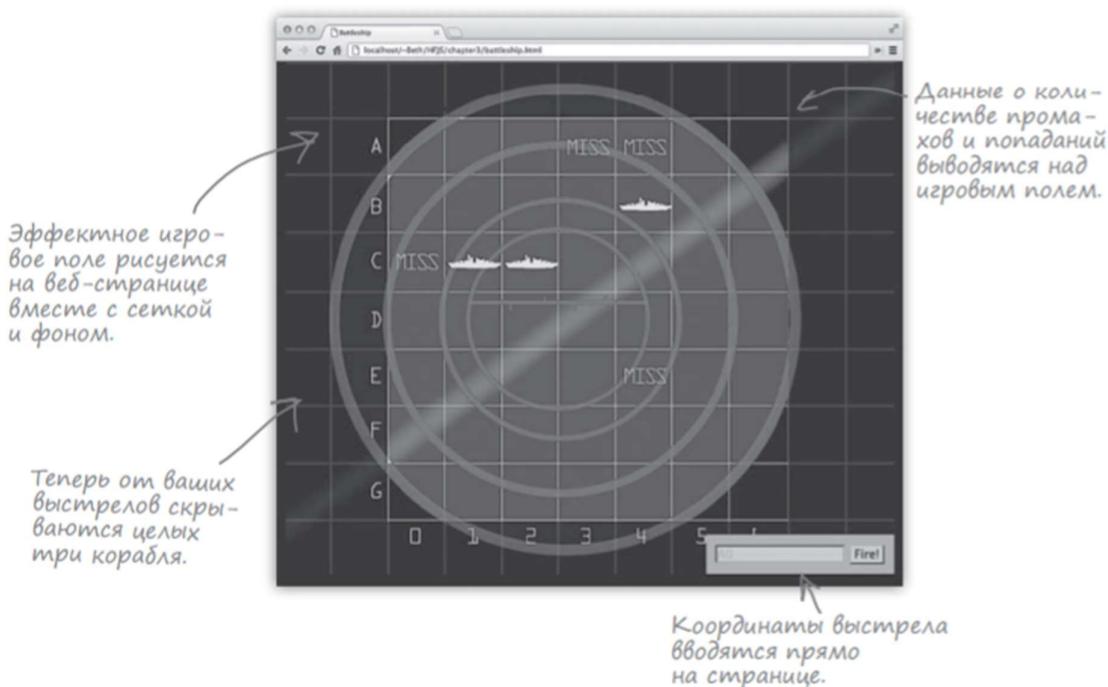


## Построение приложения

Подготовьте свой инструментарий к работе. Да, ваш инструментарий — ваши новые навыки программирования, ваше знание DOM и даже некоторое знание HTML и CSS. В этой главе мы объединим все это для создания своего первого полноценного веб-приложения. Довольно примитивных игр с одним кораблем, который размещается в одной строке. В этой главе мы построим полную версию: большое игровое поле, несколько кораблей, ввод данных пользователем прямо на веб-странице. Мы создадим структуру страницы игры в разметке HTML, применим визуальное оформление средствами CSS и напишем код JavaScript, определяющий поведение игры. Приготовьтесь: в этом уроке мы займемся полноценным, серьезным программированием и напишем вполне серьезный код.

На этот раз мы построим НАСТОЯЩУЮ игру «Морской бой»

Бесспорно, вы имеете полное право гордиться тем, что в главе 2 мы построили свой миниатюрный «Морской бой» с нуля, но давайте признаем: игра была немного ненастоящей — она работала, в нее можно было играть, но вряд ли такая программа может произвести впечатление на друзей или привлечь средства на развитие бизнеса. Чтобы продукт был действительно впечатляющим, нужно эффектное игровое поле, стильные изображения кораблей и возможность делать ходы прямо на игровом поле (вместо обобщенного диалогового окна в браузере). И вообще, хорошо бы улучшить предыдущую версию и ввести в нее поддержку всех трех кораблей. Другими словами, игра должна выглядеть примерно так:



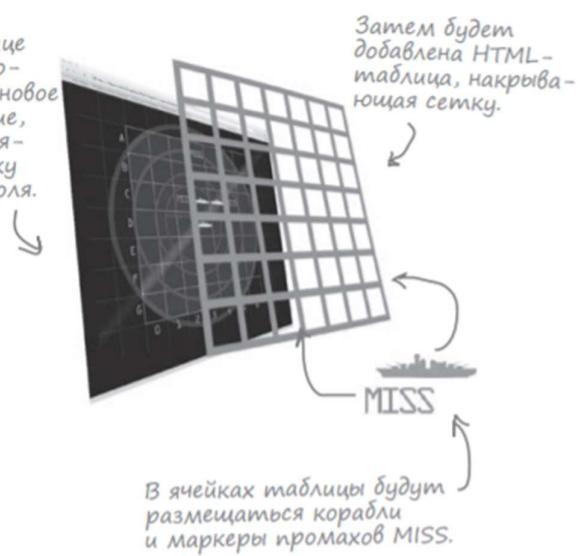
Возвращаемся к HTML и CSS

Чтобы создать современную интерактивную веб-страницу (или приложение), придется работать с тремя технологиями: HTML, CSS и JavaScript. Наверняка вы уже слышали поговорку: «HTML для структуры, CSS для стилевого оформления, JavaScript для поведения». Но мы не ограничимся простым повторением. В этой главе данный принцип будет в полной мере реализован. И начнем мы с HTML и CSS.

Наша первая задача — воспроизвести внешний вид игрового поля с предыдущей страницы. Впрочем, только воспроизвести недостаточно, нужно реализовать его так, чтобы его структура позволяла пользователю вводить данные и отображать попадания, промахи и сообщения прямо на странице.

Для этого нам, например, придется наложить фоновое изображение, чтобы имитировать экран радара. Потом на странице будет размещена более функциональная таблица HTML, в которой можно размещать корабли, маркеры попаданий и т. д. Наконец, мы воспользуемся формой HTML для получения введенных пользователем данных.

На странице будет выводиться фоновое изображение, представляющее сетку игрового поля.



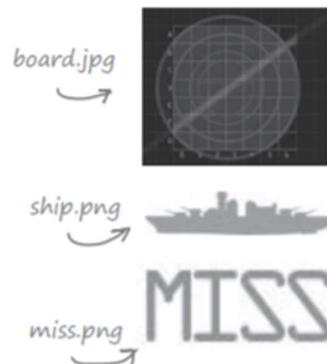
В ячейках таблицы будут размещаться корабли и маркеры промахов MISS.

Итак, займемся построением игры. На нескольких ближайших страницах мы будем заниматься основными компонентами HTML и CSS, а когда основа приложения будет построена, можно будет переходить к написанию кода JavaScript.

## ИНСТРУМЕНТЫ для создания приложения

Несколько готовых графических изображений помогут вам начать работу над новой версией игры.

*INVENTORY includes...*

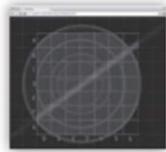


Пакет состоит из трех графических файлов: board.jpg — основной фон игрового поля, включая сетку; ship.png — небольшой кораблик для размещения на игровом поле (обратите внимание: это PNG-файл с прозрачностью, который может накладываться прямо поверх фона), и наконец, miss.png — маркер промаха, который также будет размещаться на поле. Как и положено в игре, при попадании в соответствующей клетке таблицы будет выводиться корабль, а при промахе — графический маркер промаха.

## Создание страницы HTML: общая картина

Основные пункты плана по созданию HTML-страницы:

- 1** Начнем с фона игрового поля; для этого будет выбран черный цвет фона, после чего на странице будет размещена радарная сетка.



Размещаем графику на заднем плане, чтобы страница смотрелась более эффектно и напоминала зеленый светящийся экран радара.

- 2** Затем мы создадим таблицу HTML и разместим ее поверх фона. Каждая ячейка таблицы будет соответствовать одной клетке игрового поля.

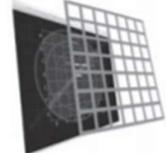
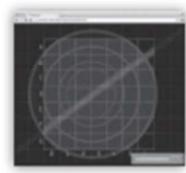


Таблица HTML у верхнего края фона образует игровое поле.

- 3** Затем мы добавим элемент формы HTML для ввода координат выстрелов, например "A4". Также будет добавлена область для вывода сообщений (например, «Ты потопил мой корабль!»).



Форма HTML для ввода данных игроком.

- 4** Остается разобраться с тем, как использовать таблицу для размещения графики кораблей (попадание) или маркера MISS (промах).



Эти изображения будут размещаться в таблице по ходу игры.

## Шаг 1: Базовая разметка HTML

За дело! Прежде всего нам потребуется страница HTML. Мы создадим простую страницу по стандарту HTML5, а также добавим стилевое оформление для фона. В страницу будет включен элемент `<body>` с одним вложенным элементом `<div>`. Элемент `<div>` будет содержать сетку игрового поля.

На следующей странице приведена исходная версия разметки HTML и CSS нашей страницы.

```
<!doctype html> ↗ Обычная страница HTML
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Battleship</title>
    <style>
      body {
        background-color: black;
      }

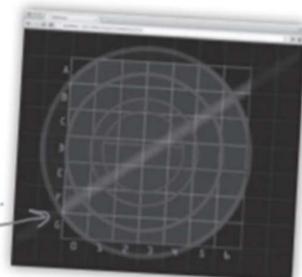
      div#board {
        position: relative;
        width: 1024px;
        height: 863px;
        margin: auto;
        background: url("board.jpg") no-repeat;
      }
    </style>
  </head>
  <body>
    <div id="board"> ↗ Здесь будет размещена таблица, представляющая игровое поле, и форма для получения пользовательского ввода.
    </div> ↗ Код будет размещен в файле battleship.js. Создайте для него пустой файл.
    <script src="battleship.js"></script>
  </body>
</html>
```

Страница будет иметь черный фон.

Игровое поле должно находиться в середине страницы, поэтому мы назначаем `width` значение `1024px` (ширина игрового поля) с автоматическим выбором величины полей.

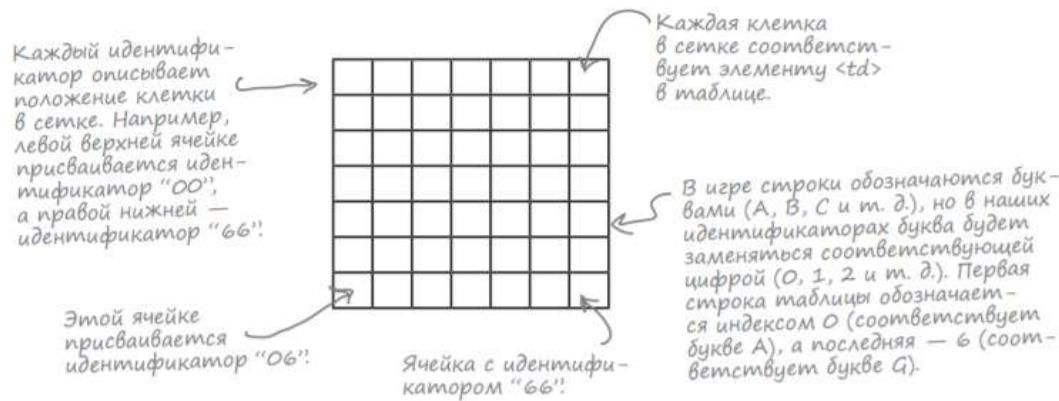
Здесь изображение `board.jpg` добавляется на страницу как фон элемента `<div>` "board". Для элемента `<div>` используется относительное позиционирование, чтобы мы могли разместить таблицу, которая будет добавлена на следующем шаге, относительно этого элемента `<div>`.

Так выглядит наша веб-страница на данный момент...



## Шаг 2: Создание таблицы

Следующий пункт – таблица. Она размещается поверх сетки фонового изображения и представляет место для размещения графики попаданий и промахов в ходе игры. Каждая ячейка (или если вы помните HTML – каждый элемент `<td>`) будет размещаться прямо поверх клетки фонового изображения. А теперь самое важное: мы присваиваем каждой ячейке уникальный идентификатор, чтобы позднее с ней можно было работать из CSS и JavaScript. Давайте посмотрим, как создать идентификаторы и добавить разметку HTML-таблицы:



Разметка HTML-таблицы. Добавьте ее между тегами `<div>`:

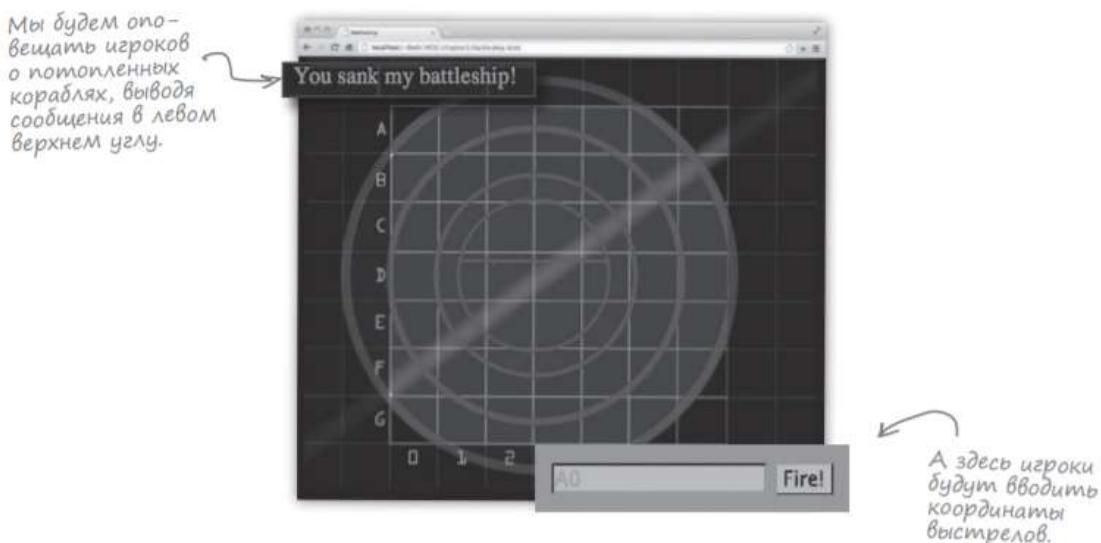
```
<div id="board"> ← Таблица вкладывается в элемент <div> "board".  
  <table>  
    <tr>  
      <td id="00"></td><td id="01"></td><td id="02"></td><td id="03"></td>  
      <td id="04"></td> <td id="05"></td><td id="06"></td>  
    </tr>  
    <tr>  
      <td id="10"></td><td id="11"></td><td id="12"></td><td id="13"></td>  
      <td id="14"></td> <td id="15"></td><td id="16"></td>  
    </tr>  
    ...  
    <tr>  
      <td id="60"></td><td id="61"></td><td id="62"></td><td id="63"></td>  
      <td id="64"></td><td id="65"></td><td id="66"></td>  
    </tr>  
  </table>  
</div>
```

Annotations on the right side of the code:

- An annotation points to the first `<td>` element in the first row and says: "Каждому элементу `<td>` присваивается идентификатор, определяемый номерами строки и столбца в таблице."
- An annotation points to the ellipsis (...) and says: "Мы опускаем часть строк для экономии бумаги. Не сомневайтесь, что вы сможете заполнить их самостоятельно."

## Шаг 3: Взаимодействие с игроком

В приложение нужно включить элемент HTML для ввода координат выстрелов (например, "A0" или "E4") и элемент для вывода сообщений (например, «Ты потопил мой корабль!»). Мы используем элемент `<form>` с полем `<input>` для ввода координат, а сообщения будут выводиться в элементе `<div>`:



```
<div id="board">  
  <div id="messageArea"></div>  
  <table>
```

Элемент `<div>` с идентификатором `messageArea` будет использоваться для вывода сообщений.

```
    ...  
  </table>  
  <form>  
    <input type="text" id="guessInput" placeholder="A0">  
    <input type="button" id="fireButton" value="Fire!">  
  </form>
```

Элемент `<form>` содержит два элемента `input`: текстовое поле для ввода выстрелов и кнопка. Обратите внимание на идентификаторы этих элементов. Они понадобятся нам позднее, когда мы будем писать код для ввода координат выстрелов.

```
</div>
```

Элемент `<div>` области сообщений, элементы `<table>` и `<form>` вкладываются в элемент `<div>` с идентификатором `"board"`. Это обстоятельство будет использовано в CSS на следующей странице.

## Добавление стилевого оформления

Если загрузить страницу сейчас (давайте, попробуйте), большинство элементов будет находиться в неподходящих местах с неправильными размерами. Следовательно, мы должны написать CSS для позиционирования элементов и проследить за тем, чтобы размеры всех элементов (например, ячеек таблиц) соответствовали размерам игрового поля.

Чтобы разместить элементы в правильных местах, необходимо использовать позиционирование CSS для формирования макета. Так как элемент `<div>` "board" использует относительное позиционирование, мы теперь можем разместить область сообщений, таблицу и форму в конкретных местах `<div>`, чтобы они отображались именно так, как нам нужно.

Начнем с элемента `<div>` "messageArea". Он вложен в элемент `<div>` "board" и должен размещаться в левом верхнем углу игрового поля:

```
body {  
    background-color: black;  
}  
  
div#board {  
    position: relative;  
    width: 1024px;  
    height: 863px;  
    margin: auto;  
    background: url("board.jpg") no-repeat;  
}  
  
div#messageArea {  
    position: absolute;  
    top: 0px;  
    left: 0px;  
    color: rgb(83, 175, 19);  
}
```

Элемент `<div>` "board" использует относительное позиционирование, а все вложенные элементы будут позиционироваться относительно этого элемента `<div>`.

Область сообщений размещается в левом верхнем углу игрового поля.

Элемент `<div>` области сообщений вложен в элемент `<div>` игрового поля, поэтому его позиция задается относительно последнего. Итак, он будет смещен на 0px по вертикали и на 0px по горизонтали относительно левого верхнего угла элемента `<div>` игрового поля.



Область сообщений должна находиться в левом верхнем углу игрового поля.



### КЛЮЧЕВЫЕ МОМЕНТЫ

- Свойство "position: relative" позиционирует элемент относительно его нормальной позиции в потоке страницы.
- Свойство "position: absolute" позиционирует элемент относительно позиции его ближайшего родителя.
- Свойства `top` и `left` задают величину смещения элемента (в пикселях) относительно его позиции по умолчанию.

Мы также можем разместить таблицу и форму в элементе `<div>` "board", снова используя абсолютное позиционирование для размещения этих элементов в точности там, где они должны

находиться. Остаток кода CSS выглядит так:

```
body {  
    background-color: black;  
}  
div#board {  
    position: relative;  
    width: 1024px;  
    height: 863px;  
    margin: auto;  
    background: url("board.jpg") no-repeat;  
}  
div#messageArea {  
    position: absolute;  
    top: 0px;  
    left: 0px;  
    color: rgb(83, 175, 19);  
}  
table {  
    position: absolute;  
    left: 173px;  
    top: 98px;  
    border-spacing: 0px;  
}  
td {  
    width: 94px;  
    height: 94px;  
}  
form {  
    position: absolute;  
    bottom: 0px;  
    right: 0px;  
    padding: 15px;  
    background-color: rgb(83, 175, 19);  
}  
form input {  
    background-color: rgb(152, 207, 113);  
    border-color: rgb(83, 175, 19);  
    font-size: 1em;  
}
```

Элементом <table> смещается на 173 пикселя по горизонтали от левого края и на 98 пикселов по вертикали от верхнего края игрового поля, поэтому таблица выравнивается по сетке фонового изображения.

Каждый элемент <td> имеет четко определенную ширину и высоту, так что ячейки таблицы выравниваются по клемкам сетки.

Элементом <form> будет отображаться в правом нижнем углу игрового поля. Он немного скрывает числа внизу справа, но это не страшно (вы все равно их знаете). Также элементу <form> назначается приятный зеленый цвет, соответствующий цвету фонового изображения.

Наконец, к двум элементам <input> применяются стили, чтобы они соответствовали оформлению игрового поля, — и работа закончена!

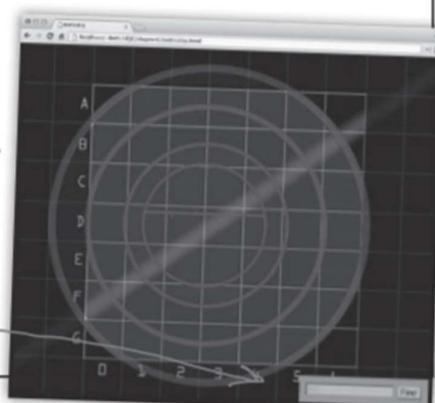


## Тест-драйв

Мы приближаемся к очередной контрольной точке. Соберите в файле HTML всю разметку HTML и CSS, после чего перезагрузите страницу в браузере. Вот что вы должны увидеть:

Таблица располагается прямо поверх сетки игрового поля (хотя на иллюстрации ее трудно рассмотреть, потому что она невидима).

Форма ввода готова к приему координат. Впрочем, с введенными данными ничего не произойдет, пока мы не напишем код.



## Шаг 4: Размещение меток промахов и попаданий

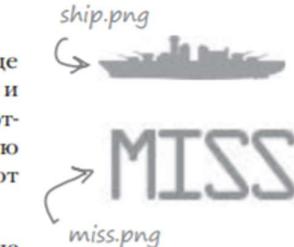
Игровое поле выглядит превосходно, вы не находите? Однако нам еще нужно придумать, как размещать на нем визуальные признаки промахов и попаданий — то есть как добавить изображение ship.png или miss.png в соответствующем месте. Сейчас мы только выясним, как написать правильную разметку или стиль для решения этой задачи, а позднее мы используем тот же прием в коде.

Итак, как же вывести на игровом поле графику ship.png или miss.png? Проще всего назначить нужное изображение фоном элемента `<td>` средствами CSS. Для этого мы создадим два класса с именами `hit` и `miss`. Мы воспользуемся свойством CSS `background`, чтобы элемент с классом “`hit`” использовал в качестве фона изображение `ship.png`, а элемент с классом “`miss`” — изображение `miss.png`. Это выглядит примерно так:

Если элемент относится к классу `hit`, ему назначается фоновое изображение `ship.png`. Если же элемент относится к классу `miss`,

```
.hit {  
    background: url("ship.png") no-repeat center center;  
}  
  
.miss {  
    background: url("miss.png") no-repeat center center;  
}
```

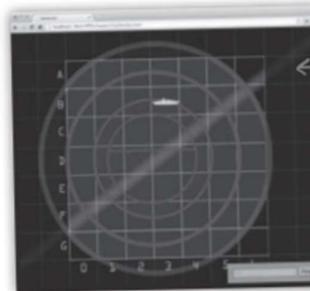
то ему назначается фоновое изображение `miss.png`.



Каждое правило CSS размещает в выбранном элементе одно изображение, выровненное по центру.

## Использование классов `hit` и `miss`

Добавьте определения классов `hit` и `miss` в CSS. Как мы собираемся использовать эти классы, спросите вы? Проведем небольшой эксперимент: допустим, корабль занимает клетки “B3”, “B4” и “B5”, пользователь стреляет в клетку B3 — попадание! Следовательно, нужно разместить в B3 изображение `ship.png`. Для этого сначала “B” преобразуется в число (A соответствует 0, B соответствует 1, и так далее), после чего в таблице находится элемент `<td>` с идентификатором “13”. В найденный элемент `<td>` добавляется класс `“hit”`:



В элементе `<td>` добавляется класс `“hit”`.

```
<tr>  
    <td id="10"></td> <td id="11"></td> <td id="12"></td> <td id="13" class="hit"></td>  
    <td id="14"></td> <td id="15"></td> <td id="16"></td>  
</tr>
```

Не забудьте добавить классы `hit` и `miss` с предыдущей страницы в CSS.

Теперь при перезагрузке страницы в клетке игрового поля “B3” выводится корабль.

То, что мы увидим при добавлении класса `“hit”` к элементу с идентификатором “13”:



## Учебные стрельбы

Прежде чем писать код размещения попаданий и промахов на игровом поле, давайте немного потренируемся в работе с CSS. Добавьте в свою разметку классы "hit" и "miss", соответствующие приведенным ниже действиям игрока. Обязательно проверьте свой ответ!

Корабль 1: A6, B6, C6

Корабль 2: C4, D4, E4

Корабль 3: B0, B1, B2

Напоминаем, что буквы  
нужно преобразовать  
в цифры: A = 0, ... G = 6.

А это выстрелы игрока:

A0, D4, F5, B2, C5, C6

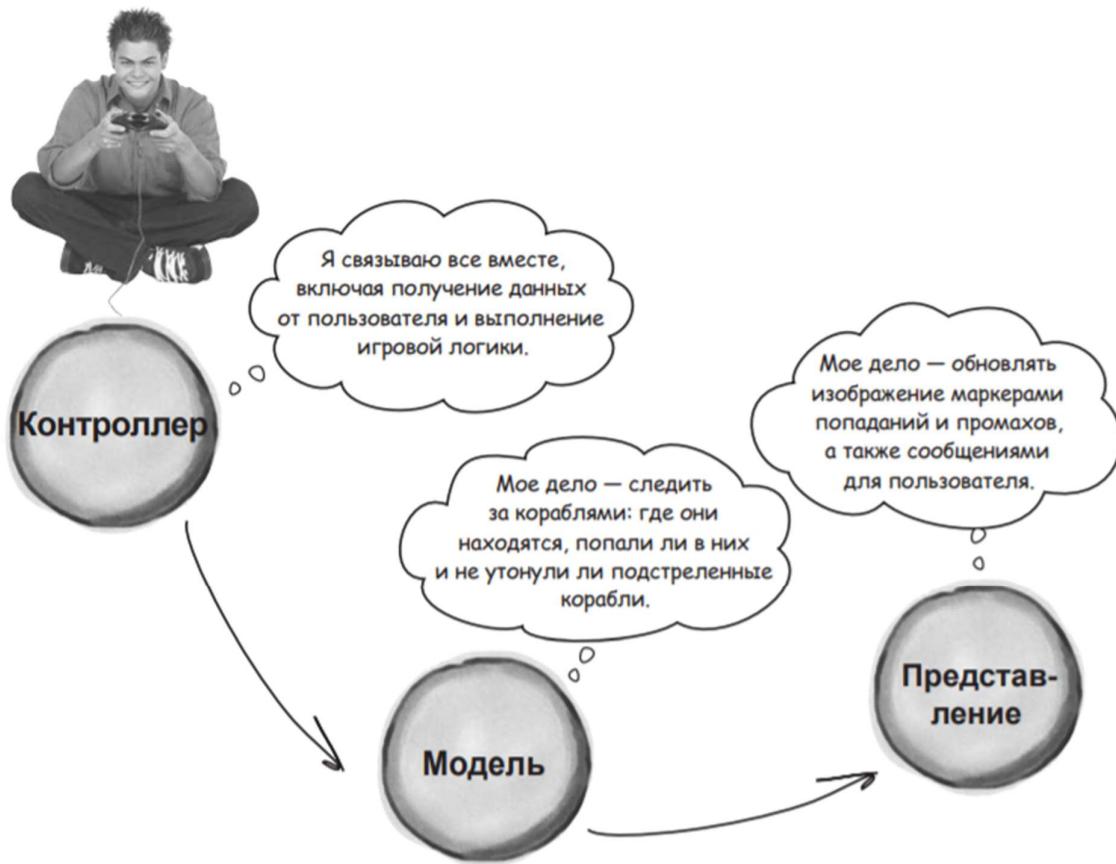
Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

Когда упражнение будет  
выполнено, удалите  
все классы, добавленные  
к элементам `<td>`. Когда  
мы перейдем к програм-  
мированию, игровое поле  
должно быть пустым.

### Как спроектировать игру

Разобравшись с HTML и CSS, вернемся к проектированию игры. В 3 уроке мы не рассматривали функции, объекты и инкапсуляцию и еще не имели дела с объектно-ориентированным проектированием, поэтому при построении первой версии игры «Морской бой» использовался процедурный подход — игра проектировалась как последовательность выполняемых действий, с логикой принятия решений и циклов. Тогда вы ничего не знали о DOM, поэтому игра была недостаточно интерактивной. В новой реализации игра будет представлять собой набор объектов, обладающих определенными обязанностями, а взаимодействие с пользователем будет осуществляться через DOM. Вы увидите, насколько этот подход упрощает задачу.

Начнем с объектов, которые нам предстоит спроектировать и реализовать. Таких объектов будет три: модель для хранения состояния игры (например, позиций кораблей и попаданий); представление, ответственное за обновление изображения; и контроллер, связывающий все воедино (обработка пользовательского ввода, обеспечение отработки игровой логики, проверка завершения игры и т. д.).



## Реализация представления

← А если нет — вам должно быть стыдно! Сделайте это сейчас!

Если вы проверили ответ к предыдущему упражнению, то вы заметили, что код представления разбит на три метода: `displayMessage`, `displayHit` и `displayMiss`. Учтите, что единственно правильного ответа не существует. Например, можно ограничиться всего двумя методами `displayMessage` и `displayPlayerGuess` и передать `displayPlayerGuess` аргумент, указывающий, попал или промахнулся игрок. Такая архитектура тоже разумна. Но мы пока будем придерживаться нашей архитектуры... поэтому давайте подумаем над реализацией первого метода — `displayMessage`:

Наш объект представления.

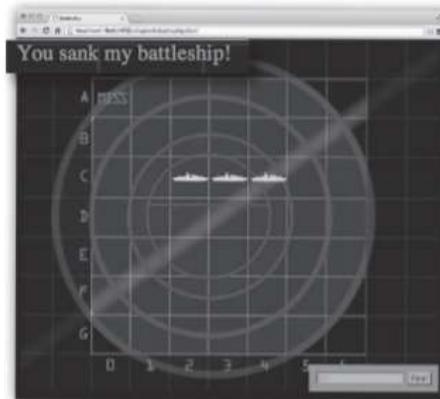
```
var view = {
    displayMessage: function(msg) {
        ← Начнем отсюда.
    },
    displayHit: function(location) {
    },
    displayMiss: function(location) {
    }
};
```

## Как работаем методом displayMessage

Чтобы реализовать метод `displayMessage`, необходимо проанализировать разметку HTML и убедиться в том, что в ней имеется элемент `<div>` с идентификатором “`messageArea`”, готовый к выводу сообщений:

```
<div id="board">
  <div id="messageArea"></div>
  ...
</div>
```

Мы используем DOM для получения доступа к этому элементу `<div>`, после чего задаем его текст при помощи свойства `innerHTML`. И помните, что при каждом изменении DOM изменения немедленно отражаются в браузере. Вот что мы сделаем...



## Реализация displayMessage

Возвращаемся к написанию кода `displayMessage`. Напомним, что этот код должен:

- Использовать DOM для получения элемента с идентификатором “`messageArea`”.
- Задавать свойству `innerHTML` элемента сообщение, переданное методу `displayMessage`.

Итак, откройте пустой файл `battleship.js` и добавьте объект `view`:

```
var view = {
  displayMessage: function(msg) {
    var messageArea = document.getElementById("messageArea");
    messageArea.innerHTML = msg;
  },
  displayHit: function(location) {
  },
  displayMiss: function(location) {
  }
};
```

Метод `displayMessage` получает один аргумент — текст сообщения.

Мы получаем элемент `messageArea` из страницы...

...и обновляем текст элемента `messageArea`, задавая его свойству `innerHTML` содержимое `msg`.

Прежде чем переходить к тестированию, напишите два других метода. В них нет ничего сложного, а мы зато сможем проверить сразу весь объект.

## Как работают методы displayHit и displayMiss

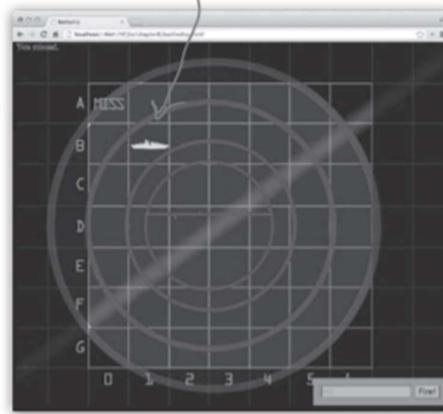
Вспомните, о чем говорилось совсем недавно: чтобы изображение появилось на игровом поле, нужно найти элемент `<td>` и добавить к нему класс "hit" или "miss". В первом случае в ячейке выводится содержимое `ship.png`, а во втором – содержимое `miss.png`.

Мы можем изменить изображение, добавляя к элементам `<td>` класс "hit" или "miss". Остается понять, как сделать это в программном коде.

```
<tr>
<td id="10"></td> <td class="hit" id="11"></td> <td id="12"></td> ...
</tr>
```

В коде мы воспользуемся DOM для получения элемента `<td>`, после чего зададим его атрибуту `class` значение "hit" или "miss" при помощи метода `setAttribute`. Как только атрибут изменится, соответствующее изображение появится в браузере. Итак, вот что мы собираемся сделать:

- Получить строковый идентификатор из двух цифр, определяющих координаты клетки для вывода маркера промаха/попадания.
- Использовать DOM для получения элемента с полученным идентификатором.
- Задать атрибуту `class` этого элемента значение "hit" (для метода `displayHit`) или "miss" (для метода `displayMiss`).



## Реализация displayHit и displayMiss

Оба метода, `displayHit` и `displayMiss`, получают аргумент `location`, определяющий ячейку для вывода маркера (попадания или промаха). Аргумент должен содержать идентификатор ячейки (элемента `<td>`) в таблице, представляющей игровое поле в HTML. Итак, прежде всего необходимо получить ссылку на этот элемент методом `getElementById`. Попробуем сделать это в методе `displayHit`:

```
displayHit: function(location) {  
    var cell = document.getElementById(location);  
},
```

Напоминаем: значение `location` образуется из строки и столбца и совпадает с идентификатором элемента `<td>`.

Следующим шагом станет добавление класса "hit" к элементу ячейки. Для этого мы можем воспользоваться методом `setAttribute`:

```
displayHit: function(location) {  
    var cell = document.getElementById(location);  
    cell.setAttribute("class", "hit");  
},
```

Элементу назначается класс "hit". При этом в ячейке таблицы немедленно появляется изображение корабля.

Добавим этот код в объект представления, а также напишем аналогичный метод `displayMiss`:

```
var view = {  
    displayMessage: function(msg) {  
        var messageArea = document.getElementById("messageArea");  
        messageArea.innerHTML = msg;  
    },  
  
    displayHit: function(location) {  
        var cell = document.getElementById(location);  
        cell.setAttribute("class", "hit");  
    },  
  
    displayMiss: function(location) {  
        var cell = document.getElementById(location);  
        cell.setAttribute("class", "miss");  
    }  
};
```

Идентификатор, созданный по введенным пользователем координатам, используется для получения обновляемого элемента.

Этому элементу назначается класс "hit".

То же самое делается в методе `displayMiss`, только элементу назначается класс "miss", отвечающий за отображение промаха на игровом поле.

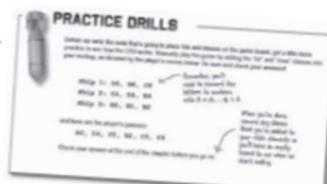
Не забудьте добавить код методов `displayHit` и `displayMiss` в файл `battleship.js`.

## Следующий тест-драйв...



Давайте проверим, как работает этот код, прежде чем двигаться дальше... Мы возьмем выстрелы из упражнения «Учебные стрельбы» и реализуем их в программном коде. Вот как выглядит последовательность, которую мы хотим реализовать:

A0, D4, F5, B2, C5, C6  
↑ ↑ ↑ ↑ ↑ ↑  
MISS HIT MISS HIT MISS HIT



Чтобы реализовать эту последовательность в коде, добавьте следующий фрагмент в конец файла `battleship.js`:

```

view.displayMiss("00");
view.displayHit("34");
view.displayMiss("55");
view.displayHit("12");
view.displayMiss("25");
view.displayHit("26");

```

↖ "AO"      ↖ "D4"      ↖ "F5"      ↖ "B2"      ↖ "C5"      ↖ "C6"

Напомним, что методы `displayHit` и `displayMiss` получают координаты выстрела, уже преобразованные из комбинации "буква+цифра" в строку из двух цифр, соответствующую идентификатору одной из ячеек таблицы.

И не забудьте протестировать `displayMessage`:

```
view.displayMessage("Tap tap, is this thing on?");
```

Для простого  
тестирования годится  
любое сообщение...

И не забудьте протестировать `displayMessage`:

```
view.displayMessage("Tap tap, is this thing on?");
```

Для простого  
тестирования годится  
любое сообщение...

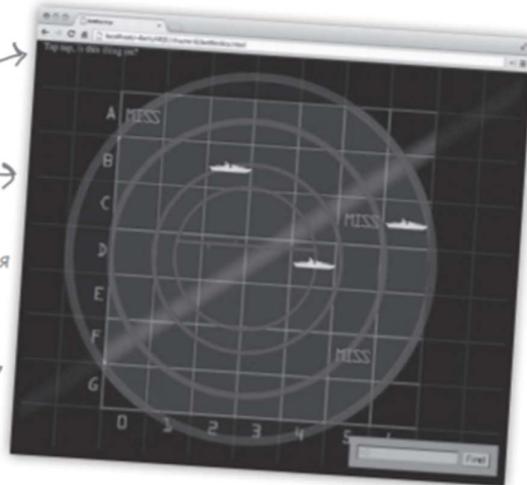
Когда это будет сделано, перезагрузите страницу в браузере и посмотрите, как изменился ее внешний вид.

**Одно из преимуществ  
разбиения кода  
на объекты и закрепления  
одной задачи  
за каждым объектом —  
это возможность  
протестировать каждый  
объект и убедиться,  
что он правильно  
выполняет свои  
обязанности.**

Сообщение выводится  
в левом верхнем углу  
игрового поля.

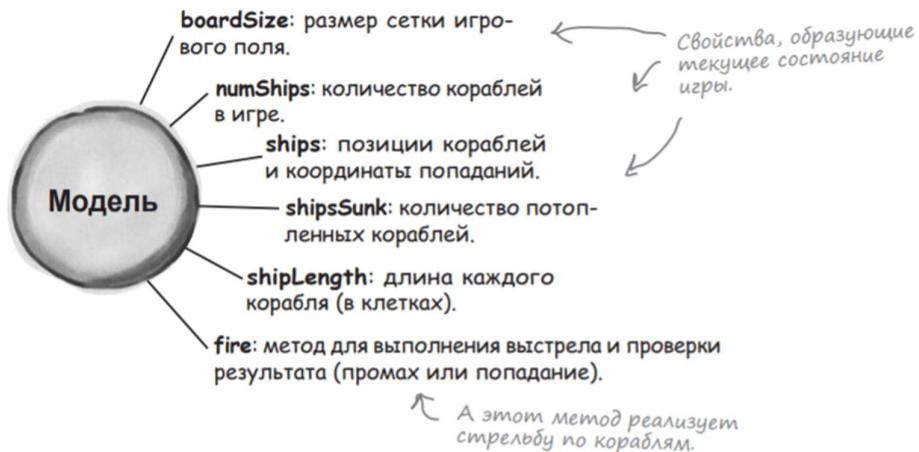
Попадания и промахи,  
для отображения которых  
используется объект  
представления, появляются  
на игровом поле.

Проверьте каждый  
маркер и убедитесь  
в том, что он выводится  
в нужном месте.



## Модель

Разобравшись с объектом представления, перейдем к модели. В объекте модели хранится текущее состояние игры. Модель также часто содержит логику, связанную с изменениями состояния. В нашем случае состояние включает позиции кораблей, координаты попаданий и счетчик потопленных кораблей. Пока вся необходимая логика ограничится проверкой того, попал ли выстрел игрока в корабль, и пометкой попадания. Объект модели должен выглядеть примерно так:



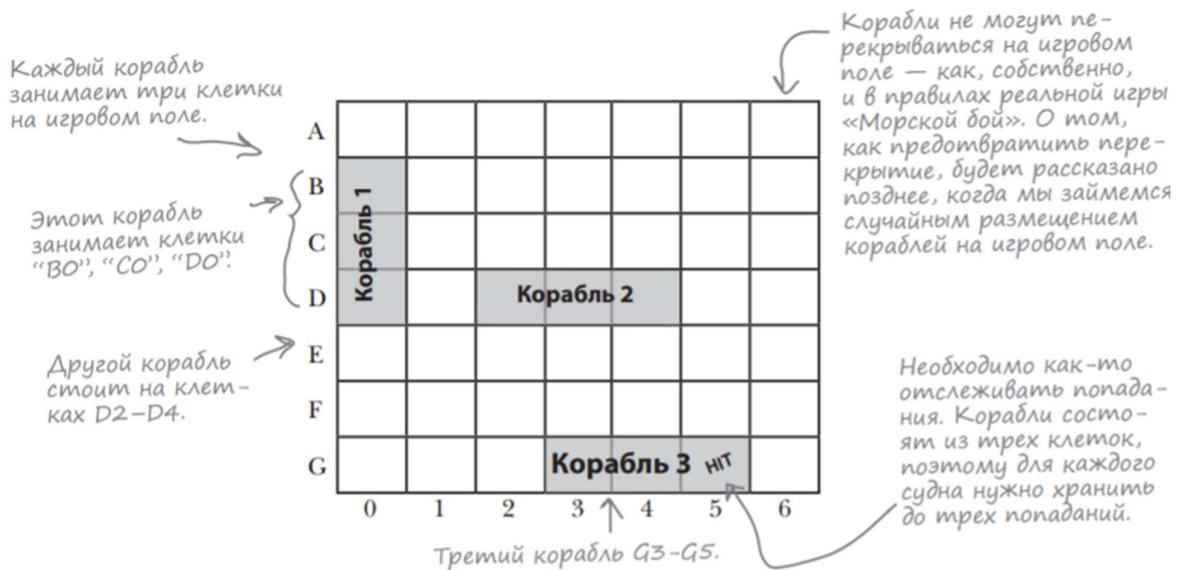
Как модель взаимодействует с представлением

При изменении состояния игры (то есть после выстрела с попаданием или промахом) представление должно обновить изображение в браузере. Для этого модель взаимодействует с представлением; к счастью, у нас есть методы, которые помогут организовать такие взаимодействия. Начнем с определения логики в модели, а потом добавим код для обновления представления.



Нам нужны большие корабли... и большое игровое поле

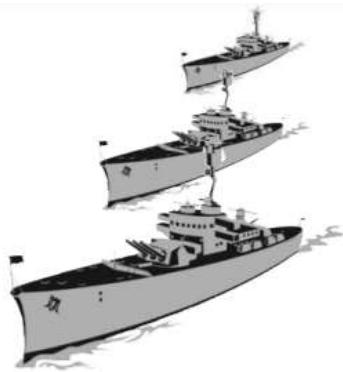
Прежде чем писать код модели, следует подумать над представлением состояния кораблей в модели. В упрощенной версии «Морского боя» из 3 урока было всего один корабль, который располагался на игровом поле 1×7. Теперь ситуация усложнилась: в игре три корабля на поле 7×7. Вот как это выглядит:



## Как мы будем представлять данные кораблей

Существует много способов организации данных кораблей, более того, вы наверняка можете предложить несколько своих вариантов. Какие бы данные вы ни использовали, их всегда можно организовать по-разному, причем у каждого способа есть «плюсы» и «минусы» — одни способы будут эффективны по затратам памяти, другие оптимизируют время выполнения, третьи более понятны программисту и т. д.

Мы выбрали относительно простой способ — каждый корабль представляется объектом, хранящим координаты занятых клеток и количество попаданий. Рассмотрим один из таких объектов:



```
var ship1 = {  
    locations: ["10", "20", "30"], ←  
    hits: ["", "", ""]  
};
```

Каждый корабль представляется объектом. ←  
Объект содержит два свойства, locations и hits.

Свойство hits содержит массив с информацией о попаданиях выстрелов в клетки. Изначально элементы массива инициализируются пустой строкой и заменяются строкой "hit" при попадании в соответствующую клетку. ←  
В свойстве locations хранится массив всех клеток, занимаемых кораблем.

← Обратите внимание: координаты представляются двумя цифрами (0 = A, 1 = B и т. д.).

Так должны выглядеть все три корабля:

```
var ship1 = { locations: ["10", "20", "30"], hits: ["", "", ""] };  
var ship2 = { locations: ["32", "33", "34"], hits: ["", "", ""] };  
var ship3 = { locations: ["63", "64", "65"], hits: ["", "", "hit"] };
```

Вместо того чтобы создавать три разные переменные для хранения информации о кораблях, мы создадим один массив для хранения всех данных:

```
var ships = [{ locations: ["10", "20", "30"], hits: ["", "", ""] },  
             { locations: ["32", "33", "34"], hits: ["", "", ""] },  
             { locations: ["63", "64", "65"], hits: ["", "", "hit"] }];
```

← В имени используется множественное число: ships. ←  
Переменной ships присваивается массив, в котором хранятся данные всех трех кораблей.

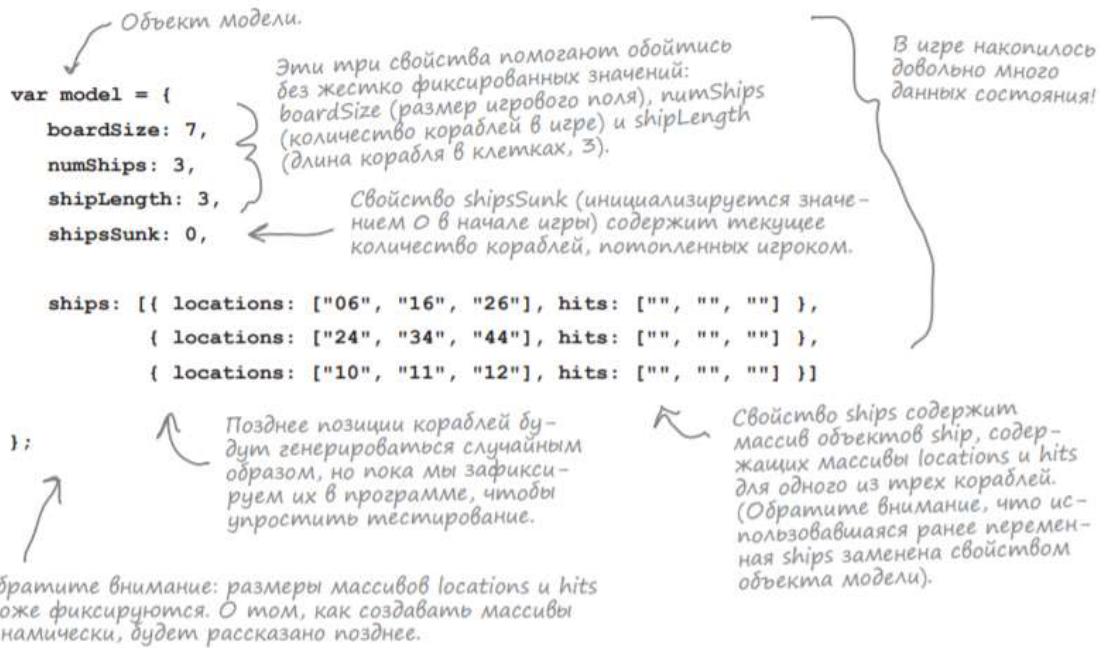
← Первый корабль... ← ...второй... ← ...и третий.  
← Обратите внимание: у этого корабля было попадание в клетке "65".

## Реализация объекта модели

Итак, мы разобрались, как в программе будут представляться корабли и попадания, теперь можно переходить к коду. Сначала мы создадим объект модели, а затем возьмем структуру данных `ships` и добавим ее как свойство. Раз уж мы этим занялись, нам понадобятся и другие свойства, например `numShips` для информации о количестве кораблей в игре. Возможно, вы удивитесь: «Ведь мы знаем, что в игре три корабля, — зачем нужно свойство `numShips`?» Представьте, что вам захочется создать новую, более сложную игру с четырьмя или пятью кораблями? Отказываясь от фиксированного значения и используя свойство (и указывая в коде свойство вместо конкретного значения), мы избавляемся от будущих проблем, потому что изменения будет достаточно внести в одном месте.

Кстати, раз уж мы заговорили о «фиксированных значениях» — исходные позиции кораблей будут фиксированными... пока. Точная информация о положении кораблей упростит тестирование и позволит сосредоточиться на логике. Код случайного размещения кораблей на игровом поле будет рассмотрен чуть позже.

Итак, создадим объект модели:



## Что должен делать метод fire

Метод `fire` преобразует выстрел игрока в промах или попадание. Мы знаем, что за маркеры промахов и попаданий отвечает объект представления `view`, но `fire` должен предоставить игровую логику определения результата выстрела (промах или попадание).

Определить, попал ли выстрел в корабль, несложно:

- Проверить каждый корабль и узнать, занимает ли он указанную клетку.
- Если клетка присутствует в списке позиций, значит, выстрел попал в цель. Программа помечает соответствующий элемент массива `hits` (и сообщает представлению о попадании). Метод возвращает `true` — признак попадания.
- Если указанная клетка не занята кораблем, значит, игрок промахнулся. Мы сообщаем об этом представлению и возвращаем `false`.



Метод `fire` должен определять, не был ли корабль потоплен, но этим мы займемся чуть позже.

## Подготовка метода fire

Для начала набросаем заготовку метода `fire`. Метод получает аргумент с координатами выстрела, перебирает все корабли и проверяет, пришлось ли попадание в очередной корабль. Код проверки попаданий будет написан чуть позже, а пока подготовим все остальное:

```
var model = {  
    boardSize: 7,  
    numShips: 3,  
    shipsSunk: 0,  
    shipLength: 3,  
    ships: [{ locations: ["06", "16", "26"], hits: ["", "", ""] },  
            { locations: ["24", "34", "44"], hits: ["", "", ""] },  
            { locations: ["10", "11", "12"], hits: ["", "", ""] }],  
    fire: function(guess) {  
        ← Метод получает координаты выстрела.  
        for (var i = 0; i < this.numShips; i++) {  
            var ship = this.ships[i];  
        }  
    }  
};  
← Не забудьте добавить запятую!  
Затем мы перебираем массив  
ships, последовательно проверяя  
каждый корабль.  
Здесь мы получаем объект корабля. Необходимо проверить, совпадают ли  
координаты выстрела с координатами одной из занимаемых им клеток.
```

## Проверка попаданий

Итак, при каждой итерации цикла необходимо проверить, присутствует ли указанная клетка в массиве `locations` объекта `ship`:

```
for (var i = 0; i < this.numShips; i++) {  
    var ship = this.ships[i];  
    locations = ship.locations;  
}  
}← Последовательно перебираем корабли.  
← Получаем массив клеток, занимаемых  
кораблем. Стоит напомнить, что  
это свойство корабля, в котором  
хранится массив.  
Здесь должен находиться код, который  
проверяет, присутствует ли указанная  
клетка в массиве locations корабля.
```

Ситуация такова: имеется строка `guess` с координатами клетки, которая ищется в массиве `locations`. Если значение `guess` встречается в массиве, значит, выстрел попал в корабль:

```
guess = "16";
locations = ["06", "16", "26"]; ← Мы должны определить, совпадает ли значение в guess с одним из значений из массива locations объекта корабля.
```

В принципе, можно было бы написать еще один цикл, который перебирает все элементы массива `locations` и сравнивает каждый элемент с `guess`; если значения совпадают, значит, выстрел попал в цель.

Но существует и другое, более простое решение:

```
var index = locations.indexOf(guess); ← Метод indexOf ищет в массиве указанное значение и возвращает его индекс (или -1, если значение отсутствует в массиве).
```

С методом `indexOf` код проверки попаданий может выглядеть так:

```
for (var i = 0; i < this.numShips; i++) {
    var ship = this.ships[i];
    locations = ship.locations;
    var index = locations.indexOf(guess);
    if (index >= 0) {
        ← // Есть попадание!
    } ← Обратите внимание на сходство метода indexOf массива с методом indexOf строки. Оба метода получают значение и возвращают индекс (или -1, если значение не найдено).
}
```

Если полученный индекс больше либо равен нулю, значит, указанная клетка присутствует в массиве `location` и выстрел попал в цель.

Решение с `indexOf` не эффективнее простого цикла, но оно чуть более понятно и явно более компактно. С первого взгляда становится очевидно, какое именно значение ищется в массиве с использованием `indexOf`. В любом случае в вашем инструментарии появился новый полезный инструмент.

## А теперь Все Вместе...

Напоследок нужно определиться еще с одним моментом: что должно происходить, когда выстрел попадает в корабль? В текущей версии мы просто отмечаем попадание в модели, что означает добавление строки "hit" в массив hits.

Давайте объединим все, о чём говорилось ранее:

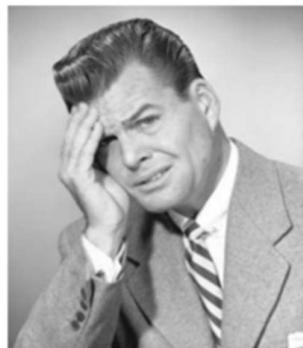
```
var model = {  
    boardSize: 7,  
    numShips: 3,  
    shipsSunk: 0,  
    shipLength: 3,  
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },  
            { locations: ["24", "34", "44"], hits: ["", "", ""] },  
            { locations: ["10", "11", "12"], hits: ["", "", ""] } ],  
  
    fire: function(guess) {  
        for (var i = 0; i < this.numShips; i++) {  
            var ship = this.ships[i];  
            var locations = ship.locations; ← Для каждого корабля...  
            var index = locations.indexOf(guess);  
            if (index >= 0) { ← Если координаты клетки присутствуют в массиве locations, значит, выстрел попал в цель.  
                ship.hits[index] = "hit"; ← Ставим отметку в массиве hits по тому же индексу.  
                return true;  
            } ← А еще нужно вернуть true, потому что выстрел оказался удачным.  
        }  
        return false; ← Если же после перебора всех кораблей попадание так и не обнаружено, игрок промахнулся, а метод возвращает false.  
    };  
};
```

Это отличное начало для работы над объектом модели. Осталось сделать еще пару вещей: определить, был ли корабль потоплен, и сообщить представлению об изменении модели, чтобы оно могло обновить изображение. Посмотрим, как это сделать...

## Я же просил выражаться покороче!

Нам снова приходится поднимать эту тему. Дело в том, что наши обращения к объектам и массивам получаются громоздкими. Присмотритесь к коду:

```
for (var i = 0; i < this.numShips; i++) {  
    var ship = this.ships[i]; ← Получаем объект ship...  
    var locations = ship.locations; ← Потом массив locations из объекта ship...  
    var index = locations.indexOf(guess); ← Затем индекс клетки в locations.  
    ...  
}
```



Пожалуй, этот код получился слишком длинным. Почему? Некоторые ссылки можно сократить посредством сцепления — ссылки на объекты объединяются в цепочки, позволяющие избежать создания временных переменных (таких, как переменная locations в приведенном коде). Почему

переменная `locations` названа временной? Потому что она используется только для промежуточного хранения массива `ship.locations`, чтобы мы могли вызвать метод `indexOf` для получения индекса `guess`. Ни для чего больше переменная `locations` в этом методе не используется. Сцепление позволяет нам избавиться от временной переменной `locations`:

```
var index = ship.locations.indexOf(guess);
```

Две выделенные строки объединены в одну строку.



## Как работает сцепление...

Сцепление в действительности представляет собой сокращенную форму записи для обращения к свойствам и методам объектов (и массивов). Давайте поближе присмотримся к тому, что было сделано для сцепления двух команд.

```
var ship = { locations: ["06", "16", "26"], hits: ["", "", ""] };
var locations = ship.locations;           Извлекаем массив locations из объекта ship.
var index = locations.indexOf(guess);    И используем его для вызова
                                         метода indexOf.
```

Две последние команды можно объединить в цепочку (избавляясь от временной переменной `locations`):

```
ship.locations.indexOf(guess)
```

1 Получаем объект ship.  
2 Объект обладает свойством locations, которое является массивом.  
3 И поддерживает метод indexOf.

## Тем временем на корабле...

Теперь мы напишем код, который будет проверять, потоплен ли корабль. Правила вам известны: корабль потоплен, когда выстрелы попали во все его клетки. Мы добавим вспомогательный метод для проверки этого условия:

Метод с именем `isSink` получает объект корабля и возвращает `true`, если корабль потоплен, или `false`, если он еще держится на плаву.

```
isSink: function(ship) {
  for (var i = 0; i < this.shipLength; i++) {
    if (ship.hits[i] !== "hit") {
      return false;
    }
  }
  return true;
}
```

Метод получает объект корабля и проверяет, помечены ли все его клетки маркером попадания.

Если есть хотя бы одна клетка, в которую еще не попал игрок, то корабль еще жив и метод возвращает `false`.

А если нет — корабль потоплен! Метод возвращает `true`.

Добавьте этот метод в объект модели, сразу же за методом `fire`.

Теперь мы можем использовать этот метод в методе `fire`, чтобы проверить, был ли корабль потоплен:

```
fire: function(guess) {  
    for (var i = 0; i < this.numShips; i++) {  
        var ship = this.ships[i];  
        var index = ship.locations.indexOf(guess);  
        if (index >= 0) {  
            ship.hits[index] = "hit";  
            if (this.isSunk(ship)) { ←  
                this.shipsSunk++;  
            }  
            return true;  
        }  
    }  
    return false;  
}, ←  
isSunk: function(ship) { ... }
```

Мы добавим проверку здесь, после того как будем точно знать, что выстрел попал в корабль. Если корабль потоплен, то мы увеличиваем счетчик потопленных кораблей в свойстве `shipsSunk` модели.

Новый метод `isSunk` добавляется сразу же после `fire`. И не забудьте, что все свойства и методы модели должны разделяться запятыми!

## Взаимодействие с представлением

Вот, собственно, и все, что нужно сказать об объекте модели. Модель содержит состояние игры и логику проверки попаданий и промахов. Не хватает только кода, который бы оповещал представление о попаданиях или промахах в модели. Давайте напишем его:

```
var model = {  
    boardSize: 7,  
    numShips: 3,  
    shipsSunk: 0,  
    shipLength: 3,  
    ships: [ { locations: ["06", "16", "26"], hits: ["", "", ""] },  
            { locations: ["24", "34", "44"], hits: ["", "", ""] },  
            { locations: ["10", "11", "12"], hits: ["", "", ""] } ],  
    fire: function(guess) {  
        for (var i = 0; i < this.numShips; i++) {  
            var ship = this.ships[i];  
            var index = ship.locations.indexOf(guess);  
            if (index >= 0) {  
                ship.hits[index] = "hit";  
                view.displayHit(guess);  
                view.displayMessage("HIT!");  
                if (this.isSunk(ship)) {  
                    view.displayMessage("You sank my battleship!");  
                    this.shipsSunk++;  
                }  
                return true;  
            }  
        }  
        view.displayMiss(guess);  
        view.displayMessage("You missed.");  
        return false;  
    },  
    isSunk: function(ship) {  
        for (var i = 0; i < this.shipLength; i++) {  
            if (ship.hits[i] !== "hit") {  
                return false;  
            }  
        }  
        return true;  
    }  
};
```

Мы приводим полный код объекта модели, чтобы вы могли окунуть взглядом всю картину.

Оповещаем представление о том, что в клетке guess следует вывести маркер попадания.

И приказываем представлению вывести сообщение "HIT!".

Сообщаем игроку, что он потонил корабль!

Сообщаем представлению, что в клетке guess следует вывести маркер промаха.

И приказываем представлению вывести сообщение о промахе.

Методы объекта представления добавляют класс "hit" или "miss" к элементу с идентификатором, содержащимся в guess. Таким образом, представление преобразует «попадания» из массива hits в разметку HTML. Не забывайте, что HTML-«попадания» нужны для отображения информации, а «попадания» в модели представляют фактическое состояние.



## Тест-драйв

Добавьте весь код модели в файл battleship.js. Протестируйте его методом fire модели, передавая строку и столбец из guess. Позиции кораблей до сих пор жестко фиксированы, поэтому вам будет легко их подбить. Добавьте собственные выстрелы (несколько дополнительных промахов). (Чтобы увидеть нашу версию кода, загрузите файл battleship\_tester.js.)



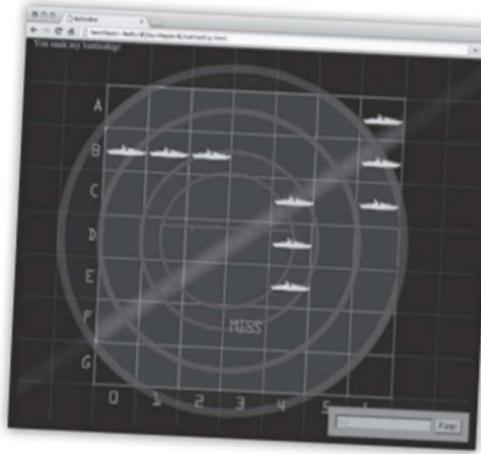
Чтобы получить такие же результаты, вам придется удалить или закомментировать предыдущий код тестирования. В файле battleship\_tester.js показано, как это делается.

```
model.fire("53");

model.fire("06");
model.fire("16");
model.fire("26");

model.fire("34");
model.fire("24");
model.fire("44");

model.fire("12");
model.fire("11");
model.fire("10");
```



Перезагрузите страницу battleship.html. Маркеры промахов и попаданий должны появиться на игровом поле.