**Тема 8. Формы, элементы управления. Анимация**

1. **Свойства и методы формы.**
2. **Фокусировка элемента.**
3. **Изменение значений элемента.**
4. **Формы: отправка, событие и метод submit.**
5. **JavaScript-анимация.**

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. Свойства и методы формы.

Формы и элементы управления, такие как <input>, имеют множество специальных свойств и событий.

Формы в документе входят в специальную коллекцию document.forms. Это – так называемая «именованная» коллекция: можно использовать для получения формы как её имя, так и порядковый номер в документе.

document.forms.my - форма с именем "my" (name="my")

document.forms[0] - первая форма в документе

Когда форма уже получена, любой элемент доступен в именованной коллекции form.elements. Например:

<form name="my">

<input name="one" value="1">

<input name="two" value="2">

</form>

<script>

// получаем форму

let form = document.forms.my; // <form name="my"> element

// получаем элемент

let elem = form.elements.one; // <input name="one"> element

alert(elem.value); // 1

</script>

Может быть несколько элементов с одним и тем же именем, это часто бывает с кнопками-переключателями radio. В этом случае form.elements[name] является коллекцией, например:

<form>

<input type="radio" name="age" value="10">

<input type="radio" name="age" value="20">

</form>

<script>

let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0].value); // 10, the first input value

</script>

Эти навигационные свойства не зависят от структуры тегов внутри формы. Все элементы, как бы глубоко они ни находились в форме, доступны в коллекции form.elements.

Форма может содержать один или несколько элементов <fieldset> внутри себя. Они также поддерживают свойство elements. Например:

<body>

<form id="form">

<fieldset name="userFields">

<legend>info</legend>

<input name="login" type="text">

</fieldset>

</form>

<script>

alert(form.elements.login); // <input name="login">

let fieldset = form.elements.userFields;

alert(fieldset); // HTMLFieldSetElement

// можно получить информацию как из формы, так и из fieldset

alert(fieldset.elements.login == form.elements.login); // true

</script>

</body>

Есть более короткая запись: можно получить доступ к элементу через form[index/name]. Вместо form.elements.login можно написать form.login. Это также работает, но есть небольшая проблема: если надо получить элемент, а затем менять его свойство name, то он всё ещё будет доступен под старым именем (также, как и под новым). В этом легче разобраться на примере:

<form id="form">

<input name="login">

</form>

<script>

alert(form.elements.login == form.login); // true, ведь это одинаковые <input>

form.login.name = "username"; // изменяем свойство name у элемента input

// form.elements обновили свои имена:

alert(form.elements.login); // undefined

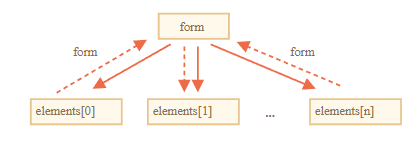
alert(form.elements.username); // input

// теперь для прямого доступа можно использовать оба имени: новое и старое

alert(form.username == form.login); // true

</script>

Для любого элемента форма доступна через element.form. Так что форма ссылается на все элементы, а эти элементы ссылаются на форму. Вот иллюстрация:



Пример:

<form id="form">

<input type="text" name="login">

</form>

<script>

// form -> element

let login = form.login;

// element -> form

alert(login.form); // HTMLFormElement

</script>

Рассмотрим элементы управления, используемые в формах, обращая внимание на их особенности.

[**input и textarea**](https://learn.javascript.ru/form-elements#input-i-textarea)

К их значению можно получить доступ через свойство input.value (строка) или input.checked (булево значение) для чекбоксов. Вот так:

input.value = "Новое значение";

textarea.value = "Новый текст";

input.checked = true; // для чекбоксов и переключателей

Обратите внимание: хоть <textarea>...</textarea> и хранит значение как вложенный HTML, не следует использовать textarea.innerHTML. Там хранится только тот HTML, который был изначально на странице, а не текущее значение.

[**select и option**](https://learn.javascript.ru/form-elements#select-i-option)

Элемент <select> имеет 3 важных свойства:

1. select.options – коллекция из элементов <option>,
2. select.value – значение выбранного в данный момент <option>,
3. select.selectedIndex – номер выбранного <option>.

Имеется три способа задать значение для <select>:

1. Найти необходимый <option> и установить в option.selected значение true.
2. Установить в select.value значение нужного нам <option>.
3. Установить в select.selectedIndex номер <option>.

Первый способ наиболее понятный, но (2) и (3) являются более удобными при работе. Вот эти способы на примере:

<select id="select">

<option value="apple">Яблоко</option>

<option value="pear">Груша</option>

<option value="banana">Банан</option>

</select>

<script>

// все три строки делают одно и то же

select.options[2].selected = true;

select.selectedIndex = 2;

select.value = 'banana';

</script>

В отличие от большинства других элементов управления, <select multiple> позволяет выбрать несколько вариантов. В этом случае необходимо пройтись по select.options, чтобы получить все выбранные значения. Например так:

<select id="select" multiple>

<option value="blues" selected>Блюз</option>

<option value="rock" selected>Рок</option>

<option value="classic">Классика</option>

</select>

<script>

// получаем все выбранные значения из списка множественного выбора

let selected = Array.from(select.options)

.filter(option => option.selected)

.map(option => option.value);

alert(selected); // Блюз,Рок

</script>

Полное описание элемента <select> доступно в спецификации <https://html.spec.whatwg.org/multipage/forms.html#the-select-element>.

Элемент <option> редко используется сам по себе. В описании [элемента option](https://html.spec.whatwg.org/multipage/forms.html#the-option-element) есть короткий синтаксис для создания элемента:

option = new Option(text, value, defaultSelected, selected);

Параметры:

* text – текст внутри,
* value – значение,
* defaultSelected – если true, то ставится HTML-атрибут selected,
* selected – если true, то элемент <option> будет выбранным.

Пример:

let option = new Option("Текст", "value");

// создаст <option value="value">Текст</option>

Тот же элемент, но выбранный:

let option = new Option("Текст", "value", true, true);

Элементы <option> имеют дополнительные свойства:

* selected – выбрана ли опция,
* index – номер опции среди других в списке <select>,
* text – содержимое опции (то, что видит посетитель).

1. Фокусировка элемента.

Элемент получает фокус, когда пользователь кликает по нему или использует клавишу Tab. Также существует HTML-атрибут autofocus, который устанавливает фокус на элемент, когда страница загружается. Есть и другие способы получения фокуса.

Фокусировка обычно означает: «приготовься к вводу данных на этом элементе», это хороший момент, чтобы инициализовать или загрузить что-нибудь. Момент потери фокуса («blur») может быть важнее. Это момент, когда пользователь кликает куда-то ещё или нажимает Tab, чтобы переключиться на следующее поле формы. Есть другие причины потери фокуса.

Потеря фокуса обычно означает «данные введены», и можно выполнить проверку введенных данных или даже отправить эти данные на сервер и так далее. В работе с событиями фокусировки есть важные особенности.

[**События focus/blur**](https://learn.javascript.ru/focus-blur#sobytiya-focus-blur)

Событие focus вызывается в момент фокусировки, а blur – когда элемент теряет фокус. Используем их для валидации (проверки) введённых данных. В примере ниже:

* Обработчик blur проверяет, введён ли email, и если нет – показывает ошибку.
* Обработчик focus скрывает это сообщение об ошибке (в момент потери фокуса проверка повторится):

<style>

.invalid { border-color: red; }

#error { color: red }

</style>

Ваш email: <input type="email" id="input">

<div id="error"></div>

<script>

input.onblur = function() {

if (!input.value.includes('@')) { // не email

input.classList.add('invalid');

error.innerHTML = 'Пожалуйста, введите правильный email.'

}

};

input.onfocus = function() {

if (this.classList.contains('invalid')) {

// удаляем индикатор ошибки, т.к. пользователь хочет ввести данные заново

this.classList.remove('invalid');

error.innerHTML = "";

}

};

</script>





Современный HTML позволяет делать валидацию с помощью атрибутов required, pattern и т.д. Иногда – этого достаточно. JavaScript можно использовать, когда требуется больше гибкости. А также можно отправлять изменённое значение на сервер, если оно правильное.

[**Методы focus/blur**](https://learn.javascript.ru/focus-blur#metody-focus-blur)

Методы elem.focus() и elem.blur() устанавливают/снимают фокус. Например, запретим посетителю переключаться с поля ввода, если введённое значение не прошло валидацию:

<style>

.error {

background: red;

}

</style>

Ваш email: <input type="email" id="input">

<input type="text" style="width:280px" placeholder="введите неверный email и кликните сюда">

<script>

input.onblur = function() {

if (!this.value.includes('@')) { // не email

// показать ошибку

this.classList.add("error");

// вернуть фокус обратно

input.focus();

} else {

this.classList.remove("error");

}

};

</script>





Если что-нибудь ввести и нажать Tab или кликнуть в другое место, тогда onblur вернёт фокус обратно.

Нельзя отменить потерю фокуса, вызвав event.preventDefault() в обработчике onblur потому, что onblur срабатывает после потери фокуса элементом.

Потеря фокуса может произойти по множеству причин. Одна из них – когда посетитель кликает куда-то ещё. Но и JavaScript может быть причиной, например:

* alert переводит фокус на себя – элемент теряет фокус (событие blur), а когда alert закрывается – элемент получает фокус обратно (событие focus);
* если элемент удалить из DOM, фокус также будет потерян. Если элемент добавить обратно, то фокус не вернётся.

Из-за этих особенностей обработчики focus/blur могут сработать тогда, когда это не требуется.

Многие элементы по умолчанию не поддерживают фокусировку. Какие именно – зависит от браузера, но одно всегда верно: поддержка focus/blur гарантирована для элементов, с которыми посетитель может взаимодействовать: <button>, <input>, <select>, <a> и т.д. С другой стороны, элементы форматирования <div>, <span>, <table> – по умолчанию не могут получить фокус. Метод elem.focus() не работает для них, и события focus/blur никогда не срабатывают.

Это можно изменить HTML-атрибутом tabindex. Цель этого атрибута – указать порядковый номер элемента, когда клавиша Tab используется для переключения между элементами. То есть: если имеется два элемента, первый имеет tabindex="1", а второй tabindex="2", то находясь в первом элементе и нажав Tab – фокус переместится на второй.

Есть два специальных значения:

* tabindex="0" делает элемент последним,
* tabindex="-1" значит, что Tab игнорирует этот элемент.

Любой элемент поддерживает фокусировку, если имеет tabindex. Например, список ниже. Кликните первый пункт в списке и нажмите Tab:

Кликните первый пункт в списке и нажмите Tab. Продолжайте следить за порядком. Обратите внимание, что много последовательных нажатий Tab могут вывести фокус из iframe с примером.

<ul>

<li tabindex="1">Один</li>

<li tabindex="0">Ноль</li>

<li tabindex="2">Два</li>

<li tabindex="-1">Минус один</li>

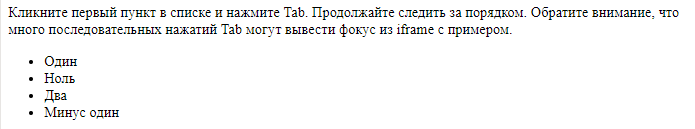
</ul>

<style>

li { cursor: pointer; }

:focus { outline: 1px dashed green; }

</style>



Порядок такой: 1 - 2 - 0 (ноль всегда последний). Обычно <li> не поддерживает фокусировку, но tabindex полностью включает её, а также события и стилизацию псевдоклассом :focus.

Можно добавить tabindex из JavaScript, используя свойство elem.tabIndex. Это даст тот же эффект.

[**События focusin/focusout**](https://learn.javascript.ru/focus-blur#sobytiya-focusin-focusout)

События focus и blur не всплывают. Например, нельзя использовать onfocus на <form>, чтобы подсветить её:

<!-- добавить класс при фокусировке на форме -->

<form onfocus="this.className='focused'">

<input type="text" name="name" value="Имя">

<input type="text" name="surname" value="Фамилия">

</form>

<style> .focused { outline: 1px solid red; } </style>





Пример выше не работает, потому что когда пользователь перемещает фокус на <input>, событие focus срабатывает только на этом элементе. Это событие не всплывает. Следовательно, form.onfocus никогда не срабатывает.

У этой проблемы два решения. Первое: забавная особенность – focus/blur не всплывают, но передаются вниз на фазе перехвата. Это сработает:

<form id="form">

<input type="text" name="name" value="Имя">

<input type="text" name="surname" value="Фамилия">

</form>

<style> .focused { outline: 1px solid red; } </style>

<script>

// установить обработчик на фазе перехвата (последний аргумент true)

form.addEventListener("focus", () => form.classList.add('focused'), true);

form.addEventListener("blur", () => form.classList.remove('focused'), true);

</script>





Второе решение: события focusin и focusout – такие же, как и focus/blur, но они всплывают. Заметьте, что эти события должны использоваться с elem.addEventListener, но не с on<event>. Второй рабочий вариант:

<form id="form">

<input type="text" name="name" value="Имя">

<input type="text" name="surname" value="Фамилия">

</form>

<style> .focused { outline: 1px solid red; } </style>

<script>

form.addEventListener("focusin", () => form.classList.add('focused'));

form.addEventListener("focusout", () => form.classList.remove('focused'));

</script>





1. Изменение значений элемента.

Рассмотрим различные события, сопутствующие обновлению данных.

[**Событие: change**](https://learn.javascript.ru/events-change-input#sobytie-change)

Событие change срабатывает по окончании изменения элемента. Для текстовых <input> это означает, что событие происходит при потере фокуса. Пока пользователь печатает в текстовом поле в примере ниже, событие не происходит. Но когда перемещает фокус в другое место, например, нажимая на кнопку, то произойдёт событие change:

<input type="text" onchange="alert(this.value)">

<input type="button" value="Button">





Для других элементов: select, input type=checkbox/radio событие запускается сразу после изменения значения:

<select onchange="alert(this.value)">

<option value="">Выберите что-нибудь</option>

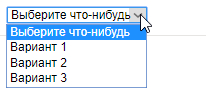
<option value="1">Вариант 1</option>

<option value="2">Вариант 2</option>

<option value="3">Вариант 3</option>

</select>





[**Событие: input**](https://learn.javascript.ru/events-change-input#sobytie-input)

Событие input срабатывает каждый раз при изменении значения. В отличие от событий клавиатуры, оно работает при любых изменениях значений, даже если они не связаны с клавиатурными действиями: вставка с помощью мыши или распознавание речи при диктовке текста. Например:

<input type="text" id="input"> oninput: <span id="result"></span>

<script>

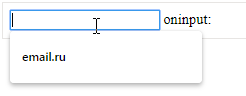
input.oninput = function() {

result.innerHTML = input.value;

};

</script>





Если надо обрабатывать каждое изменение в <input>, то это событие является лучшим выбором. С другой стороны, событие input не происходит при вводе с клавиатуры или иных действиях, если при этом не меняется значение в текстовом поле, т.е. нажатия клавиш ⇦, ⇨ и подобных при фокусе на текстовом поле не вызовут это событие.

Событие input происходит после изменения значения. Поэтому нельзя использовать event.preventDefault() там – будет уже слишком поздно, никакого эффекта не будет.

**[События: cut, copy, paste](https://learn.javascript.ru/events-change-input" \l "sobytiya-cut-copy-paste)**

Эти события происходят при вырезании/копировании/вставке данных. Они относятся к классу [ClipboardEvent](https://www.w3.org/TR/clipboard-apis/#clipboard-event-interfaces) и обеспечивают доступ к копируемым/вставляемым данным. Также можно использовать event.preventDefault() для предотвращения действия по умолчанию, и в итоге ничего не скопируется/не вставится. Например, код, приведённый ниже, предотвращает все подобные события и показывает, что надо вырезать/копировать/вставить:

<input type="text" id="input">

<script>

input.oncut = input.oncopy = input.onpaste = function(event) {

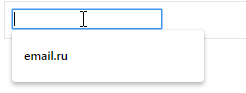
alert(event.type + ' - ' + event.clipboardData.getData('text/plain'));

return false;

};

</script>





Технически, можно скопировать/вставить всё. Например, можно скопировать файл из файловой системы и вставить его. Существует список методов [в спецификации](https://www.w3.org/TR/clipboard-apis/#dfn-datatransfer) для работы с различными типами данных, чтения/записи в буфер обмена. Но обратите внимание, что буфер обмена работает глобально, на уровне ОС. Большинство браузеров в целях безопасности разрешают доступ на чтение/запись в буфер обмена только в рамках определённых действий пользователя, к примеру, в обработчиках событий onclick.

Также запрещается генерировать «пользовательские» события буфера обмена при помощи dispatchEvent во всех браузерах, кроме Firefox.

1. Формы: отправка, событие и метод submit.

При отправке формы срабатывает событие submit, оно обычно используется для проверки (валидации) формы перед её отправкой на сервер или для предотвращения отправки и обработки её с помощью JavaScript.

Метод form.submit() позволяет инициировать отправку формы из JavaScript. Можно использовать его для динамического создания и отправки наших собственных форм на сервер. Рассмотрим их подробнее.

[**Событие: submit**](https://learn.javascript.ru/forms-submit#sobytie-submit)

Есть два основных способа отправить форму:

1. Первый – нажать кнопку <input type="submit"> или <input type="image">.
2. Второй – нажать Enter, находясь на каком-нибудь поле.

Оба действия сгенерируют событие submit на форме. Обработчик может проверить данные, и если есть ошибки, показать их и вызвать event.preventDefault(), тогда форма не будет отправлена на сервер.

В примере ниже оба действия показывают alert и форма не отправится благодаря return false:

<form onsubmit="alert('submit!');return false">

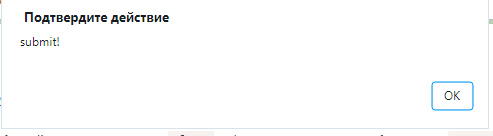
Первый пример: нажмите Enter: <input type="text" value="Текст"><br>

Второй пример: нажмите на кнопку "Отправить": <input type="submit" value="Отправить">

</form>



Первый и второй примеры, результат одинаковый:



При отправке формы по нажатию Enter в текстовом поле, генерируется событие click на кнопке <input type="submit">. При этом никакого клика на самом деле не было. Пример:

<form onsubmit="alert('submit!');return false">

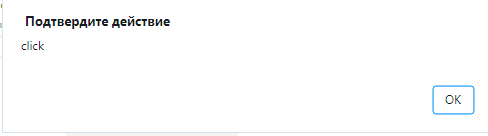
<input type="text" size="30" value="Установите фокус здесь и нажмите Enter">

<input type="submit" value="Отправить" onclick="alert('click')">

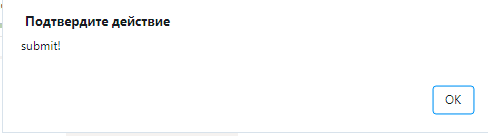
</form>



Появилось окно:



После клика на ок:



[**Метод: submit**](https://learn.javascript.ru/forms-submit#metod-submit)

Чтобы отправить форму на сервер вручную, можно вызвать метод form.submit(). При этом событие submit не генерируется. Предполагается, что если программист вызывает метод form.submit(), то он уже выполнил всю соответствующую обработку. Иногда это используют для генерации формы и отправки её вручную, например так:

let form = document.createElement('form');

form.action = 'https://google.com/search';

form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// перед отправкой формы, её нужно вставить в документ

document.body.append(form);

form.submit();

1. JavaScript-анимация.

С помощью JavaScript-анимаций можно делать вещи, которые нельзя реализовать на CSS. Например, движение по сложному пути с временной функцией, отличной от кривой Безье, или canvas-анимации.

[**Использование setInterval**](https://learn.javascript.ru/js-animation#ispolzovanie-setinterval)

Анимация реализуется через последовательность кадров, каждый из которых немного меняет HTML/CSS-свойства. Например, изменение style.left от 0px до 100px – двигает элемент. И если делать это с помощью setInterval, изменяя на 2px с небольшими интервалами времени, например, 50 раз в секунду, тогда изменения будут выглядеть плавными. Принцип такой же, как в кино: 24 кадров в секунду достаточно, чтобы создать эффект плавности. Код мог бы выглядеть так:

let timer = setInterval(function() {

if (animation complete) clearInterval(timer);

else increase style.left by 2px

}, 20); // изменять на 2px каждые 20ms, это около 50 кадров в секунду

Более детальная реализация этой анимации:

let start = Date.now(); // запомнить время начала

let timer = setInterval(function() {

let timePassed = Date.now() - start;

if (timePassed >= 2000) {

clearInterval(timer); // закончить анимацию через 2 секунды

return;

}

// отрисовать анимацию

draw(timePassed);

}, 20);

// в то время как timePassed изменяется от 0 до 2000

// left изменяет значение от 0px до 400px

function draw(timePassed) {

train.style.left = timePassed / 5 + 'px';

}

**[Использование requestAnimationFrame](https://learn.javascript.ru/js-animation" \l "ispolzovanie-requestanimationframe)**

Допустим есть несколько анимаций, работающих одновременно. Если запустить их независимо с помощью setInterval(..., 20), тогда браузеру будет необходимо выполнять отрисовку гораздо чаще, чем раз в 20ms.

Это происходит из-за того, что каждая анимация имеет своё собственное время старта и «каждые 20 миллисекунд» для разных анимаций – разные. Интервалы не выравнены и будет несколько независимых срабатываний в течение 20ms. Другими словами:

setInterval(function() {

animate1();

animate2();

animate3();

}, 20)

меньше нагружают систему, чем три независимых функции:

setInterval(animate1, 20); // независимые анимации

setInterval(animate2, 20); // в разных местах кода

setInterval(animate3, 20);

Эти независимые анимации лучше сгруппировать вместе, тогда они будут легче для браузера, а значит – не грузить процессор и более плавно выглядеть.

Также стоит учитывать следующее: когда CPU перегружен или есть другие причины делать перерисовку реже (например, когда вкладка браузера скрыта), не следует делать её каждые 20ms.

Спецификация [Animation timing](http://www.w3.org/TR/animation-timing/) описывает функцию requestAnimationFrame, которая решает все описанные проблемы и делает даже больше. Синтаксис:

let requestId = requestAnimationFrame(callback)

Такой вызов планирует запуск функции callback на ближайшее время, когда браузер сочтёт возможным осуществить анимацию. Если в callback происходит изменение элемента, тогда оно будет сгруппировано с другими requestAnimationFrame и CSS-анимациями. Таким образом браузер выполнит один геометрический пересчёт и отрисовку, вместо нескольких. Значение requestId может быть использовано для отмены анимации:

// отмена запланированного запуска callback

cancelAnimationFrame(requestId);

Функция callback имеет один аргумент – время, прошедшее с момента начала загрузки страницы в миллисекундах. Это значение может быть получено с помощью вызова [performance.now()](https://developer.mozilla.org/ru/docs/Web/API/Performance/now). Как правило, callback запускается очень быстро, если только не перегружен CPU или не разряжена батарея ноутбука, или у браузера нет какой-то ещё причины замедлиться.

Код ниже показывает время между первыми 10 запусками requestAnimationFrame. Обычно оно 10-20 мс:

<script>

let prev = performance.now();

let times = 0;

requestAnimationFrame(function measure(time) {

document.body.insertAdjacentHTML("beforeEnd", Math.floor(time - prev) + " ");

prev = time;

if (times++ < 10) requestAnimationFrame(measure);

})

</script>

[**Структура анимации**](https://learn.javascript.ru/js-animation#struktura-animatsii)

Теперь можно создать более сложную функцию анимации с помощью requestAnimationFrame:

function animate({timing, draw, duration}) {

let start = performance.now();

requestAnimationFrame(function animate(time) {

// timeFraction изменяется от 0 до 1

let timeFraction = (time - start) / duration;

if (timeFraction > 1) timeFraction = 1;

// вычисление текущего состояния анимации

let progress = timing(timeFraction);

draw(progress); // отрисовать её

if (timeFraction < 1) {

requestAnimationFrame(animate);

}

});

}

Функция animate имеет три аргумента, которые описывают анимацию:

1. duration – продолжительность анимации. Например, 1000.
2. timing(timeFraction) – функция расчёта времени, как CSS-свойство transition-timing-function, которая будет вычислять прогресс анимации (как ось y у кривой Безье) в зависимости от прошедшего времени (0 в начале, 1 в конце).

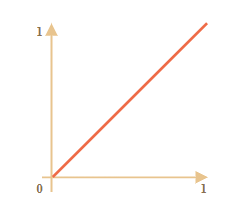
Например, линейная функция значит, что анимация идёт с одной и той же скоростью:

function linear(timeFraction) {

return timeFraction;

}

График функции:



Это похоже на значение linear для transition-timing-function.

1. draw(progress) – функция отрисовки, которая получает аргументом значение прогресса анимации и отрисовывает его. Значение progress=0 означает что анимация находится в начале, и значение progress=1 – в конце.

Эта та функция, которая рисует анимацию. Пример функции для сдвига элемента:

function draw(progress) {

train.style.left = progress + 'px';

}

В отличие от CSS-анимаций, можно создать любую функцию расчёта времени и любую функцию отрисовки. Функция расчёта времени не будет ограничена только кривой Безье, а функция draw может менять не только свойства, но и создавать новые элементы (например, для создания анимации фейерверка).

[**Функции расчёта времени**](https://learn.javascript.ru/js-animation#funktsii-raschyota-vremeni)

Самый простой пример линейной функции расчёта времени рассматривался выше. Рассмотрим другие.

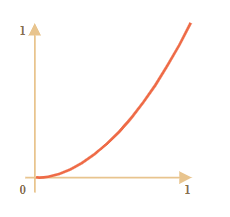
[*Степень n*](https://learn.javascript.ru/js-animation#stepen-n)*.* Если надо ускорить анимацию, то можно возвести progress в степень *n*. Например, параболическая кривая:

function quad(timeFraction) {

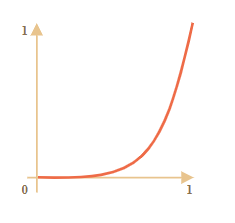
return Math.pow(timeFraction, 2)

}

График:



Повышение степени увеличивает скорость анимации. Вот график для функции progress в степени 5:



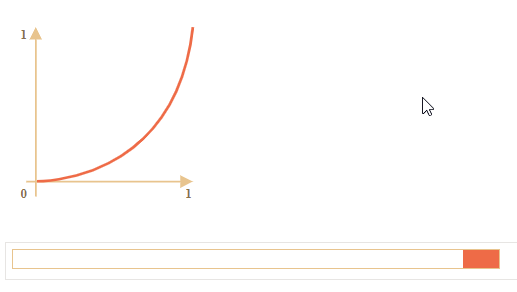
*[Дуга](https://learn.javascript.ru/js-animation" \l "duga)*. Функция:

function circ(timeFraction) {

return 1 - Math.sin(Math.acos(timeFraction));

}

График:



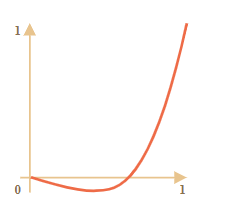
*[Обратно: выстрел из лука](https://learn.javascript.ru/js-animation" \l "obratno-vystrel-iz-luka)*. Эта функция совершает «выстрел из лука». В начале «натягивается тетива», а затем «выстрел». В отличие от предыдущей функции, теперь всё зависит от дополнительного параметра *x*– «коэффициента эластичности». Он определяет силу «натяжения тетивы». Код:

function back(x, timeFraction) {

return Math.pow(timeFraction, 2) \* ((x + 1) \* timeFraction - x)

}

График для *x* = 1.5:



*[Отскоки](https://learn.javascript.ru/js-animation" \l "otskoki)*. Представьте, что бросили мяч вниз. Он падает, ударяется о землю, подскакивает несколько раз и останавливается. Функции bounce делает то же самое, но в обратном порядке: «отскоки» начинаются сразу. Для этого заданы специальные коэффициенты:

function bounce(timeFraction) {

for (let a = 0, b = 1, result; 1; a += b, b /= 2) {

if (timeFraction >= (7 - 4 \* a) / 11) {

return -Math.pow((11 - 6 \* a - 11 \* timeFraction) / 4, 2) + Math.pow(b, 2)

}

}

}

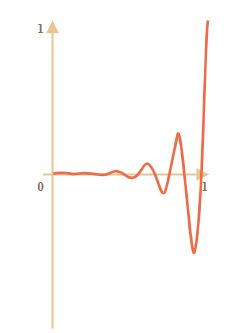
[*Эластичная анимация*](https://learn.javascript.ru/js-animation#elastichnaya-animatsiya). Ещё одна «эластичная» функция, которая принимает дополнительный параметр *x* для «начального отрезка».

function elastic(x, timeFraction) {

return Math.pow(2, 10 \* (timeFraction - 1)) \* Math.cos(20 \* Math.PI \* x / 3 \* timeFraction)

}

График для *x* = 1.5:



**[Реверсивные функции: ease\*](https://learn.javascript.ru/js-animation" \l "reversivnye-funktsii-ease)**

Прямое использование функций расчёта времени называется «easeIn». Иногда нужно показать анимацию в обратном режиме. Преобразование функции, которое даёт такой эффект, называется «easeOut».

В режиме «easeOut» timing функции оборачиваются функцией timingEaseOut:

timingEaseOut(timeFraction) = 1 - timing(1 - timeFraction)

Другими словами, имеется функция «преобразования» – makeEaseOut, которая берет «обычную» функцию расчёта времени и возвращает обёртку над ней (преобразованный вариант):

function makeEaseOut(timing) {

return function(timeFraction) {

return 1 - timing(1 - timeFraction);

}

}

Например, можно взять функцию bounce описанную выше:

let bounceEaseOut = makeEaseOut(bounce);

Таким образом, отскоки будут не в начале функции, а в конце:

function makeEaseOut(timing) {

return function(timeFraction) {

return 1 - timing(1 - timeFraction);

}

}

function bounce(timeFraction) {

for (let a = 0, b = 1, result; 1; a += b, b /= 2) {

if (timeFraction >= (7 - 4 \* a) / 11) {

return -Math.pow((11 - 6 \* a - 11 \* timeFraction) / 4, 2) + Math.pow(b, 2)

}

}

}

let bounceEaseOut = makeEaseOut(bounce);

brick.onclick = function() {

animate({

duration: 3000,

timing: bounceEaseOut,

draw: function(progress) {

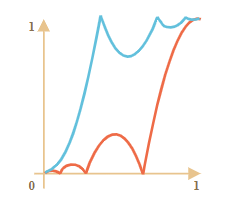
brick.style.left = progress \* 500 + 'px';

}

});

};

Ниже можно увидеть, как трансформации изменяют поведение функции:



Если раньше анимационный эффект, такой как отскоки, был в начале, то после трансформации он будет показан в конце. На графике выше красным цветом обозначена обычная функция и синим – после easeOut. Обычный скачок – объект сначала медленно скачет внизу, а затем резко подпрыгивает вверх. Обратный easeOut – объект вначале прыгает вверх, и затем скачет там.

Можно применить эффект дважды – в начале и конце анимации. Такая трансформация называется «easeInOut». Для функции расчёта времени, анимация будет вычисляться следующим образом:

if (timeFraction <= 0.5) { // первая половина анимации

return timing(2 \* timeFraction) / 2;

} else { // вторая половина анимации

return (2 - timing(2 \* (1 - timeFraction))) / 2;

}

Код функции-обёртки:

function makeEaseInOut(timing) {

return function(timeFraction) {

if (timeFraction < .5)

return timing(2 \* timeFraction) / 2;

else

return (2 - timing(2 \* (1 - timeFraction))) / 2;

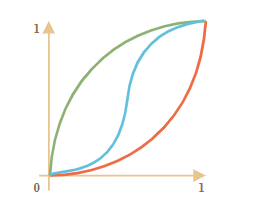
}

}

bounceEaseInOut = makeEaseInOut(bounce);

Функция «easeInOut» объединяет два графика в один: easeIn (обычный) для первой половины анимации and easeOut (обратный) – для второй половины.

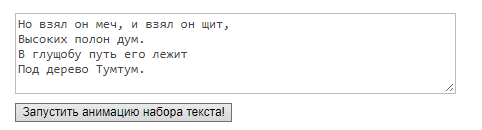
Разница хорошо заметна, если сравнивать графики easeIn, easeOut и easeInOut для функции circ. Красный – обычный вариант circ (easeIn), зелёный – easeOut, синий – easeInOut:



Как видно, график первой половины анимации представляет собой уменьшенный easeIn, а второй – уменьшенный easeOut. В результате, анимация начинается и заканчивается одинаковым эффектом.

Вместо передвижения элемента можно делать что-нибудь ещё. Всё, что нужно – это правильно написать функцию draw.

Вот пример «скачущей» анимации набирающегося текста:



// style.css

textarea {

display: block;

border: 1px solid #BBB;

color: #444;

font-size: 110%;

}

button {

margin-top: 10px;

}

// index.html

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

<link rel="stylesheet" href="style.css">

<script src="https://js.cx/libs/animate.js"></script>

</head>

<body>

<textarea id="textExample" rows="5" cols="60">Но взял он меч, и взял он щит,

Высоких полон дyм.

В глyщобy пyть его лежит

Под дерево Тyмтyм.

</textarea>

<button onclick="animateText(textExample)">Запустить анимацию набора текста!</button>

<script>

function animateText(textArea) {

let text = textArea.value;

let to = text.length,

from = 0;

animate({

duration: 5000,

timing: bounce,

draw: function(progress) {

let result = (to - from) \* progress + from;

textArea.value = text.substr(0, Math.ceil(result))

}

});

}

function bounce(timeFraction) {

for (let a = 0, b = 1, result; 1; a += b, b /= 2) {

if (timeFraction >= (7 - 4 \* a) / 11) {

return -Math.pow((11 - 6 \* a - 11 \* timeFraction) / 4, 2) + Math.pow(b, 2)

}

}

}

</script>

</body>

</html>