UNIVERSITÀ DI PAVIA

Collegio Alessandro Volta
Via Adolfo Ferrata, 17, Pavia (PV)

WEB DESIGN

Lecture 5 – Dynamic Scripting with JS & Design Languages

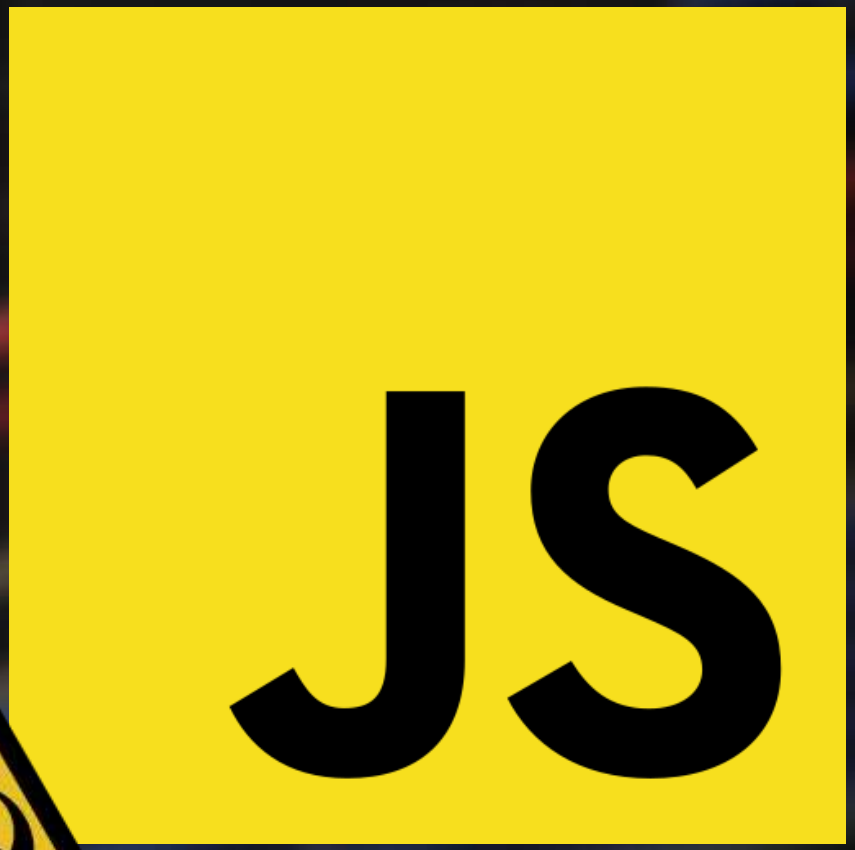
Giovanni Nicola D'Aloisio

Department of Physics – University of Pavia
Classe di Scienze, Tecnologie e Società – IUSS Pavia

E-Mail: giovanninicola.daloisio01@universitadipavia.it

Stop marking, start scripting!

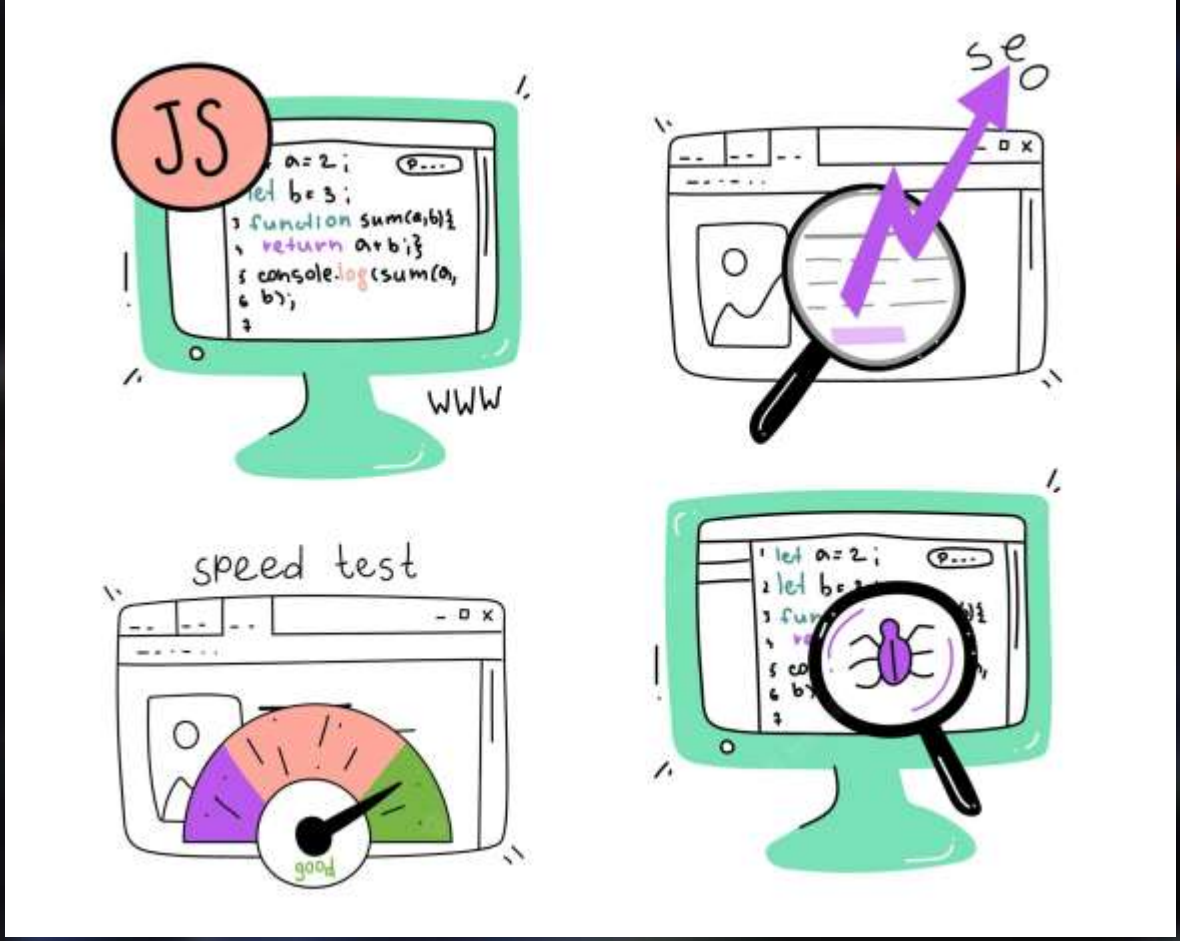
- JavaScript is an incredibly popular scripting language that remains poorly understood by many of the people who use it. This is largely because it can easily be picked up in the context of a specific project without learning it from the ground up.
- This is NOT a bad thing. JavaScript is a forgiving language and it's accessible, and JavaScript programmers come from a tremendously diverse set of backgrounds.
- As of 2022, 98% of websites use JavaScript on the client side for webpage behavior. All web browsers have a dedicated JS engine to execute the code on users' devices.



Please be careful and pay attention.
We are about to start coding.

What is a script?

- In IT, the term script is used when referring to a particular kind of software, written using a scripting language.
- A scripting language is a (usually) interpreted programming language that is used to manipulate, customize, and automate the facilities of an existing system.
- A scripting language's primitives are usually API calls, and the scripting language allows them to be combined into more programs.
- JavaScript was born as a scripting language, and later became a programming language (thanks to Node.js, Elektron.js, and others).



Working with the browser

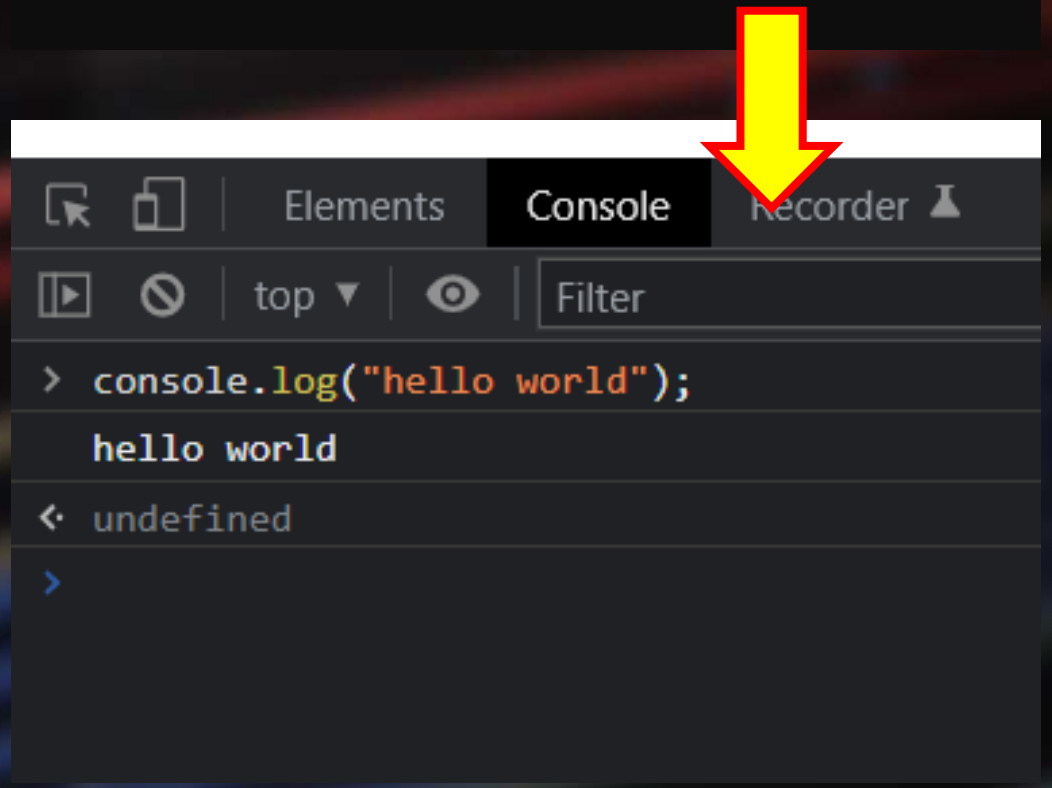
- We'll work with Chrome browser;
- Open up the browser and type `about:blank` into the address bar;
- You can access the JavaScript console by going to:

  View > Developer > JavaScript Console

- Type the code on the right at the prompt;
- Admire your first script.

JavaScript

```
console.log("hello world");
```



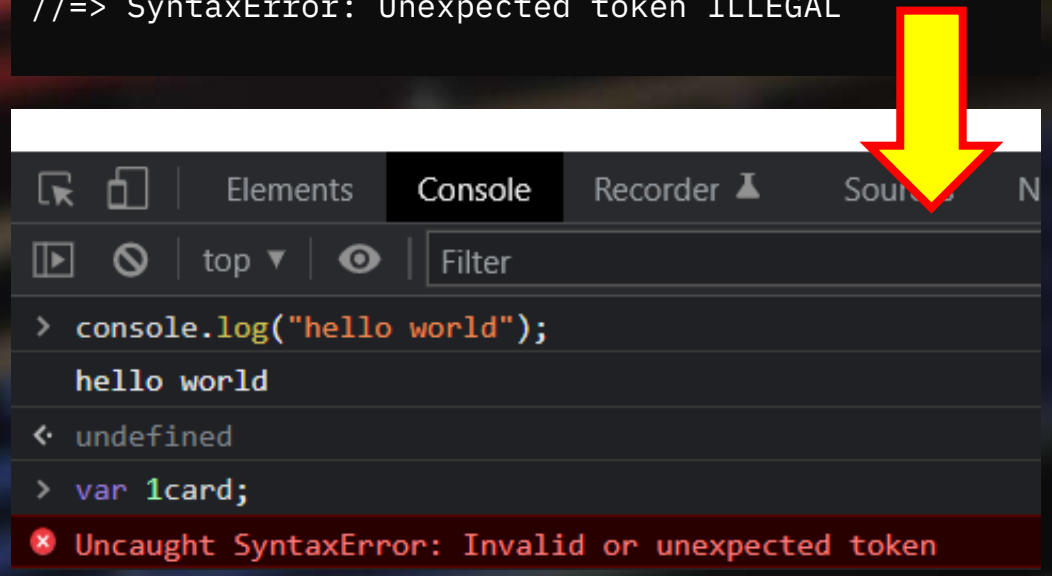
Declaring variables

- Computer programming is largely about storing and manipulating individual pieces of information, called data. Variables are names we can assign to data so that we are able to read or manipulate them later on.
- To declare a variable in JavaScript, we use the `var` keyword followed by the identifier we want to use; in this case, card1.
- We'll use letters and occasionally numbers to create our identifiers, although JavaScript won't let us use a number as the first character in an identifier.

JavaScript

```
// this identifier is fine
var card1;

// this identifier is not allowed, because it
starts with a number
var 1card;
//=> SyntaxError: Unexpected token ILLEGAL
```



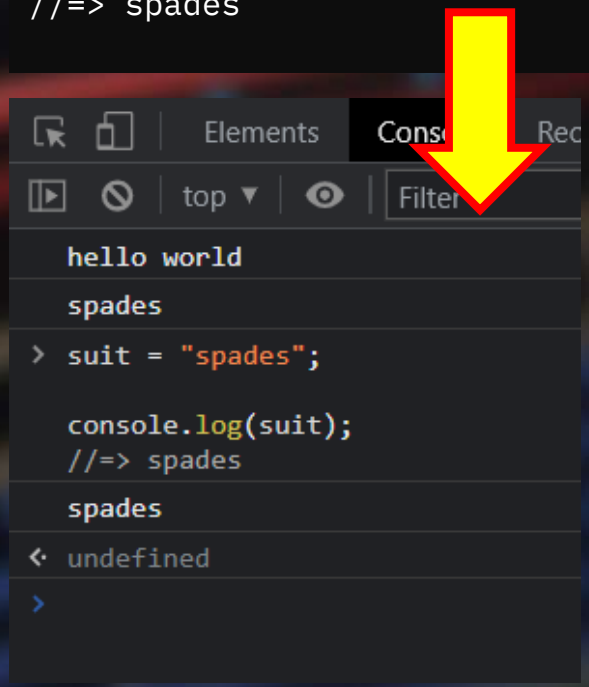
Defining variables

- Once we've declared a variable, the next step is define it; that is, to assign it a value. To do that, we'll use the assignment operator "=".
- An assignment statement takes the value on the right-hand side of the = symbol, and stores it in the variable on the left-hand side. If we refer to the variable suit in our program after we assign it a value, we'll get its associated value.
- If we print the variable, we'll see its associated value.

JavaScript

```
suit = "spades";

console.log(suit);
//=> spades
```



Declaring + defining variables

- It's sometimes easier to give a variable a value at the same time as we declare it. We can combine the process in a single line.
- For example, if we were writing a program to post to Twitter, we might create a variable to store the tweet that we want to post.

JavaScript

```
// declare and define a variable called tweet
var tweet = "oh how i love twitter";

console.log(tweet);
//=> oh how i love twitter
```

```
> // declare and define a variable called tweet
  var tweet = "oh how i love twitter";

  console.log(tweet);
  //=> oh how i love twitter

  oh how i love twitter
```

Manipulating values via variables

- Once we have some variables that store values, we can use those variables to construct new values.
- For example, we can create a variable that stores a value that represents a card.
- In the first code we've used the concatenation operator for strings "+" to construct a new string by gluing other strings together.
- Similarly, we may want to tweet at someone; look at the second code, as example.

JavaScript

```
// create a card by concatenating the suit and the
rank

var rank = ace;
var card = rank + " of " + suit;

console.log(card);
//=> ace of spades
```

JavaScript

```
var tweep = "@semmypurewal";
var comment = "hey your book sucks!"
var tweet = tweep + ": " + comment;
console.log(tweet);
//=> @semmypurewal: hey your book sucks!
```

Non-string variables

- Variables can store things other than strings.
- Notice that number values don't have quotes around them, but we can still concatenate them with strings to construct new strings.
- Unlike strings we can do mathematical stuff on numbers.
- You'll see that our burritoPrice and taxRate variable names have multiple words. In that situation, we format them using camel case which means we make the first letter of the first word lowercase, and the first letter of each subsequent word uppercase.

JavaScript

```
var age = 8;

var message = "You're only " + age + "? You
shouldn't be using Facebook.";
console.log(message);
//=> You're only 8? You shouldn't be using
Facebook.
```

JavaScript

```
var burritoPrice = 5.99;
var taxRate = 0.09;

var tax = burritoPrice * taxRate;
console.log(tax);
//=> 0.5391

var totalCost = burritoPrice + tax;
console.log(burritoPrice);
//=> 6.529100000000001
```

Reassigning variables

JavaScript

```
var twitterUser = "@twitter_user1";
var greeting = "hello, ";

var tweet = greeting + twitterUser;
console.log(tweet);
//=> hello, @twitter_user1

twitterUser = "@another_tweep";
tweet = greeting + twitterUser;
console.log(greeting);
//=> hello, @another_tweep
```

JavaScript

```
var count = 10;
console.log(count);
//=> 10

count = count + 1;
console.log(count);
//=> 11

count = count + 10;
console.log(count);
//=> 21
```

Functions

- We learned that programming is about storing and manipulating data. Variables are our main method of storing data; we're able to read their values and write new ones. To combine a number of these reads and writes into more complex operations, we can use functions.

- We want to calculate the tax we'd have to pay on a bunch of different kinds of burritos. We can do it by manually doing multiplications and sums, or by *defining* a single function to be *called* every time we need it.

JavaScript

```
var burritoPrice = 4.99;
var superBurritoPrice = 5.99;
var gigaBurritoPrice = 27.99;
var taxRate = 0.01;

// Without functions

var burritoTax =  burritoPrice * taxRate;
var burritoTotal = burritoPrice + burritoTax;

var superBurritoTax =  superBurritoPrice * taxRate;
var superBurritoTotal = superBurritoPrice +
superBurritoTax;

var gigaBurritoTax =  gigaBurritoPrice * taxRate;
var gigaBurritoTotal = gigaBurritoPrice +
gigaBurritoTax;

// With functions

var totalWithTax = function (rate, price) {
    var tax = price * rate;
    return price + tax;
};

var gigaBurritoTotal = totalWithTax(taxRate,
gigaBurritoPrice);

console.log(gigaBurritoTotal);
```

Functions

- We can define a function which builds an HTML paragraph element. The input to the function is the content string, and the output is the content surrounded by HTML paragraph tags.
- Notice that our taggedString variable is hidden from the outside world, and is only available for reading or writing inside the makeHtmlParagraph function. We call variables like this local variables.
- Our local variables must NOT have the same name as any of our function arguments, because if we accidentally use the same name, we could accidentally overwrite the value in one of our input variables, which can result in all kinds of nasty things.

JavaScript (HTML paragraphs)

```
// I define the function
var makeHtmlParagraph = function (content) {
    var taggedString = "<p>" + content + "</p>";
    return taggedString;
};

// I call the function, in defining a variable
var paragraph = makeHtmlParagraph("hello world!");

// Then I plot it
console.log(paragraph);
//=> <p>hello world!</p>
```

JavaScript (variable scope)

```
var add = function (x, y) {
    var x = 100;  // we lose the argument x
    console.log(x);
    return x + y;
}

var result = add (10, 20);
//=> 100
console.log(result);
//=> 120//=> <p>hello world!</p>
```

Types

- Programmers often write functions in JavaScript without considering what happens when the arguments contain values of unexpected types. This can lead to bugs in even simple JavaScript programs.

- If we mix and match the types, sometimes the results can be unexpected. The most basic way to test to see the type of a value in JavaScript is to use the `typeof` operator.

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	remainder

JavaScript

```
addThree(5, 2, 10);
//=> 17

addThree("Hello", " World", "!");
//=> Hello World!
```

JavaScript

```
addThree("Hello", 5, 10);
//=> Hello510

addThree(5, 10, "Hello");
//=> 15Hello

addThree("5", 10, "Hello");
//=> 510Hello
```

JavaScript

```
typeof addThree(5, 2, 10);
//=> "number"

typeof addThree(5, 10, "Hello");
//=> "string"
```

Extending the number operations with Math

- Beyond the basic arithmetic operations, we can do most other operations that a scientific calculator can do. The extended operators live inside the Math object as functions, and we access them using the dot operator.
- The random function generates numbers between 0 and 1. For generating a natural number between 0 and 9, you can use:

JavaScript

```
var rand = Math.random();
//=> sets rand to 0.475040664896369

rand * 10;
//=> 4.75040664896369

Math.floor(rand * 10);
//=> 4
```

JavaScript

```
Math.pow(2, 3);  // Power function
//=> 8

2/3  // Simple division
//=> 0.6666666666666666

var longDecimal = 3.1415926535897;

Math.round(longDecimal);
//=> 3, since it's less than 3.5

Math.floor(longDecimal);
//=> 3, since it's the biggest whole number below

Math.ceil(longDecimal);
//=> 4, since it's the smallest whole number above

Math.max(7, 2, 10, 5);
//=> 10

Math.random();  // Useful for MC simulations
//=> 0.23129316372796893
```

String Types and Built-In Methods

- Strings are a little more interesting than numbers, because they have built in operations called *methods* that generate new values (often strings) by applying functions to the current string.
- We access string methods using the dot operator.
- You can also check to see if a string contains another string by using the indexOf method.
- This method returns the index of the substring (starting at 0 for the first position), or -1 if the substring does not appear.

JavaScript

```
"Hello World!".toLowerCase();
//=> hello world!

"Hello World!".toUpperCase();
//=> HELLO WORLD!

var greeting = "hello there!";
greeting.toUpperCase();
//=> HELLO THERE!


var tweet = "LOL, this is my tweet on twitter";
tweet.indexOf("LOL");
//=> 0

tweet.indexOf("tweet");
//=> 16

tweet.indexOf("facebook");
//=> -1
```

String Types and Built-In Methods

- You can also grab a slice out of a string.
- You can get the length of the string, but the length property is not a method, so you shouldn't use the parentheses.
- On top of that, you can always chain method calls.
- We'll also occasionally want to extract individual characters from a string. We can do this by providing an index as the input to the charAt method.
- The charAt method is convenient because it also allows us to use a variable for the index, and we can always get the very last character, by combining charAt with the length property.

JavaScript

```
tweet.slice(0, 3);
//=> LOL

tweet.length
//=> 47

tweet.slice(tweet.indexOf("tweet", tweet.length));
//=> "tweet on twitter but not really"

tweet.slice(25,32).toUpperCase();
//=> TWITTER

tweet.charAt(6);
//=> h

// notice that the indices start at 0
tweet.charAt(0);
//=> L

var strValue = "hello world!";
// notice the index is the length minus 1, since
the indices start at 0
strValue.charAt(strValue.length - 1);
//=> !
```

Boolean Types and Boolean Expressions

- The last important basic type in JavaScript is the Boolean type. There are exactly two boolean values - `true` and `false`.
- Usually, a boolean value is the result of a boolean expression. The expressions can be built up using JavaScript's built in comparison operations.
- We can even use these comparison operators on strings. We need to be careful, because the ordering isn't always straightforward at first.

JavaScript

```
var isACard = true;
console.log(isACard);
//=> true

var isANumber = false;
console.log(isANumber);
//=> false

typeof isANumber;
//=> boolean

5 < 7;   // is less than
//=> true

5 <= 5;  // is less than or equal to
//=> true

"aardvark" < "zebra";
//=> true

"aardvark" < "Zebra";
//=> false, because upper-case letters come first
```

Boolean Types and Boolean Expressions

- There are 5 basic comparison operators which evaluate to booleans. We can use these operators to ask things about ordered types like numbers and strings.
- Once we have boolean values, or expressions that evaluate to boolean values, we can use several boolean operators to build up more complex expressions.
- The && operator represents the logical "and". This returns true if both the expression on its left and expression on its right return true. Similarly, we can use the logical "or" operator, ||, and the the ! (not) operator, that you already should know.

JavaScript

```
"aardvark" !== "AArdvark".toLowerCase();
//=> false

var age = 25;
age > 0 && age < 18;
//=> false

var age = 12;
age > 0 && age < 18;
//=> true

var value = 5;
typeof value === "number" || typeof value === "string";
//=> true

var value = "hello";
typeof value === "number" || typeof value === "string";
//=> true

var value = true;
!(typeof value === "number" || typeof value ===
"string");
//=> true
```

Conditionals

- Boolean types give our programs the ability to change their behavior based on values computed while the program is running. The simplest way to do this is by using if statements.
- An if statement accepts a boolean expression and a block of code. It executes the code only if the boolean expression evaluates to true.
- If the value stored in age is bigger than 13, both logging statements are executed. Similarly, if the boolean expression evaluates to false the code block will be skipped over.

JavaScript

```
var age = 25;

if (age >= 13) {
    console.log("You can have a Facebook account!");
}
console.log("finished!");

//=> You can have a Facebook account!
//=> finished!

var age = 11;

if (age > 13) {
    console.log("You can have a Facebook account!");
}
console.log("finished!");

//=> finished!
```

Conditionals

- Let's try using an if statement in a function. Suppose we want to project the more irreverent side of our personality on Twitter by making sure that every single one of our tweets includes "lol". We can write a function using an if statement that guarantees that's always the case.
- One approach to writing this function would be to check if the input already contains "lol", and if it doesn't, to add it to the end.

JavaScript

```
var improveTweet = function (tweet) {
    var result = tweet;

    if (tweet.indexOf("lol") === -1 && tweet.indexOf("LOL") === -1) {
          result = result + " lol";
    }

    return result;
}

improveTweet("this tweet needs to be improved");
//=> this tweet needs to be improved lol

improveTweet("this tweet is already great lol");
//=> this tweet is already great lol

improveTweet("LOL, no need to do anything here");
//=> LOL, no need to do anything here

improveTweet("my car was stolen and i was fired from my job today");
//=> my car was stolen and i was fired from my job today lol
```

else clauses in if statements

- if statements allow you to include an else clause, which executes only if the boolean expression evaluates to false.
- Look at the first code. It is a simple application of the if-else clause.
- We can also use if-else statements to write more interesting functions. Suppose we want to write a function that accepts either an opening or closing HTML tag, and returns the tag name associated with it.

JavaScript

```
var heightInInches = 40;
var minHeightInFeet = 4;

if (heightInInches/12 >= minHeightInFeet) {
    console.log("You can ride Space Mountain!");
} else {
    console.log("Sorry, you're not allowed to ride Space Mountain.");
}

console.log("finished!");

//=> Sorry, you're not allowed to ride Space Mountain.
//=> finished!

var getTagName = function (tag) {
    var tagName;
    if (tag.charAt(1) === "/") {
        tagName = tag.slice(2, tag.length - 1);
    } else {
        tagName = tag.slice(1, tag.length - 1);
    }
    return tagName;
};

getTagName("<p>");
//=> p

getTagName("</article>");
//=> article
```

JavaScript

```
var improveTweet = function (tweet) {
    var random = Math.floor(Math.random() * 2); // generate either a 1 or a 0
    var result = tweet;
    var expression;

    if (random === 0) {
        expression = "lol";
    } else {
        expression = "omg";
    }

    if (result.indexOf(expression) === -1 && result.indexOf(expression.toUpperCase()) === -1) {
        result = result + " " + expression;
    }

    return result;
};

improveTweet("this is a normal tweet");
//=> this is a normal tweet lol

improveTweet("lol, that last one was funny");
// in this case, we generate 'omg' while the tweet already contains lol
//=> lol, that last one was funny omg

improveTweet("lol what happens when we now generate lol again?");
// in this case we generate lol, so it doesn't get added again
//=> lol what happens when we now generate lol again?
```

if-else if and nested if statements

- Sometimes we'll want our functions to behave differently depending on a set of conditions. For example, suppose we wanted to write a function that would translate a given hour of the day into a user-friendly greeting.
- We can write code that does this by cascading if-else statements.
- There's a single return statement at the end. This is a common pattern that you'll see again and again.

JavaScript

```
var greetingByHour = function (hour) {
    var result;  // we'll define this variable below

    if (0 <= hour && hour <= 5) {
        result = "Wow, it's early!";
    } else if (5 < hour && hour <= 12) {
        result = "Good Morning!";
    } else if (12 < hour && hour <= 17) {
        result = "Good Afternoon!";
    } else if (17 < hour && hour <= 20) {
        result = "Good Evening!";
    } else if (20 < hour && hour <= 24) {
        result = "Shouldn't you be in bed?";
    } else {
        result = "Oh gosh, this is awkward -- that's not a time.";
    }

    return result;
}

// 6 in the morning
greetingByHour(6);
//=> Good Morning!

// 6 in the evening
greetingByHour(18);
//=> Good Evening!
```

if-else if and nested if statements

- Although we'll try to keep things mostly simple, it's worth noting that if statements and if-else statements can contain nested if and if-else statements.
- For example, here's one way to find the maximum of 3 numbers.

JavaScript

```
var maxOfThree = function (numA, numB, numC) {
    var result;

    if (numA >= numB) {
        if (numA >= numC) {
            result = numA;
        } else {
            result = numC;
        }
    } else {
        if (numB >= numC) {
            result = numB;
        } else {
            result = numC;
        }
    }

    return result;
} //=> Good Evening!
```

if-else if and nested if statements

- While nesting if statements is pretty normal and you'll find it in a lot of code, these situations can very often be simplified. Let's look at a couple of techniques for simplifying nested if statements.
- First, we could use compound boolean expressions.
- The flat structure here is much easier to read than the first example, but there is a much better technique we can use for this case.

JavaScript

```
var maxOfThree = function (numA, numB, numC) {
    var result;

    if (numA >= numB && numA >= numC) {
        result = numA;
    } else if (numB >= numA && numB >= numC) {
        result = numB;
    } else {
        result = numC;
    }

    return result;
}
```

What if we created a maxOfTwo function and then called that?

JavaScript

```
var maxOfThree = function (numA, numB, numC) {
    var bigger = maxOfTwo(numA, numB);
    var biggest = maxOfTwo(bigger, numC);

    return biggest;
}
```

Writing robust functions

- One thing that if statements allow us to do is to issue errors in cases where inputs to a function don't meet our expectations.
- JavaScript has an operator called throw which allows us to terminate the program to let the programmer (or a user) know that it has entered an unexpected state.

JavaScript

```
var addThree = function (a, b, c) {
    if (!isNumber(a) || !isNumber(b) || !isNumber(c)) {
        throw "all of the arguments to addThree must be numbers!"
    }
    return a + b + c;
}

addThree(1, 2, 3);
//=> 6

addThree(1, 2, "hello");
//=> the arguments to addThree must be numbers!

addThree("hello", 1, 2);
//=> the arguments to addThree must be numbers!
```

While loops as generalizations of if statements

- An if statement accepts a condition and a block of code to execute if the condition is true.
- A while loop extends this idea in that it continuously executes the block of code as long as the condition stays true.
- This can easily lead to infinite loops as in the previous example. When that happens in the Chrome developer tools, our only hope is to close the tab and start over. We'll need to be careful!
- Although infinite loops are great for crashing your web browser, we generally want to modify the value that the condition is testing.

JavaScript

```
while (num % 2 === 0) {
    console.log("num is even!");
}
//=> if num is even, this will run forever!!

var num = 150;

while (num % 13 !== 0) {
    console.log(num + " is not divisible by 13.");

    // keep adding one to num
    num = num + 1;
}
console.log("the first number bigger than 150 that
is divisible by 13 is " + num);
```

for-loops

- Most of the looping we will end up doing looks like the pattern we already saw.

1. The initialization condition, executed at first, before the loop starts;
2. The update condition, executed every time the loop body ends;
3. The continuation condition, checked prior to executing the loop body;
4. The loop body, executed every time the continuation condition is found to be true.

- It's common enough that JavaScript provides a different kind of loop called a for loop which makes this structure more obvious.

- Here is our last example, using a for loop.

JavaScript

```
while (num % 2 === 0) {
    console.log("num is even!");
}
//=> if num is even, this will run forever!!

var num = 1;  // Initialization condition

while (num <= 10) {  // Continuation condition
    if (num % 2 === 0) {
        console.log(num + " is even!");
    } else {
        console.log(num + " is odd!");
    }
    num = num + 1;  // Update condition
}
//=> 1 is odd!
//=> 2 is even!
//=> 3 is odd!
//=> 4 is even!
//=> 5 is odd!
//=> 6 is even!
//=> 7 is odd!
//=> 8 is even!
//=> 9 is odd!
//=> 10 is even!
```

Calculating properties using a loop

- Suppose we want to know the largest divisor of a number.
- We know that 1 evenly divides every number, so we can start with the assumption that 1 is the largest divisor.
- Next, we can start at 2 and work our way up to the number itself.
- If we ever see a new divisor, we know it will be larger than the one we've already seen, so we can replace the current largest.

JavaScript

```
var largestDivisor = function (num) {
    if (typeof num !== "number" || num <= 0) {
        throw "largestDivisor requires num to be a
positive number!";
    }

    var largestDivisorSoFar = 1;
    var divisor;

    for (divisor = 2; divisor < num; divisor = divisor +
1) {
        if (divisor % num === 0) {
            largestDivisorSoFar = divisor;
        }
    }

    return largestDivisorSoFar;
}
```

Transforming strings with loops

- Since we've seen that we can easily find the length of a string with the length property and we can access individual characters with the charAt method, we can pretty easily use a for loop to iterate over all the characters in a string.
- We can use this approach to perform interesting operations on strings.
- We can use this approach to perform interesting operations on strings. For example, suppose we wanted to remove all vowels from a string.

JavaScript

```
var message = "hello world!";
var index;

for (index = 0; index < message.length; index = index +
1) {
    console.log(message.charAt(index));
}

//=> h
//=> e
//=> l
//=> l
//=> o
//=>
//=> w
//=> o
//=> r
//=> l
//=> d
//=> !
```

Transforming strings with loops

JavaScript

```
var removeVowels = function (message) {
    if (typeof message !== "string") {
        throw "the input must be a string!";
    }

    // start off with the empty string
    var result = "";
    var index;

    for (index = 0; index < message.length; index = index + 1) {
        // if it's not a vowel, concatentate it to the result
        if (!isVowel(message.charAt(index))) {
            result = result + message.charAt(index);
        }
    }

    return result;
}

removeVowels("hello world!");
//=> hll wrld!

removeVowels("aeiou");
//=>
```

Breaking out of a loop

- Suppose we wanted to find the first lower-case letter in a string and then return it. We might start by writing a helper function to let us know if a character is a lower-case letter.
- Disaster! It's actually returning the last lowerCaseLetter! We need some way to stop the loop as soon as we find a lower case letter, then return it.
- However, before we fix our function, we have another problem. What should happen when there are no lower-case letters in the string? We could throw an error, but it's probably not an error condition.

JavaScript

```
var firstLowerCaseLetter = function (message) {
    var result;
    var index;

    for (index = 0; index < message; index = index + 1) {
        if (isLowerCaseLetter(message.charAt(index))) {
            result = message.charAt(index);
        }
    }

    return result;
}

firstLowerCaseLetter("Hello World!");
//=> d

firstLowerCaseLetter("This is a tweet.");
//=> t
```

Breaking out of a loop

- We could use a special value to denote the absence of a lower-case letter. That's the approach the indexOf function takes - it returns -1 in the case the substring is not found.
- This approach will work pretty well for us in this case. Let's return the empty string, "", when there are no lower-case letters.

JavaScript

```
var firstLowerCaseLetter = function (message) {
    var index;
    // initialize result to the default value
    // in case we don't find anything
    var result = "";

    for (index = 0; index < message; index = index + 1) {
        if (isLowerCaseLetter(message.charAt(index))) {
            result = message.charAt(index);
        }
    }

    return result;
}
```

Breaking out of a loop

- Now how do we stop when we find the first lower-case letter?
- One approach is to use the continuation condition to check to see if the result variable has changed.
- This will cause the continuation condition to return false once we set the result variable to a lower-case letter.
- It works pretty well, but it makes our condition more difficult to read.

JavaScript

```
var firstLowerCaseLetter = function (message) {
    var index;
    // initialize result to the default value in
    // case we don't find anything
    var result = "";

    // add a check to see if result is still "" before
continuing
    for (index = 0; index < message && result === "";
index = index + 1) {
        if (isLowerCaseLetter(message.charAt(index))) {
            result = message.charAt(index);
        }
    }

    return result;
}
```

Iterating backwards

- When we loop over numbers, there's no reason we have to count forwards! It turns out that it's sometimes more convenient to move backwards through a set of numbers or the characters in a string.
- For example, suppose we wanted to list all the numbers between 1 and 5 in reverse.
- We'd start at the end, and then subtract 1 from the counter value.

JavaScript

```
function countDownFrom = function (num) {
    if (typeof num !== "number" || num < 1) {
        throw "the input should be a positive
number!!!");
    }

    var count;

    for(count = num; count > 0; count = count - 1) {
        console.log(count);
    }
}

countDownFrom(5);
//=> 5
//=> 4
//=> 3
//=> 2
//=> 1
```

Arrays

- Arrays are the most basic compound data type in JavaScript. They allow you to associate a list of values with a single variable name.
- We can declare an array variable in the same way that we declare any other variable, but when we define its value, we use square brackets and separate its individual elements with commas.
- Once we've created an array, we can access its individual elements using the square-bracket operator and an index, which is simply the distance the element is from the beginning of the array.

JavaScript

```
var greetings = [ "hola", "aloha", "hello", "bonjour",
"hallo" ];

var primes = [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ];

primes[3];
//=> 7

primes[1]
//=> 3

var french = greetings[3];
console.log(french + "!");
//=> bonjour!

primes[-1]
//=> undefined

greetings[15]
//=> undefined
```

Mutating an array

- We can add elements to an array after we've created it. We can also change the values in an array after we've created it, just like we change other variables. To start out, let's create a new empty array by using the empty square brackets.
- We can add elements to the end of the array by calling its push method, and change the values contained in an existing array by using the assignment operator.
- Just because you can do something doesn't mean you should! In general, we want to avoid mutating arrays unless absolutely necessary.

JavaScript

```
var suits = [];
console.log(suits);
//=> []

suits.push("clubs");
console.log(suits);
//=> ["clubs"]

suits.push("diamonds");
console.log(suits);
//=> ["clubs", "diamonds"]

suits.push("hearts");
suits.push("spades");
//=> ["clubs", "diamonds", "hearts", "spades"]

// replace "hearts" with "coins"
suits[2] = "coins";
//=> ["clubs", "diamonds", "coins", "spades"]
```

Operating with arrays

- One thing that's really nice about arrays is that we can use variables to index into them.
- If this discussion about arrays seems familiar, it's because we've seen something like them before. You can think of arrays as a generalization of strings, where instead of being a sequence of characters like a string, it's a sequence of arbitrary values. In fact, JavaScript is flexible enough to allow us to access the elements of a string in the same way we access elements of arrays.
- You can think of an array's push method as the + operator on a string with adding a single character. But there's a much more general way of summing two arrays: the concat method.

JavaScript

```
var secondIndex = 1;

suits[secondIndex];
//=> "diamonds«

suits[secondIndex + 1];
//=> "coins«

var greeting = "hello!";

greeting[0];
//=> h

greeting[greeting.length - 1];
//=> !

greeting[greeting.length - 2];
//=> o

var firstArray = ["hello", "world"];
var secondArray = ["goodbye", "world"];
firstArray.concat(secondArray);
//=> ["hello", "world", "goodbye", "world"];
```

Operating with arrays

- In general, the array methods are different from the string methods whenever it makes sense.
- For example, the toUpperCase method wouldn't make any sense for an array, while several of the array methods we'll learn in this chapter wouldn't make sense for a string.
- That said, some methods and properties are shared.
- For example, the length property is the same for both arrays and strings. And both the indexOf and slice methods exists on an array.
- They work exactly like you'd expect.

JavaScript

```
"hello".length;
//=> 5

["this", "is", "an", "array"].length
//=> 4

var places = [ "first", "second", "third", "fourth",
"fifth" ]

places.indexOf("third");
//=> 2

places.indexOf("sixth");
//=> -1

places.slice(1, 3);
//=> [ "second", "third" ]
```

JavaScript (loops)

```
var secondElementOf = function (arr) {
   return arr[1];
}

var printEachElement = function (list) {
    var index;

    for (index = 0; index < list.length; index = index
+ 1) {
        console.log(list[index]);
    }
}
```

JavaScript (summing arrays)

```
var sumAnArray = function (listOfNumbers) {
    var sum = 0;
    var index;

    for (index = 0; index < listOfNumbers.length;
index = index + 1) {
        sum = sum + listOfNumbers[index];
    }

    return sum;
}
```

JavaScript (typeof ?)

```
typeof [1, 2, 3, 4];
//=> object

typeof [];
//=> object

Array.isArray([1, 2, 3, 4])
//=> true



Array.isArray(3);
//=> false
```

JavaScript (finding the smallest)

```
var smallestNumber = function (listOfNumbers) {
    var smallestSoFar = listOfNumbers[0];
    var index;

    for (index = 1; index < listOfNumbers.length;
index = index + 1) {
        if (listOfNumbers[index] < smallestSoFar) {
            smallestSoFar = listOfNumbers[index];
        }
    }

    return smallestSoFar;
}
```

Array iterators

- We learned about arrays, and we saw how we can use for-loops to iterate over them. Since this is a pretty common thing to do, JavaScript's provided us some easier ways to do it.
- JavaScript arrays have a rich set of built in methods that make iterating over them and calculating a properties a real pleasure. Most modern JavaScript developers prefer these to the traditional for-loops because they're easier to use and less error prone.
- The simplest way we can sum a list of numbers is to use the array's forEach method. The forEach method actually takes a function as an argument and applies it to each element in order.

JavaScript

```
Lorem ipsum
var printElements = function (listOfNumbers) {
    var printElement = function (number) {
        console.log(number);
    }

    listOfNumbers.forEach(printElement);
}

printElements([5,6,7,8]);
//=> 5
//=> 6
//=> 7
//=> 8

var sum = function (listOfNumbers) {
    var sum = 0;

    listOfNumbers.forEach(function (number) {
        sum = sum + number;
    });

    return sum;
}
```

map

- Let's look at another common operation, taking a list of values and transforming each value to produce another list.
- Suppose we wanted to take an array of numbers and produce a new array of numbers that are the previous numbers doubled.
- This would work fine, but there's an even better way! The pattern of constructing a new array by applying a function to every element occurs so frequently, we have a function called map that does exactly that. In fact, it allows us to remove the result variable altogether!

JavaScript

```
var numbers = [1,2,3,4,5,6];

// Without map
var doubleNumbers = function (numbers) {
    var result = []; // create an empty array

    numbers.forEach(function (number) {
        result.push(number * 2);
    })

    return result;
}

var doubles = doubleNumbers(numbers);
console.log(doubles);
//=> [2,4,6,8,10,12]

// Using map
var doubles = function (nums) {
    return nums.map(function (num) {
        return num * 2;
    });
}
console.log(doubles);
//=> [2,4,6,8,10,12]
```

map

- That's all map does - it returns a new array that is the same length as the old array, with the specified function applied to each element!
- Let's suppose we want to create an array that is simply the first letters of an array of strings. This would be a perfect candidate for the map function.
- Similarly, we can map a list of boolean values to their opposites.

JavaScript

```
// Array that is the first letters of an array of strings
["hi", "everyone", "loves", "lists", "of",
"words"].map(function (word) {
    return word[0];
});
//=> ["h", "e", "l", "l", "o", "w"]

// Map a list of Boolean values to their opposites
[true, false, true, true, false, true].map(function (val)
{
    return !val;
});
//=> [false, true, false, false, true, false]
```

Chaining functions that return arrays

- Since map returns an array, we can immediately chain a call to forEach. This means we effectively call the next function on the returned array.
- In the previous problem section, we created a range function that generated an array with numbers from the given range. Since that function returns an array, we can chain that as well.
- Chaining functions with the range function gives us a nice approach to learning about the other features of JavaScript arrays.

JavaScript

```
// Example 1
var numbers = [1,2,3,4];

numbers.map(function (number) {
    return number * 2;
}).forEach(function (number) {
    console.log(number);
});
//=> 2
//=> 4
//=> 6
//=> 8

// Example 2
range(1,4).map(function (number) {
    return number * 2;
}).forEach(function (number) {
    console.log(number);
});
//=> 2
//=> 4
//=> 6
//=> 8
```

filter

- The filter method allows us to create a new array which only includes the elements of the previous array that pass some basic boolean test.
- For example, suppose we wanted all of the even numbers in an array.
- Combining this with the range function, we can print out all the even numbers less than 100 in a pretty interesting way.

JavaScript

```
// Only print even numbers from my set
var nums = [ 5, 10, 15, 20, 25, 30, 35, 36, 37, 38, 39,
40 ];

nums.filter(function (elt) {
    return elt % 2 === 0;
});
//=> [ 10, 20, 30, 36, 38, 40 ]

// Print all the even numbers less than 100
range(0, 100).filter(function (elt) {
    return elt % 2 === 0;
}).forEach(function (elt) {
    console.lot(elt);
});
```

some and every

- Suppose we were trying to do some very basic analysis of the sentiment of a set of tweets, and wanted to know if any of the tweets in the list contain the word "awesome".
- Using techniques found in the previous section and a forEach loop, we can create a function that does something like this…
- …or we could use the filter method and check the length property of the resulting array.
- This is a nice solution in that it removes the need for extraneous variables. But it has one significant disadvantage.

JavaScript

```
var containsAwesome = function (tweets) {
    var result = false;

    tweets.forEach(function (tweet) {
        if (tweet.toLowerCase().indexOf("awesome") > -1)
{
            result = true;
        }
    });

    return result;
};

// With filter
var containsAwesome = function (tweets) {
    return tweets.filter(function (tweet) {
        return tweet.toLowerCase().indexOf("awesome") > -
1;
    }).length > 0;
};
```

some and every

- If you think back, we learned how to break out of a for-loop early when we need to. There's no need to process the entire array if we've found what we ware looking for. Unfortunately, the forEach, filter, and map methods have no way to break out of a loop early.
- Fortunately JavaScript has two functions that do have this property. The some method is a good example. It returns true if any of the elements pass the true/false test, and it stops.
- Similarly to some, the every method returns true if all of the elements pass the test, and it breaks out of the loop early if any of the elements evaluate to false.

JavaScript

```
// some
var containsAwesome = function (tweets) {
    return tweets.some(function (tweet) {
        console.log("testing: " + tweet);
        return tweet.toLowerCase().indexOf("awesome") > -
1;
    });
};

containsAwesome([ "sad tweet", "awesome tweet",
"unprocessed tweet", "another tweet" ]);
//=> testing: sad tweet
//=> testing: awesome tweet
//=> true

// every
var allAwesome = function (tweets) {
    return tweets.every(function (tweet) {
        return tweet.toLowerCase().indexOf("awesome") > -
1;
    });
};
```

reduce

- What if we have to compute something more complex than just a true or false?
- Consider of summing a list of numbers contained in an array. Is there some way we can leverage built-in array methods so we can remove even more extra variables?
- It turns out we can - the reduce method allows us to build up a very general computation by carrying an additional value between calls. Let's start by considering our solution to the sum problem using the forEach method.

JavaScript

```
// with sumSoFar
var sum = function (listOfNumbers) {
    var sumSoFar = 0;

    listOfNumbers.forEach(function (number) {
        sumSoFar = sumSoFar + number;
    });

    return sumSoFar;
}

// without sumSoFar
var numbers = [5,6,7,8,9,10];

numbers.reduce(function (sumSoFar, number) {
    return sumSoFar + number;
});
//=> 45

// sum function with no local variables
var sum = function (listOfNumbers) {
    return listOfNumbers.reduce(function (sumSoFar,
number) {
        return sumSoFar + number;
    });
};
```

reduce

- There are times when it doesn't make sense to make the first argument to reduce's function be the first element in the array.
- For example, suppose we wanted to write a function that accepts an array of strings and returns them combined into a paragraph as sentences.
- Assuming we have a function called capitalize that capitalizes the first word in a sentence, we can easily achieve this using a forEach loop and a temporary local variable.
- You'll see this pattern a lot in the wild.

JavaScript

```
// function with capitalize
var paragraphify = function (list) {
    var result = "";  // initialize to empty list

    list.forEach(function (sentence) {
        result = result + capitalize(sentence) + ".";  //
add a space and a period
    });

    return result;
};

// function with reduce - very used
var paragraphify = function (list) {
    return list.reduce(function (paragraph, sentence) {
        return result + capitalize(sentence) + ".";
    }, "");  // <= the second argument is ""
};

// function we want
paragraphify( [ "hello world", " this is a tweet,
goodbye" ] );
//=> Hello world. This is a tweet. Goodbye.
```

Converting between strings and arrays

- We mentioned previously that arrays and strings are very similar, but that strings don't enjoy some of the useful array methods we've been exploring.
- We can use a string's split method to turn it into an actual array of characters.
- The split method is pretty flexible. For example, suppose we were dealing with a string containing comma separated values (CSVs).
- We can even just split on spaces.

JavaScript

```
var greeting = "hello";

greeting.split("");
//=> ["h","e","l","l","o"]
```

JavaScript

```
var values = "gracie,loki,dahlia,ally";  //=> these are my dogs!
var names = values.split(",");
//=> [ "gracie", "loki", "dahlia", "ally" ]
```

JavaScript

```
var tweet = "this is a tweet!";
var words = tweet.split(" ");
//=> [ "this", "is", "a", "tweet!" ];
```

Objects

- Sometimes it's useful to have data stored in structures that are indexed by strings. In JavaScript, these are called objects.
- Notice that defining one is similar to defining an array, but we use curly-braces instead of square-brackets. In addition we have to specify the key and the value for each entry. We do that by separating them with colons.
- As is usually the case, there's nothing special about the identifier we use for the variable name. We can create multiple cards with different variables.

JavaScript

```
var card = { "rank": "ace", "suit": "spades" };

var person = { "name": "Semmy", "age": 37 };

var anotherCard = { "rank": "two", "suit": "clubs" };

var anotherPerson = { "name": "Jennifer", "age": 25 };

var greetings = {
    "spanish" : "hola",
    "hawaiian": "aloha",
    "english" : "hello",
    "french"  : "bonjour",
    "german"  : "hallo"
};
```

Accessing elements of an object

- Like arrays, we can use the square-bracket operator and a key to access the data stored in an object.
- The advantage here over arrays should be clear.

JavaScript

```
person["name"];
//=> "Semmy"

card["suit"];
//=> "spades“

// here greetings is an object, and the semantics of the
code is clear
greetings["spanish"];
//=> "hola"

greetings["german"];
//=> "hallo"

// here greetings is an array, and things aren't as clear
greetings[0];
//=> "hola"

greetings[4];
//=> "hallo“
```

Accessing elements of an object

- For example, when we use the greetings object, the code becomes more readable since we know the language we're referring to (instead of having to use an arbitrary index).
- The trade-off is that there is no concrete ordering of the values in an object, unlike an array where there is a clear sense of the order that the data is stored. Ordering matters for some sets of data but not others, so whether this is good or bad depends on the types you are storing.
- Like an array, if we try to access an element that doesn't exist we get the special JavaScript value undefined as a result.

JavaScript

```
// here greetings is an object
greetings["spanish"];   //=> "hola"

// here greetings is an array, and things aren't as clear
greetings[0];           //=> "hola"

greetings["swahili"];   //=> undefined

person.name;            //=> Semmy

greetings.french;       //=> bonjour

var list = {
    "1": "first",
    "2": "second"
};

list["2"];              //=> second

list.2;
//=> Syntax Error: Unexpected Numbervar list = {
    "1": "first",
    "2": "second"
};

list["2"];              //=> second

list.2;                 //=> Syntax Error: Unexpected Number
```

Mutating an object

- For example, when we use the greetings object, the code becomes more readable since we know the language we're referring to (instead of having to use an arbitrary index).
- The trade-off is that there is no concrete ordering of the values in an object, unlike an array where there is a clear sense of the order that the data is stored. Ordering matters for some sets of data but not others, so whether this is good or bad depends on the types you are storing.
- Like an array, if we try to access an element that doesn't exist we get the special JavaScript value undefined as a result.

JavaScript

```
// here greetings is an object
greetings["spanish"];   //=> "hola"

// here greetings is an array, and things aren't as clear
greetings[0];           //=> "hola"

greetings["swahili"];   //=> undefined

person.name;            //=> Semmy

greetings.french;       //=> bonjour

var list = {
    "1": "first",
    "2": "second"
};

list["2"];              //=> second

list.2;
//=> Syntax Error: Unexpected Numbervar list = {
    "1": "first",
    "2": "second"
};

list["2"];              //=> second

list.2;                 //=> Syntax Error: Unexpected Number
```

Mutating an object

- Like arrays, we can mutate objects by setting the values with an assignment.
- We can also add new keys and values by using assignments.
- You can use expressions or variables to index into an array.
- You can also use expressions and variables to extract values from objects.

JavaScript

```
person.name;            //=> "Semmy"

person.name = "Heather";

person.name;            //=> Heather

person.age;             //=> 37

person["hometown"] = "Boston";

person["hometown"];     //=> "Boston“

var array = [ "hello", "world", "this", "is", "a", "list" ];

var lastIndex = array.length - 1;

array[lastIndex];       //=> "list“

var card = { "rank":"ace", "suit":"spades" };
var key = "rank";

card[key];
//=> "ace" var card = { "rank":"ace", "suit":"spades" };
var key = "rank";

card[key];              //=> "ace"
```

Using array functions on objects

- Since objects contain both keys and values, it's not immediately obvious how something like map would work.
- Should it map over the keys, or the values, or both? Should it return an object or an array?
- You can easily extract the set of keys and the collection of values as arrays and then operate on them if you wish.
- To get the keys, you can use the Object.keys function.
- How can you get the values? You can combine the map function with the Object.keys function!

JavaScript

```
var card = { "rank":"ace", "suit":"spades" };
var keys = Object.keys(card);

keys;
//=> [ "rank", "suit" ]

keys[0];
//=> "rank“

var values = Object.keys(card).map(function (key) {
    return card[key];
});

values;
//=> [ "ace", "spades" ];

values.map(function (value) {
    return capitalize(value);
});
//=> [ "Ace", "Spades" ]
```

Arrays of objects

- An array of objects allows us to store a collection of data that might be more complex than simply numbers or strings.
- For example, we could represent a card hand using an array of card objects.
- Using the fact that we can build an array of objects, we can access our array functions with objects in a much clearer way. For example, suppose instead of having an object like card2.
- Then we can easily use our array functions!

JavaScript

```
var hand = [
    { "rank":"ace",   "suit":"spades" },
    { "rank":"five",  "suit":"clubs" },
    { "rank":"ten",   "suit":"diamonds" },
    { "rank":"queen", "suit":"clubs" },
    { "rank":"five",  "suit":"hearts" }
];

var card2 = { "suit": "spades", "rank": "two" };

var cardArray = [
    { "key" : "suit", "value": "spades" },
    { "key" : "rank", "value": "two" }
];

cardArray.map(function (element) {
    var key = element.key;
    var value = element.value;

    return capitalize(value);
});
//=> [ "Spades", "Two" ]
```

Nested objects

- In previous sections, we've been modeling tweets as strings that are less than 140 characters. It turns out that a tweet is a lot more than just a string - it also includes a lot of metadata including a timestamp, geographic information, and the number of times the tweet has been retweeted, among many other things. So if we get a real tweet, it might look something like the 1st example on the right.
- Tweet objects are complex; they even contain sub-objects. For instance, you can extract the user object from the tweet to get information.
- And you can assign these sub-objects to new variables.

JavaScript

```
// 1st example
// assume getTweet returns a tweet object
var tweet = getTweet();

tweet.text;
//=> "this is an awesome tweet!"

tweet.created_at;
//=> "Mon Oct 20 14:06:17 +0000 2014"

tweet.source;
//=> "<a href="http://twitter.com" rel="nofollow">Twitter Web
Client</a>“

// 2nd example
tweet.user.name;
//=> "Semmy Purewal"

tweet.user.screen_name;
//=> "semmypurewal“

// 3rd example
var user = tweet.user;

user.followers_count;
//=> 483
```

Nested objects

- Creating nested object literals isn't much different than creating normal object literals - it's just that the values are occasionally objects themselves. Look at the 1st example.
- Similarly, objects can contain nested arrays. For example, suppose we wanted to create a user object that had a list of tweets associated with that user. Look at the 2nd example.
- When you access the element that is an array, you can use all the normal array methods on it. Look at the 3rd example.

JavaScript

```
var tweet = {
    "text": " this is an awesome tweet!",
    "created_at": "Mon Oct 20 14:06:17 +0000 2014",
    "source": "<a href="http://twitter.com" rel="nofollow">Twitter
Web Client</a>",
    "user": {
        "name": "Semmy Purewal",
        "screen_name": "semmypurewal",
        "followers_count": 483
    }
};

var user = {
    "name": "Semmy Purewal",
    "screen_name": "semmypurewal",
    "tweets": [
        "this is a tweet.",
        "this is another tweet!"
    ]
};

user.tweets.forEach(function (tweet) {
    console.log(tweet);
};
//=> this is a tweet.
//=> this is another tweet!
```


Observe:
Engineers in their natural habitat.
Please do not tap on glass,
engineers are easily frightened by
outsiders.
STARECAT.COM

What is a Design Language?

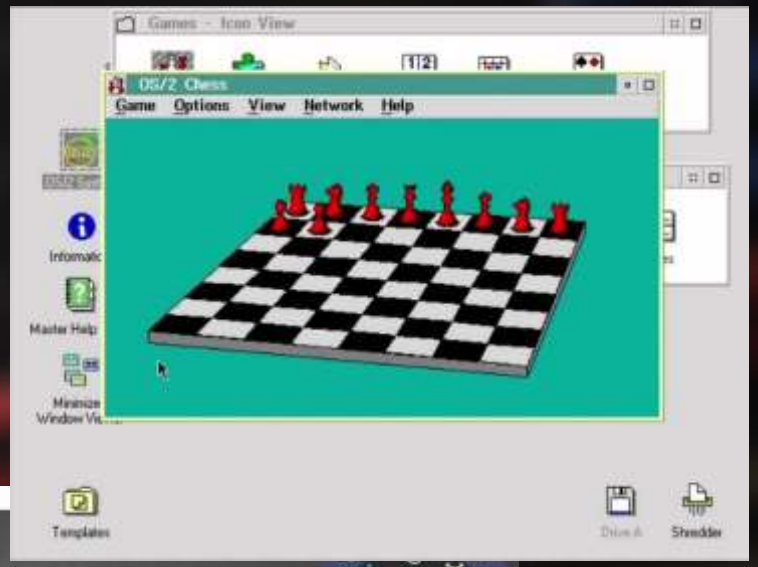
The definition of a Language (Dictionary.com):

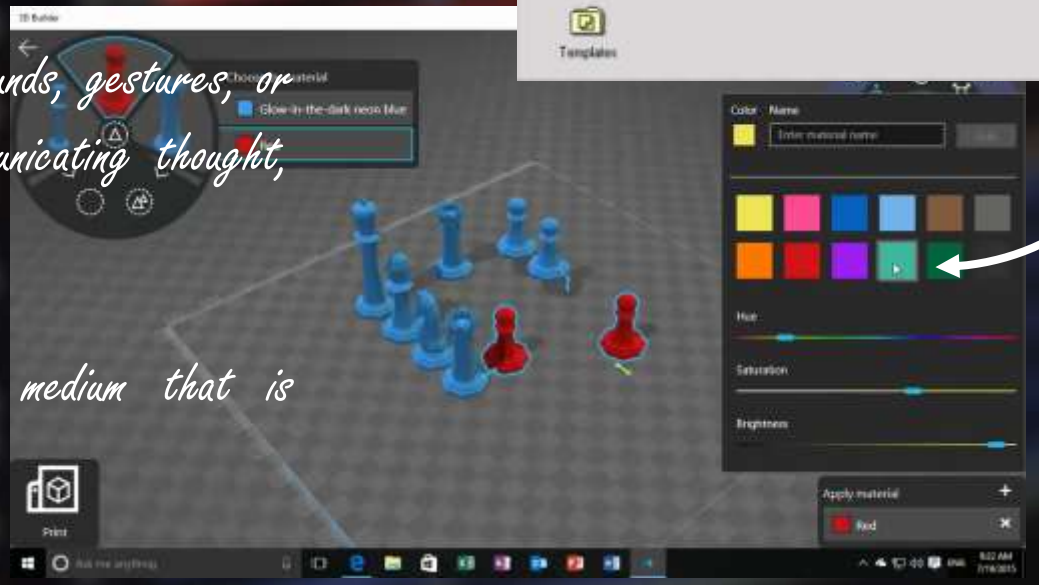
"[4.] any set or system of such symbols as used in a more or less uniform fashion by a number of people, who are thus enabled to communicate intelligibly with one another."

"[5.] any system of formalized symbols, signs, sounds, gestures, or the like used or conceived as a means of communicating thought, emotion, etc."

"[7.] communication of meaning in any way; medium that is expressive, significant, etc."

1989



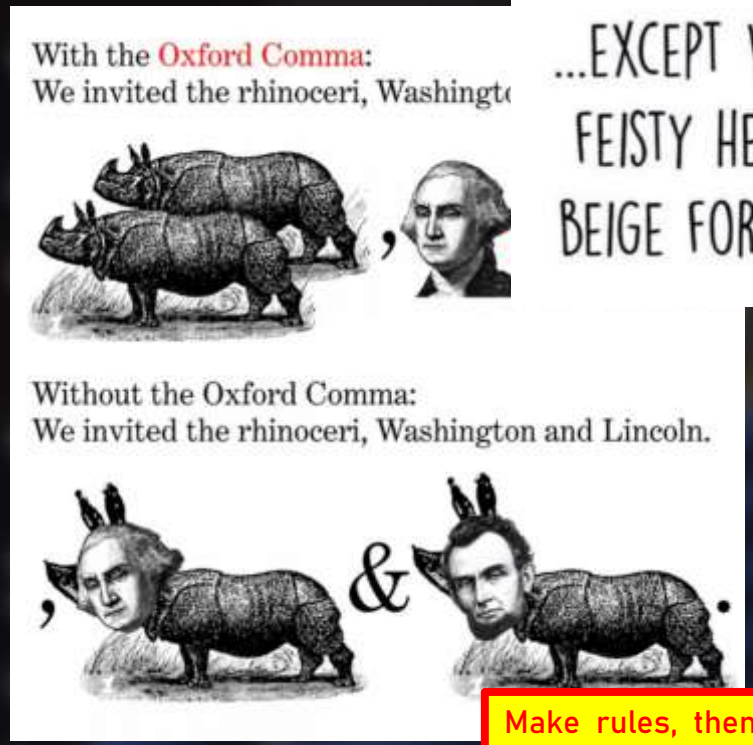


2015

Fundamentals of Design

- Words have meaning;
- Punctuation has meaning;
- Types of words (nouns, adjectives, and verbs) have meaning;
- Grouped words have meaning & give formal structure;
- Sentences have meaning & give formal structure;
- Paragraphs have meaning & give formal structure!

From here we define more complex, larger methods of communication above these, level such as how to structure scientific objectives, term papers, and a myriad of story structures.

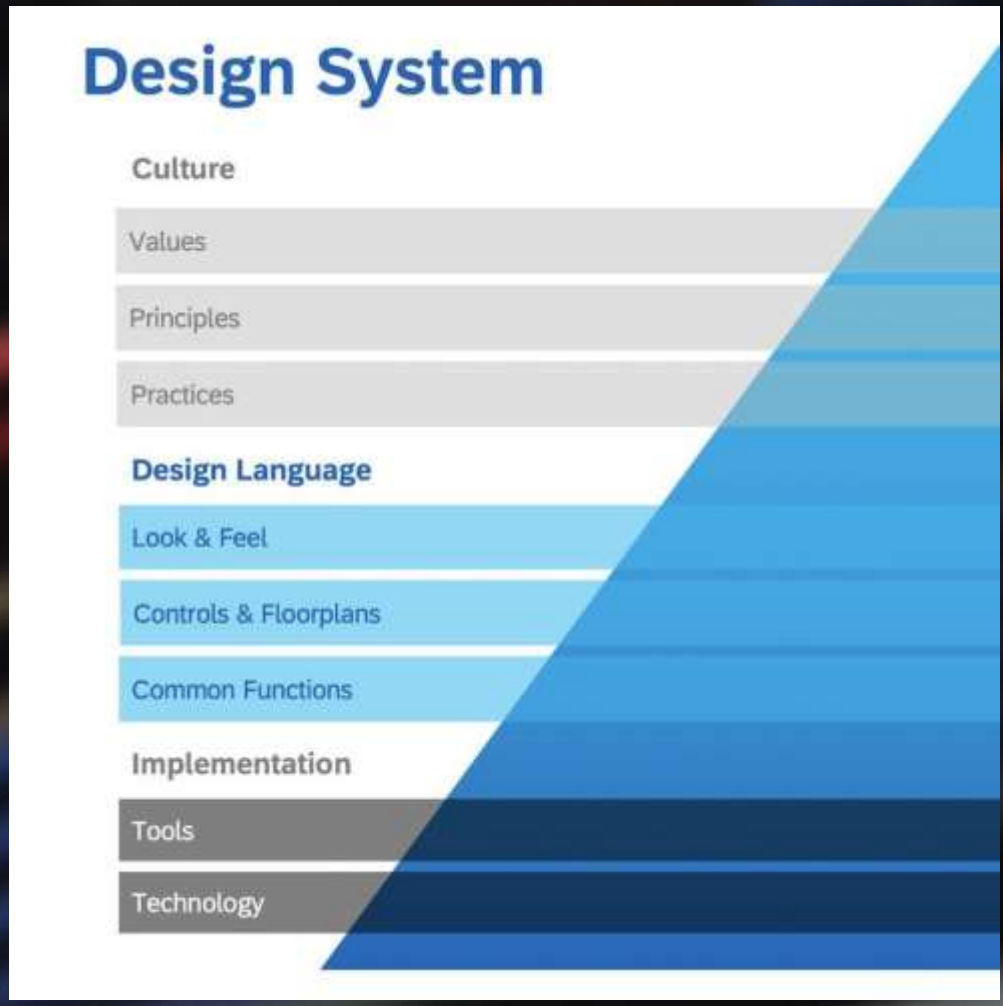


Make rules, then follow them. And make sure your rules are logical, not confusing.

How to Develop a Design Language

Usually, design languages are not rigorously defined, but still, the languages share some common elements, including:

- A family of UI components and patterns. These reusable building blocks can be used to create user interfaces.
- Style guides. Information about the visual properties of design (e.g., colors, fonts, and whitespace), as well as instructions on how to use them in a design.
- Documentation of semantics. This helps define the meaning and intent behind the elements of design (e.g., colors, shapes, and fonts), and of naming conventions for individual styles and components.

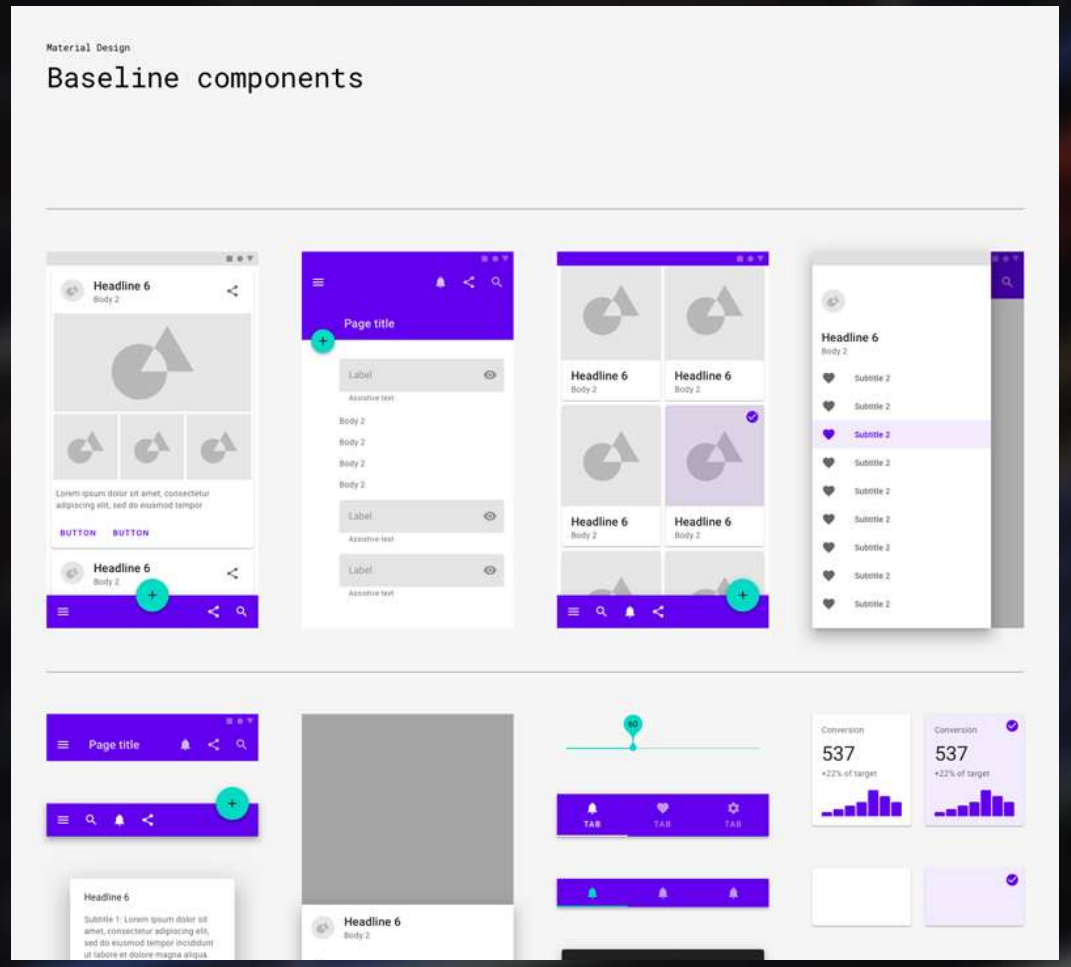


Why create a design language?

Consistency

Efficiency and cost

Brand identity and authenticity

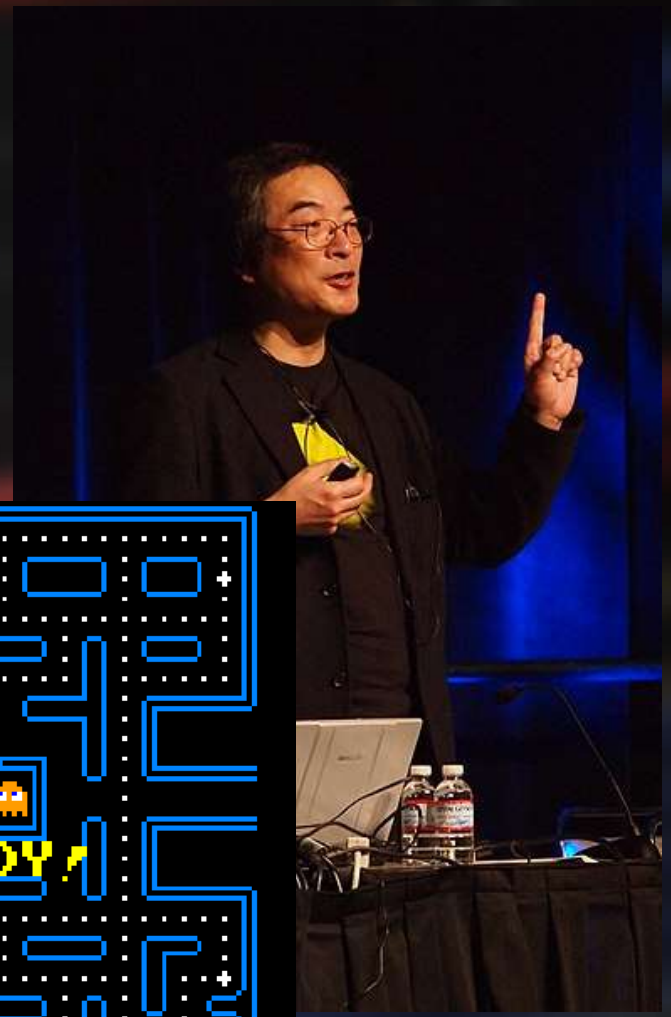


Some important (open source) design languages:

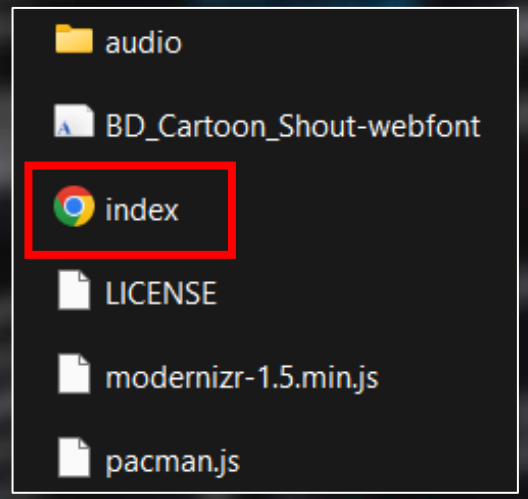
- Fluent Design System;
- Material Design.

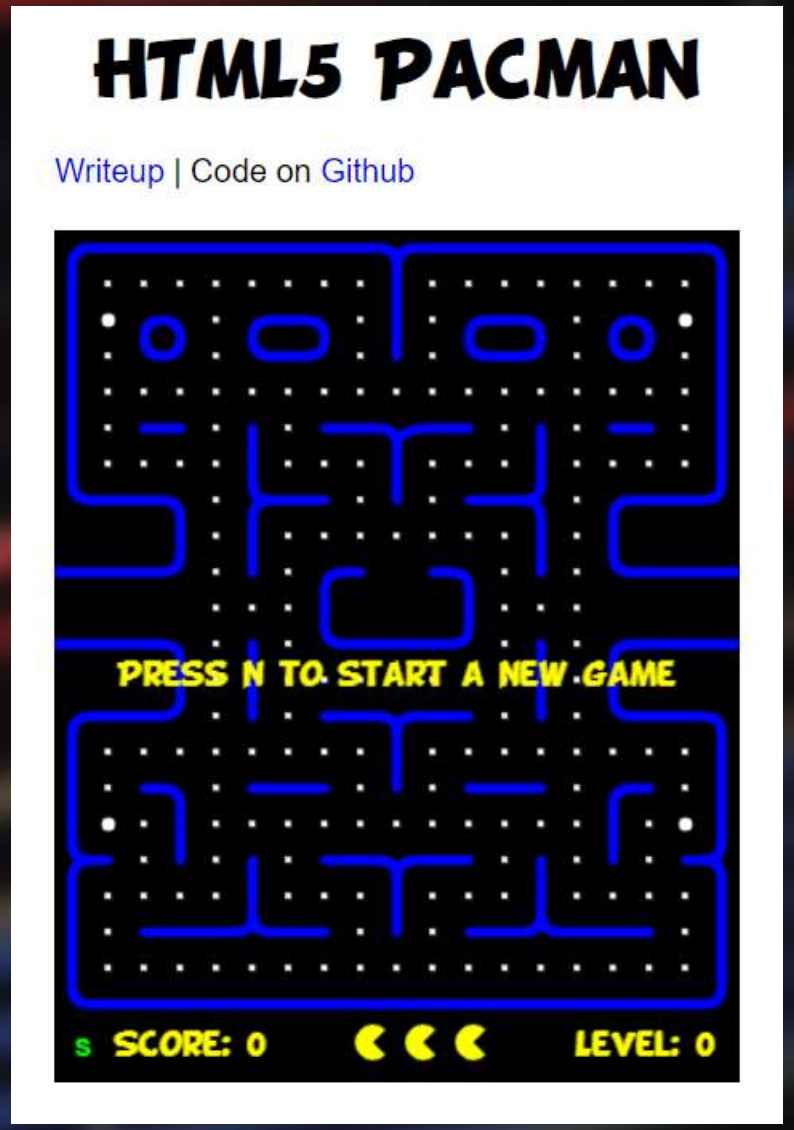
PAC-MAN

- Pac-Man, originally called Puck Man in Japan, is a 1980 maze action video game developed and released by Namco for arcades.
- Game development began in early 1979, directed by Toru Iwatani with a nine-man team. Iwatani wanted to create a game that could appeal to women as well as men, because most video games of the time had themes of war or sports.
- The in-game characters were made to be cute and colorful to appeal to younger players. The original Japanese title of Puck Man was derived from the titular character's hockey puck–like shape; the title was changed for the North American release to mitigate vandalism.



GitHub Link


audio
BD_Cartoon_Shout-webfont
index
LICENSE
modernizr-1.5.min.js
pacman.js


HTML5 PACMAN
Writeup | Code on Github
PRESS N TO START A NEW GAME
SCORE: 0
LEVEL: 0

You at the end of this lecture…

…with me

But I've still got some more examples…