


Language Features

Features of the Less language

less v2.7.1 has been released - **See what's new** (<https://github.com/less/less.js/blob/master/CHANGELOG.md>)

Overview

Edit the markdown source for "features-overview"  (<https://github.com/less/less-docs/blob/master/content/features-overview.md>)

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 vanilla CSS.

 (<https://github.com/less/less-docs/blob/master/content/features-overview.md>)

Variables

These are pretty self-explanatory:

```
@nice-blue: #5B83AD;
@light-blue: @nice-blue + #111;

#header {
  color: @light-blue;
}
```

Outputs:

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

Note that variables are actually "constants" in that they can only be defined once.

Mixins

Mixins are a way of including ("mixing in") a bunch of properties from one rule-set into another rule-set. 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 rule-sets. Well, we just have to drop in the name of the class where we want the 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`. (Note that you can also use `#ids` as mixins.)

Learn more

- More about mixins
- Parametric Mixins

Nested Rules

Less gives you the ability to use nesting instead of, or in combination with cascading. Let's say we have the following CSS:

```
#header {
  color: black;
}
#header .navigation {
  font-size: 12px;
}
#header .logo {
  width: 300px;
}
```

In Less, we can also write it this way:

```
#header {
  color: black;
  .navigation {
    font-size: 12px;
  }
  .logo {
    width: 300px;
  }
}
```

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

You can also bundle pseudo-selectors with your mixins using this method. Here's the classic clearfix hack, rewritten as a mixin (& represents the current selector parent):

```
.clearfix {
  display: block;
  zoom: 1;

  &:after {
    content: " ";
    display: block;
    font-size: 0;
    height: 0;
    clear: both;
    visibility: hidden;
  }
}
```

See also

- Parent Selectors

Nested Directives and Bubbling

Directives such as `media` or `keyframe` can be nested in the same way as selectors. Directive is placed on top and relative order against other elements inside the same ruleset remains unchanged. This is called bubbling.

Conditional directives e.g. `@Media` , `@supports` and `@document` have also selectors copied into their bodies:

```
.screen-color {
  @media screen {
    color: green;
    @media (min-width: 768px) {
      color: red;
    }
  }
  @media tv {
    color: black;
  }
}
```

outputs:

```
@media screen {
  .screen-color {
    color: green;
  }
}
@media screen and (min-width: 768px) {
  .screen-color {
    color: red;
  }
}
@media tv {
  .screen-color {
    color: black;
  }
}
```

Remaining non-conditional directives, for example `font-face` or `keyframes` , are bubbled up too. Their bodies do not change:

```
#a {
  color: blue;
  @font-face {
    src: made-up-url;
  }
  padding: 2 2 2 2;
}
```

outputs:

```
#a {
  color: blue;
}
@font-face {
  src: made-up-url;
}
#a {
  padding: 2 2 2 2;
}
```

Operations

Arithmetical operations `+`, `-`, `*`, `/` can operate on any number, color or variable. If it is possible, mathematical operations take units into account and convert numbers before adding, subtracting or comparing them. The result has leftmost explicitly stated unit type. If the conversion is impossible or not meaningful, units are ignored. Example of impossible conversion: px to cm or rad to %.

```
// numbers are converted into the same units
@conversion-1: 5cm + 10mm; // result is 6cm
@conversion-2: 2 - 3cm - 5mm; // result is -1.5cm

// conversion is impossible
@incompatible-units: 2 + 5px - 3cm; // result is 4px

// example with variables
@base: 5%;
@filler: @base * 2; // result is 10%
@other: @base + @filler; // result is 15%
```

Multiplication and division do not convert numbers. It would not be meaningful in most cases - a length multiplied by a length gives an area and css does not support specifying areas. Less nor operate on numbers as they are and assign explicitly stated unit type to the result.

```
@base: 2cm * 3mm; // result is 6cm
```

Colors are split into their red, green, blue and alpha dimensions. The operation is applied to each color dimension separately. E.g., if the user added two colors, then the green dimension of the result is equal to sum of green dimensions of input colors. If the user multiplied a color by a number, each color dimension will get multiplied.

Note: arithmetic operation on alpha is not defined, because math operation on colors do not have standard agreed upon meaning. Do not rely on current implementation as it may change (<https://github.com/less/less.js/issues/2694>) in later versions.

An operation on colors always produces valid color. If some color dimension of the result ends up being bigger than `ff` or smaller than `00`, the dimension is rounded to either `ff` or `00`. If alpha ends up being bigger than `1.0` or smaller than `0.0`, the alpha is rounded to either `1.0` or `0.0`.

```
@color: #224488 / 2; //results in #112244
background-color: #112244 + #111; // result is #223355
```

Escaping

Escaping allows you to use any arbitrary string as property or variable value. Anything inside `~"anything"` or `~'anything'` is used as is with no changes except interpolation.

```
.weird-element {
  content: ~"^/* some horrible but needed css hack";
}
```

results in:

```
.weird-element {
  content: ^/* some horrible but needed css hack;
}
```

Functions

Less provides a variety of functions which transform colors, manipulate strings and do maths. They are documented fully in the function reference.

Using them is pretty straightforward. The following example uses percentage to convert 0.5 to 50%, increases the saturation of a base color by 5% and then sets the background color to one that is lightened by 25% and spun by 8 degrees:

```
@base: #f04615;
@width: 0.5;

.class {
  width: percentage(@width); // returns `50%`
  color: saturate(@base, 5%);
  background-color: spin(lighten(@base, 25%), 8);
}
```

Namespaces and Accessors

(Not to be confused with CSS `@namespace` (<http://www.w3.org/TR/css3-namespace/>) or namespace selectors (<http://www.w3.org/TR/css3-selectors/#typenmsp>)).

Sometimes, you may want to group your 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 reuse or 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;
}
```

Note that variables declared within a namespace will be scoped to that namespace only and will not be available outside of the scope via the same syntax that you would use to reference a mixin (`#Namespace > .mixin-name`). So, for example, you can't do the following: (`#Namespace > @this-will-not-work`).

Scope

Scope in Less is very similar to that of programming languages. Variables and mixins are first looked for 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
  }
}
```

Variables and mixins do not have to be declared before being used so the following Less code is identical to the previous example:

```
@var: red;

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

See also

- Lazy Loading

Comments

Both block-style and inline comments may be used:

```
/* One hell of a block
style comment! */
@var: red;

// Get in line!
@var: white;
```

Importing

Importing works pretty much as expected. You can import a `.less` file, and all the variables in it will be available. The extension is optionally specified for `.less` files.

```
@import "library"; // library.less
@import "typo.css";
```

Variables

Edit the markdown source for "variables" [🔗 \(https://github.com/less/less-docs/blob/master/content/features/variables.md\)](https://github.com/less/less-docs/blob/master/content/features/variables.md)

Control commonly used values in a single location.

[🔗 \(https://github.com/less/less-docs/blob/master/content/features/variables.md\)](https://github.com/less/less-docs/blob/master/content/features/variables.md)

Overview

It's not uncommon to see the same value repeated dozens *if not hundreds of times* across your stylesheets:

```
a,
.link {
  color: #428bca;
}
.widget {
  color: #fff;
  background: #428bca;
}
```

Variables make your code easier to maintain by giving you a way to control those values from a single location:

```
// Variables
@link-color:      #428bca; // sea blue
@link-color-hover: darken(@link-color, 10%);

// Usage
a,
.link {
  color: @link-color;
}
a:hover {
  color: @link-color-hover;
}
.widget {
  color: #fff;
  background: @link-color;
}
```

Variable Interpolation

The examples above focused on using variables to control *values in CSS rules*, but they can also be used in other places as well, such as selector names, property names, URLs and `@import` statements.

Selectors

Version: 1.4.0

```
// Variables
@my-selector: banner;

// Usage
.@{my-selector} {
  font-weight: bold;
  line-height: 40px;
  margin: 0 auto;
}
```

Compiles to:

```
.banner {
  font-weight: bold;
  line-height: 40px;
  margin: 0 auto;
}
```

URLs

```
// Variables
@images: "../img";

// Usage
body {
  color: #444;
  background: url("@{images}/white-sand.png");
}
```

Import Statements

Version: 1.4.0

Syntax: `@import "{themes}/tidal-wave.less";`

Note that before v2.0.0, only variables which have been declared in the root or current scope were considered and that only the current file and calling files were considered when looking for a variable.

Example:

```
// Variables
@themes: "../../src/themes";

// Usage
@import "{themes}/tidal-wave.less";
```

Properties

Version: 1.6.0

```
@property: color;

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

Compiles to:

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

Variable Names

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.";
```

Lazy Loading

Variables are lazy loaded and do not have to be declared before being used.

Valid Less snippet:

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

@var: @a;
@a: 9%;
```

this is valid Less too:

```
.lazy-eval-scope {
  width: @var;
  @a: 9%;
}

@var: @a;
@a: 100%;
```

both compile into:

```
.lazy-eval-scope {
  width: 9%;
}
```

When defining a variable twice, the last definition of the variable is used, searching from the current scope upwards. This is similar to css itself where the last property inside a definition is used to determine the value.

For instance:

```
@var: 0;
.class {
  @var: 1;
  .brass {
    @var: 2;
    three: @var;
    @var: 3;
  }
  one: @var;
}
```

Compiles to:

```
.class {
  one: 1;
}
.class .brass {
  three: 3;
}
```

Default Variables

We sometimes get requests for default variables - an ability to set a variable only if it is not already set. This feature is not required because you can easily override a variable by putting the definition afterwards.

For instance:

```
// library
@base-color: green;
@dark-color: darken(@base-color, 10%);

// use of library
@import "library.less";
@base-color: red;
```

This works fine because of Lazy Loading - base-color is overridden and dark-color is a dark red.

Extend

Edit the markdown source for "extend"  (<https://github.com/less/less-docs/blob/master/content/features/extend.md>)

Extend is a Less pseudo-class which merges the selector it is put on with ones that match what it references.

 (<https://github.com/less/less-docs/blob/master/content/features/extend.md>)

Released v1.4.0 (CHANGELOG.md)

```
nav ul {
  &:extend(.inline);
  background: blue;
}
```

In the rule set above, the `:extend` selector will apply the "extending selector" (`nav ul`) onto the `.inline` class *wherever the `.inline` class appears*. The declaration block will be kept as-is, but without any reference to the extend (because extend isn't css).

So the following:

```
nav ul {
  &:extend(.inline);
  background: blue;
}
.inline {
  color: red;
}
```

Outputs

```
nav ul {
  background: blue;
}
.inline,
nav ul {
  color: red;
}
```

Notice how the `nav ul:extend(.inline)` selector gets output as `nav ul` - the extend gets removed before output and the selector block left as-is. If no properties are put in that block then it gets removed from the output (but the extend still may affect other selectors).

Extend Syntax

The extend is either attached to a selector or placed into a ruleset. It looks like a pseudo-class with selector parameter optionally followed by the keyword `all` :

Example:

```
.a:extend(.b) {}

// the above block does the same thing as the below block
.a {
  &:extend(.b);
}
```

```
.c:extend(.d all) {
  // extends all instances of ".d" e.g. ".x.d" or ".d.x"
}
.c:extend(.d) {
  // extends only instances where the selector will be output as just ".d"
}
```

It can contain one or more classes to extend, separated by commas.

Example:

```
.e:extend(.f) {}
.e:extend(.g) {}

// the above an the below do the same thing
.e:extend(.f, .g) {}
```

Extend Attached to Selector

Extend attached to a selector looks like an ordinary pseudo-class with selector as a parameter. A selector can contain multiple extend clauses, but all extends must be at the end of the selector.

- Extend after the selector: `pre:hover:extend(div pre)` .
- Space between selector and extend is allowed: `pre:hover :extend(div pre)` .
- Multiple extends are allowed: `pre:hover:extend(div pre):extend(.bucket tr)` - Note this is the same as `pre:hover:extend(div pre, .bucket tr)`
- This is NOT allowed: `pre:hover:extend(div pre).nth-child(odd)` . Extend must be last.

If a ruleset contains multiple selectors, any of them can have the extend keyword. Multiple selectors with extend in one ruleset:

```
.big-division,
.big-bag:extend(.bag),
.big-bucket:extend(.bucket) {
  // body
}
```

Extend Inside Ruleset

Extend can be placed into a ruleset's body using `&:extend(selector)` syntax. Placing extend into a body is a shortcut for placing it into every single selector of that ruleset.

Extend inside a body:

```
pre:hover,
.some-class {
  &:extend(div pre);
}
```

is exactly the same as adding an extend after each selector:


```
pre:hover:extend(div pre),
.some-class:extend(div pre) {}
```

Extending Nested Selectors

Extend is able to match nested selectors. Following less:

Example:

```
.bucket {
  tr { // nested ruleset with target selector
    color: blue;
  }
}
.some-class:extend(.bucket tr) {} // nested ruleset is recognized
```

Outputs

```
.bucket tr,
.some-class {
  color: blue;
}
```

Essentially the extend looks at the compiled css, not the original less.

Example:

```
.bucket {
  tr & { // nested ruleset with target selector
    color: blue;
  }
}
.some-class:extend(tr .bucket) {} // nested ruleset is recognized
```

Outputs

```
tr .bucket,
.some-class {
  color: blue;
}
```

Exact Matching with Extend

Extend by default looks for exact match between selectors. It does matter whether selector uses leading star or not. It does not matter that two nth-expressions have the same meaning, they need to have to same form in order to be matched. The only exception are quotes in attribute selector, less knows they have the same meaning and matches them.

Example:

```
.a.class,
.class.a,
.class > .a {
  color: blue;
}
.test:extend(.class) {} // this will NOT match the any selectors above
```

Leading star does matter. Selectors `*.class` and `.class` are equivalent, but extend will not match them:

```
*.class {
  color: blue;
}
.noStar:extend(.class) {} // this will NOT match the *.class selector
```

Outputs

```
*.class {
  color: blue;
}
```

Order of pseudo-classes does matter. Selectors `link:hover:visited` and `link:visited:hover` match the same set of elements, but extend treats them as different:

```
link:hover:visited {
  color: blue;
}
.selector:extend(link:visited:hover) {}
```

Outputs

```
link:hover:visited {
  color: blue;
}
```

Nth Expression

Nth expression form does matter. Nth-expressions `1n+3` and `n+3` are equivalent, but `extend` will not match them:

```
:nth-child(1n+3) {
  color: blue;
}
.child:extend(:nth-child(n+3)) {}
```

Outputs

```
:nth-child(1n+3) {
  color: blue;
}
```

Quote type in attribute selector does not matter. All of the following are equivalent.

```
[title=identifier] {
  color: blue;
}
[title='identifier'] {
  color: blue;
}
[title="identifier"] {
  color: blue;
}

.noQuote:extend([title=identifier]) {}
.singleQuote:extend([title='identifier']) {}
.doubleQuote:extend([title="identifier"]) {}
```

Outputs

```
[title=identifier],
.noQuote,
.singleQuote,
.doubleQuote {
  color: blue;
}

[title='identifier'],
.noQuote,
.singleQuote,
.doubleQuote {
  color: blue;
}

[title="identifier"],
.noQuote,
.singleQuote,
.doubleQuote {
  color: blue;
}
```

Extend "all"

When you specify the `all` keyword last in an `extend` argument it tells Less to match that selector as part of another selector. The selector will be copied and the matched part of the selector only will then be replaced with the `extend`, making a new selector.

Example:

```
.a.b.test,
.test.c {
  color: orange;
}
.test {
  &:hover {
    color: green;
  }
}

.replacement:extend(.test all) {}
```

Outputs

```
.a.b.test,
.test.c,
.a.b.replacement,
.replacement.c {
  color: orange;
}
.test:hover,
.replacement:hover {
  color: green;
}
```

You can think of this mode of operation as essentially doing a non-destructive search and replace.

Selector Interpolation with Extend

Extend is NOT able to match selectors with variables. If selector contains variable, extend will ignore it.

There is a pending feature request for this but it is not an easy change. However, extend can be attached to interpolated selector.

Selector with variable will not be matched:

```
@variable: .bucket;
@{variable} { // interpolated selector
  color: blue;
}
.some-class:extend(.bucket) {} // does nothing, no match is found
```

and extend with variable in target selector matches nothing:

```
.bucket {
  color: blue;
}
.some-class:extend(@{variable}) {} // interpolated selector matches nothing
@variable: .bucket;
```

Both of the previous examples compile into:

```
.bucket {
  color: blue;
}
```

However, :extend attached to an interpolated selector works:

```
.bucket {
  color: blue;
}
@{variable}:extend(.bucket) {}
@variable: .selector;
```

previous less compiles into:

```
.bucket, .selector {
  color: blue;
}
```

Scoping / Extend Inside @media

Extend written inside a media declaration should match only selectors inside the same media declaration:

```
@media print {
  .screenClass:extend(.selector) {} // extend inside media
  .selector { // this will be matched - it is in the same media
    color: black;
  }
}
.selector { // ruleset on top of style sheet - extend ignores it
  color: red;
}
@media screen {
  .selector { // ruleset inside another media - extend ignores it
    color: blue;
  }
}
```

compiles into:

```

@media print {
  .selector,
  .screenClass { /* ruleset inside the same media was extended */
    color: black;
  }
}
.selector { /* ruleset on top of style sheet was ignored */
  color: red;
}
@media screen {
  .selector { /* ruleset inside another media was ignored */
    color: blue;
  }
}

```

Extend written inside a media declaration does not match selectors inside nested declaration:

```

@media screen {
  .screenClass:extend(.selector) {} // extend inside media
  @media (min-width: 1023px) {
    .selector { // ruleset inside nested media - extend ignores it
      color: blue;
    }
  }
}

```

compiles into:

```

@media screen and (min-width: 1023px) {
  .selector { /* ruleset inside another nested media was ignored */
    color: blue;
  }
}

```

Top level extend matches everything including selectors inside nested media:

```

@media screen {
  .selector { /* ruleset inside nested media - top level extend works */
    color: blue;
  }
  @media (min-width: 1023px) {
    .selector { /* ruleset inside nested media - top level extend works */
      color: blue;
    }
  }
}

.topLevel:extend(.selector) {} /* top level extend matches everything */

```

compiles into:

```

@media screen {
  .selector,
  .topLevel { /* ruleset inside media was extended */
    color: blue;
  }
}
@media screen and (min-width: 1023px) {
  .selector,
  .topLevel { /* ruleset inside nested media was extended */
    color: blue;
  }
}

```

Duplication Detection

Currently there is no duplication detection.

Example:

```

.alert-info,
.widget {
  /* declarations */
}

.alert:extend(.alert-info, .widget) {}

```

Outputs

```
.alert-info,  
.widget,  
.alert,  
.alert {  
  /* declarations */  
}
```

Use Cases for Extend

Classic Use Case

The classic use case is to avoid adding a base class. For example, if you have

```
.animal {  
  background-color: black;  
  color: white;  
}
```

and you want to have a subtype of animal which overrides the background color then you have two options, firstly change your HTML

```
<a class="animal bear">Bear</a>
```

```
.animal {  
  background-color: black;  
  color: white;  
}  
.bear {  
  background-color: brown;  
}
```

or have simplified html and use extend in your less. e.g.

```
<a class="bear">Bear</a>
```

```
.animal {  
  background-color: black;  
  color: white;  
}  
.bear {  
  &:extend(.animal);  
  background-color: brown;  
}
```

Reducing CSS Size

Mixins copy all of the properties into a selector, which can lead to unnecessary duplication. Therefore you can use extends instead of mixins to move the selector up to the properties you wish to use, which leads to less css being generated.

Example - with mixin:

```
.my-inline-block() {  
  display: inline-block;  
  font-size: 0;  
}  
.thing1 {  
  .my-inline-block;  
}  
.thing2 {  
  .my-inline-block;  
}
```

Outputs

```
.thing1 {  
  display: inline-block;  
  font-size: 0;  
}  
.thing2 {  
  display: inline-block;  
  font-size: 0;  
}
```

Example (with extends):

```
.my-inline-block {
  display: inline-block;
  font-size: 0;
}
.thing1 {
  &:extend(.my-inline-block);
}
.thing2 {
  &:extend(.my-inline-block);
}
```

Outputs

```
.my-inline-block,
.thing1,
.thing2 {
  display: inline-block;
  font-size: 0;
}
```

Combining Styles / A More Advanced Mixin

Another use-case is as an alternative for a mixin - because mixins can only be used with simple selectors, if you have two different blocks of html, but need to apply the same styles to both you can use extends to relate two areas.

Example:

```
li.list > a {
  // list styles
}
button.list-style {
  &:extend(li.list > a); // use the same list styles
}
```

Mixins

Edit the markdown source for "mixins"  (<https://github.com/less/less-docs/blob/master/content/features/mixins.md>)

"mix-in" properties from existing styles

 <https://github.com/less/less-docs/blob/master/content/features/mixins.md>

You can mix-in class selectors and id selectors, e.g.

```
.a, #b {
  color: red;
}
.mixin-class {
  .a();
}
.mixin-id {
  #b();
}
```

which results in:

```
.a, #b {
  color: red;
}
.mixin-class {
  color: red;
}
.mixin-id {
  color: red;
}
```

Notice that when you call the mixin, the parentheses are optional.

```
// these two statements do the same thing:
.a();
.a;
```

Not Outputting the Mixin

If you want to create a mixin but you do not want that mixin to be output, you can put parentheses after it.

```
.my-mixin {
  color: black;
}
.my-other-mixin() {
  background: white;
}
.class {
  .my-mixin;
  .my-other-mixin;
}
```

outputs

```
.my-mixin {
  color: black;
}
.class {
  color: black;
  background: white;
}
```

Selectors in Mixins

Mixins can contain more than just properties, they can contain selectors too.

For example:

```
.my-hover-mixin() {
  &:hover {
    border: 1px solid red;
  }
}
button {
  .my-hover-mixin();
}
```

Outputs

```
button:hover {
  border: 1px solid red;
}
```

Namespaces

If you want to mixin properties inside a more complicated selector, you can stack up multiple id's or classes.

```
#outer {
  .inner {
    color: red;
  }
}

.c {
  #outer > .inner;
}
```

and again both `>` and whitespace are optional

```
// all do the same thing
#outer > .inner;
#outer > .inner();
#outer .inner;
#outer .inner();
#outer.inner;
#outer.inner();
```

One use of this is known as namespacing. You can put your mixins under a id selector and this makes sure it won't conflict with another library.

Example:

```
#my-library {
  .my-mixin() {
    color: black;
  }
}
// which can be used like this
.class {
  #my-library > .my-mixin();
}
```

Guarded Namespaces

If namespace have a guard, mixins defined by it are used only if guard condition returns true. Namespace guard is evaluated exactly the same way as guard on mixin, so next two mixins work the same way:

```
#namespace when (@mode=huge) {  
  .mixin() { /* */ }  
}  
  
#namespace {  
  .mixin() when (@mode=huge) { /* */ }  
}
```

The `default` function is assumed to have the same value for all nested namespaces and mixin. Following mixin is never evaluated, one of its guards is guaranteed to be false:

```
#sp_1 when (default()) {  
  #sp_2 when (default()) {  
    .mixin() when not(default()) { /* */ }  
  }  
}
```

The `!important` keyword

Use the `!important` keyword after mixin call to mark all properties inherited by it as `!important`:


Example:

```
.foo (@bg: #f5f5f5, @color: #900) {  
  background: @bg;  
  color: @color;  
}  
.unimportant {  
  .foo();  
}  
.important {  
  .foo() !important;  
}
```

Results in:

```
.unimportant {  
  background: #f5f5f5;  
  color: #900;  
}  
.important {  
  background: #f5f5f5 !important;  
  color: #900 !important;  
}
```

Parametric Mixins

Edit the markdown source for "mixins-parametric"  (<https://github.com/less/less-docs/blob/master/content/features/mixins-parametric.md>)

How to pass arguments to mixins

 (<https://github.com/less/less-docs/blob/master/content/features/mixins-parametric.md>)

Mixins can also take arguments, which are variables passed to the block of selectors when it is mixed in.

For example:

```
.border-radius(@radius) {  
  -webkit-border-radius: @radius;  
  -moz-border-radius: @radius;  
  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) {
  -webkit-border-radius: @radius;
  -moz-border-radius: @radius;
  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: -moz-pre-wrap;
  white-space: pre-wrap;
  word-wrap: break-word;
}

pre { .wrap }
```

Which would output:

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

Mixins with Multiple Parameters

Parameters are either *semicolon* or *comma* separated. It is recommended to use *semicolon*. The symbol comma has double meaning: it can be interpreted either as a mixin parameters separator or css list separator.

Using comma as mixin separator makes it impossible to create comma separated lists as an argument. On the other hand, if the compiler sees at least one semicolon inside mixin call or declaration, it assumes that arguments are separated by semicolons and all commas belong to css lists:

- two arguments and each contains comma separated list: `.name(1, 2, 3; something, else)`,
- three arguments and each contains one number: `.name(1, 2, 3)`,
- use dummy semicolon to create mixin call with one argument containing comma separated css list: `.name(1, 2, 3;)`,
- comma separated default value: `.name(@param1: red, blue;)`.

It is legal to define multiple mixins with the same name and number of parameters. Less will use properties of all that can apply. If you used the mixin with one parameter e.g. `.mixin(green);`, then properties of all mixins with exactly one mandatory parameter will be used:

```
.mixin(@color) {
  color-1: @color;
}
.mixin(@color; @padding: 2) {
  color-2: @color;
  padding-2: @padding;
}
.mixin(@color; @padding; @margin: 2) {
  color-3: @color;
  padding-3: @padding;
  margin: @margin @margin @margin @margin;
}
.some .selector div {
  .mixin(#008000);
}
```

compiles into:

```
.some .selector div {
  color-1: #008000;
  color-2: #008000;
  padding-2: 2;
}
```

Named Parameters

A mixin reference can supply parameters values by their names instead of just positions. Any parameter can be referenced by its name and they do not have to be in any special order:

```
.mixin(@color: black; @margin: 10px; @padding: 20px) {
  color: @color;
  margin: @margin;
  padding: @padding;
}
.class1 {
  .mixin(@margin: 20px; @color: #33acfe);
}
.class2 {
  .mixin(#efca44; @padding: 40px);
}
```

compiles into:

```
.class1 {
  color: #33acfe;
  margin: 20px;
  padding: 20px;
}
.class2 {
  color: #efca44;
  margin: 10px;
  padding: 40px;
}
```

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) {
  -webkit-box-shadow: @arguments;
  -moz-box-shadow: @arguments;
  box-shadow: @arguments;
}
.big-block {
  .box-shadow(2px; 5px);
}
```

Which results in:

```
.big-block {
  -webkit-box-shadow: 2px 5px 1px #000;
  -moz-box-shadow: 2px 5px 1px #000;
  box-shadow: 2px 5px 1px #000;
}
```

Advanced Arguments and the @rest Variable

You can use `...` if you want your mixin to take a variable number of arguments. Using this after a variable name will assign those arguments to the variable.

```
.mixin(...) {           // matches 0-N arguments
.mixin() {              // matches exactly 0 arguments
.mixin(@a: 1) {         // matches 0-1 arguments
.mixin(@a: 1; ...) {    // matches 0-N arguments
.mixin(@a; ...) {       // matches 1-N arguments
```

Furthermore:

```
.mixin(@a; @rest...) {
  // @rest is bound to arguments after @a
  // @arguments is bound to all arguments
}
```

Pattern-matching

Sometimes, you may want to change the behavior 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`.

Mixins as Functions

Edit the markdown source for "mixins-as-functions"  (<https://github.com/less/less-docs/blob/master/content/features/mixins-as-functions.md>)

Return variables or mixins from mixins

 <https://github.com/less/less-docs/blob/master/content/features/mixins-as-functions.md>

Variables and mixins defined in a mixin are visible and can be used in caller's scope. There is only one exception, a variable is not copied if the caller contains a variable with the same name (that includes variables defined by another mixin call). Only variables present in callers local scope are protected. Variables inherited from parent scopes are overridden.

Example:

```
.mixin() {
  @width: 100%;
  @height: 200px;
}

.caller {
  .mixin();
  width: @width;
  height: @height;
}
```

Results in:

```
.caller {
  width: 100%;
  height: 200px;
}
```

Thus variables defined in a mixin can act as its return values. This allows us to create a mixin that can be used almost like a function.

Example:

```
.average(@x, @y) {
  @average: ((@x + @y) / 2);
}

div {
  .average(16px, 50px); // "call" the mixin
  padding: @average;    // use its "return" value
}
```

Results in:

```
div {
  padding: 33px;
}
```

Variables defined directly in callers scope cannot be overridden. However, variables defined in callers parent scope is not protected and will be overridden:

```
.mixin() {
  @size: in-mixin;
  @definedOnlyInMixin: in-mixin;
}

.class {
  margin: @size @definedOnlyInMixin;
  .mixin();
}

@size: globally-defined-value; // callers parent scope - no protection
```

Results in:

```
.class {
  margin: in-mixin in-mixin;
}
```

Finally, mixin defined in mixin acts as return value too:

```
.unlock(@value) { // outer mixin
  .doSomething() { // nested mixin
    declaration: @value;
  }
}

#namespace {
  .unlock(5); // unlock doSomething mixin
  .doSomething(); //nested mixin was copied here and is usable
}
```

Results in:

```
#namespace {
  declaration: 5;
}
```

Passing Rulesets to Mixins

Edit the markdown source for "detached-rulesets" [🔗 \(https://github.com/less/less-docs/blob/master/content/features/detached-rulesets.md\)](https://github.com/less/less-docs/blob/master/content/features/detached-rulesets.md)

Allow wrapping of a css block, defined in a mixin

 <https://github.com/less/less-docs/blob/master/content/features/detached-rulesets.md>

Released v1.7.0 (CHANGELOG.md)

A detached ruleset is a group of css properties, nested rulesets, media declarations or anything else stored in a variable. You can include it into a ruleset or another structure and all its properties are going to be copied there. You can also use it as a mixin argument and pass it around as any other variable.

Simple example:

```
// declare detached ruleset
@detached-ruleset: { background: red; };

// use detached ruleset
.top {
  @detached-ruleset();
}
```

compiles into:

```
.top {  
  background: red;  
}
```

Parentheses after a detached ruleset call are mandatory. The call `@detached-ruleset;` would NOT work.

It is useful when you want to define a mixin that abstracts out either wrapping a piece of code in a media query or a non-supported browser class name. The rulesets can be passed to mixin so that the mixin can wrap the content, e.g.

```
.desktop-and-old-ie(@rules) {  
  @media screen and (min-width: 1200px) { @rules(); }  
  html.lt-ie9 & { @rules(); }  
}  
  
header {  
  background-color: blue;  
  
  .desktop-and-old-ie({  
    background-color: red;  
  });  
}
```

Here the `desktop-and-old-ie` mixin defines the media query and root class so that you can use a mixin to wrap a piece of code. This will output

```
header {  
  background-color: blue;  
}  
@media screen and (min-width: 1200px) {  
  header {  
    background-color: red;  
  }  
}  
html.lt-ie9 header {  
  background-color: red;  
}
```

A ruleset can be now assigned to a variable or passed in to a mixin and can contain the full set of less features, e.g.

```
@my-ruleset: {  
  .my-selector {  
    background-color: black;  
  }  
};
```

You can even take advantage of media query bubbling, for instance

```
@my-ruleset: {  
  .my-selector {  
    @media tv {  
      background-color: black;  
    }  
  }  
};  
@media (orientation:portrait) {  
  @my-ruleset();  
}
```

which will output

```
@media (orientation: portrait) and tv {  
  .my-selector {  
    background-color: black;  
  }  
}
```

A detached ruleset call unlocks (returns) all its mixins into caller the same way as mixin calls do. However, it does NOT return variables.

Returned mixin:

```
// detached ruleset with a mixin  
@detached-ruleset: {  
  .mixin() {  
    color:blue;  
  }  
};  
// call detached ruleset  
.caller {  
  @detached-ruleset();  
  .mixin();  
}
```

Results in:

```
.caller {
  color: blue;
}
```

Private variables:

```
@detached-ruleset: {
  @color:blue; // this variable is private
};
.caller {
  color: @color; // syntax error
}
```

Scoping

A detached ruleset can use all variables and mixins accessible where it is *defined* and where it is *called*. Otherwise said, both definition and caller scopes are available to it. If both scopes contains the same variable or mixin, declaration scope value takes precedence.

Declaration scope is the one where detached ruleset body is defined. Copying a detached ruleset from one variable into another cannot modify its scope. The ruleset does not gain access to new scopes just by being referenced there.

Lastly, a detached ruleset can gain access to scope by being unlocked (imported) into it.

Definition and Caller Scope Visibility

A detached ruleset sees the caller's variables and mixins:

```
@detached-ruleset: {
  caller-variable: @caller-variable; // variable is undefined here
  .caller-mixin(); // mixin is undefined here
};

selector {
  // use detached ruleset
  @detached-ruleset();

  // define variable and mixin needed inside the detached ruleset
  @caller-variable: value;
  .caller-mixin() {
    variable: declaration;
  }
}
```

compiles into:

```
selector {
  caller-variable: value;
  variable: declaration;
}
```

Variable and mixins accessible from definition win over those available in the caller:

```
@variable: global;
@detached-ruleset: {
  // will use global variable, because it is accessible
  // from detached-ruleset definition
  variable: @variable;
};

selector {
  @detached-ruleset();
  @variable: value; // variable defined in caller - will be ignored
}
```

compiles into:

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

Referencing *Won't* Modify Detached Ruleset Scope

A ruleset does not gain access to new scopes just by being referenced there:

```
@detached-1: { scope-detached: @one @two; };
.one {
  @one: visible;
  .two {
    @detached-2: @detached-1; // copying/renaming ruleset
    @two: visible; // ruleset can not see this variable
  }
}

.use-place {
  .one > .two();
  @detached-2();
}
```

throws an error:

```
ERROR 1:32 The variable "@one" was not declared.
```

Unlocking *Will* Modify Detached Ruleset Scope

A detached ruleset gains access by being unlocked (imported) inside a scope:

```
#space {
  .importer-1() {
    @detached: { scope-detached: @variable; }; // define detached ruleset
  }
}

.importer-2() {
  @variable: value; // unlocked detached ruleset CAN see this variable
  #space > .importer-1(); // unlock/import detached ruleset
}

.use-place {
  .importer-2(); // unlock/import detached ruleset second time
  @detached();
}
```


compiles into:

```
.use-place {
  scope-detached: value;
}
```

Import Directives

Edit the markdown source for "import-directives"  (<https://github.com/less/less-docs/blob/master/content/features/import-directives.md>)

Import styles from other style sheets

 <https://github.com/less/less-docs/blob/master/content/features/import-directives.md>

In standard CSS, `@import` at-rules must precede all other types of rules. But Less.js doesn't care where you put `@import` statements.

Example:

```
.foo {
  background: #900;
}
@import "this-is-valid.less";
```

File Extensions

`@import` statements may be treated differently by Less depending on the file extension:

- If the file has a `.css` extension it will be treated as CSS and the `@import` statement left as-is (see the inline option below).
- If it has *any other extension* it will be treated as Less and imported.
- If it does not have an extension, `.less` will be appended and it will be included as a imported Less file.

Examples:

```
@import "foo"; // foo.less is imported
@import "foo.less"; // foo.less is imported
@import "foo.php"; // foo.php imported as a less file
@import "foo.css"; // statement left in place, as-is
```

The following options can be used to override this behavior.

Import Options

Less offers several extensions to the CSS `@import` CSS at-rule to provide more flexibility over what you can do with external files.

Syntax: `@import (keyword) "filename";`

The following import directives have been implemented:

- `reference` : use a Less file but do not output it
- `inline` : include the source file in the output but do not process it
- `less` : treat the file as a Less file, no matter what the file extension
- `css` : treat the file as a CSS file, no matter what the file extension
- `once` : only include the file once (this is default behavior)
- `multiple` : include the file multiple times
- `optional` : continue compiling when file is not found

More than one keyword per `@import` is allowed, you will have to use commas to separate the keywords:

Example: `@import (optional, reference) "foo.less";`

reference

Use `@import (reference)` to import external files, but without adding the imported styles to the compiled output unless referenced.

Released v1.5.0 (CHANGELOG.md)

Example: `@import (reference) "foo.less";`

Imagine that `reference` marks every directive and selector with a *reference flag* in the imported file, imports as normal, but when the CSS is generated, "reference" selectors (as well as any media queries containing only reference selectors) are not output. `reference` styles will not show up in your generated CSS unless the reference styles are used as mixins or extended.

Additionally, **reference** produces different results depending on which method was used (mixin or extend):

- extend**: When a selector is extended, only the new selector is marked as *not referenced*, and it is pulled in at the position of the `reference @import` statement.
- mixins**: When a `reference` style is used as an implicit mixin, its rules are mixed-in, marked "not reference", and appear in the referenced place as normal.

reference example

This allows you to pull in only specific, targeted styles from a library such as Bootstrap (<https://github.com/twbs/bootstrap>) by doing something like this:

`.navbar:extend(.navbar all) {}`

And you will pull in only `.navbar` related styles from Bootstrap.

inline

Use `@import (inline)` to include external files, but not process them.

Released v1.5.0 (CHANGELOG.md)

Example: `@import (inline) "not-less-compatible.css";`

You will use this when a CSS file may not be Less compatible; this is because although Less supports most known standards CSS, it does not support comments in some places and does not support all known CSS hacks without modifying the CSS.

So you can use this to include the file in the output so that all CSS will be in one file.

less

Use `@import (less)` to treat imported files as Less, regardless of file extension.

Released v1.4.0 (CHANGELOG.md)

Example:

`@import (less) "foo.css";`

CSS

Use `@import (css)` to treat imported files as regular CSS, regardless of file extension. This means the import statement will be left as it is.

Released v1.4.0 (CHANGELOG.md)

Example:


```
@import (css) "foo.less";
```

outputs

```
@import "foo.less";
```

once

The default behavior of `@import` statements. It means the file is imported only once and subsequent import statements for that file will be ignored.

Released v1.4.0 (CHANGELOG.md)

This is the default behavior of `@import` statements.

Example:

```
@import (once) "foo.less";
@import (once) "foo.less"; // this statement will be ignored
```

multiple

Use `@import (multiple)` to allow importing of multiple files with the same name. This is the opposite behavior to `once`.

Released v1.4.0 (CHANGELOG.md)

Example:

```
// file: foo.less
.a {
  color: green;
}
// file: main.less
@import (multiple) "foo.less";
@import (multiple) "foo.less";
```

Outputs

```
.a {
  color: green;
}
.a {
  color: green;
}
```

optional

Use `@import (optional)` to allow importing of a file only when it exists. Without the `optional` keyword Less throws a `FileError` and stops compiling when importing a file that can not be found.

Released v2.3.0 (CHANGELOG.md)

Mixin Guards

Edit the markdown source for "mixin-guards"  (<https://github.com/less/less-docs/blob/master/content/features/mixin-guards.md>)

Conditional mixins

 <https://github.com/less/less-docs/blob/master/content/features/mixin-guards.md>

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;
}
```

Guard Comparison Operators

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.
}
```

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 }
```

Guard Logical Operators

You can use logical operators with guards. The syntax is based on CSS media queries.

Use the `and` keyword to combine guards:

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

You can emulate the *or* operator by separating guards with a comma `,`. If any of the guards evaluate to true, it's considered a match:

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

Use the `not` keyword to negate conditions:

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

Type Checking Functions

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 is in a specific unit in addition to being a number, you may use one of:

- ispixel
- ispercentage
- isem
- isunit

Conditional Mixins

(FIXME) Additionally, the `default` function may be used to make a mixin match depending on other mixing matches, and you may use it to create "conditional mixins" similar to `else` or `default` statements (of `if` and `case` structures respectively):

```
.mixin (@a) when (@a > 0) { ... }  
.mixin (@a) when (default()) { ... } // matches only if first mixin does not, i.e. when @a <= 0
```

CSS Guards

Edit the markdown source for "css-guards"  (<https://github.com/less/less-docs/blob/master/content/features/css-guards.md>)

"if"s around selectors

 <https://github.com/less/less-docs/blob/master/content/features/css-guards.md>

Released v1.5.0 (CHANGELOG.md)

Guards can also be applied to css selectors, which is syntactic sugar for declaring the mixin and then calling it immediately.

For instance, before 1.5.0 you would have had to do this:

```
.my-optional-style() when (@my-option = true) {  
  button {  
    color: white;  
  }  
}  
.my-optional-style();
```

Now, you can apply the guard directly to a style.

```
button when (@my-option = true) {  
  color: white;  
}
```

You can also achieve an `if` type statement by combining this with the `&` feature, allowing you to group multiple guards.

```
& when (@my-option = true) {  
  button {  
    color: white;  
  }  
  a {  
    color: blue;  
  }  
}
```

Loops

Edit the markdown source for "loops"  (<https://github.com/less/less-docs/blob/master/content/features/loops.md>)

Creating loops

 <https://github.com/less/less-docs/blob/master/content/features/loops.md>

In Less a mixin can call itself. Such recursive mixins, when combined with Guard Expressions and Pattern Matching, can be used to create various iterative/loop structures.

Example:

```
.loop(@counter) when (@counter > 0) {
  .loop((@counter - 1)); // next iteration
  width: (10px * @counter); // code for each iteration
}

div {
  .loop(5); // launch the loop
}
```

Output:

```
div {
  width: 10px;
  width: 20px;
  width: 30px;
  width: 40px;
  width: 50px;
}
```

A generic example of using a recursive loop to generate CSS grid classes:


```
.generate-columns(4);

.generate-columns(@n, @i: 1) when (@i <= @n) {
  .column-@{i} {
    width: (@i * 100% / @n);
  }
  .generate-columns(@n, (@i + 1));
}
```


Output:

```
.column-1 {
  width: 25%;
}
.column-2 {
  width: 50%;
}
.column-3 {
  width: 75%;
}
.column-4 {
  width: 100%;
}
```

Merge

Edit the markdown source for "merge"  (<https://github.com/less/less-docs/blob/master/content/features/merge.md>)

Combine properties

 (<https://github.com/less/less-docs/blob/master/content/features/merge.md>)

The `merge` feature allows for aggregating values from multiple properties into a comma or space separated list under a single property. `merge` is useful for properties such as `background` and `transform`.

Comma

Append property value with comma

Released v1.5.0 (CHANGELOG.md)

Example:

```
.mixin() {
  box-shadow+: inset 0 0 10px #555;
}

.myclass {
  .mixin();
  box-shadow+: 0 0 20px black;
}
```

Outputs

```
.myclass {
  box-shadow: inset 0 0 10px #555, 0 0 20px black;
}
```

Space

Append property value with space

Released v1.7.0 (CHANGELOG.md)

Example:

```
.mixin() {
  transform+_: scale(2);
}
.myclass {
  .mixin();
  transform+_: rotate(15deg);
}
```

Outputs


```
.myclass {
  transform: scale(2) rotate(15deg);
}
```

To avoid any unintentional joins, `merge` requires an explicit `+` or `+_` flag on each join pending declaration.

Parent Selectors

Edit the markdown source for "parent-selectors"  (<https://github.com/less/less-docs/blob/master/content/features/parent-selectors.md>)

Referencing parent selectors with `&`

 <https://github.com/less/less-docs/blob/master/content/features/parent-selectors.md>

The `&` operator represents the parent selectors of a nested rule and is most commonly used when applying a modifying class or pseudo-class to an existing selector:

```
a {
  color: blue;
  &:hover {
    color: green;
  }
}
```

results in:

```
a {
  color: blue;
}

a:hover {
  color: green;
}
```

Notice that without the `&`, the above example would result in `a :hover` rule (a descendant selector that matches hovered elements inside of `<a>` tags) and this is not what we typically would want with the nested `:hover`.

The "parent selectors" operator has a variety of uses. Basically any time you need the selectors of the nested rules to be combined in other ways than the default. For example another typical use of the `&` is to produce repetitive class names:

```
.button {
  &-ok {
    background-image: url("ok.png");
  }
  &-cancel {
    background-image: url("cancel.png");
  }

  &-custom {
    background-image: url("custom.png");
  }
}
```

output:

```
.button-ok {
  background-image: url("ok.png");
}
.button-cancel {
  background-image: url("cancel.png");
}
.button-custom {
  background-image: url("custom.png");
}
```

Multiple &

& may appear more than once within a selector. This makes it possible to repeatedly refer to a parent selector without repeating its name.

```
.link {
  & + & {
    color: red;
  }

  & & {
    color: green;
  }

  && {
    color: blue;
  }

  &, &ish {
    color: cyan;
  }
}
```

will output:

```
.link + .link {
  color: red;
}
.link .link {
  color: green;
}
.link.link {
  color: blue;
}
.link, .linkish {
  color: cyan;
}
```

Note that & represents all parent selectors (not just the nearest ancestor) so the following example:

```
.grand {
  .parent {
    & > & {
      color: red;
    }

    & & {
      color: green;
    }

    && {
      color: blue;
    }

    &, &ish {
      color: cyan;
    }
  }
}
```

results in:

```
.grand .parent > .grand .parent {
  color: red;
}
.grand .parent .grand .parent {
  color: green;
}
.grand .parent.grand .parent {
  color: blue;
}
.grand .parent,
.grand .parentish {
  color: cyan;
}
```

Changing Selector Order

It can be useful to prepend a selector to the inherited (parent) selectors. This can be done by putting the `&` after current selector. For example, when using Modernizr, you might want to specify different rules based on supported features:

```
.header {
  .menu {
    border-radius: 5px;
    .no-borderradius & {
      background-image: url('images/button-background.png');
    }
  }
}
```

The selector `.no-borderradius &` will prepend `.no-borderradius` to its parent `.header .menu` to form the `.no-borderradius .header .menu` on output:

```
.header .menu {
  border-radius: 5px;
}
.no-borderradius .header .menu {
  background-image: url('images/button-background.png');
}
```

Combinatorial Explosion

`&` can also be used to generate every possible permutation of selectors in a comma separated list:

```
p, a, ul, li {
  border-top: 2px dotted #366;
  & + & {
    border-top: 0;
  }
}
```

This expands to all possible (16) combinations of the specified elements:

```
p,
a,
ul,
li {
  border-top: 2px dotted #366;
}
p + p,
p + a,
p + ul,
p + li,
a + p,
a + a,
a + ul,
a + li,
ul + p,
ul + a,
ul + ul,
ul + li,
li + p,
li + a,
li + ul,
li + li {
  border-top: 0;
}
```



Less.js and these docs are maintained by the core less team ([../about/#team](https://github.com/less/less/blob/master/CHANGELOG.md)).

Documentation source code released under the MIT License (<https://github.com/less/less-docs/blob/master/LICENSE-MIT>), documentation under CC BY 3.0 (<http://creativecommons.org/licenses/by/3.0/>).

Currently v2.7.1 · [Less.js Issues \(https://github.com/less/less.js/issues\)](https://github.com/less/less.js/issues) · [Less Docs Issues \(https://github.com/less/less-docs/issues?&state=open\)](https://github.com/less/less-docs/issues?&state=open) · [Changelog \(https://github.com/less/less.js/blob/master/CHANGELOG.md\)](https://github.com/less/less.js/blob/master/CHANGELOG.md)

