

Fork me on GitHub

The dynamic stylesheet language.

LESS extends CSS with dynamic behavior such as **variables**, **mixins**, **operations** and **functions**. LESS runs on both the **client-side** (Chrome, Safari, Firefox) and server-side, with **Node.js** and **Rhino**.



Download less.js

version 1.3.0

Write some LESS:

```
@base: #f938ab;

.box-shadow(@style, @c) when (iscolor(@c)) {
  box-shadow: @style @c;
  -webkit-box-shadow: @style @c;
  -moz-box-shadow: @style @c;
}

.box-shadow(@style, @alpha: 50%) when (isnumber(@alpha)) {
  .box-shadow(@style, rgba(0, 0, 0, @alpha));
}

.box {
  color: saturate(@base, 5%);
  border-color: lighten(@base, 30%);
  div { .box-shadow(0 0 5px, 30%) }
}
```

Include less.js with your styles:

```
<link rel="stylesheet/less" type="text/css"
href="styles.less">
<script src="less.js" type="text/javascript"></script>
```

overview

usage

language

source

about

Follow

Variables

Variables allow you to specify widely used values in a single place, and then re-use them throughout the style sheet, making global changes as easy as changing one line of code.

```
// LESS

@color: #4D926F;

#header {
  color: @color;
}

h2 {
  color: @color;
}
```

```
/* Compiled CSS */

#header {
  color: #4D926F;
}

h2 {
  color: #4D926F;
}
```

Mixins

Mixins allow you to embed all the properties of a class into another class by simply including the class name as one of its properties. It's just like variables, but for whole classes. Mixins can also behave like functions, and take arguments, as seen in the example bellow.

```
// LESS

.rounded-corners (@radius: 5px) {
  border-radius: @radius;
  -webkit-border-radius: @radius;
  -moz-border-radius: @radius;
}
```

```
/* Compiled CSS */

#header {
  border-radius: 5px;
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
}
```

```

}

#header {
  .rounded-corners;
}
#footer {
  .rounded-corners(10px);
}

}
#footer {
  border-radius: 10px;
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
}

```

Nested Rules

Rather than constructing long selector names to specify inheritance, in Less you can simply nest selectors inside other selectors. This makes inheritance clear and style sheets shorter.

```

// LESS

#header {
  h1 {
    font-size: 26px;
    font-weight: bold;
  }
  p { font-size: 12px;
    a { text-decoration: none;
      &:hover { border-width: 1px }
    }
  }
}

/* Compiled CSS */

#header h1 {
  font-size: 26px;
  font-weight: bold;
}
#header p {
  font-size: 12px;
}
#header p a {
  text-decoration: none;
}
#header p a:hover {
  border-width: 1px;
}

```

Functions & Operations

Are some elements in your style sheet proportional to other elements? Operations let you add, subtract, divide and multiply property values and colors, giving you the power to create complex relationships between properties. Functions map one-to-one with JavaScript code, allowing you to manipulate values however you want.

```

// LESS

@the-border: 1px;
@base-color: #111;
@red:       #842210;

#header {
  color: @base-color * 3;
  border-left: @the-border;
  border-right: @the-border * 2;
}
#footer {
  color: @base-color + #003300;
  border-color: desaturate(@red, 10%);
}

/* Compiled CSS */

#header {
  color: #333;
  border-left: 1px;
  border-right: 2px;
}
#footer {
  color: #114411;
  border-color: #7d2717;
}

```

```
}
```

Client-side usage

Link your `.less` stylesheets with the `rel` set to “stylesheet/less”:

```
<link rel="stylesheet/less" type="text/css" href="styles.less">
```

Then download `less.js` from the top of the page, and include it in the `<head>` element of your page, like so:

```
<script src="less.js" type="text/javascript"></script>
```

Make sure you include your stylesheets *before* the script.

Watch mode

Watch mode is a client-side feature which enables your styles to refresh automatically as they are changed.

To enable it, append ‘`#!watch`’ to the browser URL, then refresh the page. Alternatively, you can run `less.watch()` from the console.

Server-side usage

Installation

The easiest way to install LESS on the server, is via `npm`, the node package manager, as so:

```
$ npm install less
```

Use

Once installed, you can invoke the compiler from node, as such:

```
var less = require('less');

less.render('.class { width: 1 + 1 }', function (e, css) {
  console.log(css);
});
```

which will output

```
.class {  
  width: 2;  
}
```

you may also manually invoke the parser and compiler:

```
var parser = new(less.Parser);  
  
parser.parse('.class { width: 1 + 1 }', function (err, tree) {  
  if (err) { return console.error(err) }  
  console.log(tree.toCSS());  
});
```

Configuration

You may pass some options to the compiler:

```
var parser = new(less.Parser)({  
  paths: ['.'], ['./lib'], // Specify search paths for @import directives  
  filename: 'style.less' // Specify a filename, for better error messages  
});  
  
parser.parse('.class { width: 1 + 1 }', function (e, tree) {  
  tree.toCSS({ compress: true }); // Minify CSS output  
});
```

Command-line usage

Less comes with a binary, which lets you invoke the compiler from the command-line, as such:

```
$ lessc styles.less
```

This will output the compiled CSS to `stdout`, you may then redirect it to a file of your choice:

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

To output minified CSS, simply pass the `-x` option. If you would like more involved minification, the [YUI CSS Compressor](#) is also available with the `--yui-compress` option.

The Language

As an extension to CSS, LESS is not only backwards compatible with CSS, but the extra features it adds use *existing* CSS syntax. This makes learning LESS a *breeze*, and if in doubt, lets you fall back to CSS.

Variables

These are pretty self-explanatory:

```
@nice-blue: #5B83AD;  
@light-blue: @nice-blue + #111;  
  
#header { color: @light-blue; }
```

Outputs:

```
#header { color: #6c94be; }
```

It is also possible to define variables with a variable name:

```
@fnord: "I am fnord.";  
@var: 'fnord';  
content: @@var;
```

Which compiles to:

```
content: "I am fnord.";
```

Note that variables in LESS are actually ‘constants’ in that they can only be defined once.

Mixins

In LESS, it is possible to include a bunch of properties from one ruleset into another ruleset. So say we have the following class:

```
.bordered {  
  border-top: dotted 1px black;
```

```
border-bottom: solid 2px black;
}
```

And we want to use these properties inside other rulesets. Well, we just have to drop in the name of the class in any ruleset we want to include its properties, like so:

```
#menu a {
  color: #111;
  .bordered;
}
.post a {
  color: red;
  .bordered;
}
```

The properties of the `.bordered` class will now appear in both `#menu a` and `.post a`:

```
#menu a {
  color: #111;
  border-top: dotted 1px black;
  border-bottom: solid 2px black;
}
.post a {
  color: red;
  border-top: dotted 1px black;
  border-bottom: solid 2px black;
}
```

Any CSS *class* or *id* ruleset can be mixed-in that way.

Parametric Mixins

LESS has a special type of ruleset which can be mixed in like classes, but accepts parameters. Here's the canonical example:

```
.border-radius (@radius) {
  border-radius: @radius;
  -moz-border-radius: @radius;
  -webkit-border-radius: @radius;
}
```

And here's how we can mix it into various rulesets:

```
#header {
```

```
.border-radius (4px);
}
.button {
  .border-radius (6px);
}
```

Parametric mixins can also have default values for their parameters:

```
.border-radius (@radius: 5px) {
  border-radius: @radius;
  -moz-border-radius: @radius;
  -webkit-border-radius: @radius;
}
```

We can invoke it like this now:

```
#header {
  .border-radius;
}
```

And it will include a 5px border-radius.

You can also use parametric mixins which don't take parameters. This is useful if you want to hide the ruleset from the CSS output, but want to include its properties in other rulesets:

```
.wrap () {
  text-wrap: wrap;
  white-space: pre-wrap;
  white-space: -moz-pre-wrap;
  word-wrap: break-word;
}

pre { .wrap }
```

Which would output:

```
pre {
  text-wrap: wrap;
  white-space: pre-wrap;
  white-space: -moz-pre-wrap;
  word-wrap: break-word;
}
```

The @arguments variable

`@arguments` has a special meaning inside mixins, it contains all the arguments passed, when the mixin was called. This is useful if you don't want to deal with individual parameters:

```
.box-shadow (@x: 0, @y: 0, @blur: 1px, @color: #000) {
  box-shadow: @arguments;
  -moz-box-shadow: @arguments;
  -webkit-box-shadow: @arguments;
}
.box-shadow(2px, 5px);
```

Which results in:

```
box-shadow: 2px 5px 1px #000;
-moz-box-shadow: 2px 5px 1px #000;
-webkit-box-shadow: 2px 5px 1px #000;
```

Pattern-matching and Guard expressions

Sometimes, you may want to change the behaviour of a mixin, based on the parameters you pass to it. Let's start with something basic:

```
.mixin (@s, @color) { ... }

.class {
  .mixin(@switch, #888);
}
```

Now let's say we want `.mixin` to behave differently, based on the value of `@switch`, we could define `.mixin` as such:

```
.mixin (dark, @color) {
  color: darken(@color, 10%);
}
.mixin (light, @color) {
  color: lighten(@color, 10%);
}
.mixin (@_, @color) {
  display: block;
}
```

Now, if we run:

```
@switch: light;
```

```
.class {
  .mixin(@switch, #888);
}
```

We will get the following CSS:

```
.class {
  color: #a2a2a2;
  display: block;
}
```

Where the color passed to `.mixin` was lightened. If the value of `@switch` was dark, the result would be a darker color.

Here's what happened:

- The first mixin definition didn't match because it expected dark as the first argument.
- The second mixin definition matched, because it expected light.
- The third mixin definition matched because it expected any value.

Only mixin definitions which matched were used. Variables match and bind to any value. Anything other than a variable matches only with a value equal to itself.

We can also match on arity, here's an example:

```
.mixin (@a) {
  color: @a;
}
.mixin (@a, @b) {
  color: fade(@a, @b);
}
```

Now if we call `.mixin` with a single argument, we will get the output of the first definition, but if we call it with *two* arguments, we will get the second definition, namely `@a` faded to `@b`.

Guards

Guards are useful when you want to match on *expressions*, as opposed to simple values or arity. If you are familiar with functional programming, you have probably encountered them already.

In trying to stay as close as possible to the declarative nature of CSS, LESS has opted to implement conditional execution via **guarded mixins** instead of `if/else` statements, in the vein of `@media` query feature specifications.

Let's start with an example:

```
.mixin (@a) when (lightness(@a) >= 50%) {
```

```

    background-color: black;
}
.mixin (@a) when (lightness(@a) < 50%) {
    background-color: white;
}
.mixin (@a) {
    color: @a;
}

```

The key is the **when** keyword, which introduces a guard sequence (here with only one guard). Now if we run the following code:

```

.class1 { .mixin(#ddd) }
.class2 { .mixin(#555) }

```

Here's what we'll get:

```

.class1 {
    background-color: black;
    color: #ddd;
}
.class2 {
    background-color: white;
    color: #555;
}

```

The full list of comparison operators usable in guards are: `>` `>=` `=` `=<` `<`. Additionally, the keyword `true` is the only truthy value, making these two mixins equivalent:

```

.truth (@a) when (@a) { ... }
.truth (@a) when (@a = true) { ... }

```

Any value other than the keyword `true` is falsy:

```

.class {
    .truth(40); // Will not match any of the above definitions.
}

```

Guards can be separated with a comma `,`—if any of the guards evaluates to true, it's considered as a match:

```

.mixin (@a) when (@a > 10), (@a < -10) { ... }

```

Note that you can also compare arguments with each other, or with non-arguments:

```
@media: mobile;

.mixin (@a) when (@media = mobile) { ... }
.mixin (@a) when (@media = desktop) { ... }

.max (@a, @b) when (@a > @b) { width: @a }
.max (@a, @b) when (@a < @b) { width: @b }
```

Lastly, if you want to match mixins based on value type, you can use the *is** functions:

```
.mixin (@a, @b: 0) when (isnumber(@b)) { ... }
.mixin (@a, @b: black) when (iscolor(@b)) { ... }
```

Here are the basic type checking functions:

- iscolor
- isnumber
- isstring
- iskeyword
- isurl

If you want to check if a value, in addition to being a number, is in a specific unit, you may use one of:

- ispixel
- ispercentage
- isem

Last but not least, you may use the **and** keyword to provide additional conditions inside a guard:

```
.mixin (@a) when (isnumber(@a)) and (@a > 0) { ... }
```

And the **not** keyword to negate conditions:

```
.mixin (@b) when not (@b > 0) { ... }
```

Nested rules

LESS gives you the ability to use *nesting* instead of, or in combination with cascading. Lets say we have the following CSS:

```
#header { color: black; }
#header .navigation {
  font-size: 12px;
}
#header .logo {
```

```

width: 300px;
}
#header .logo:hover {
  text-decoration: none;
}

```

In LESS, we can also write it this way:

```

#header {
  color: black;

  .navigation {
    font-size: 12px;
  }
  .logo {
    width: 300px;
    &:hover { text-decoration: none }
  }
}

```

Or this way:

```

#header      { color: black;
  .navigation { font-size: 12px }
  .logo      { width: 300px;
    &:hover   { text-decoration: none }
  }
}

```

The resulting code is more concise, and mimics the structure of your DOM tree.

Notice the `&` combinator—it's used when you want a nested selector to be concatenated to its parent selector, instead of acting as a descendant. This is especially important for pseudo-classes like `:hover` and `:focus`.

For example:

```

.bordered {
  &.float {
    float: left;
  }
  .top {
    margin: 5px;
  }
}

```

Will output

```
.bordered.float {
  float: left;
}
.bordered .top {
  margin: 5px;
}
```

Operations

Any number, color or variable can be operated on. Here are a couple of examples:

```
@base: 5%;
@filler: @base * 2;
@other: @base + @filler;

color: #888 / 4;
background-color: @base-color + #111;
height: 100% / 2 + @filler;
```

The output is pretty much what you expect—LESS understands the difference between colors and units. If a unit is used in an operation, like in:

```
@var: 1px + 5;
```

LESS will use that unit for the final output—6px in this case.

Brackets are also authorized in operations:

```
width: (@var + 5) * 2;
```

And are required in compound values:

```
border: (@width * 2) solid black;
```

Color functions

LESS provides a variety of functions which transform colors. Colors are first converted to the *HSL* color-space, and then manipulated at the channel level:

```

lighten(@color, 10%);    // return a color which is 10% *lighter* than @color
darken(@color, 10%);    // return a color which is 10% *darker* than @color

saturate(@color, 10%);  // return a color 10% *more* saturated than @color
desaturate(@color, 10%); // return a color 10% *less* saturated than @color

fadein(@color, 10%);    // return a color 10% *less* transparent than @color
fadeout(@color, 10%);   // return a color 10% *more* transparent than @color
fade(@color, 50%);      // return @color with 50% transparency

spin(@color, 10);        // return a color with a 10 degree larger in hue than @color
spin(@color, -10);       // return a color with a 10 degree smaller hue than @color

mix(@color1, @color2);   // return a mix of @color1 and @color2

```

Using them is pretty straightforward:

```

@base: #f04615;

.class {
  color: saturate(@base, 5%);
  background-color: lighten(spin(@base, 8), 25%);
}

```

You can also extract color information:

```

hue(@color);           // returns the `hue` channel of @color
saturation(@color);    // returns the `saturation` channel of @color
lightness(@color);     // returns the `lightness` channel of @color
alpha(@color);         // returns the `alpha` channel of @color

```

This is useful if you want to create a new color based on another color's channel, for example:

```

@new: hsl(hue(@old), 45%, 90%);

```

@new will have @old's *hue*, and its own saturation and lightness.

Math functions

LESS provides a couple of handy math functions you can use on number values:

```

round(1.67); // returns `2`
ceil(2.4);   // returns `3`
floor(2.6);  // returns `2`

```

If you need to turn a value into a percentage, you can do so with the percentage function:

```
percentage(0.5); // returns `50%`
```

Namespaces

Sometimes, you may want to group your variables or mixins, for organizational purposes, or just to offer some encapsulation. You can do this pretty intuitively in LESS—say you want to bundle some mixins and variables under `#bundle`, for later re-use, or for distributing:

```
#bundle {
  .button () {
    display: block;
    border: 1px solid black;
    background-color: grey;
    &:hover { background-color: white }
  }
  .tab { ... }
  .citation { ... }
}
```

Now if we want to mixin the `.button` class in our `#header a`, we can do:

```
#header a {
  color: orange;
  #bundle > .button;
}
```

Scope

Scope in LESS is very similar to that of programming languages. Variables and mixins are first looked up locally, and if they aren't found, the compiler will look in the parent scope, and so on.

```
@var: red;

#page {
  @var: white;
  #header {
    color: @var; // white
  }
}
```

```
#footer {
  color: @var; // red
}
```

Comments

CSS-style comments are preserved by LESS:

```
/* Hello, I'm a CSS-style comment */
.class { color: black }
```

Single-line comments are also valid in LESS, but they are ‘silent’, they don’t show up in the compiled CSS output:

```
// Hi, I'm a silent comment, I won't show up in your CSS
.class { color: white }
```

Importing

You can import `.less` files, and all the variables and mixins in them will be made available to the main file. The `.less` extension is optional, so both of these are valid:

```
@import "lib.less";
@import "lib";
```

If you want to import a CSS file, and don’t want LESS to process it, just use the `.css` extension:

```
@import "lib.css";
```

The directive will just be left as is, and end up in the CSS output.

String interpolation

Variables can be embedded inside strings in a similar way to ruby or PHP, with the `@{name}` construct:

```
@base-url: "http://assets.fnord.com";
background-image: url("@{base-url}/images/bg.png");
```

Escaping

Sometimes you might need to output a CSS value which is either not valid CSS syntax, or uses proprietary syntax which LESS doesn't recognize.

To output such value, we place it inside a string prefixed with `~`, for example:

```
.class {
  filter: ~"ms:alwaysHasItsOwnSyntax.For.Stuff()";
}
```

This is called an “escaped value”, which will result in:

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

JavaScript evaluation

JavaScript expressions can be evaluated as values inside .less files. This is done by wrapping the expression with back-ticks:

```
@var: `"hello".toUpperCase() + '!'`;
```

Becomes:

```
@var: "HELLO!";
```

Note that you may also use interpolation and escaping as with strings:

```
@str: "hello";
@var: ~`"@{str}".toUpperCase() + '!'`;
```

Becomes:

```
@var: HELLO!;
```

It is also possible to access the JavaScript environment:

```
@height: `document.body.clientHeight`;
```

If you want to parse a JavaScript string as a hex color, you may use the `color` function:

```
@color: color(`window.colors.baseColor`);  
@darkcolor: darken(@color, 10%);
```

In Other Languages

Russian: <http://lesscss.ru>

Chinese: <http://lesscss.net>

Japanese: <http://less-ja.studiomohawk.com/>

Belorussian: <http://www.designcontest.com/show/lesscss-be>

About

LESS was developed by [Alexis Sellier](#), more commonly known as [cloudhead](#).

powered by [LESS](#)

Copyright © Alexis Sellier 2010-2012