Практическая работа №14 «Действия браузера по умолчанию. Генерация пользовательских событий. Основы событий мыши. Предотвращение копирования. Drag'n'Drop. JavaScript-анимации»

Действия браузера по умолчанию

Многие события автоматически влекут за собой действие браузера.

Например:

- Клик по ссылке инициирует переход на новый URL.
- Нажатие на кнопку «отправить» в форме – отсылку её на сервер.
- Зажатие кнопки мыши над текстом и её движение в таком состоянии – инициирует его выделение.

Если мы обрабатываем событие в JavaScript, то зачастую такое действие браузера нам не нужно. К счастью, его можно отменить.

Отмена действия браузера

Есть два способа отменить действие браузера:

- Основной способ – это воспользоваться объектом event. Для отмены действия браузера существует стандартный метод event.preventDefault().
- Если же обработчик назначен через on<событие> (не через addEventListener), то также можно вернуть false из обработчика.

В следующем примере при клике по ссылке переход не произойдёт:

```
<a href="/" onclick="return false">Нажми здесь</a>
или
<a href="/" onclick="event.preventDefault()">здесь</a>

```

Возвращать true не нужно

Обычно значение, которое возвращает обработчик события, игнорируется. Единственное исключение – это return false из обработчика, назначенного

через on<событие>. В других случаях return не нужен, он никак не обрабатывается.

Пример: меню

Рассмотрим меню для сайта, например:

```
<ul id="menu" class="menu">
  <li><a href="/html">HTML</a></li>
  <li><a href="/javascript">JavaScript</a></li>
  <li><a href="/css">CSS</a></li>
</ul>
```

В HTML-разметке все элементы меню являются не кнопками, а ссылками, то есть тегами <a>. В этом подходе есть некоторые преимущества, например:

- Некоторые посетители очень любят сочетание «правый клик – открыть в новом окне». Если мы будем использовать <button> или , то данное сочетание работать не будет.
- Поисковые движки переходят по ссылкам <a href="..."> при индексации.

Поэтому в разметке мы используем <a>. Но нам необходимо обрабатывать клики в JavaScript, а стандартное действие браузера (переход по ссылке) – отменить.

Например, вот так:

```
menu.onclick = function(event) {
  if (event.target.nodeName != 'A') return;

  let href = event.target.getAttribute('href');
  alert( href ); // может быть подгрузка с сервера, генерация интерфейса и т.п.

  return false; // отменить действие браузера (переход по ссылке)
};

```

Если мы уберём return false, то после выполнения обработчика события браузер выполнит «действие по умолчанию» – переход по адресу из href. А это нам здесь не нужно, мы обрабатываем клик сами. Кстати, использование здесь

делегирования событий делает наше меню очень гибким. Мы можем добавить вложенные списки и стилизовать их с помощью CSS – обработчик не потребует изменений.

События, вытекающие из других

Некоторые события естественным образом вытекают друг из друга. Если мы отменим первое событие, то последующие не возникнут. Например, событие mousedown для поля <input> приводит к фокусировке на нём и запускает событие focus. Если мы отменим событие mousedown, то фокусирования не произойдёт.

Действий браузера по умолчанию достаточно много:

- mousedown – начинает выделять текст (если двигать мышкой).
- click на <input type="checkbox"> – ставит или убирает галочку в input.
- submit – при нажатии на <input type="submit"> или при нажатии клавиши Enter в форме данные отправляются на сервер.
- keydown – при нажатии клавиши в поле ввода появляется символ.
- contextmenu – при правом клике показывается контекстное меню браузера.
- …и многие другие…

Все эти действия можно отменить, если мы хотим обработать событие исключительно при помощи JavaScript. Чтобы отменить действие браузера по умолчанию, используйте event.preventDefault() или return false. Второй метод работает, только если обработчик назначен через on<событие>.

Опция passive: true для addEventListener сообщает браузеру, что действие по умолчанию не будет отменено. Это очень полезно для некоторых событий на мобильных устройствах, таких как touchstart и touchmove, чтобы сообщить

браузеру, что он не должен ожидать выполнения всех обработчиков, а ему следует сразу приступать к выполнению действия по умолчанию, например, к прокрутке.

Если событие по умолчанию отменено, то значение event.defaultPrevented становится true, иначе false.

Сохраняйте семантику, не злоупотребляйте

Технически, отменяя действия браузера по умолчанию и добавляя JavaScript, мы можем настроить поведение любого элемента. Например, мы можем заставить ссылку <a> работать как кнопку, а кнопку <button> вести себя как ссылка (перенаправлять на другой URL). Но нам следует сохранять семантическое значение HTML элементов. Например, не кнопки, а тег <a> должен применяться для переходов по ссылкам.

Помимо того, что это «хорошо», это делает ваш HTML лучше с точки зрения доступности для людей с ограниченными возможностями и с особых устройств. Также, если мы рассматриваем пример с тегом <a>, то обратите внимание: браузер предоставляет возможность открывать ссылки в новом окне (кликая правой кнопкой мыши или используя другие возможности). И пользователям это нравится. Но если мы заменим ссылку кнопкой и стилизуем её как ссылку, используя CSS, то специфичные функции браузера для тега <a> всё равно работать не будут.

Генерация пользовательских событий

Можно не только назначать обработчики, но и генерировать события из JavaScript-кода. Пользовательские события могут быть использованы при создании графических компонентов. Например, корневой элемент нашего меню, реализованного при помощи JavaScript, может генерировать события, относящиеся к этому меню: open (меню раскрыто), select (выбран пункт меню)

и т.п. А другой код может слушать эти события и узнавать, что происходит с меню.

Можно генерировать не только совершенно новые, придуманные нами события, но и встроенные, такие как click, mousedown и другие. Это бывает полезно для автоматического тестирования.

Конструктор Event

Встроенные классы для событий формируют иерархию аналогично классам для DOM-элементов. Её корнем является встроенный класс Event. Событие встроенного класса Event можно создать так:

```
let event = new Event(type[, options]);
```

Где:

- *type* – тип события, строка, например "click" или же любой придуманный нами – "my-event".
- *options* – объект с тремя необязательными свойствами:
    - bubbles: true/false – если true, тогда событие всплывает.
    - cancelable: true/false – если true, тогда можно отменить действие по умолчанию.
    - composed: true/false – если true, тогда событие будет всплывать наружу за пределы Shadow DOM.

По умолчанию все три свойства установлены в **false**: {bubbles: false, cancelable: false, composed: false}.

Метод dispatchEvent

После того, как объект события создан, мы должны запустить его на элементе, вызвав метод elem.dispatchEvent(event). Затем обработчики отреагируют на него, как будто это обычное браузерное событие. Если при создании указан

флаг bubbles, то оно будет всплывать. В примере ниже событие click инициируется JavaScript-кодом так, как будто кликнули по кнопке:

```
<button id="elem" onclick="alert('Клик!');">Автоклик</button>

<script>
  let event = new Event("click");
  elem.dispatchEvent(event);
</script>
```

event.isTrusted

Можно легко отличить «настоящее» событие от сгенерированного кодом. Свойство event.isTrusted принимает значение true для событий, порождаемых реальными действиями пользователя, и false для генерируемых кодом.

Пример всплытия

Мы можем создать всплывающее событие с именем "hello" и поймать его на document. Всё, что нужно сделать – это установить флаг bubbles в true:

```
<h1 id="elem">Привет из кода!</h1>

<script>
  // ловим на document...
  document.addEventListener("hello", function(event) { // (1)
    alert("Привет от " + event.target.tagName); // Привет от H1
  });

  // ...запуск события на элементе!
  let event = new Event("hello", {bubbles: true}); // (2)
  elem.dispatchEvent(event);

  // обработчик на document сработает и выведет сообщение.

</script>
```

Обратите внимание:

1. Мы должны использовать addEventListener для наших собственных событий, т.к. on<event>-свойства существуют только для встроенных событий, то есть document.onhello не сработает.
2. Мы обязаны передать флаг bubbles:true, иначе наше событие не будет всплывать.

Механизм всплытия идентичен как для встроенного события (click), так и для пользовательского события (hello). Также одинакова работа фаз всплытия и погружения.

MouseEvent, KeyboardEvent и другие

Для некоторых конкретных типов событий есть свои специфические конструкторы. Вот небольшой список конструкторов для различных событий пользовательского интерфейса, которые можно найти в спецификации UI Event:

- UIEvent
- FocusEvent
- MouseEvent
- WheelEvent
- KeyboardEvent
- …

Стоит использовать их вместо new Event, если мы хотим создавать такие события. К примеру, new MouseEvent("click"). Специфический конструктор позволяет указать стандартные свойства для данного типа события. Например, clientX/clientY для события мыши:

```
let event = new MouseEvent("click", {
  bubbles: true,
  cancelable: true,
  clientX: 100,
  clientY: 100
});

alert(event.clientX); // 100
```

Обратите внимание: этого нельзя было бы сделать с обычным конструктором Event. Впрочем, использование конкретного конструктора не является обязательным, можно обойтись Event, а свойства записать в объект отдельно, после создания, вот так: event.clientX=100. Здесь это скорее вопрос удобства и желания следовать правилам. События, которые генерирует браузер, всегда имеют правильный тип. Полный список свойств по типам событий вы найдёте в спецификации, например, MouseEvent.

Пользовательские события

Для генерации событий совершенно новых типов, таких как "hello", следует использовать конструктор new CustomEvent. Технически CustomEvent абсолютно идентичен Event за исключением одной небольшой детали. У второго аргумента-объекта есть дополнительное свойство detail, в котором можно указывать информацию для передачи в событие.

Например:

```
<h1 id="elem">Привет для Васи!</h1>

<script>
  // дополнительная информация приходит в обработчик вместе с событием
  elem.addEventListener("hello", function(event) {
    alert(event.detail.name);
  });

  elem.dispatchEvent(new CustomEvent("hello", {
    detail: { name: "Вася" }
  }));
</script>
```

Свойство detail может содержать любые данные. Надо сказать, что никто не мешает и в обычное new Event записать любые свойства. Но CustomEvent предоставляет специальное поле detail во избежание конфликтов с другими свойствами события. Кроме того, класс события описывает, что это за событие, и если оно не браузерное, а пользовательское, то лучше использовать CustomEvent, чтобы явно об этом сказать.

Вложенные события обрабатываются синхронно

Обычно события обрабатываются асинхронно. То есть, если браузер обрабатывает onclick и в процессе этого произойдёт новое событие, то оно ждёт, пока закончится обработка onclick. Исключением является ситуация, когда событие инициировано из обработчика другого события. Тогда управление сначала переходит в обработчик вложенного события и уже после этого возвращается назад. В примере ниже событие menu-open обрабатывается синхронно во время обработки onclick:

```
<button id="menu">Меню (нажми меня)</button>

<script>
  menu.onclick = function() {
    alert(1);

    // alert("Вложенное событие")
    menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    }));

    alert(2);
  };

  document.addEventListener('menu-open', () => alert('вложенное событие'))
</script>

```

Порядок вывода: 1 → вложенное событие → 2.

Обратите внимание, что вложенное событие menu-open успевает всплыть и запустить обработчик на document. Обработка вложенного события полностью завершается до того, как управление возвращается во внешний код

(onclick). Это справедливо не только для dispatchEvent, но и для других ситуаций. JavaScript в обработчике события может вызвать другие методы, которые приведут к другим событиям – они тоже обрабатываются синхронно.

Если нам это не подходит, то мы можем либо поместить dispatchEvent (или любой другой код, инициирующий события) в конец обработчика onclick, либо, если это неудобно, можно обернуть генерацию события в setTimeout с нулевой задержкой:

```
<button id="menu">Меню (нажми меня)</button>

<script>
  menu.onclick = function() {
    alert(1);

    // alert(2)
    setTimeout(() => menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    })));

    alert(2);
  };

  document.addEventListener('menu-open', () => alert('вложенное событие'))
</script>
```

Теперь dispatchEvent запускается асинхронно после исполнения текущего кода, включая mouse.onclick, поэтому обработчики полностью независимы.

Новый порядок вывода: 1 → 2 → вложенное событие.

Чтобы сгенерировать событие из кода, вначале надо создать объект события.

Базовый конструктор Event(name, options) принимает обязательное имя события и options – объект с двумя свойствами:

- bubbles: true чтобы событие всплывало.
- cancelable: true если мы хотим, чтобы event.preventDefault() работал.

Особые конструкторы встроенных событий MouseEvent, KeyboardEvent и другие принимают специфичные для каждого конкретного типа событий свойства. Например, clientX для событий мыши. Для пользовательских событий стоит применять конструктор CustomEvent. У него есть дополнительная опция detail, с помощью которой можно передавать информацию в объекте события. После чего все обработчики смогут получить к ней доступ через event.detail. Несмотря на техническую возможность генерировать встроенные браузерные события типа click или keydown, пользоваться ей стоит с большой осторожностью. Весьма часто, когда разработчик хочет сгенерировать встроенное событие – это вызвано «кривой» архитектурой кода. Как правило, генерация встроенных событий полезна в следующих случаях:

- Либо как явный и грубый хак, чтобы заставить работать сторонние библиотеки, в которых не предусмотрены другие средства взаимодействия.
- Либо для автоматического тестирования, чтобы скриптом «нажать на кнопку» и посмотреть, произошло ли нужное действие.

Пользовательские события со своими именами часто создают для улучшения архитектуры, чтобы сообщить о том, что происходит внутри наших меню, слайдеров, каруселей и т.д.

Основы событий мыши

Эти события бывают не только из-за мыши, но и эмулируются на других устройствах, в частности, на мобильных, для совместимости.

Типы событий мыши

mousedown/mouseup

Кнопка мыши нажата/отпущена над элементом.

mouseover/mouseout

Курсор мыши появляется над элементом и уходит с него.

mousemove

Каждое движение мыши над элементом генерирует это событие.

click

Вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши.

dblclick

Вызывается двойным кликом на элементе.

contextmenu

Вызывается при попытке открытия контекстного меню, как правило, нажатием правой кнопки мыши. Но, заметим, это не совсем событие мыши, оно может вызываться и специальной клавишей клавиатуры.

Порядок событий

Как вы можете видеть из приведённого выше списка, действие пользователя может вызвать несколько событий. Например, клик мышью вначале вызывает mousedown, когда кнопка нажата, затем mouseup и click, когда она отпущена. В случае, когда одно действие инициирует несколько событий, порядок их выполнения фиксирован. То есть обработчики событий вызываются в следующем порядке: mousedown → mouseup → click.

Кнопки мыши

События, связанные с кликом, всегда имеют свойство button, которое позволяет получить конкретную кнопку мыши. Обычно мы не используем его для событий click и contextmenu, потому что первое происходит только при

щелчке левой кнопкой мыши, а второе – только при щелчке правой кнопкой мыши. С другой стороны, обработчикам mousedown и mouseup может потребоваться event.button, потому что эти события срабатывают на любую кнопку, таким образом button позволяет различать «нажатие правой кнопки» и «нажатие левой кнопки».

Возможными значениями event.button являются:

Состояние кнопки	**event.button**
Левая кнопка (основная)	0
Средняя кнопка (вспомогательная)	1
Правая кнопка (вторичная)	2
Кнопка X1 (назад)	3
Кнопка X2 (вперёд)	4

Большинство мышек имеют только левую и правую кнопку, поэтому возможные значения это 0 или 2. Сенсорные устройства также генерируют аналогичные события, когда кто-то нажимает на них.

Также есть свойство event.buttons, в котором все нажатые в данный момент кнопки представлены в виде целого числа, по одному биту на кнопку. На практике это свойство используется очень редко, вы можете найти подробную информацию по адресу MDN, если вам это когда-нибудь понадобится.

Устаревшее свойство event.which

В старом коде вы можете встретить event.which свойство – это старый нестандартный способ получения кнопки с возможными значениями:

- event.which == 1 – левая кнопка,
- event.which == 2 – средняя кнопка,
- event.which == 3 – правая кнопка.

На данный момент event.which устарел, нам не следует его использовать.

Средняя кнопка сейчас – скорее экзотика, и используется очень редко.

Модификаторы: shift, alt, ctrl и meta

Все события мыши включают в себя информацию о нажатых клавишах-модификаторах.

Свойства события:

- shiftKey: `Shift`
- altKey: `Alt` (или `Opt` для Mac)
- ctrlKey: `Ctrl`
- metaKey: `Cmd` для Mac

Они равны true, если во время события была нажата соответствующая клавиша. Например, кнопка внизу работает только при комбинации `Alt+Shift`+клик:

```
<button id="button">Нажми Alt+Shift+Click на мне!</button>

<script>
  button.onclick = function(event) {
    if (event.altKey && event.shiftKey) {
      alert('Ура!');
    }
  };
</script>
```

В Windows и Linux клавишами-модификаторами являются `Alt`, `Shift` и `Ctrl`. На Mac есть ещё одна: `Cmd`, которой соответствует свойство metaKey. В большинстве приложений, когда в Windows/Linux используется `Ctrl`, на Mac используется `Cmd`. То есть, когда пользователь Windows

нажимает Ctrl+Enter и Ctrl+A, пользователь Mac нажимает Cmd+Enter или Cmd+A, и так далее.

Поэтому, если мы хотим поддерживать такие комбинации, как Ctrl+клик, то для Mac имеет смысл использовать Cmd+клик. Это удобней для пользователей Mac. Даже если мы и хотели бы заставить людей на Mac использовать именно Ctrl+клик, это довольно сложно. Проблема в том, что левый клик в сочетании с Ctrl интерпретируется как *правый клик* на MacOS и генерирует событие contextmenu, а не click как на Windows/Linux.

Поэтому, если мы хотим, чтобы пользователям всех операционных систем было удобно, то вместе с ctrlKey нам нужно проверять metaKey. Для JS-кода это означает, что мы должны проверить if (event.ctrlKey || event.metaKey).

Не забывайте про мобильные устройства

Комбинации клавиш хороши в качестве дополнения к рабочему процессу. Так что, если посетитель использует клавиатуру – они работают. Но если на их устройстве его нет – тогда должен быть способ жить без клавиш-модификаторов.

Координаты: clientX/Y, pageX/Y

Все события мыши имеют координаты двух видов:

1. Относительно окна: clientX и clientY.
2. Относительно документа: pageX и pageY.

Если у нас есть окно размером 500x500, и курсор мыши находится в левом верхнем углу, то значения clientX и clientY равны 0, независимо от того, как прокручивается страница. А если мышь находится в центре окна, то значения clientX и clientY равны 250 независимо от того, в каком месте

документа она находится и до какого места документ прокручен. В этом они похожи на position:fixed.

Отключаем выделение

Двойной клик мыши имеет побочный эффект, который может быть неудобен в некоторых интерфейсах: он выделяет текст. Например, двойной клик на текст ниже выделяет его в дополнение к нашему обработчику:

```
<span ondblclick="alert('dblclick')">Сделайте двойной клик на мне</span>
```

Если зажать левую кнопку мыши и, не отпуская кнопку, провести мышью, то также будет выделение, которое в интерфейсах может быть «не кстати». Есть несколько способов запретить выделение. В данном случае самым разумным будет отменить действие браузера по умолчанию при событии mousedown, это отменит оба этих выделения.

Предотвращение копирования

Если мы хотим отключить выделение для защиты содержимого страницы от копирования, то мы можем использовать другое событие: oncopy.

```
<div oncopy="alert('Копирование запрещено!');return false">
  Уважаемый пользователь,
  Копирование информации запрещено для вас.
  Если вы знаете JS или HTML, вы можете найти всю нужную вам информацию в исходном коде страницы.
</div>

```

Если вы попытаетесь скопировать текст в , у вас это не получится, потому что срабатывание события oncopy по умолчанию запрещено. Конечно, пользователь имеет доступ к HTML-коду страницы и может взять текст оттуда, но не все знают, как это сделать.

События мыши имеют следующие свойства:

- Кнопка: button.

- Клавиши-модификаторы (true если нажаты): altKey, ctrlKey, shiftKey и metaKey (Mac).
    - Если вы планируете обработать `Ctrl`, то не забудьте, что пользователи Mac обычно используют `Cmd`, поэтому лучше проверить if (e.metaKey || e.ctrlKey).
- Координаты относительно окна: clientX/clientY.
- Координаты относительно документа: pageX/pageY.

Действие по умолчанию события mousedown – начало выделения, если в интерфейсе оно скорее мешает, его можно отменить.

Drag'n'Drop с событиями мыши

Drag’n’Drop – отличный способ улучшить интерфейс. Захват элемента мышкой и его перенос визуально упростят что угодно: от копирования и перемещения документов (как в файловых менеджерах) до оформления заказа («положить в корзину»).

В современном стандарте HTML5 есть раздел о Drag and Drop – и там есть специальные события именно для Drag’n’Drop переноса, такие как dragstart, dragend и так далее.

Они интересны тем, что позволяют легко решать простые задачи. Например, можно перетащить файл в браузер, так что JS получит доступ к его содержимому.

Но у них есть и ограничения. Например, нельзя организовать перенос «только по горизонтали» или «только по вертикали». Также нельзя ограничить перенос внутри заданной зоны. Есть и другие интерфейсные задачи, которые такими встроенными событиями не реализуемы. Кроме того, мобильные устройства плохо их поддерживают.

Алгоритм Drag’n’Drop

Базовый алгоритм Drag’n’Drop выглядит так:

1. При mousedown – готовим элемент к перемещению, если необходимо (например, создаём его копию).
2. Затем при mousemove передвигаем элемент на новые координаты путём смены left/top и position:absolute.
3. При mouseup – остановить перенос элемента и произвести все действия, связанные с окончанием Drag’n’Drop.

Это и есть основа Drag’n’Drop. Позже мы сможем расширить этот алгоритм, например, подсветив элементы при наведении на них мыши.

В следующем примере эти шаги реализованы для переноса мяча:

```
ball.onmousedown = function(event) { // (1) отследить нажатие

  // (2) подготовить к перемещению:
  // разместить поверх остального содержимого и в абсолютных координатах
  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;
  // переместим в body, чтобы мяч был точно не внутри position:relative
  document.body.append(ball);
  // и установим абсолютно спозиционированный мяч под курсор

  moveAt(event.pageX, event.pageY);

  // передвинуть мяч под координаты курсора
  // и сдвинуть на половину ширины/высоты для центрирования
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
    ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
  }

  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }

  // (3) перемещать по экрану
  document.addEventListener('mousemove', onMouseMove);

  // (4) положить мяч, удалить более ненужные обработчики событий
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };

};
```

Если запустить этот код, то мы заметим нечто странное. При начале переноса мяч «раздваивается» и переносится не сам мяч, а его «клон». Всё потому, что браузер имеет свой собственный Drag’n’Drop, который автоматически запускается и вступает в конфликт с нашим. Это происходит именно для картинок и некоторых других элементов. Его нужно отключить:

```
ball.ondragstart = function() {
  return false;
};
```

Теперь всё будет в порядке.

Ещё одна деталь – событие mousemove отслеживается на document, а не на ball. С первого взгляда кажется, что мышь всегда над мячом и обработчик mousemove можно повесить на сам мяч, а не на документ. Но, как мы помним, событие mousemove возникает хоть и часто, но не для каждого пикселя. Поэтому из-за быстрого движения указатель может спрыгнуть с мяча и оказаться где-нибудь в середине документа (или даже за пределами окна). Вот почему мы должны отслеживать mousemove на всём document, чтобы поймать его.

Правильное позиционирование

В примерах выше мяч позиционируется так, что его центр оказывается под указателем мыши:

```
ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
```

Неплохо, но есть побочные эффекты. Мы, для начала переноса, можем нажать мышью на любом месте мяча. Если мячик «взят» за самый край – то в начале переноса он резко «прыгает», центрируясь под указателем мыши. Было бы лучше, если бы изначальный сдвиг курсора относительно элемента сохранялся. Где захватили, за ту «часть элемента» и переносим:

Обновим наш алгоритм:

1. Когда человек нажимает на мячик (mousedown) – запомним расстояние от курсора до левого верхнего угла шара в переменных shiftX/shiftY. Далее будем удерживать это расстояние при перетаскивании.

Чтобы получить этот сдвиг, мы можем вычесть координаты:

```
// onmousedown
let shiftX = event.clientX - ball.getBoundingClientRect().left;
let shiftY = event.clientY - ball.getBoundingClientRect().top;

```

2. Далее при переносе мяча мы позиционируем его с тем же сдвигом относительно указателя мыши, вот так:

```
// onmousemove
// ball has position:absoute
ball.style.left = event.pageX - shiftX + 'px';
ball.style.top = event.pageY - shiftY + 'px';

```

Итоговый код с правильным позиционированием:

```
ball.onmousedown = function(event) {

  let shiftX = event.clientX - ball.getBoundingClientRect().left;
  let shiftY = event.clientY - ball.getBoundingClientRect().top;

  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;
  document.body.append(ball);

  moveAt(event.pageX, event.pageY);

  // переносит мяч на координаты (pageX, pageY),
  // дополнительно учитывая изначальный сдвиг относительно указателя мыши
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - shiftX + 'px';
    ball.style.top = pageY - shiftY + 'px';
  }

  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }

  // передвигаем мяч при событии mousemove
  document.addEventListener('mousemove', onMouseMove);

  // отпустить мяч, удалить ненужные обработчики
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };

};

ball.ondragstart = function() {
  return false;
};

```

Различие особенно заметно, если захватить мяч за правый нижний угол. В предыдущем примере мячик «прыгнет» серединой под курсор, в этом – будет плавно переноситься с текущей позиции.

Цели переноса (droppable)

В предыдущих примерах мяч можно было бросить просто где угодно в пределах окна. В реальности мы обычно берём один элемент и перетаскиваем в другой. Например, «файл» в «папку» или что-то ещё. Абстрактно говоря, мы

берём перетаскиваемый (draggable) элемент и помещаем его в другой элемент «цель переноса» (droppable).

Нам нужно знать:

- куда пользователь положил элемент в конце переноса, чтобы обработать его окончание
- и, желательно, над какой потенциальной целью (элемент, куда можно положить, например, изображение папки) он находится в процессе переноса, чтобы подсветить её.

Решение довольно интересное и немного хитрое, давайте рассмотрим его. Возможно, установить обработчики событий mouseover/mouseup на элемент – потенциальную цель переноса может сработать, однако это не работает.

Проблема в том, что при перемещении перетаскиваемый элемент всегда находится поверх других элементов. А события мыши срабатывают только на верхнем элементе, но не на нижнем. Например, у нас есть два элемента : красный поверх синего (полностью перекрывает). Не получится поймать событие на синем, потому что красный сверху:

```
<style>
  div {
    width: 50px;
    height: 50px;
    position: absolute;
    top: 0;
  }
</style>
<div style="background:blue" onmouseover="alert('никогда не сработает')"></div>
<div style="background:red" onmouseover="alert('над красным!')"></div>
```

То же самое с перетаскиваемым элементом. Мяч всегда находится поверх других элементов, поэтому события срабатывают на нём. Какие бы обработчики мы ни ставили на нижние элементы, они не будут выполнены. Вот почему первоначальная идея поставить обработчики на потенциальные цели переноса нереализуема. Обработчики не сработают.

Существует метод document.elementFromPoint(clientX, clientY). Он возвращает наиболее глубоко вложенный элемент по заданным координатам окна (или null, если указанные координаты находятся за пределами окна). Мы можем использовать его, чтобы из любого обработчика событий мыши выяснить, над какой мы потенциальной целью переноса, вот так:

```
// внутри обработчика события мыши
ball.hidden = true; // (*) прячем переносимый элемент

let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
// elemBelow - элемент под мячом (возможная цель переноса)

ball.hidden = false;
```

Заметим, нам нужно спрятать мяч перед вызовом функции (*). В противном случае по этим координатам мы будем получать мяч, ведь это и есть элемент непосредственно под указателем: elemBelow=ball. Так что мы прячем его и тут же показываем обратно. Мы можем использовать этот код для проверки того, над каким элементом мы «летим», в любое время. И обработать окончание переноса, когда оно случится. Расширенный код onMouseMove с поиском потенциальных целей переноса:

```
// потенциальная цель переноса, над которой мы пролетаем прямо сейчас
let currentDroppable = null;

function onMouseMove(event) {
  moveAt(event.pageX, event.pageY);

  ball.hidden = true;
  let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
  ball.hidden = false;

  // событие mousemove может произойти и когда указатель за пределами окна
  // (мяч перетащили за пределы экрана)

  // если clientX/clientY за пределами окна, elementFromPoint вернёт null
  if (!elemBelow) return;

  // потенциальные цели переноса помечены классом droppable (может быть и другая логика)
  let droppableBelow = elemBelow.closest('.droppable');

  if (currentDroppable != droppableBelow) {
    // мы либо залетаем на цель, либо улетаем из неё
    // внимание: оба значения могут быть null
    //   currentDroppable=null,
    //     если мы были не над droppable до этого события (например, над пустым пространством)
    //   droppableBelow=null,
    //     если мы не над droppable именно сейчас, во время этого события

    if (currentDroppable) {
      // логика обработки процесса "вылета" из droppable (удаляем подсветку)
      leaveDroppable(currentDroppable);
    }
    currentDroppable = droppableBelow;
    if (currentDroppable) {
      // логика обработки процесса, когда мы "влетаем" в элемент droppable
      enterDroppable(currentDroppable);
    }
  }
}
```

В приведённом ниже примере, когда мяч перетаскивается через футбольные ворота, ворота подсвечиваются. Теперь в течение всего процесса в переменной currentDroppable мы храним текущую потенциальную цель переноса, над которой мы сейчас, можем её подсветить или сделать что-то ещё.

Мы рассмотрели основной алгоритм Drag’n’Drop.

Ключевые идеи:

1. Поток
   событий: ball.mousedown → document.mousemove → ball.mouseup (не забудьте отменить браузерный ondragstart).
2. В начале перетаскивания: запоминаем начальное смещение указателя относительно элемента: shiftX/shiftY – и сохраняем его при перетаскивании.
3. Выявляем потенциальные цели переноса под указателем с помощью document.elementFromPoint.

На этой основе можно сделать многое.

- На mouseup – по-разному завершать перенос: изменять данные, перемещать элементы.
- Можно подсвечивать элементы, пока мышь «пролетает» над ними.
- Можно ограничить перетаскивание определённой областью или направлением.
- Можно использовать делегирование событий для mousedown/up. Один обработчик событий на большой зоне, который проверяет event.target, может управлять Drag’n’Drop для сотен элементов.
- И так далее.

Существуют фреймворки, которые строят архитектуру поверх этого алгоритма, создавая такие классы, как DragZone, Droppable, Draggable. Большинство из них делают вещи, аналогичные описанным выше. Вы можете и сами создать вашу собственную реализацию переноса, как видите, это достаточно просто, возможно, проще, чем адаптация чего-то готового.

Кривые Безье

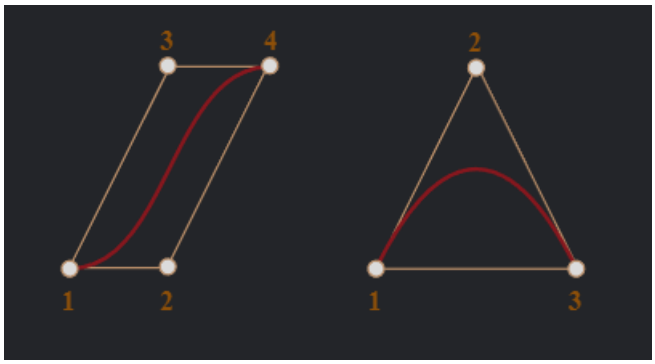
Кривые Безье используются в компьютерной графике для рисования плавных изгибов, в CSS-анимации и много где ещё. Это очень простая вещь, которую стоит изучить один раз, а затем чувствовать себя комфортно в мире векторной графики и продвинутых анимаций.

Опорные точки

Кривая Безье задаётся опорными точками. Их может быть две, три, четыре или больше. Например:

Если вы посмотрите внимательно на эти кривые, то «на глазок» заметите:

Точки не всегда на кривой. Это совершенно нормально. Степень кривой равна числу точек минус один. Для двух точек – это линейная кривая (т.е. прямая), для трёх точек – квадратическая кривая (парабола), для четырёх – кубическая. Кривая всегда находится внутри выпуклой оболочки, образованной опорными точками:



Благодаря последнему свойству в компьютерной графике можно оптимизировать проверку пересечения двух кривых. Если их выпуклые оболочки не пересекаются, то и кривые тоже не пересекутся. Таким образом, проверка пересечения выпуклых оболочек в первую очередь может дать быстрый ответ на вопрос о наличии пересечения. Проверить пересечение или выпуклые оболочки гораздо проще, потому что это прямоугольники, треугольники и т.д. (см. рисунок выше), гораздо более простые фигуры, чем кривая. Основная ценность кривых Безье для рисования в том, что, двигая точки, кривую можно менять, причём кривая при этом меняется интуитивно понятным образом.

JavaScript-анимации

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

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 мы будем 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.

Реализация этой анимации:

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

let timer = setInterval(function() {
  // сколько времени прошло с начала анимации?
  let timePassed = Date.now() - start;

  if (timePassed >= 2000) {
    clearInterval(timer); // закончить анимацию через 2 секунды
    return;
  }

  // отрисовать анимацию на момент timePassed, прошедший с начала анимации
  draw(timePassed);

}, 20);

// в то время как timePassed идёт от 0 до 2000
// left изменяет значение от 0px до 400px
function draw(timePassed) {
  train.style.left = timePassed / 5 + 'px';
}
```

Использование requestAnimationFrame

Теперь давайте представим, что у нас есть несколько анимаций, работающих одновременно. Если мы запустим 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 у нас будет несколько независимых срабатываний в течение 20ms. Эти независимые перерисовки лучше

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

Существует ещё одна вещь, про которую надо помнить: когда CPU перегружен или есть другие причины делать перерисовку реже (например, когда вкладка браузера скрыта), нам не следует делать её каждые 20ms.

Спецификация 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(). Как правило, callback запускается очень скоро, 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

Теперь мы можем 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 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;
}
```

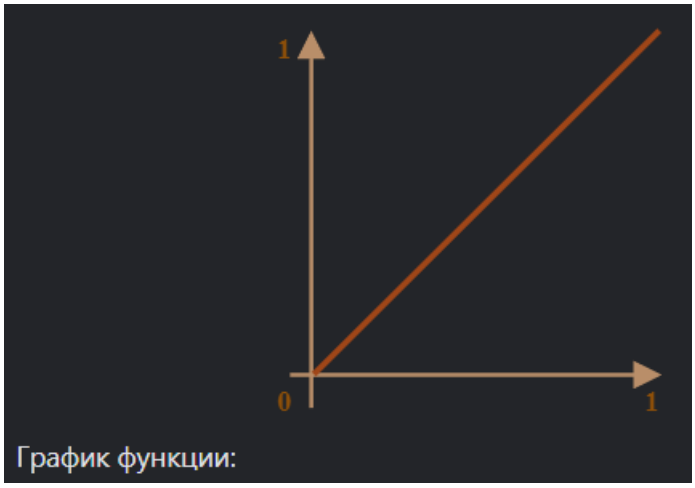


График функции:

Это как если бы в transition-timing-function передать значение linear.

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 на самом деле и рисует анимацию. Вот как она могла бы двигать элемент:

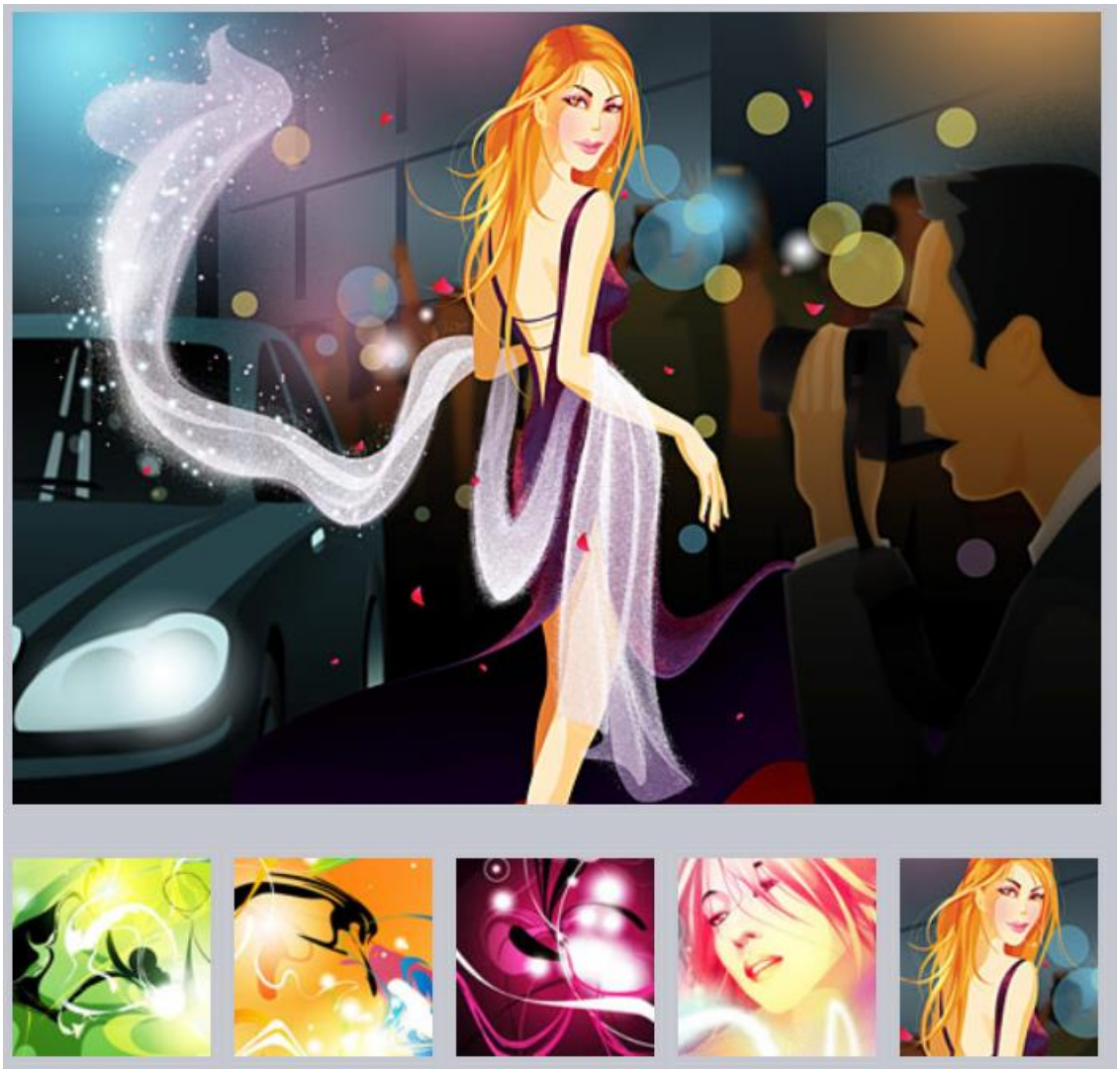
```
function draw(progress) {
  train.style.left = progress + 'px';
}
```

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).

Задачи к практической работе №14

1. Сделайте так, чтобы при клике на ссылки внутри элемента id="contents" пользователю выводился вопрос о том, действительно ли он хочет покинуть страницу, и если он не хочет, то прерывать переход по ссылке. (Содержимое #contents может быть загружено динамически и присвоено при помощи innerHTML. Так что найти все ссылки и поставить на них обработчики нельзя. Используйте делегирование. Содержимое может иметь вложенные теги, в том числе внутри ссылок, например, ...)

2. Создайте галерею изображений, в которой основное изображение изменяется при клике на уменьшенный вариант.



3. Создайте список, в котором элементы могут быть выделены, как в файловых менеджерах. При клике на элемент списка выделяется только этот элемент (добавляется класс .selected), отменяется выделение остальных элементов. Если клик сделан вместе с Ctrl (Cmd для Mac), то выделение переключается на элементе, но остальные элементы при этом не изменяются. (Предотвратите стандартное для браузера выделение текста при кликах.)

4. Создайте слайдер: 

Слайдер должен нормально работать при резком движении мыши влево или вправо за пределы полосы. При этом бегунок должен останавливаться чётко в нужном конце полосы. При нажатом бегунке мышь может выходить за пределы полосы слайдера, но слайдер пусть всё равно работает (это удобно для пользователя).

5. Добавьте возможность перетаскивать элементы (товары) в «корзину» сайта. При этом итоговая стоимость корзины должна изменяться в соответствии с ценой товара и количеством перенесенных элементов.

6. Создайте любую анимацию для сайта используя JavaScript. Разместите от двух различных по действиям анимаций так, чтобы это не мешало пользователю и аутентично смотрелось на сайте.