Object methods, "this"

Objects are usually created to represent entities of the real world, like users, orders and so on:

```
let user = {
  name: "John",
  age: 30
};
```

And, in the real world, a user can *act*: select something from the shopping cart, login, logout etc.
Actions are represented in JavaScript by functions in properties.

Method examples

For a start, let's teach the `user` to say hello:

```
let user = {
  name: "John",
  age: 30
};

user.sayHi = function() {
  alert("Hello!");
};

user.sayHi(); // Hello!
```

Here we've just used a Function Expression to create a function and assign it to the property `user.sayHi` of the object.
Then we can call it as `user.sayHi()`. The user can now speak!
A function that is a property of an object is called its *method*.
So, here we've got a method `sayHi` of the object `user`.
Of course, we could use a pre-declared function as a method, like this:

```
let user = {
  // ...
};

// first, declare
function sayHi() {
  alert("Hello!");
}

// then add as a method
user.sayHi = sayHi;

user.sayHi(); // Hello!
```

Object-oriented programming
When we write our code using objects to represent entities, that's called [object-oriented programming], in short: "OOP".
OOP is a big thing, an interesting science of its own. How to choose the right entities? How to organize the interaction between them? That's architecture, and there are great books on that topic,

like “Design Patterns: Elements of Reusable Object-Oriented Software” by E. Gamma, R. Helm, R. Johnson, J. Vissides or “Object-Oriented Analysis and Design with Applications” by G. Booch, and more.

Method shorthand

There exists a shorter syntax for methods in an object literal:

```
// these objects do the same

user = {
  sayHi: function() {
    alert("Hello");
  }
};

// method shorthand looks better, right?
user = {
  sayHi() { // same as "sayHi: function(){...}"
    alert("Hello");
  }
};
```

As demonstrated, we can omit `"function"` and just write `sayHi()`.
To tell the truth, the notations are not fully identical. There are subtle differences related to object inheritance (to be covered later), but for now they do not matter. In almost all cases, the shorter syntax is preferred.

This in methods

It’s common that an object method needs to access the information stored in the object to do its job.
For instance, the code inside `user.sayHi()` may need the name of the `user`.
To access the object, a method can use the `this` keyword.
The value of `this` is the object “before dot”, the one used to call the method.
For instance:

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    // "this" is the "current object"
    alert(this.name);
  }

};

user.sayHi(); // John
```

Here during the execution of `user.sayHi()`, the value of `this` will be `user`.
Technically, it’s also possible to access the object without `this`, by referencing it via the outer variable:

```
let user = {
```

```
  name: "John",
  age: 30,

  sayHi() {
    alert(user.name); // "user" instead of "this"
  }

};
```

…But such code is unreliable. If we decide to copy `user` to another variable, e.g. `admin = user` and overwrite `user` with something else, then it will access the wrong object.
That's demonstrated below:

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    alert( user.name ); // leads to an error
  }

};


let admin = user;
user = null; // overwrite to make things obvious

admin.sayHi(); // TypeError: Cannot read property 'name' of null
```

If we used `this.name` instead of `user.name` inside the `alert`, then the code would work.

This is not bound

In JavaScript, keyword `this` behaves unlike most other programming languages. It can be used in any function, even if it's not a method of an object.
There's no syntax error in the following example:

```
function sayHi() {
  alert( this.name );
}
```

The value of `this` is evaluated during the run-time, depending on the context.
For instance, here the same function is assigned to two different objects and has different "this" in the calls:

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
  alert( this.name );
}

// use the same function in two objects
user.f = sayHi;
admin.f = sayHi;
```

```
// these calls have different this
// "this" inside the function is the object "before the dot"
user.f(); // John  (this == user)
admin.f(); // Admin  (this == admin)

admin['f'](); // Admin (dot or square brackets access the method – doesn't matter)
```

The rule is simple: if `obj.f()` is called, then `this` is `obj` during the call of `f`. So it's either `user` or `admin` in the example above.

Calling without an object: `this == undefined`

We can even call the function without an object at all:

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined
```

In this case `this` is `undefined` in strict mode. If we try to access `this.name`, there will be an error.

In non-strict mode the value of `this` in such case will be the *global object* (`window` in a browser, we'll get to it later in the chapter **global-object**. This is a historical behavior that `"use strict"` fixes.

Usually, such call is a programming error. If there's `this` inside a function, it expects to be called in an object context.

The consequences of unbound `this`

If you come from another programming language, then you are probably used to the idea of a "bound `this`", where methods defined in an object always have `this` referencing that object.

In JavaScript `this` is "free", its value is evaluated at call-time and does not depend on where the method was declared, but rather on what object is "before the dot".

The concept of run-time evaluated `this` has both pluses and minuses. On the one hand, a function can be reused for different objects. On the other hand, the greater flexibility creates more possibilities for mistakes.

Here our position is not to judge whether this language design decision is good or bad. We'll understand how to work with it, how to get benefits and avoid problems.

Arrow functions have no this

Arrow functions are special: they don't have their "own" `this`. If we reference `this` from such a function, it's taken from the outer "normal" function.

For instance, here `arrow()` uses `this` from the outer `user.sayHi()` method:

```
let user = {
  firstName: "Ilya",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};

user.sayHi(); // Ilya
```

That's a special feature of arrow functions, it's useful when we actually do not want to have a separate `this`, but rather to take it from the outer context. Later in the chapter **arrow functions** we'll go more deeply into arrow functions.

CSS-animations

CSS animations make it possible to do simple animations without JavaScript at all.
JavaScript can be used to control CSS animations and make them even better, with little code.

Css transition

The idea of CSS transitions is simple. We describe a property and how its changes should be animated. When the property changes, the browser paints the animation.
That is, all we need is to change the property, and the fluid transition will be done by the browser.
For instance, the CSS below animates changes of `background-color` for 3 seconds:

```
.animated {
  transition-property: background-color;
  transition-duration: 3s;
}
```

Now if an element has `.animated` class, any change of `background-color` is animated during 3 seconds.
Click the button below to animate the background:

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

<style>
  #color {
    transition-property: background-color;
    transition-duration: 3s;
  }
</style>

<script>
  color.onclick = function() {
    this.style.backgroundColor = 'red';
  };
</script>
```

There are 4 properties to describe CSS transitions:
`transition-property`
`transition-duration`
`transition-timing-function`
`transition-delay`
We'll cover them in a moment, for now let's note that the common `transition` property allows declaring them together in the order: `property duration timing-function delay`, as well as animating multiple properties at once.
For instance, this button animates both `color` and `font-size`:

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

<style>
#growing {
  transition: font-size 3s, color 2s;
```

```
}
</style>

<script>
growing.onclick = function() {
  this.style.fontSize = '36px';
  this.style.color = 'red';
};
</script>
```

Now, let's cover animation properties one by one.

Transition property

In `transition-property`, we write a list of properties to animate, for instance: `left`, `margin-left`, `height`, `color`. Or we could write `all`, which means "animate all properties".
Do note that, there are properties which can not be animated. However, CSS animated properties.

Transition duration

In `transition-duration` we can specify how long the animation should take. The time should be in time: in seconds `s` or milliseconds `ms`.

Transition delay

In `transition-delay` we can specify the delay *before* the animation. For instance, if `transition-delay` is `1s` and `transition-duration` is `2s`, then the animation starts 1 second after the property change and the total duration will be 2 seconds.
Negative values are also possible. Then the animation is shown immediately, but the starting point of the animation will be after given value (time). For example, if `transition-delay` is `-1s` and `transition-duration` is `2s`, then animation starts from the halfway point and total duration will be 1 second.
Here the animation shifts numbers from `0` to `9` using CSS `translate` property:

script.js
style.css
index.html
The `transform` property is animated like this:

```
#stripe.animate {
  transform: translate(-90%);
  transition-property: transform;
  transition-duration: 9s;
}
```

In the example above JavaScript adds the class `.animate` to the element – and the animation starts:

```
stripe.classList.add('animate');
```

We could also start it from somewhere in the middle of the transition, from an exact number, e.g. corresponding to the current second, using a negative `transition-delay`.
Here if you click the digit – it starts the animation from the current second:
Result
script.js

style.css
index.html
JavaScript does it with an extra line:

```
stripe.onclick = function() {
  let sec = new Date().getSeconds() % 10;
  // for instance, -3s here starts the animation from the 3rd second
  stripe.style.transitionDelay = '-' + sec + 's';
  stripe.classList.add('animate');
};
```

Transition timing function

The timing function describes how the animation process is distributed along its timeline. Will it start slowly and then go fast, or vice versa.
It appears to be the most complicated property at first. But it becomes very simple if we devote a bit time to it.
That property accepts two kinds of values: a Bezier curve or steps. Let's start with the curve, as it's used more often.

Bezier curve

The timing function can be set as a Bezier curve with 4 control points that satisfy the conditions:
First control point: `(0,0)`.
Last control point: `(1,1)`.
For intermediate points, the values of `x` must be in the interval `0..1`, `y` can be anything.
The syntax for a Bezier curve in CSS: `cubic-bezier(x2, y2, x3, y3)`. Here we need to specify only 2nd and 3rd control points, because the 1st one is fixed to `(0,0)` and the 4th one is `(1,1)`.
The timing function describes how fast the animation process goes.
The `x` axis is the time: `0` – the start, `1` – the end of `transition-duration`.
The `y` axis specifies the completion of the process: `0` – the starting value of the property, `1` – the final value.
The simplest variant is when the animation goes uniformly, with the same linear speed. That can be specified by the curve `cubic-bezier(0, 0, 1, 1)`.
Here's how that curve looks:
…As we can see, it's just a straight line. As the time (`x`) passes, the completion (`y`) of the animation steadily goes from `0` to `1`.
The train in the example below goes from left to right with the permanent speed (click it):

style.css
index.html

```
.train {
  left: 0;
  transition: left 5s cubic-bezier(0, 0, 1, 1);
  /* click on a train sets left to 450px, thus triggering the animation
*/
}
```

The CSS `transition` is based on that curve:
…And how can we show a train slowing down?
We can use another Bezier curve: `cubic-bezier(0.0, 0.5, 0.5 ,1.0)`.
The graph:
As we can see, the process starts fast: the curve soars up high, and then slower and slower.
Here's the timing function in action (click the train):

Result
style.css
index.html

CSS:

```
.train {
  left: 0;
  transition: left 5s cubic-bezier(0, .5, .5, 1);
  /* click on a train sets left to 450px, thus triggering the animation
*/
}
```

There are several built-in curves: `linear`, `ease`, `ease-in`, `ease-out` and `ease-in-out`.
The `linear` is a shorthand for `cubic-bezier(0, 0, 1, 1)` – a straight line, which we described above.
Other names are shorthands for the following `cubic-bezier`:

ease*	**ease-in**	**ease-out**	**ease-in-out**
`(0.25, 0.1, 0.25, 1.0)`	`(0.42, 0, 1.0, 1.0)`	`(0, 0, 0.58, 1.0)`	`(0.42, 0, 0.58, 1.0)`

* – by default, if there's no timing function, `ease` is used.
So we could use `ease-out` for our slowing down train:

```
.train {
  left: 0;
  transition: left 5s ease-out;
  /* same as transition: left 5s cubic-bezier(0, .5, .5, 1); */
}
```

But it looks a bit differently.
A Bezier curve can make the animation exceed its range.
The control points on the curve can have any y coordinates: even negative or huge ones. Then the Bezier curve would also extend very low or high, making the animation go beyond its normal range.
In the example below the animation code is:

```
.train {
  left: 100px;
  transition: left 5s cubic-bezier(.5, -1, .5, 2);
  /* click on a train sets left to 450px */
}
```

The property `left` should animate from `100px` to `400px`.
But if you click the train, you'll see that:
First, the train goes *back*: `left` becomes less than `100px`.
Then it goes forward, a little bit farther than `400px`.
And then back again – to `400px`.
Result
style.css
index.html
Why it happens is pretty obvious if we look at the graph of the given Bezier curve:
We moved the y coordinate of the 2nd point below zero, and for the 3rd point we made it over `1`, so the curve goes out of the "regular" quadrant. The y is out of the "standard" range `0..1`.
As we know, y measures "the completion of the animation process". The value `y = 0` corresponds to the starting property value and `y = 1` – the ending value. So values `y<0` move the property beyond the starting `left` and `y>1` – past the final `left`.
That's a "soft" variant for sure. If we put y values like `-99` and `99` then the train would jump out of the range much more.
But how do we make a Bezier curve for a specific task? There are many tools.

For instance, we can do it on the site https://cubic-bezier.com.
Browser developer tools also have special support for Bezier curves in CSS:
Open the developer tools with `F12` (Mac: `Cmd+Opt+I`).
Select the `Elements` tab, then pay attention to the `Styles` sub-panel at the right side.
CSS properties with a word `cubic-bezier` will have an icon before this word.
Click this icon to edit the curve.

Steps

The timing function `steps(number of steps[, start/end])` allows splitting an transition into multiple steps.
Let's see that in an example with digits.
Here's a list of digits, without any animations, just as a source:
In the HTML, a stripe of digits is enclosed into a fixed-length `<div id="digits">`:

```
<div id="digit">
  <div id="stripe">0123456789</div>
</div>
```

The `#digit` div has a fixed width and a border, so it looks like a red window.
We'll make a timer: the digits will appear one by one, in a discrete way.
To achieve that, we'll hide the `#stripe` outside of `#digit` using `overflow: hidden`, and then shift the `#stripe` to the left step-by-step.
There will be 9 steps, a step-move for each digit:

```
#stripe.animate  {
  transform: translate(-90%);
  transition: transform 9s steps(9, start);
}
```

The first argument of `steps(9, start)` is the number of steps. The transform will be split into 9 parts (10% each). The time interval is automatically divided into 9 parts as well, so `transition: 9s` gives us 9 seconds for the whole animation – 1 second per digit.
The second argument is one of two words: `start` or end.
The `start` means that in the beginning of animation we need to make the first step immediately.
A click on the digit changes it to 1 (the first step) immediately, and then changes in the beginning of the next second.
The process is progressing like this:
`0s` – `-10%` (first change in the beginning of the 1st second, immediately)
`1s` – `-20%`
…
`8s` – `-90%`
(the last second shows the final value).
Here, the first change was immediate because of `start` in the `steps`.
The alternative value end would mean that the change should be applied not in the beginning, but at the end of each second.
So the process for `steps(9, end)` would go like this:
`0s` – `0` (during the first second nothing changes)
`1s` – `-10%` (first change at the end of the 1st second)
`2s` – `-20%`
…
`9s` – `-90%`
There are also some pre-defined shorthands for `steps(...)`:

`step-start` – is the same as `steps(1, start)`. That is, the animation starts immediately and takes 1 step. So it starts and finishes immediately, as if there were no animation.
`step-end` – the same as `steps(1, end)`: make the animation in a single step at the end of `transition-duration`.
These values are rarely used, as they represent not a real animation, but rather a single-step change. We mention them here for completeness.

Event transitionend

When the CSS animation finishes, the `transitionend` event triggers.
It is widely used to do an action after the animation is done. Also we can join animations.
For instance, the ship in the example below starts to sail there and back when clicked, each time farther and farther to the right:
The animation is initiated by the function go that re-runs each time the transition finishes, and flips the direction:

```
boat.onclick = function() {
  //...
  let times = 1;

  function go() {
    if (times % 2) {
      // sail to the right
      boat.classList.remove('back');
      boat.style.marginLeft = 100 * times + 200 + 'px';
    } else {
      // sail to the left
      boat.classList.add('back');
      boat.style.marginLeft = 100 * times - 200 + 'px';
    }

  }

  go();

  boat.addEventListener('transitionend', function() {
    times++;
    go();
  });
};
```

The event object for `transitionend` has a few specific properties:
`event.propertyName`
The property that has finished animating. Can be good if we animate multiple properties simultaneously.
`event.elapsedTime`
The time (in seconds) that the animation took, without `transition-delay`.

Keyframes

We can join multiple simple animations together using the `@keyframes` CSS rule.

It specifies the “name” of the animation and rules – what, when and where to animate. Then using the `animation` property, we can attach the animation to the element and specify additional parameters for it.
Here’s an example with explanations:

```
<div class="progress"></div>

<style>
  @keyframes go-left-right {        /* give it a name: "go-left-right"
*/
    from { left: 0px; }             /* animate from left: 0px */
    to { left: calc(100% - 50px); } /* animate to left: 100%-50px */
  }

  .progress {
    animation: go-left-right 3s infinite alternate;
    /* apply the animation "go-left-right" to the element
       duration 3 seconds
       number of times: infinite
       alternate direction every time
    */

    position: relative;
    border: 2px solid green;
    width: 50px;
    height: 20px;
    background: lime;
  }
</style>
```

There are many articles about `@keyframes` and a detailed specification.
You probably won’t need `@keyframes` often, unless everything is in constant motion on your sites.

Performance

Most CSS properties can be animated, because most of them are numeric values. For instance, `width`, `color`, `font-size` are all numbers. When you animate them, the browser gradually changes these numbers frame by frame, creating a smooth effect.
However, not all animations will look as smooth as you’d like, because different CSS properties cost differently to change.
In more technical details, when there’s a style change, the browser goes through 3 steps to render the new look:

Layout: re-compute the geometry and position of each element, then
Paint: re-compute how everything should look like at their places, including background, colors,
Composite: render the final results into pixels on screen, apply CSS transforms if they exist.
During a CSS animation, this process repeats every frame. However, CSS properties that never affect geometry or position, such as `color`, may skip the Layout step. If a `color` changes, the browser doesn’t calculate any new geometry, it goes to Paint → Composite. And there are few properties that directly go to Composite. You can find a longer list of CSS properties and which stages they trigger at https://csstriggers.com.

The calculations may take time, especially on pages with many elements and a complex layout. And the delays are actually visible on most devices, leading to “jittery”, less fluid animations.
Animations of properties that skip the Layout step are faster. It’s even better if Paint is skipped too.
The `transform` property is a great choice, because:
CSS transforms affect the target element box as a whole (rotate, flip, stretch, shift it).
CSS transforms never affect neighbour elements.
…So browsers apply `transform` “on top” of existing Layout and Paint calculations, in the Composite stage.

In other words, the browser calculates the Layout (sizes, positions), paints it with colors, backgrounds, etc at the Paint stage, and then applies `transform` to element boxes that need it.
Changes (animations) of the `transform` property never trigger Layout and Paint steps. More than that, the browser leverages the graphics accelerator (a special chip on the CPU or graphics card) for CSS transforms, thus making them very efficient.
Luckily, the `transform` property is very powerful. By using `transform` on an element, you could rotate and flip it, stretch and shrink it, move it around. So instead of `left/margin-left` properties we can use `transform: translateX(…)`, use `transform: scale` for increasing element size, etc.

The `opacity` property also never triggers Layout (also skips Paint in Mozilla Gecko). We can use it for show/hide or fade-in/fade-out effects.
Paring `transform` with `opacity` can usually solve most of our needs, providing fluid, good-looking animations.
For example, here clicking on the `#boat` element adds the class with `transform: translateX(300)` and `opacity: 0`, thus making it move `300px` to the right and disappear:

```
<img src="https://js.cx/clipart/boat.png" id="boat">

<style>
#boat {
  cursor: pointer;
  transition: transform 2s ease-in-out, opacity 2s ease-in-out;
}

.move {
  transform: translateX(300px);
  opacity: 0;
}
</style>
<script>
  boat.onclick = () => boat.classList.add('move');
</script>
```

Here’s a more complex example, with `@keyframes`:

```
<h2 onclick="this.classList.toggle('animated')">click me to start / stop</h2>
<style>
  .animated {
    animation: hello-goodbye 1.8s infinite;
    width: fit-content;
  }
  @keyframes hello-goodbye {
    0% {
      transform: translateY(-60px) rotateX(0.7turn);
      opacity: 0;
```

```
    }
    50% {
      transform: none;
      opacity: 1;
    }
    100% {
      transform: translateX(230px) rotateZ(90deg) scale(0.5);
      opacity: 0;
    }
  }
</style>
```