

# Less

ПУТЕВОДИТЕЛЬ

ДЛЯ

НОВИЧКОВ

Denis Malinochkin

Published  
with GitBook



---

# Содержание

Введение	1.1
Предисловие	1.2
Благодарности	1.3
Целевая аудитория	1.4
Глава 1. Основы	1.5
Препроцессоры	1.5.1
CSS-препроцессоры	1.5.2
Грань между Less и CSS	1.5.3
Компиляция и отладка	1.5.4
Использование плагинов	1.5.5
Глава 2. Работа с селекторами, медиа-запросами и файлами	1.6
Комментарии	1.6.1
Вложенные правила	1.6.2
Ссылка на родителя селектора	1.6.3
Группировка селекторов	1.6.4
Использование медиавыражений	1.6.5
Импорт стилей	1.6.6
Домашнее задание	1.6.7
Глава 3. Переменные и примеси	1.7
Использование переменных	1.7.1
Интерполяция переменных	1.7.2
Наименование переменных	1.7.3
Использование примесей	1.7.4
Специальные параметры и сопоставление шаблонов	1.7.5
Дополнительные возможности примесей	1.7.6
Работа с набором правил	1.7.7
Домашнее задание	1.7.8
Глава 4. Операции со свойствами и встроенные функции	1.8
Слияние свойств	1.8.1
Строки и списки	1.8.2

---

Работа с изображениями	1.8.3
Работа с типами данных и единицами измерения	1.8.4
Математические функции	1.8.5
Манипуляции с цветами	1.8.6
Домашнее задание	1.8.7
Глава 5. Инструкции (операторы)	1.9
Условные конструкции (защита примесей)	1.9.1
Циклические конструкции	1.9.2
Домашнее задание	1.9.3
Глава 6. JavaScript в Less	1.10
Обзор возможностей	1.10.1
Работа с переменными и примесями	1.10.2

Добро пожаловать в **Путеводитель для новичков по CSS-препроцессору Less!** Эта книга поможет вам разобраться со всеми возможностями, которые предоставляет этот замечательный CSS-препроцессор. Неважно, новичок вы, впервые решивший разобраться с препроцессорами или бородатый веб-разработчик, который повидал на своём веку многие технологии — каждый найдёт что-нибудь интересное для себя.

## Необходимые умения:

Прежде чем вы перейдёте к чтению, я бы хотел рассказать о нескольких основных правилах:

- У вас установлен [Node.js](#).
- Вы умеете пользоваться пакетным менеджером [npm](#).
- Вы имеете представление о работе с консолью.
- Вы хорошо знаете HTML (HTML5), CSS3 и как-нибудь разбираетесь в JavaScript.

## Текущее положение дел

На текущий момент основная часть книги написана и соответствует реальности. Хотя работа ещё далека от того момента, когда можно сказать «хватит» — книгу уже можно рекомендовать к прочтению. Я стараюсь поддерживать актуальность изложенной в ней информации и пристально слежу за каждым выходом новой версии Less.

## Прочее

Если у вас нет времени на прочтение этой книги, то могу порекомендовать вам пройти два курса проекта HTML Academy:

- [«Знакомство с LESS»](#)
- [«Примеси в LESS»](#)

---

[www.canonium.com](http://www.canonium.com)

[Анонс книги на Canonium](#)

[Репозиторий на GitHub](#)



# Предисловие

По моему мнению, революция, которую совершили CSS-препроцессоры полностью перевернула представление о написании стилей. Многие люди уже тогда рассматривали возможность перехода с «чистого» CSS на препроцессорный. Такие миграции продолжают до сих пор, но с умеренным темпом, так как основная масса разработчиков уже стала использовать этот удобный инструмент. Даже самые упёртые и недоверчивые сторонники «чистого» CSS начали бросать свои взгляды на препроцессоры.

Сейчас рынок препроцессоров делится на три основных лагеря. Для того, чтобы люди могли сделать правильный выбор, многие разработчики пишут статьи, в которых рассказывают о преимуществах одного CSS-препроцессора перед другими. Такие статьи будут интересны всем без исключения, хотя бы ради холивара и вечных споров. Однако, этот интерес утихнет тогда, когда вы выберете себе инструмент по душе. Таким инструментом для меня стал Less, которому и будет посвящена эта книга.

В этой книге речь пойдёт об одном из самых популярных CSS-препроцессоров — Less. Во многом, Less получил свою популярность благодаря практически родному для CSS синтаксису, лёгкому в освоении функционалу, фреймворку Bootstrap, отличной и доступной документации, а также тому, что он написан на JavaScript. А, как известно, нет ничего роднее, ближе и удобнее для веб-разработчика, чем JavaScript. Возможно, для многих Ruby (на нём были написаны первые версии Less) ближе, но не для меня.

Эта книга — не энциклопедия и не справочник по Less. Здесь вы не найдёте сухого описания всех возможностей этого препроцессора, дословного перевода или пересказа документации, а также скучных примеров, которые никому не нужны, кроме автора. Моя задача — познакомить вас с этим препроцессором, дать базовое представление о нём и помочь организовать работу с ним. Проще говоря, моя цель — это направить вас на путь истинный.

Повторюсь, что в рамках этой книги я постараюсь максимально простым языком, понятным даже новичкам, рассказать о возможностях Less. И для того, чтобы это сделать максимально понятно, я буду использовать большое количество примеров, заданий и разговоров о том, почему лучше делать так, а не иначе.

Все примеры в этой книге будут составляться таким образом, чтобы охватывать реальные ситуации. Однако, в некоторых частях книги возможно появление так называемых примеров из сферического вакуума, которые полностью отображают суть

проблемы и её решения, но в повседневной разработке встречаются крайне редко или не встречаются вовсе.

Под заданиями я понимаю тематические упражнения, нацеленные на закрепление пройденного материала. Без попыток применения полученных знаний на практике — ваши знания равны нулю. Нет ничего лучше, чем бросаться с некоторым количеством знаний в голову на реальную задачу.

## **Возможные опечатки и неточности**

Я могу допускать ошибки, ошибаться и говорить всякий бред. Поэтому не стоит читать эту книгу с надеждой на то, что это единственное лекарство от всех ваших бед.

Прочитав эту книгу, вы не станете успешным веб-разработчиком, а всего лишь познакомитесь с такой технологией, вернее даже сказать, инструментом, как CSS-препроцессор Less.

# Благодарности

Выражаю огромную благодарность всем людям, принимавшим участие в написании этой книги. Без вас у меня ничего бы не вышло.



## Целевая аудитория

Я предполагаю, что читатель обладает знанием HTML, CSS и имеет базовое представление о JavaScript. Помимо этого, потребуется умение открытия консоли и воспроизведения строк, приведённых в этой книге. Также приветствуется представление о системах сборки наподобие Grunt, Gulp, Brunch или Broccoli.

Независимо от того, как давно вы занимаетесь веб-разработкой и какой у вас за плечами багаж знаний, эта книга поможет понять, почему так важно использование CSS-препроцессоров и почему Less отлично подходит для того, чтобы стать вашим первым CSS-препроцессором.

Людам, которые привыкли к сухому изложению текста, скучным пересказам документации и отсутствию шуток — книга категорически запрещается к прочтению. Вот такие вот дела.

# Глава 1. Основы

Прежде чем начинать использовать технологии, хорошо было бы понять, что они из себя представляют и зачем вообще могут пригодиться. В этой главе я расскажу о том, что такое препроцессор и CSS-препроцессор, приведу несколько популярных проектов в этой области и постараюсь передать их суть. Наконец, вы узнаете о препроцессоре Less, о котором и написана эта книга.

Какой богатый внутренний мир. Ну-ка дай-ка взглянуть.

Gearbox Software — Duke Nukem Forever

# Препроцессоры

Пожалуй, начнём с определения:

**Препроцессор** (от англ. Preprocessor) — это инструмент, преобразующий код из одного синтаксиса в другой. Обычно, на вход препроцессора поступает код, написанный с использованием синтаксических конструкций, понятных этому препроцессору.

Стоит сказать, что препроцессор может полностью замещать синтаксические конструкции языка или частично расширять их, то есть добавлять новые конструкции. Так как препроцессор преобразует код, то помимо входа у него должен быть и выход. Иначе какой смысл в его работе, ведь другие программы не смогут воспользоваться его трудами.

Как правило, на выходе ожидается код с более низким уровнем. То есть код, лишённый синтаксических конструкций, вносимых препроцессором. Сейчас нам не важно, что происходит с такими конструкциями, допустим, что препроцессор их раскрывает или удаляет в соответствии с заложенными в него правилами. Иначе говоря, на выходе мы получаем код, понятный программе, которая будет использовать его после препроцессора.

## Пример

Рассмотрим работу препроцессора на простейшем примере. Условия задачи у нас будут такими:

Пусть на вход препроцессора поступает строка, включающая в себя последовательность слов, разделённых пробелами, запятыми и точками. Основная функция препроцессора — замена ключевого слова «.CSS.» на его полную форму записи «Cascading Style Sheets». На выходе препроцессора имеем преобразованную строку.

Итак, исходя из условий мы понимаем, что «.CSS.» — это синтаксическая конструкция, которую должен обработать препроцессор.

Допустим, что на вход была подана следующая строка:

```
Developers use CSS preprocessors to build .CSS. faster.
```

Как мы видим, в этой строке два раза встречается слово «CSS». В первом случае это будет обычным словом как, например, «use» или «build», а во втором — синтаксической конструкцией препроцессора.

Далее препроцессор преобразует строку в соответствии с заложенным в него функционалом, и, в зависимости от настроек, на выходе имеется преобразованная строка.

```
Developers use CSS preprocessors to build Cascading Style Sheets faster.
```

Так работают препроцессоры в самом примитивном случае. В больших проектах все куда сложнее.

## Некоторые понятия

Кстати, для того, чтобы понимать некоторые статьи и книги, нужно запомнить, что вместо слова «преобразует», могут использоваться слова «компилирует» и «транслирует».

Если вы ничего не поняли из этой части главы, не расстраивайтесь — это нормально. Понимание придёт к вам на следующей странице. Конечно, при желании.

# CSS-препроцессоры

Если рассматривать препроцессоры вместе с CSS, то получается картина более понятная, нежели чем рассматривать понятие препроцессора отдельно.

## Определение

**CSS препроцессор** (от англ. CSS preprocessor) — это надстройка над CSS, которая добавляет ранее недоступные возможности для CSS, с помощью новых синтаксических конструкций.

Основная задача препроцессора — это предоставление удобных синтаксических конструкций для разработчика, чтобы упростить, и тем самым, ускорить разработку и поддержку стилей в проектах.

CSS препроцессоры преобразуют код, написанный с использованием препроцессорного языка, в чистый и валидный CSS-код.

При помощи препроцессоров вы можете писать код, который нацелен на:

- Читабельность для человека
- Структурированность и логичность
- Производительность

И это лишь малая часть того, что может дать вам препроцессор. Но не стоит забегать вперёд.

## Синтаксический сахар

Перед тем, как перейти к дальнейшему рассмотрению CSS-препроцессоров, давайте обновим наш лексикон новым понятием — «синтаксический сахар».

**Синтаксический сахар** (от англ. syntactic sugar) — это дополнения синтаксиса языка программирования, которые не вносят каких-то существенных изменений или новых возможностей, но делают этот язык более читабельным для человека.

Синтаксический сахар вводит в язык альтернативные варианты записи заложенных в этот язык конструкций. Под альтернативными вариантами записи стоит понимать более короткие или удобные конструкции для человека, которые в конечном итоге

будут преобразовываться препроцессором в исходный язык, без синтаксического сахара.

Если попытаться применить это понятие к CSS-препроцессорам, то оно, в общем случае, полностью описывает их суть. Ещё раз напомним, что основной задачей препроцессоров является упрощение и ускорение разработки, а как это ещё можно сделать, если не ввести альтернативные варианты записи?

## Какие бывают CSS-препроцессоры?

Пора перейти к более конкретным примерам, а именно к самим CSS-препроцессорам. На момент написания книги можно выделить три популярных препроцессора:

- Less
- Sass (SCSS)
- Stylus

И несколько незначительных для нас игроков:

- Closure Stylesheets
- CSS Crush

О первой тройке мы поговорим отдельно немногим ниже, а вот о двух последних разговора вообще не будет, в виду их непопулярности. При желании, описания этих препроцессоров с лёгкостью можно найти в поисковике.

## Какой смысл использования препроцессоров?

Как я уже отметил выше, основные плюсы — это читабельность кода, структурирование и повышение производительности.

Существуют также и другие причины того, чтобы начать использовать препроцессор уже сегодня. Я хочу заострить на этом внимание, так как разработчики раньше, да многие и сейчас, отнекиваются от использования препроцессоров, находя их сложными, непонятными и ненужными.

### CSS — это сложно

Стандартный CSS — это сложно. Синтаксис без использования вложенности, которую предлагают CSS-препроцессоры, просто напросто сложен для зрительного восприятия. Кроме того, нужно помнить имя родителя при вложенности. Отсутствие нормальных переменных и «функций» делает CSS-код грязным и узконаправленным.

## Доступная документация

Прошли те времена, когда документация от препроцессоров была доступна только людям «в теме». Сейчас любой желающий может в кратчайшие сроки освоить любой из препроцессоров, причём с минимальными затратами сил.

## Простота использования

Использовать препроцессоры стало проще, чем раньше, причём намного проще. Для этого нужно лишь установить программу, которая будет следить за файлами, предназначенными для препроцессора, и при их изменении будет компилировать содержимое этих файлов в чистый CSS-код.

Для более продвинутых пользователей есть специальные сборщики проектов. Не думайте, что если вы используете программу для препроцессоров, а не сборщик проектов, то вы недостаточно круты. На самом деле, такие сборщики предлагают полный контроль и расширенные настройки, а не делают из вас джедая.

## Структура и логичность кода

Самым популярным предлагаемым функционалом любого CSS-препроцессора является возможность вкладывать селекторы друг в друга. Я не буду сейчас приводить пример, так как о возможностях Less, включая вложенность, будет написана соответствующая часть книги. На этом этапе вам стоит знать лишь то, что при использовании препроцессоров, можно вкладывать один селектор в другой, а другой в третий, получая что-то, похожее на оглавление книги:

1. Родительский селектор
  - 1.1. Вложенный селектор
  - 1.2. Вложенный селектор
    - 1.2.1. Вложенный селектор
  - 1.3. Вложенный селектор

Конечно, в реальной жизни селекторы не могут начинаться с цифр, однако, для проведения параллели между вложенностью и оглавлением, думаю такое упущение здесь уместно.

## Примеси

Если говорить совсем кратко, то, используя **примеси** (от англ. Mixins), можно сделать код переиспользуемым. Это помогает избежать вспомогательных классов в разметке или дублирования свойств от селектора к селектору.

## Модульность

Еще одним бонусом, который прямо сейчас уговорил бы меня начать пользоваться CSS-препроцессором, будет возможность вкладывать файлы в файлы, то есть проще говоря, производить конкатенацию файлов в заданной последовательности. Да, это можно организовать и на чистом CSS, но вкуче с другими возможностями получается очень мощный инструмент.

При этом мы получаем возможность делиться модулями (библиотеками примесей), которые создали для своих нужд и посчитали полезными для других людей. Получается, что любой разработчик может загрузить вашу библиотеку и использовать её в своих целях, вызывая по мере необходимости написанные вами примеси.

## Почему бы не подождать развития CSS?

Развитие CSS идёт очень маленькими и неуверенными шагами, так как W3C придерживается приоритета скорости срабатывания CSS (производительности). С одной стороны это правильно и очень важно, но с другой — это отсутствие удобства для разработчиков.

В пример поставлю одну из спецификаций CSS4, которую ввели под давлением разработчиков — селектор по родителю. Столь долгий путь от идеи до принятия решения был из-за того, что W3C считало такой селектор медленным и дальнейшее его использование на сайтах привело бы к диким тормозам. Конечно же, речь идёт о повсеместном применении этого селектора, а не о единичных случаях.

Так что не стоит ждать в ближайшее время революций и изменений, способных затмить функционал и возможности CSS-препроцессоров.

## Разновидности препроцессоров

Разумеется, как и в любой другой области, всегда есть конкуренция, и на рынке препроцессоров сейчас три главных, враждующих между собой лагеря:

- Less



- Sass (SCSS)
- Stylus

Враждующими я их называю, потому что каждый приверженец одного из препроцессоров считает своим долгом поливать нечистотами представителей других, скажем так, конфессий. Такая неприязнь особенно часто проявляется у любителей препроцессора Sass, который считается старейшим и мощнейшим из всех трёх препроцессоров.

Для полной картины, я хочу привести краткую справку по каждому препроцессору:

## Less

Собственно, герой этой книги. Самый популярный на момент написания книги препроцессор. Основан в 2009 году Алексис Сельер (Alexis Sellier) и написан на JavaScript (изначально был написан на Ruby, но Алексис вовремя сделал правильный шаг). Имеет все базовые возможности препроцессоров и даже больше, но не имеет условных конструкций и циклов в привычном для нас понимании. Основным плюсом является его простота, практически стандартный для CSS синтаксис и возможность расширения функционала за счёт системы плагинов.

## Sass (SCSS)

Самый мощный из CSS-препроцессоров. Имеет довольно большое сообщество разработчиков. Основан в 2007 году как модуль для HAML и написан на Ruby (есть порт на C++). Имеет куда больший ассортимент возможностей в сравнении с Less. Возможности самого препроцессора расширяются за счёт многофункциональной библиотеки Compass, которая позволяет выйти за рамки CSS и работать, например, со спрайтами в автоматическом режиме.

Имеет два синтаксиса:

- Sass (Syntactically Awesome Style Sheets) — упрощённый синтаксис CSS, который основан на идентичности. Считается устаревшим.
- SCSS (Sassy CSS) — основан на стандартном для CSS синтаксисе.

## Stylus

Самый молодой, но в тоже время самый перспективный CSS-препроцессор. Основан в 2010 году небезызвестной в наших кругах личностью TJ Holowaychuk. Говорят, это самый удобный и расширяемый препроцессор, а ещё он гибче Sass. Написан на

JavaScript. Поддерживает уйму вариантов синтаксиса от подобного CSS до упрощённого (отсутствуют `:`, `;`, `{}` и некоторые скобки).

## Грань между Less и CSS

Если попытаться напрочь своё воображение и не вдаваться в подробности, то окажется, что вы уже используете препроцессор Less, хотя и косвенно. Да, это не шутка. Любой CSS-файл считается валидным Less-файлом. Для этого даже не нужно менять расширение файла. Однако, такое положение дел возможно, если вы подключаете этот файл к другому файлу, который уже имеет расширение `.less` или работаете с консолью. О том, как работать с файлами в Less я буду говорить позднее, а вот пример валидного Less-файла я обязательно приведу для большего понимания темы.

Ниже приведён фрагмент самого обычного CSS-кода, который я использую у себя в проектах. Весь этот код хранится в файле `styles.css` вместе с сотнями других строчек, но нас интересует именно синтаксис, а не файл и его содержимое.

```
.area {
  margin-right: auto;
  margin-left: auto;
}

.area:before,
.area:after {
  display: table;
  content: " ";
}

.area:after {
  clear: both;
}

@media (min-width: 768px) {
  .area {
    width: 750px;
    padding-right: 15px;
    padding-left: 15px;
  }
}
```

Я думаю объяснять не стоит, что здесь мы имеем дело с обычным CSS. На этом этапе нам не важно, что и как делают эти строчки, главное, что они реальны и могут использоваться в любом CSS-файле уже сейчас.

А теперь возьмём эти стили и, ничего не меняя, поместим их в файл с таким же именем, но имеющим расширение `.less`. Я понимаю, что до сих пор мы не рассматривали возможность компиляции less-файлов, но поверьте мне на слово — файл скомпилируется без ошибок и будет точно таким же по содержанию.

<code>_test.less</code>	<code>test.css</code>
<pre>1 .area { 2   margin-right: auto; 3   margin-left: auto; 4 } 5 6 .area:before, 7 .area:after { 8   display: table; 9   content: " "; 10 } 11 12 .area:after { 13   clear: both; 14 } 15 16 @media (min-width: 768px) { 17   .area { 18     width: 750px; 19     padding-right: 15px; 20     padding-left: 15px; 21   } 22 }</pre>	<pre>1 .area { 2   margin-right: auto; 3   margin-left: auto; 4 } 5 .area:before, 6 .area:after { 7   display: table; 8   content: " "; 9 } 10 .area:after { 11   clear: both; 12 } 13 @media (min-width: 768px) { 14   .area { 15     width: 750px; 16     padding-right: 15px; 17     padding-left: 15px; 18   } 19 } 20</pre>

На изображении выше представлено состояние less-файла до компиляции и после. Сразу видно, что они идентичны и различаются лишь наличием пустых строк после селекторов в Less-файле, а также расширением.

Из всего этого напрашивается вывод, что даже если вы не будете использовать возможности Less полностью или будете пользоваться лишь некоторыми из них, то ничего страшного не случится. Вас никто не обязывает менять свой любимый синтаксис на какой-либо другой. Наоборот, препроцессор Less будет только рад этому, ведь в таком случае на компиляцию уйдёт меньше времени. Если говорить ещё проще, то препроцессор можно применять лишь для конкретных целей, например, для построения удобной и хорошо поддерживаемой структуры, которую можно организовать и на чистом CSS, но с меньшим удобством.

О структуре проекта я подробнее расскажу ближе к концу этой удивительной истории, когда к вам, по идее, придёт понимание того, чем вы занимались на протяжении всего времени, проведённого с этой книгой. Поэтому не волнуйтесь — всему своё время.

# Компиляция

Если вы помните, то препроцессоры предлагают нам свой вариант синтаксиса для некоторых или всех конструкций языка, надстройкой над которым они являются. И CSS-препроцессоры не исключение.

Для того, чтобы браузер понимал код, написанный с использованием синтаксических конструкций препроцессора, его нужно компилировать в понятный для него язык. Таким языком для браузера, как не сложно догадаться, является CSS.

Существует несколько вариантов того, как можно перейти от Less к CSS.

## Компиляция в браузере (less.js)

Наиболее простой способ использования CSS-препроцессора, но в тоже время малопопулярный. Альтернативные решения удобнее и предоставляют наиболее интересный функционал. Применяется на этапе разработки или отладки проекта, когда важен результат компиляции, а не её скорость.

Основан на идее подключения стилей с расширением `.less` к документу, используя стандартный тег `<link>`, но с изменённым атрибутом `rel`. А также осуществляется подключение файла библиотеки.

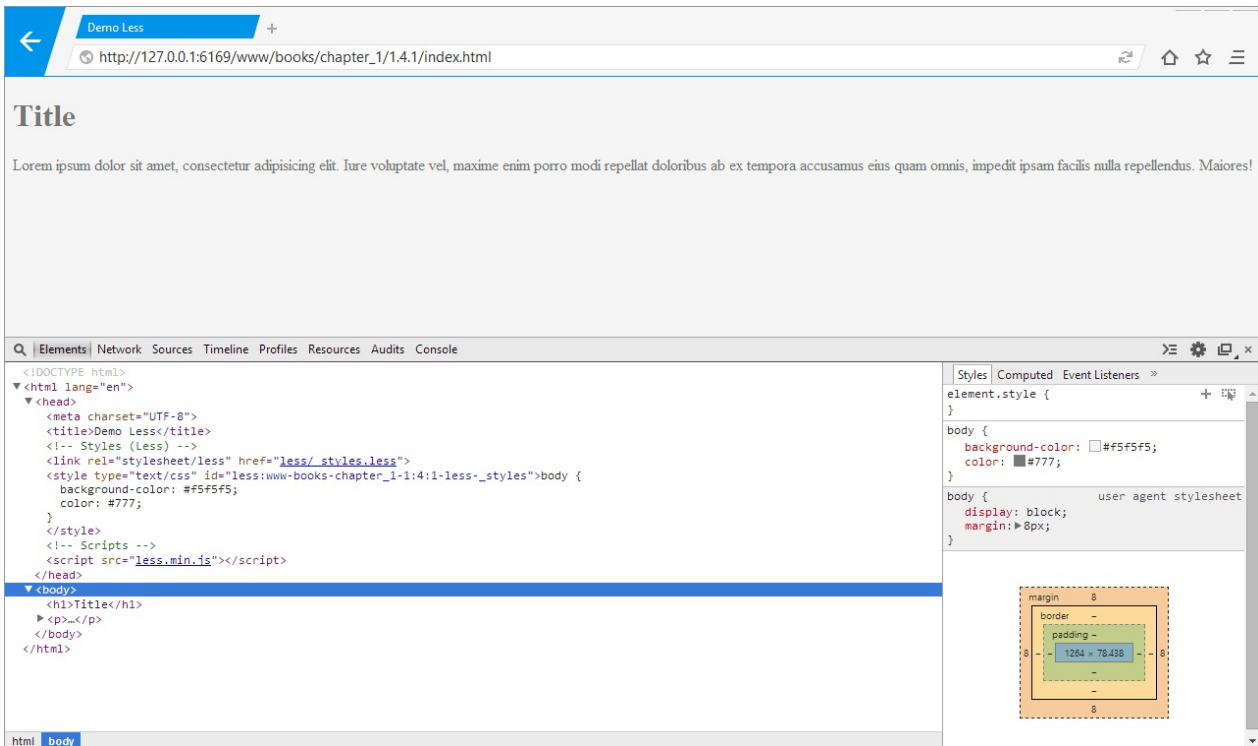
```
<link rel="stylesheet/less" href="...">
<script src="less.min.js"></script>
```

После компиляции скрипт проводит инъекцию полученного CSS-кода в секцию `head` документа посредством тега `<style>`.

Способ не желателен к применению на так называемом «продакшене» в виду того, что имеет серьёзные проблемы со скоростью и сильно зависит от производительности устройства, а также скорости интернет-соединения. Помимо этого увеличивается объем загружаемых данных, так как браузеру пользователя приходится загружать less-файлы и файл библиотеки. Только после полной загрузки необходимых ресурсов начинается процесс компиляции less-кода в CSS.

### Пример 1.4.1

Пример демонстрирует использование CSS-препроцессора Less прямоком в браузере, без предварительной компиляции CSS.



На изображении показана структура документа, подключённые файлы стилей, файл библиотеки, а также произведённая инъекция стилей в секцию `head`.

Разметка страницы:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Demo Less</title>

  <!-- Styles (Less) -->
  <link rel="stylesheet/less" href="less/_styles.less">

  <!-- Scripts -->
  <script src="less.min.js"></script>
</head>
<body>
  <h1>Title</h1>
  <p>...</p>
</body>
</html>
```

Содержимое файла `_styles.less` :

```
body {  
  background-color: #f5f5f5;  
  color: #777;  
}
```

## Пример 1.4.2

Если в файле стилей допущена ошибка, то компилятор отобразит сообщение, в котором будет присутствовать следующая информация:

- Код ошибки;
- Описание ошибки;
- Название файла, в котором обнаружена ошибка;
- Номер строки и столбца, где допущена ошибка;
- Код, порождающий ошибку;

Такой подробный отчёт позволит разработчику в удобной форме получить полную информацию о допущенной ошибке и в максимально короткие сроки приступить к её устранению.

The screenshot shows a browser window with the URL `http://127.0.0.1:6807/www/books/chapter_1/1.4.2/index.html`. The console displays a red error message: **ParseError: Unrecognised input. Possibly missing something**. Below the message, it specifies the error location: `in _styles.less on line 6, column 18:`. The code snippet shows line 5 as blank and line 6 as `.thisIsMySelector`. The browser's developer tools are open, showing the HTML structure and the 'Styles' panel with the 'body' style rules.

## Пример 1.4.3

При необходимости доступны некоторые настройки для управления преобразованиями, происходящими в препроцессоре и, как следствие, получаемыми данными на выходе. Причём настройки можно менять, используя JavaScript, так и с

помощью атрибутов `data` у тега `link`.

В этом примере добавляется возможность автообновления страницы при изменении подключённого файла стилей. Для активации механизма слежения за изменением ресурсов, необходимо добавить к адресу страницы маркер `#!watch`. Теперь, когда вы будете изменять less-файлы в проекте, которые подключены к активной странице, будет происходить автоматическая компиляция этого файла.

```
...
<head>
  ...
  <!-- Scripts -->
  <script>
    less = {
      env: 'development'
    };
  </script>
  <script src="less.min.js"></script>
  <script>less.watch();</script>
</head>
...
```

Другие параметры настроек можно посмотреть в документации. Так как этот метод находится в статусе «не рекомендуется», то и подробно рассматривать его нет смысла.

## Компиляция из командной строки (lessc)

Как мы уже хорошо знаем, препроцессор Less написан на JavaScript. А ещё лучше мы знаем, что лучший друг JavaScript теперь — Node.js. Отсюда вытекают новые возможности для работы в командной строке. И Less предлагает нам такой инструмент для работы в ней.

Работа из командной строки предполагает наличие установленного Node.js. Помимо этого, необходимо глобально установить пакет `less` — это можно сделать командой:

```
$ npm install -g less
```

Рассмотрим синтаксис команд `npm`:

- `npm` - пакетный менеджер;
- `i` - сокращение от `install`, то есть «установить»;
- `-g` - флаг, который указывает на то, что пакет будет установлен глобально;



- `less` - имя устанавливаемого пакета;

Общение с консолью осуществляется на примитивном уровне. Бегло ознакомимся с основными доступными командами:

### Компилирование файла с именем `_styles.less` без сохранения результата:

```
$ lessc _styles.less
```

### Компилирование файла `_styles.less` с сохранением результата в файл

`_main.css` :

```
$ lessc _styles.less > _main.css
```

Помимо двух базовых команд можно передавать параметры скрипту, в зависимости от которых будет выполняться то или иное действие с вашим кодом. Например, ниже показана передача параметра `x`, который говорит скрипту (компилятору), что на выходе пользователь ожидает увидеть минифицированный файл.

```
$ lessc -x _styles.less
```

И здесь наступает один важный момент, который может вызвать недопонимание. Если вы компилируете файл без сохранения результатов, то можно использовать параметр `x`. Но если вы собираетесь сохранить скомпилированный CSS-код, то будьте добры использовать плагины для его минификации. Проще говоря, при таком раскладе параметр `x` не работает.

Как вы уже поняли, для Less доступно некоторое множество плагинов, которые добавляют различный функционал препроцессору. Включением и выключением этих плагинов, а также их управлением, можно заниматься и с помощью консольных команд.

Допустим, что у пользователя установлен пакет для минификации CSS-файлов `less-plugin-clean-css`. Тогда пользователю будет доступен новый параметр `--clean-css` для передачи скрипту. Более детально о доступных плагинах и возможности их применения мы поговорим позднее.

Но и это ещё не все. Пользователю также доступна типичная для консольных утилит команда `--help`, которая расскажет обо всех доступных к использованию параметрах.

```
lessc --help  
lessc --h
```

В случае, как и с компиляцией в браузере, этот метод не особо популярен, так как существуют другие, более интересные по возможностям.

Если вас заинтересовал именно этот способ, то обязательно посетите документацию препроцессора. Кроме тех параметров, что были озвучены здесь, там вы сможете найти около двух десятков различных других специфических параметров, благодаря которым ваша работа с препроцессором в командной строке станет куда гибче.

## Компиляция, используя системы сборки

Настал тот момент, когда все скучные способы компиляции пройдены, а впереди есть ещё, которые выглядят куда интереснее всех остальных, включая и этот. Но я не могу обойти стороной самый популярный и гибкий способ.

Сейчас, если вы пишете Open Source проект, то хорошим тоном будет автоматизировать всё, что возможно и имеет смысл. Это делается для того, чтобы другие разработчики могли использовать ваши наработки и не мучиться с подготовкой проекта к «употреблению». Помимо этого, автоматизация действий, будь то банальное переименование файла или компиляция файлов препроцессора, экономит время не только других разработчиков, но и ваше.

**Система сборки** — это инструмент для автоматической, гибкой и удобной сборки проектов из командной строки с использованием ранее обозначенных инструкций (задач).

Если попытаться упростить эту формулировку, то получится следующее:

*Система сборки* — это инструмент, который автоматически преобразует проект в соответствии с заранее написанными правилами.

Первым популярным сборщиком был Grunt, позднее появился Gulp и самыми молодыми сейчас являются Brunch и Broccoli. Так повелось, что Grunt и Gulp воюют на одном универсальном поприще, покрывая своим функционалом и доступными плагинами почти весь спектр задач. А вот Brunch сразу же обозначил себя как сборщик для фронтенда и слегка намекает, что первые двое ему не родня, да и он не такой, как они.

Для Grunt и Gulp доступно большое количество различных пакетов, которыми можно выполнять практически любые задачи: от простейшего переименования файлов до сборки крупных приложений. Причём у Grunt их намного больше. А вот Brunch и

Browsers таким похвастаться не могут, но все основные пакеты для фронтенд разработки доступны уже сейчас. Практически любой недостающий функционал можно добавить, написав свой пакет, и делается это довольно просто.

К счастью, настройка систем сборки — это не основная тема книги. Поэтому конкретно рассматривать каждый из сборщиков я не буду, а предложу посмотреть уже готовые конфигурационные файлы для компиляции Less в архиве, идущем вместе с книгой.

## Приложения для компиляции

Вот он, **самый простой и удобный способ** для проектов, использующих препроцессоры, причём не только CSS-препроцессоры, но и JS и HTML.

Существуют такие приложения, которые позволяют управлять проектами без написания кода, использования командной строки и систем сборки. Они написаны для людей, желающих делать своё дело и не вникать в некоторые тонкости, хотя бы на начальном этапе своей карьеры.

Такие приложения имеют довольно обширный функционал и, как правило, умеют:

- Компилировать файлы различных препроцессоров (Less, Stylus, Jade, CoffeeScript и т.д.);
- Проверять файлы на ошибки и соответствие правилам (общим или проекта);
- Обрабатывать файлы (минификация, расстановка префиксов в CSS и т.д.);
- Автоматизировать некоторые часто используемые действия;
- Локальный сервер для тестирования проектов на этапе разработки;

Среди всех подобных приложений можно выделить следующие решения:

- Prepros
- CodeKit (только OS X)
- Mixture
- Koala

Ранее я использовал Prepros для своих локальных проектов, так как он предлагает наиболее интересный для меня функционал. Остальные решения также хороши, но мне больше всего приглянулся именно он.

Проекты Koala и Mixture более не обновляются и представлены здесь лишь в дань уважения.

Если вы не хотите писать конфигурационные файлы для Grunt, Gulp и им подобным системам сборки, то это ваш выбор.

## Альтернативные методы

Существуют решения для отдельно взятых сред (редакторов, IDE и т.д.), позволяющие использовать CSS-препроцессоры. В некоторых IDE есть встроенные средства для использования препроцессоров, а в тех, где нет, в общем случае, можно установить необходимые плагины, добавляющие такую возможность. По [этой ссылке](#) приведён обширный список таких плагинов.

Если для использования препроцессора Less на сервере, построенном с применением Node.js, требуется лишь официальный пакет доступный в npm, то на других платформах необходимы специальные библиотеки, а иногда и несколько (привет Java). Такой подход обеспечивает обмен переменными между Less и использующим его языком, что позволяет добиться компиляции файлов, в зависимости от контекста действий пользователя в приложении.

## Отладка

**Отладка** — это процесс обнаружения, локализации и исправления возникающих ошибок в работе приложения. В случае с CSS-препроцессором, приложением будут являться препроцессорные файлы, так как именно в них могут возникать ошибки или «неточности». И если с ошибками бороться нам поможет сам компилятор, то исправлять «неточности» будет сложнее, из-за некоторых особенностей препроцессоров.

## Карта кода (Source Maps)

Во время разработки и после неё, скомпилированные файлы стилей и исходные файлы могут сильно различаться. Происходит это из-за компиляции и обработки файлов. Если в процессе компиляции происходит раскрытие конструкций, написанных на препроцессорном языке в «чистый» CSS-код, то на выходе получается, как правило, большее количество строк кода. Выражается это в том, что в инспекторе браузера у тега стилей элемента указан один номер строки, а на самом деле он совсем другой.

### Пример 1.4.4

В этом примере представлен препроцессорный и скомпилированный файлы. Обратите внимание лишь на отличающееся количество строк, работу препроцессора и её результат на выходе.

## Файл препроцессора:

```
// Variables
@header-background: #181e21;
@header-color: #fff;

.global-header {
  position: relative;
  background-color: @header-background;
  color: @header-color;

  h1 {
    font-size: 44px;
    line-height: 50px;

    small {
      font-size: 24px;
      line-height: 36px;
    }
  }
}
```

## Скомпилированный файл:

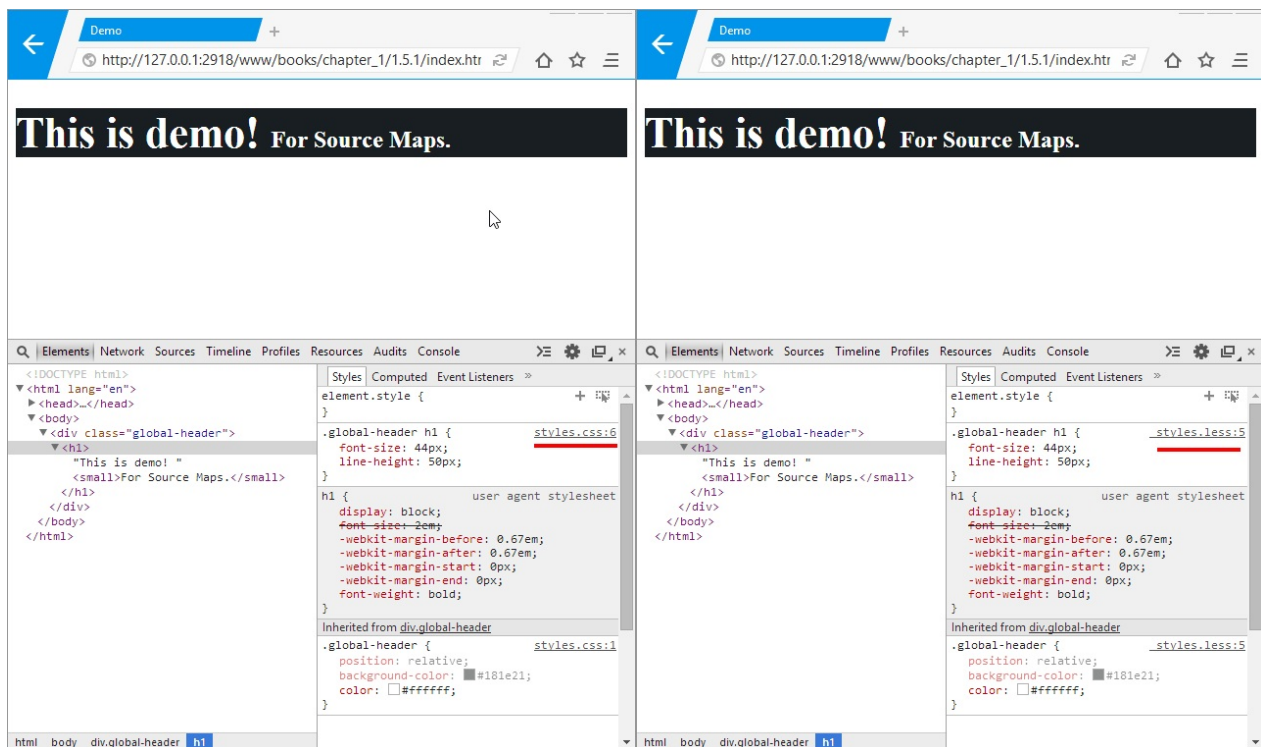
```
.global-header {
  position: relative;
  background-color: #181e21;
  color: #ffffff;
}
.global-header h1 {
  font-size: 44px;
  line-height: 50px;
}
.global-header h1 small {
  font-size: 24px;
  line-height: 36px;
}
```

Замечательно, с физикой проблемы разобрались. Для её решения на помощь спешит так называемая «карта кода». Явление это не уникальное и встретить его можно, например в Visual Studio 2012, правда там карта действительно похожа на карту, а тут информация, ориентированная лишь на браузер. Когда я говорю, что файл предназначен для браузера, то имею в виду, что понять его человеку будет практически невозможно:

```
{
  "version": 3,
  "sources": [
    "_styles.less"
  ],
  "names": [],
  "mappings": "AAIA;EACE,kBAAA;EACA,yBAAA;EACA,cAAA;;AAHF,cAKE;EACE,eAAA;EACA,iBAAA;;A
APJ,CAKE,GAIE;EACE,eAAA;EACA,iBAAA",
  "file": "undefined"
}
```

Получить такую карту кода можно, добавив параметр `--source-map` при компиляции, выбрав соответствующие пакеты для системы сборки или изменив настройки программы.

Теперь обратите внимание на изображение окна браузера, в котором слева показан скомпилированный файл без карты кода, а справа — с ней.



В первом случае видно, что селектор объявлен на 6 строке CSS-файла, но так как мы работаем с исходниками в формате Less, то нам эта цифра ни о чём не говорит — у нас на 6-ой строке находится другой код.

Во втором случае, когда применяется карта кода, все становится на свои места и указывается не только правильный номер строки объявления селектора, но и файл её содержащий. Название файла будет полезно тогда, когда я расскажу вам про подключение файлов друг к другу.



# Использование плагинов

Как я уже отмечал ранее, CSS-препроцессор Less имеет возможность подключения плагинов, расширяющих основной функционал препроцессора. Все доступные на сегодняшний момент плагины можно найти в пакетном менеджере npm, используя для поиска ключевое слово — `"less-plugin"`. Среди них довольно большое количество интересных и даже необходимых плагинов. Однако, давайте сначала посмотрим на то, как подключать плагины при компиляции less-файлов из консоли.

Дальнейшие манипуляции будут производиться с плагином `less-plugin-functions`, добавляющим возможность создания пользовательских функций.

Первым делом установим этот плагин локально или глобально. Я буду устанавливать глобально, поэтому у себя в консоли выполняю следующую команду:

```
$ npm i -g less-plugin-functions
```

Теперь обратимся к документации плагина и напишем небольшой пример, демонстрирующий работу функций:

```
// Исходя из документации, функции всегда должны начинаться со слова `function`  
.function-hello() {  
  return: "Hello!";  
}  
  
// Вызов функции производится без ключевого слова `function`  
.test {  
  content: hello();  
}
```

Когда компилятор встретит на своём пути какую-нибудь функцию, он подставит вместо её вызова значение, возвращаемое этой функцией. Всё как в JavaScript. Для того, чтобы скомпилировать этот файл, необходимо выполнить команду:

```
$ lessc --functions styles.less  
.test {  
  content: "Hello!";  
}
```

Заметьте, так как я не указал имя файла, в который компилятору необходимо сохранить результат компиляции — полученный результат был выведен в консоль.



Разумеется, простым возвращением результата функционал этого плагина не заканчивается: вы можете производить вычисления, конкатенировать значения свойств, вызывать другие функции и примеси...

Чтобы применить к файлу несколько плагинов, необходимо просто добавить их в команду, выполняемую в консоли:

```
$ lessc --functions --inline-urls styles.less
```

## Использование плагинов в браузере

В документации сказано, что любой плагин может работать с компилятором, подключаемым как внешний скрипт к странице с одним лишь условием: подключение плагина и его описание должно быть произведено раньше, чем будет подключён сам файл библиотеки `less.js` :

```
<script src="plugin.js"></script>
<script>
less = {
  plugins: [plugin]
};
</script>
<script src="less.min.js"></script>
```

Здесь кроется одна большая проблема — чтобы начать использовать плагины в браузере, необходимо найти хотя бы один плагин, который способен это делать без предварительной сборки. Дело в том, что браузер не умеет работать с `require()`, используемым для подключения файлов скриптов в мире Node.js. Почти все плагины используют `require()`, так как разбиты на несколько логических файлов. Поэтому, перед тем, как приступить к подключению заветного плагина к странице — его нужно собрать, используя для этого, например, Browserify. К счастью, это не входит в план книги.

## Глава 2. Работа с селекторами, медиа-запросами и файлами

Многие статьи начинаются со слов об удобном использовании переменных, примесей и прочих радостях. Я же начну свой рассказ о том, как же хорошо стало работать с селекторами, медиа-запросами и файлами. Мне кажется, что это наиболее важный инструментарий, который предлагает нам препроцессор. Да, работать с переменными одно удовольствие, но с селекторами приходится работать куда чаще.

В этой главе я расскажу о комментариях, вложенных селекторах, работе с файлами, группировке селекторов и многих других интересных вещах.

Если немножко подождать, все само упадёт тебе в лапы.

Twentieth Century Fox — Гарфилд (Garfield)

# Комментарии

Хороший код должен сопровождаться комментариями, хотя и бытует мнение, что прекрасный код вовсе не нуждается в них. Конечно, повсеместно писать комментарии не нужно, а вот ключевые моменты лучше всего сопровождать короткими записками.

Комментарии делают ваш код чище, а также помогают достичь понимания с другими разработчиками, работающими с вашим кодом. Самое главное, чтобы вы сами понимали то, что написали и старались излагать свои мысли максимально кратко, ясно и однозначно.

Если проект, над которым вы работаете используется не только вашим отделом, компанией или внутренним сообществом, то целесообразнее писать комментарии на английском языке. Иначе, если ваш проект применяется только внутри одной компании, в которой официальным языком принят, например, русский язык, то логичнее всего будет использовать именно его.

## Базовый синтаксис

Препроцессор Less поддерживает несколько синтаксисов написания комментариев. Самый очевидный — это стандартный для CSS синтаксис. Если отключена минификация (компрессия, сжатие) кода, то такие комментарии, содержащиеся между `/* */`, будут сохраняться в CSS-файле после компиляции. Поэтому используйте такой вид комментариев с умом.

```
/* Это стандартный однострочный комментарий для CSS */

/*
  А это стандартный блочный комментарий для CSS.
  При необходимости можно использовать и его.
*/
```

Помимо стандартного синтаксиса предлагается и препроцессорный, но только лишь однострочный вариант записи. Комментарии начинающиеся с `//` не будут сохраняться в CSS-файл после компиляции.

```
// Это препроцессорный однострочный комментарий
```

## Особые комментарии

Иногда требуется сохранить какие-то комментарии после компиляции и даже минификации кода. Такой подход чаще всего используется для включения в файл информации о лицензии, авторских правах, версии библиотеки и прочих важных сообщений. Как и в CSS для этого используются комментарии, заключённые в `/*! */`.

```
/*! Комментарий, который будет всегда сохраняться */
```

Также допустима запись `/*! !*/`, но она считается избыточной и применяется крайне редко, если вообще применяется.

```
/*! Комментарий, который будет всегда сохраняться !*/
```

## Вложенные комментарии

К сожалению, вкладывать комментарии в комментарии, как и CSS, Less не умеет. Однако, допустимо смешивать комментарии, заключённые в `/* */` и однострочные комментарии, начинающиеся с `//`.

```
/*  
  // Смешивание комментариев  
*/
```

Можно попытаться вложить комментарии и наоборот:

```
// /* Этот комментарий вложен в другой комментарий */
```

После компиляции less-файлов, в первом случае комментарий будет отображаться в скомпилированном CSS-файле, а вот втором случае — нет.

# Вложенные правила

В хорошо структурированных таблицах стилей нет необходимости присваивать каждому элементу классы. Достаточно лишь более подробно описывать стили элементов, используя возможность вкладывать селекторы в другие селекторы. К слову, такие селекторы называются **вложенными** и представляют собой объёмную структуру.

## Пример 2.2.1

Ниже приведён так называемый «подробный CSS», в котором представлена модель вложенности одних селекторов в другие селекторы.

```
.global-header {
  background-color: #f5f5f5;
  color: #443d3d;
  border-bottom: 1px solid #ddd;
}

.global-header h1 {
  margin-top: 0;
  margin-bottom: 0;
  font-size: 4rem;
}

.global-header h1 small {
  font-size: 2rem;
}

.global-header .header-actions {
  background-color: #fff;
  padding-top: 10px;
  padding-bottom: 10px;
  text-align: center;
}
```

## Решение проблем

Все это хорошо, но ровно до тех пор, пока имена классов короткие, глубина вложенности не велика, а ваши глаза в состоянии уследить за этой структурой. Лично я, до тех пор, пока не стал пользоваться CSS-препроцессорами, писал код именно так, попутно разделяя CSS-файл на логические блоки и пытаюсь уследить за его чистотой.

Это удобно и практично, но мне быстро надоело сортировать код по вложенности, и моя таблица стилей иногда была похожа на мешанину из букв и цифр. Разумеется, что перед финальной сборкой проекта все это исправлялось, но это не тот случай, на который хочется тратить время.

Представьте себе оглавление простейшей книги, например, такое:

```
1. Глава 1. Это моя первая глава книги
1.1. Это мой первый рассказ о том, что я сделал
1.1.1. Немного подробнее о том, что я сделал
1.2. Это мой второй рассказ о том, что я сделал
1.X. ...
```

Здесь довольно легко понять, что пункт 1.1 — это следствие первого пункта. Но тем не менее, если переписать это в более читабельном формате, можно сэкономить время на обработке цифр в голове. Абстрактно, можно представить следующий формат:

```
Это моя первая глава книги
  Это мой первый рассказ о том, что я сделал
    Немного подробнее о том, что я сделал

  Это мой второй рассказ о том, что я сделал
```

Именно такая модель вложенности присутствует в Less, когда один селектор, в прямом смысле слова, вкладывается в другой селектор. Таким образом получается легко поддерживаемая, читаемая и приятная глазу структура. Если же попытаться спроецировать такую модель на CSS-код, то она будет иметь вид:

```
.class-1 {
  property: value;

  .class-2 {
    property: value;
  }

  .class-3 {
    property: value;
  }
}
```

Для большей наглядности я предлагаю обратиться к конкретному примеру, в котором я постараюсь сопоставить классический CSS и Less код.

## Пример 2.2.2

Здесь я переписал код из *примера 2.2.1*, заменяя классический синтаксис на препроцессорный.

```
.global-header {
  background-color: #f5f5f5;
  color: #443d3d;
  border-bottom: 1px solid #ddd;

  h1 {
    margin-top: 0;
    margin-bottom: 0;
    font-size: 4rem;

    small {
      font-size: 2rem;
    }
  }

  .header-actions {
    background-color: #fff;
    padding-top: 10px;
    padding-bottom: 10px;
    text-align: center;
  }
}
```

На изображении ниже представлена модель вложенности, описываемая в этом примере. Для наглядности и понимания сути происходящего экран разделен на две части, где слева код написан на препроцессорном языке, а справа на «чистом» CSS.

_styles.less	styles.css
<pre>1 .global-header { 2   background-color: #f5f5f5; 3   color: #443d3d; 4   border-bottom: 1px solid #ddd; 5 6   h1 { 7     margin-top: 0; 8     margin-bottom: 0; 9     font-size: 4rem; 10 11     small { 12       font-size: 2rem; 13     } 14   } 15 16   .header-actions { 17     background-color: #fff; 18     padding-top: 10px; 19     padding-bottom: 10px; 20     text-align: center; 21   } 22 }</pre>	<pre>1 .global-header { 2   background-color: #f5f5f5; 3   color: #443d3d; 4   border-bottom: 1px solid #ddd; 5 } 6 .global-header h1 { 7   margin-top: 0; 8   margin-bottom: 0; 9   font-size: 4rem; 10 } 11 .global-header h1 small { 12   font-size: 2rem; 13 } 14 .global-header .header-actions { 15   background-color: #fff; 16   padding-top: 10px; 17   padding-bottom: 10px; 18   text-align: center; 19 } 20</pre>

## Мысли и советы

Согласитесь, теперь ваш код становится интуитивно понятным и удобным для чтения. В такой структуре сложно запутаться и потеряться среди селекторов, ведь здесь чётко видна их вложенность и не нужно помнить имя родительского селектора — за нас это делает компилятор.

### Предостережение!

Постарайтесь запомнить раз и навсегда, что вкладывать селекторы друг в друга можно бесконечно, но делать это строго **не рекомендуется!** Многие разработчики советуют следить за тем, чтобы структура, в общем случае, не превышала **трёх вложений**. Нет необходимости вкладывать селекторы, начиная от родительского, на такую глубину. Максимально допустимый уровень, в крайних случаях, *пять вложений*. Старайтесь избегать крайних случаев, если это действительно не требуется.



## Ссылка на родителя селектора

Благодаря вложенности, мы можем писать бесподобный структурированный код, который будет понятен всем разработчикам и даже тому, кто его написал. Но сейчас у нас на повестке дня стоит вопрос о том, как в этой модели вложенности работать с псевдоклассами, псевдоэлементами и комбинированными селекторами. Для этого в Less есть специальный символ — `&` (родительский селектор).

Символ `&` играет особую и важную роль в Less. С помощью этого символа можно обращаться к родителю текущего селектора, а также сшивать, объединять их и создавать внутри них области видимости (подробнее в главе 3).

## Прямое объединение селекторов

Родительский селектор может использоваться для объединения любых селекторов, псевдоэлементов и псевдоклассов.

### Пример 2.3.1

Наиболее часто, символ `&` применяется для добавления псевдоклассов к селекторам. Например, с помощью него можно добавить эффект при наведении, используя псевдокласс `:hover`. Посмотрите на следующий пример:

```
a {
  color: #777;
  text-decoration: none;

  &:hover {
    color: #a31515;
  }
}
```

После компиляции селектор `a` и псевдокласс `:hover` будут объединены, а их свойства поделены между ними в соответствии с их объявлениями по вложенности:

```
a {
  color: #777;
  text-decoration: none;
}

a:hover {
  color: #a31515;
}
```

## Пример 2.3.2

Как я уже говорил раньше, объединять можно и селекторы с другими селекторами. Для примера ниже приводится код объединения `.class-1` с `.class-2`.

```
.class-1 {
  background-color: #fff;

  &.class-2 {
    color: #000;
  }
}
```

На выходе компилятора получается сдвоенный селектор:

```
.class-1 {
  background-color: #fff;
}
.class-1.class-2 {
  color: #000;
}
```

Обратите внимание, что препроцессор не запрещает использовать стандартный синтаксис `.class-1.class-2 {}`. Но при таком стиле записи теряется весь смысл и шарм, который нам дарит препроцессор.

## Обратное объединение селекторов

У автомобилей есть передний и задний ход, такой же функционал реализован и у символа `&`. Такие возможности позволяют писать более гибкий код, когда нужно добавить какой-то селектор перед другим селектором. Вместо того, чтобы писать новую структуру, можно просто использовать этот функционал.

## Пример 2.3.3

Примером может послужить поддержка IE, когда с помощью JavaScript в тег `body` или `html` добавляются классы `.ie6`, `.ie7` и им подобные. Возможность использования обратного объединения позволяет писать более гибкий код, когда нужно изменить контекст применения селектора.

Представим, что нашему проекту требуется поддержка IE7, который не умеет работать со свойством `border-image`. А по дизайну в любом случае должна быть рамка между двумя элементами, иначе они превратятся в тыкву.

На деле имеем такой код:

```
.item-card {
  background-color: #f5f5f5;
  border-image: url("images/bg-image.png") 30 round round;
}
```

Вместо того, чтобы писать новый класс `.ie7 .item-card {}` и плодить сложно поддерживаемую структуру, можно использовать принцип обратного объединения.

```
.main {
  .item-card {
    background-color: #f5f5f5;
    border-image: url("images/bg-image.png") 30 round round;

    .ie7 & {
      border: 1px solid #a31515;
    }
  }
}
```

Во время компиляции, селектор, у которого имеется ссылка на родителя справа от имени, будет опущен ниже по дереву вложенности вне зависимости от глубины вложенности. То есть при обратном объединении родительский селектор всегда ссылается на корневой селектор всего объявления.

```
.main .item-card {
  background-color: #f5f5f5;
  border-image: url("images/bg-image.png") 30 round round;
}
.ie7 .main .item-card {
  border: 1px solid #a31515;
}
```

**Внимание!**

Если написать `.ie7 &` без пробела, то после компиляции селектор будет комбинированным, а не вложенным:

```
.item-card {
  background-color: #f5f5f5;
  border-image: url("images/bg-image.png") 30 round round;
}
.ie7.item-card {
  border: 1px solid #a31515;
}
```

## Сшивание (склеивание) селекторов

Не редко требуется производить операцию склеивания имён текущего и родительского селектора. Такая практика применяется при создании новых классов на основе старого. Кроме того, такая операция просто необходима тем людям, кто использует методологию БЭМ (и ей подобные) при написании стилей.

### Пример 2.3.4

Стоит задача стилизации кнопок для проекта. Кнопки имеют различные цвета, в зависимости от действия и контекста использования. Изначально кнопки имеют серый цвет. Кнопка, предназначенная для добавления новой записи имеет зелёный цвет, а для удаления — красный.

Если опустить основные свойства, рассматривая лишь те, что необходимы для определения цвета кнопок, то получается следующий код:

```
.button {
  background-color: #ddd;
  color: #000;
}

.button-add {
  background-color: green;
  color: #fff;
}

.button-remove {
  background-color: red;
  color: #fff;
}
```

С помощью препроцессора этот код можно структуризировать, и тем самым упростить. Для этого используется сшивание селекторов. В этом случае наш код преобразуется таким образом:

```
.button {
  background-color: #ddd;
  color: #000;

  &-add {
    background-color: green;
    color: #fff;
  }

  &-remove {
    background-color: red;
    color: #fff;
  }
}
```

А ещё этот код можно оптимизировать, указав сначала общий для двух селекторов цвет текста, а уже потом индивидуальный цвет фона. Но не стоит забывать, что такая оптимизация не имеет никакого эффекта в случае, если после компиляции код будет минифицирован, так как практически любой современный минификатор умеет группировать селекторы с общими свойствами.

```
.button {
  background-color: #ddd;
  color: #000;

  &-add,
  &-remove {
    color: #fff;
  }

  &-add { background-color: green; }
  &-remove { background-color: red; }
}
```

## Многократное и комбинированное использование

Символ `&` может использоваться в селекторе больше одного раза. Это даёт возможность несколько раз ссылаться на родительский селектор, но не писать его имя.

## Пример 2.3.5

Немного академический пример, который не несёт в себе смысловой нагрузки, но всё таки отображает суть происходящего:

```
.header {
  .item {
    & + & {
      color: red;
    }

    & & {
      color: green;
    }

    && {
      color: blue;
    }

    &, &-box {
      color: yellow;
    }
  }
}
```

Важно понимать, что обращение к родительскому селектору происходит по всей его ветви селекторов, а не только к ближайшему селектору от места обращения.

```
.header .item + .header .item {
  color: red;
}
.header .item .header .item {
  color: green;
}
.header .item.header .item {
  color: blue;
}
.header .item,
.header .item-box {
  color: yellow;
}
```

Такое использование символа `&` практически нигде не используется (кроме случая `&, &-box`). Если вы его встретите где-нибудь, то будьте уверены, что сегодня ваш день.

## Пример 2.3.6

Из примера 2.3.5 следует, что с помощью такого вот взаимодействия текущего селектора с родительским, можно творить неведомые ранее чудеса. Это может пригодиться тогда, когда нужно комбинировать селекторы между собой. Наиболее ярко суть происходящего описывает пример из документации.

```
ul, li {
  border-top: 2px dotted #366;

  & + & {
    border-top: 0;
  }
}
```

Суть такого метода заключается в том, что компилятор сначала возьмёт весь перечень селекторов, добавит к нему свойства и сохранит полученный результат. В результате чего получится:

```
ul,
li {
  border-top: 2px dotted #366;
}
```

Далее компилятор встретит на своём пути два вызова родительского селектора и поймёт, что кроме них у него имеется перечень селекторов. Немного подумав, он начнёт их комбинировать, то есть перебирать все доступные комбинации. На практике это будет выглядеть следующим образом:

```
ul + ul,
ul + li,
li + ul,
li + li {
  border-top: 0;
}
```

Но что будет, если указать три вызова родительского селектора ( `& + & + &` )? Правильно, будет комбинация доступного перечня по трём позициям:

```
ul + ul + ul,  
ul + ul + li,  
ul + li + ul,  
ul + li + li,  
li + ul + ul,  
li + ul + li,  
li + li + ul,  
li + li + li {  
  border-top: 0;  
}
```

## Мысли и советы

Описанный в этой части функционал на практике имеет довольно спорные моменты. Во-первых, некоторые люди против применения возможности склеивания имён вследствие того, что это затрудняет поиск селектора в редакторах. Во-вторых, нужно осознанно ссылаться на родителя, как в случае прямого , так и в случае обратного объединения.

Также не стоит злоупотреблять возможностью комбинирования селекторов. Да, в некоторых случаях это может сэкономить время написания кода, но в итоге сэкономленное время будет тратиться в будущем на понимание происходящего.



# Группировка селекторов

Для того, чтобы уменьшить количество кода после компиляции, а также упростить работу с селекторами, в Less был введён специальный псевдокласс `:extend()`. Этот псевдокласс позволяет производить группировку селекторов (объединение) за счёт перечисления нескольких классов в одном месте, при условии, что все эти селекторы имеют общие свойства. Проще говоря, псевдокласс `:extend()` автоматизирует следующий процесс:

- Найти селекторы, у которых есть одинаковые свойства.
- Выбрать базовый селектор.
- Перечислить все найденные селекторы в объявлении базового селектора.
- Все новые селекторы добавлять в список селекторов базового объявления.

Под списком селекторов понимается последовательность селекторов, разделяемая с помощью запятой. Я более чем уверен, что вы уже встречали такие списки и даже их использовали в своих проектах.

Следующий код демонстрирует такой список:

```
.header,  
.main,  
.footer {  
  background-color: #f5f5f5;  
}
```

Раньше такие мероприятия производились вручную, и то при желании разработчика. Поверьте, это очень ресурсоёмкое занятие и самое ужасное — это когда элемент на странице меняет своё оформление. В этом случае приходится пользоваться поиском и повторять определённую последовательность действий каждый раз, когда встречается искомый селектор. Выполняются такие мероприятия лениво, неохотно и без особого энтузиазма. К счастью, разработчики препроцессора Less нашли способ борьбы с этим недугом.

Окей, а что же тогда предлагают препроцессоры? — писать код и не обращать внимание на такие мелочи. Конечно, такое говорится с оговоркой на то, что изредка придётся использовать препроцессорный псевдокласс `:extend()`.

Рассмотрим несколько тривиальных примеров для погружения в курс дела и получения первого опыта работы с `:extend()`.

## Пример 2.4.1

Чтобы понять, как работает этот псевдокласс, представим, что у нас есть три селектора и свойства у них одинаковые.

```
.class-1 {  
  background-color: #fff;  
  color: #000;  
}  
  
.class-2 {  
  background-color: #fff;  
  color: #000;  
}  
  
.class-3 {  
  background-color: #fff;  
  color: #000;  
}
```

В этом случае можно полностью объединить `.class-1`, `.class-2` и `.class-3`. Если провести такие манипуляции с нашим примером, то получится следующий код:

```
.class-1,  
.class-2,  
.class-3 {  
  background-color: #fff;  
  color: #000;  
}
```

Less предлагает эквивалентное решение, но в тоже время более простое. Он говорит нам, что нужно объявить полностью лишь селектор `.class-1`, а другие сгруппировать с ним. Для этого предлагается использовать `:extend()`.

```
.class-1 {  
  background-color: #fff;  
  color: #000;  
}  
  
.class-2:extend(.class-1) {}  
  
.class-3 {  
  &:extend(.class-1);  
}
```

Важно понимать, что использовать `:extend()` можно как и любой другой псевдокласс. Имеется в виду, что допустимо как прямое написание ( `.class:extend()` ), так и с использованием родительского селектора ( `.class { &:extend() }` ).

Проведя процедуру компиляции на выходе мы получим все тот же CSS, который и получили при ручной оптимизации:

```
.class-1,  
.class-2,  
.class-3 {  
  background-color: #fff;  
  color: #000;  
}
```

## Пример 2.4.2

Можно указывать как один селектор для расширения, так и несколько.

```
.class-1 { background-color: #fff; }  
.class-2 { background-color: #fff; }  
  
.selector {  
  &:extend(.class-1, .class-2);  
}
```

В результате такого объявления класс `.selector` будет группироваться и с первым ( `.class-1` ) селектором, и со вторым ( `.class-2` ).

```
.class-1,  
.selector {  
  background-color: #fff;  
}  
.class-2,  
.selector {  
  background-color: #fff;  
}
```

## Расширение правил

Селекторы можно не только группировать, но и расширять. Все эти громкие слова на самом деле пляшут вокруг все того же `:extend()`. Ранее рассматривались примеры, в которых все свойства у селекторов были одинаковыми и это те самые идеальные

случаи. На практике такое встречается редко, но все таки встречается. Чаще всего попадают другие случаи, когда часть свойств совпадают, а часть — нет. В этом случае опять пригодится этот же псевдокласс.

## Пример 2.4.3

В студии «Артемка и КО» разрабатывается дизайн для продающего лендинга известной компании, производящей конфеты.

```
.global-header {
  background-color: #fff;
  color: #000;
}

.global-navigation {
  border: 1px solid #ddd;
}
```

По дизайну, селектору `.global-navigation` полагается белый фон и чёрный цвет у текста. Можно было бы просто скопировать недостающие свойства, но по желанию верстальщика решили использовать `:extend()`, тем самым получив недостающие свойства.

```
.global-header {
  background-color: #fff;
  color: #000;
}

.global-navigation {
  &:extend(.global-header);
  border: 1px solid #ddd;
}
```

По сути своей, это одно и то же, что и группировка селекторов, но я должен был рассказать об этом. Сделано это для того, чтобы вы не подумали о том, что группировать можно лишь те селекторы, у которых совпадают свойства.

## Группировка с цепочкой селекторов

Препроцессоры настолько умные, что в состоянии уследить за цепочкой селекторов с одним именем. Делается это с помощью все того же псевдокласса `:extend()`, но с добавлением некоего параметра `all`.

## Пример 2.4.4

Студия «Артемка» получила свои первые правки по макету для компании, производящей конфеты. Заказчик хочет, чтобы подвал сайта имел такие же стили, как и шапка сайта.

```
.global-header {  
  background-color: #fff;  
  
  .area {  
    text-align: center;  
  }  
}  
  
.global-header:hover {  
  background-color: #000;  
}
```

Верстальщик вспоминает о возможности группировки селекторов `.global-footer` со всеми селекторами, имеющими имя `.global-header`. На ум ему приходит только одно:

```
.global-footer {  
  &:extend(.global-header);  
}
```

Однако, после компиляции он получает не совсем желанный результат. Селектор `.global-footer` группируется лишь с самым первым селектором, а остальные игнорируются.

```
.global-header,  
.global-footer {  
  background-color: #fff;  
}  
.global-header .area {  
  text-align: center;  
}  
.global-header:hover {  
  background-color: #000;  
}
```

Для решения этой проблемы ему необходимо использовать параметр `all`, говорящий компилятору о том, что селектор хочет сгруппироваться со всеми совпадающими по имени селекторами. Передать этот параметр нужно следующим образом:

```
.global-footer {
  &:extend(.global-header all);
}
```

В итоге получается требуемый результат. Студия «Артемка» продолжает работу над проектом, а заказчик продолжает присылать желанные правки.

```
.global-header,
.global-footer {
  background-color: #fff;
}
.global-header .area,
.global-footer .area {
  text-align: center;
}
.global-header:hover,
.global-footer:hover {
  background-color: #000;
}
```

## Контекстная группировка

Less поддерживает группировку с конкретным селектором. Речь идёт про то, что `:extend()` может принимать в качестве цели не только название селектора, но и селектор с необходимой вложенностью, контекстом и некоторыми другими параметрами.

Псевдокласс `:extend()` умеет работать с:

- `nth`: `:extend(:nth-child(n+3))` .
- атрибутами: `:extend([title=identifier])` .
- псевдоклассами: `:extend(link:visited:hover)` .
- псевдоэлементами: `:extend(link:before)` .

### Пример 2.4.5

Верстальщик получает макет, в котором стили оформления товаров в каталоге и статей в блоге имеют общие стили, но лишь в шапке.

```
.item {
  background-color: #fff;
  border: 1px solid #ddd;

  .header {
    padding: 25px;
  }
}
```

Ему хочется унаследовать стили `.item .header`, но не хочется городить лишнего кода. Приходится использовать контекстную группировку с применением уже знакомого нам псевдокласса `:extend()`.

```
.item {
  background-color: #fff;
  border: 1px solid #ddd;

  .header {
    padding: 25px;
  }
}

.article {
  &:extend(.item .header);
}
```

После компиляции получается код, который полностью соответствует ожиданиям верстальщика и его требованиям:

```
.item {
  background-color: #fff;
  border: 1px solid #ddd;
}
.item .header,
.article {
  padding: 25px;
}
```

## Мысли и советы

Пожалуй, это самый объёмный по материалу функционал для изучения, предоставляемый CSS-препроцессором Less. В то же время, это наименее используемый функционал в реальных проектах, так как приходится запоминать имена

селекторов для группировки, а при достаточно объёмном проекте это не позволительная роскошь.

Я бы советовал аккуратно работать с `:extend()` и использовать его лишь тогда, когда это действительно оправданный шаг. Благодаря известному всем разработчикам фреймворку Bootstrap, я стал использовать `:extend()` лишь для группировки селекторов с селектором `.clearfix`. Такой подход позволяет исключить появление дополнительных классов в документе, тем самым делая структуру более чистой.



# Использование медиавыражений

В 2015 году непозволительно иметь сайт, ориентирующийся только на пользователей стационарных ПК и ноутбуков. Благодаря доступности мобильных устройств и интернета, а также повсеместного внедрения Wi-Fi сетей, сайты всё больше получают «мобильных» посетителей. Я уже давно стал замечать, что использую ноутбук лишь для производства контента, а его потребление происходит, в основном, с помощью планшета и, изредка, смартфона.

Для того, чтобы ориентировать сайт на все платформы нужно использовать медиавыражения, которые определяются с помощью директивы `@media` .

## Медиавыражения

Пожалуй, перед тем, как бросаться на рассмотрение возможностей Less, которые помогут работать с медиавыражениями, я напомним, что они из себя представляют.

**Медиавыражения** (от англ. media query) — это объявление директивы `@media` с одним или несколькими условиями, в зависимости от которых определяется будут ли срабатывать объявления внутри этой директивы или нет.

В простейшем случае медиавыражение имеет такой вид:

```
@media (min-width: 992px) {  
  .class {  
    display: none;  
  }  
}
```

Если при объявлении директивы используется несколько условий, то они объединяются с помощью ключевого слова `and` :

```
@media (min-width: 768px) and (orientation: landscape) {  
  .class {  
    display: none;  
  }  
}  
  
@media tv and (min-width: 992px) and (orientation: landscape) {  
  .class {  
    display: block;  
  }  
}
```

Такие конструкции позволяют применять те или иные свойства к элементам, в зависимости от выполнения некоторых условий. В основном, нас интересует: разрешение окна браузера, ориентация устройства и пиксельное соотношение (от англ. pixel ratio). Существуют и другие, но они применяются крайне редко.

## Медиавыражения в Less

Препроцессор Less не привносит в синтаксис медиавыражений кардинальных изменений. Различие с CSS заключается в том, что здесь доступно два контекста использования и присутствует несколько особенностей. Рассматривать их отдельно и приводить примеры я не буду, а вот об особенностях поговорим подробнее, но позднее.

**Стандартный контекст использования** такой же, как и в классическом CSS:

```
@media (min-width: 992px) {  
  .class {  
    display: none;  
    // какие-то другие стили  
  }  
}
```

**Стандартный для Less контекст использования.** Обратите внимание, медиавыражение вложено в селектор. А как уже известно, на этом и строится вся магия Less.

```
.class {  
  .two {  
    @media (min-width: 992px) {  
      display: none;  
      // какие-то другие стили  
    }  
  }  
}
```

После работы компилятора имеем следующий CSS-код:

```
@media (min-width: 992px) {  
  .class .two {  
    display: none;  
  }  
}
```

Второй вариант использовать предпочтительнее, ввиду того, что он не требует слежения за правильностью структуры и вложенности. При этом нет необходимости исправлять названия селекторов, если они поменяются в процессе разработки, ведь компилятор сам уследит, к чему относится директива `@media`, а также объявленные внутри неё свойства и селекторы.

Кроме положительных черт, у этого метода есть и отрицательные. Заключаются они в том, что разработчик практически не в состоянии повлиять на структуру вложенности. Единственный способ — это использование ссылки на родительский селектор. С другой стороны, желание повлиять на структуру медиавыражения возникает очень редко и на деле оно излишне, конечно, если не вдаваться в дебри оптимизации.

## Вложенные медиавыражения

В самом начале главы вы узнали о том, что селекторы можно вкладывать друг в друга. Так вот, в этом случае медиавыражения схожи с селекторами. Вложенные конструкции будут склеиваться в одно медиавыражение с несколькими условиями. То есть, условия будут объединяться, а на каждом уровне вложенности будет собираться своё отдельное объявление директивы `@media`.

### Пример 2.5.1

Рассмотрим пример того, как происходит компиляция вложенных медиавыражений. Напишем какое-нибудь непростое вложенное медиавыражение с вложенными селекторами:

```
.one {
  @media (min-width: 768px) {
    background-color: #f5f5f5;

    .two {
      @media (max-width: 992px) {
        color: #000;
      }
    }
  }
}
```

Когда компилятор начнёт обрабатывать такую структуру, сначала будет создано медиавыражение с условием `(min-width: 768px)` и его свойствами, а уже потом со списком условий `(min-width: 768px) and (max-width: 992px)`.

```
@media (min-width: 768px) {
  .one {
    background-color: #f5f5f5;
  }
}
@media (min-width: 768px) and (max-width: 992px) {
  .one .two {
    color: #000;
  }
}
```

### Предупреждение!

На практике такую возможность лучше не использовать. Да, это круто, когда есть способ слегка оптимизировать код в исходниках, но за счёт его читаемости. После компиляции всё равно будет два отдельных медиавыражения, то есть никакой выгоды в итоге вы не получите. Поэтому не стоит ухудшать читаемость кода за счёт сомнительной и бесполезной оптимизации.

## Медиавыражения и группировка селекторов

Директива `@media` представляет собой замкнутую систему. Из школьного курса физики или химии известно, что замкнутая система — это такая система, которая не обменивается с окружающей средой ни веществом, ни энергией. В нашем случае энергией и веществом будут свойства и селекторы, объявленные внутри директивы. В идеальном случае, медиавыражения не могут воздействовать друг на друга.

Получается, что группировать селекторы можно только внутри такой системы, не выходя за её пределы. Но, так как в реальности абсолютно замкнутых систем не бывает, воздействовать на неё извне мы сможем.

## Пример 2.5.2

Давайте исходить от противного и экспериментально подтвердим факт того, что медиавыражения — это замкнутая система. Допустим, что у нас есть структура, в которой объявлены два вложенных медиавыражения, а также отдельно стоящий селектор.

```
@media screen {
  .one {
    &:extend(.three, .two);
    background-color: #fff;
  }

  @media (min-width: 992px) {
    .two {
      &:extend(.one);
      color: #777;
    }
  }
}

.three {
  border-right: 1px solid #000;
}
```

Если выполнить компиляцию этого кода, то выполнится процедура объединения условий медиавыражения (пример 2.5.1), а вот группировки селекторов не произойдёт. Причём группировка не наблюдается ни в первом, ни во втором случае. Кроме того, изменения не коснутся и внешнего селектора. Отлично, значит утверждение о замкнутой системе верное.

```
@media screen {
  .one {
    background-color: #fff;
  }
}
@media screen and (min-width: 992px) {
  .two {
    color: #777;
  }
}
.three {
  border-right: 1px solid #000;
}
```

## Пример 2.5.3

В этом примере будем рассматривать случай, когда мы можем воздействовать на замкнутую систему (медиавыражения). Для этого повторим поставленный в примере 2.5.2 эксперимент, но добавим группировку селекторов не в сами медиавыражения, а в отдельно стоящий селектор.

```
@media screen {
  .one {
    background-color: #fff;
  }

  @media (min-width: 992px) {
    .two {
      color: #777;
    }
  }
}

.test {
  &:extend(.one, .two);
  border-right: 1px solid #000;
}
```

После работы компилятора на выходе имеем CSS-код, в котором группировка селекторов всё таки выполняется. Происходит это из-за того, что извне воздействовать на медиавыражения можно.

```
@media screen {  
  .one,  
  .test {  
    background-color: #fff;  
  }  
}  
@media screen and (min-width: 992px) {  
  .two,  
  .test {  
    color: #777;  
  }  
}  
.test {  
  border-right: 1px solid #000;  
}
```

# Импорт стилей

Если вы хоть раз подключали таблицы стилей через директиву `@import` в CSS, то эта часть главы полностью изменит ваше представление о ней, так как в Less все куда интереснее и гибче. Если же вы никогда ранее не делали импорт стилей в CSS, то добро пожаловать в этот удивительный мир, позволяющий разбивать таблицы стилей на несколько частей.

## Импорт стилей в CSS

В CSS директива `@import` позволяет импортировать стили из других таблиц. Проще говоря, можно разбить одну большую таблицу стилей на несколько маленьких.

Делается это следующим образом:

```
@import url("имя файла");
@import "имя файла";
```

Самым важным минусом выступает тот факт, что такие подключения должны предшествовать другим стилям в таблице, где подключается дополнительный файл. То есть сделать так, как написано в коде ниже **нельзя**:

```
.class {
  background-color: #fff;
}

@import "имя файла";
```

Разрешено делать только так:

```
@import "имя файла";

.class {
  background-color: #fff;
}
```

Причём имя файла должно быть с расширением `.css`.

Я не буду вдаваться в подробности и обойду стороной возможность указывать типы носителей, тем более они сейчас нас не интересуют, да и на практике такое подключение дополнительных таблиц стилей не приветствуется.



# Импорт стилей в Less

В Less импорт стилей происходит с помощью всё той же директивы, но с расширенным функционалом. Перед именем файла можно указывать (необязательно) ключевое слово, которое указывает компилятору, как ему поступать с файлом.

```
@import (keyword) "имя файла";
```

Тем более, вы можете комбинировать ключевые слова для достижения определённых целей. Например, если нужно использовать css-файл как less-файл, но при этом не выводить его содержимого.

Во-первых, в Less не регламентируется то, где возможно подключение других таблиц стилей. Вы можете использовать директиву `@import` до объявления селекторов, после или даже между ними:

```
.one {  
  background-color: #000;  
}  
  
@import "имя файла";  
  
.two {  
  background-color: #fff;  
}
```

Во вторых, расширение файла может быть любым, главное — чтобы в нём содержался валидный CSS- или Less-синтаксис. Но здесь начинают действовать специфичные для Less правила:

## Файлы с расширением `.css` :

Если при подключении файла с помощью директивы `@import` будет указано расширение `.css`, то такой файл подключается как обычный css-файл и не обрабатывается компилятором.

## Файлы без расширения:

Если при подключении файла с помощью директивы `@import` у него не будет указано расширение, то такой файл подключается как less-файл и обрабатывается компилятором.

## Файлы с другими расширениями:

Если при подключении файла с помощью директивы `@import` у него будет указано расширение, но оно не соответствует ни `.css` ни `.less`, то такой файл подключается как less-файл и будет обрабатываться компилятором.

Все это вкупе с доступными опциями для директивы `@import`, которые будут рассматриваться позднее, даёт вам возможность построения гибкой и хорошо поддерживаемой структуры проекта. Про структуру проекта я буду рассказывать намного позднее, так как без полного понимания доступных возможностей Less она вам не пригодится.

## Пример 2.6.1

Рассмотрим пример, который отображает всю суть импорта стилей в Less. Для этого создадим в директории `import/` следующие файлы: `_duckduckgo.less`, `_mail.css`, `_yandex.less` и `_yahoo.less`. В этих файлах объявим одноимённые с названиями файлов классы и укажем с помощью свойства `color` официальный цвет сервиса. Кроме них нам понадобится файл `_styles.less`, к которому будут подключаться эти файлы.

Для наглядности я предлагаю посмотреть на карту директории этого примера:

```
2.6.1/
├─ import/
│   ├─ _duckduckgo.less
│   ├─ _mail.css
│   ├─ _yahoo.less
│   └─ _yandex.less
└─ _styles.less
```

Теперь я предлагаю взглянуть на содержимое файла `_styles.less` и то, что получилось после его компиляции.

**Содержимое файла `_styles.less` :**

```
@import "import/_mail.css";
@import "import/_duckduckgo";

.canonium {
  color: #53599a;
}

@import "import/_yahoo";
@import "import/_yandex";
```

Как не сложно заметить, подключение таблиц стилей происходит до и после содержимого файла, что не допускается в CSS, но разрешено в Less.

**Содержимое файла `styles.css` , полученное после компиляции:**

```
@import "import/_mail.css";
.duckduckgo {
  color: #de5833;
}
.canonium {
  color: #53599a;
}
.yahoo {
  color: #400191;
}
.yandex {
  color: #ffcc00;
}
```

Во время компиляции происходит конкатенация содержимого файлов. А файлы с расширением `.css` подключаются стандартным для CSS способом.

## Опции импорта

С помощью ключевых слов (опций) можно управлять тем, как компилятор будет обрабатывать файлы. Например, с помощью таких опций можно заставить компилятор конкатенировать содержимое CSS-файла, а не подключать его с помощью директивы.

Я предлагаю подробнее остановиться на каждом ключевом слове и посмотреть примеры их работы. Все примеры будут основываться на структуре, представленной в примере 2.6.1.

### Опция (`less`)

С помощью этого ключевого слова можно попросить компилятор рассматривать подключаемый файл как less-файл, то есть производить его компиляцию, а также конкатенацию с тем файлом, где происходит его подключение.

Может пригодиться при использовании файлов, которые имеют не стандартное расширение, например, `.variables` или `.mixin`.

### Пример 2.6.2

Необходимо подключить файл `_mail.css` :

```
.mail {  
  color: #168de2;  
}  
  
.mail .orange {  
  color: #ffa930;  
}
```

Если подключить этот файл как раньше, с помощью импорта без ключевых слов (`@import "import/_mail.css";`), то файл будет подключён как обычный CSS-файл в файле `styles.css`.

```
@import "import/_mail.css";  
.canonium {  
  color: #53599a;  
}
```

Но нам необходимо провести конкатенацию этих двух файлов. Для этого требуется указать ключевое слово `less`. Тогда файл `styles.css` приобретает необходимый нам вид:

```
.canonium {  
  color: #53599a;  
}  
.mail {  
  color: #168de2;  
}  
.mail .orange {  
  color: #ffa930;  
}
```

## Опция (css)

Полная противоположность опции (`less`). На этот раз мы можем заставить любой файл подключаться стандартным для CSS способом.

### Пример 2.6.3

На этот раз стоит задача подключения файла стандартным для CSS способом. Без лишних слов. Просто посмотрите на код:

```
@import (css) "import/_duckduckgo.less";

.canonium {
  color: #53599a;
}
```

После компиляции:

```
@import "import/_duckduckgo.less";
.canonium {
  color: #53599a;
}
```

## Опция (reference)

Замечательная опция, позволяющая использовать less-файлы, но не выводить их содержимое до тех пор, пока оно не будет явно вызвано. Пригодится в тех случаях, когда нужно использовать определённый селектор, а остальное содержимое файла не нужно. Такое поведение положительно сказывается на работе с библиотеками, которые имеют избыточный функционал в виде множества селекторов, а вам необходимы лишь некоторые из них.

## Пример 2.6.4

Для демонстрации работы этого ключевого слова используем группировку селекторов. Помимо этого, можно использовать примеси, но мы их пока затрагивать не будем.

Немного изменим файл `_yandex.less`, чтобы можно было продемонстрировать работу более наглядно:

```
.yandex {
  color: #ffcc00;
}

.yandex-topbar {
  background-color: #e4491b;
}
```

В файле `_styles.less` добавим ключевое слово `(reference)` :

```
.canonium {  
  color: #53599a;  
}  
  
@import (reference) "import/_yandex";
```

Если сейчас скомпилировать этот файл, то кроме селектора `canonium` в нём ничего не будет. Необходимо добавить явный вызов селектора.

Добавим явный вызов в файл `_styles.less`, в виде псевдокласса `:extend()`. Делается это следующим образом:

```
.canonium {  
  color: #53599a;  
}  
  
@import (reference) "import/_yandex";  
  
.topbar {  
  &:extend(.yandex-topbar);  
}
```

Теперь, после компиляции в получившемся CSS-файле у класса `.topbar` появятся все свойства класса `.yandex-topbar`, объявленного в импортируемом файле. При этом класс `.yandex` в этот файл добавлен не будет.

```
.canonium {  
  color: #53599a;  
}  
.topbar {  
  background-color: #e4491b;  
}
```

## Опция (inline)

Задача этой опции сказать компилятору, что разработчик ожидает на выходе подключённый файл, но без обработки компилятором. Такая опция может пригодиться при подключении CSS-файла, в котором присутствуют конструкции, которые в Less необходимо преобразовывать. Например, используемое в IE свойство `filter: ms:alwaysHasItsOwnSyntax.For.Stuff();`, требующее экранирования в Less.

## Пример 2.6.5

Представим, что перед нами стоит задача разработки проекта, использующего фильтры. Пусть к такому проекту подключается какая-нибудь библиотека, которая существует в виде CSS-файла. Тогда у нас могут возникнуть некоторые проблемы.

Добавим в файл `_mail.css` класс, содержащий свойство `filter` :

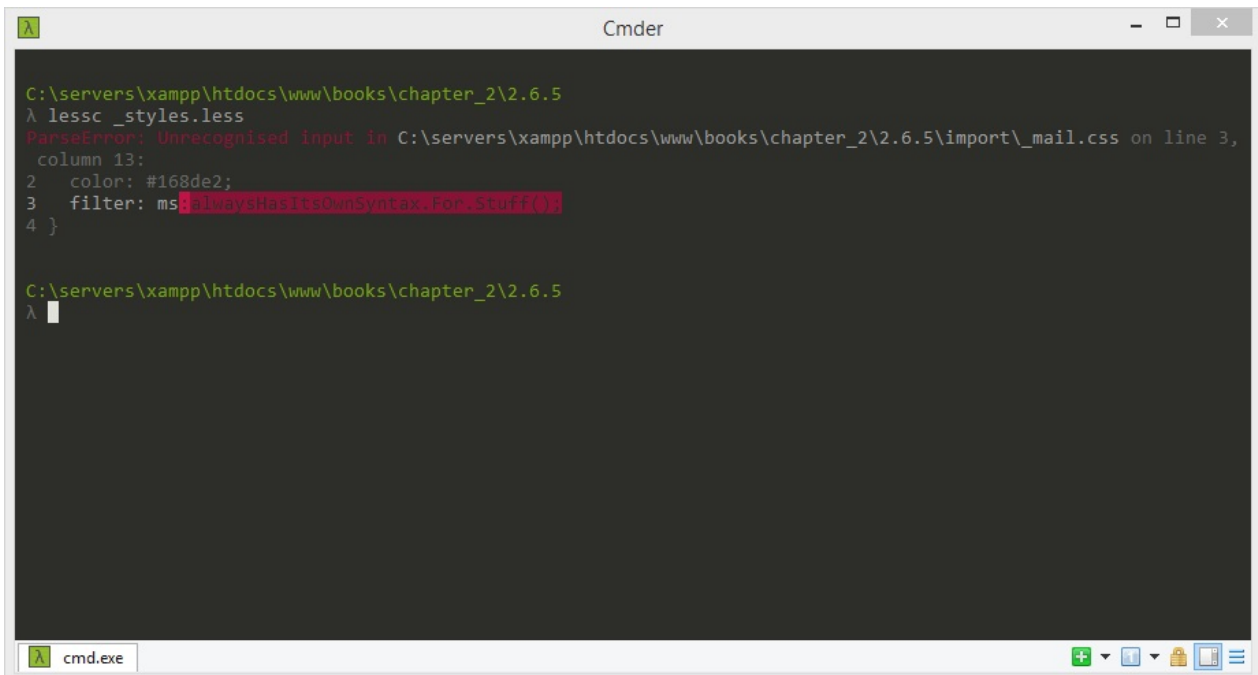
```
.mail {
  color: #168de2;
  filter: ms:alwaysHasItsOwnSyntax.For.Stuff();
}
```

Помимо этого, добавим ключевое слово `(less)` в файле `_styles.less` , чтобы CSS-файл склеивался с less-файлом:

```
@import (less) "import/_mail.css";

.canonium {
  color: #53599a;
}
```

Производим компиляцию стандартным способом. Но не тут-то было, вместо счастливого финала наш компилятор как-то приуныл и выдал ошибку. В ней говорится, что он не понимает предоставленный синтаксис и ему хочется, чтобы мы его исправили.



```
C:\servers\xampp\htdocs\www\books\chapter_2\2.6.5
λ lessc _styles.less
ParseError: Unrecognised Input in C:\servers\xampp\htdocs\www\books\chapter_2\2.6.5\import\_mail.css on line 3,
column 13:
2   color: #168de2;
3   filter: ms:alwaysHasItsOwnSyntax.For.Stuff();
4 }

C:\servers\xampp\htdocs\www\books\chapter_2\2.6.5
λ
```

Именно сейчас и пригодится рассматриваемая опция `(inline)` . Дописываем её в директиву `@import` через запятую:

```
@import (less, inline) "import/_mail.css";
```

Повторяем процесс компиляции, и на этот раз все будет хорошо. Компилятор после сложного трудового дня немного отдохнёт, а мы насладимся его работой:

```
.mail {  
  color: #168de2;  
  filter: ms:alwaysHasItsOwnSyntax.For.Stuff();  
}  
.canonium {  
  color: #53599a;  
}
```

## Опция (once) и (multiple)

Я не зря объединил эти две опции. Они представляют собой абсолютную дуальную пару, то есть они полностью противоположны по значению.

**Ключевое слово (once)** запрещает многократное подключение файла с таким именем. Эта опция включена в Less изначально, и прописывать её в директиве `@import` не нужно.

**Ключевое слово (multiple)** разрешает многократное подключение файла с таким именем.

## Пример 2.6.6

Попробуем провести эксперимент. Сначала объявим директиву `@import` без опций, потом запретим многократное подключение с помощью `(once)`, а затем разрешим, используя `(multiple)`. После этого обговорим полученные результаты. Приступаем.

Объявим директиву `@import` **без опций**:

```
@import "import/_yahoo";  
  
.canonium {  
  color: #53599a;  
}  
  
@import "import/_yahoo";
```

В полученном результате файл подключился ровно один раз, так как все директивы импорта файлов содержат опцию `(once)` по умолчанию.



```
.yahoo {  
  color: #400191;  
}  
.canonium {  
  color: #53599a;  
}
```

Объявим директиву `@import` с опцией **(once)**:

```
@import (once) "import/_yandex";  
  
.canonium {  
  color: #53599a;  
}  
  
@import (once) "import/_yandex";
```

И снова тот же результат. Очевидно, что опция `(once)` все таки установлена по умолчанию, а её повторное применение ничего не меняет.

```
.yandex {  
  color: #ffcc00;  
}  
.canonium {  
  color: #53599a;  
}
```

Объявим директиву `@import` с опцией **(multiple)**:

```
@import (multiple) "import/_duckduckgo";  
  
.canonium {  
  color: #53599a;  
}  
  
@import (multiple) "import/_duckduckgo";
```

На этот раз скомпилированный CSS-код содержит селектор `.duckduckgo` дважды, а это значит, что опция `(multiple)` сработала.

```
.duckduckgo {
  color: #de5833;
}
.canonium {
  color: #53599a;
}
.duckduckgo {
  color: #de5833;
}
```

## Опция (optional)

Эта опция позволяет продолжать компиляцию, если подключаемый файл не найден. Если не использовать это ключевое слово, то при отсутствии файла компилятор будет бросаться в вас ошибкой `FileError`. Скорее всего, такой функционал пригодится тем, кто строит фреймворк, основанный на ограниченном количестве модулей и компонентов, которые можно подключать и отключать.

## Пример 2.6.7

В этом примере есть обязательный для компиляции файл `_duckduckgo.less`, а также два необязательных: `_yahoo.less` и `_yandex.less`. Один из этих файлов будет удалён.

```
@import "import/_duckduckgo";
@import (optional) "import/_yahoo";
@import (optional) "import/_yandex";

.canonium {
  color: #53599a;
}
```

Пусть будет отсутствовать файл `_yahoo.less`. Тогда после компиляции получится следующий CSS-код:

```
.duckduckgo {
  color: #de5833;
}
.yandex {
  color: #ffcc00;
}
.canonium {
  color: #53599a;
}
```

## Мысли и советы

Несколько советов, которые позволят сделать ваш код лучше на этом этапе:

- Не указывайте расширение файлов без явной на то нужды, это делает структуру более приятной для чтения.
- Создайте один файл, в котором будут объявлены все директивы импорта файлов в проекте.

## Домашнее задание

В этой главе вы узнали, как можно работать с селекторами в Less, использовать медиавыражения и импортировать файлы друг в друга. Для закрепления полученных знаний я советую выполнить несложное домашнее задание.

### Постановка задачи

Необходимо разработать главную страницу простейшей галереи, используя все полученные знания из этой главы.

### Техническое задание

Главная страница галереи адаптируется под необходимые устройства и имеет четыре основных состояния отображения, каждое из которых подробно рассматривается ниже.

Так как это домашнее задание, а не реальный проект, то для упрощения его выполнения, я предоставляю всю необходимую информацию заранее:

#### Стилевое оформление:

- Фон области с заголовком: `#ecf0f1` ;
- Фон области с картинками: `#bdc3c7` ;
- Расстояние между картинками: `20px` ;
- При фокусе и наведении мышки на картинки их прозрачность должна быть `0.5` ;

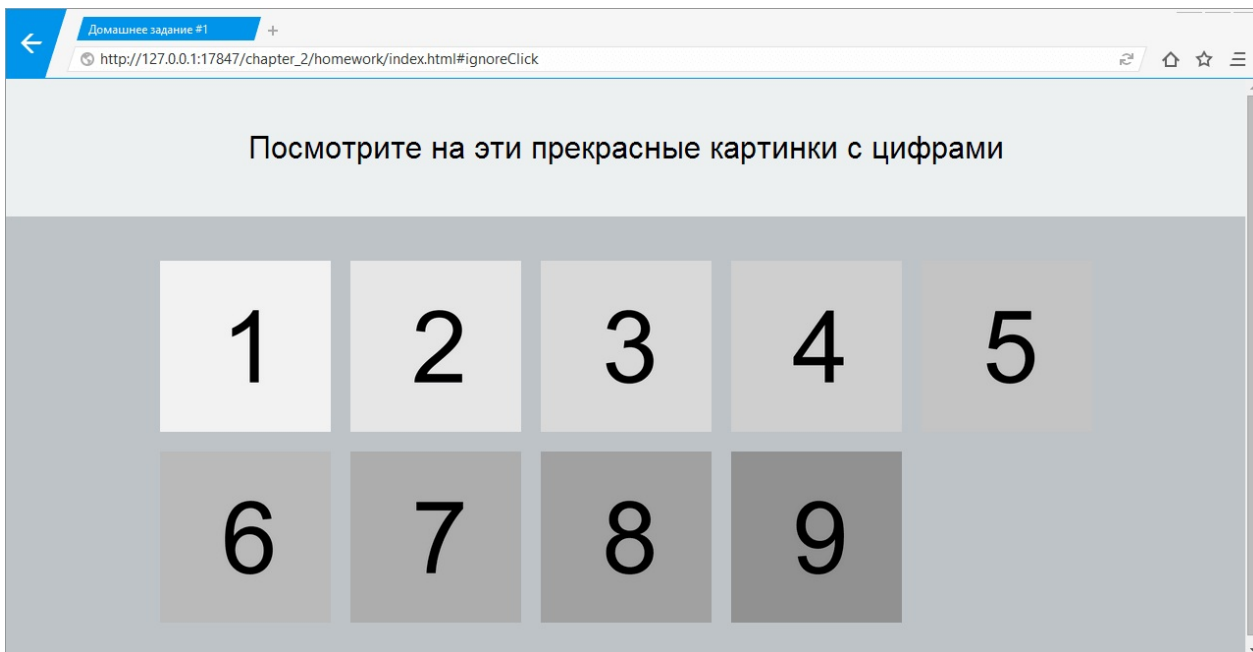
#### Размеры контейнеров и точки для медиавыражений:

- Настольные компьютеры и ноутбуки: `992px` и больше (контейнер: `970px` );
- Планшеты: от `768px` до `992px` (контейнер: `750px` );
- Смартфоны: от `540px` до `768px` ;
- Мобильные телефоны: от `0px` до `540px` ;

Размер шрифта заголовка, сам шрифт, его цвет и отступы не имеют значения в рамках этой задачи. Кроме того, картинки представленные ниже на скриншотах предлагается заменить на свои. Лучше всего, если это будут картинки котиков.

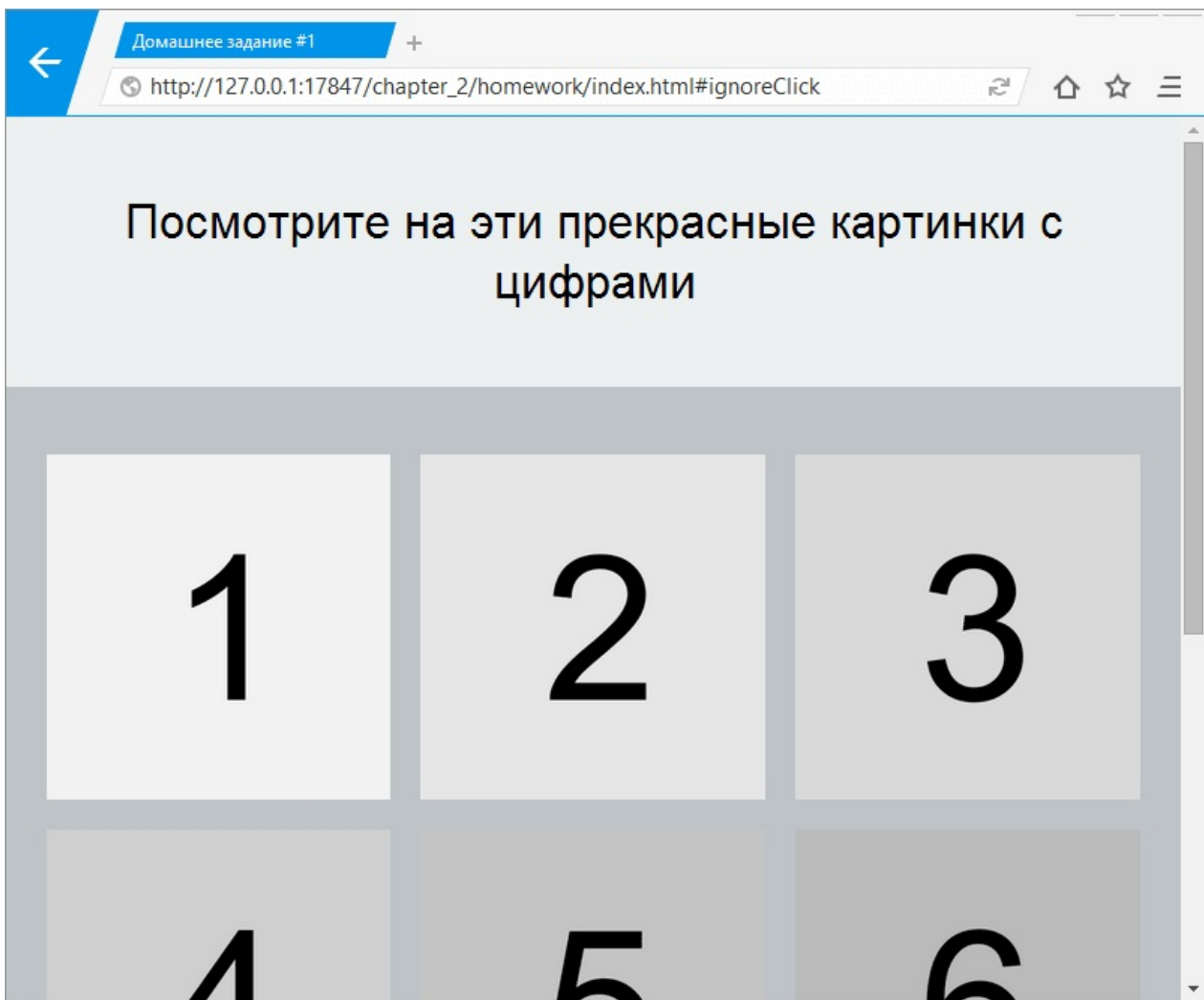
### Настольные компьютеры и ноутбуки

При разрешении окна браузера больше `992px` требуется, чтобы контент располагался посередине страницы и занимал `970px` ширины. При этом картинки располагались в ряд по **пять** штук и занимали всю предоставленную им площадь.



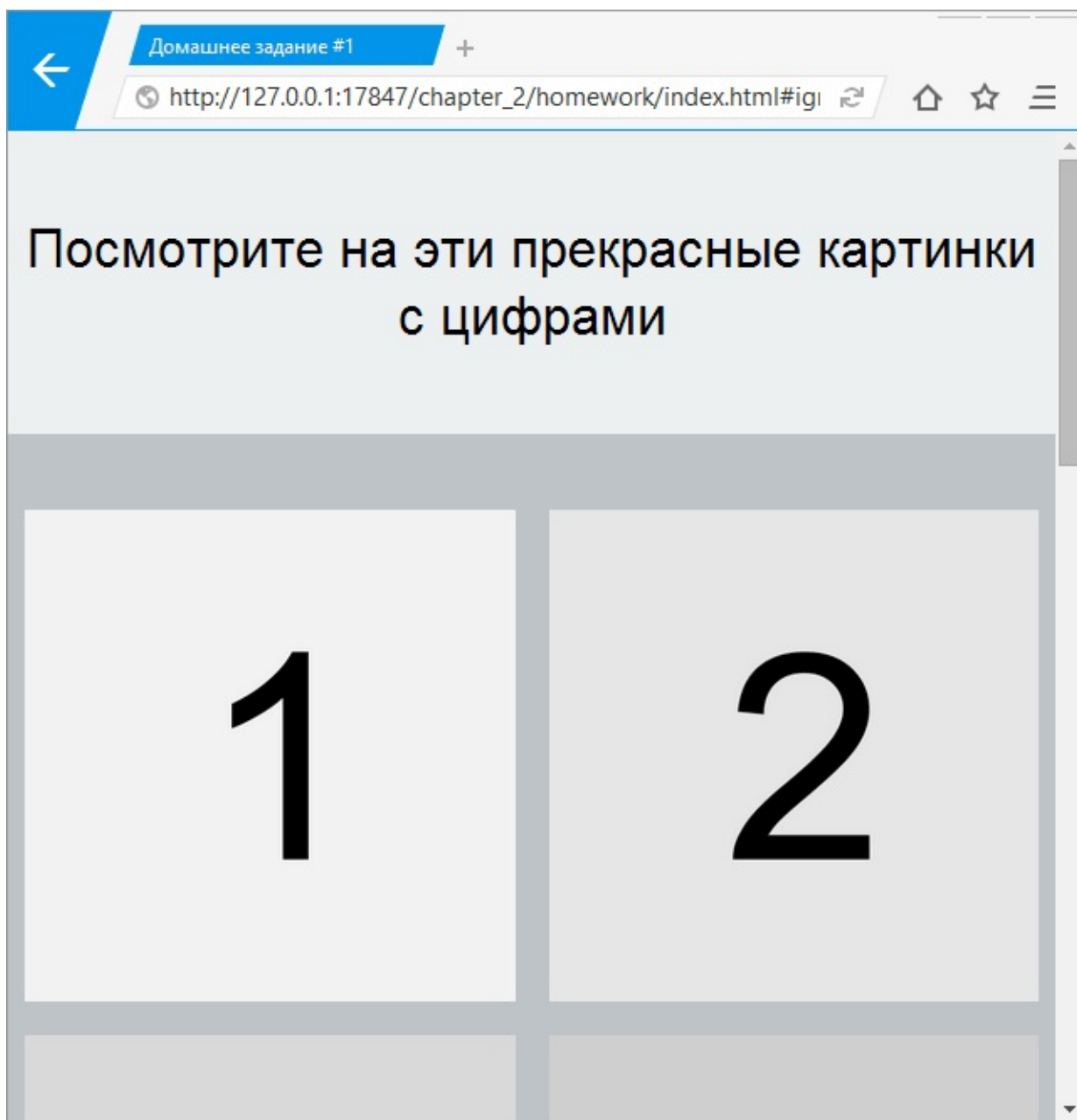
## Планшеты

На планшетных компьютерах требуется, чтобы все условия, описанные ранее, соблюдались в полной мере, с той лишь разницей, что ширина области контента равнялась `750px`, а картинки располагались по **три** в ряд.



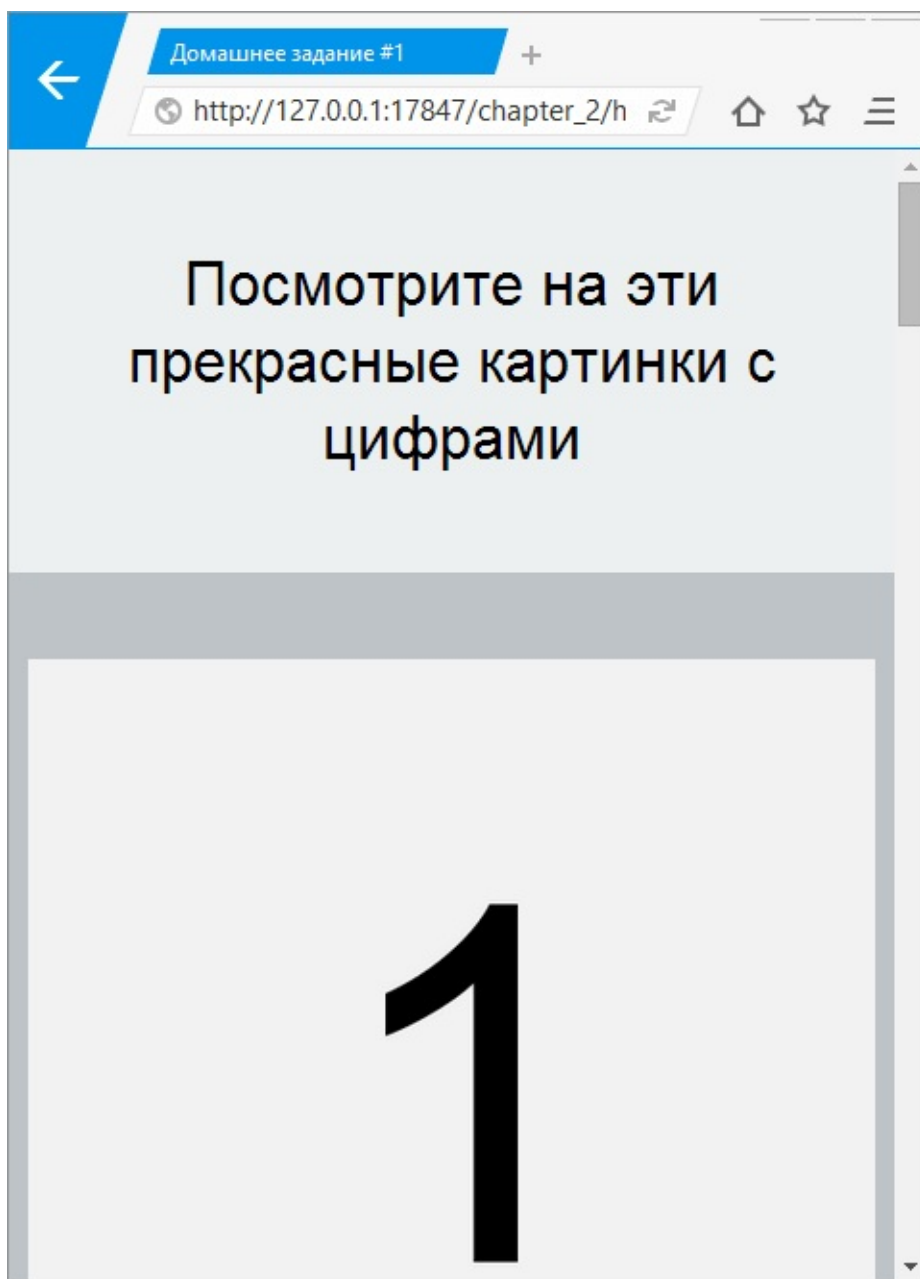
## Смартфоны

Необходимо, чтобы на смартфонах картинки располагались по **две** в ряд, а область контента занимала всю доступную ширину окна браузера.



## Мобильные телефоны

Мобильные телефоны должны отображать картинки по **одной** в строке, то есть друг за другом в столбце. Область контента должна занимать всю доступную ширину окна браузера.



## Советы

При желании используйте все доступные вам технологии и приёмы. Желательно, чтобы в решении использовались (но не обязательно):

- Блочная модель `border-box` ;
- [normalize.css](#);

Не волнуйтесь, у вас все получится! Если у вас возникают какие-либо сложности, то посмотрите решение этого задания в архиве, который прилагается к этой книге.

### Внимание

Решение этого домашнего задания будет предоставлено вместе с архивом примеров после завершения написания книги.





## Глава 3. Переменные и примеси

Во всех языках программирования существуют переменные и функции, которые позволяют делать код чистым и переиспользуемым. Как мы все знаем, CSS, по сути своей, не является языком программирования в прямом смысле этого выражения. Но, с помощью препроцессоров, можно попытаться исправить это положение.

Из этой главы вы узнаете, как использовать переменные и примеси, правильно давать им имена, а также некоторые особенности и тонкости их применения в реальных проектах.

Может, звучит глупо, но именно глупости запоминаются лучше всего.

Вверх (Up)

# Использование переменных

В любом полноценном языке программирования есть **переменные** (от англ. variables), которые представляют собой именованную область памяти и позволяют записывать в себя какие-либо данные. Переменные дают возможность обращаться к себе по имени, а также изменять присвоенные им значения.

Переменные, имеющие заранее заданный тип данных, с которым они могут работать, называют *типизированными*. Такие переменные могут работать только с данными заданного типа и, если попытаться присвоить значение другого типа, то нас будет ожидать ошибка компиляции. Соответственно, если переменным можно присваивать значения любого типа, то их называют *динамическими*.

Процесс объявления (определения) переменных и присваивания им значения называют *инициализацией*.

Обычно, переменные *переопределяемые*, то есть при инициализации в них можно записать значение и, по ходу работы программы, изменять его столько раз, сколько нужно разработчику. К сожалению, существуют переменные с неизменяемыми значениями — *непереопределяемые*.

В Less переменные динамические, переопределяемые и требующие инициализации.

Любая переменная в Less инициализируется как директива, то есть с использованием символа собачки (@) в начале.

```
@block-background: #f5f5f5;  
@block-color: #333;  
@block-border-color: #ddd;
```

Для того, чтобы использовать значение переменной при написании кода, необходимо её вызвать. Делается это так:

```
.block {  
  background-color: @block-background;  
  color: @block-color;  
  border: 1px solid @block-border-color;  
}
```

## Хранение данных

В переменных можно хранить любые данные, которые подходят под шаблон `@name: value;`. Для примера я подобрал несколько значений, которые можно присвоить переменным:

```
@var-color: #ffff00;
@var-string-0: header;
@var-string-1: 0 auto;
@var-number: 4768;
@var-value: 14px;
@var-rule: {
  color: red;
};
@var-url-0: "../images/nichosi-meme.png";
@var-url-1: url("../images/nichosi-meme.png");
```

## Операции с переменными (операторы)

Оператором называется нечто, принимающее одно или более значений (или выражений, если говорить на жаргоне программирования), и вычисляющее новое значение (таким образом, вся конструкция может рассматриваться как выражение).

В Less присутствует два режима интерпретации операций с выражениями: обычный и строгий.

### Арифметические операции (обычный режим)

Допускается производить операции с выражениями, значения которых соответствуют строкам.

Пример	Название	Результат	Числа	Строки
<code>-@a</code>	Отрицание	Смена знака <code>@a</code>	+	ошибка
<code>@a + @b</code>	Сложение	Сумма <code>@a</code> и <code>@b</code>	+	игнорируется*
<code>@a - @b</code>	Вычитание	Разность <code>@a</code> и <code>@b</code>	+	игнорируется*
<code>@a * @b</code>	Умножение	Произведение <code>@a</code> и <code>@b</code>	+	игнорируется*
<code>@a / @b</code>	Деление	Частное от деления <code>@a</code> на <code>@b</code>	+	игнорируется*

«\*» — игнорируется при условии, что производится операция с явно указанной строкой в выражении не через переменную, а напрямую. Иначе ошибка.

### Арифметические операции (строгий режим)

Для включения строгого режима арифметических операций нужно взять выражения в круглые скобки (выражение). В таком режиме запрещается производить операции со строками.

Пример	Название	Результат	Числа	Строки
(-@a)	Отрицание	Смена знака @a	+	ошибка
(@a + @b)	Сложение	Сумма @a и @b	+	ошибка
(@a - @b)	Вычитание	Разность @a и @b	+	ошибка
(@a * @b)	Умножение	Произведение @a и @b	+	ошибка
(@a / @b)	Деление	Частное от деления @a на @b	+	ошибка

Перечисленные в таблицах операции можно производить с числами, строками, цветами и числами с указанной размерностью. Ниже представлена таблица, демонстрирующая операцию и её результат с различными входными данными.

@a	@b	@a + @b	@a - @b	@a * @b	@a / @b
1	1	2	0	1	1
1px	2	3px	-1px	2px	0.5px
5%	4	9%	1%	20%	1.25%
2%	3px	5%	-1%	6%	0.6666..66%
0.33	11%	11.33%	-10.67%	3.6300...03%	0.0300...02%
#6699cc	25%	#6b9ed1	#6194c7	#ffffff	#141f29
#666	#333	#999999	#333	#ffffff	#020202

## Экранирование

**Экранирование** — это замена управляющих символов на соответствующие текстовые подстановки. Делается это для того, чтобы использовать некоторые строковые конструкции в рамках языка, которые компилятор может принять за конструкции самого языка.

В Less такой опасной конструкцией могут стать фильтры, которые для IE часто принимают вид:

```
.element {
  filter: ms:alwaysHasItsOwnSyntax.For.Stuff();
}
```

Если попытаться скомпилировать этот блок кода, компилятор будет нарываться на ошибку и любезно сообщать её нам:

```
$ lessc _styles.less > styles.css

ParseError: Unrecognised input in _styles.less on li ne 24, column 13:

23 .element {
24   filter: ms:alwaysHasItsOwnSyntax.For.Stuff();
25 }
```

Разумеется, ничего хорошего из этого не выйдет, но решение этой проблемы, как гласит название раздела — *экранирование*.

Экранирование в Less может осуществляться двумя способами. Первый путь подразумевает под собой использование функции `e()`, а второй — синтаксической конструкции `~""` или `~''`. Оба способа ожидают на входе строку, окружённую кавычками, которые при компиляции будут удалены.

```
.element {
  filter: e("ms:alwaysHasItsOwnSyntax.For.Stuff()");
  filter: ~"ms:alwaysHasItsOwnSyntax.For.Stuff()";
  filter: ~'ms:alwaysHasItsOwnSyntax.For.Stuff()';
}
```

Экранирование с помощью функции `e()` считается устаревшим. Создатели Less предлагают повсеместно внедрять синтаксическую конструкцию `~""` или её аналог с одинарными кавычками.

## Области видимости

Практически во всех языках программирования есть базовые **области видимости** (от англ. scope), представляющие собой область программы, в которой доступно значение объявленной переменной. Если переменная объявлена внутри функции, то её область видимости ограничивается этой функцией, а сама переменная называется *локальной*. Если же переменная объявлена вне функции, то она — *глобальная*.

Less очень похож на язык программирования и так же имеет две области видимости, но с различием в том, что функциями считаются селекторы и примеси. Если переменная объявлена внутри селектора или примеси, то она *локальная*, если нет, то *глобальная*.

Причём Less никак не противится глобальных переменных, наоборот, будет лучше, если все объявленные переменные будут глобальными. Я имею в виду то, что в JavaScript, например, создание лишних глобальных переменных считается роскошью и наказывается. В Less же картина резко противоположная и на то есть весомые причины.

Обычно, в Less переменные хранятся в большом файле с названием `_variables.less`, и вызвано это тем, что так действительно удобнее организовывать хранение, изменение и поддержку актуальности данных. При необходимости разработчик может использовать глобальные переменные где угодно, то есть в любом файле или селекторе. При этом не требуется искать переменные по всем файлам проекта. Разумеется, что речь идёт про редакторы, у которых нет автодополнения переменных.

Свою область видимости в Less создают селекторы, ссылки на родителя селектора (`&`) и примеси.

## Пример 3.1.1

Рассмотрим пример, демонстрирующий работу переменных и области видимости в Less.

Создадим файл `_styles.less` и добавим туда несколько глобальных переменных и селекторов.

```
// Variables
@body-background: #f5f5f5;
@body-font-size: 14px;

// Code
body {
  background-color: @body-background;
  font-size: @body-font-size;
}

.block {
  @block-color: #333;
  @block-font-size: 28px;

  font-size: @block-font-size;
  color: @block-color;
}

.element {
  font-size: @block-font-size;
}
```

При компиляции возникнет ошибка из-за того, что переменная `@block-font-size` локальная и не может быть использована вне селектора `.block`. При этом глобальные переменные никаких ошибок породить не могут, так как они доступны во всем коде.

```
$ lessc _styles.less > styles.css

NameError: variable @block-font-size is undefined in _styles.less on line 20, column 14:

19 .element {
20   font-size: @block-font-size;
21 }
```

## Ленивая загрузка

Переменные в Less настолько ленивые, что их можно объявлять до их непосредственного применения (вызова) и после него. В теории это значит, что переменная инициализируется в точке её вызова, а это значит, что вы можете использовать переменные до того, как вы их объявите.

```
.lazy {
  width: @lazy-width;
}

@lazy-width: @lazy-height;
@lazy-height: 200px;
```

После компиляции получится следующий код:

```
.lazy {
  width: 200px;
}
```

Но при таком использовании нужно помнить, что локальные переменные превосходят по важности глобальные. Это правило описано ранее в части «Области видимости». То есть, если использовать две одинаковые по имени переменные, но одна из них будет локальной, а другая глобальной, то все будет зависеть от их места определения и использования.

В этом случае переменная `@lazy-height` будет локальной для объявления селектора `.lazy` и именно её значение будет использоваться при подстановке в `@lazy-width`.



```
.lazy {  
  width: @lazy-width;  
  @lazy-height: 100px;  
}  
  
@lazy-width: @lazy-height;  
@lazy-height: 200px;
```

После компиляции свойству `width` будет присвоено значение `100px` :

```
.lazy {  
  width: 100px;  
}
```

Кроме того, в Less, как и в CSS, роль играет последовательность объявления переменных. Имеется в виду, что будет использоваться та переменная, которая объявлена ниже всех.

```
@element-height: 100px; // 1  
  
.element {  
  @element-height: 200px; // 2  
  height: @element-height;  
  
  &-footer {  
    height: @element-height;  
    @element-height: 300px; // 3  
    height: @element-height;  
    @element-height: 400px; // 4  
  }  
  
  @element-height: 500px; // 5  
}
```

В этом запутанном блоке кода происходят немислимые вещи, отображающие всю суть ленивой загрузки. Рассмотрим происходящее более подробно:

- Объявление (1) сразу же можно забыть, так как оно глобальное, а внутри селектора `.element` находятся локальные переменные.
- Локальные объявления (2, 3) тоже можно не учитывать, так как ниже происходит инициализация других переменных.
- Локальное объявление (4) будет использовано внутри объявления `.element-footer`, которое формируется с использованием родительского селектора.
- Локальное объявление (5) будет использовано внутри селектора `.element`.

```
.element {  
  height: 500px;  
}  
.element-footer {  
  height: 400px;  
}
```

Не стоит запоминать или заучивать все происходящее в этой части главы, просто будьте уверены, если переменная объявлена ниже всех, то применяться будет именно она. Разумеется, что в рамках её локальности или глобальности.

## Переменные по умолчанию

На основании «ленивой загрузки» в Less предоставляется возможность организации переменных по умолчанию. Возможно, это не такой удобный и явный способ, который предлагает Sass, но, тем не менее, такое практикуется, нужно знать и использовать во благо своих целей.

Ниже приведён код, в котором переопределяется переменная `@color-green`. Получается, что в файле библиотеки вместо глобальной переменной, объявленной внутри неё, будет использована переданная переменная `@color-green` со значением `#003300`.

```
// Переменные в файле _library.less  
@color-green: green;  
@color-red: red;  
  
// Импорт библиотеки и переопределенная переменная  
@import "_library";  
@color-green: #003300;
```

Такое поведение может пригодиться при использовании библиотек или фреймворков, в которых разработчику необходимо переопределить переменные. Но вы должны понимать, что на самом деле значения переменных не изменяются, а происходит та самая «ленивая загрузка».

Переопределение переменных делает настройку структур, таких как Bootstrap 3, более гибкой и простой. Но об этом я расскажу вам в соответствующей главе.

# Интерполяция переменных

Если вы раньше работали с PHP или Ruby, то должны были хотя бы раз использовать интерполяцию переменных. **Интерполяция переменных** — это получение значения одной переменной в зависимости от другой или других переменных.

В Less есть конструкции типа `@{}`, которые технически похожи на *интерполяцию переменных*. Такие конструкции могут быть использованы в селекторах, переменных и строках. Далее будут рассмотрены все популярные и почти не применяемые варианты использования интерполяции переменных.

Я не зря вынес эту интересную тему в отдельный раздел, так как с помощью интерполяции переменных можно творить чудеса и делать ваш проект или библиотеку максимально настраиваемыми.

## Интерполяция переменных в строках

Начинать нужно с основ, так как, если есть желание овладеть технологией в полной мере, то именно основы дадут понимание того, как что-то работает или почему это что-то не работает.

Основами интерполяции переменных являются строки. Именно в них чаще всего может понадобиться подставить значение одной переменной и получить необходимое значение другой переменной, к которой относится эта строка. Это позволяет настраивать пути до изображений, шрифтов, или библиотек не используя поиск по всему проекту для их редактирования.

Интерполяция переменных в строках, как это ни странно, происходит в строках тогда, когда название переменной оборачивается в специальную конструкцию, которая выглядит следующим образом:

```
@russia-city-odintsovo: Одинцово;  
@english-city-odintsovo: Odintsovo;  
@russia-hello: "Привет, @{russia-city-odintsovo}!";  
@english-hello: "Hello, @{english-city-odintsovo}!";
```

Конструкции `@{russia-city-odintsovo}` и `@{english-city-odintsovo}` интерполируются и если вызвать эти переменные, то вместо них будет подставлено название города на соответствующем языке:

```
.hello-russia {
  content: "Привет, Одинцово!";
}
.hello-english {
  content: "Hello, Odintsovo!";
}
```

## Пример 3.2.1

В этом примере мы подробно разберём то, как происходит интерполяция переменных. Инициализируем несколько переменных и поместим их в начале файла `_styles.less` так, как это было описано в начале главы:

```
// Variables
@icon-font-name: fontawesome-webfont;
@icon-font-path: "../fonts";
```

Чтобы воспользоваться шрифтом FontAwesome, в проекте необходимо использовать директиву `@font-face`. Ради экономии места я не стал вставлять сюда весь код примера, он, как и любой код в примерах, будет доступен в архиве.

```
@font-face {
  font-family: 'FontAwesome';
  src: url('@{icon-font-path}/@{icon-font-name}.eot?v=4.3.0');
  src: url('@{icon-font-path}/@{icon-font-name}.eot?#iefix&v=4.3.0') format('embedded-opentype'),
  ...
}
```

На выходе компилятора получим следующий код:

```
@font-face {
  font-family: 'FontAwesome';
  src: url('../fonts/fontawesome-webfont.eot?v=4.3.0');
  src: url('../fonts/fontawesome-webfont.eot?#iefix&v=4.3.0') format('embedded-opentype'),
  ...
}
```

Обратите внимание, если поменять вызов переменной `@{icon-font-path}` на `@icon-font-path` то, вместо того, чтобы вставить значение переменной, компилятор попросту принял `@icon-font-path` за часть строки. Именно в этом и заключается суть интерполяции переменной — грубо говоря, она заставляет интерпретатор отличать строки от переменных и при необходимости подставлять их значение в строку.

```
@font-face {
  font-family: 'FontAwesome';
  src: url('@icon-font-path/fontawesome-webfont.eot?v=4.3.0');
  src: url('@icon-font-path/fontawesome-webfont.eot?#iefix&v=4.3.0') format('embedded-opentype'),
  ...
}
```

## Интерполяция переменных в селекторах и свойствах

Иногда требуется делать код гибким вплоть до того, чтобы селекторы изменяли своё имя в зависимости от значения переменных. Наиболее вероятно, что такое поведение пригодится в библиотеках или фреймворках, в которых разработчики хотят разрешить изменять префикс классов их детища.

Интерполяция переменных в селекторах по конструкции полностью повторяет таковую в строках:

```
@lang-prefix: site;
@lang-russia: rus;
@lang-english: eng;

.#{@lang-prefix}-#{@lang-russia} {
  content: "@{lang-russia}";
}

.#{@lang-prefix}-#{@lang-english} {
  content: "@{lang-english}";
}
```

Компилятор сам позаботится о формировании имени класса из двух переменных, инициализированных ранее:

```
.site-rus {
  content: "rus";
}
.site-eng {
  content: "eng";
}
```

Никто не запрещает использовать интерполяцию и в рамках свойств. Конструкция вида:

```
@property: color;

.widget {
  @property: #0ee;
  background-@property: #999;
}
```

будет компилироваться в:

```
.widget {
  color: #0ee;
  background-color: #999;
}
```

## Интерполяция переменных внутри переменных

Важно понимать, что хороший код должен быть адаптируемым. То есть имена селекторов и значения свойств должны меняться в зависимости от контекста использования, если это требуется разработчику.

С помощью примесей, которые будут рассматриваться позднее, можно писать адаптируемый и переиспользуемый код. Для этого в примесях используются значения, передаваемые как аргументы. В зависимости от передаваемых значений, можно формировать названия переменных, которые в последствии будут вызываться.

### Пример 3.2.2

Так как примеси мы ещё не проходили, а интерполяцию переменных внутри переменных показать и объяснить нужно, то представим себе ситуацию, в которой у нас есть несколько переменных:

```
// Variables
@color-prefix: color;

@color-grey: #fafafa;
@color-red: #ffebee;
```

Разработчику нужно выбрать серый и красный цвет, но префикс переменных, в которых хранится искомое значение, ему не известен — он хранится в переменной

```
@color-prefix .
```

Чтобы получить цвета, разработчику нужно построить переменную на основе уже известной ему переменной:

```
.grey {
  background-color: e("@{@{color-prefix}-grey}");
}

.red {
  background-color: e("@{@{color-prefix}-red}");
}
```

Обратите внимание на то, что здесь используется экранирование, так как конструкция внутри очень похожа на конструкцию переменной, но в то же время не являющаяся ей. Далее нужно обернуть конструкцию в `@{}` столько раз, сколько раз планируется собирать переменную из частей. Проще говоря, мы конструируем строку, а потом получаем значение переменной, имя которой совпадает с этой строкой.

Однако, ложкой дёгтя является то, что при такой интерполяции значение всегда приводится к строке. Это поведение стоит отнести к частному случаю, который также имеет своё решение, пусть и не самое элегантное.

## Частый случай интерполяции переменных внутри переменных

В некоторых случаях появляется необходимость получить значение переменной, имя которой задано в другой переменной. Примером может служить примесь, которая выставляет отступы блока в зависимости от переданной в неё строки. Рассмотрим эту проблему более детально.

Допустим, разработчик опытным путем или на глаз определил, что все блоки на странице делятся на два типа: с маленьким и большим отступом. Здесь под отступом стоит понимать, как внутренний, так и внешний отступ. Для стандартизации кода эти размеры будут вынесены в переменные:

```
// Small
@margin-small: 40px;
@padding-small: 20px;

// Large
@margin-large: 80px;
@padding-large: 40px;
```

В будущем разработчик напишет примесь, но, так как сейчас я о примесях ещё ничего не говорил, вместо неё будем использовать переменную `@size`. В этой переменной будет содержаться имя размера блока, то есть `small` или `large`. Попробуем получить отступы для произвольного блока, используя описанный выше метод интерполяции переменных внутри переменных:

```
// Small
@margin-small: 40px;
@padding-small: 20px;

@size: "small";

.block {
  margin: ~"@{margin-@{size}}";
}
```

Этот код будет скомпилирован в CSS без ошибок и свойству `margin` будет присвоено значение `40px`. Однако, в этой синтаксической конструкции скрывается немного чёрной магии. Понять проблему поможет простая операция сложения, которую необходимо провести с интерполяцией. Давайте попробуем сложить внешний и внутренний отступ.

```
// Small
@margin-small: 40px;
@padding-small: 20px;

@size: "small";

.block {
  margin: ~"@{margin-@{size}}" + ~"@{padding-@{size}}";
}
```

Вот это поворот! Ошибка!

```
ParseError: Unrecognised input in _styles.less on line 8, column 32:
7 .block {
8   margin: ~"@{margin-@{size}}" + ~"@{padding-@{size}}";
9 }
```

Не буду томить читателя своими догадками и анализом происходящего — Less не умеет складывать строки. Функция *экранирования* возвращает **строку**. Да, вот так вот просто — сложить строку со строкой или строку с числом в Less нельзя. По крайней мере, в версии `2.7.1` это сделать нельзя. Впрочем, эта возможность присутствует в Sass и Stylus.



Так как же решить возникшую проблему? — использовать интерполяцию переменных с помощью конструкции `@@`. К сожалению, для этого придётся ввести дополнительную переменную для каждого слагаемого.

```
// Small
@margin-small: 40px;
@padding-small: 20px;

@size: "small";

.block {
  @margin: "margin-@{size}";
  @padding: "padding-@{size}";
  margin: @@margin + @@padding;
}
```

После компиляции свойству `margin` будет присвоено значение `60px`. Это происходит вследствие того, что в переменной `@margin` после интерполяции находится имя переменной `margin-small`. Точно такая же ситуация со значением в переменной `@padding`. При вызове переменной с помощью конструкции `@@` на её место подставляется значение переменной, имя которой указано в строке.

# Наименование переменных

В этой части главы я поделюсь с вами тактикой наименования и использования переменных. Вся информация, представленная здесь, основывается на многолетнем опыте работы с Less и большом количестве руководств по написанию таблиц стилей. У вас есть основания не доверять мне, но я бы советовал все же прочитать и обдумать описанные методики.

Для того, чтобы погрузиться в «программирование» на Less, как и в любом другом языке, нужно правильно именовать переменные. Я не шучу, это очень важная и животрепещущая тема для всех новичков и даже тех, кто уже может себя по праву считать заслуженным программистом.

## Имя переменной

Когда переменная имеет имя `a` или `razmer-wrifty`, а правила наименования меняются каждые десять строк, то уже сразу можно говорить о разработчике плохо. Просто примите как должное, что название переменной должно быть на английском языке, вне зависимости от того, на каком языке написаны комментарии к коду. Будь вы хоть трижды патриот своей страны или вы не знаете английский.

**Хорошим примером могут послужить следующие переменные:**

```
@body-background: #f5f5f5;  
@body-font-size: 14px;  
@header-background: #fff;  
@btn-border-radius: 4px;  
@navbar-item-line-height: 20px;
```

## Разделение слов в имени

Используйте дефис, нижнее подчеркивание (`under_score`) или верблюжью нотацию (`camelCase`) для разделения слов в имени переменных.

**Хорошим примером могут послужить следующие переменные:**

```
// Дефис
@grid-breakpoint-xsmall: 540px;
@navbar-item-color: #777;
@pagination-border-color: #ddd;

// Нижнее подчёркивание (under_score)
@hamburger_border_radius: @border_radius;
@line_height_computed: 24px;
@font_awesome_path: "../vendor/font-awesome/font";

// Верблюжья нотация (camelCase)
@labelFontSize: 0.8em;
@commentBackground: #FFFFFF;
@authorHoverColor: @primaryColorHover;
```

## Содержательное имя

Старайтесь давать переменным такие имена, которые сочетают в себе простоту, содержательность и логичность.

**Хорошим примером могут послужить следующие переменные:**

```
@brand-color: #ffff66;
@btn-default-background: #fcfcfc;
@btn-default-hover-background: #ccc;
@article-item-title-hover-color: @brand-color;
```

## Последовательность имени

Самым трудным по праву считается соблюдение баланса между простотой, содержательностью и логичностью имени переменной. В идеале нужно, чтобы имя было коротким и рассказывало все о переменной и контексте её применения.

Рассмотрим последовательность имени переменных на основе разметки:

```
<article class="article-card">
  <header class="head">
    <h1 class="title">
      <a href="#ignoreClick">Заголовок</a>
    </h1>
  </header>
  <div class="content">
    <p>Текст карточки</p>
  </div>
  <footer class="footer">
    <ul class="tag-widget">
      <li><a href="#tag">Проекты</a></li>
      <li><a href="#tag">Less</a></li>
    </ul>
  </footer>
</article>
```

**Хорошим примером могут послужить следующие переменные:**

```
@article-card-background: #fff;
@article-card-border-radius: 4px;

// Цвет и размер шрифта ссылки в заголовке
@article-card-title-color: #777;
@article-card-title-hover-color: #333;
@article-card-title-font-size: 2.4rem;

// Размер шрифта блока контента
@article-card-content: 1.6rem;

// Виджет тегов
// Переиспользуемое представление тегов
// В различных частях макета
@tag-item-color: #777;
@tag-item-hover-color: #333;
```

## Единый стиль

Используйте единый стиль наименования переменных. Например, если вы один раз использовали дефис для разделения слов в имени переменных, то используйте его во всём проекте.

# Использование примесей

Когда-то давно, наверное, только ленивый разработчик не мечтал о переиспользуемом CSS-коде. Вы только представьте себе то, что какой-нибудь блок кода может быть вызван повторно, и все его свойства станут доступны и второму селектору. Тогда это были желания, сейчас же это реальность.

В отличие от других CSS-препроцессоров, в Less любой объявленный селектор может использоваться как примесь. **Примесь** (от англ. mix-in) — набор свойств и селекторов, расширяющий поведение другой сущности (селектора).

Рассмотрим типичное объявление, которое уже миллионы раз встречалось в вашем CSS-коде:

```
.bordered {  
  border-top: dotted 1px #333;  
  border-bottom: solid 2px #333;  
}
```

Казалось бы, ничего необычного, но ровно до тех пор, пока не сделать следующее:

```
.article {  
  .bordered;  
  color: #443d3d;  
}
```

После компиляции селектор `.bordered` безвозмездно отдаст все свои свойства и вложенные правила селектору `.article`. При этом в конечном CSS-файле будут объявлены оба этих селектора.

```
.bordered {  
  border-top: dotted 1px #333;  
  border-bottom: solid 2px #333;  
}  
.article {  
  border-top: dotted 1px #333;  
  border-bottom: solid 2px #333;  
  color: #443d3d;  
}
```

На самом деле, такая запись практически не используется на практике и не называется примесью. Настоящая примесь — она какая-то другая, волшебная. Но различие в них настолько мало, что и заметить бывает сложно:

```
.bordered() {  
  border-top: dotted 1px #333;  
  border-bottom: solid 2px #333;  
}  
  
.article {  
  .bordered;  
  color: #443d3d;  
}
```

В этой записи я добавил скобки после селектора `.bordered`, и уже сейчас он может гордо носить название примеси. В этом случае при компиляции не будет создан класс `.bordered`, так как у него указаны скобки после имени. Такая конструкция говорит компилятору, что она чистейшая примесь, которая не хочет быть скомпилирована без явных на то причин.

```
.article {  
  border-top: dotted 1px #333;  
  border-bottom: solid 2px #333;  
  color: #443d3d;  
}
```

Неважно, будут ли указаны скобки при вызове примеси (`.bordered();`) или нет (`.bordered;`) — результат будет одинаковым. Однако, я бы советовал их указывать из-за того, что примеси могут иметь параметры.

## Примеси с параметрами

Для того, чтобы примеси были переиспользуемыми в различных контекстах, у них могут быть указаны параметры, в зависимости от которых может меняться цвет, фон и другие значения. Параметры указываются в скобках после имени примеси и представляют собой обычные локальные переменные.

```
.bordered(@_color) {  
  border-top: dotted 1px @_color;  
  border-bottom: solid 2px @_color;  
}  
  
.article {  
  .bordered(#ccc);  
  color: #443d3d;  
}
```

Такая конструкция напоминает функции в JavaScript или других языках программирования, но на деле ей не является, так как работает скорее как макрос.

```
.article {  
  border-top: dotted 1px #cccccc;  
  border-bottom: solid 2px #cccccc;  
  color: #443d3d;  
}
```

Параметров может быть неограниченное число, главное — чтобы они помещались на вашем экране и были читаемы.

Разделять параметры при объявлении примеси и её вызове можно запятыми (`@a, @b`), либо точкой с запятой (`@a; @b`).

## Пример 3.4.1

В этом примере отображена возможность работы с вложенными селекторами внутри примеси.

```
.clearfix() {  
  &:before,  
  &:after {  
    display: table;  
    content: "";  
  }  
  
  &:after {  
    clear: both;  
  }  
}  
  
.navbar {  
  .clearfix();  
}
```

Ничего необычного, получен вполне ожидаемый результат:

```
.navbar:before,  
.navbar:after {  
  display: table;  
  content: "";  
}  
.navbar:after {  
  clear: both;  
}
```

Замечу ещё раз, что из-за указанных скобок у селектора `.clearfix`, соответствующий класс не будет объявлен после компиляции.

## Значения параметров по умолчанию

Важной отличительной чертой примесей является возможность указывать значения по умолчанию для переменных. То есть в случае, если примесь вызвана, а значения для параметров не были переданы или переданы частично, то ошибки компилятор не выдаст, а возьмёт значение, указанное по умолчанию.

```
.bordered(@_color: #ccc) {  
  border-top: dotted 1px @_color;  
  border-bottom: solid 2px @_color;  
}  
  
.article {  
  .bordered();  
  color: #443d3d;  
}
```

Эта запись не имела бы смысла, но так как указано значение по умолчанию, то при компиляции будет подставлено именно оно:

```
.article {  
  border-top: dotted 1px #cccccc;  
  border-bottom: solid 2px #cccccc;  
  color: #443d3d;  
}
```

Все было бы хорошо, если бы для всех параметров указывались значения по умолчанию и их не нужно было изменять в зависимости от контекста применения примеси. На практике редко встречаются примеси, у которых используются указанные по умолчанию значения.



Рассмотрим пару примеров, которые показывают различные ситуации, часто встречающиеся на практике в реальных проектах.

## Пример 3.4.2

Иногда необходимо изменить не все параметры, а лишь некоторые из них. Делается это следующим образом:

```
.transition(@function: ease, @duration: .3s, @property: all) {  
  transition-timing-function: @function;  
  transition-duration: @duration;  
  transition-property: @property;  
}  
  
.link {  
  .transition(@duration: 1s);  
}
```

Вместо того, чтобы писать все три значения для переменных, я указал конкретную переменную и обновил её значение. Компилятор возьмёт значения по умолчанию для переменных `@function` и `@property`, а значение переменной `@duration` будет взято из объявления вызова примеси.

```
.link {  
  transition-timing-function: ease;  
  transition-duration: 1s;  
  transition-property: all;  
}
```

## Пример 3.4.3

Если нужно изменить переменные, указанные слева, то писать переменные не нужно. Просто передайте при вызове необходимое количество значений.

```
.transition(@function: ease, @duration: .3s, @property: all) {
  transition-timing-function: @function;
  transition-duration: @duration;
  transition-property: @property;
}

.block {
  .transition(linear);
}

.link {
  .transition(linear, .5s);
}
```

После компиляции получится следующая картина:

```
.block {
  transition-timing-function: linear;
  transition-duration: 0.3s;
  transition-property: all;
}

.link {
  transition-timing-function: linear;
  transition-duration: 0.5s;
  transition-property: all;
}
```

## Пример 3.4.4

Внутри примеси можно творить все что угодно с переданными переменными и теми, что были указаны глобально. Этот пример я не могу продемонстрировать на реальном коде, так как стараюсь избегать взаимодействия глобальных и локальных переменных внутри примеси.

```
@global: 3;

.calc(@a: 1, @b: 2, @c: @global) {
  @d: @a + @b + @c;
  content: @d;
}

.test {
  .calc();
}
```

Судя по правилам математики должно получиться 6.

```
.test {  
  content: 6;  
}
```

## Мысли и советы

Используйте значения параметров по умолчанию, если это может сократить вызов примеси. Однако, при этом не должен потеряться смысл вызова, то есть разработчику должно быть понятно, что и как делает эта примесь.

# Специальные параметры и сопоставление шаблонов

Примеси являются очень мощным инструментом в рамках Less. Их возможности практически ограничены лишь фантазией разработчика, а спектр применения довольно широк.

## Специальные параметры

Кроме обычных локальных переменных для примеси, можно использовать специальные параметры, которые имеют несколько другое предназначение.

### Переменная `@arguments`

Переменная `@arguments` представляет собой синоним для всех переданных в примесь значений при её вызове. Такое поведение может быть полезно при использовании примеси с большим количеством переменных, предназначенных для одного свойства.

```
.box-shadow(@x: 0, @y: 0, @blur: 1px, @color: #333) {
  -webkit-box-shadow: @arguments;
  -moz-box-shadow: @arguments;
  box-shadow: @arguments;
}

.big-block {
  .box-shadow(2px, 5px);
}
```

Так как переменная `@arguments` содержит в себе значения всех переданных переменных, то результат будет следующим:

```
.big-block {
  -webkit-box-shadow: 2px 5px 1px #333;
  -moz-box-shadow: 2px 5px 1px #333;
  box-shadow: 2px 5px 1px #333;
}
```

К счастью, такие примеси уходят в прошлое благодаря плагину `Autoprefixer`, который является представителем эры постпроцессоров и, наверное, наиболее часто используемым из них.

## Переменная `@rest`

Переменная, позволяющая указывать переменное количество передаваемых параметров. Звучит сложно, но на деле все проще:

```
.mixin(@width: 152px, @height: 20px, @rest...) {
  width: @width;
  height: @height;
  border: @rest;
}

.class {
  .mixin(304px, 40px, 2px, solid, #ddd);
}
```

Обратите внимание на троеточие ( `...` ) после переменной `@rest` . Если бы его не было, то переменная `@rest` являлась бы обычной переменной со всеми вытекающими отсюда проблемами в виде ошибки компилятора.

Первые два значения будут присвоены переменным `@width` и `@height` соответственно, а остальные значения будут доступны переменной `@rest` .

```
.class {
  width: 304px;
  height: 40px;
  border: 2px solid #ddddd;
}
```

Переменные `@rest` и `@arguments` можно использовать вместе:

```
.mixin(@width: 152px, @height: 20px, @rest...) {
  content: @arguments;
}

.class {
  .mixin(304px, 40px, 2px, solid, #ddd);
}
```

В этом случае свойство `content` получит все переданные переменные:

```
.class {  
  content: 304px 40px 2px solid #ddddd;  
}
```

Как и в случае с `@arguments`, используется крайне редко, так как такое поведение примесей избыточно и практически не имеет места применения.

## Сопоставление шаблонов

Иногда требуется изменять поведение примеси в зависимости от передаваемых в неё параметров. Поведение примеси может зависеть от количества переданных переменных, их значений и некоторых других факторов, о которых будет говориться подробнее в следующей главе.

Я не стал оформлять это как пример, так как здесь идёт объяснение основ сопоставления шаблонов, но вы можете найти описанный здесь код как пример 3.4.5 в архиве, прилагаемом к этой книге.

Допустим, что имеется примесь, которая генерирует цвет фона, шрифта и рамки:

```
.mixin(@color, @bg, @border-color) {  
  color: @color;  
  background-color: @bg;  
  border: 1px solid @border-color;  
}
```

Требуется изменять цвет и фон блока, например, в зависимости от количества переданных переменных. Если переменная одна, то менять только цвет шрифта, если их две, то менять цвет шрифта и фон, иначе все параметры. Это можно сделать следующим образом:

```
.mixin(@color) {
  color: @color;
}

.mixin(@color, @bg) {
  color: @color;
  background-color: @bg;
}

.mixin(@color, @bg, @border-color) {
  color: @color;
  background-color: @bg;
  border: 1px solid @border-color;
}
```

Рассмотрим этот случай подробнее, так как такие конструкции будут часто встречаться в крупных фреймворках и хитрых проектах.

Здесь не играет роли в каком порядке будут объявлены примеси, так как компилятор сам узнает количество переданных значений при вызове примеси и сопоставит их с той примесью, к которой оно подходит. Однако, такая схема работает до тех пор, пока не появится ещё одна примесь с одинаковым количеством переменных, но об этом немного позднее.

Итак, попробуем передать одну переменную:

```
.test-1 {
  .mixin(#333);
}
```

Сперва может показаться, что такой вызов породит ошибку из-за того, что действует правило «сверху вниз» и использоваться будет последняя примесь. Но это не так.

```
.test-1 {
  color: #333333;
}
```

Хорошо, здесь сработала первая примесь, которой для работы требуется один параметр.

Попробуем передать все три параметра:

```
.test-3 {
  .mixin(#333, #fff, #ddd);
}
```

Получим код, который порождает третья примесь:

```
.test-3 {  
  color: #333333;  
  background-color: #ffffff;  
  border: 1px solid #dddddd;  
}
```

Все это хорошо работает и вроде как даже понятно, но в игру вступает любопытство:

- Что будет, если у всех трёх примесей указаны параметры по умолчанию и примесь вызывается без значений?
- Что будет, если у одной из примесей указаны параметры по умолчанию и примесь вызывается без значений?
- Что будет, если у одной из примесей указаны параметры по умолчанию, у другой частично и примесь вызывается с одним значением?
- Что будет, если объявлены примеси с одинаковым числом параметров?

Ответы на эти вопросы я предлагаю узнать и понять с помощью примеров описанных ниже:

## Пример 3.5.1

Что будет, если у всех трёх примесей указаны параметры по умолчанию и примесь вызывается без значений?

Я сокращаю количество примесей до двух, чтобы не показывать один и тот же код от примера к примеру.

```
.mixin(@color: #777) {  
  color: @color;  
}  
  
.mixin(@color: #000, @bg: #fff) {  
  color: @color;  
  background-color: @bg;  
}
```

Итак, у всех примесей указаны значения параметров по умолчанию. Можно предположить, что при вызове будут использованы обе примеси, так как обе они подходят под шаблон вызова.



```
.class {  
  color: #777777;  
  color: #000000;  
  background-color: #ffffff;  
}
```

## Пример 3.5.2

Что будет, если у одной из примесей указаны параметры по умолчанию и примесь вызывается без значений?

```
.mixin(@color: #777) {  
  color: @color;  
}  
  
.mixin(@color, @bg) {  
  color: @color;  
  background-color: @bg;  
}
```

При компиляции будет использоваться та примесь, что полностью соответствует шаблону вызова. В нашем случае при вызове не указаны значения переменных и компилятор будет использовать ту примесь, у которой указаны значения по умолчанию.

```
.class {  
  color: #777777;  
}
```

## Пример 3.5.3

Что будет, если у одной из примесей указаны параметры по умолчанию, у другой частично и примесь вызывается с одним значением?

В этом случае все будет зависеть от того, какой параметр имеет значение по умолчанию у второй примеси.

Допустим, что первый:

```

.mixin(@color: #777) {
    color: @color;
}

.mixin(@color: #333, @bg) {
    color: @color;
    background-color: @bg;
}

.class {
    .mixin(#000);
}
    
```

Вместо того, чтобы использовать первую примесь, которая полностью удовлетворяет шаблону вызова, компилятор использует вторую и в ответ отдаёт нам ошибку, говорящую о том, что примеси не хватает передаваемых значений.

```

SyntaxError: wrong number of arguments for .mixin (1 for 2) in _styles.less on line 11
, column 3:
10 .class {
11   .mixin(#000);
12 }
    
```

Если же значение по умолчанию указано у второго параметра, то компилятор спокойно выдаст:

```

.class {
    color: #000000;
    background-color: #ffffff;
}
    
```

## Пример 3.5.4

Что будет, если объявлены примеси с одинаковым числом параметров?

Довольно интересный случай, который может сыграть с вами злую шутку или же наоборот помочь вам писать запутанный код.

Вернёмся к полному коду примера 3.4.5 и добавим зловредную примесь. Для наглядности поменяем параметры местами:

```
.mixin(@color) {
  color: @color;
}

.mixin(@color, @bg) {
  color: @color;
  background-color: @bg;
}

.mixin(@color, @bg, @border-color) {
  color: @color;
  background-color: @bg;
  border: 1px solid @border-color;
}

.mixin(@bg, @color) {
  color: @color;
  background-color: @bg;
}
```

Компилятор будет недоволен происходящим, но работу свою он знает и выдаст следующее:

```
.class {
  color: #333333;
  background-color: #f5f5f5;
  color: #f5f5f5;
  background-color: #333333;
}
```

Получается, что в случае, когда объявлены примеси с одинаковыми именами и количеством параметров на выходе будут иметься свойства сразу двух примесей.

## Мысли и советы

Старайтесь не использовать переменные `@arguments` и `@rest` там, где это не нужно. Такой подход может запутать новых разработчиков вашей команды или даже вас самих.

Сопоставление шаблонов — это, конечно, круто, но не забывайте, что ваш код должен быть элегантным и максимально простым.

# Дополнительные возможности примесей

Для наглядности я попытался разделить базовые возможности примесей от тех, что можно считать дополнительными. Конечно, все они нужны и важны, но дополнительные возможности все таки используются не часто. Можно даже сказать, что такие возможности практически не используются в виду их специфичности и области применения.

## Примеси как функции

Идеология функции заключается в том, что она должна возвращать значение, а её вызов можно использовать как выражение. В Less организована работа лишь первой части этого утверждения.

Для примера я приведу код, используемый для генерации иконки меню. Сейчас такую иконку называют «гамбургером» из-за своего внешнего вида.

```
.hamburger-settings(@width: 32px, @height: 3px, @gutter: 5px, @color: #000,
                    @border-radius: 0, @duration: .3s,
                    @timing-function: ease) {
  @hamburger-width: @width;
  @hamburger-height: @height;
  @hamburger-gutter: @gutter;
  @hamburger-color: @color;
  @hamburger-border-radius: @border-radius;
  @hamburger-duration: @duration;
  @hamburger-timing-function: @timing-function;
}
```

В том месте, где необходимо получить эти настройки, нужно просто вызвать эту функцию. При этом изменив необходимые значения.

```
.hamburger-settings(24px, 3px, 5px, #777, @timing-function: linear);
```

Суть этого метода в том, чтобы дать пользователю в удобном для него виде изменять настройки библиотеки.

С помощью этого свойства примесей можно организовать настоящие математические функции, разумеется, если это необходимо.

Например, площадь треугольника, если известно основание и высота:

```
.areaTriangle(@a, @h) {  
  @calcAreaTriangle: (0.5 * @a * @h);  
}  
  
.class {  
  .areaTriangle(20, 50);  
  content: @calcAreaTriangle;  
}
```

или переводить единицы измерения:

```
.pxToEm(@value, @base: 16px) {  
  @calcEm: (@value / @base) + 0em;  
}  
  
.class {  
  .pxToEm(20px);  
  content: @calcEm;  
}
```

или найти среднее между двумя числами:

```
.average(@x, @y) {  
  @calcAverage: ((@x + @y) / 2);  
}  
  
.class {  
  .average(30px, 50px);  
  padding: @calcAverage;  
}
```

## Пространство имён

Пространства имён представляют собой способ организации различных примесей. Их можно сравнить с директориями в файловой системе или с ящиками в картотеке. Проще говоря, пространства имён позволяют сортировать примеси по условным категориям и избегать конфликта их имён.

Вы можете объявить примесь внутри селектора:

```
.selector {  
  .mixin(@color: #333) {  
    color: @color;  
  }  
}
```

При этом основное свойство примеси будет сохраняться, то есть после компиляции примесь не создаст новый класс. Однако, теперь обратиться к примеси с таким объявлением напрямую не получится. Будет выдаваться ошибка.

В случае, когда указано пространство имён, необходимо указывать полный путь до примеси.

```
// не работает / ошибка (Error: .mixin is undefined)  
.class {  
  .mixin();  
}  
  
// работает (разные варианты)  
.class {  
  .selector > .mixin();  
  .selector .mixin();  
  .selector.mixin();  
}
```

Такой подход позволяет создавать примеси, которые не будут конфликтовать с именами других примесей в проекте или используемых библиотеках.

## Ключевое слово **!important**

Не так давно была выпущена версия компилятора, исправляющая ошибочную работу этого ключевого слова, поэтому перед работой проверьте, что ваш компилятор имеет версию либо ниже `2.3.0`, либо выше.

Основная суть функции — добавление ко всем свойствам примеси этого ключевого слова.

```
.mixin(@color: #333, @bg: #f5f5f5) {
  color: @color;

  .nested {
    background-color: @bg;
  }
}

.class {
  .mixin() !important;
}
```

После компиляции к каждому свойству будет добавлено ключевое слово `!important`.  
Причём не важно, будет ли это свойство коренного селектора, либо вложенного.

```
.class {
  color: #333333 !important;
}
.class .nested {
  background-color: #f5f5f5 !important;
}
```

# Работа с набором правил

Все те счастливики, что используют Node.js знают, что без обратного вызова (от англ. callback) невозможно написать приложение. В Less роль функций обратного вызова в примесях берёт на себя **набор правил** (от англ. rule set — блок/набор правил).

Основная идея набора правил заключается в возможности присвоить переменной не только какое-то значение, но и целый блок кода, состоящий из свойств, селекторов и прочих интересностей.

На практике это выглядит следующим образом:

```
@element: {
  color: #777;

  &.active {
    color: #000;
  }
};

.item {
  @element();
}
```

Такая конструкция напоминает селекторы и, в частности, примеси. Но, надеюсь, в таком ключе использовать её читатель не будет.

```
.item {
  color: #777;
}
.item.active {
  color: #000;
}
```

За таким поведением переменных стоит другая идея — передача свойств в примеси. Пригодится такая возможность, когда разработчик захочет упростить себе жизнь, написав пару примесей. Например, разработчику надоело писать длинные медиавыражения или он решил ограничить область видимости переменных.

Рассмотрим пример, иллюстрирующий преимущество использования набора правил.



## Набор правил как функция обратного вызова

В случае с медиавыражениями понятно: нужно всего лишь описать заранее само выражение и передавать туда необходимые свойства и селекторы.

Представленный здесь код доступен как пример 3.7.1.

```
.screen(@min, @max, @ruleset) {
  @media (min-width: @min) and (max-width: (@max - 1)) {
    @ruleset();
  }
}
```

Теперь при вызове такой примеси ему нужно передать два размера экрана и набор правил. При компиляции вместо переменной `@ruleset` будут подставлены те значения, что были переданы при вызове в фигурных скобках.

```
.class {
  background-color: #000;

  .screen(768px, 1200px, {
    background-color: #fff;
  });
}
```

После компиляции получим следующий CSS-код:

```
.class {
  background-color: #000;
}
@media (min-width: 768px) and (max-width: 1199px) {
  .class {
    background-color: #fff;
  }
}
```

Для наглядности посмотрите на callback в JavaScript и сравните получившуюся у нас примесь с ним.

Объявление и вызов:

```
function mySandwich(param1, param2, callback) {
  console.log('Мой сэндвич включает в себя ' + param1 + ' и ' + param2 + '.');
  callback();
}

mySandwich('котлету', 'сыр', function() {
  console.log('А также соус, огурцы и прочие вкусняшки.');
```

В консоль выведется:

```
Мой сэндвич включает в себя котлету и сыр.
А также соус, огурцы и прочие вкусняшки.
```

Если посмотреть на функцию обратного вызова в Less и JavaScript, то несложно заметить их сходство. Для тех, кто знаком с Node.js — это поможет понять сущность набора правил в Less, благодаря проведению параллели между Less и JavaScript. Тем же, кто не знаком с JavaScript, есть куда стремиться.

## Набор правил как область видимости

Представленный здесь код доступен как пример 3.7.2.

Обратите внимание на следующий код:

```
@variable: global;

@detached-ruleset: {
  variable: @variable;
};

selector {
  @detached-ruleset();
  @variable: value;
}
```

На первый взгляд свойство `variable` будет иметь значение `value`, так как здесь будет действовать правило ленивой загрузки переменных. Однако, так как мы имеем дело с набором правил, свойству будет присвоено значение `global`.

```
selector {
  variable: global;
}
```

Происходит это из-за того, что переменная `@variable` локальная для селектора `selector`, а переменная, содержащая вызов этой переменной объявлена глобально, следовательно, изначально поиск значения переменной будет происходить в глобальной области, а потом уже в локальной области.

Если же убрать глобальную переменную `@variable`:

```
@detached-ruleset: {  
  variable: @variable;  
};  
  
selector {  
  @detached-ruleset();  
  @variable: value;  
}
```

то значение будет браться от локальной переменной:

```
selector {  
  variable: value  
}
```

Другие интересные примеры применения области видимости, в случае использования набора правил, вы можете найти в официальной документации. Вызвано это тем, что они очень похожи друг на друга, а раздувать объяснение похожих примеров — не особо интересное занятие.

## Мысли и советы

Не используйте переменные с набором правил как примеси. Машины созданы для того, чтобы ездить, а самолёты, чтобы летать.

Основная задача переменных с набором правил — передача свойств в примеси. Старайтесь использовать эту возможность именно так, и никак иначе.

## Домашнее задание

В этой главе вы узнали о том, что такое переменные, какие значения в них можно хранить и как их использовать на практике. Кроме того, вы познакомились с примесями, позволяющими повторно использовать блоки кода. Самое время выполнить домашнее задание на эту тему.

### Постановка задачи

Необходимо разработать набор кнопок для сайта, используя все полученные знания из этой главы.

### Техническое задание

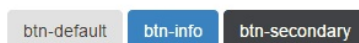
Каждая кнопка имеет общий стиль, но при этом различается размерами и цветами.

Обязательным условием выполнения будет использование переменных для хранения цветов, размеров и прочих величин, которые могут меняться на любой стадии развития проекта. А также примесей, которые будут менять размер и цвет кнопки в зависимости от передаваемых параметров.

Ниже приведены изображения кнопок, которые необходимо разработать.

#### Состояние до нажатия и после

##### Состояние по умолчанию

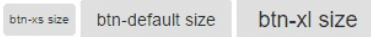


##### Нажатое состояние



#### Размеры кнопок

## Различные размеры



Для домашнего задания я заранее подготовил таблицу со всеми значениями переменных, которые могут понадобиться вам в работе.

### Цвет фона и шрифта

Свойство	По умолчанию	Второй тип	Третий тип
background	#e0e0e0	#3b83c0	#404245
color	#333	#fff	#fff
hover background	#e8e8e8	#458ac6	#1b1c1d
hover color	#333	#fff	#fff

### Размеры

Свойство	Маленький	Стандартный	Большой
font-size	10px	14px	18px
line-height	20px	20px	20px
padding	2px 4px	6px 12px	10px 16px

Не волнуйтесь, у вас все получится! Если у вас возникают какие-либо сложности, то посмотрите решение этого задания в архиве, который прилагается к этой книге.

#### Внимание

Решение этого домашнего задания будет предоставлено вместе с архивом примеров после завершения написания книги.

## Глава 4. Операции со свойствами и встроенные функции

Для удобства пользователей препроцессора Less его разработчики добавили некоторые встроенные функции.

В этой главе мы поговорим о возможности слияния свойств, которое может пригодиться при наличии одинаковых свойств в двух селекторах (примесях), встроенных функциях для работы со строками, списками (массивами), изображениями и типами данных. А также затронем тему манипуляции цветами и поговорим о наличии всевозможных математических функциях, которые зачем-то могут пригодиться в CSS.

Терпимость — это когда прощают чужие ошибки; такт — когда не замечают их.

Артур Шницлер

## Слияние свойств

При создании набора примесей может понадобиться манипулировать значениями свойств, то есть адаптировать их в зависимости от контекста использования, используя условные конструкции (подробнее смотрите в главе 5).

Для начала рассмотрим пример, отображающий проблему при вызове двух примесей, имеющих одинаковые свойства.

```
.depth-top() {  
  box-shadow: 0 2px 5px 0 rgba(0, 0, 0, 0.16);  
}  
  
.depth-bottom() {  
  box-shadow: 0 2px 10px 0 rgba(0, 0, 0, 0.12);  
}
```

Допустим, что имеется код, который генерирует одно и тоже свойство, но с разными значениями. В первом случае это будет тень сверху блока, а во втором — снизу. С таким подходом можно встретиться в модном сейчас направлении — материальный дизайн. Вызовем эти примеси для какого-нибудь произвольного блока:

```
.block {  
  .depth-top();  
  .depth-bottom();  
}
```

Как не сложно было догадаться, мы получили два одинаковых свойства, но значения у них разные. В таком подходе кроется одна большая проблема.

```
.block {  
  box-shadow: 0 2px 5px 0 rgba(0, 0, 0, 0.16);  
  box-shadow: 0 2px 10px 0 rgba(0, 0, 0, 0.12);  
}
```

И проблема эта в том, что второе свойство переопределит первое. Это приведёт к тому, что тень не будет состоять из верхней и нижней частей — браузер покажет лишь нижнюю, так как она стоит ниже в селекторе и заменяет собой верхнюю.

Благодаря Less значения свойств можно конкатенировать, и делается это двумя способами:

- Через запятую (свойства `box-shadow`, `font-family` и т.д.)
- Через пробел (свойства `transform`, `text-overflow` и т.д.)

## Слияние свойств через запятую

Для того, чтобы решить проблему, просто добавим плюс (+) перед двоеточием свойства. Это скажет компилятору, что при встрече двух одинаковых свойств в одном селекторе — его целью будет их объединение.

```
.depth-top() {  
  box-shadow+: 0 2px 5px 0 rgba(0, 0, 0, 0.16);  
}  
  
.depth-bottom() {  
  box-shadow+: 0 2px 10px 0 rgba(0, 0, 0, 0.12);  
}
```

После компиляции результат оправдывает ожидание. Свойство одно и оно включает в себя конкатенированное значение двух примесей, записанных через запятую, что и соответствует синтаксису этого свойства.

```
.block {  
  box-shadow: 0 2px 5px 0 rgba(0, 0, 0, 0.16), 0 2px 10px 0 rgba(0, 0, 0, 0.12);  
}
```

Важно заметить, что такой подход будет работать только со свойствами, у которых указан плюс перед двоеточием. То есть, если у селектора уже есть свойство `box-shadow` без явного указания на необходимость слияния значений, то конкатенация значений проводиться не будет.

## Слияние свойств через пробел

По сути своей, никакого отличия от слияния свойств через запятую здесь нет — добавляется лишь нижнее подчёркивание после плюса перед двоеточием.



```
.scale(@scale) {
  transform+_: scale(@scale);
}

.rotate(@angle) {
  transform+_: rotate(@angle);
}

.translate(@px) {
  transform+_: translateX(@px);
}

.block {
  .scale(1.75);
  .rotate(45deg);
  .translate(10px);
}
```

В результате, после компиляции получим следующее валидное для этого свойства значение:

```
.block {
  transform: scale(1.75) rotate(45deg) translateX(10px);
}
```

# Строки и списки

## Экранирование

Компилятор, сам по себе, штука умная. Но даже у умных штук есть проблемы. Например, компилятор не может сразу определить, является ли предлагаемая ему для обработки конструкция валидной. Проще говоря, если конструкция не предназначена компилятору, то может случиться порождение ошибки. Для предотвращения таких проблемных ситуаций, используется экранирование значения переменных, которое было подробнее рассмотрено в главе 3.

## Функция кодирования (Escape)

Вместо того, чтобы ещё раз говорить об экранировании, давайте рассмотрим функцию кодирования символов — `escape`.

Функция `escape`, позволяет закодировать символы, которые могут вызывать проблемы в CSS. Например, такими символами являются: пробел, `#`, `^`, `(`, `)`, `;` и т.д.

Зачем это может пригодиться в Less? — одним создателям известно. Конечно, можно предположить, что таким образом можно вставлять текст в значение свойства `content` или что-то в этом духе, но это не работает, так как требуются кавычки, которые эта функция кодирует. Пусть это останется загадкой.

Я честно пытался найти применение этой функции, но ничего не получилось. Поэтому наслаждайтесь замечательным примером из документации:

```
escape('a=1') // a%3D1
```

## Форматирование строк

Помимо встроенной функции кодирования, в Less имеется довольно редко используемая функция форматирования строк.

Эта функция позволяет формировать строку, используя управляющие последовательности (`%`) в ней. Синтаксис этой функции имеет следующий вид:

```
@string: %(строка, аргументы...);
```

Все аргументы, переданные в функцию, являются значениями для заполнителей, кроме первого аргумента. Первый аргумент всегда является строкой, содержащей сами заполнители. Заполнитель — это символ процента ( `%` ) и следующая за ним буква ( `S, s, D, d, A, a` ).

Если в строке указывается прописная буква ( `A, D, S` ), то специальные символы, переданные в строку в значениях, будут экранированы. Исключением являются символы ( `'~!` ). Строчные буквы ( `a, d, s` ) оставляют специальные символы без изменений.

Заполнители `D, d` и `A, a` могут быть заменены любым аргументом (цветом, цифрой, выражением), но будут включать в себя кавычки. Заполнители `S, s` вычисляют выражение и подставляют его без кавычек.

Например, можно использовать этот функционал, чтобы формировать значение свойства `content` :

```
@string: "В этот замечательный день %S, я хочу сказать вам: %a";
@date: "(28.06.2015)";
@message: в Less так много интересных функций;

.block {
  content: %(@string, @date, @message);
}
```

Компилятор просто подставит строки из переменных в шаблон:

```
.block {
  content: "В этот замечательный день (28.06.2015), я хочу сказать вам: в Less так мно
го интересных функций";
}
```

### Замечание

Для того, чтобы использовать в строке символ процента как обычный символ процента — необходимо просто удвоить его ( `%%` ).

## Функция замены (Replace)

Эта функция позволяет заменить указанное значение в строке на то, что будет передано в аргументах функции. Синтаксис этой функции имеет следующий вид:

```
@string: replace(строка, паттерн, замещающая строка, флаги);
```

Под паттерном понимается строка или регулярное выражение, поиск которых будет производиться в переданной строке. Замещающая строка — это обычная строка, которая будет вставлена на место найденных совпадений с паттерном.

Соответственно, флаги передаются лишь в том случае, если планируется использовать паттерн как регулярное выражение.

Вот пример, демонстрирующий работу этой функции:

```
.block {  
  content: replace("Less имеет много нужных функций", "нужных", "интересных");  
}
```

В приведённом выше примере, слово «нужных» будет заменено на слово «интересных», что в итоге даст довольно правдивую строку:

```
.block {  
  content: "Less имеет много интересных функций";  
}
```

## Пример 4.2.1

В этом примере будет рассмотрен один из возможных способов применения функции замены значений в строке. Для этого я предлагаю написать примесь, которая задаёт фоновое изображение и его увеличенную версию для дисплеев с большей плотностью пикселей.

```
.bg(@path, @width, @height) {  
  background-image: url(@path);  
  
  @media only screen and (min-resolution: (1.5 * 96dpi)), only screen and (min-resolution: (1.5 * 1dppx)) {  
    @retina: replace(@path, "(\\.[0-9a-z]+)$", "@2x$1", "i");  
    background-image: url(@retina);  
    background-size: @width @height;  
  }  
}
```

```
.block {  
  .bg("../images/header-bg.png", 18px, 18px);  
}
```

Обратите внимание на переменную `@retina`, значение которой вычисляется на основе работы функции замены значений в строке. В качестве паттерна при вызове функции используется регулярное выражение, извлекающее из строки расширение файла. Затем к расширению прибавляется строка `@2x`, и уже новое значение возвращается в переменную `@retina`.

Результат работы этой примеси имеет следующий вид:

```
.block {
  background-image: url("../images/header-bg.png");
}

@media only screen and (min-resolution: 144dpi), only screen and (min-resolution: 1.5dppx) {
  .block {
    background-image: url("../images/header-bg@2x.png");
    background-size: 18px 18px;
  }
}
```

## Списки

Наиболее интересной возможностью Less, из описанных в этой части, является работа со списками, которые в мире программирования называются массивами.

Списки в Less выглядят следующим образом:

```
@list: "one", two, three four;
```

В приведённом мной примере, список состоит из трёх элементов: "one", two и three four.

Препроцессор Less не обязывает вас использовать запятые, то есть список может выглядеть следующим образом:

```
@list: "one" two three four;
```

Кавычки считаются вместе с элементом, вокруг которого они поставлены. Они также не обязательны, поэтому их можно опустить, если элемент не состоит из двух и более слов.

**Замечание**

Элементом списка может являться строка, переменная (исключая наборы правил) и другой список.

Для того, чтобы достать из списка какой-либо элемент, используется функция `extract`. Важно запомнить, что нумерация здесь начинается не с 0, как это принято, а с 1.

Допустим, что у нас есть частичный список глав этой книги и нам нужно достать значение текущей главы. Решается эта задача очень просто:

```
@list: "Основы", "Работа с селекторами, медиа-запросами и файлами", "Переменные и примеси", "Операции со свойствами и встроенные функции";
```

Для разнообразия применим возможность форматирования текста, описанную выше:

```
.block {
  @currentStatus: e(extract(@list, 4));
  content: %( "Текущая глава: %a", @currentStatus);
}
```

```
.block {
  content: "Текущая глава: Операции со свойствами и встроенные функции";
}
```

Напомню, что обычно функция `e()` применяется для экранирования. Однако, здесь она применяется лишь для того, чтобы избавиться от кавычек, которые являются частью элемента списка.

Помимо функции `extract()`, иногда применяется функция для определения длины списка — `length()`. Применим её к списку глав:

```
.block {
  content: length(@list);
}
```

Попробуйте догадаться, что выведет компилятор. Ну конечно же! Четыре:

```
.block {
  content: 4;
}
```

К сожалению, аналогов других методов, принятых в JavaScript в Less нет.

## Пример 4.2.2

Теперь давайте поработаем со списком в более серьезном ключе. Допустим, что у нас есть список префиксов для селекторов сетки (xs, sm, md, lg) и нам нужно добавить их при построении сетки.

К сожалению, данный пример нельзя продемонстрировать без применения цикла, поэтому сейчас будет немного слегка не очевидной магии. Просто знайте, что на **каждой итерации цикла** в селектор будет подставляться **значение из списка**. Более подробно о циклах смотрите в главе 5.

```
@column-name: col;
@column-count: 4;
@column-prefix: xs, sm, md, lg;

// Генератор селекторов
.generate-class(@indexCount, @indexPrefix: 1) when (@indexPrefix =< length(@column-prefix)) {

  // Получаем элемент списка
  @prefix: extract(@column-prefix, @indexPrefix);

  // Формируем селектор
  @{column-name}-@{prefix}-@{indexCount} {
    width: @indexCount * (100% / @column-count);
  }

  // Порождаем следующую итерацию
  .generate-class(@indexCount, @indexPrefix + 1);
}

// Генератор сетки
.make-grid(@indexCount: 1) when (@indexCount =< @column-count) {

  // Вызываем генератор селекторов
  .generate-class(@indexCount);

  // Порождаем следующую итерацию
  .make-grid(@indexCount + 1);
}

// Вызываем генератор сетки
.make-grid();
```

Кстати, в скомпилированном виде это выглядит так:

```
.col-xs-1 { width: 25%; }  
.col-sm-1 { width: 25%; }  
.col-md-1 { width: 25%; }  
.col-lg-1 { width: 25%; }  
  
/*  
* ...  
* Здесь ещё восемь классов, которые я убрал ради экономии места  
* ...  
*/  
  
.col-xs-4 { width: 100%; }  
.col-sm-4 { width: 100%; }  
.col-md-4 { width: 100%; }  
.col-lg-4 { width: 100%; }
```

Поздравляю с боевым крещением, так как циклы — это самое сложное, что есть в Less. Но не волнуйтесь — мы с ними еще встретимся в следующей главе, где будем говорить о них намного подробнее.



# Работа с изображениями

## Размер изображения

Отличительной особенностью Less версии 2.2.0 стали встроенные функции для работы с изображениями, которые позволяют получить следующие данные:

- Размер (ширина и высота) — `image-size(url)` .
- Ширину — `image-width(url)` .
- Высоту — `image-height(url)` .

Для того, чтобы получить ширину изображения, необходимо указать путь до него, относительно того файла, в котором вызывается эта функция.

### Замечание

Все примеры из этой части доступны в архиве, который прилагается к этой книге.

Рассмотрим два случая, которые могут вводить новичков в заблуждение:

### Случай 1. Один файл

Допустим, что работа ведётся с less-файлом, который находится в директории:

```
styles/less/_styles.less
```

В этом файле необходимо узнать ширину или высоту изображения, которое находится в директории:

```
images/logo.png
```

В этом случае, любая из трёх функций должна вызываться следующим образом:

```
@logo-path: "../../images/avatar.jpg";

.logo {
  @logo-width: image-width(@logo-path);
  @logo-height: image-height(@logo-path);
  @logo-size: image-size(@logo-path);

  content: "@{logo-size} [{logo-width} x @{logo-height}]";
}
```

В итоге, если вы правильно укажете путь до файла изображения из файла, где была вызвана функция, свойство `content` получит значение:

```
.logo {
  content: "460px 460px [460px x 460px]";
}
```

Если же путь будет не верным, то компилятор выдаст ошибку.

## Случай 2. Структурный файл

Ранее путь писался относительно того файла, в котором была вызвана та или иная функция для получения размера изображения. Допустим, что у нас имеется конфигурация:

- Рабочий файл: `styles/less/components/_styles.less`
- Главный файл: `styles/less/main.less`
- Файл изображения: `images/logo.png`

Содержимое каждого файла будет таким:

```
// file: `styles/less/main.less`

@import "components/_styles.less";

// file: `styles/less/components/_styles.less`

@logo-path: "../../images/avatar.jpg";

.logo {
  @logo-width: image-width(@logo-path);
  @logo-height: image-height(@logo-path);
  @logo-size: image-size(@logo-path);

  content: "@{logo-size} [{logo-width} x @{logo-height}]";
}
```

Хотя файл `_styles.less` поменял своё расположение — при компиляции ошибки не будет. Дело в том, что путь, указанный в переменной `@logo-path` полностью соответствует ситуации.

Вся хитрость заключается в файле `main.less`, который является точкой входа для компилятора. Именно относительно него нужно прописывать любой путь к изображениям.

При попытке указать путь, относительно файла `_styles.less`, компилятор выдаст ошибку, говорящую о том, что по указанному пути файл не найден:

```
FileError: error evaluating function `image-width`: '../../images/avatar.jpg' wasn't found.

C:\<-- path -->\chapter_4\examples\4.3.2\styles\less\components\_styles.less on line 4
, column 16:
3  .logo {
4    @logo-width: image-width(@logo-path);
5    @logo-height: image-height(@logo-path);
```

## Встраиваемые ресурсы

Под встроенными ресурсами стоит понимать закодированный ресурс (картинку, скрипт, стили), содержащийся внутри строки кода.

Сейчас, в виду того, что дизайн веб-приложений становится плоским и материальным, а возможности браузеров растут (количество изображений в оформлении падает), довольно часто можно встретить следующую конструкцию в стилях:

```
.logo {
  /* Пример из документации */
  background-image: url('data:image/jpeg;base64,bm90IGFjdHVhbGx5IGEganBlZyBmaWx1Cg==');
}
```

Этот подход используется для того, чтобы уменьшить число HTTP-запросов к серверу и должен применяться только тогда, когда количество и размер изображений достаточно мал.

При обнаружении такой конструкции в подключённых к странице стилях браузер декодирует строку, получает исходное изображение и выводит пользователю.

В багажнике препроцессора Less есть мощная функция `data-uri()`, предоставляющая возможность кодировать ресурсы без использования сторонних инструментов.

Синтаксис этой функции таков:

```
.logo {  
  background-image: data-uri(mimetype, path);  
}
```

Параметр **mimetype** определяет строку, содержащую тип изображения для браузера в соответствии с принятым документом RFC 1521. Этот параметр необязателен и его можно опустить, так как компилятор сам в состоянии понять, какой ему ресурс предоставили для кодирования.

Вот пример того, какие изображения можно закодировать с помощью этой функции и её результатов работы:

```
.logo {  
  background-image: data-uri("image.jpg");  
  // url("data:image/jpeg;base64,/9j/7gAQWRvYmUAZAAAAAAAAA/9sAQwA...")  
  
  background-image: data-uri("image/jpeg;base64", "image.jpg");  
  // url("data:image/jpeg;base64,/9j/7gAQWRvYmUAZAAAAAAAAA/9sAQwA...")  
  
  background-image: data-uri("image.png");  
  // url("data:image/jpeg;base64,iVBORw0KGgoAAAANSUHEugAAAEAAAAB...")  
  
  background-image: data-uri("image.gif");  
  // url("data:image/gif;base64,R0lGODlhQABAAPeAAP7+/v39/fz8/Pv7...")  
  
  background-image: data-uri("image.svg");  
  // url("data:image/svg+xml,%3C%3Fxml%20version%3D%221.0%22%20e...")  
  
  background-image: data-uri("image/svg+xml;charset=UTF-8", "image.svg");  
  // url("data:image/svg+xml;charset=UTF-8,%3C%3Fxml%20version%3...")  
}
```

Как не сложно заметить, различный результат проявился только у пары, где используется SVG изображение.

## SVG-градиенты

Кроме стандартных функций для работы с изображениями и их кодированием, препроцессор Less умеет генерировать SVG-градиенты. Для этого используется функция, которая принимает на вход три параметра: *направление*, *цвет* и *позиция*. Вот синтаксис этой функции:

```
.gradient {  
  background-image: svg-gradient(ellipse at center, blue, red 15%);  
}
```

Такой вызов функции породит Base64-код, в котором закодировано изображение сгенерированного градиента:

```
.gradient {  
  background-image: url('data:image/svg+xml,%3C%3Fxml%20version%3D%221.0%22%20%3F%3E%3Csvg%20xmlns%3D%22http%3A%2F%2Fwww.w3.org%2F2000%2Fsvg%22%20version%3D%221.1%22%20width%3D%22100%25%22%20height%3D%22100%25%22%20viewBox%3D%220%200%201%201%22%20preserveAspectRatio%3D%22none%22%3E%3CradialGradient%20id%3D%22gradient%22%20gradientUnits%3D%22userSpaceOnUse%22%20cx%3D%2250%25%22%20cy%3D%2250%25%22%20r%3D%2275%25%22%3E%3Cstop%20offset%3D%220%25%22%20stop-color%3D%22%230000ff%22%2F%3E%3Cstop%20offset%3D%2215%25%22%20stop-color%3D%22%23ff0000%22%2F%3E%3C%2FradialGradient%3E%3Crect%20x%3D%22-50%22%20y%3D%22-50%22%20width%3D%22101%22%20height%3D%22101%22%20fill%3D%22url(%23gradient)%22%20%2F%3E%3C%2Fsvg%3E');  
}
```

При декодировании этого кода можно получить внутренности изображения и убедиться, что функция создала верный градиент:

```
<?xml version="1.0" ?>  
  
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="100%" height="100%" viewBox="0 0 1 1" preserveAspectRatio="none">  
  <radialGradient id="gradient" gradientUnits="userSpaceOnUse" cx="50%" cy="50%" r="75%">  
    <stop offset="0%" stop-color="#0000ff"/>  
    <stop offset="15%" stop-color="#ff0000"/>  
  </radialGradient>  
  
  <rect x="-50" y="-50" width="101" height="101" fill="url(#gradient)" />  
</svg>
```

Конечно, функция может генерировать не только эллиптические градиенты. В список поддерживаемых направлений входят следующие значения:

- to (top | right | bottom | left)
- комбинации: to top bottom | to right bottom и прочие
- ellipse
- ellipse at center

### **Внимание**

Если вы используете компилятор версии ниже 2.2.0, то Base64 кодирование не поддерживается и, как следствие, функция `data-uri()` не будет работать. Кроме того, функция `svg-gradient()` не будет кодировать результат своей работы в Base64-код.

# Работа с типами данных и единицами измерения

## Функции для работы с единицами измерений

При написании примесей может понадобиться проверять переданные параметры при вызове, используя условные конструкции. Для этого Less имеет следующие функции:

- `convert()`
- `unit()`
- `get-unit()`

Давайте поговорим немного подробнее о каждой из них.

### Функция `convert()`

Как несложно догадаться, эта функция преобразует одну единицу измерения в другую. Принцип её работы очень прост:

- Если единицы измерения совместимы, то происходит преобразование и возвращается преобразованное число.
- Если единицы измерения не совместимы, то возвращается число без преобразований.

Список поддерживаемых единиц измерения:

- Длина: `m`, `cm`, `mm`, `in`, `pt` и `pc`.
- Время: `s` и `ms`.
- Углы: `rad`, `deg`, `grad` и `turn`.

Например, если нужно конвертировать метры в миллиметры, то функцию следует записать так:

```
.selector {
  content: convert(10m, "mm");
  // В результате
  // 10000mm
}
```

## Функция `unit()` и `get-unit()`

Обе функции предназначены для получения информации о переданном числе. Первая функция отбрасывает единицы и возвращает число, а вторая наоборот — отбрасывает число и возвращает единицы измерения.

Посмотрим на результат работы функций, если на вход подать различные данные. Для простоты и наглядности селектор и свойство убраны.

```
unit(10px) // 10
unit(10rem) // 10
unit(10cm) // 10

// Такой единицы нет в CSS
unit(10apple) // 10
```

Обратите внимание, что функция `unit()` не анализирует передаваемые ей данные, а просто возвращает число. Однако, если передать строку, то компилятор заподозрит что-то неладное и породит ошибку.

В случае с функцией `get-unit()` ситуация идентичная, исключая то, что при передаче строки ошибки не будет:

```
get-unit(10px) // px
get-unit(10rem) // rem
get-unit(10cm) // cm

// Такой единицы нет в CSS
get-unit(10apple) // apple

// Строка
get-unit("10string") // ""
```

## Работа с типами данных

Для определения типа данных в Less можно прибегнуть к помощи встроенных функций, которые возвращают логическое значение `true` или `false`.

Набор таких функций покрывает все типы, используемые в препроцессоре.

- `isnumber(value)`
- `isstring(value)`
- `iscolor(value)`
- `iskeyword(value)`



- `isurl(value)`
- `ispixel(value)`
- `isem(value)`
- `ispercentage(value)`
- `isruleset(value)`

Как следует из названия функций, каждая из них проверяет является ли переданное в неё значение валидным. Однако, существует несколько особенностей, которые стоит запомнить:

- Функция `isnumber(value)` не анализирует единицы измерения.
- Функция `iscolor(value)` поддерживает все принятые в CSS варианты записи цвета.

Отдельного внимания заслуживает функция `isunit()`, которая в отличие от других функций принимает на вход два параметра. Первый параметр — это значение, а второй — единицы измерения.

Синтаксис функции имеет следующий вид:

```
isunit(value, unit)
```

Эта функция работает следующим образом:

- Получаем единицы измерения параметра `value`.
- Сравниваем единицы измерения параметра `value` со значением параметра `unit`.
- Если значения равны, то возвращается `true`, иначе `false`.

# Математические функции

Для удобства работы с числами доступны математические функции, которые представляют собой прослойку между Less-функциями и встроенным объектом `Math` в JavaScript.

Кратко рассмотрим основные функции, которые могут пригодиться при работе с препроцессором при построении фреймворков и максимально унифицированных less-файлов.

## Округление значений

При необходимости значения, получаемые после проведения математических операций, можно округлять, используя стандартные методы `ceil()`, `floor()` и `round()`, представленные в Less в виде функций. Кроме того, Less предоставляет новую функцию `percentage()`.

**Функция** `ceil()` всегда округляет значения в большую сторону до целой части:

```
ceil(14.3) // 15
ceil(13.9) // 14
ceil(14.5) // 15
```

**Функция** `floor()` всегда округляет значения в меньшую сторону до целой части (отбрасывает дробную часть):

```
floor(14.3) // 14
floor(13.9) // 13
floor(14.5) // 14
```

**Функция** `round()` округляет значения в соответствии с правилами математики и заданным количеством знаков после запятой:

```
round(14.3) // 14
round(13.9) // 14
round(14.5) // 15

// Знаки после запятой
round(14.0714) // 14
round(14.0714, 1) // 14.1
round(14.0714, 2) // 14.07
round(14.0714, 3) // 14.071
round(14.0714, 4) // 14.074
round(14.0714, 7) // 14.074
```

**Функция** `percentage()` преобразует дробное значение в процентное:

```
percentage(0.25) // 25%
percentage(1 / 25) // 4%
```

Важно заметить, что дробная запись `1 / 25` сначала вычисляется как математическая операция, а уже потом передаётся в функцию. То есть, по сути своей, работает с числами с плавающей запятой.

## Прочие функции

Математические функции, представленные ниже, являются вторичными и применяются крайне редко. Дело в том, что возможность применять Less для сложных вычислений предоставляется очень редко, ведь это всего-лишь надстройка над CSS.

Тригонометрические функции:

- `sin(value)`
- `cosvalue)`
- `tan(value)`

Обратные тригонометрические функции:

- `asin(value)`
- `acos(value)`
- `atan(value)`

Модуль числа, модуль между числами:

- `abs(value)`
- `mod(valueOne, valueTwo)`

Поиск минимального и максимального значения:

- `min(list)` // `min(1, 14, 19, 0.3)`
- `max(list)` // `max(1%, 10%, 4%, 8%)`

Работа со степенями (квадратный корень и степень):

- `sqrt(value)`
- `pow(value, power)` // `value` в `power` степени

Математические константы:

- `pi()`

# Манипуляции с цветами

Для того, чтобы разрабатывать максимально унифицированные less-файлы, нужно автоматизировать свой код. Например, если ваше веб-приложение поддерживает темы, которые, как ни странно, могут менять свой цвет. В этом случае необходимо максимально возможно автоматизировать получение оттенков основного цвета темы. И это только самый базовый пример того, где могут понадобиться манипуляции с цветом.

Препроцессор Less предоставляет широкий спектр функций, позволяющих работать с цветами настолько тесно, насколько это вообще возможно.

## Функции определения цвета

При определении цветов Less заменяет некоторые стандартные объявления, принятые в CSS. Так, например, работает функция `rgb()`, возвращающая вместо традиционной записи цвет в формате HEX.

Если функции `rgb()` и `rgba()` в особых комментариях не нуждаются, то прочие требуют к себе особого внимания и понимания. Однако, сначала посмотрим на синтаксис для первых двух:

```
rgb(red, green, blue)
rgba(red, green, blue, alpha)
```

Параметры функции полностью совместимы с синтаксисом, принятым в CSS. Разница лишь в том, что функция `rgb()` преобразует RGB-формат записи цвета в HEX-формат. Также можно использовать не только стандартные значения из диапазона 0..255, но и процентную запись (0..100%).

### Замечание

Функция `rgba()` не производит никаких манипуляций, так как HEX-формат не поддерживает альфа-канал, задающий прозрачность.

Еще одним представителем семейства RGB является функция `argb()`, которая преобразует RGB-формат записи цвета в модифицированный HEX-формат, используемый в браузерах Internet Explorer и некоторых платформах (.NET, Android). Эта функция принимает на вход цвет в формате RGBA:

```
argb(rgba(red, green, blue, alpha))
```

Важно отметить, что после компиляции HEX-код цвета имеет формат `#AARRGGBB`, а не привычный для нас `#RRGGBBAA`.

## Семейство HSL

Название формата HSL образовано от сочетания первых букв **Hue** (оттенок), **Saturate** (насыщенность) и **Lightness** (светлота). Синтаксис функции имеет вид:

```
hsl(hue, saturation, lightness)
```

Оттенок задаётся с помощью градусной меры и может иметь значение от 0 до 359. Эти значения соответствуют градусам на цветовом круге.

Второе и третье числа в скобках означают насыщенность и светлоту соответственно и задаётся в процентах. Чем ниже значение насыщенности, тем более приглушённым становится цвет. Значение *saturation*, равное нулю, приведёт к серому цвету, при этом неважно, какое значение имеет *hue*. Третье значение позволяет указать яркость цвета. Чем значение ниже, тем более тёмный получается оттенок цвета, чем выше, тем светлее. При значении 100% у *lightness* получается белый цвет, соответственно, если 0% – чёрный.

HSL и HSLA поддерживаются всеми браузерами, кроме Internet Explorer ниже 8 версии. Однако, нам не нужно заботиться о поддержке браузерами, так как препроцессор Less преобразует запись функций `hsl()` и `hsla()`, соответственно в HEX и RGBA форматы.

```
hsl(60, 100%, 50%) // #ffff00  
hsla(60, 100%, 50%, .5) // rgba(255, 255, 0, 0.5)
```

### Замечание

Помимо процентного определения значения насыщенности и светлоты можно использовать значение в диапазоне от 0 до 1.

## Семейство HSV (HSB)

Аббревиатура HSV расшифровывается как: **Hue** (тон), **Saturation** (насыщенность), **Value** (значение). Иногда эту модель называют HSB, где последняя буква означает **Brightness** (яркость). Следует отметить, что HSV и HSL — две разные цветовые

модели (Lightness — светлота, что отличается от яркости, которую задаёт параметр value).

Синтаксис данных функций имеет вид:

```
hsv(hue, saturation, value)
hsva(hue, saturation, value, alpha)
```

Также как и в случае с HSL препроцессор преобразует получаемые значения в hex и RGBA:

```
hsv(60, 100%, 50%) // #808000
hsva(60, 100%, 50%, .5) // rgba(128, 128, 0, 0.3)
```

### Замечание

Помимо процентного определения значения насыщенности и яркости можно использовать значение в диапазоне от 0 до 1.

## Функции цветового канала

Как уже отмечалось выше, в Less используются функции для работы с форматами RGB, HSL и HSV, а также их аналогами с возможностью указания альфа-канала, задающего прозрачность. Чтобы получить значения каналов этих форматов, можно использовать встроенные функции.

Для RGB:

- `red()`
- `green()`
- `blue()`

Для HSL:

- `hue()`
- `saturation()`
- `lightness()`

Для HSV:

- `hsvhue()`
- `hsvsaturation()`
- `hsvvalue()`

Общие функции для RGB, HSL и HSV:

- `alpha()`
- `luma()`
- `luminance()`

Функции `luma()` и `luminance()` вычисляют, соответственно, яркость (яркость по восприятию или свечение) и яркость по восприятию (свечение) без коррекции гаммы.

#### Замечание

Функция `luminance()` устарела и, начиная с версии 1.7.0, функция `luma()` вычисляет яркость по восприятию без коррекции гаммы.

## Функции операций с цветами

В Less встроены функции для операций с цветами, основанные на работе с каналами. Все представленные функции работают с форматом HSL, и перед вычислением, если используется формат HEX, происходит его преобразование в HSL.

### Манипулирование насыщенностью цвета (Saturate и Desaturate)

Функция `saturate()` позволяет увеличить насыщенность цвета, заданного в формате HSL. Так, например, чтобы увеличить насыщенность цвета, параметры которого имеют значения `60`, `40%`, `50%` на `20%`, нужно:

```
@hs1: hsl(60, 40%, 50%) // #b2b34d
saturate(@hs1, 20%) // #cccc33
```

Функция `desaturate()` является обратной функцией `saturate()` и уменьшает значение насыщенности цвета. Например, вручную увеличим насыщенность цвета, который использовался ранее и проверим работу функции `saturate()` :

```
@hs1: hsl(60, 60%, 50%) // #cccc33
desaturate(@hs1, 20%) // #b3b34c
```

Обратите внимание на то, что значение полученное в ходе выполнения функции `desaturate()` отличается от того, что было задано в первом примере.

### Осветление и затемнение цвета (Lighten и Darken)



Функция `lighten()` (осветление) увеличивает значение параметра светлоты (Lightness) на указанное значение, тогда как функция `darken()`, обратная ей, уменьшает его.

Это две самые часто используемые функции, которые позволяют быстро получить цвета, например, для активного состояния пунктов меню или других элементов.

## Работа с прозрачностью (Fade, Fadein, Fadeout)

Функция `fade()` устанавливает значение альфа-канала, задающего прозрачность, в формате HSL, тем самым преобразуя формат из HSL в HSLA.

Соответственно, функция `fadein()` увеличивает значение альфа-канала, а функция `fadeout()`, обратная ей, уменьшает его.

## Управление тоном (Spin)

Эта функция, в отличие от предыдущих, не имеет пары и на её использование накладываются некоторые ограничения.

Функция `spin()` позволяет задавать значение, на которое отклоняется угол тона (Hue) на цветовом круге. На вход, как и другие функции, она принимает цвет в удобном для вас формате и положительное или отрицательное значение, на которое будет отклоняться угол тона. Например, если значение угла тона соответствует 245 и передать в функцию значение -45, то новое значение будет 200:

```
spin(hsl(245, 50%, 50%), -45) // #4095bf
hsl(200, 50%, 50%) // #4095bf
```

Обратите внимание, что если вы передаёте в функцию `spin()` значение в формате RGB, то оно преобразуется в формат HSL и не сохраняет оттенок серого, так как оттенок не имеет смысла, если нет никакого насыщения.

Таким образом, если в функцию передать оттенок серого, то результат будет равен исходному, за исключением того, что Less может преобразовать короткую запись цвета в полную:

```
spin(#ccc, 40%) // #cccccc
spin(#aaaaaa, 10%) // #aaaaaa
```

## Смешивание цветов в пропорциях (Mix)

Функция позволяет смешивать цвета в заданной пропорции, при этом учитывая значения альфа-канала. Попробуем смешать два белых цвета с указанной прозрачностью и пропорциями:

```
mix(rgba(255, 255, 255, 1), rgba(255, 255, 255, .5), 50%)
// rgba(255, 255, 255, 0.75)
mix(rgba(255, 255, 255, .8), rgba(255, 255, 255, .2), 60%)
// rgba(255, 255, 255, 0.56)
```

Если не углубляться в вычисление значений красного, зелёного и синего, то показать правильность работы этой функции можно на основе расчёта значения прозрачности:

```
1 * 50 / 100 + 0.5 * (1 - 50 / 100)
// 0.75
0.8 * 60 / 100 + 0.2 * (1 - 60 / 100)
// 0.56
```

## Обесцвечивание цвета (Grayscale)

Функция `grayscale()` позволяет обесцветить цвет, что соответствует удалению его насыщенности. По сути своей, представляет собой обёртку над функцией `desaturate()`.

## Нахождение контрастного цвета (Contrast)

Функция `contrast()` находит цвет, контрастный на фоне другого. Позволяет автоматизировать подбор контрастного цвета, в зависимости от цвета на входе. Так, например, контрастный цвет для белого — чёрный:

```
contrast(#fff) // #000
```

Функция принимает на вход до четырёх параметров:

- `color` — основной цвет для сравнения
- `dark` — тёмный цвет (опционально, изначально чёрный).
- `light` — светлый цвет (опционально, изначально белый).
- `threshold` — порог, позволяющий задать преобладание тёмного или светлого цвета при определении контрастного цвета (опционально, изначально 43%).

Посмотрим на работу порогового значения на примере:

```
contrast(green, black, white, 30%) // #000  
contrast(green, black, white, 43%) // #fff  
contrast(green, black, white, 80%) // #fff
```

## Функции смешивания цветов

При необходимости цвета можно смешивать, используя встроенные в препроцессор функции. По принципу работы они похожи на аналоги в графическом пакете Adobe Photoshop или любом другом, имеющем этот функционал. Для вычислений используются цвета в формате RGB, поэтому, перед тем как начать работу, происходит автоматическое преобразование любых других форматов в формат RGB.

### Режим наложения цветов (Multiply и Screen)

Функция **Multiply (Умножение)** накладывает один цвет на другой. При этом происходит перемножение значений каналов RGB. В результате цвет становится темнее.

Функция **Screen (Экран)** является противоположной для первой функции и при вычислениях использует инвертированные значения каналов RGB. Эта функция осветляет цвет.

### Управление контрастностью (Overlay, Soft light и Hard light)

Все эти функции входят в группу функций контрастности, которые увеличивают контрастность цвета.

Функция **Overlay (Перекрытие)** основана на работе функций **Multiply** и **Screen**, которые были рассмотрены ранее. Эта функция равномерно смешивает два цвета и делает их светлее или темнее, в зависимости от определяющего цвета, передаваемого как первый параметр.

Функция **Soft light (Мягкий свет)** делает результирующий цвет более темным или светлым в зависимости от совмещенного цвета. Если вносимый цвет (второй параметр) светлее 50-процентного серого, исходный цвет становится светлее. Иначе, если вносимый цвет темнее 50-процентного серого, исходный цвет затемняется.

Функция **Hard light (Жёсткий свет)** работает так же, как и функция **Overlay**. Разница лишь в том, что передаваемые параметры меняются местами при вызове функции **Overlay**.

## Функции разницы и исключения (Difference и Exclusion)

Функция **Difference (Разница)** вычисляет математический модуль между двумя цветами, что соответствует вычитанию одного цвета из другого. Смешивание с белым цветом приводит к инвертированию значений основного цвета, смешивание с чёрным цветом не влечёт за собой каких-либо изменений и возвращает исходный цвет.

Функция **Exclusion (Исключение)** аналогична работе функции **Difference**, но применяется для цветов с низкой контрастностью.

## Функция среднего и отрицания (Average и Negation)

Все предыдущие функции смешивания цветов основывались на рекомендациях консорциума всемирной сети (W3C). Функции **Average** и **Negation** были разработаны для удобства создателями препроцессора.

Функция **Average (Среднее)** находит среднее между двумя цветами. Работа этой функции основана на принципах школьной математики, где слагаемыми выступают числа. Так как мы работаем с RGB-форматом, то здесь происходит нахождение среднего для каждого канала между двумя цветами.

Функция **Negation (Отрицания)** основана на работе функции **Difference** с той лишь разницей, что результат вычитается из 1. В результате получается цвет, который ярче исходного.

## Выводы и мысли

Если сайт имеет различные цветовые палитры для тем, то возможно автоматически манипулировать цветами. При этом, если изменить одну единственную переменную, в которой указан основной цвет темы, может полностью смениться палитра сайта, причём без ручного изменения значения цвета у каждой переменной. Удобный и логичный функционал, но на деле он используется крайне редко, так как быстрее и проще определить цвет по предоставленному дизайнером изображению, нежели чем строить конструкции из функций для манипуляции цветом.

## Домашнее задание

На этот раз читателю предлагается решить несколько задач для закрепления пройденного материала.

### Задача 1. Вычисление значения

На основе знаний, полученных из главы 3 и 4, необходимо написать примесь, которая возвращает переменную `@result`. Для этого предлагается использовать примесь как функцию.

В этой функции требуется найти результат следующего математического выражения:

$$\frac{\sqrt{a} + \cos(b) + \max(a, b, c)}{c^5}$$

При решении следует использовать математические функции.

### Задача 2. Работа с цветом

На основе знаний, полученных в этой главе, требуется написать less-файл, в котором:

- Инициализирована переменная `@color` с любым произвольным цветом.
- Определены переменные `@color-darken` и `@color-light`, значения которых зависят от переменной `@color` следующим образом:
  - Значение `@color-dark` является значением переменной `@color`, затемнённым на 25%.
  - Значение `@color-light` является значением переменной `@color`, осветлённым на 25%.
- Определён цвет ссылки по умолчанию, как значение переменной `@color-dark`, а цвет при наведении курсора мыши является значением переменной `@color-light`.

Не волнуйтесь, у вас все получится! Если у вас возникают какие-либо сложности, то посмотрите решение этого задания в архиве, который прилагается к этой книге.



## Глава 5. Инструкции (операторы)

Препроцессор Less включает несколько инструкций (операторов), называемых также *управляющими инструкциями (конструкциями)*. Набор инструкций тривиален и включает в себя только классический `if` и `while`, причём со своим синтаксисом, назначением и большим количеством ограничений.

Категоричность — признак ограниченности.

Конфуций

## Условные конструкции (защита примесей)

Условная конструкция — это самая часто используемая почти во всех языках программирования инструкция, к сожалению, отсутствующая в явном виде в Less.

Обычно, различают три вида условных конструкций по количеству ветвей: одно-, дву- и многоветвевую.

В JavaScript очень часто встречаются двуветвевые условные конструкции, где в зависимости от результата выражения выполняется тот или иной блок (true или false):

```
if (выражение) {  
  // True  
} else {  
  // False  
}
```

## Ситуация в Less

В Less нет такого понятия, как условная конструкция. Здесь оперируют понятием — защита примесей. Ниже представлен обычный `if` :

```
.mixin(@a, @b) when (выражение) {  
  // True  
}
```

Примесь будет выполняться только тогда, когда выражение, указанное после ключевого слова `when` будет истинно.

## Операторы отношений

Операторы отношений: «меньше» ( `<` ), «больше» ( `>` ), «меньше или равно» ( `=<` или `<=` ), «больше или равно» ( `>=` ) и «равно» ( `=` ) сравнивают значения так же, как и в JavaScript. Каждый из них возвращает логическое значение.



### Замечание

Оператор отношения «меньше или равно» (`<=` или `<=<`) отличается от аналогичного в JavaScript, где его запись имеет только один вид (`<=`). В соответствии с документацией принято использовать (`<=<`).

## Логическое НЕ, И и ИЛИ

Условия могут содержать ключевое слово `not`, являющееся аналогом привычного нам по JavaScript отрицания `!`.

Например, следующая примесь будет выполняться, если переданное ей значение `@value` не равно нулю.

```
.mixin(@value) when not (@value = 0) {  
  color: #777;  
}
```

Также, условия можно объединять, создавая более конкретные условия. Для этого используется ключевое слово `and`.

```
.mixin(@value) when (@value > 0) and (@value <= 100) {  
  color: #777;  
}
```

Ключевое слово `and` можно опускать, используя взамен запятую:

```
.mixin(@value) when (@value > 0), (@value <= 100) {  
  color: #777;  
}
```

Начиная с версии **2.6.0** компилятор поддерживает ключевое слово `or`, соответствующее логическому «ИЛИ». Синтаксис повторяет синтаксис ключевого слова `and`, но никаких синонимов взамен не предлагает (у `and` синонимом является запятая):

```
.mixin(@value) when (@value > 0) or (@value = -1) {  
  content: @value;  
}
```

### Замечание

Логический оператор `and` имеет больший приоритет, чем логический оператор `or`, как и в любом языке программирования.

Помимо написанных вручную условий, можно использовать некоторые встроенные функции для проверки величин на совпадение с различными типами и размерностями.

## Работа с типами данных

В главе 4 были рассмотрены функции для определения типа данных, которыми оперирует пользователь препроцессора Less. Все эти функции можно использовать в условном операторе. Причём важно отметить, что Less автоматически преобразует результат выражения в логическое значение, как это делает JavaScript, вызывая для него функцию `Boolean()`.

Так, например, рассмотренная ранее функция `isnumber()`, проверяющая являются ли предоставленные ей данные числом, может применяться следующим образом:

```
.return-number(@number) when (isnumber(@number)) {
  content: @number;
}

.true {
  .return-number(123);
}

.false {
  .return-number("123");
}
```

В первом случае будет создан селектор `.true`, так как значение, передаваемое в примесь, является числом. Во втором же случае, передаваемое значение — строка, что означает `false`.

## Конструкция `if {} else {}`

С помощью ключевого слова `default()` можно определить примесь по умолчанию, которая будет выполняться при условии, что другие примеси не сработали.

```
.mixin(@a, @b) when (@a > @b) {  
  content: if;  
}  
  
.mixin(@a, @b) when (default()) {  
  content: else;  
}
```

Вызовем примесь `mixin` с параметрами (3, 1). В этом случае выполняется первая примесь, потому что введённые данные подходят под указанное в ней выражение (3 больше чем 1).

Если вызвать эту же примесь с параметрами (1, 1), то выполнится вторая примесь, так как выражение первой примеси вернуло `false`.

## Конструкция `if {} else if {} else {}`

Ситуация полностью аналогична предыдущей, с той лишь разницей, что примесей будет три:

```
.mixin(@a, @b) when (@a > @b) {  
  content: if;  
}  
  
.mixin(@a, @b) when (@a < @b) {  
  content: else if;  
}  
  
.mixin(@a, @b) when (default()) {  
  content: else;  
}
```

Теперь условная конструкция выводит `if`, если *a* больше *b*, `else if`, если *a* меньше *b*, иначе `else`.

### Пример 5.1.1

Разберём работу примеси, которая генерирует внутренние отступы блока по правилу:

- Если указан один параметр, то генерировать свойство с одним значением.
- Если указано два параметра, то генерировать свойство с двумя значениями.
- и т.д.

Итак, необходимо использовать значения по умолчанию для того, чтобы не приходилось каждый раз указывать значения для всех параметров:

```
.padding(@t: none, @r: none, @b: none, @l: none) {
}

```

Теперь можно приступить к написанию условий:

```
.padding(@t: none, @r: none, @b: none, @l: none) {
  & when not (@t = none) { padding+_: @t; }
  & when not (@r = none) { padding+_: @r; }
  & when not (@b = none) { padding+_: @b; }
  & when not (@l = none) { padding+_: @l; }
}

```

Здесь использовалась возможность слияния значения свойств, которая обсуждалась в главе 4.

В итоге мы имеем примесь, которая выводит:

```
.padding(1px) // padding: 1px;
.padding(1px, 2px) // padding: 1px 2px;
...

```

## Примеси как функции и условные конструкции

Следует аккуратно использовать сокращённую запись условной конструкции, которая имеет вид:

```
.mixin(@a) {
  & when (@a = 1) {
    @return: value;
  }
}

```

Если вызвать эту примесь сейчас с параметром 1 и попытаться получить значение переменной `@return`, то компилятор выдаст ошибку, говорящую о том, что такой переменной нет:

```
.test {
  .mixin(1);
  content: @return;
}

```

Ошибка имеет вид:

```
NameError: variable @return is undefined in
C:\...\a.test on line 9, column 12:
8   .mixin(1);
9   content: @return;
10 }
```

Кстати, вы заметили, что я создал файл с расширением `.test`, и компилятор корректно его обработал? Напомню, что это обсуждалось в главе 1 (Импорт стилей в Less).

Связано это с тем, что ссылка на родительский селектор (`&`) создаёт новую область видимости. Подробнее об этом в главе 3 в разделе «Области видимости».

В тоже время, если использовать полную запись, то ошибки не будет:

```
.mixin(@a) when (@a = 1) {
  @return: value;
}

.test {
  .mixin(1);
  content: @return;
}
```

Этот код будет генерировать селектор `.test` со свойством `content`, имеющим значение переменной `@return`.

# Циклические конструкции

В Less нет стандартных циклов `for` или `while`. Однако, текущих возможностей формирования циклических конструкций хватает для всего круга задач, связанного с вёрсткой.

Как я уже говорил ранее, циклы строятся на принципе защиты примесей с помощью условного оператора (`when`). Таким образом, нужно понимать, что все циклы в Less построены на принципе рекурсий или, как напоминает нам JavaScript, цикле с предусловием `while` (выполнение условия проверяется перед тем, как войти в тело цикла).

Давайте подробно разберём пример, код которого формирует некоторое количество классов, жутко напоминающих простую сетку. Вот код с небольшими комментариями очевидных вещей:

```
@column-name: col; // Префикс класса сетки
@column-count: 4; // Количество классов

.generate-column(@i: 1) when (@i =< @column-count) {
  .{@column-name}-@{i} {
    width: @i * (100% / @column-count);
  }

  .generate-column(@i + 1);
}

.generate-column();
```

Основную работу выполняет примесь `generate-column`, принимающая на вход параметр  $i$  и защищённая с помощью ключевого слова `when`. Отсюда следует, что вся магия циклов в Less заключается в условной конструкции и рекурсивном вызове.

Так как примесь принимает на вход параметр  $i$ , который по сути является итерационной переменной, а также у примеси указано условие, при котором она будет работать, получается, что мы создаём простой вариант рекурсии. Она будет существовать до тех пор, пока значение переменной `@i` не сравняется с таковым у `@column-count` или не будет больше него.

Заглянем внутрь примеси. Здесь формируется имя селектора по правилу: *префикс + номер итерации*. Уже затем для этого селектора вычисляется значение ширины, как: *100% ширины, делённое на количество колонок и умноженное на шаг итерации*.

Обратите внимание на строку `.generate-column(@i + 1)`. Эта команда просит компилятор вызвать эту же примесь, но с увеличенным на единицу шагом итерации. Весь процесс повторяется вновь, но уже с увеличенным на единицу значением переменной `@i`. В этом и заключается магия.

Для наглядности процесса попробуем вместе с компилятором пройти пару итераций:

```
/* Итерация 1 | @i: 1 | Новое значение @i: 2 */
.col-1 { width: 25%; }

/* Итерация 2 | @i: 2 | Новое значение @i: 3 */
.col-1 { width: 25%; }
.col-2 { width: 50%; }

/* Итерация 3 | @i: 3 | Новое значение @i: 4 */
.col-1 { width: 25%; }
.col-2 { width: 50%; }
.col-3 { width: 75%; }

/* Итерация 4 | @i: 4 | Новое значение @i: 5 */
.col-1 { width: 25%; }
.col-2 { width: 50%; }
.col-3 { width: 75%; }
.col-4 { width: 100%; }
```

После того, как шаг итерации достигает максимально допустимого значения, происходит выход из рекурсии и компилятор продолжает бороздить красоты нашего кода. В нашем случае этот момент наступает на четвёртом шаге рекурсии, когда новое значение переменной `@i` равно 5, а максимальное число шагов итераций, заданное переменной `@column-count`, определено как 4.

Таким образом строятся и более сложные циклы, в которых участвуют несколько счётчиков, примесей или даже списков.

## Обработка списка

Рассмотрим ещё один пример, где в цикле генерируются вспомогательные классы на основе списка. Этот пример вы уже встречали в главе 4, когда речь шла про списки (массивы).

Напомню его устрашающий код, который тогда мог повергнуть неподготовленного читателя в ужас:

```
// Настройки
@column-name: col;
@column-count: 4;
@column-prefix: xs, sm, md, lg;

// Генератор селекторов
.generate-class(@indexCount, @indexPrefix: 1) when (@indexPrefix =< length(@column-prefix)) {

    // Получаем элемент списка
    @prefix: extract(@column-prefix, @indexPrefix);

    // Формируем селектор
    @{column-name}-@{prefix}-@{indexCount} {
        width: @indexCount * (100% / @column-count);
    }

    // Порождаем следующую итерацию
    .generate-class(@indexCount, @indexPrefix + 1);
}

// Генератор сетки
.make-grid(@indexCount: 1) when (@indexCount =< @column-count) {

    // Вызываем генератор селекторов
    .generate-class(@indexCount);

    // Порождаем следующую итерацию
    .make-grid(@indexCount + 1);
}

// Вызываем генератор сетки
.make-grid();
```

Теперь будем разбираться с ним вплоть до мелочей. Начнём с того, что перепишем весь этот код с Less на JavaScript. Зачем? — скоро все поймёте.

## Понимание деталей

Если абстрагироваться от Less к JavaScript, то здесь объявлены две функции, которые, как кирпичики лего, позволяют строить лего-замок. Каждая функция выполняет свою простейшую задачу и, на практике, все функции можно было бы объединить в одну, конечно, при условии, что вам нравится запутанный код.

Для понимания происходящего, проведём параллель между обычным JavaScript-кодом и примесями в Less.



## Настройки

В коде, приведённом ниже, объявляется объект `columnOptions`, хранящий настройки генератора сетки, такие как:

- `Name` — Имя столбца
- `Count` — Количество столбцов
- `Prefix` — Массив префиксов

```
// Настройки
var columnOptions = {
  name: 'col',
  count: 4,
  prefix: ['xs', 'sm', 'md', 'lg']
};
```

Если сравнивать с `Less`, то там настройки хранятся в переменных. Так, например, в JavaScript свойство объекта `Prefix` соответствует переменной `@column-prefix` в `Less`.

```
// Настройки
@column-name: col;
@column-count: 4;
@column-prefix: xs, sm, md, lg;
```

## Вызов

Так как генерировать сетку на JavaScript никто не будет, то и вывод результатов будет осуществляться в консоль:

```
// Вызываем генератор сетки
console.log(makeGrid(columnOptions));
```

Разумеется, что в `Less` примесь вызывается стандартным образом:

```
// Вызываем генератор сетки
.make-grid();
```

Итак, переходим вместе с кодом в примесь **make-grid** и отслеживаем работу дальше.

## Генератор сетки

Код ниже представляет собой полный эквивалент примеси **make-grid**, переписанной на классический ванильный JavaScript.

Переменная `consoleReturn` представляет собой массив объектов, содержащих в себе пары ключей:

- `Name` — Имя текущего префикса (`xs`, `sm..`).
- `Selector` — Объект, содержащий информацию о колонке (имя и ширина), который возвращает функция `generateSelector`.

Далее в цикле производится проход по всем, указанным в настройках префиксам и на каждой итерации инициируется вызов функции, генерирующей информацию о селекторе.

Соответственно, вся полученная информация возвращается пользователю в консоль для анализа.

```
// Генератор сетки
var makeGrid = function(o) {
  var consoleReturn = [];
  for (var index = 0; index < o.count; index++) {
    consoleReturn.push({
      name: o.prefix[index],
      selector: generateSelector(o, index)
    });
  }

  return consoleReturn;
};
```

Теперь снова посмотрим на нашу оригинальную примесь `make-grid` и разберёмся с тем, что там происходит:

```
// Генератор сетки
.make-grid(@indexCount: 1) when (@indexCount =< @column-count) {

  // Вызываем генератор селекторов
  .generate-class(@indexCount);

  // Порождаем следующую итерацию
  .make-grid(@indexCount + 1);
}
```

Если исключить тот факт, что в Less данные возвращаются автоматически и в рекурсивно вызываемой примеси `generate-class`, то можно с уверенностью сказать, что эта примесь работает так же, как и функция, написанная на JavaScript.

Вместо цикла `for` здесь используется рекурсия, которая работает до тех пор, пока итерационная переменная `@indexCount` меньше, либо равна количеству колонок, указанному в настройках.

На каждой итерации вызывается примесь генератора классов сетки, в которую передаётся переменная, указывающая номер итерации. Далее вызывается текущая примесь со значением итерационной переменной, увеличенной на единицу, чтобы породить рекурсию. То есть принцип действия полностью соответствует примеру, рассмотренному в начале этой части.

Проводя аналогию с функцией на JavaScript, можно выделить следующие наблюдения:

- Аналогом цикла `for` в Less является строка `.make-grid(@indexCount + 1)`
- Аналогом выражения цикла `for` в Less является условие `when` или, как его принято называть — выражение защиты примесей.

Перейдём к рассмотрению примеси, генерирующей селекторы.

## Генератор селекторов

Ниже представлен эквивалент примеси **generate-class** на языке JavaScript, с той лишь разницей, что создавать рекурсию в JavaScript — кощунство, поэтому тут применяется стандартный цикл `for`.

```
// функция генерации селекторов
var generateSelector = function(o, indexCount) {
  var selectors = [];
  for (var index = 0; index < o.prefix.length; index++) {
    var name = o.name + '-' + o.prefix[indexCount] + '-' + index;
    var width = index * (100 / o.count) + '%';
    selectors.push({
      name: name,
      width: width
    });
  }

  return selectors;
};
```

Эта функция принимает на вход объект настроек и номер текущей итерации. Массив `selectors` будет содержать имя и ширину селектора для всех префиксов из массива. Получается, что в нём будет храниться эквивалент CSS-кода:

```
.col-xs-1 {  
  width: 25%;  
}  
.col-sm-1 {  
  width: 25%;  
}
```

Углубляться в то, как он будет представлен не имеет смысла, так как главное здесь — понимать суть. В итоге, из этой функции будет возвращаться массив колонок одного префикса.

Снова обратим внимание на оригинальную less-примесь и проанализируем её действие:

```
// Генератор селекторов  
.generate-class(@indexCount, @indexPrefix: 1) when (@indexPrefix =< length(@column-prefix)) {  
  
  // Получаем элемент списка  
  @prefix: extract(@column-prefix, @indexPrefix);  
  
  // Формируем селектор  
  .@{column-name}-@{prefix}-@{indexCount} {  
    width: @indexCount * (100% / @column-count);  
  }  
  
  // Порождаем следующую итерацию  
  .generate-class(@indexCount, @indexPrefix + 1);  
}
```

Как уже отмечалось выше, `when` играет роль выражения цикла, а последняя в примеси строка порождает новую итерацию этой же примеси, что эквивалентно новой итерации стандартного цикла `for`.

В отличие от JavaScript кода, примесь генерирует селектор и его свойства, а также возвращает их в виде CSS-кода по правилу:

```
Имя колонки + текущий префикс + номер итерации
```

Ширина колонки вычисляется тем же методом, что и в первом примере, описанном в этой части.

Таким образом, примесь будет рекурсивно выполняться до тех пор, пока не будут созданы селекторы для всех префиксов, указанных в переменной `@column-prefix`.

## Результаты

После того, как компилятор пройдёт все итерации для каждой примеси, он вернёт сгенерированные селекторы на то место, где была вызвана управляющая примесь **make-grid**.

Если не изменять определённые ранее настройки, результат будет следующим:

```
.col-xs-1 { width: 25%; }  
.col-sm-1 { width: 25%; }  
.col-md-1 { width: 25%; }  
.col-lg-1 { width: 25%; }  
...  
.col-xs-4 { width: 100%; }  
.col-sm-4 { width: 100%; }  
.col-md-4 { width: 100%; }  
.col-lg-4 { width: 100%; }
```

Вывод JavaScript функций имеет приблизительно такой же вид с поправкой на то, что там данные не предоставляют ценности для CSS.

## Выводы

Полностью рассмотрев два примера оценить всю природу циклических конструкций в Less нельзя. Зато можно понять их базовую сущность. Со временем вы научитесь составлять и более сложные конструкции, если это потребует поставленная задача. Да, циклы в Less имеют слегка непривычный вид для JavaScript-разработчиков, но разобравшись с ними раз и навсегда, проблем в дальнейшем не предвидится.

## Домашнее задание

В этом домашнем задании вам предстоит написать код для генерации вспомогательных классов на основе предоставляемых данных в виде двух списков.

Входные данные будут такими:

```
// Имена классов
@listNames: black, red, purple, green, blue;
// Цвет
@listColors: #000000, #ff0000, #800080, #008000, #0000ff;
```

Индексы двух списков совпадают. Так, например, класс `black` имеет цвет `#000000`, а `purple` — `#800080`.

Результат должен совпадать с вручную написанным CSS-кодом:

```
.color-black { color: #000000; }
.color-red    { color: #ff0000; }
...
.color-blue   { color: #0000ff; }
```

### Замечание

Результат отформатирован вручную для наглядности.

## Глава 6. JavaScript в Less

Этой главой заканчивается основная, фундаментальная часть книги, рассказывающая о возможностях CSS-препроцессора Less. Финал должен быть красивым, и обеспечить его может только недокументированная возможность исполнения JavaScript-кода в less-файлах.

Из этой главы вы узнаете, как использовать JavaScript-код в Less, манипулировать введенными данными, обрабатывать и возвращать их.

Ваши дома — коробки, а ваши машины — это коробки на колесах. На работу в коробке и в коробке домой.

Посылка (The Box)

# Обзор возможностей

Одной из самых интересных особенностей препроцессора Less является то, что его создатели предусмотрели возможность исполнения JavaScript-кода в less-файлах. Казалось бы, это весомое преимущество и полная свобода действий, но не всё так просто.

Во-первых, JavaScript-код может быть лишь частью операции присвоения (переменные, свойства). Это означает, что введённая строка, содержащая JavaScript-код, будет интерпретироваться компилятором и обрабатываться.

Во-вторых, компилятор накладывает сразу несколько ограничений:

- Если возвращаемый результат число, то оно преобразуется к строке.
- Если возвращаемый результат строка, то она заключается в кавычки.
- Если возвращаемый результат массив, то все элементы массива конкатенируются и возвращаются как строка.
- Если ни одно из выше перечисленных условий не работает, то результат просто приводится к строке.

Отсюда следует самое главное ограничение — **результатом интерпретации выражения всегда будет строка**.

Это очень сильно ограничивает возможность использования столь диковинной для CSS-препроцессора функции.

Так как же намекнуть компилятору о том, что перед ним не просто строка, содержащая JavaScript-код, а строка, которую следует интерпретировать? — очень просто:

```
@js: `выражение`;
```

Иногда может понадобиться избавиться от обрамляющих кавычек. Для этого применяется ранее рассмотренная тильда ( `~`` ), позволяющая экранировать содержимое строки:

```
@js: ~`выражение`;
```

Достаточно всего лишь обернуть строку, код которой необходимо интерпретировать в апострофы.



### Замечание

Не используйте точку с запятой перед закрывающим апострофом. В противном случае вы получите ошибку интерпретации.

## Простейшие выражения

Простейшими, интерпретируемыми выражениями могут быть любые инструкции. Ниже я предлагаю рассмотреть некоторые из возможных JavaScript-инструкций, которые вы можете использовать в компиляторе Less.

Для начала компилятор может просто вернуть число или, допустим, массив:

```
.test-js {
  content: `1`;
  content: `[1, 2, 3]`;
}

// После компиляции
.test-js {
  content: 1;
  content: 1, 2, 3;
}
```

Заметьте, что массив был возвращён как строка, о чём и говорилось в самом начале.

Теперь посмотрим на то, что компилятор умеет складывать числа и даже считать математические функции:

```
.test-js {
  content: `1 + 6 + Math.cos(1)`;
}

// После компиляции
.test-js {
  content: 7.54030231;
}
```

Помимо каких-либо арифметических операций можно обращаться к методам Node.js, если вы компилируете код под этой платформой:

```
.test-js {
  content: `process.platform`;
}
```

Так как моей основной платформой пока ещё является Windows, то результат можно легко предугадать:

```
.test-js {
  content: "win32";
}
```

Однако, никто не запрещает использовать JavaScript на полную в разумных пределах. В следующем примере выводится информация об используемой памяти процессом Node.js в байтах:

```
.test-js {
  content: `JSON.stringify(process.memoryUsage())`;
}

// Компилируется в
.test-js {
  content: '{"rss":23982080,"heapTotal":15454976,"heapUsed":8824096}';
}
```

Для особых ценителей уточню, что использовать `require()` здесь нельзя, поэтому поднять веб-сервер в less-файле нельзя. Зато можно писать функции:

```
.test-js {
  content: `(function() {
    var a = 2;
    var b = Math.pow(a, 4);

    return a + b;
  })()`;
}
```

Результатом работы этого выражения будет число 18 ( `a = 2` , `b = 16` ), как несложно догадаться, преобразованное к строке:

```
.test-js {
  content: 18;
}
```

# Работа с переменными и примесями

Чтобы компилятор выполнял хоть сколько-нибудь полезные действия, интерпретируя JavaScript-код — ему нужны значения, которые он может получить из переменных. Для того, чтобы получить значения переменных, необходимо использовать один из представленных ниже вариантов синтаксиса:

```
.test-js {
  @test: 123;
  content: `@{test}`;
  content: `this.test.toJS()`;
}
```

Оба варианта хорошо работают с локальными переменными. С глобальными переменными работает лишь первый способ, так как `this`, во втором случае, явно указывает на контекст, то есть селектор. Без контекста `test.toJS()` работать не будет.

Результатом компиляции будет:

```
.test-js {
  content: 123;
  content: "123";
}
```

Далее я приведу несколько примеров работы с переменными, используя JavaScript-код в Less:

```
.test-js {
  // Интерполяция
  @world: "world";
  content: ~`'hello' + ' ' + @{world}`;

  // Списки
  @list: 1, 2, 3;
  @list-js: ~`@{list}.join(', ')`;
  content: @list-js;
}
```

Компилятор создаст два свойства `content` и присвоит им вполне валидные значения:

```
.test-js {  
  content: hello world;  
  content: 1, 2, 3;  
}
```

Первый пример демонстрирует возможность интерполяции строк напрямую в JavaScript, второй — работу со списками.

Немного подробнее остановимся на списках. Ранее я уже говорил, что списки — это альтернатива массивам в JavaScript. Списки в Less можно итерировать и узнавать их длину. Тот список, что определён в переменной `@list` никаких вопросов не вызывает:

```
.test-js {  
  @list: 1, 2, 3;  
  content: length(@list);  
  content: extract(@list, 1);  
}  
  
// На выходе получаем  
.test-js {  
  content: 3; // Длина  
  content: 1; // Первый элемент массива  
}
```

А вот значение переменной `@list-js` списком не является, так как на выходе JavaScript-кода всегда находится строка:

```
.test-js {  
  @list: 1, 2, 3;  
  @list-js: ~`@{list}.join(', ')`;  
  content: length(@list-js);  
  content: extract(@list-js, 1);  
}  
  
// На выходе получаем  
.test-js {  
  content: 1; // Длина  
  content: 1, 2, 3; // Первый элемент массива  
}
```

## Примеси

Наиболее очевидным применением возможностей JavaScript-кода в Less является создание примесей, которые на вход получают какое-то количество переменных, обрабатывают их, используя JavaScript и возвращают строку, как результат.

## Конечное число переменных

Самым простым способом получить значения из переменных в Less является следующая функция:

```
(function(a, b) {  
  return a + b;  
})('@{a}', '@{b}')
```

Используя обёртку в виде примеси, её можно представить в удобном для использования виде:

```
.mixin(@a, @b) {  
  @js: ~`(function(a, b) { return a + b; })('@{a}', '@{b}')`;   
  
  content: @js;  
}  
  
.test-js {  
  .mixin(1, 2);  
}
```

Результатом компиляции будет являться сумма двух чисел, переданных, как аргументы примеси `.mixin()`:

```
.test-js {  
  content: 12;  
}
```

## Неопределённое число переменных

Если для проведения операций в выражении требуется большое количество переменных, или их количество неизвестно заранее, то на помощь приходит следующая функция, возвращающая массив всех переданных аргументов:

```
(function(args) {  
  return args;  
})('@{arguments}')
```

Записывая эту функцию в переменную и используя примесь, у которой на вход подаётся переменное количество аргументов, получим следующий less-код:

```
.mixin(...) {
  @js: ~`(function(args){ return args; })('@{arguments}')`;

  content: @js;
}

.test-js {
  .mixin(3, 123);
}
```

И, как я уже сказал, после компиляции будет доступен массив всех переданных значений:

```
.test-js {
  content: [3, 123];
}
```

В Less с таким результатом сделать ничего не получится (мешают квадратные скобки), поэтому на практике лучше всего использовать следующую модификацию предложенной функции:

```
(function(args) {
  return args;
})((function() {
  var args = '@{arguments}';
  return args.replace(/^\[|\]$\/g, '');
}))()
```

Этот вариант записи удаляет квадратные скобки, используя метод `replace()`, при этом делая получаемый на выходе массив немного лучше:

```
.mixin(...) {
  @js: ~`function(args){return args}(function(){var args='@{arguments}';return args.replace(/^\[|\]$\/g,'')})();`;

  content: @js;
}

.test-js {
  .mixin(3, 123);
}
```

К сожалению, как я и показывал ранее — массив преобразуется в строку, содержимое которой нельзя итерировать.

```
.test-js {
  content: 3, 123;
}
```

## Преобразование значений

Конечно, на практике мало пользы от того, что вы можете получить, распарсить и отдать результат обратно — необходимо с ним как-то взаимодействовать.

В приведённом ниже примере последнему в списке значению добавляется единица измерения `deg`:

```
(function(args) {
  return args = args || '0, 0, 0, 0', args = args.replace(/, \s*\d+$/, function(args) {
    return args + 'deg'
  })
})((function() {
  var args = '{@arguments}';
  return args = args.replace(/^\[|\]$ /g, '')
}))()
```

В итоге примесь имеет вид:

```
.rotate3d(...) {
  @js: ~`(function(args) { return args = args || '0, 0, 0, 0', args = args.replace(/, \s*\d+$/, function(args) { return args + 'deg' }) })((function() { var args = '{@arguments}'; return args = args.replace(/^\[|\]$ /g, '') })())` ;
  transform: rotate3d(@js);
}

.test-js {
  .rotate3d(1, 0, 0, 50);
}
```

После компиляции получится отформатированное значение свойства `transform`:

```
.test-js {
  transform: rotate3d(1, 0, 0, 50deg);
}
```

Если вызвать эту же примесь без аргументов, то будет выводиться результат по умолчанию, то есть нули:

```
.test-js {  
  transform: rotate3d(0, 0, 0, 0deg);  
}
```

Описанные в этой главе примеры доступны под номерами 6.1.1 - 6.1.4 и работают только с компиляторами, написанными на JavaScript.

## Выводы и мысли

Да, Less умеет интерпретировать JavaScript-код, записанный в переменной или в значении свойства, но получаемая от этого польза слишком мала и не покрывает потраченных на это усилий. Используя JavaScript в Less вы загрязняете его и усложняете для восприятия.