JavaScript general theory

Book: O'Reilly - JavaScript. The Definitive Guide, 6th ed.

First, a note about $.

Strange but true, you can use "`$`" as a function name in JavaScript. It is shorthand for `jQuery()`. Which you can use if you want. jQuery can be ran in compatibility mode if another library is using the `$` already. Just use `jQuery.noConflict()`. `$` is pretty commonly used as a selector function in JS.

In jQuery the `$` function does much more than select things though.

- You can pass it a selector to get a collection of matching elements from the DOM.
- You can pass it a function to run when the document is ready (similar to `body.onload()` but better).
- You can pass it a string of HTML to turn into a DOM element which you can then inject into the document.
- You can pass it a DOM element or elements that you want to wrap with the jQuery object.

Here is the documentation.

Type, Values and Variables (chpt. 1-5)

- Variables hold reusable data in a program.
- JavaScript will throw an error if you try to reassign `const` variables.
- You can reassign variables that you create with the `let` keyword.
- Unset variables store the primitive data type `undefined`.
- Mathematical assignment operators make it easy to calculate a new value and assign it to the same variable.
- The `+` operator is used to interpolate (combine) multiple strings.
- In JavaScript ES6, backticks (') and `${}` are used to interpolate values into a string.

Variable Scope

- *Scope* is the idea in programming that some variables are accessible/inaccessible from other parts of the program.
- *Global Scope* refers to variables that are accessible to every part of the program.
- *Block Scope* refers to variables that are accessible only within the block they are defined.

The *scope* of a variable is the region of your program source code in which it is defined. A *global* variable has global scope; it is defined everywhere in your JavaScript code. On the other hand, variables declared within a function are defined only within the body of the function. They are *local* variables and have local scope. Function parameters also count as local variables and are defined only within the body of the function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable:

```
let scope = "global";          // Declare a global variable
function checkscope() {
    let scope = "local";       // Declare a local variable with the same name
    return scope;              // Return the local value, not the global one
}
checkscope()                   // => "local"
```

Dates and Time

```
let then = new Date(2010, 0, 1);  // The 1st day of the 1st month of 2010
let later = new Date(2010, 0, 1,  // Same day, at 5:10:30pm, local time
                     17, 10, 30);
let now = new Date();           // The current date and time
let elapsed = now - then;       // Date subtraction: interval in milliseconds
later.getFullYear()             // => 2010
later.getMonth()                // => 0: zero-based months
later.getDate()                 // => 1: one-based days
later.getDay()                  // => 5: day of week.  0 is Sunday 5 is Friday.
later.getHours()                // => 17: 5pm, local time
later.getUTCHours()            // hours in UTC time; depends on timezone
later.toString()               // => "Fri Jan 01 2010 17:10:30 GMT-0800 (PST)"
later.toUTCString()            // => "Sat, 02 Jan 2010 01:10:30 GMT"
later.toLocaleDateString()     // => "01/01/2010"
later.toLocaleTimeString()     // => "05:10:30 PM"
later.toISOString()            // => "2010-01-02T01:10:30.000Z"; ES5 only
```

Control Flow

- `if/else` statements make binary decisions and execute different code based on conditions.
- All conditions are evaluated to be truthy or falsy.
- We can add more conditional statements to `if/else` statements with `else if`.
- `switch` statements make complicated `if/else` statements easier to read and achieve the same result.
- The ternary operator (`?`) and a colon (`:`) allow us to refactor simple `if/else` statements.
- Comparison operators, including `<`, `>`, `<=`, and `>=` can compare two variables or values.
- After two values are compared, the conditional statement evaluates to `true` or `false`.
- The logical operator `&&` checks if both sides of a condition are truthy.
- The logical operator `||` checks if either side is truthy.
- The logical operator `!==` checks if the two sides are not equal.
- An exclamation mark (`!`) switches the truthiness / falsiness of the value of a variable.
- One equals symbol (`=`) is used to assign a value to a variable.
- Three equals symbols (`===`) are used to check if two variables are equal to each other.

Difference between == and ===

Using the `==` operator (Equality)

```
true == 1; //true, because 'true' is converted to 1 and then compared
"2" == 2;  //true, because "2" is converted to 2 and then compared
```

Using the `===` operator (Identity)

```
true === 1; //false
"2" === 2;  //false
```

This is because the **equality operator `==` does type coercion**, meaning that the interpreter implicitly tries to convert the values before comparing.

On the other hand, the **identity operator `===` does not do type coercion**, and thus does not convert the values when comparing.

General objects characteristics and Arrays (chpt. 6-7)

- Arrays are lists and are a way to store data in JavaScript.
- Arrays are created with brackets `[]`.
- Each item inside of an array is at a numbered position, starting at `0`.
- We can access one item in an array using its numbered position, with syntax like: `myArray[0]`.
- We can also change an item in an array using its numbered position, with syntax like `myArray[0] = "new string";`
- Arrays have a `length` property, which allows you to see how many items are in an array.
- Arrays have their own methods, including `.push()` and `.pop()`, which add and remove items from an array, respectively.
- Arrays have many other methods that perform different functions, such as `.slice()` and `.shift()`. You can read the documentation for any array method on the Mozilla Developer Network website.
- Variables that contain arrays can be declared with `let` or `const`. Even when declared with `const`, arrays are still mutable; they can be changed. However, a variable declared with `const` cannot be reassigned.

Esempio:

```
let groceryList = ['orange juice', 'bananas', 'coffee beans', 'brown rice', 'pasta', 'coconut oil',
'plantains'];

groceryList.shift();
groceryList.unshift('popcorn');
console.log(groceryList);

console.log(groceryList.slice(1, 4));
console.log(groceryList);
```

Output:

```
[ 'popcorn',
  'bananas',
  'coffee beans',
  'brown rice',
  'pasta',
  'coconut oil',
  'plantains' ]
[ 'bananas', 'coffee beans', 'brown rice' ]
[ 'popcorn',
  'bananas',
  'coffee beans',
  'brown rice',
  'pasta',
  'coconut oil',
  'plantains' ]
```

Prototypes

Every JavaScript object has a second JavaScript object (or `null`, but this is rare) associated with it. This second object is known as a prototype, and the first object inherits properties from the prototype.

All objects created by object literals have the same prototype object, and we can refer to this prototype object in JavaScript code as `Object.prototype`. Objects created using the `new` keyword and a constructor invocation use the value of the **prototype** property of the constructor function as their prototype. So the object created by `new Object()` inherits from `Object.prototype` just as the object created by `{}` does. Similarly, the object created by `new Array()` uses `array.prototype` as its prototype, and the object created by `new Date()` uses `Date.prototype` as its prototype.

All of the built-in constructors (and most user-defined constructors) have a prototype that inherits from `Object.prototype`. For example, `Date.prototype` inherits properties from `Object.prototype`, so a Date object created by `new Date()` inherits properties from both `Date.prototype` and `Object.prototype`. This linked series of prototype objects is known as a *prototype chain*.

Functions (chpt. 8)

- *Functions* are written to perform a task.
- Functions take data, perform a set of tasks on the data, and then return the result.
- We can define parameters to be used when calling the function.
- When calling a function, we can pass in *arguments*, which will set the function's parameters.
- We can use `return` to return the result of a function which allows us to call functions anywhere, even inside other functions.

Iterators

- `.forEach()` is used to execute the same code on every element in an array but does not change the array and returns `undefined`.
- `.map()` executes the same code on every element in an array and returns a new array with the updated elements.
- `.filter()` checks every element in an array to see if it meets certain criteria and returns a new array with the elements that return truthy for the criteria.
- All iterator methods can be written using arrow function syntax. In fact, given the succinctness and the implicit return of arrow function syntax, this is quickly becoming the preferred way to write these types of method calls.
- You can visit the Mozilla Developer Network. to learn more about iterator methods (and all other parts of JavaScript!).

- Additional iterator methods such as `.some()`, `.every()`, `.reduce()` perform different functions.

Esempio:

```
let cities = ['Nashville', 'Charlotte', 'Asheville', 'Austin', 'Boulder'];

let nums = [1, 50, 75, 200, 350, 525, 1000];

//  Choose a method that will return undefined
cities.forEach(city => console.log('Have you visited ' + city + '?'));

// Choose a method that will return a new array
let longCities = cities.filter(city => city.length > 7);

// Choose a method that will return a new array
let smallerNums = nums.map(num => num - 5);

// Choose a method that will return a boolean value
nums.every(num => num < 0);
```

Output:

```
Have you visited Nashville?
Have you visited Charlotte?
Have you visited Asheville?
Have you visited Austin?
Have you visited Boulder?
```

Objects

- Objects store key-value pairs and let us represent real-world things in JavaScript.
- Properties in objects are separated by commas. Key-value pairs are always separated by a colon.
- You can add or edit a property within an object with dot notation.
- A method is a function in an object.
- `this` helps us with scope inside of object methods. `this` is a dynamic variable that can change depending on the object that is calling the method.
- **Getter** and **setter** methods allow you to process data before accessing or setting property values.

Esempio:

```
let person = {
  _name: 'Lu Xun',
  _age: 137,

  set age(newAge) {
    if (typeof newAge === 'number') {
      this._age = newAge;
      console.log(`${newAge} is valid input.`)
    } else {
      console.log(`${newAge} is not a valid input.`)
      return 'Invalid input';
    }
  },
  get age() {
    console.log(`${this._name} is ${this._age} years old.`)
    return this._age;
  }
};

//person.age = 'Thirty-nine';
//person.age = 39;
console.log(person.age);
```

Output:

```
Lu Xun is 137 years old.
137
```

Classes

- *Classes* are templates for objects.
- Javascript calls a *constructor* method when we create a new instance of a class.
- *Inheritance* is when we create a parent class with properties and methods that we can extend to child classes.
- We use the `extends` keyword to create a subclass.
- The `super` keyword calls the `constructor()` of a parent class.
- Static methods are called on the class, but not on instances of the class.

Esempio:

```
class HospitalEmployee {
  constructor(name) {
    this._name = name;
    this._remainingVacationDays = 20;
  }

  get name() {
    return this._name;
  }

  get remainingVacationDays() {
    return this._remainingVacationDays;
  }

  takeVacationDays(daysOff) {
    this._remainingVacationDays -= daysOff;
  }

  static generatePassword() {
    const randomPassword = Math.floor(Math.random()*10000);
    return randomPassword;
  }
}

class Nurse extends HospitalEmployee {
  constructor(name, certifications) {
    super(name);
    this._certifications = certifications;
  }

  get certifications() {
    return this._certifications;
  }

  addCertification(newCertification) {
    this.certifications.push(newCertification);
  }
}

const nurseOlynyk = new Nurse('Olynyk', ['Trauma','Pediatrics']);
nurseOlynyk.takeVacationDays(5);
console.log(nurseOlynyk.remainingVacationDays);
nurseOlynyk.addCertification('Genetics');
console.log(nurseOlynyk.certifications);
console.log(HospitalEmployee.generatePassword());
```

Output:

```
15
[ 'Trauma', 'Pediatrics', 'Genetics' ]
3289
```

Browser Compatibility and Transpilation

- ES5 — The old JavaScript version that is supported by all modern web browsers.
- ES6 — The new(er) JavaScript version that is *not* supported by all modern web browsers. The syntax is more readable, similar to other programming languages, and addresses the source of common bugs in ES5.
- caniuse.com — a website you can use to look up HTML, CSS, and JavaScript browser compatibility information.
- Babel — A JavaScript package that transpiles JavaScript ES6+ code to ES5.
- `npm init` — A terminal command that creates a **package.json** file.
- **package.json** — A file that contains information about a JavaScript project.
- `npm install` — A command that installs Node packages.
- `babel-cli` — A Node package that contains command line tools for Babel.
- `babel-preset-env` — A Node package that contains ES6+ to ES5 syntax mapping information.
- **.babelrc** — A file that specifies the version of the JavaScript source code.
- "`build`" script — A package.json script that you use to tranpsile ES6+ code to ES5.
- `npm run build` — A command that runs the build script and transpiles ES6+ code to ES5.

For future reference, here is a list of the steps needed to set up a project for transpilation: 1. Initialize your project using `npm init` and create a directory called src 2. Install babel dependencies by running 3. `npm install babel-cli -D` `npm install babel-preset-env -D`

4. Create a **.babelrc** file inside your project and add the following code inside it:
5. `{`
6. `"presets": ["env"] }`
7. Add the following script to your scripts object in package.json: `"build": "babel src -d lib"`
8. Run `npm run build` whenever you want to transpile your code from your src to lib directories.

Intermediate JavaScript Modules

- *Modules* in JavaScript are reusable pieces of code that can be exported from one program and imported for use in another program.
- `module.exports_exports` the module for use in another program.
- `require()` imports the module for use in the current program. ES6 introduced a more flexible, easier syntax to export modules:
- default exports `use export default` to export JavaScript objects, functions, and primitive data types.
- named exports use the `export` keyword to export data in variables.
- named exports can be aliased with the `as` keyword.
- `import` is a keyword that imports any object, function, or data type.

Esempio: *airplane.js*

```
export let availableAirplanes = [
{name: 'AeroJet',
 fuelCapacity: 800,
 availableStaff: ['pilots', 'flightAttendants', 'engineers', 'medicalAssistance', 'sensorOperators'],
 maxSpeed: 1200,
 minSpeed: 300
},
{name: 'SkyJet',
 fuelCapacity: 500,
 availableStaff: ['pilots', 'flightAttendants'],
 maxSpeed: 800,
 minSpeed: 200
}
];

export let flightRequirements = {
  requiredStaff: 4,
  requiredSpeedRange: 700
};

export function meetsStaffRequirements(availableStaff, requiredStaff) {
  if (availableStaff.length >= requiredStaff) {
    return true;
  } else {
    return false;
  }
};

export function meetsSpeedRangeRequirements(maxSpeed, minSpeed, requiredSpeedRange) {
  let range = maxSpeed - minSpeed;
  if (range > requiredSpeedRange) {
    return true;
    } else {
    return false;
  }
};

export default meetsSpeedRangeRequirements;
```

missionControl.js

```
import { availableAirplanes, flightRequirements, meetsStaffRequirements} from './airplane';

import meetsSpeedRangeRequirements from './airplane';

function displayFuelCapacity() {
  availableAirplanes.forEach(function(element) {
    console.log('Fuel Capacity of ' + element.name + ': ' + element['fuelCapacity']);
  });
}

displayFuelCapacity();

function displayStaffStatus() {
  availableAirplanes.forEach(function(element) {
   console.log(element.name + ' meets staff requirements: ' + meetsStaffRequirements(element.availableStaff, f
  });
}

displayStaffStatus();

function displaySpeedRangeStatus() {
  availableAirplanes.forEach(function(element) {
   console.log(element.name + ' meets speed range requirements:' + meetsSpeedRangeRequirements(element.maxSpee
  });
}

displaySpeedRangeStatus();
```

Output:

```
Fuel Capacity of AeroJet: 800
Fuel Capacity of SkyJet: 500
AeroJet meets staff requirements: true
SkyJet meets staff requirements: false
```

```
AeroJet meets speed range requirements:true
SkyJet meets speed range requirements:false
```

Requests I

1. JavaScript is the language of the web because of its asynchronous capabilities. A set of tools that are used together to take advantage of JavaScript's asynchronous capabilities is called AJAX, which stands for Asynchronous JavaScript and XML.
2. There are four HTTP request methods, two of which are `GET` and `POST`.
3. `GET` requests only request information from other sources.
4. `POST` methods can introduce new information to other sources in addition to requesting it.
5. `GET` requests can be written using an `XMLHttpRequest` object and vanilla JavaScript.
6. `POST` requests can also be written using an `XMLHttpRequest` object and vanilla JavaScript.
7. Writing `GET` and `POST` requests with `XHR` objects and vanilla JavaScript requires constructing the `XHR` object using `new`, setting the `responseType`, creating a function that will handle the response object, and opening and sending the request.
8. Much of the boilerplate used to write `GET` and `POST` requests with `XHR` and vanilla JavaScript can be reduced by using the `$.ajax()` method from the jQuery library.
9. jQuery provides other helper methods that can further reduce boilerplate such as `$.get()`, `$.post()`, and `$.getJSON()`.
10. Determining how to correctly write the requests and how to properly implement them requires carefully reading the documentation of the API with which you're working.

// XMLHttpRequest GET

```
const xhr = new XMLHttpRequest(); // creates new object
const url = 'http://api-to-call.com/endpoint;

xhr.responseType = 'json';
xhr.onreadystatechange = function () {
  if (xhr.readyState === XMLHttpRequest.DONE) {
    // Code to execute with response
  }
}; // this block handles response

xhr.open('GET', url); // opens request
xhr.send(); // sends object
```

// XMLHttpRequest POST

```
const xhr = new XMLHttpRequest(); // creates new object
const url = 'http://api-to-call.com/endpoint;
const data = JSON.stringify({id: '200'}); // converts data to a string

xhr.responseType = 'json';
xhr.onreadystatechange = function () {
  if (xhr.readyState === XMLHttpRequest.DONE) {
    // Code to execute with response
  }
}; // this block handles response

xhr.open('POST', url); // opens request
xhr.send(data); // sends object
```

// jQuery GET

```
$.ajax({ // settings
  const url = 'http://api-to-call.com/endpoint,
  type: 'GET',
  dataType: 'json',
  success(response) { // handles response if successful
```

```
    // Code to excecute with response on success
  },
  error(jqXHR, status, errorThrown) { // handles response if unsuccessful
    // Code to excecute with response on failure
  }
});
```

// jQuery POST

```
$.ajax({ // settings
  const url = 'http://api-to-call.com/endpoint,
  type: 'POST',
  data: JSON.stringify({id: '200'}),
  dataType: 'json',
  success(response) { // handles response if successful
    // Code to excecute with response on success
  },
  error(jqXHR, status, errorThrown) { // handles response if unsuccessful
    // Code to excecute with response on failure
  }
});
```

Requests II

1. `GET` and `POST` requests can be created a variety of ways.
2. We can asynchronously request data from APIs using AJAX. `fetch()` and `async/await` are new technologies developed in ES6 and ES7 respectively.
3. Promises are a new type of JavaScript object that represent data that will eventually be returned from a request.
4. `fetch()` is a web API that can be used to create requests. `fetch()` will return Promises.
5. We can chain `.then()` methods to handle Promises returned by fetch.
6. The `.json()` method converts a returned Promise to a JSON object.
7. `async` is a keyword that is used to create functions that will return Promises.
8. `await` is a keyword that is used to tell a program to continue moving through the message queue while a Promise resolves.
9. `await` can only be used within functions declared with `async`.

// fetch GET

```
fetch('http://api-to-call.com/endpoint').then(response => { // sends request
  if (response.ok) { // converts response object to JSON
    return response.json();
  }
  throw new Error('Request Failed!'); // handles errors
}, networkError => console.log(networkError.message)
).then(jsonResponse => { // handles success
  // Code to excecute with jsonResponse
});
```

// fetch POST

```
fetch('http://api-to-call.com/endpoint', {
  method: 'POST',
  body: JSON.stringify({id: '200'})
}).then(response => {
    if (response.ok) { // converts response object to JSON
      return response.json();
  }
  throw new Error('Request Failed!'); // handles errors
}, networkError => console.log(networkError.message)
).then(jsonResponse => { // handles success
  // Code to excecute with jsonResponse
});
```

// async await GET

```
async function getData() {
  try {
    let response = await fetch('http://api-to-call.com/endpoint'); // sends request
    if (response.ok) { // handles response if successful
      let jsonResponse = await response.json();
      // Code to excecute with jsonResponse
    }
    throw new Error('Request Failed!') // handles response if unsuccessful
  } catch (error) {
    console.log(error);
  }
}
```

// async await POST

```
async function getData() {
  try {
    let response = await fetch('http://api-to-call.com/endpoint', {
      method: 'POST',
      body: JSON.stringify({id: '200'})
    }); // sends request
    if (response.ok) { // handles response if successful
      let jsonResponse = await response.json();
      // Code to excecute with jsonResponse
    }
    throw new Error('Request Failed!') // handles response if unsuccessful
  } catch (error) {
    console.log(error);
  }
}
```

Scripting documents: Document Object Model (DOM) (chpt. 15)

Every Window object has a document property that refers to a Document object. The Document object represents the content of the window, and it is the subject of this chapter. The Document object does not stand alone, however. It is the central object in a larger API, known as the Document Object Model, or DOM, for representing and manipulating document content.

The Document Object Model, or DOM, is the fundamental API for representing and manipulating the content of HTML and XML documents.

Consider the following simple HTML document:

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
</html>
```

The DOM representation of this document is the tree at page 11 (Fig. 1):

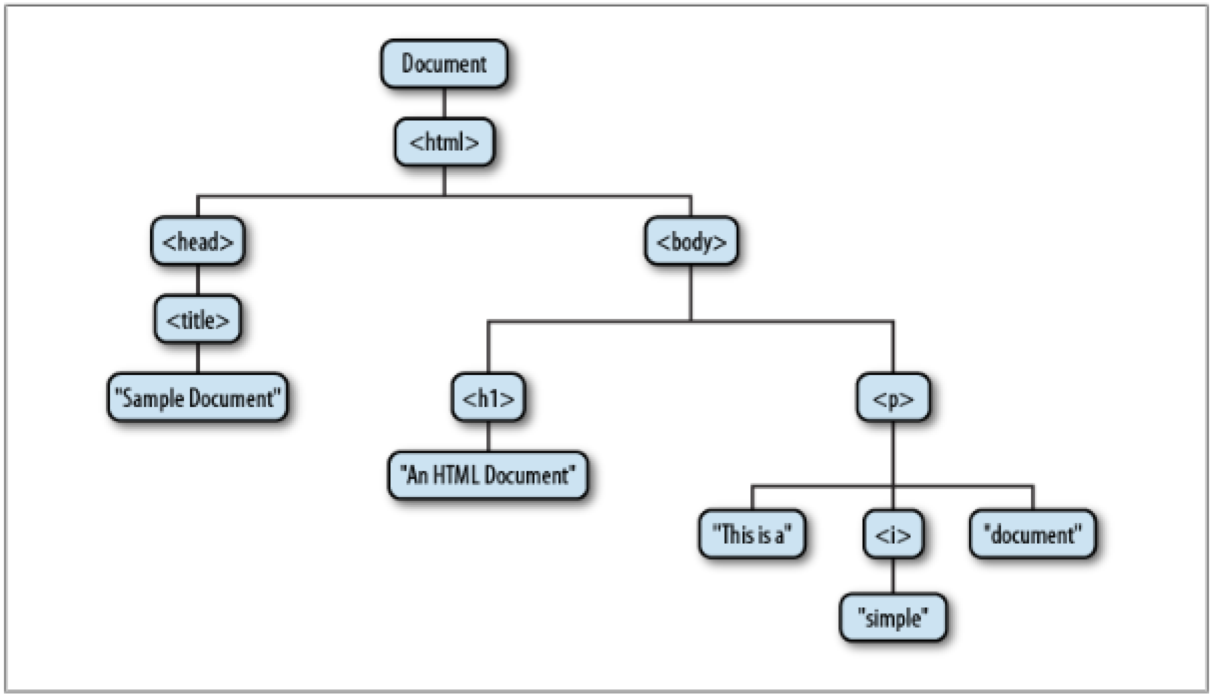


Figure 1: DOM three

CSS (chpt. 16)

Some of the properties of a CSS file:

Property	Description	Values / Examples
position	Specifies the type of positioning applied to an element	static, absolute, fixed, relative. `position: absolute;`
top, left	Specify the position of the top and left edges of an element	`top: 10px; left: 10px;`
bottom, right	Specify the position of the bottom and right edges of an element	`width: 10px; height: 10px;`
width, height	Specify the size of an element	`background-color: blue`
z-index	Specifies the “stacking order” of an element relative to any overlapping elements; defines a third dimension of element positioning	the element with the highest z-index appears on top of all the others. `z-index: -1;`
display	Specifies how and whether an element is displayed	none, inline, block, inline-block. `display: inline;`
visibility	Specifies whether an element is visible	visible, hidden. `visibility: hidden;`
clip	Defines a “clipping region” for an element; only portions of the element within this region are displayed	`clip: rect(0px,60px,200px,0px);`
overflow	Specifies what to do if an element is bigger than the space allotted for it	scroll, hidden, auto, visible. `overflow: scroll;`
margin, border, padding	Specify spacing and borders for an element	margin-top/right/bottom/left. border-width/style/color. padding-top/right/bottom/left. `margin: 10px 5px 15px 20px; border: 4px solid blue;`.
background	Specifies the background color or image of an element.	The color can be written as name color, HEX, rgb, rgba: `background-color: rgba (255, 255, 255, 0.5);` or `background-image: url("paper.gif");`

Property	Description	Values / Examples
opacity	Specifies how opaque (or translucent) an element is. This is a CSS3 property, supported by some browsers. A working alternative exists for IE.	0-1. `opacity: 0.5;`

Events (chpt. 17)

An *event handler* or *event listener* is a function that handles or responds to an event.

An event object is an object that is associated with a particular event and contains details about that event. Event objects are passed as an argument to the event handler function.

All event objects have a **type** property that specifies the event type and a **target** property that specifies the event target.

Each event type defines a set of properties for its associated event object. The object associated with a mouse event includes the coordinates of the mouse pointer, for example, and the object associated with a keyboard event contains details about the key that was pressed and the modifier keys that were held down.

Esempio:

```
let mouse = {
  x: undefined,
  y: undefined
}

window.addEventListener('mousemove', function (event) {
  mouse.x = event.x;
  mouse.y = event.y;
  console.log(mouse);
});

window.addEventListener('resize', function() {
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;
  init();
});
```

Un altro esempio:

```
<button id="mybutton">Click me</button>
```

```
let b = document.getElementById("mybutton");
b.onclick = function() {
  alert("Thanks for clicking me!");
};
b.addEventListener("click", function() {
  alert("Thanks again!");
}, false);
```

Main mouse events: *click, contextmenu, dblclick, mousedown, mouseup, mousemove, mouseover, mouseout, mouseenter, mouseleave.*

Scripted Media and Graphics (chpt. 21)

The ability to dynamically generate sophisticated graphics in the browser is important for several reasons: * The code used to produce graphics on the client side is typically much smaller than the images themselves, creating a substantial bandwidth savings. * Dynamically generating graphics from real-time data uses a lot of CPU cycles. Off-loading this task to the client reduces the load on the server, potentially saving on hardware costs. * Generating graphics on the client is consistent with modern web application architecture in which servers provide data and clients manage the presentation of that data.

The following HTML fragment is a simple example: it creates an image that changes when the mouse moves over it:

```
<img src="images/help.gif"
    onmouseover="this.src='images/help_rollover.gif'"
    onmouseout="this.src='images/help.gif'">
```

SVG: Scalable Vector Graphics

SVG is an XML grammar for graphics. The word “vector” in its name indicates that it is fundamentally different from raster image formats, such as GIF, JPEG, and PNG, that specify a matrix of pixel values. Instead, an SVG “image” is a precise, resolution-independent (hence “scalable”) description of the steps necessary to draw the desired graphic.

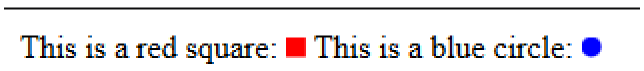
Here is what a simple SVG file looks like:

```
<!-- Begin an SVG figure and declare our namespace -->
<svg xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 1000 1000">  <!-- Coordinate system for figure -->
  <defs>                       <!-- Set up some definitions we'll use -->
    <linearGradient id="fade"> <!-- a color gradient named "fade" -->
      <stop offset="0%" stop-color="#008"/>     <!-- Start a dark blue -->
      <stop offset="100%" stop-color="#ccf"/>  <!-- Fade to light blue -->
    </linearGradient>
  </defs>
  <!-- Draw a rectangle with a thick black border and fill it with the fade -->
  <rect x="100" y="200" width="800" height="600"
     stroke="black" stroke-width="25" fill="url(#fade)"/>
</svg>
```

This piece of code:

```
<!DOCTYPE html>
<html>
<body>
This is a red square: <svg width="10" height="10">
  <rect x="0" y="0" width="10" height="10" fill="red"/>
</svg>
This is a blue circle: <svg width="10" height="10">
  <circle cx="5" cy="5" r="5" fill="blue"/>
</svg>
</body>
</html>
```

Gives this output:



This is a red square: 🟥 This is a blue circle: ●

Figure 2: SVG example

This is how we creare an analog clock displayed as a SVG:

```
<!DOCTYPE HTML>
<html>
<head>
<title>Analog Clock</title>
<script>
function updateTime() { // Update the SVG clock graphic to show current time
    var now = new Date();                       // Current time
    var min = now.getMinutes();                 // Minutes
    var hour = (now.getHours() % 12) + min/60;  // Fractional hours
    var minangle = min*6;                       // 6 degrees per minute
    var hourangle = hour*30;                    // 30 degrees per hour
    // Get SVG elements for the hands of the clock
    var minhand = document.getElementById("minutehand");
    var hourhand = document.getElementById("hourhand");
    // Set an SVG attribute on them to move them around the clock face
    minhand.setAttribute("transform", "rotate(" + minangle + ",50,50)");
    hourhand.setAttribute("transform", "rotate(" + hourangle + ",50,50)");
    // Update the clock again in 1 minute
    setTimeout(updateTime, 60000);
}
</script>
<style>
/* These CSS styles all apply to the SVG elements defined below */
#clock {                          /* styles for everything in the clock */
   stroke: black;                 /* black lines */
   stroke-linecap: round;         /* with rounded ends */
   fill: #eef;                    /* on a light blue gray background */
}
#face { stroke-width: 3px;}       /* clock face outline */
#ticks { stroke-width: 2; }       /* lines that mark each hour */
#hourhand {stroke-width: 5px;}    /* wide hour hand */
#minutehand {stroke-width: 3px;}  /* narrow minute hand */
#numbers {                        /* how to draw the numbers */
    font-family: sans-serif; font-size: 7pt; font-weight: bold;
    text-anchor: middle; stroke: none; fill: black;
}
</style>
</head>
<body onload="updateTime()">
  <!-- viewBox is coordinate system, width and height are on-screen size -->
  <svg id="clock" viewBox="0 0 100 100" width="150" height="150">
    <defs>   <!-- Define a filter for drop-shadows -->
     <filter id="shadow" x="-50%" y="-50%" width="200%" height="200%">
        <feGaussianBlur in="SourceAlpha" stdDeviation="1" result="blur" />
        <feOffset in="blur" dx="1" dy="1" result="shadow" />
        <feMerge>
          <feMergeNode in="SourceGraphic"/><feMergeNode in="shadow"/>
        </feMerge>
      </filter>
    </defs>
    <circle id="face" cx="50" cy="50" r="45"/>  <!-- the clock face -->
    <g id="ticks">                              <!-- 12 hour tick marks -->
      <line x1='50' y1='5.000' x2='50.00' y2='10.00'/>
      <line x1='72.50' y1='11.03' x2='70.00' y2='15.36'/>
      <line x1='88.97' y1='27.50' x2='84.64' y2='30.00'/>
      <line x1='95.00' y1='50.00' x2='90.00' y2='50.00'/>
      <line x1='88.97' y1='72.50' x2='84.64' y2='70.00'/>
      <line x1='72.50' y1='88.97' x2='70.00' y2='84.64'/>
      <line x1='50.00' y1='95.00' x2='50.00' y2='90.00'/>
      <line x1='27.50' y1='88.97' x2='30.00' y2='84.64'/>
      <line x1='11.03' y1='72.50' x2='15.36' y2='70.00'/>
      <line x1='5.000' y1='50.00' x2='10.00' y2='50.00'/>
      <line x1='11.03' y1='27.50' x2='15.36' y2='30.00'/>
      <line x1='27.50' y1='11.03' x2='30.00' y2='15.36'/>
    </g>
    <g id="numbers">                     <!-- Number the cardinal directions-->
      <text x="50" y="18">12</text><text x="85" y="53">3</text>
      <text x="50" y="88">6</text><text x="15" y="53">9</text>
    </g>
    <!-- Draw hands pointing straight up. We rotate them in the code. -->
    <g id="hands" filter="url(#shadow)"> <!-- Add shadows to the hands -->
```

```
      <line id="hourhand" x1="50" y1="50" x2="50" y2="24"/>
      <line id="minutehand" x1="50" y1="50" x2="50" y2="20"/>
    </g>
  </svg>
</body>
</html>
```

This is the output (the SVG recreates itself to be always displaying the correct time):

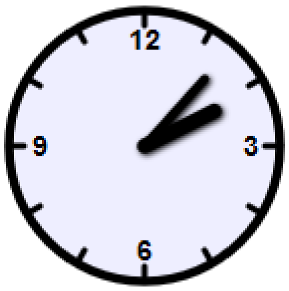


Figure 3: Clock SVG Example

Canvas (2d)

```
<!DOCTYPE html>
<html>
    <head>
        <title>Smiley Face</title>
        <link rel='stylesheet' type='text/css' href='stylesheet.css'/>
    </head>
    <body>
        <canvas id="a" width="200" height="200">
            This text is displayed if your browser does not support HTML5 Canvas.
        </canvas>
        <script type='text/javascript' src='script.js'></script>
    </body>
</html>
```

```
canvas {
    border: 1px dotted black;
}
```

```
// Set up!
var a_canvas = document.getElementById("a");
var context = a_canvas.getContext("2d");

// Draw the face
context.fillStyle = "yellow";
context.beginPath();
context.arc(95, 85, 40, 0, 2*Math.PI);
context.closePath();
context.fill();
context.lineWidth = 2;
context.stroke();
context.fillStyle = "black";

// Draw the left eye
context.beginPath();
context.arc(75, 75, 5, 0, 2*Math.PI);
context.closePath();
context.fill();

// Draw the right eye
context.beginPath();
context.arc(114, 75, 5, 0, 2*Math.PI);
context.closePath();
context.fill();

// Draw the mouth
context.beginPath();
context.arc(95, 90, 26, Math.PI, 2*Math.PI, true);
context.closePath();
context.fill();

// Write "Hello, World!"
context.font = "30px Garamond";
context.fillText("Hello, World!",15,175);
```

Output:

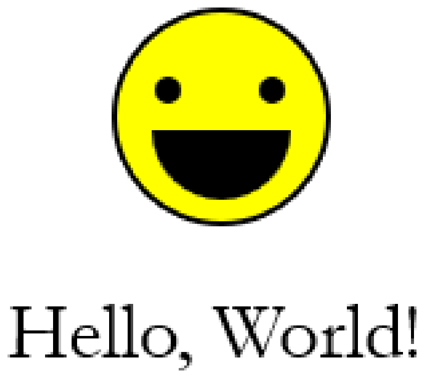


Figure 4: Canvas image example

Graphics attributes of the Canvas API

Property	Meaning
fillStyle	the color, gradient, or pattern for fills
font	the CSS font for text-drawing commands
globalAlpha	transparency to be added to all pixels drawn
globalCompositeOperation	how to combine new pixels with the ones underneath
lineCap	how the ends of lines are rendered
lineJoin	how vertexes are rendered
lineWidth	the width of stroked lines
miterLimit	maximum length of acute mitered vertexes
textAlign	horizontal alignment of text
textBaseLine	vertical alignment of text
shadowBlur	how crisp or fuzzy shadows are
shadowColor	the color of drop shadows
shadowOffsetX	the horizontal offset of shadows
shadowOffsetY	the vertical offset of shadows
strokeStyle	the color, gradient, or pattern for lines