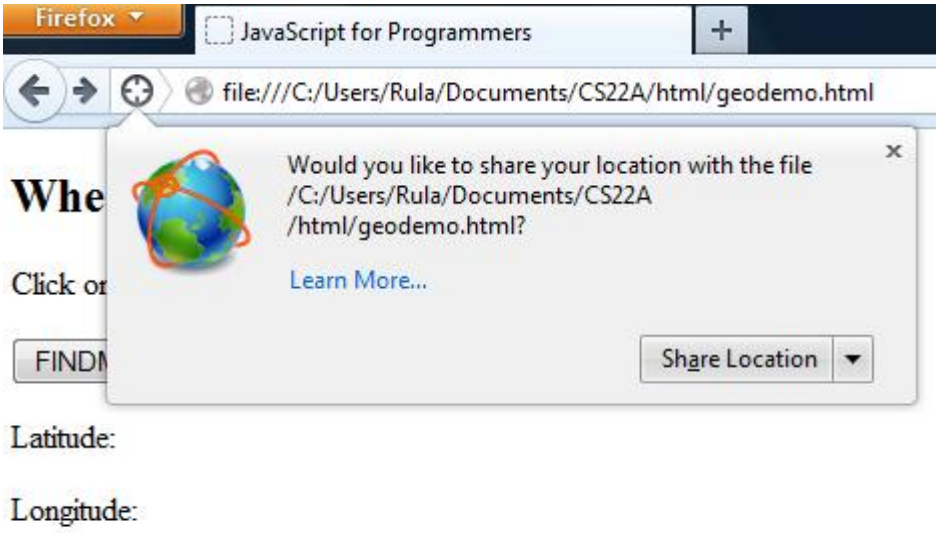
We use the **getCurrentPosition()** method to access the position. This method is **asynchronous**. It takes a callback function. We invoke it as shown in our scipt geodemo.js below.

geodemo.js

```
// Show the latitude and the longitude corresponding to the given position
function showPosition(position)
```

```
 {
    document.getElementById("latitude").textContent = position.coords.latitude;
    document.getElementById("longitude").textContent = position.coords.longitude;
 }
document.getElementById("findme").addEventListener("click", function (){
   navigator.geolocation.getCurrentPosition(showPosition);
});
```

Let's open the web page geodemo.html in Firefox and click on the FINDME button:



The browser first asks for our permission to share our location. If we give our permission, then our position will be available to the script.

watchPosition() is another method that keeps returning the updated position as the user moves and until we invoke clearPosition(). It allows us to track the user's movement.

Note that the position is more accurate for devices with GPS.

18.7. Web Workers

Client side JavaScript is essentially single threaded. If a function takes too long to perform a certain task the web page becomes unresponsive to user input.

The Web Workers API allows us to **run computationally intensive JavaScript code in a background thread without affecting the performance of the web page**.

We'll illustrate the use of web workers with a very simple example.

Let's start with a source document, workerdemo.html based on calculator.html. For the sake of simplicity, we'll put our html source document and scripts in the same directory in this example.

```
<!DOCTYPE html>
```

```
<html>
<head>
<meta charset="utf-8">
<title>JavaScript for Programmers</title>
</head>
<body>
<h2>Web Workers at work</h2>
<h4>This is our old calculator</h2>
<p>Please enter two numbers:  </p>
<input id = "first" type="number">
<br>
<input id = "second" type="number">
<br>
<p> And the answer is:</p>
<p id="answer"></p>
<p>Click on the button to start a web worker in the background.</p>
<input id = "start" type="button" value="START">
<p id="result"></p>
<script defer src="workerdemo.js"></script>
</body>
</html>
```

Here we have the simple calculator from our previous example but we also have a button that allows the user to start some computationally intensive task in the background. For now we'll assume that the task will be performed in the script compute.js.

Let's write our main script workerdemo.js:

```
function update() {
   // Get the two input numbers
   var firstNumber = parseFloat(document.getElementById("first").value) || 0;
   var secondNumber = parseFloat(document.getElementById("second").value) || 0;
   // Then  compute the sum
   var myAnswer = firstNumber + secondNumber;
   // And write it in the “answer” element if it is a valid answer
```

```
   document.getElementById("answer").textContent = myAnswer;
};
function startWorker() {
   var worker = new Worker("compute.js");  // instantiate worker object
   worker.addEventListener("message", function(event) {
     document.getElementById("result").textContent = event.data;
   });
};

document.getElementById("first").addEventListener("input", update, false);
document.getElementById("second").addEventListener("input", update, false);
document.getElementById("start").addEventListener("click", startWorker, false);
```

This script is very similar to the add.js script we've seen earlier. It has one new function that is called when the user clicks on the START button. That function startWorker() starts by **instantiating a new worker object that will execute to code found in the script compute.js.**

```
   var worker = new Worker("compute.js");
```

The main script and the worker script can now communicate via messages: we add an event listener to our worker object so that when a message is detected we display the data returned in that message to the web page.

For demonstration purposes, we'll write our compute.js script as follows:

```
// This web worker performs some computations
// and returns the result
var sum = 0;
for (var i = 0; i < 90000000; i++) {
   sum += Math.pow(i, 5)
   }
postMessage(sum);
```

This web worker computes the sum of the first 90,000,000 numbers raised to the power of 5. It sends a message back to the server with the result.

In practice, a web worker may be performing some complex calculations or image processing…

Once we have saved workerdemo.html, workerdemo.js and compute.js in the same directory, we can open workerdemo.html in the browser, click on the START button to

start the “compute.js” script in the background then enter some numbers.

The important thing to note here is that **while compute.js is performing the computation in the background (and before the result is displayed under the START button), the web page is responsive to our input and the calculator part of the web page is still working**. This would not have been the case if the calculations were performed as part of the main thread.

18.8. Web Sockets

The WebSocket API implements a new communication protocol that enables us to develop **low latency, real-time applications** such as multi-player online games.

Instead of the stateless (restful) HTTP requests and responses that we’ve seen so far, web sockets define a **persistent bidirectional connection between a client and a server.**

To create a connection, the client script has to instantiate a web socket object and specify a url:

```
var connection = new WebSocket ("ws://host:port/resource");
```

ws is the protocol here. We can also use wss for the secure version (corresponding to https). Then we specify the host, possibly the port number and resource. Web sockets may share ports with http connections.

Once we have created a connection, we can send data on it as plain text. JSON encoding is very handy here when we have some complex data to transmit. The latest browsers also allow binary data to be sent.

```
connection.send (“Hello Server World!”);
```

We can also register event handlers on the connection.

```
connection.onmessage (function (event) {
   var  message = event.data; // Get the message
   //do something with it
};
```

When we are done we can close the connection:

```
connection.close();
```

The important thing to realize here is that once we open the connection, the server can send us messages at any time without waiting for a specific request.

The web socket API is new and not available yet in all browsers.

socket.io is a library that has become very popular: it includes a client side library that runs in the browser and a server side library for node. It primarily uses the web socket protocol, but when the browser does not support it, it can fall back on other communication methods.

19. The Model-View-Controller Design Pattern

19.1. Overview

The model-view-controller (MVC) pattern is a software design pattern that is widely used in web application development.

The model view-controller paradigm organizes an application into three components.

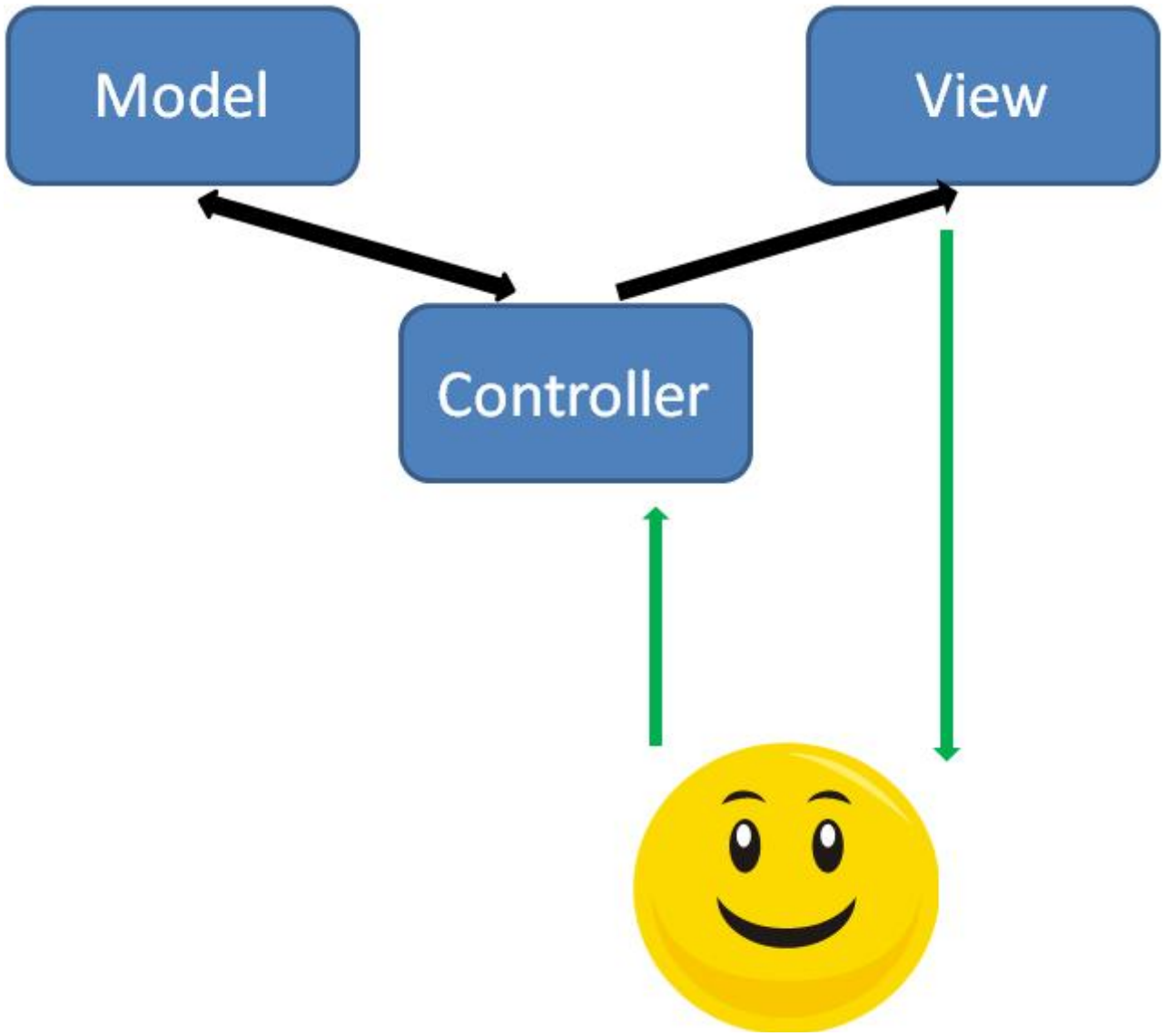
The **model** manages the **data** manipulated by the application.

The **view** deals with the specific **presentation** of the data to the user.

The **controller** contains the **logic** of the application. It mediates the interaction with the model and the view.

Each component has a well defined interface and interacts with the other components through that interface.

The interaction between the components and the user can be illustrated as follows:



Note that the user 'sees' the application through the view component. However the user's actions are handled by the controller component.

The model component is accessible only through the controller. The controller can send queries as well as updates to the model.

The controller does not present data to the user directly. It invokes the view component to do so.

Consider a simple business review web application such as the one we encountered in assignment 4.

The model deals with the data associated with the various businesses. That data includes the names, descriptions, ratings, prices, addresses and so on and is probably stored in some database on the server.

The application may include several views: one that presents the user with the data specific to one business and another that includes some summary data about several businesses at once (a view with all businesses sorted by rating, for example).

Finally the controller is responsible for handling the interaction with a user. For example it may respond to a given user request by getting a subset of the data from the model to be presented with a given view.

19.2. Server Side or Client Side?

Web applications used to follow a thin client approach where the model, view and controller resided entirely on the server. Nowadays, it is more common to see MVC components on the client side.

Going back to our business review app, let's say that the user is currently viewing a number of nearby restaurants. What if the user wants to focus on the restaurants with the highest ratings and wants to view these restaurants sorted by (decreasing) rating. There is obviously no need to go back to the server for this. Handling this request in the client side JavaScript 'controller' is easy and will result in a more responsive application.

Similarly we may have a **model** component on the client side to **manage the data in local storage or session storage or even to get data from the server through ajax calls**.

Finally, the **view** component on the client side would be responsible for **updating the web page through DOM manipulation.**

MVC Frameworks

Several frameworks that enforce the MVC design pattern are available on the server side as well as on the client side.

Rails and ASP.NET MVC are popular server side (non JavaScript) MVC frameworks.

On the client side, JavaScriptMVC is an open source MVC framework based on jQuery.

19.3. Advantages

Even though applying the model-view-controller design pattern to a web application

may introduce some overhead work, the benefits of doing so become clear in the long term and as the size and complexity of the application increase.

Here are some of the advantages of the model-view-controller paradigm:

1. Complexity tamed: instead of looking at the daunting task of implementing all aspects of the web application at once, we can focus on one component at a time.

2. Easier maintenance: changes made to one component will not affect the others (as long as the interface is kept intact.) Maintenance of the web application becomes easier.

3. Reuse: a controller written for one web application may be reusable in a completely different application with different model and views.

4. Testing: the three components can be tested independently. Stubs and mocks may be used to emulate the functionality of the 'other' components.

5. Specialization: the separation of the components allows different developers to work on the different aspects of the web application. This is useful when developers have specialized skills.

6. Parallel Development: the three components can be developed in parallel by different teams.

19.4. Example

Let's go back to our simple calculator example and implement it using the MVC design pattern. To justify the addition of a model component, we'll add the requirement that the last numbers entered by the user be stored in the local storage.

We'll modify our HTML source document to point to the new MVC script.

calculatormvc.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>My Simple Calculator</title>
</head>
<body>
<h2> Let's add some numbers! </h2>
<p>Please enter two numbers:  </p>
<input id = "first" type="number">
<br>
<input id = "second" type="number">
<br>
```

 And the answer is:

<script defer src="../scripts/**addmvc.js**"></script>

</body>

</html>

We modify our initial JavaScript add.js code to separate the code into three distinct components: a model, a controller and a view. **Each of these components will be a JavaScript object and we'll define methods on these objects to handle the tasks associated with them.**

Note that in this trivial example, the controller only sends updates to the model but in a more general case, the controller may use the model to get access to the data as well.

addmvc.js

```
// MVC implementation of our simple calculator
// The controller consists of one function: update.
// It is invoked when the user types anything in the input fields
var controller = {
    update: function () {
    // Get the input from the view
    var input = view.getInput();
    // Then  compute the sum
    var firstNumber = parseFloat(input[0]) || 0;
    var secondNumber = parseFloat(input[1]) || 0;
    var myAnswer = firstNumber + secondNumber;
    // update the model
    model.save(firstNumber, secondNumber);
    // update the view
    view.showResult(myAnswer);
    }
};
// The model saves the data to local storage.
var model = {
    save: function (first, second) {
        localStorage.setItem("firstNumber", first);
        localStorage.setItem("secondNumber", second);
    }
};
```

```
// The view component has 2 functions: getInput and showResult
var view = {
    getInput: function () {
        var firstInput = document.getElementById("first").value ;
        var secondInput = document.getElementById("second").value;
        return ([firstInput, secondInput]); // return an array with the 2 inputs
    },

    showResult: function (result) {
        document.getElementById("answer").textContent = result;
    }
};

// Event listeners
document.getElementById("first").addEventListener("input", controller.update, false);
document.getElementById("second").addEventListener("input", controller.update, false);
```

20. Web Application Design Considerations

20.1. 3-Tier Architecture

In module 15 we introduced a high level view of the web application as a client server architecture.

Web applications on the server side are usually structured in three tiers.

The **presentation tier** usually consists of an HTTP server which listens on an HTTP port and accepts requests from the outside world. We have written our own server in node that accepts requests and serves static HTML documents.

In a more complex web application the HTTP server forwards requests to the **logic (or application) tier**. The logic tier contains our actual application code. It may generate dynamic content based on user requests or credentials. It may also collect and update information from the user or it may even support financial transactions. To access the application data, the logic tier relies on the persistence tier.

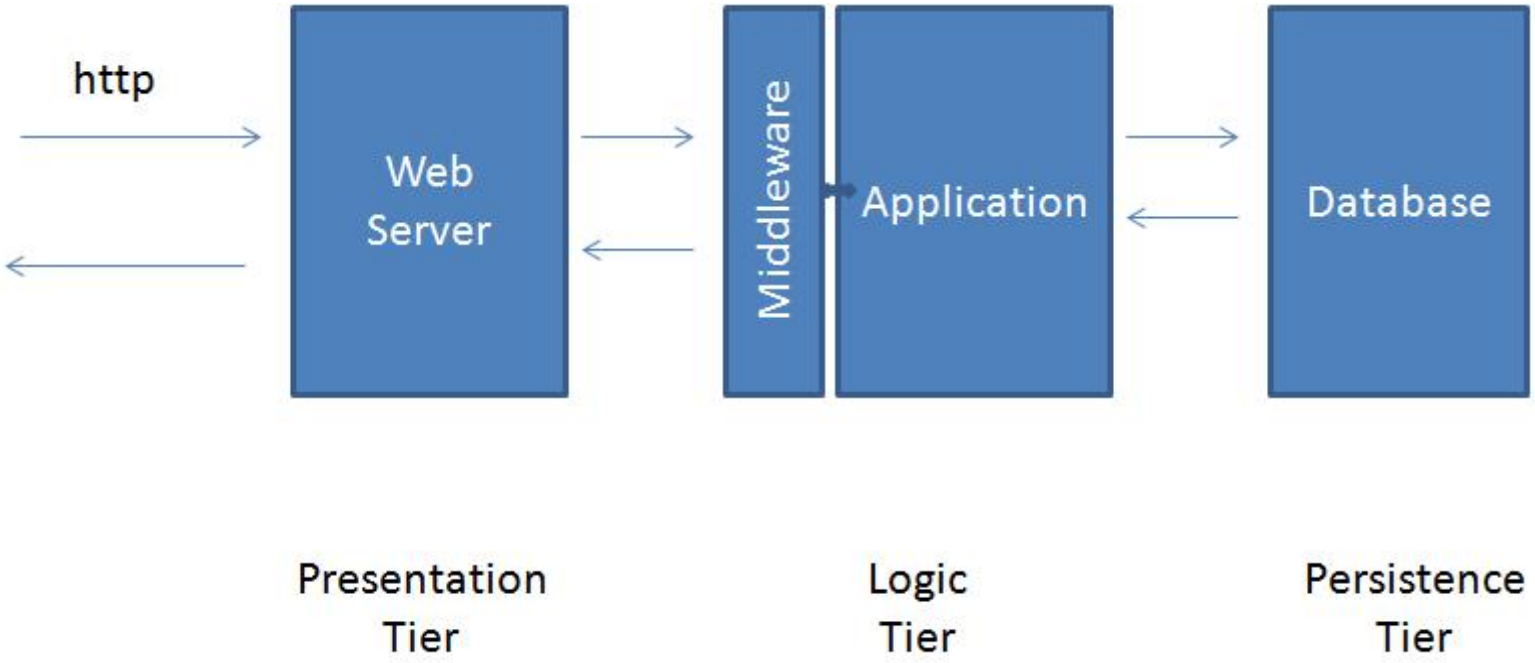
With complex web applications, we also need an application server. The application server handles the details of HTTP requests for our logic tier and allows us to **define a high level routing to be associated with each request**. For example, it can route incoming HTTP requests directly to the appropriate components in our application based on the request verb (GET requests vs POST requests).

Because an application server sits between the presentation tier and our actual application code, it is referred to as **middleware**.

Connect and Express are two middleware frameworks available for node.

The **persistence tier** is usually implemented using a database management system such as MySQL.

The three tiers may actually reside on different computers. It's even common for each tier to span several computers.



It's important to note that the lines between the three tiers are sometimes blurred. For example we can implement a web application using Apache Tomcat in conjunction with Java servlets. Tomcat is both a web server and servlet container: it listens for http requests and as needed, creates and initializes a servlet to service a given request.

Whereas the java servlets belong to the logic tier, tomcat itself also belongs to the presentation tier.

20.2. The Web Stack

When we develop a web application, we are faced with choices at all levels.

What web server do we use? Some servers (such as the one we implemented in node) support event driven, asynchronous handling of requests. Others such as Apache web servers create a new process for each request.

What framework/programming language is best suited for our application (on the server side)?

What about the persistence tier?

We commonly refer to a combination of choices as a web stack.

In making our choices, we need to take into account several factors: security considerations, anticipated load, scalability, ease of development (based on our expertise), complexity of the application, maintainability and so on. It's also helpful to use a well-tested solution.

LAMP is one of the earliest and most commonly used web stacks.

Its name refers to its original components:

- Linux for the operating system.
- Apache for the HTTP server.
- MySQL for the database management system.
- PHP (or Perl or Python) for the programming language used in the application code.

The four components of LAMP are free and open-source software.

The LAMP stack is used to power thousands of sites today: it offers a well-tested powerful and reliable solution that is also easy to implement.

20.3. Scalability

It's important for a web application to be able to handle growth. One approach for building scalability into web applications is to **allow each tier to span several computers**.

In this case we have several web servers handling http requests and routing them to one of several application servers. We can then add computers to each tier to match the growing demand. This is known as horizontal scaling.

Load balancers (which could be dedicated software or hardware) are responsible for distributing the work load evenly.

For this approach to work it is important that a component does not need to share anything with its counterpart on a different computer so there is no bottleneck involved around that shared resource. This is referred to as the **shared-nothing architecture**.

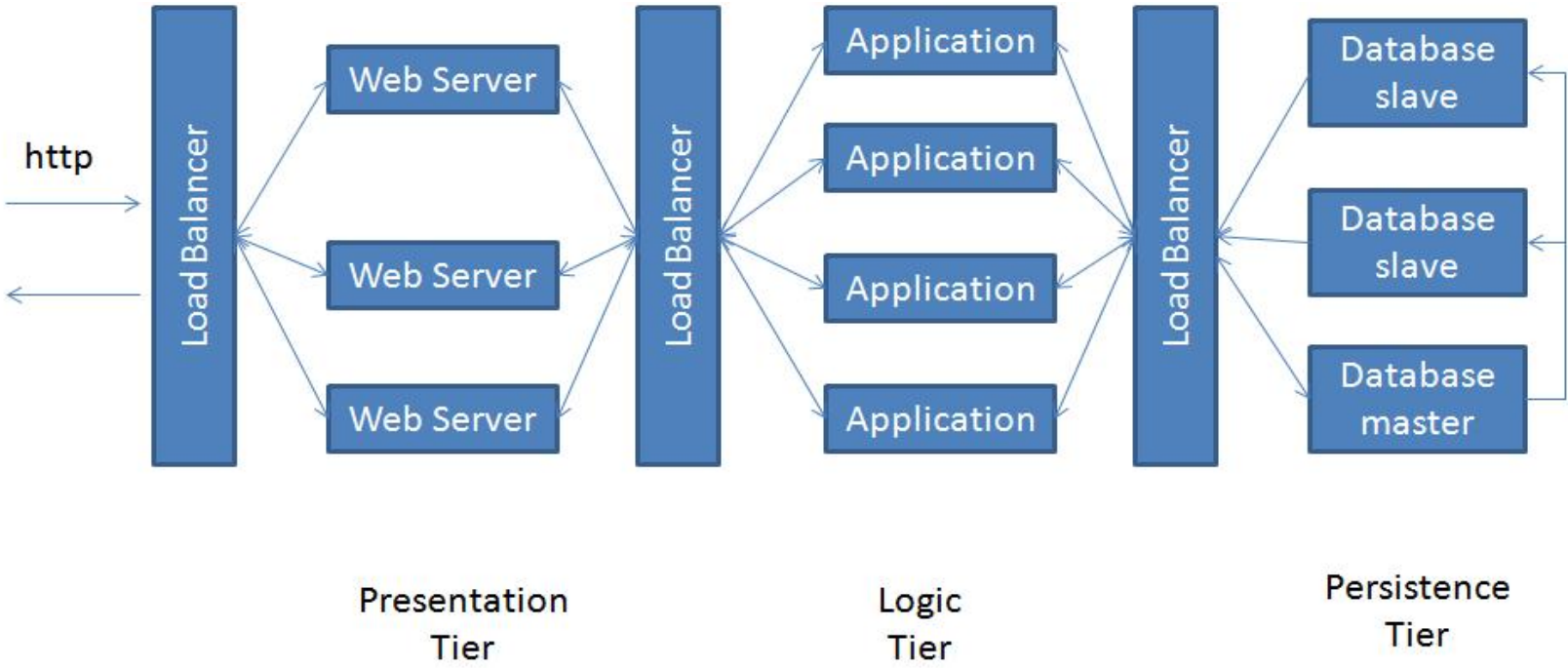
The fact **that HTTP is a stateless protocol** where requests are **RESTful** makes that possible: each **request from any client contains all of the information necessary to service the request**.

As a result, any server in the presentation layer can handle any request. Any application server in the logic tier can handle any request (even if the previous request from that client was assigned to a different server).

Scaling becomes more challenging when we get to the persistence tier. The shared nothing approach does not apply to databases that receive updates. Techniques such as database replication or sharding are used to alleviate bottlenecks in the persistence tier.

When the database is read much more frequently than it is written, several copies of the database are kept on different computers. The copies are referred to as slaves. The original is the master. Any slave can perform reads, only the master can perform writes, and the master updates the slaves as quickly as possible.

Another approach is sharding. The database is split horizontally (by rows) and the smaller parts reside on different computers. The split may be along geographical location or any other criteria.



21. Testing

21.1. What do we test?

It is essential to test a web application thoroughly before releasing it. It is even better to **write the tests before writing the code**. That is what **Test Driven Development (TDD)** is all about: you write the test based on how you would like the code to behave, then you write the code that makes the test pass. This forces you to really understand your requirements before you start writing the code.

For a web application, there are several testing aspects that need to be considered. Some of these aspects, such as functional testing or usability testing, are common to most software development endeavors. Others, such as compatibility testing, load testing and security testing, address challenges inherent to web applications.

Functional Testing:

Does our application do what it is supposed to do? Does it behave correctly? Does it produce the expected outcome?

There are several tools available to create test cases and automate functional testing. We'll take a closer look at Selenium in the next section.

Usability Testing:

Is our application easy to navigate? Is it obvious to the user which actions are available? Does the user interface match the target audience's needs?

Compatibility Testing

Does our application run on all browsers, or on all 'modern' browsers, or on a subset thereof?

Does it run on all devices including phones and tablets or is the functionality dependent on the device?

Security Testing:

Does our application protect the user's information? Is it vulnerable to any kind of hackers' attacks?

Performance Testing:

What response time does the user experience under normal conditions? Are there any bottlenecks in our application that we need to address? We'll look more closely at performance issues in web applications in an upcoming section.

Load Testing:

Does our application perform well regardless of how many users it is serving? Remember the issues surrounding the launch of the health care web site healthcare.gov…

21.2. Selenium

Selenium is a **browser automation tool** that is commonly used for automating the testing of web applications. It includes two major components: Selenium IDE and Selenium WebDriver.

Selenium IDE is a Firefox plugin. It lets us record our interactions with the browser as a script. The script is recorded in Selenese, a special language for Selenium. We can then edit the script or play it back. Selenese provides commands for performing actions in a browser (such as clicking a button or typing something), as well as commands for retrieving data from the resulting pages.

Scripts generated with the Selenium IDE can also be exported to Selenium WebDriver.

Selenium WebDriver accepts commands, sends them to a browser and retrieves the results. The commands may be written in Selenese or in any of the supported languages (through special language drivers). The core languages supported at this time are Java, C#, Ruby, Python and JavaScript (Node).

Selenium IDE Demo

To install the Selenium IDE, follow the detailed instructions at: http://docs.seleniumhq.org/docs/02_selenium_ide.jsp#

After you restart Firefox and open the IDE, read through the IDE features (described on that page) to gain some familiarity with the tool.

In the following demo, we'll just record some interactions with our guessing game app (guess.html) and demonstrate how we can check the results.

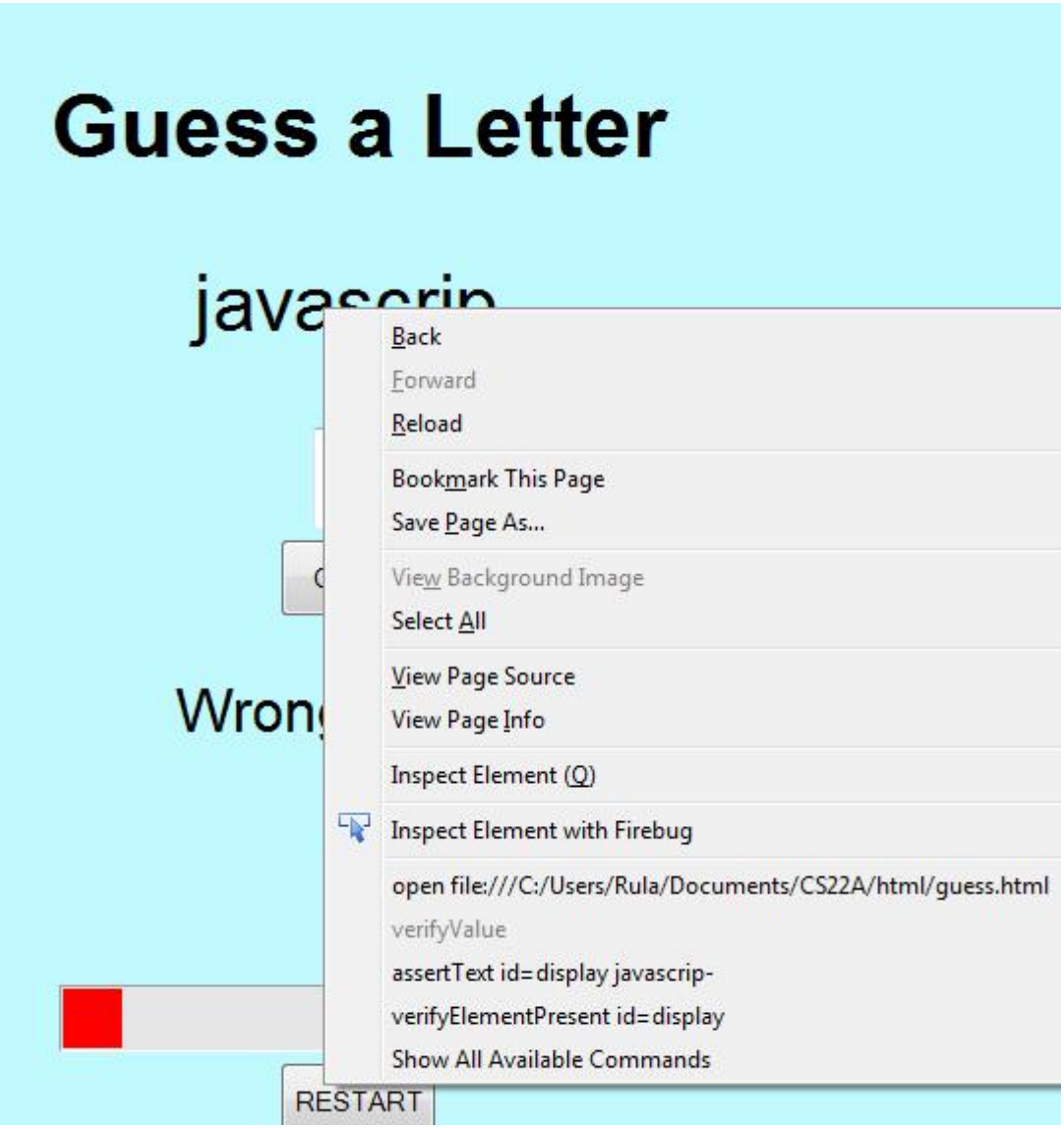
We'll first open Firefox and the Selenium IDE and make sure that the red recording button is activated.

In the browser address bar, we'll just enter the file: scheme address of the word guessing game guess.html (file:///.../..../guess.html). The exact address will depend on your configuration.

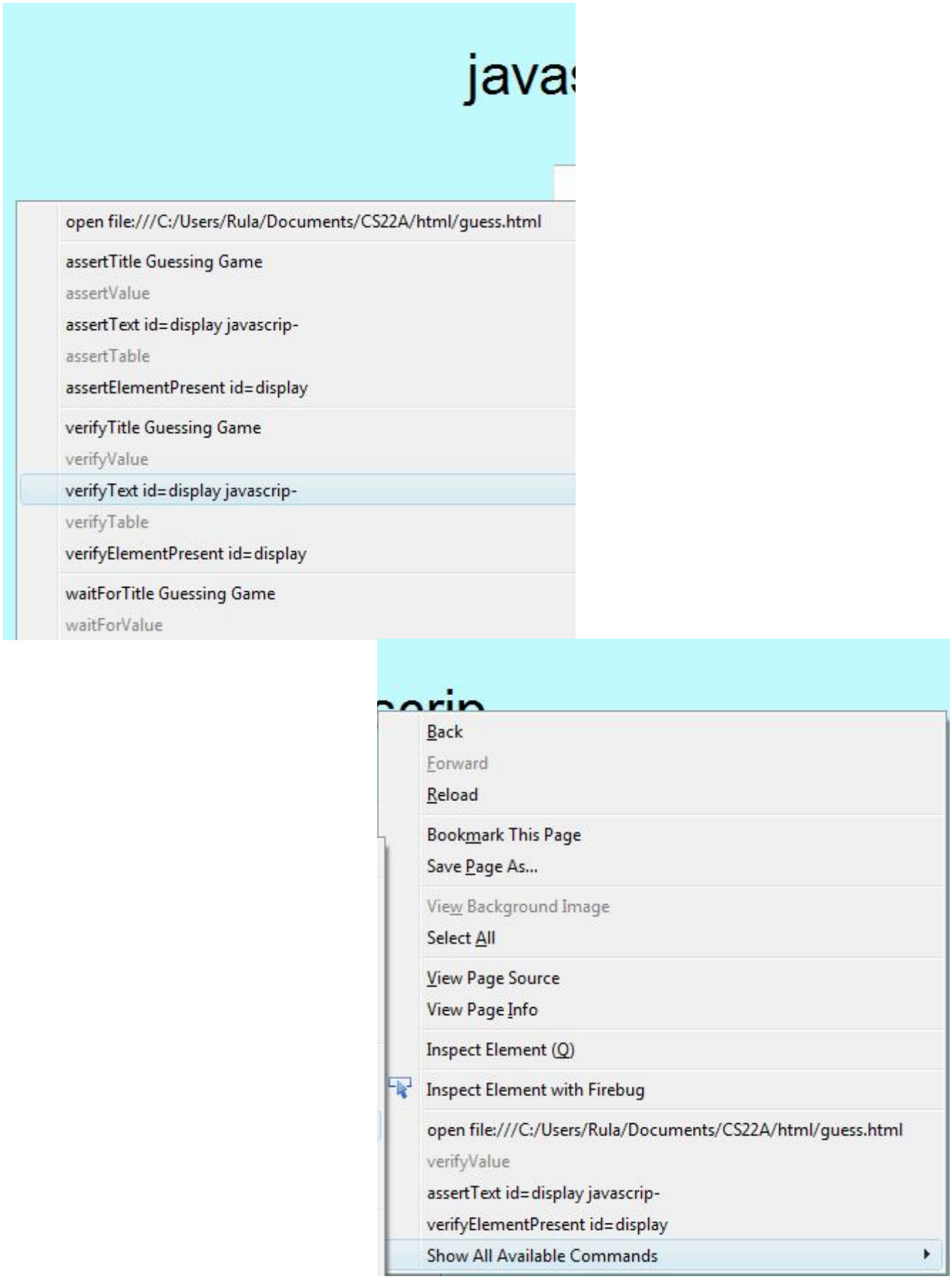
Then we'll start interacting with the app. For demonstration purposes, we'll enter the

following letters, in order, pressing the GUESS button after each letter: j, a, v, s, c, r, i, p then q.

At this point we'll move our mouse over the dashed display javascrip- and right click. The following options are shown:

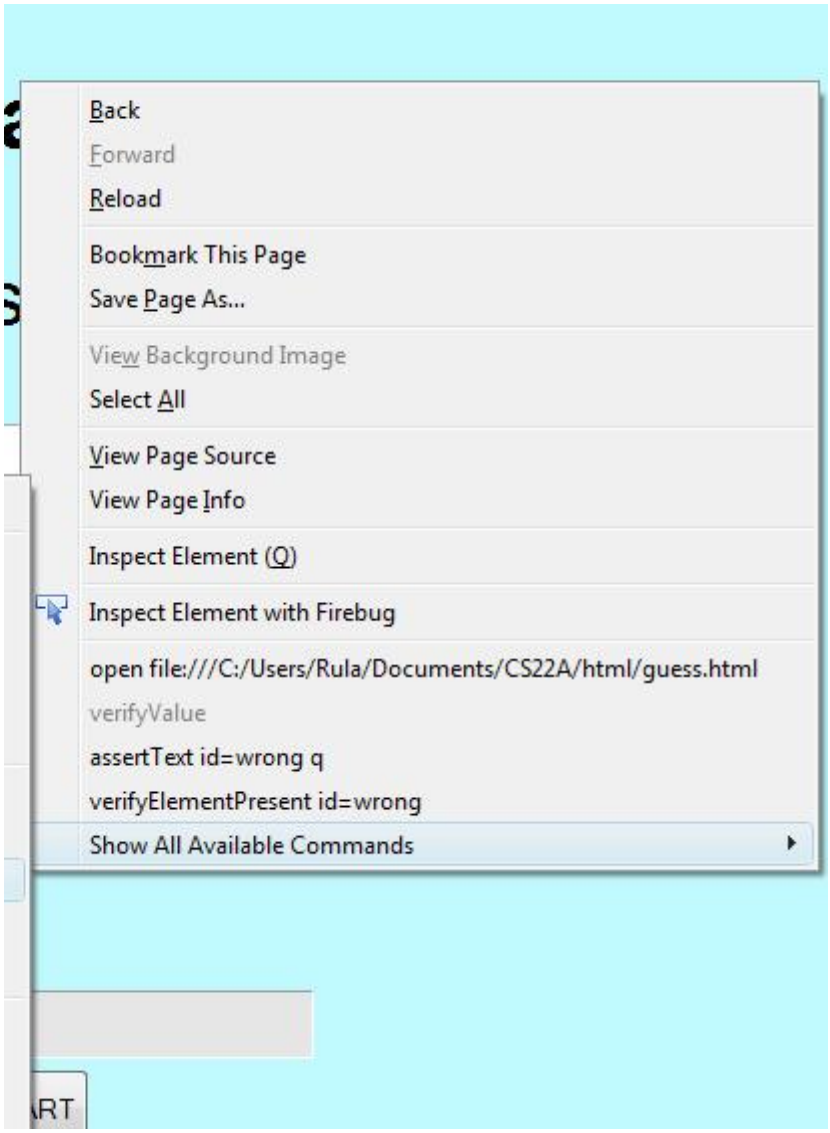


We'll select ''Show All Available Commands' first and then 'verify Text id=display javascrip-' as shown below.


java
open file:///C:/Users/Rula/Documents/CS22A/html/guess.html
assertTitle Guessing Game
assertValue
assertText id=display javascrip-
assertTable
assertElementPresent id=display
verifyTitle Guessing Game
verifyValue
verifyText id=display javascrip-
verifyTable
verifyElementPresent id=display
waitForTitle Guessing Game
waitForValue
Back
Forward
Reload
Bookmark This Page
Save Page As...
View Background Image
Select All
View Page Source
View Page Info
Inspect Element (Q)
Inspect Element with Firebug
open file:///C:/Users/Rula/Documents/CS22A/html/guess.html
verifyValue
assertText id=display javascrip-
verifyElementPresent id=display
Show All Available Commands

This is how we specify to Selenium to check the page against a known result. We are simply checking that all the correct letters that we guessed are displayed in the HTML element whose id is display and the last character is still a dash: verify Text id=display javascrip-

The next thing to check is that the letter q has appeared under the Wrong Letters. We can do that again by right clicking on the corresponding element and selecting ‘verify Text id=wrong q’ as shown below.



At this point we can stop recording, save the test case and of course replay it many many times. We can also edit the script generated.

The one thing to note here is that there are **three main options to check results: an assert option, a verify option and a waitFor option.**

When an assert fails, the test is aborted. When a verify fails, the test will continue execution, logging the failure.

waitFor commands wait for some condition to become true (which can be useful for testing Ajax applications). They will succeed immediately if the condition is already true. They will fail and halt the test if the condition does not become true within a certain timeout setting.

Let's demonstrate the use of waitFor with our ajaxdemo.html example. Just remember to serve that page with our in house node server.

Here again we start recording in Selenium then in the Firefox browser, we

type: http://localhost:8080/ajaxdemo.html

After the page has loaded, we right click on the heading Ajax Demo and select the command: 'waitForText css=h2 Ajax Demo'. We are basically telling Selenium to wait for 'Ajax Demo' to appear in the h2 header.

We then press on the CLICK button. The line 'This is the additional data requested from the server.' is supposed to appear under the button. We can then right click on the line and select the command: 'waitForText css=span This is the additional data requested from the server.'

So here we are basically telling Selenium to wait for the Ajax response from the server.

At this point we can stop recording, save the test case and of course replay it over and over…

21.3. Performance Tuning and Latency

Several metrics may be used to assess the performance of a web application.

Latency is defined as the time between making a request and beginning to see a result. It may also be defined as the time between making a request and the completion of the response.

In both cases we are concerned about the response time that the user sitting in front of the browser, experiences. How long does it take to load the web page?

That response time is determined by the following components:

- How long does it take to generate the web page (on the server)?
- How long does it take for the browser to parse it?
- How long does it take to download all the components (scripts, stylesheets, images)?
- How long does it take for the browser to render it?

For most applications, the response time is dominated by the time needed to download all the components. As web applications have become increasingly complex, the size of of the associated components has considerably increased. Depending on where the user is located, and depending on their network access, it can take a significant amount of time to fetch all the components.

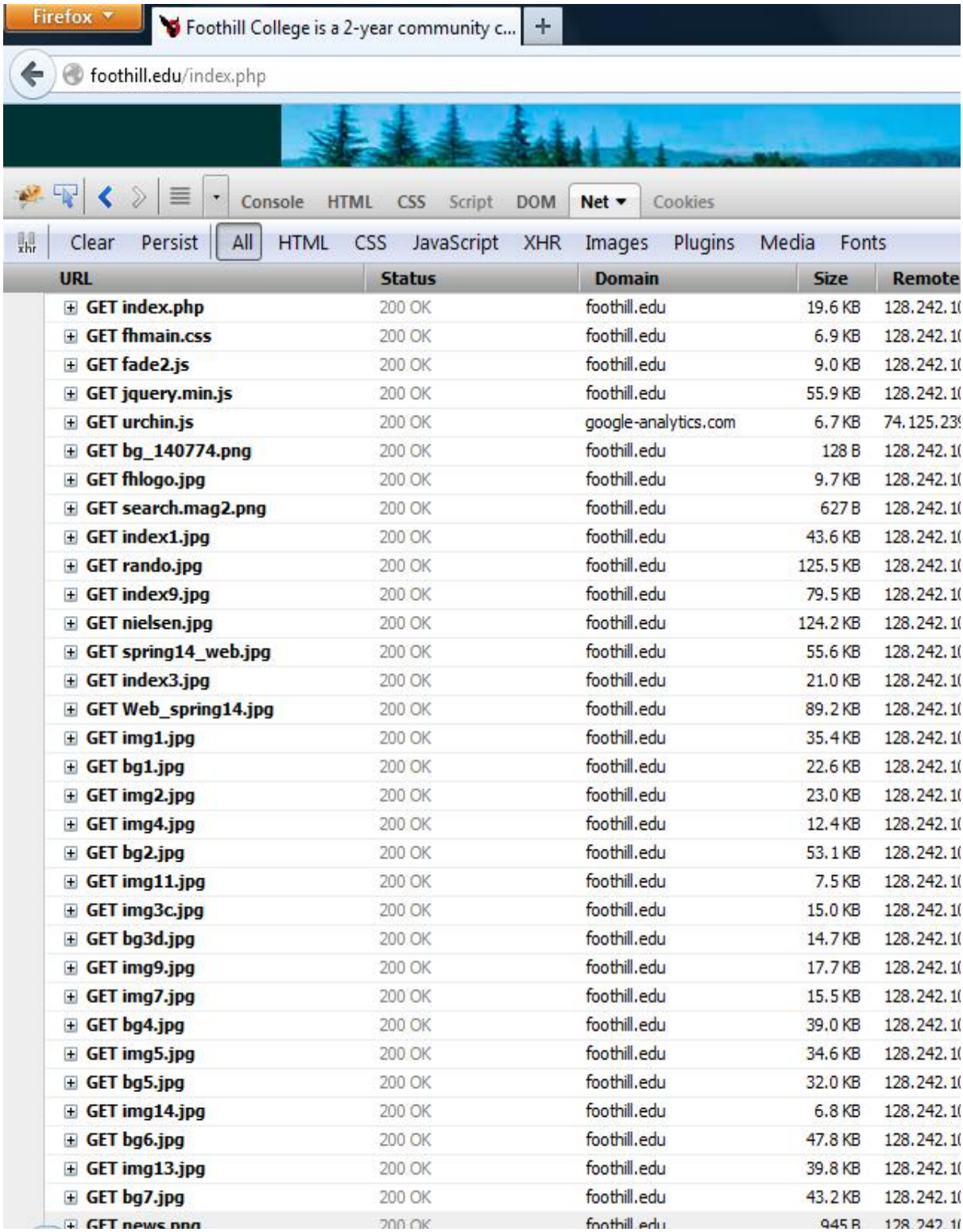
Many tools are available to measure the performance of a web application from the client side. The tricky part is being able to replicate a typical user's environment.

Firebug includes a Net panel that may be extremely helpful in identifying performance issues.

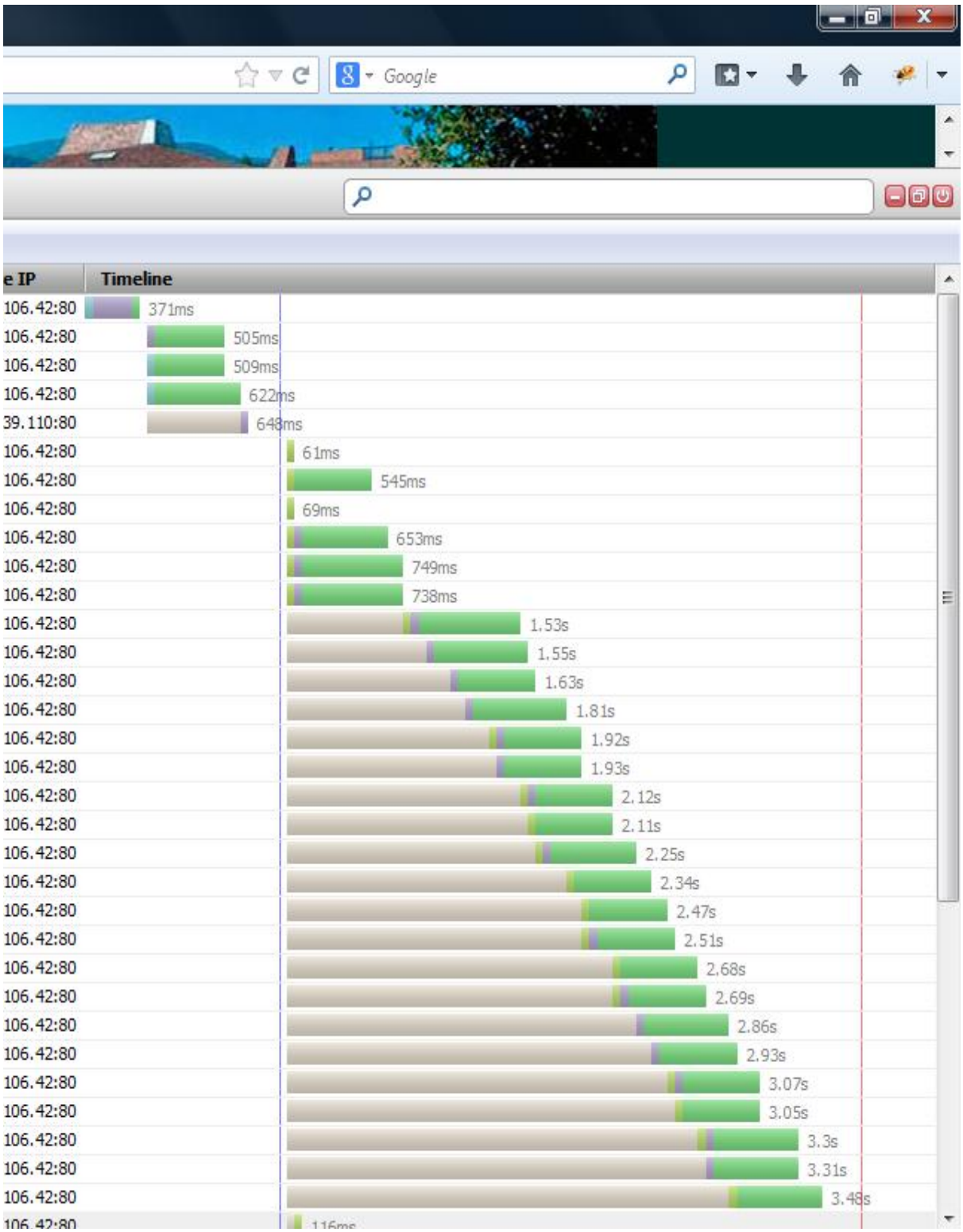
To see how that works, we first open Firebug, enable the Net panel by clicking on the Net tab and selecting Enable. We can then type a url address in the address bar, and see detailed information and a timeline corresponding to all the requests that had to be sent to get the initial web page to load.

The screenshot below shows the timeline corresponding to loading the page

at http://foothill.edu/index.php



URL	Status	Domain	Size	Remote
GET index.php	200 OK	foothill.edu	19.6 KB	128.242.1
GET fhmain.css	200 OK	foothill.edu	6.9 KB	128.242.1
GET fade2.js	200 OK	foothill.edu	9.0 KB	128.242.1
GET jquery.min.js	200 OK	foothill.edu	55.9 KB	128.242.1
GET urchin.js	200 OK	google-analytics.com	6.7 KB	74.125.23
GET bg_140774.png	200 OK	foothill.edu	128 B	128.242.1
GET fhlogo.jpg	200 OK	foothill.edu	9.7 KB	128.242.1
GET search.mag2.png	200 OK	foothill.edu	627 B	128.242.1
GET index1.jpg	200 OK	foothill.edu	43.6 KB	128.242.1
GET rando.jpg	200 OK	foothill.edu	125.5 KB	128.242.1
GET index9.jpg	200 OK	foothill.edu	79.5 KB	128.242.1
GET nielsen.jpg	200 OK	foothill.edu	124.2 KB	128.242.1
GET spring14_web.jpg	200 OK	foothill.edu	55.6 KB	128.242.1
GET index3.jpg	200 OK	foothill.edu	21.0 KB	128.242.1
GET Web_spring14.jpg	200 OK	foothill.edu	89.2 KB	128.242.1
GET img1.jpg	200 OK	foothill.edu	35.4 KB	128.242.1
GET bg1.jpg	200 OK	foothill.edu	22.6 KB	128.242.1
GET img2.jpg	200 OK	foothill.edu	23.0 KB	128.242.1
GET img4.jpg	200 OK	foothill.edu	12.4 KB	128.242.1
GET bg2.jpg	200 OK	foothill.edu	53.1 KB	128.242.1
GET img11.jpg	200 OK	foothill.edu	7.5 KB	128.242.1
GET img3c.jpg	200 OK	foothill.edu	15.0 KB	128.242.1
GET bg3d.jpg	200 OK	foothill.edu	14.7 KB	128.242.1
GET img9.jpg	200 OK	foothill.edu	17.7 KB	128.242.1
GET img7.jpg	200 OK	foothill.edu	15.5 KB	128.242.1
GET bg4.jpg	200 OK	foothill.edu	39.0 KB	128.242.1
GET img5.jpg	200 OK	foothill.edu	34.6 KB	128.242.1
GET bg5.jpg	200 OK	foothill.edu	32.0 KB	128.242.1
GET img14.jpg	200 OK	foothill.edu	6.8 KB	128.242.1
GET bg6.jpg	200 OK	foothill.edu	47.8 KB	128.242.1
GET img13.jpg	200 OK	foothill.edu	39.8 KB	128.242.1
GET bg7.jpg	200 OK	foothill.edu	43.2 KB	128.242.1
GET news.png	200 OK	foothill.edu	945 B	128.242.1



We can also select to view a subset of the requests such as JavaScript only or images only.

Note that the timeline will depend on whether or not some of these resources have been cached earlier.

21.4. Load Testing and Throughput

Load testing is primarily focused on back end performance. Its objective is to determine the **maximum throughput** that a web application can sustain. **Throughput is defined as the number of requests (or transactions) that an application can handle per unit of time.**

Several load testing tools are available today. They typically generate a heavy concurrent load, verify responses, and produce a variety of reports. To determine the maximum throughput, the number of requests needs to be increased until the throughput levels off or starts to drop. That usually happens because of a bottleneck in the application (commonly found in the persistence layer.)

22. Mobile Platforms

22.1. Web Applications vs Native Applications

On mobile platforms, users have access to two main categories of applications.

Native applications live on the mobile device and are installed through an app store. They are **developed specifically for one platform**, and can take advantage of the device features.

Web applications are available through a browser. Users first access them through a web address and then have the option to install them on the home screen.

22.2. Special Considerations

From a user's perspective the distinction between mobile web applications and native applications is anticipated to become more blurred, as mobile browsers gain direct access to the hardware of mobile devices (including accelerometers, GPS and possibly cameras), and the speed and capabilities of browser based applications increase.

However there are still some issues surrounding the use of web applications on mobile platforms. Here are some of them:

The small screen size represents some usability challenges.

To remedy that, web applications can offer a scaled-down version optimized for small screens. This can be achieved through the use of CSS3 media queries and other Responsive Web Design (RWD) techniques (sizing images in relative units, etc.) With these techniques, we can use the same code to present content at its best on the device on which it is being viewed.

The data connection speed will usually be slower on mobile devices.

As a result, it is important for mobile web applications to **minimize the amount of data that the user has to download.**

This can be achieved by reducing the number of files that have to be served as well as **keeping the sizes of these files smaller.**

We may want to reconsider the use of images and take full advantage of CSS3 features and HTML5 elements such as canvas to generate images instead of downloading them. For the images we download, it is important to keep their size as

small as possible. After all, they will be displayed be displayed on a smaller screen.

The use of the **application cache** becomes essential when dealing with mobile devices. We have seen how the application cache allows us to have web applications available offline, when the device does not have internet access. It is also useful when data access is slow.

Finally, it is a common practice to ‘minify’ JavaScript code as well as syle sheets. **Minification** removes all comments and unnecessary white space characters from a file to reduce its size.